Understanding Neural Networks: A Structured Guide to Training and Optimization

Md Shahriar Forhad (https://github.com/Shahriar88)

February 15, 2026

Contents

List of Figures

List of Tables

Chapter 1

Activation Functions, Loss Functions, Optimizers, and Training Mechanics

1.1 Activation Functions

1.1.1 Sigmoid

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

Applications: Binary classification outputs, logistic regression, simple neural networks. **Advantages:**

- Smooth, differentiable.
- Maps input to $(0, 1)$ range for probability interpretation.

Disadvantages:

- Vanishing gradients for large $|x|$.
- Non-zero-centered outputs slow convergence.

1.1.2 Tanh

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Applications: Hidden layers of RNNs, MLPs. **Advantages:**

- Zero-centered outputs.
- Stronger gradients than Sigmoid.

Disadvantages:

- Still suffers from vanishing gradients.
- Costlier than ReLU.

1.1.3 ReLU

$$\text{ReLU}(x) = \max(0, x)$$

Applications: CNNs, fully connected layers in deep networks. **Advantages:**

- Simple and efficient.
- Sparse activations reduce computation.
- Mitigates vanishing gradient problem.

Disadvantages:

- "Dying ReLU" problem for negative inputs.
- Unbounded positive outputs.

1.1.4 Leaky ReLU

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

Applications: CNNs where ReLU causes dead neurons. **Advantages:** Avoids dying ReLU; small slope for negatives. **Disadvantages:** Requires α tuning.

1.1.5 Softmax

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Applications: Multiclass classification output layers. **Advantages:** Converts logits to probabilities. **Disadvantages:** Sensitive to large logits; saturation slows learning.

1.1.6 GELU

$$\text{GELU}(x) = x\Phi(x)$$

Applications: Transformers, BERT, Vision Transformers. **Advantages:** Smooth and probabilistic activation. **Disadvantages:** More computation-heavy.

1.2 Loss Functions

1.2.1 Mean Squared Error (MSE)

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Applications: Regression problems, autoencoders. **Advantages:** Penalizes large errors heavily; convex. **Disadvantages:** Sensitive to outliers.

1.2.2 Mean Absolute Error (MAE)

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

Applications: Robust regression. **Advantages:** Robust to outliers. **Disadvantages:** Non-differentiable at 0.

1.2.3 Huber Loss

$$L_\delta(e) = \begin{cases} \frac{1}{2}e^2, & |e| \le \delta \\ \delta(|e| - \frac{1}{2}\delta), & |e| > \delta \end{cases}$$

Applications: Regression with moderate outliers. **Advantages:** Combines smoothness (MSE) and robustness (MAE). **Disadvantages:** Requires tuning δ.

1.2.4 Cross Entropy Loss

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log p_{ik}$$

Applications: Classification (binary, multiclass, multi-label). **Advantages:** Probabilistic; aligns with maximum likelihood. **Disadvantages:** Sensitive to noisy labels; overconfident predictions.

1.2.5 Focal Loss

$$L = -\alpha(1 - p_t)^\gamma \log(p_t)$$

Applications: Object detection (Mask R-CNN, RetinaNet). **Advantages:** Handles class imbalance. **Disadvantages:** Hyperparameters α, γ need tuning.

1.2.6 Dice / IoU Loss

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|}, \qquad L = 1 - \text{Dice}$$

Applications: Image segmentation (Mask R-CNN, U-Net). **Advantages:** Works well with imbalanced masks. **Disadvantages:** Non-linear, harder optimization.

1.2.7 KL Divergence

$$D_{KL}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

Applications: Variational Autoencoders, distillation. **Advantages:** Measures information difference. **Disadvantages:** Asymmetric; can be unstable.

Loss Functions

Loss	**Typical Use**	**Advantages**	**Disadvantages**
MSE ($\frac{1}{N}\sum(y - \hat{y})^2$)	Regression	Convex, smooth, strong penalty on large errors	Sensitive to outliers
MAE ($\frac{1}{N}\sum\|y - \hat{y}\|$)	Robust regression	Robust to outliers	Non-differentiable at 0; slower convergence
Huber / Smooth-L_1	Regression, box regression	Combines MSE and MAE; robust and smooth	Requires tuning of δ
Binary CE	Binary classification	Probabilistic; aligns with MLE	Sensitive to noisy labels
Multiclass CE	Multiclass classification	Standard loss with softmax; probabilistic interpretation	Overconfident predictions are heavily penalized
BCEWithLogits	Multi-label classification	Numerically stable (sigmoid + CE combined)	Ignores label dependencies
Focal Loss	Imbalanced classification/detection	Focuses learning on hard examples	Requires tuning of α and γ
Dice Loss	Image segmentation (masks)	Handles class imbalance; measures overlap directly	Unstable for very small targets
IoU (Jaccard) Loss	Segmentation / bounding boxes	Directly optimizes IoU metric	Non-smooth; slower early convergence
Smooth-L_1 (boxes)	Object detection (R-CNNs)	Robust to outliers; standard for box regression	Scale-sensitive
KL Divergence	VAEs, knowledge distillation	Measures divergence between distributions	Asymmetric; unstable when $Q \approx 0$
Triplet Loss	Metric learning	Learns discriminative embedding spaces	Needs triplet mining; margin tuning
Contrastive Loss	Siamese networks / similarity tasks	Learns pairwise distance relationships	Requires balanced positive/negative pairs
Cosine Embedding	Text/vision embeddings	Scale-invariant and simple	Loses magnitude information
Perceptual Loss	Super-resolution / style transfer	Encourages perceptual similarity using deep features	Computationally heavy; needs pretrained ϕ
Total Variation	Image smoothing / denoising	Removes noise, encourages smoothness	Can over-smooth fine details

1.3 Optimizers

1.3.1 Stochastic Gradient Descent (SGD)

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

Applications: Classical ML, CNNs. **Advantages:** Simple, effective. **Disadvantages:** Sensitive to learning rate; oscillates.

1.3.2 Momentum

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta_t), \quad \theta_{t+1} = \theta_t - \eta v_t$$

Applications: Deep CNNs. **Advantages:** Faster convergence; less noise. **Disadvantages:** Needs tuning of β.

1.3.3 Adagrad

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_\theta J(\theta_t)$$

Applications: Sparse features (e.g. NLP embeddings). **Advantages:** Adaptive learning rate. **Disadvantages:** Learning rate decays too fast.

1.3.4 RMSProp

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2, \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Applications: RNNs, time series, non-stationary data. **Advantages:** Stable, adaptive. **Disadvantages:** Sensitive to β.

1.3.5 Adam

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Applications: Deep learning, NLP, Transformers. **Advantages:** Combines momentum + adaptive rate. **Disadvantages:** May generalize poorly.

1.3.6 AdamW

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right)$$

Applications: Transformers, BERT, ViTs. **Advantages:** Decoupled weight decay $\rightarrow$ better generalization. **Disadvantages:** Slightly more computation.

1.3.7 LAMB

$$r_t = \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \quad \theta_{t+1} = \theta_t - \eta \frac{\|\theta_t\|}{\|r_t\|} r_t$$

Applications: Large-batch Transformer training. **Advantages:** Enables distributed training. **Disadvantages:** Complex; more tuning.

1.4 Summary Tables

Activation Functions

Function	Pros	Cons
Sigmoid	Probabilistic, smooth	Vanishing gradient
Tanh	Zero-centered	Saturation at extremes
ReLU	Sparse, fast	Dead neurons
Leaky ReLU	Fixes dead ReLU	Hyperparam α
Softmax	Probabilities	Saturation
GELU	Smooth	Slower compute

Optimizers

Optimizer	Pros	Cons
SGD	Simple, reliable	Slow, oscillates
Momentum	Smooth convergence	Needs tuning
RMSProp	Stable	Sensitive β
Adam	Fast, adaptive	May overfit
AdamW	Best generalization	Slightly slower

1.5 Neuron and Feature Representation

Mathematical identification of a feature In a hidden layer, each coordinate of the activation vector corresponds to a learned feature. Mathematically,

$$\text{feature}_j \equiv a_j.$$

That is, the activation produced by neuron j represents the j-th feature.

What is a neuron? Strict definition (mathematically correct). A neuron (also called a *unit*) is the computational element that maps its parameters and activation function to an output:

$$(\mathbf{w}_j, b_j, \sigma) \longmapsto a_j.$$

The **neuron** itself is the **function** defined by its weights, bias, and nonlinearity; the scalar quantities z_j and a_j are intermediate values produced by this function.

Neuron and feature (functional form) A neuron (or unit) is a parametric nonlinear function defined by weights, bias, and an activation function. Given an input vector $\mathbf{x}$, it computes

$$z_j = \mathbf{w}_j^\top \mathbf{x} + b_j, \qquad a_j = \sigma(z_j).$$

The activation a_j is the neuron output and is interpreted as a learned feature.

Summary

$$\text{neuron} = (\mathbf{w}, b, \sigma), \qquad \text{pre-activation} = z, \qquad \text{activation} = a, \qquad \text{feature} = a.$$

Thus, the output of a neuron is the feature it represents.

1.6 Forward Pass

The forward pass computes activations layer by layer using learned weights and biases. Each layer applies linear transformations followed by nonlinear activation functions (e.g., sigmoid, ReLU, etc.).

Table 1.1: Forward Pass Computations and Data Structures

Step	**Calculation**	**Data Structure**	**Multiplication Type**
Forward Pass			
Input to 1st hidden layer (HI_1)	$HI_1 = W_1 \cdot X + B_1$	HI_1: Matrix, W_1: Matrix, X: Matrix, B_1: Vector	**Dot Product** (between W_1 and X)
Output of 1st hidden layer (HO_1)	$HO_1 = \sigma(HI_1)$	HO_1: Vector	**Element-wise** (Sigmoid applied element-wise)
Input to 2nd hidden layer (HI_2)	$HI_2 = W_2 \cdot HO_1 + B_2$	HI_2: Vector, W_2: Matrix, HO_1: Vector, B_2: Vector	**Dot Product** (between W_2 and HO_1)
Output of 2nd hidden layer (HO_2)	$HO_2 = \sigma(HI_2)$	HO_2: Vector	**Element-wise** (Sigmoid applied element-wise)
Input to output layer (HO_final)	$HO_{final} = W_3 \cdot HO_2 + B_3$	HO_{final}: Vector, W_3: Matrix, HO_2: Vector, B_3: Vector	**Dot Product** (between W_3 and HO_2)
Final output $\hat{Y}$	$\hat{Y} = \sigma(HO_{final})$	$\hat{Y}$: Vector	**Element-wise** (Sigmoid applied element-wise)
Error Calculation	$E = Y - \hat{Y}$	E: Vector, Y: Vector, $\hat{Y}$: Vector	**Element-wise** (Subtraction)

1.7 Backpropagation

$$Error\_Layer\_N = \sigma_d'(Input\_N)\Big(Error\_Layer\_N + 1(W\_N + 1)^T\Big), \qquad Input\_N = WX + B.$$

1.8 Weight and Bias Updates

$$W_{new} = W_{old} + lr \cdot Own\_Error \cdot Own\_Input^T$$

$$Own\_Input = X, \quad Own\_Input\_N = HO_{N-1} = \sigma(HI_{N-1})$$

Table 1.2: Comprehensive Backpropagation Steps

Step	**Calculation**	**Data Structure**	**Multiplication Type**
Backpropagation			
Error at output layer	$err_{HO_{final}} = E \cdot \sigma'(HO_{final})$	$err_{HO_{final}}$: Vector, E: Vector, $\sigma'(HO_{final})$: Vector	**Element-wise** (Multiplication)
Error in 2nd hidden layer (err_HO_2)	$err_{HO_2} = err_{HO_{final}} \cdot W_3^T \cdot \sigma'(HI_2)$	err_{HO_2}: Vector, W_3^T: Matrix (transpose), $\sigma'(HI_2)$: Vector	**Dot Product** (between $err_{HO_{final}}$ and W_3^T) followed by **Element-wise** multiplication with $\sigma'(HI_2)$
Error in 1st hidden layer (err_HO_1)	$err_{HO_1} = err_{HO_2} \cdot W_2^T \cdot \sigma'(HI_1)$	err_{HO_1}: Vector, W_2^T: Matrix (transpose), $\sigma'(HI_1)$: Vector	**Dot Product** (between err_{HO_2} and W_2^T) followed by **Element-wise** multiplication with $\sigma'(HI_1)$
Error Calculation	$E = Y - \hat{Y}$	E: Vector, Y: Vector, $\hat{Y}$: Vector	**Element-wise** (Subtraction)

Table 1.3: Weight and Bias Update Steps

Step	**Calculation**	**Data Structure**	**Multiplication Type**
Weight and Bias Updates			
Update W_1 and B_1	$W_1 = W_1 + lr \cdot X^T \cdot err_{HO_1}$, $B_1 = B_1 + lr \cdot err_{HO_1}$	W_1: Matrix, X^T: Matrix (transpose), err_{HO_1}: Vector	**Dot Product** (between X^T and err_{HO_1})
Update W_2 and B_2	$W_2 = W_2 + lr \cdot HO_1^T \cdot err_{HO_2}$, $B_2 = B_2 + lr \cdot err_{HO_2}$	W_2: Matrix, HO_1^T: Matrix (transpose), err_{HO_2}: Vector	**Dot Product** (between HO_1^T and err_{HO_2})
Update W_3 and B_3	$W_3 = W_3 + lr \cdot HO_2^T \cdot err_{HO_{final}}$, $B_3 = B_3 + lr \cdot err_{HO_{final}}$	W_3: Matrix, HO_2^T: Matrix (transpose), $err_{HO_{final}}$: Vector	**Dot Product** (between HO_2^T and $err_{HO_{final}}$)

Chapter 2

General Structure of Forward Pass and Backpropagation

2.1 Overview

The general structure of the forward pass and backpropagation is the same for all basic neural networks, regardless of how many hidden layers or neurons are used. The following principles hold true for most feedforward neural networks (also known as **multilayer perceptrons (MLPs)**).

2.2 Forward Pass

2.2.1 Basic Structure

- The network consists of an input layer, one or more hidden layers, and an output layer.
- Each layer computes a weighted sum of its inputs, adds a bias, and applies an activation function (such as sigmoid, ReLU, or tanh) to produce its output.

2.2.2 Computation Flow

- Data flows forward through the network from the input layer to the output layer.
- The output of one layer becomes the input to the next layer.

2.3 Backpropagation

2.3.1 Error Propagation

- After computing the final output during the forward pass, the network compares the predicted output with the expected (target) output.
- The error is calculated using a loss function (for example, mean squared error or cross-entropy).
- This error is propagated backward from the output layer to the hidden layers, adjusting the neuron weights to minimize the error.

2.3.2 Weight Update

- The gradients (rate of change of error with respect to weights) are computed using the chain rule of calculus.
- The weights and biases are updated using gradient descent or its variants (e.g., stochastic gradient descent).

2.3.3 General Steps

1. Compute the error at the output layer.
2. Backpropagate the error through each preceding layer.
3. Update the weights and biases to reduce the overall error.

2.4 Universality of the Process

The process of forward pass and backpropagation described above holds for all basic feedforward neural networks, regardless of:

- The number of hidden layers.
- The number of neurons per layer.
- The activation function used.

2.5 Common Variations

While the core steps remain the same, there are variations among different network types and techniques:

- **Activation Functions:** Modern networks often use ReLU or tanh instead of sigmoid in hidden layers.
- **Loss Functions:** Classification problems typically use cross-entropy loss, while regression problems often use mean squared error (MSE).
- **Learning Algorithms:** Standard gradient descent can be replaced by advanced optimizers such as Adam, RMSProp, or Adagrad.

2.6 Summary

- The concept of forward pass and backpropagation is **universal** to most feedforward neural networks.
- Differences arise mainly in the architecture (number of layers or neurons) and the type of activation, loss function, and optimization algorithm used.
- As networks become deeper and more complex, the same principles apply, but across more layers.

Examples

- **Deep Neural Networks (DNNs):** Contain more hidden layers, but the core forward and backward propagation remain identical.
- **Convolutional Neural Networks (CNNs)** and **Recurrent Neural Networks (RNNs):** Both employ forward and backward propagation with specialized modifications.

In essence, for any basic feedforward neural network, the process of forward pass and backpropagation remains fundamentally the same.

2.7 Worked Example I: Single-Neuron Forward and Backpropagation

Table 2.1: Forward propagation for a single-neuron network

Step	Quantity	Expression	Numerical value
1	Input	x	2
2	Weight	w	3
3	Bias	b	1
4	Linear output	$z = wx + b$	7
5	Activation	$\hat{y} = \sigma(z)$	0.999088
6	Target	y	1
7	Loss	$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$	4.16×10^{-7}

Table 2.2: Backpropagation for the output layer (layer 2)

Step	Gradient	Formula	Value	Value comes from
7	$\frac{\partial \mathcal{L}}{\partial \hat{y}}$	$\hat{y} - y$	=0.119587	Forward step 5 ($\hat{y}$), forward step 6 (y)
8	$\frac{\partial \hat{y}}{\partial z_2}$	$\hat{y}(1 - \hat{y})$	0.105233	Forward step 5 ($\hat{y}$)
9	$\frac{\partial \mathcal{L}}{\partial z_2}$	Step 7 × Step 8	=0.012580	Backprop steps 7 and 8
10	$\frac{\partial z_2}{\partial w_2}$	a_1	0.999088	Forward step 3 (a_1)
11	$\frac{\partial \mathcal{L}}{\partial w_2}$	Step 9 × Step 10	=0.012569	Backprop step 9, forward step 3
12	$\frac{\partial z_2}{\partial b_2}$	1	1	Affine layer definition
13	$\frac{\partial \mathcal{L}}{\partial b_2}$	Step 9 × Step 12	=0.012580	Backprop step 9

Note. Each gradient is computed using the chain rule and depends exclusively on quantities obtained during the forward pass or gradients propagated from subsequent layers. This explicit dependency structure mirrors the computation graph used by automatic differentiation frameworks.

2.8 Worked Example II: Three-Layer Forward and Backpropagation

2.8.1 Network Definition (Scalar)

$$z_1 = w_1 x + b_1, \quad a_1 = \mathrm{ReLU}(z_1),$$

Table 2.3: Backpropagation for the hidden layer (layer 1)

Step	Gradient	Formula	Value	Value comes from
14	$\frac{\partial z_2}{\partial a_1}$	w_2	4	Forward definition of layer 2
15	$\frac{\partial \mathcal{L}}{\partial a_1}$	Step 9 × Step 14	$=0.050319$	Backprop step 9, forward step 4
16	$\frac{\partial a_1}{\partial z_1}$	$a_1(1-a_1)$	0.000911	Forward step 3 (a_1)
17	$\frac{\partial \mathcal{L}}{\partial z_1}$	Step 15 × Step 16	$=4.584 \times 10^{-5}$	Backprop steps 15 and 16
18	$\frac{\partial z_1}{\partial w_1}$	x	2	Forward step 1 (x)
19	$\frac{\partial \mathcal{L}}{\partial w_1}$	Step 17 × Step 18	$=9.168 \times 10^{-5}$	Backprop step 17, forward step 1
20	$\frac{\partial z_1}{\partial b_1}$	1	1	Affine layer definition
21	$\frac{\partial \mathcal{L}}{\partial b_1}$	Step 17 × Step 20	$=4.584 \times 10^{-5}$	Backprop step 17

$$z_2 = w_2 a_1 + b_2, \quad a_2 = \tanh(z_2),$$
$$z_3 = w_3 a_2 + b_3, \quad \hat{y} = \sigma(z_3), \qquad \mathcal{L} = \tfrac{1}{2}(y - \hat{y})^2.$$

2.8.2 Activation Derivatives

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 1, & z > 0 \\ 0, & z \le 0 \end{cases}, \qquad \frac{d}{dz}\tanh(z) = 1 - \tanh^2(z), \qquad \frac{d}{dz}\sigma(z) = \sigma(z)\big(1 - \sigma(z)\big).$$

2.8.3 Numerical Values Used

$$x = 2,\ (w_1, b_1) = (1.5, -1),\ (w_2, b_2) = (0.5, 0.1),\ (w_3, b_3) = (2, -0.3),\ y = 1.$$

Table 2.4: Forward propagation for a three-layer network with three different activation functions

Step	Quantity	Expression	Numerical value
1	Input	x	2
2	Layer 1 weight	w_1	1.5
3	Layer 1 bias	b_1	$=1$
4	Pre-activation (L1)	$z_1 = w_1 x + b_1$	2.0
5	Activation (L1, ReLU)	$a_1 = \mathrm{ReLU}(z_1)$	2.0
6	Layer 2 weight	w_2	0.5
7	Layer 2 bias	b_2	0.1
8	Pre-activation (L2)	$z_2 = w_2 a_1 + b_2$	1.1
9	Activation (L2, tanh)	$a_2 = \tanh(z_2)$	0.800499
10	Layer 3 weight	w_3	2
11	Layer 3 bias	b_3	$=0.3$
12	Pre-activation (L3)	$z_3 = w_3 a_2 + b_3$	1.300998
13	Output activation (sigmoid)	$\hat{y} = \sigma(z_3)$	0.786003
14	Target	y	1
15	Loss	$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$	0.022897

Table 2.5: Backpropagation through the output layer (layer 3, sigmoid)

Step	Gradient	Formula	Value	Value comes from
16	$\frac{\partial \mathcal{L}}{\partial \hat{y}}$	$\hat{y} - y$	=0.213997	Forward step 13 ($\hat{y}$), step 14 (y)
17	$\frac{\partial \hat{y}}{\partial z_3}$	$\hat{y}(1-\hat{y})$	0.168202	Forward step 13 ($\hat{y}$)
18	$\frac{\partial \mathcal{L}}{\partial z_3}$	Step 16 × Step 17	=0.035995	Backprop steps 16 and 17
19	$\frac{\partial z_3}{\partial w_3}$	a_2	0.800499	Forward step 9 (a_2)
20	$\frac{\partial \mathcal{L}}{\partial w_3}$	Step 18 × Step 19	=0.028814	Backprop step 18, forward step 9
21	$\frac{\partial z_3}{\partial b_3}$	1	1	Affine layer definition
22	$\frac{\partial \mathcal{L}}{\partial b_3}$	Step 18 × Step 21	=0.035995	Backprop step 18
23	$\frac{\partial z_3}{\partial a_2}$	w_3	2	Forward step 10 (w_3)
24	$\frac{\partial \mathcal{L}}{\partial a_2}$	Step 18 × Step 23	=0.071990	Backprop step 18, forward step 10

Table 2.6: Backpropagation through hidden layer 2 (layer 2, tanh)

Step	Gradient	Formula	Value	Value comes from
25	$\frac{\partial a_2}{\partial z_2}$	$1 - a_2^2$	0.359201	Forward step 9 (a_2)
26	$\frac{\partial \mathcal{L}}{\partial z_2}$	Step 24 × Step 25	=0.025859	Backprop step 24, backprop step 25
27	$\frac{\partial z_2}{\partial w_2}$	a_1	2.0	Forward step 5 (a_1)
28	$\frac{\partial \mathcal{L}}{\partial w_2}$	Step 26 × Step 27	=0.051718	Backprop step 26, forward step 5
29	$\frac{\partial z_2}{\partial b_2}$	1	1	Affine layer definition
30	$\frac{\partial \mathcal{L}}{\partial b_2}$	Step 26 × Step 29	=0.025859	Backprop step 26
31	$\frac{\partial z_2}{\partial a_1}$	w_2	0.5	Forward step 6 (w_2)
32	$\frac{\partial \mathcal{L}}{\partial a_1}$	Step 26 × Step 31	=0.012929	Backprop step 26, forward step 6

Table 2.7: Backpropagation through hidden layer 1 (layer 1, ReLU)

Step	Gradient	Formula	Value	Value comes from
33	$\frac{\partial a_1}{\partial z_1}$	$\mathbb{1}[z_1 > 0]$	1	Forward step 4 ($z_1 = 2.0 > 0$)
34	$\frac{\partial \mathcal{L}}{\partial z_1}$	Step 32 × Step 33	=0.012929	Backprop step 32, backprop step 33
35	$\frac{\partial z_1}{\partial w_1}$	x	2	Forward step 1 (x)
36	$\frac{\partial \mathcal{L}}{\partial w_1}$	Step 34 × Step 35	=0.025859	Backprop step 34, forward step 1
37	$\frac{\partial z_1}{\partial b_1}$	1	1	Affine layer definition
38	$\frac{\partial \mathcal{L}}{\partial b_1}$	Step 34 × Step 37	=0.012929	Backprop step 34

Note. Each gradient is computed using the chain rule and depends exclusively on quantities obtained during the forward pass or gradients propagated from subsequent layers. The three activation derivatives appear explicitly: sigmoid uses $\hat{y}(1-\hat{y})$, tanh uses $1-a_2^2$, and ReLU uses the gate $\mathbb{1}[z_1 > 0]$.

2.9 Matrix-Form Backpropagation, Product Types, and Forward-Pass Memory

Goal. This section presents forward and backward propagation for a three-layer feedforward neural network using vector and matrix notation. The formulation explicitly identifies the type of each algebraic operation (matrix–vector product, Hadamard product, outer product) and clarifies which intermediate quantities must be stored during the forward pass to enable backpropagation.

Notation and dimensions. Let the input be $\mathbf{x} \in \mathbb{R}^{d_0}$ and define $\mathbf{a}_0 := \mathbf{x}$. For layers $k \in \{1, 2, 3\}$:

$$\mathbf{W}_k \in \mathbb{R}^{d_k \times d_{k-1}}, \qquad \mathbf{b}_k \in \mathbb{R}^{d_k}, \qquad \mathbf{z}_k \in \mathbb{R}^{d_k}, \qquad \mathbf{a}_k \in \mathbb{R}^{d_k}.$$

Each affine transformation uses a **matrix–vector product**:

$$\mathbf{z}_k = \mathbf{W}_k \mathbf{a}_{k-1} + \mathbf{b}_k.$$

Three-layer network with heterogeneous activations.

$$\mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1), \qquad \mathbf{a}_2 = \tanh(\mathbf{z}_2), \qquad \hat{\mathbf{y}} = \mathbf{a}_3 = \sigma(\mathbf{z}_3).$$

For a single training sample, the loss is

$$\mathcal{L} = \tfrac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2.$$

Activation derivatives (element-wise).

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 1, & z > 0 \\ 0, & z \le 0 \end{cases}, \qquad \frac{d}{dz}\tanh(z) = 1 - \tanh^2(z), \qquad \frac{d}{dz}\sigma(z) = \sigma(z)\big(1 - \sigma(z)\big).$$

The symbol $\odot$ denotes the *Hadamard (element-wise) product*.

Forward propagation (matrix form).

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1, \qquad \mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1), \tag{2.1}$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2, \qquad \mathbf{a}_2 = \tanh(\mathbf{z}_2), \tag{2.2}$$

$$\mathbf{z}_3 = \mathbf{W}_3\mathbf{a}_2 + \mathbf{b}_3, \qquad \hat{\mathbf{y}} = \sigma(\mathbf{z}_3). \tag{2.3}$$

Backpropagation (matrix form). Define error signals ("deltas"):

$$\boldsymbol{\delta}_k := \frac{\partial \mathcal{L}}{\partial \mathbf{z}_k} \in \mathbb{R}^{d_k}.$$

Layer 3 (sigmoid output).

$$\boldsymbol{\delta}_3 = (\hat{\mathbf{y}} - \mathbf{y}) \ \odot \ \big(\hat{\mathbf{y}} \odot (1 - \hat{\mathbf{y}})\big).$$

Layer 2 (tanh).

$$\boldsymbol{\delta}_2 = (\mathbf{W}_3^\top \boldsymbol{\delta}_3) \ \odot \ \big(1 - \mathbf{a}_2^{\odot 2}\big).$$

Layer 1 (ReLU).

$$\boldsymbol{\delta}_1 = (\mathbf{W}_2^\top \boldsymbol{\delta}_2) \ \odot \ \mathbb{1}[\mathbf{z}_1 > 0].$$

Table 2.8: Quantities stored during forward propagation and their role in backpropagation

Stored quantity	**Saved during forward pass**	**Used in backpropagation**
$\mathbf{x} = \mathbf{a}_0$	Input layer	$\boldsymbol{\delta}_1 \mathbf{x}^\top$
$\mathbf{z}_1$	Pre-activation (layer 1)	ReLU mask $\mathbb{1}[\mathbf{z}_1 > 0]$
$\mathbf{a}_1$	Activation (layer 1)	$\boldsymbol{\delta}_2 \mathbf{a}_1^\top$
$\mathbf{z}_2$	Pre-activation (layer 2)	$1 - \mathbf{a}_2^{\odot 2}$
$\mathbf{a}_2$	Activation (layer 2)	$\boldsymbol{\delta}_3 \mathbf{a}_2^\top$
$\mathbf{z}_3$	Pre-activation (output layer)	$\hat{\mathbf{y}} \odot (1 - \hat{\mathbf{y}})$
$\hat{\mathbf{y}}$	Network output	$\hat{\mathbf{y}} - \mathbf{y}$

Parameter gradients. For each layer $k \in \{1, 2, 3\}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_k} = \boldsymbol{\delta}_k \, \mathbf{a}_{k-1}^\top, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}_k} = \boldsymbol{\delta}_k.$$

The product $\boldsymbol{\delta}_k \, \mathbf{a}_{k-1}^\top$ is an **outer product**.

Forward-pass memory (saved tensors).

Table 2.9: Product types appearing in forward and backward propagation

Symbol / form	**Name**	**Where it appears**
$\mathbf{Wa}$, $\mathbf{W}^\top \boldsymbol{\delta}$	Matrix–vector product	Forward affine map; backward error propagation
$\odot$	Hadamard (element-wise) product	Activation derivatives and masks
$\boldsymbol{\delta} \, \mathbf{a}^\top$	Outer product	Weight gradients

Summary of product types.

2.10 Chain Rule Accumulation and Gradient Flow

Motivation. In backpropagation, gradients are accumulated whenever a variable influences the loss through multiple computational paths. This section makes the summation in the chain rule explicit and connects it directly to how automatic differentiation frameworks accumulate gradients.

Computation graph example. Consider the scalar computation

$$\begin{aligned} v_3 &= x^2, \\ v_5 &= 2v_3, \\ v_6 &= v_3 + 1, \\ f &= v_5\, v_6. \end{aligned}$$

The intermediate variable v_3 feeds into two downstream nodes.

Forward pass. For $x = 3$:

$$v_3 = 9, \qquad v_5 = 18, \qquad v_6 = 10, \qquad f = 180.$$

Backward pass. The multivariable chain rule gives

$$\frac{\partial f}{\partial v_3} = \frac{\partial f}{\partial v_6}\frac{\partial v_6}{\partial v_3} + \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_3}.$$

Evaluating:

$$\frac{\partial f}{\partial v_5} = v_6 = 10, \qquad \frac{\partial f}{\partial v_6} = v_5 = 18,$$
$$\frac{\partial v_5}{\partial v_3} = 2, \qquad \frac{\partial v_6}{\partial v_3} = 1.$$

Thus,

$$\frac{\partial f}{\partial v_3} = 18 + 20 = 38.$$

Interpretation. The summation corresponds exactly to gradient accumulation in backpropagation: each outgoing edge contributes a partial gradient, and the total gradient is their sum.

2.11 Hessian–Vector Products and Second-Order Structure

Let $f(\boldsymbol{\theta})$ be a scalar loss with parameters $\boldsymbol{\theta} \in \mathbb{R}^W$. The gradient is $\nabla f(\boldsymbol{\theta}) \in \mathbb{R}^W$ and the Hessian is

$$\mathbf{H} = \nabla^2 f(\boldsymbol{\theta}) \in \mathbb{R}^{W \times W}.$$

Explicit construction of $\mathbf{H}$ is infeasible for modern neural networks. Instead, second-order methods rely on the *Hessian–vector product*

$$\mathbf{Hu} = \nabla^2 f(\boldsymbol{\theta})\, \mathbf{u},$$

for a chosen direction $\mathbf{u} \in \mathbb{R}^W$.

A key identity enables efficient computation:

$$\mathbf{Hu} = \nabla_{\boldsymbol{\theta}} \left(\nabla f(\boldsymbol{\theta})^\top \mathbf{u} \right).$$

This identity allows Hessian–vector products to be computed using standard backpropagation without forming $\mathbf{H}$ explicitly.

2.11.1 Worked Example: Scalar Hessian–Vector Product

Consider

$$f(w_1, w_2) = w_1^2 + 3w_1w_2 + 2w_2^2.$$

The gradient is

$$\nabla f = \begin{pmatrix} 2w_1 + 3w_2 \\ 3w_1 + 4w_2 \end{pmatrix},$$

and the Hessian is

$$\nabla^2 f = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix}.$$

For $\mathbf{u} = (1, -1)^\top$,

$$\mathbf{Hu} = \begin{pmatrix} -1 \\ -1 \end{pmatrix}.$$

Using the nabla identity,

$$\nabla f^\top \mathbf{u} = -(w_1 + w_2), \qquad \nabla_{\mathbf{w}}(-(w_1 + w_2)) = \begin{pmatrix} -1 \\ -1 \end{pmatrix},$$

confirming the result.

2.11.2 Hessian–Vector Products via Backpropagation: Concrete Three-Layer Example

Forward pass. The forward pass computes

$$\hat{y} = w_3\big(w_2(w_1x + b_1) + b_2\big) + b_3,$$

$$\mathcal{L} = \tfrac{1}{2}\big(\hat{y} - y\big)^2.$$

First backward pass (gradient computation). Backpropagation yields the gradient of the loss with respect to all parameters:

$$\nabla f(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial b_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \\ \frac{\partial \mathcal{L}}{\partial b_2} \\ \frac{\partial \mathcal{L}}{\partial w_3} \\ \frac{\partial \mathcal{L}}{\partial b_3} \end{pmatrix} = (\hat{y} - y) \begin{pmatrix} w_3w_2x \\ w_3w_2 \\ w_3a_1 \\ w_3 \\ a_2 \\ 1 \end{pmatrix}.$$

This is the standard gradient computed during ordinary backpropagation.

Concrete three-layer network usage. To make the role of the Hessian–vector product explicit in a deep learning setting, consider the three-layer scalar network

$$z_1 = w_1x + b_1, \qquad a_1 = z_1,$$

$$z_2 = w_2a_1 + b_2, \qquad a_2 = z_2,$$

$$z_3 = w_3a_2 + b_3, \qquad \hat{y} = z_3, \qquad \mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2,$$

with parameter vector

$$\boldsymbol{\theta} = (w_1, b_1, w_2, b_2, w_3, b_3)^\top.$$

First backward pass (gradient). Applying standard backpropagation yields

$$\nabla f(\boldsymbol{\theta}) = (\hat{y} - y) \begin{pmatrix} w_3 w_2 x \\ w_3 w_2 \\ w_3 a_1 \\ w_3 \\ a_2 \\ 1 \end{pmatrix}.$$

This is the gradient used by first-order optimizers such as SGD or Adam.

Scalar construction for curvature probing. Given a direction vector

$$\mathbf{u} = (u_{w_1}, u_{b_1}, u_{w_2}, u_{b_2}, u_{w_3}, u_{b_3})^\top,$$

we form the scalar

$$s(\boldsymbol{\theta}) = \nabla f(\boldsymbol{\theta})^\top \mathbf{u} = (\hat{y} - y)\Big(u_{w_1} w_3 w_2 x + u_{b_1} w_3 w_2 + u_{w_2} w_3 a_1 + u_{b_2} w_3 + u_{w_3} a_2 + u_{b_3}\Big).$$

This scalar measures how the gradient of the loss changes when the parameters are perturbed along the direction $\mathbf{u}$.

Second backward pass (Hessian–vector product). Differentiating $s(\boldsymbol{\theta})$ with respect to the parameters gives

$$\nabla_{\boldsymbol{\theta}} s(\boldsymbol{\theta}) = \nabla^2 f(\boldsymbol{\theta})\, \mathbf{u} = \mathbf{H}\mathbf{u}.$$

Each component of this vector captures the second-order sensitivity of the loss with respect to the corresponding parameter along the chosen direction $\mathbf{u}$.

How this fits into forward and backward propagation. Operationally, the computation can be summarized as:

- **Forward pass:** compute activations $(a_1, a_2, \hat{y})$ and the loss $\mathcal{L}$.
- **First backward pass:** compute the gradient $\nabla f(\boldsymbol{\theta})$ using standard backpropagation.
- **Scalar projection:** form $s(\boldsymbol{\theta}) = \nabla f(\boldsymbol{\theta})^\top \mathbf{u}$.
- **Second backward pass:** backpropagate through $s(\boldsymbol{\theta})$ to obtain $\mathbf{H}\mathbf{u}$.

Interpretation in deep learning terms.

- The first backward pass computes *slopes* of the loss.
- The second backward pass computes how those slopes *change* along a specific direction in parameter space.
- The vector $\mathbf{u}$ selects which direction of curvature is being examined.
- No Hessian matrix is formed; curvature is accessed implicitly through backpropagation.

Practical meaning. In large neural networks, the same mechanism applies when $\boldsymbol{\theta}$ contains millions of parameters. Hessian–vector products computed in this manner are used inside iterative solvers such as conjugate gradient to obtain curvature-aware update directions, analyze sharpness of minima, and enable Hessian-free optimization—all while preserving the standard forward and backward propagation structure used in deep learning frameworks.

Role of the second backward pass: clarification. The purpose of the second backward pass depends on how the Hessian–vector product is used. The same mathematical operation serves two distinct roles in deep learning.

- **Standard first-order training (SGD, Adam).**
  - Training consists of one forward pass and one backward pass per batch.
  - The backward pass computes the gradient $\nabla f(\boldsymbol{\theta})$.
  - Parameter updates depend only on first-order information.
  - No Hessian–vector products are required.
  - In this setting, a second backward pass is unnecessary.
- **Second backward pass used for analysis or diagnostics.**
  - The second backward pass computes $\mathbf{Hu} = \nabla^2 f(\boldsymbol{\theta})\,\mathbf{u}$.
  - The direction $\mathbf{u}$ is chosen externally (e.g., the gradient direction or a random vector).
  - Quantities such as $\mathbf{u}^\top \mathbf{Hu}$ are used to assess curvature, sharpness, or stability.
  - Training updates are not modified.
  - The second pass is used only to monitor or analyze convergence behavior.
- **Second backward pass used for optimization.**
  - In second-order or Hessian-free methods, the second backward pass directly influences parameter updates.
  - Optimization algorithms such as conjugate gradient solve linear systems of the form

    $$\mathbf{Hp} = -\nabla f(\boldsymbol{\theta})$$

    using repeated Hessian–vector products.
  - Each Hessian–vector product requires a second backward pass.
  - The resulting direction $\mathbf{p}$ determines how the model parameters are updated.
  - In this case, the second backward pass is part of the training algorithm itself.
- **Why the second backward pass is not always used.**
  - Hessian–vector products are computationally more expensive than gradients.
  - Stochastic noise in mini-batch training complicates curvature estimation.
  - First-order methods often provide sufficient performance at lower computational cost.
  - As a result, curvature-based methods are applied selectively.

Summary.

- One backward pass computes the slope of the loss.
- A second backward pass computes how that slope changes.
- The second backward pass may be used for analysis only or as an integral part of optimization.
- Whether it affects training depends on how the Hessian–vector product is used.

When Hessian–vector products are used in practice. The use of Hessian–vector products depends on the choice of optimization strategy.

- **Typical deep learning training.**
  - Most deep learning models are trained using first-order optimizers such as SGD, Adam, or RMSProp.
  - These methods require only one forward pass and one backward pass per batch.
  - Curvature information is not explicitly computed.
  - Hessian–vector products are therefore *not* used during standard training.
- **Second-order and curvature-aware optimization.**
  - When second-order information is desired, Hessian–vector products become essential.
  - Methods such as Hessian-free optimization, Newton–CG, and other curvature-aware algorithms rely on repeated evaluations of $\mathbf{Hu}$.
  - The second backward pass is used to compute how gradients change along selected directions.
  - This additional information is used to construct improved update directions for the model parameters.
- **Practical implication.**
  - In most applications, first-order methods are preferred due to their simplicity and computational efficiency.
  - Hessian–vector products are used selectively, either for advanced optimization or for analyzing the geometry of the loss landscape.
  - When a second-order optimizer is employed, the additional backward pass becomes a necessary and integral part of the training procedure.

Key takeaway.

- Standard training does not require Hessian–vector products.
- Hessian–vector products become useful and necessary when second-order optimization methods are employed.

Role and selection of the direction vector in second-order optimization. Second-order optimization methods do not operate on the Hessian matrix $\mathbf{H}$ directly. Instead, they rely on Hessian–vector products of the form $\mathbf{Hu}$, which require the specification of a direction vector $\mathbf{u}$ in parameter space.

- **Why a direction vector is required.**
  - The Hessian $\mathbf{H} \in \mathbb{R}^{W \times W}$ is too large to form or store explicitly in deep learning models.
  - Second-order methods therefore interact with the Hessian only through its action on vectors.
  - The vector $\mathbf{u}$ specifies the direction along which curvature information is extracted.
- **Direction vectors are not arbitrary parameters.**
  - The vector $\mathbf{u}$ is not learned and is not part of the model.

  - It is generated algorithmically by the optimization method.
  - Its purpose is to probe curvature or construct update directions.

- **Typical choices of the direction vector u.**
  - *Gradient direction:*
$$\mathbf{u} = \nabla f(\boldsymbol{\theta}),$$
used to measure curvature along the descent direction and to assess sharpness or stability.
  - *Conjugate gradient search directions:*
$$\mathbf{u} = \mathbf{p}_k,$$
where $\mathbf{p}_k$ is the current search direction generated by the conjugate gradient algorithm when solving
$$\mathbf{H}\mathbf{p} = -\nabla f(\boldsymbol{\theta}).$$
  - *Random probe directions:* used to estimate average curvature or spectral properties of the Hessian.

- **How the direction vector affects optimization.**
  - The quality of the second-order update depends on the choice of $\mathbf{u}$.
  - In Newton–CG and Hessian-free optimization, the direction vectors are chosen adaptively to approximate the Newton step.
  - Poorly chosen directions yield poor curvature estimates and ineffective updates.

- **Connection to the second backward pass.**
  - Each chosen direction $\mathbf{u}$ triggers a Hessian–vector product $\mathbf{H}\mathbf{u}$.
  - Computing $\mathbf{H}\mathbf{u}$ requires a second backward pass through the scalar $\nabla f(\boldsymbol{\theta})^\top \mathbf{u}$.
  - Multiple directions may be evaluated during a single optimization step.

Key takeaway.

- Second-order optimization requires both gradient information and carefully chosen direction vectors.
- Hessian–vector products provide curvature information only along those directions.
- The effectiveness of second-order methods depends critically on how the direction vectors are defined and updated.

Where Newton–CG is used in practice. Newton–Conjugate Gradient (Newton–CG) methods appear in deep learning and scientific computing primarily in settings where curvature information is valuable and the optimization problem is sufficiently structured.

- **Hessian-free neural network training.**
  - Newton–CG is the core optimization engine in Hessian-free training methods.
  - The Newton update direction $\mathbf{p}$ is obtained by approximately solving
$$\mathbf{H}\mathbf{p} = -\nabla f(\boldsymbol{\theta})$$
using conjugate gradient iterations.

  - Each CG iteration requires Hessian–vector products **Hu** rather than the full Hessian.
  - **Example applications:**
    * training deep autoencoders for dimensionality reduction,
    * optimization of recurrent neural networks (RNNs) and LSTMs suffering from vanishing or exploding gradients,
    * training very deep feedforward networks where SGD converges slowly due to ill-conditioned curvature.

- **Large-scale scientific and engineering optimization.**
  - Newton–CG is widely used in physics-based optimization problems involving smooth, differentiable objective functions.
  - **Example applications:**
    * inverse problems in imaging (e.g., tomography and deconvolution),
    * parameter estimation in PDE-constrained optimization,
    * aerodynamic shape optimization and structural mechanics,
    * fluid dynamics and climate modeling simulations.
  - In these problems, curvature information dramatically reduces the number of required optimization iterations.

- **Classical machine learning models.**
  - Newton–CG is commonly used when the loss function is convex or nearly convex.
  - **Example applications:**
    * logistic regression with large feature sets,
    * generalized linear models (GLMs),
    * kernel-based learning methods with smooth regularization.
  - In these settings, Newton–CG often converges in far fewer iterations than gradient descent.

- **Gauss–Newton and Fisher-based methods in deep learning.**
  - Newton–CG is frequently applied to Gauss–Newton or Fisher information matrix approximations of the Hessian.
  - These approximations are positive semi-definite, making CG numerically stable.
  - **Example applications:**
    * training neural networks with squared-error or likelihood- based losses,
    * natural gradient methods and trust-region approaches,
    * curvature-aware fine-tuning of pretrained deep networks.

- **Why Newton–CG is not the default in deep learning.**
  - Each Newton step requires multiple Hessian–vector products, increasing computational cost.
  - Mini-batch stochasticity introduces noise into curvature estimates.
  - First-order methods (SGD, Adam) often achieve competitive performance with significantly lower overhead.
  - As a result, Newton–CG is used selectively rather than as a default optimizer.

Summary.

- Newton–CG is most useful when curvature information significantly improves convergence.
- It is widely used in Hessian-free neural network training and large-scale scientific and engineering optimization.
- In deep learning, it is typically applied with Gauss–Newton or Fisher approximations rather than the exact Hessian.
- Its limited use is due to computational cost and stochastic noise, not lack of effectiveness.

2.11.3 Role of the Second Backward Pass

- **First-order training (SGD, Adam):** one forward pass and one backward pass compute gradients.
- **Analysis and diagnostics:** a second backward pass computes curvature quantities such as $\mathbf{u}^\top \mathbf{H}\mathbf{u}$ without affecting updates.
- **Second-order optimization:** Hessian–vector products directly determine update directions in methods such as Newton–CG and Hessian-free optimization.

The second backward pass is therefore optional and task-dependent.

2.11.4 Direction Vectors in Second-Order Optimization

The vector $\mathbf{u}$ is not a model parameter. It is generated by the optimization algorithm.

Typical choices include:

- the gradient direction $\mathbf{u} = \nabla f(\boldsymbol{\theta})$,
- conjugate gradient search directions,
- random probe directions for curvature estimation.

Each choice extracts curvature information along a specific direction.

2.11.5 Newton–CG and Hessian-Free Methods

Newton–CG methods solve

$$\mathbf{H}\mathbf{p} = -\nabla f(\boldsymbol{\theta})$$

using conjugate gradient iterations. Each iteration requires a Hessian–vector product.

These methods appear in:

- Hessian-free neural network training,
- RNN and LSTM optimization,
- inverse problems and PDE-constrained optimization,
- Gauss–Newton and Fisher-based approximations.

They are used selectively due to computational cost and stochastic noise.

Table 2.10: Comparison of PyTorch behavior during training and inference

Aspect	**Training mode (autograd enabled)**	**Inference mode (autograd disabled)**
Forward pass behavior	Records the full computational graph of tensor operations	Performs forward computation only
Intermediate activations	Stored in memory for backpropagation	Not stored
Gradient tracking	Tracks tensors with `requires_grad=True`	No gradient tracking
Operation metadata	Saved to enable chain rule application during backward pass	Not saved
Backward pass	Required to compute gradients of the loss	Not performed
Gradient storage	Gradients accumulated in `.grad` fields of parameters	No gradient buffers allocated
Higher-order derivatives	Supported when explicitly requested (e.g., Hessian–vector products)	Not available
Memory usage	Higher due to graph and activation storage	Significantly reduced
Runtime performance	Slower due to bookkeeping overhead	Faster and more efficient
Typical use case	Model training and optimization	Evaluation, validation, and deployment
How mode is enabled	Default behavior when gradients are required	Using `torch.no_grad()` and `model.eval()`

2.12 Training vs. Inference in Automatic Differentiation Frameworks

Key takeaway. During training, PyTorch retains intermediate computations and constructs a computational graph to enable automatic differentiation. During inference, this bookkeeping is disabled to reduce memory usage and improve execution speed, while preserving the same forward computation.

Clarification: meaning of the computational graph. In this context, the term *graph* refers to PyTorch's dynamic computational graph constructed during the forward pass when automatic differentiation is enabled. This graph is a directed acyclic graph whose nodes correspond to tensor operations and whose edges represent data dependencies between tensors. Each node stores the local derivative information required to apply the chain rule during backpropagation, along with references to its parent tensors. During the backward pass, PyTorch traverses this graph in reverse to compute gradients of the loss with respect to model parameters.

Chapter 3

Jacobians in Backpropagation and Second-Order Analysis

3.1 Overview

Backpropagation is often introduced algorithmically as a sequence of local gradient computations propagated backward through a network. At a mathematical level, however, backpropagation is an exact application of the multivariable chain rule and can be understood as repeated multiplication by Jacobian transposes.

This chapter presents a unified Jacobian-based interpretation of backpropagation, explains why explicit Jacobian matrices are rarely constructed in practice, and illustrates the concepts through both symbolic and numerical examples. The connection to second-order methods and Hessian–vector products is also clarified.

3.2 Jacobian View of the Chain Rule

Consider a feedforward network expressed as a composition of functions:

$$\mathbf{a}_0 = \mathbf{x}, \qquad \mathbf{a}_k = f_k(\mathbf{a}_{k-1}), \quad k = 1, \dots, L,$$

with scalar loss

$$\mathcal{L} = \ell(\mathbf{a}_L).$$

The Jacobian of layer k with respect to its input is

$$\mathbf{J}_k = \frac{\partial \mathbf{a}_k}{\partial \mathbf{a}_{k-1}}.$$

Applying the multivariable chain rule yields

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_0} = \mathbf{J}_1^\top \mathbf{J}_2^\top \cdots \mathbf{J}_L^\top \frac{\partial \mathcal{L}}{\partial \mathbf{a}_L}.$$

Thus, backpropagation is mathematically equivalent to multiplying the upstream gradient by a sequence of Jacobian transposes.

3.3 Why Jacobians Are Not Formed Explicitly

If layer k maps $\mathbb{R}^{d_{k-1}} \to \mathbb{R}^{d_k}$, then

$$\mathbf{J}_k \in \mathbb{R}^{d_k \times d_{k-1}}.$$

In modern neural networks, d_k and d_{k-1} may be millions, making explicit construction of $\mathbf{J}_k$ infeasible.

Backpropagation requires only the product

$$\mathbf{J}_k^\top \mathbf{v},$$

where $\mathbf{v}$ is the upstream gradient. Automatic differentiation frameworks compute this product implicitly using local derivatives and the chain rule, avoiding explicit Jacobian storage.

This operation is known as a **vector–Jacobian product (VJP)**.

3.4 Three-Layer Network: Matrix Form

Consider a three-layer feedforward network:

$$\mathbf{a}_0 = \mathbf{x},$$

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{a}_0 + \mathbf{b}_1, \quad \mathbf{a}_1 = \phi(\mathbf{z}_1),$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2, \quad \mathbf{a}_2 = \psi(\mathbf{z}_2),$$

$$z_3 = \mathbf{W}_3\mathbf{a}_2 + b_3, \quad \hat{y} = \sigma(z_3),$$

with scalar loss

$$\mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2.$$

The Jacobians of each layer with respect to its input are:

$$\mathbf{J}_1 = \operatorname{diag}(\phi'(\mathbf{z}_1))\,\mathbf{W}_1,$$

$$\mathbf{J}_2 = \operatorname{diag}(\psi'(\mathbf{z}_2))\,\mathbf{W}_2,$$

$$\mathbf{J}_3 = \sigma'(z_3)\,\mathbf{W}_3.$$

Backpropagation computes:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = (\hat{y} - y)\,\mathbf{J}_3, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_2}\mathbf{J}_2, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{a}_0} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1}\mathbf{J}_1.$$

Equivalently, for any intermediate activation $\mathbf{a}_n$, the chain rule gives

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_n} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}}\,\mathbf{J}_L\mathbf{J}_{L-1}\cdots\mathbf{J}_{n+1},$$

where $\mathbf{J}_k = \partial \mathbf{a}_k / \partial \mathbf{a}_{k-1}$. This expression makes explicit that backpropagation is a sequence of vector–Jacobian products propagating the loss gradient from the output layer back to layer n.

3.5 Numerical Example with Explicit Jacobians (with activation and loss derivatives)

We use a three-layer network with three different nonlinearities: Layer 1 uses ReLU, Layer 2 uses tanh, and the output layer uses a sigmoid. The loss is mean squared error (single sample).

3.5.1 Network definition

Let $d_0 = d_1 = d_2 = 2$ and define

$$\mathbf{a}_0 = \mathbf{x}, \qquad \mathbf{z}_1 = \mathbf{W}_1\mathbf{a}_0 + \mathbf{b}_1, \quad \mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1),$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2, \quad \mathbf{a}_2 = \tanh(\mathbf{z}_2),$$

$$z_3 = \mathbf{W}_3\mathbf{a}_2 + b_3, \quad \hat{y} = \sigma(z_3), \qquad \mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2.$$

We choose

$$\mathbf{x} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad \mathbf{W}_1 = \begin{pmatrix} 1 & 2 \\ -0.5 & 1.5 \end{pmatrix}, \quad \mathbf{b}_1 = \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix},$$

$$\mathbf{W}_2 = \begin{pmatrix} 0.7 & -1.2 \\ 1.1 & 0.3 \end{pmatrix}, \quad \mathbf{b}_2 = \begin{pmatrix} 0.05 \\ -0.1 \end{pmatrix}, \quad \mathbf{W}_3 = \begin{pmatrix} 1.4 & -0.8 \end{pmatrix}, \quad b_3 = 0.2, \quad y = 1.$$

3.5.2 Activation derivatives and loss derivative

The element-wise derivatives used in the Jacobians are:

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 1, & z > 0 \\ 0, & z \le 0 \end{cases}, \qquad \frac{d}{dz}\tanh(z) = 1 - \tanh^2(z), \qquad \sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

The loss derivative with respect to the prediction is

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y.$$

3.5.3 Forward pass (numerical)

Layer 1 (ReLU).

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 = \begin{pmatrix} 1 & 2 \\ -0.5 & 1.5 \end{pmatrix}\begin{pmatrix} 2 \\ 1 \end{pmatrix} + \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix} = \begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix}.$$

$$\mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1) = \begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix} \quad \text{(both entries are positive).}$$

Layer 2 (tanh).

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2 = \begin{pmatrix} 0.7 & -1.2 \\ 1.1 & 0.3 \end{pmatrix}\begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix} + \begin{pmatrix} 0.05 \\ -0.1 \end{pmatrix} = \begin{pmatrix} 2.56 \\ 4.50 \end{pmatrix}.$$

$$\mathbf{a}_2 = \tanh(\mathbf{z}_2) = \begin{pmatrix} \tanh(2.56) \\ \tanh(4.50) \end{pmatrix} = \begin{pmatrix} 0.988119 \\ 0.999753 \end{pmatrix}.$$

Output layer (sigmoid).

$$z_3 = \mathbf{W}_3\mathbf{a}_2 + b_3 = \begin{pmatrix} 1.4 & -0.8 \end{pmatrix}\begin{pmatrix} 0.988119 \\ 0.999753 \end{pmatrix} + 0.2 = 0.783564.$$

$$\hat{y} = \sigma(z_3) = \frac{1}{1 + e^{-z_3}} = 0.686448, \qquad \mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2 = \tfrac{1}{2}(0.686448 - 1)^2.$$

3.5.4 Local derivatives (numerical)

Loss derivative.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y = 0.686448 - 1 = -0.313552.$$

Sigmoid derivative.

$$\sigma'(z_3) = \hat{y}(1-\hat{y}) = 0.686448(1 - 0.686448) = 0.215271.$$

tanh **derivative (element-wise).**

$$\tanh'(\mathbf{z}_2) = 1 - \mathbf{a}_2^{\odot 2} = \begin{pmatrix} 1 - 0.988119^2 \\ 1 - 0.999753^2 \end{pmatrix} = \begin{pmatrix} 0.023620 \\ 0.000494 \end{pmatrix}.$$

ReLU derivative (element-wise). Since $\mathbf{z}_1 = (4.1, 0.3)^\top$ is strictly positive component-wise,

$$\mathrm{ReLU}'(\mathbf{z}_1) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

3.5.5 Explicit Jacobians

Jacobian of Layer 1: $\mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1)$, $\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$.

$$\mathbf{J}_1 = \frac{\partial \mathbf{a}_1}{\partial \mathbf{x}} = \mathrm{diag}(\mathrm{ReLU}'(\mathbf{z}_1))\,\mathbf{W}_1 = \mathbf{I}\,\mathbf{W}_1 = \mathbf{W}_1.$$

Jacobian of Layer 2: $\mathbf{a}_2 = \tanh(\mathbf{z}_2)$, $\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2$.

$$\mathbf{J}_2 = \frac{\partial \mathbf{a}_2}{\partial \mathbf{a}_1} = \mathrm{diag}(\tanh'(\mathbf{z}_2))\,\mathbf{W}_2 = \begin{pmatrix} 0.023620 & 0 \\ 0 & 0.000494 \end{pmatrix}\begin{pmatrix} 0.7 & -1.2 \\ 1.1 & 0.3 \end{pmatrix} = \begin{pmatrix} 0.016535 & -0.028345 \\ 0.000543 & 0.000148 \end{pmatrix}.$$

Jacobian of Output: $\hat{y} = \sigma(z_3)$, $z_3 = \mathbf{W}_3\mathbf{a}_2 + b_3$.

$$\mathbf{J}_3 = \frac{\partial \hat{y}}{\partial \mathbf{a}_2} = \sigma'(z_3)\,\mathbf{W}_3 = 0.215271\begin{pmatrix} 1.4 & -0.8 \end{pmatrix} = \begin{pmatrix} 0.301332 & -0.172190 \end{pmatrix}.$$

3.5.6 Backpropagation via Jacobian products (numerical)

Backpropagation is a sequence of vector–Jacobian products:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\,\frac{\partial \hat{y}}{\partial \mathbf{a}_2} = (\hat{y} - y)\,\mathbf{J}_3.$$

Numerically,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = (-0.313552)\begin{pmatrix} 0.301332 & -0.172190 \end{pmatrix} = \begin{pmatrix} -0.094506 & 0.053988 \end{pmatrix}.$$

Next,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_2}\,\mathbf{J}_2,$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \begin{pmatrix} -0.094506 & 0.053988 \end{pmatrix}\begin{pmatrix} 0.016535 & -0.028345 \\ 0.000543 & 0.000148 \end{pmatrix} = \begin{pmatrix} -0.001435 & 0.002267 \end{pmatrix}.$$

Finally,

$$\nabla_{\mathbf{x}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1}\,\mathbf{J}_1 = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1}\,\mathbf{W}_1,$$

$$\nabla_{\mathbf{x}}\mathcal{L} = \begin{pmatrix} -0.001435 & 0.002267 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ -0.5 & 1.5 \end{pmatrix} = \begin{pmatrix} -0.002876 & 0.000531 \end{pmatrix}.$$

Equivalently, written as a column vector:

$$\nabla_{\mathbf{x}}\mathcal{L} = \begin{pmatrix} -0.002876 \\ 0.000531 \end{pmatrix}.$$

3.6 Same Numerical Example via Explicit Chain Rule (no Jacobian shortcut)

This section computes the same gradient as Section **??** (Jacobian products), but now by writing the chain rule explicitly using the loss derivative and the activation derivatives.

3.6.1 Local derivatives used

Recall

$$\mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2, \qquad \hat{y} = \sigma(z_3), \qquad z_3 = \mathbf{W}_3\mathbf{a}_2 + b_3,$$

$$\mathbf{a}_2 = \tanh(\mathbf{z}_2), \qquad \mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2, \qquad \mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1), \qquad \mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1.$$

The scalar loss derivative is

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y.$$

The activation derivatives are

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)), \qquad \tanh'(z) = 1 - \tanh^2(z), \qquad \mathrm{ReLU}'(z) = \mathbb{1}[z > 0].$$

3.6.2 Forward values (from the numerical example)

Using the same numerical values as in the Jacobian section:

$$\mathbf{z}_1 = \begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix}, \quad \mathbf{a}_1 = \begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix}, \quad \mathbf{z}_2 = \begin{pmatrix} 2.56 \\ 4.50 \end{pmatrix}, \quad \mathbf{a}_2 = \begin{pmatrix} 0.98811896 \\ 0.99975321 \end{pmatrix},$$

$$z_3 = 0.78356397, \quad \hat{y} = 0.68644772, \quad y = 1.$$

Therefore

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y = -0.31355228, \qquad \sigma'(z_3) = \hat{y}(1 - \hat{y}) = 0.21523725.$$

For the tanh layer (element-wise):

$$\tanh'(\mathbf{z}_2) = \begin{pmatrix} 1 - a_{2,1}^2 \\ 1 - a_{2,2}^2 \end{pmatrix} = \begin{pmatrix} 0.02362093 \\ 0.00049352 \end{pmatrix}.$$

For ReLU:

$$\mathrm{ReLU}'(\mathbf{z}_1) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad (\text{since } z_{1,1} = 4.1 > 0,\ z_{1,2} = 0.3 > 0).$$

3.6.3 Backward pass by chain rule (step-by-step)

Step 1: from loss to output pre-activation. Because $\hat{y} = \sigma(z_3)$,

$$\frac{\partial \mathcal{L}}{\partial z_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3} = (\hat{y} - y)\, \sigma'(z_3).$$

Numerically:

$$\frac{\partial \mathcal{L}}{\partial z_3} = (-0.31355228)(0.21523725) = -0.06748670.$$

Step 2: from z_3 to $\mathbf{a}_2$. Since $z_3 = \mathbf{W}_3 \mathbf{a}_2 + b_3$ with $\mathbf{W}_3 = (1.4,\ -0.8)$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = \frac{\partial \mathcal{L}}{\partial z_3} \frac{\partial z_3}{\partial \mathbf{a}_2} = \frac{\partial \mathcal{L}}{\partial z_3} \mathbf{W}_3.$$

Numerically:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = (-0.06748670) \begin{pmatrix} 1.4 & -0.8 \end{pmatrix} = \begin{pmatrix} -0.09448138 & 0.05398936 \end{pmatrix}.$$

Step 3: from $\mathbf{a}_2$ to $\mathbf{z}_2$ (tanh gate). Because $\mathbf{a}_2 = \tanh(\mathbf{z}_2)$ element-wise,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} \operatorname{diag}\big(\tanh'(\mathbf{z}_2)\big).$$

Numerically:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2} = \begin{pmatrix} -0.09448138 & 0.05398936 \end{pmatrix} \begin{pmatrix} 0.02362093 & 0 \\ 0 & 0.00049352 \end{pmatrix} = \begin{pmatrix} -0.00223199 & 0.00002664 \end{pmatrix}.$$

Step 4: from $\mathbf{z}_2$ to $\mathbf{a}_1$ (linear map). Since $\mathbf{z}_2 = \mathbf{W}_2 \mathbf{a}_1 + \mathbf{b}_2$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_2} \frac{\partial \mathbf{z}_2}{\partial \mathbf{a}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_2} \mathbf{W}_2,$$

with

$$\mathbf{W}_2 = \begin{pmatrix} 0.7 & -1.2 \\ 1.1 & 0.3 \end{pmatrix}.$$

Numerically:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \begin{pmatrix} -0.00223199 & 0.00002664 \end{pmatrix} \begin{pmatrix} 0.7 & -1.2 \\ 1.1 & 0.3 \end{pmatrix} = \begin{pmatrix} -0.00153294 & 0.00268614 \end{pmatrix}.$$

Step 5: from $\mathbf{a}_1$ to $\mathbf{z}_1$ (ReLU gate). Because $\mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1)$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} \operatorname{diag}\big(\mathrm{ReLU}'(\mathbf{z}_1)\big).$$

Here $\mathrm{ReLU}'(\mathbf{z}_1) = (1, 1)^\top$, so the diagonal is $\mathbf{I}$ and

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \begin{pmatrix} -0.00153294 & 0.00268614 \end{pmatrix}.$$

Step 6: from $\mathbf{z}_1$ to input $\mathbf{x}$. Since $\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$,

$$\nabla_{\mathbf{x}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_1}\,\mathbf{W}_1, \quad \mathbf{W}_1 = \begin{pmatrix} 1 & 2 \\ -0.5 & 1.5 \end{pmatrix}.$$

Numerically:

$$\nabla_{\mathbf{x}}\mathcal{L} = \begin{pmatrix} -0.00153294 & 0.00268614 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ -0.5 & 1.5 \end{pmatrix} = \begin{pmatrix} -0.00287601 & 0.00096332 \end{pmatrix}.$$

Equivalently as a column vector:

$$\nabla_{\mathbf{x}}\mathcal{L} = \begin{pmatrix} -0.00287601 \\ 0.00096332 \end{pmatrix}.$$

3.6.4 Connection to the Jacobian form

Grouping the same steps into layer Jacobians gives exactly:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = (\hat{y} - y)\,\mathbf{J}_3, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_2}\mathbf{J}_2, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{a}_0} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_1}\mathbf{J}_1,$$

with

$$\mathbf{J}_1 = \operatorname{diag}(\mathrm{ReLU}'(\mathbf{z}_1))\,\mathbf{W}_1, \qquad \mathbf{J}_2 = \operatorname{diag}(\tanh'(\mathbf{z}_2))\,\mathbf{W}_2, \qquad \mathbf{J}_3 = \sigma'(z_3)\,\mathbf{W}_3.$$

Interpretation.

- The loss derivative $\partial\mathcal{L}/\partial\hat{y}$ produces the initial upstream signal.
- The output Jacobian $\mathbf{J}_3$ injects the sigmoid derivative and the linear weights $\mathbf{W}_3$.
- The hidden Jacobian $\mathbf{J}_2$ injects the tanh derivative via a diagonal gate.
- The input Jacobian $\mathbf{J}_1$ injects the ReLU derivative via another diagonal gate.
- At no point is any full Jacobian of the entire network constructed; only local Jacobians are used in products.

Note. Minor differences at the last digit can occur depending on rounding of $\tanh(\cdot)$ and $\sigma(\cdot)$.

3.7 Connection to Second-Order Derivatives

The Jacobian framework naturally extends to second-order analysis. Given the gradient $\nabla f(\boldsymbol{\theta})$, the Hessian–vector product is defined as

$$\mathbf{H}\mathbf{u} = \nabla_{\boldsymbol{\theta}}\big(\nabla f(\boldsymbol{\theta})^\top \mathbf{u}\big).$$

This computation requires:

- one forward pass,
- one backward pass to compute $\nabla f(\boldsymbol{\theta})$,
- formation of the scalar $\nabla f(\boldsymbol{\theta})^\top \mathbf{u}$,
- a second backward pass.

No Hessian matrix is ever formed.

3.8 Summary

- Backpropagation is mathematically equivalent to multiplying by Jacobian transposes.
- Automatic differentiation computes vector–Jacobian products implicitly.
- Explicit Jacobian matrices are avoided due to prohibitive size.
- The same machinery enables efficient computation of Hessian–vector products.
- First-order and second-order optimization methods differ only in how many backward passes they require.

3.9 Worked Example III: Full Forward/Backward Pass and Gradient Descent Update

We consider a three-layer scalar network with three different activation functions: Layer 1 uses ReLU, Layer 2 uses tanh, and the output layer uses a sigmoid. The loss is mean squared error (single sample):

$$\mathcal{L} = \tfrac{1}{2}\big(\hat{y} - y\big)^2.$$

3.9.1 Network definition

$$\begin{aligned} z_1 &= w_1 x + b_1, \qquad a_1 = \mathrm{ReLU}(z_1), \\ z_2 &= w_2 a_1 + b_2, \qquad a_2 = \tanh(z_2), \\ z_3 &= w_3 a_2 + b_3, \qquad \hat{y} = \sigma(z_3), \qquad \mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2. \end{aligned}$$

3.9.2 Derivatives used (local)

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 1, & z > 0 \\ 0, & z \le 0 \end{cases}, \qquad \frac{d}{dz}\tanh(z) = 1 - \tanh^2(z), \qquad \sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big).$$

The loss derivative is

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y.$$

3.9.3 Numerical values

We choose:

$$\begin{gathered} x = 2, \quad y = 1, \\ (w_1, b_1) = (1.5, -1), \qquad (w_2, b_2) = (0.5, 0.1), \qquad (w_3, b_3) = (2, -0.3). \end{gathered}$$

3.9.4 Forward pass (all steps)

Table 3.1: Forward pass for a three-layer network: ReLU → tanh → sigmoid

Step	Quantity	Expression	Value
1	Input	x	2
2	Layer 1 pre-activation	$z_1 = w_1 x + b_1$	$1.5(2) + (-1) = 2.0$
3	Layer 1 activation (ReLU)	$a_1 = \mathrm{ReLU}(z_1)$	2.0
4	Layer 2 pre-activation	$z_2 = w_2 a_1 + b_2$	$0.5(2.0) + 0.1 = 1.1$
5	Layer 2 activation (tanh)	$a_2 = \tanh(z_2)$	0.800499
6	Layer 3 pre-activation	$z_3 = w_3 a_2 + b_3$	$2(0.800499) - 0.3 = 1.300998$
7	Output activation (sigmoid)	$\hat{y} = \sigma(z_3)$	0.786003
8	Target	y	1
9	Loss (MSE, single sample)	$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$	0.022897

3.9.5 Backward pass (explicit derivatives and gradients)

Step A: loss to output.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y = 0.786003 - 1 = -0.213997.$$

Step B: sigmoid derivative.

$$\frac{\partial \hat{y}}{\partial z_3} = \hat{y}(1-\hat{y}) = 0.786003(1-0.786003) = 0.168202.$$

Step C: delta at layer 3.

$$\delta_3 := \frac{\partial \mathcal{L}}{\partial z_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_3} = (-0.213997)(0.168202) = -0.035995.$$

Gradients for w_3, b_3. Since $z_3 = w_3 a_2 + b_3$:

$$\frac{\partial \mathcal{L}}{\partial w_3} = \delta_3 \frac{\partial z_3}{\partial w_3} = \delta_3 a_2 = (-0.035995)(0.800499) = -0.028814,$$

$$\frac{\partial \mathcal{L}}{\partial b_3} = \delta_3 \frac{\partial z_3}{\partial b_3} = \delta_3 = -0.035995.$$

Step D: propagate to a_2.

$$\frac{\partial \mathcal{L}}{\partial a_2} = \delta_3 \frac{\partial z_3}{\partial a_2} = \delta_3 w_3 = (-0.035995)(2) = -0.071990.$$

Step E: tanh derivative.

$$\frac{\partial a_2}{\partial z_2} = 1 - a_2^2 = 1 - (0.800499)^2 = 0.359201.$$

Step F: delta at layer 2.

$$\delta_2 := \frac{\partial \mathcal{L}}{\partial z_2} = \frac{\partial \mathcal{L}}{\partial a_2}\frac{\partial a_2}{\partial z_2} = (-0.071990)(0.359201) = -0.025859.$$

Gradients for w_2, b_2. Since $z_2 = w_2 a_1 + b_2$:

$$\frac{\partial \mathcal{L}}{\partial w_2} = \delta_2 \frac{\partial z_2}{\partial w_2} = \delta_2 a_1 = (-0.025859)(2.0) = -0.051718,$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \delta_2 = -0.025859.$$

Step G: propagate to a_1.

$$\frac{\partial \mathcal{L}}{\partial a_1} = \delta_2 \frac{\partial z_2}{\partial a_1} = \delta_2 w_2 = (-0.025859)(0.5) = -0.012929.$$

Step H: ReLU derivative. Here $z_1 = 2.0 > 0$, so $\mathrm{ReLU}'(z_1) = 1$.

Step I: delta at layer 1.

$$\delta_1 := \frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial a_1}\mathrm{ReLU}'(z_1) = (-0.012929)(1) = -0.012929.$$

Gradients for w_1, b_1. Since $z_1 = w_1 x + b_1$:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \delta_1 \frac{\partial z_1}{\partial w_1} = \delta_1 x = (-0.012929)(2) = -0.025859,$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \delta_1 = -0.012929.$$

3.9.6 Gradient summary

Collecting all gradients:

$$\frac{\partial \mathcal{L}}{\partial w_1} = -0.025859, \quad \frac{\partial \mathcal{L}}{\partial b_1} = -0.012929,$$
$$\frac{\partial \mathcal{L}}{\partial w_2} = -0.051718, \quad \frac{\partial \mathcal{L}}{\partial b_2} = -0.025859,$$
$$\frac{\partial \mathcal{L}}{\partial w_3} = -0.028814, \quad \frac{\partial \mathcal{L}}{\partial b_3} = -0.035995.$$

3.9.7 Gradient descent update (one step)

Let $\eta > 0$ be the learning rate. Gradient descent updates each parameter by

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}.$$

Therefore,

$$w_1^+ = w_1 - \eta \frac{\partial \mathcal{L}}{\partial w_1} = 1.5 - \eta(-0.025859) = 1.5 + 0.025859\,\eta, \quad b_1^+ = b_1 - \eta \frac{\partial \mathcal{L}}{\partial b_1} = -1 - \eta(-0.012929) = -1 + 0.01292$$
$$w_2^+ = 0.5 - \eta(-0.051718) = 0.5 + 0.051718\,\eta, \quad b_2^+ = 0.1 - \eta(-0.025859) = 0.1 + 0.025859\,\eta,$$
$$w_3^+ = 2 - \eta(-0.028814) = 2 + 0.028814\,\eta, \quad b_3^+ = -0.3 - \eta(-0.035995) = -0.3 + 0.035995\,\eta.$$

Numerical update example. For $\eta = 0.1$:

$$w_1^+ = 1.502586, \quad b_1^+ = -0.998707, \quad w_2^+ = 0.505172, \quad b_2^+ = 0.102586, \quad w_3^+ = 2.002881, \quad b_3^+ = -0.296401.$$

Interpretation. All gradients above are negative, so each parameter increases slightly under the update rule $\theta \leftarrow \theta - \eta\,\partial\mathcal{L}/\partial\theta$. This is consistent with reducing the error $\hat{y} - y < 0$ for this sample.

3.10 Jacobian, Delta, and Parameter Gradients in Backpropagation

Yes — the summary is **conceptually correct**. We restate it concisely below, tightening the language and highlighting one subtle but important point to establish a clear and robust mental model.

3.10.1 Jacobian

Jacobian denotes the derivative of a layer's output with respect to its input. For a layer defined as

$$a_\ell = f_\ell(a_{\ell-1}),$$

the Jacobian is

$$J_\ell = \frac{\partial a_\ell}{\partial a_{\ell-1}}.$$

The Jacobian:

- describes the **local transformation** performed by the layer,
- does **not** depend on the loss function,
- is a property of the layer itself (weights and activation).

In practice, full Jacobian matrices are never formed explicitly; instead, backpropagation relies on efficient **vector–Jacobian products**.

3.10.2 Delta

Delta is the gradient of the final loss with respect to the pre-activation (latent variable) of a layer. Formally,

$$\delta_\ell = \frac{\partial \mathcal{L}}{\partial z_\ell}.$$

Delta:

- **depends on the final loss**,
- is the quantity that propagates backward through the network,
- lives in the same space as the latent variable z_ℓ.

Thus, describing delta as the "gradient of the final loss with respect to the latent or hidden representation" is exactly correct.

3.10.3 Gradients of Weights and Biases

Gradients of the parameters are always gradients of the **final loss**. Specifically,

$$\frac{\partial \mathcal{L}}{\partial W_\ell}, \quad \frac{\partial \mathcal{L}}{\partial b_\ell}.$$

These gradients are computed directly from delta:

$$\frac{\partial \mathcal{L}}{\partial W_\ell} = \delta_\ell \, a_{\ell-1}^\top, \qquad \frac{\partial \mathcal{L}}{\partial b_\ell} = \delta_\ell.$$

Weights and biases never possess local gradients independent of the loss.

3.10.4 Conceptual Hierarchy

Object	Interpretation
Jacobian J_ℓ	Local sensitivity of layer output to its input
Delta δ_ℓ	Gradient of the final loss w.r.t. layer pre-activation
$\nabla_{W_\ell}\mathcal{L}$	Gradient of the final loss w.r.t. weights
$\nabla_{b_\ell}\mathcal{L}$	Gradient of the final loss w.r.t. bias

$$
\begin{aligned}
z_\ell &= W_\ell a_{\ell-1} + b_\ell && \text{(pre-activation)} \\
a_\ell &= \phi_\ell(z_\ell) && \text{(post-activation)} \\
J_\ell &= \frac{\partial a_\ell}{\partial a_{\ell-1}} && \text{(layer Jacobian)} \\
\delta_L &= \frac{\partial \mathcal{L}}{\partial z_L} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_L} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \, \phi_L'(z_L) && \text{(delta; output-layer pre-activation gradient, chain rule)} \\
\delta_\ell &= \frac{\partial \mathcal{L}}{\partial z_\ell} = \left(W_{\ell+1}^\top \delta_{\ell+1}\right) \odot \phi_\ell'(z_\ell), \quad \ell < L && \text{(delta; hidden-layer pre-activation gradient)} \\
\nabla_{W_\ell}\mathcal{L} &= \frac{\partial \mathcal{L}}{\partial W_\ell} = \delta_\ell \, a_{\ell-1}^\top && \text{(weight gradient)} \\
\nabla_{b_\ell}\mathcal{L} &= \frac{\partial \mathcal{L}}{\partial b_\ell} = \delta_\ell && \text{(bias gradient)}
\end{aligned}
$$

3.10.5 A Subtle but Important Clarification

Referring to the Jacobian as the "gradient of a layer's output with respect to its input" is correct, provided one remembers that:

- the Jacobian is generally **matrix-valued**,
- a gradient is a special case corresponding to scalar output.

Thus, gradients are a subset of Jacobians, and in deep networks Jacobians are almost always handled implicitly.

3.10.6 Mental Model

> **Jacobians describe local layer physics; deltas carry global error; parameter gradients convert error into learning.**

3.10.7 Summary

- Jacobians represent local layer derivatives,
- deltas represent gradients of the final loss with respect to latent variables,
- weight and bias gradients are always gradients of the final loss,
- backpropagation consists of chaining Jacobians with deltas.

This structure applies uniformly across multilayer perceptrons, convolutional networks, transformers, and automatic differentiation systems.

3.10.8 Chronological computation table (forward → backward → update, with Jacobians)

3.11 Worked Micro-Example: The First Backprop Signal (dv_7) Comes From the Loss

This section emphasizes a simple but universal point: *the backward pass must start from the derivative of the chosen loss with respect to the final node.* In other words, the **first upstream signal** is determined entirely by the loss function.

3.11.1 Forward computation (scalar graph)

Consider the scalar computation

$$v_1 = x_1, \qquad v_2 = x_2, \qquad v_3 = \exp(v_2), \qquad v_4 = v_1 v_3, \qquad v_5 = v_1 v_2, \qquad v_6 = \cos(v_5), \qquad v_7 = v_4 + v_6.$$

Thus the final output is

$$v_7(x_1, x_2) = x_1 e^{x_2} + \cos(x_1 x_2).$$

3.11.2 The universal first step in backprop

Backpropagation begins at the output node by differentiating the loss $\mathcal{L}$ with respect to v_7:

$$\boxed{dv_7 \;=\; \frac{\partial \mathcal{L}}{\partial v_7}}$$

This quantity dv_7 is the **initial upstream gradient** that seeds the entire backward pass. Once dv_7 is known, all other local derivatives propagate backward through the same chain rule mechanics.

Table 3.2: Chronological calculations with explicit Jacobians J_1, J_2, J_3

Step	Quantity	Formula	Value	Jacobian role
Forward pass (explicitly computed)				
1	Input	x	2	—
2	Pre-act (L1)	$z_1 = w_1 x + b_1$	2.0	—
3	Act (L1, ReLU)	$a_1 = \mathrm{ReLU}(z_1)$	2.0	—
4	Pre-act (L2)	$z_2 = w_2 a_1 + b_2$	1.1	—
5	Act (L2, tanh)	$a_2 = \tanh(z_2)$	0.800499	—
6	Pre-act (L3)	$z_3 = w_3 a_2 + b_3$	1.300998	—
7	Output	$\hat{y} = \sigma(z_3)$	0.786003	—
8	Loss	$\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$	0.022897	—
Local derivatives (explicitly computed)				
9	Loss deriv.	$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y$	-0.213997	scalar loss derivative
10	Sigmoid deriv.	$\sigma'(z_3) = \hat{y}(1 - \hat{y})$	0.168202	local nonlinearity
11	tanh deriv.	$\tanh'(z_2) = 1 - a_2^2$	0.359201	local nonlinearity
12	ReLU deriv.	$\mathrm{ReLU}'(z_1) = \mathbb{1}[z_1 > 0]$	1	local nonlinearity
Backward pass (Jacobian–vector products)				
13	Delta (L3)	$\delta_3 = \frac{\partial \mathcal{L}}{\partial z_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \sigma'(z_3)$	-0.035995	chain rule
14	J_3	$J_3 = \frac{\partial \hat{y}}{\partial a_2} = \sigma'(z_3) w_3$	0.336404	conceptual Jacobian
15	Grad w_3	$\frac{\partial \mathcal{L}}{\partial w_3} = \delta_3 a_2$	-0.028814	via J_3
16	Grad b_3	$\frac{\partial \mathcal{L}}{\partial b_3} = \delta_3$	-0.035995	via J_3
17	Prop to a_2	$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} J_3$	-0.071990	vector–Jacobian product
18	Delta (L2)	$\delta_2 = \frac{\partial \mathcal{L}}{\partial z_2} = \frac{\partial \mathcal{L}}{\partial a_2} \tanh'(z_2)$	-0.025859	chain rule
19	J_2	$J_2 = \frac{\partial a_2}{\partial a_1} = (1 - a_2^2) w_2$	0.179600	conceptual Jacobian
20	Grad w_2	$\frac{\partial \mathcal{L}}{\partial w_2} = \delta_2 a_1$	-0.051718	via J_2
21	Grad b_2	$\frac{\partial \mathcal{L}}{\partial b_2} = \delta_2$	-0.025859	via J_2
22	Prop to a_1	$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} J_2$	-0.012929	vector–Jacobian product
23	Delta (L1)	$\delta_1 = \frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial a_1} \mathrm{ReLU}'(z_1)$	-0.012929	chain rule
24	J_1	$J_1 = \frac{\partial a_1}{\partial x} = \mathrm{ReLU}'(z_1) w_1$	1.5	conceptual Jacobian
25	Grad w_1	$\frac{\partial \mathcal{L}}{\partial w_1} = \delta_1 x$	-0.025859	via J_1
26	Grad b_1	$\frac{\partial \mathcal{L}}{\partial b_1} = \delta_1$	-0.012929	via J_1
Gradient descent update (explicitly applied)				
27	Update rule	$\theta^+ = \theta - \eta\, \partial \mathcal{L} / \partial \theta$	—	applied to all parameters
28	w_1^+	$1.5 - \eta(-0.025859)$	1.502586 ($\eta = 0.1$)	—
29	b_1^+	$-1 - \eta(-0.012929)$	-0.998707 ($\eta = 0.1$)	—
30	w_2^+	$0.5 - \eta(-0.051718)$	0.505172 ($\eta = 0.1$)	—
31	b_2^+	$0.1 - \eta(-0.025859)$	0.102586 ($\eta = 0.1$)	—
32	w_3^+	$2 - \eta(-0.028814)$	2.002881 ($\eta = 0.1$)	—
33	b_3^+	$-0.3 - \eta(-0.035995)$	-0.296401 ($\eta = 0.1$)	—

Color convention: Blue quantities are explicitly computed during forward or backward propagation. Red quantities denote conceptual Jacobians shown for explanation only and are never formed explicitly.

3.11.3 Why sometimes $dv_7 = 1$

If we choose the objective to be the output itself,

$$\mathcal{L}(x_1, x_2) = v_7(x_1, x_2),$$

then

$$dv_7 = \frac{\partial \mathcal{L}}{\partial v_7} = \frac{\partial v_7}{\partial v_7} = 1.$$

This is the common "minimize the output" setup, and it corresponds to writing in code:

```
dv7 = 1.0.
```

3.11.4 Target-matching: making v_7 approach a desired value

Often we do not want to minimize v_7 directly. Instead, we want v_7 to match a target value t. A standard choice is the squared error loss:

$$\mathcal{L}(x_1, x_2) = \frac{1}{2}\big(v_7(x_1, x_2) - t\big)^2.$$

Differentiating with respect to v_7 gives

$$dv_7 = \frac{\partial \mathcal{L}}{\partial v_7} = \frac{\partial}{\partial v_7}\left[\frac{1}{2}(v_7 - t)^2\right] = v_7 - t.$$

So the only change needed in the backward initialization is:

```
dv7 = v7 - t.
```

Two important special cases are:

$$t = 0 \;\Rightarrow\; dv_7 = v_7, \qquad t = 1 \;\Rightarrow\; dv_7 = v_7 - 1.$$

3.11.5 Different losses give different dv_7 (only the first line changes)

The computational graph (the equations defining $v_1, \ldots, v_7$) does not change. What changes is *only* the loss derivative at the output node. Table 3.3 lists common choices.

Table 3.3: Examples of loss functions and the corresponding initial backprop signal $dv_7 = \partial \mathcal{L}/\partial v_7$.

Goal	**Loss $\mathcal{L}(v_7)$**	$dv_7 = \partial \mathcal{L}/\partial v_7$
Minimize v_7	$\mathcal{L} = v_7$	$dv_7 = 1$
Match target t	$\mathcal{L} = \frac{1}{2}(v_7 - t)^2$	$dv_7 = v_7 - t$
Robust (L1) match	$\mathcal{L} = \lvert v_7 - t \rvert$	$dv_7 = \operatorname{sign}(v_7 - t)$
Soft (log-squared) match	$\mathcal{L} = \log\big(1 + (v_7 - t)^2\big)$	$dv_7 = \dfrac{2(v_7 - t)}{1 + (v_7 - t)^2}$

Interpretation.

- The backward pass always starts by computing $dv_7 = \partial \mathcal{L}/\partial v_7$.
- For $\mathcal{L} = v_7$, the seed gradient is constant: $dv_7 = 1$.
- For target matching $\mathcal{L} = \frac{1}{2}(v_7 - t)^2$, the seed gradient becomes the error: $dv_7 = v_7 - t$.
- After choosing dv_7, the remaining backward pass uses the same chain rule steps through $v_6, v_5, v_4, v_3, v_2, v_1$.

3.12 Parameter Shapes and Gradient Shapes

A frequent source of confusion in backpropagation is why a weight vector sometimes produces a vector gradient, while in other cases the gradient appears as a matrix. The resolution lies entirely in the dimensionality of the parameter itself.

The fundamental rule is:

The gradient of a parameter always has the same shape as the parameter.

This section clarifies this principle by contrasting single-neuron and multi-neuron (layer-wise) parameterizations.

3.12.1 Single Neuron Case

Consider a single neuron with input $a \in \mathbb{R}^n$ and parameters $w \in \mathbb{R}^n$, $b \in \mathbb{R}$:

$$z = w^\top a + b.$$

The loss $\mathcal{L}$ depends on z, and the gradient with respect to the weights is

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w} = \delta\, a, \qquad \delta := \frac{\partial \mathcal{L}}{\partial z}.$$

Thus, the gradient $\partial\mathcal{L}/\partial w$ is a vector with the same shape as w.

3.12.2 Layer Case (Multiple Neurons)

Now consider a fully connected layer with m neurons:

$$z = Wa + b,$$

where

$$W \in \mathbb{R}^{m\times n}, \qquad a \in \mathbb{R}^n, \qquad b \in \mathbb{R}^m, \qquad z \in \mathbb{R}^m.$$

Each row of W corresponds to the weight vector of one neuron. The loss gradient with respect to W is obtained by stacking the gradients of all neurons:

$$\frac{\partial \mathcal{L}}{\partial W} = \begin{pmatrix} \delta_1 a^\top \\ \delta_2 a^\top \\ \vdots \\ \delta_m a^\top \end{pmatrix} = \delta\, a^\top, \qquad \delta := \frac{\partial \mathcal{L}}{\partial z} \in \mathbb{R}^m.$$

This expression is an outer product between the delta vector and the input activation, producing a matrix with the same shape as W.

3.12.3 Summary Table: Parameters and Their Gradients

3.12.4 Operational Summary

For a linear layer $z = Wa + b$, the core backpropagation operations are:

$$\boxed{\begin{aligned} z &= Wa + b, \\ \delta &= \frac{\partial \mathcal{L}}{\partial z}, \\ \nabla_W \mathcal{L} &= \delta\, a^\top, \\ \nabla_b \mathcal{L} &= \delta, \\ \frac{\partial \mathcal{L}}{\partial a} &= W^\top \delta. \end{aligned}}$$

Table 3.4: Relationship between parameters and gradient shapes

Scenario	**Parameter**	**Shape**	**Gradient**	**Gradient Shape**
Single neuron	w	$(n,)$	$\nabla_w \mathcal{L}$	$(n,)$
Single neuron	b	scalar	$\partial \mathcal{L}/\partial b$	scalar
Layer (m neurons)	W	(m, n)	$\nabla_W \mathcal{L}$	(m, n)
Layer (m neurons)	b	$(m,)$	$\nabla_b \mathcal{L}$	$(m,)$
Activation	a	$(n,)$	$\partial \mathcal{L}/\partial a$	$(n,)$
Pre-activation	z	$(m,)$	$\delta = \partial \mathcal{L}/\partial z$	$(m,)$

The outer product $\delta\, a^\top$ appears because each weight connects one input component to one output component, requiring a distinct gradient for every pair.

Key takeaway. A gradient is never larger or smaller than the object it differentiates. Its shape always mirrors the shape of the parameter itself.

3.12.5 Reference Python Implementation

Table 3.5: Distinction between local derivatives, deltas, and gradients in backpropagation

Quantity	**Mathematical form**	**Name**	**Role**
Layer output sensitivity	$\delta_z = \dfrac{\partial \mathcal{L}}{\partial z}$	**Delta**	Backward-propagated error signal
Local layer derivative	$\dfrac{\partial z}{\partial W}$	Local derivative / local Jacobian	Layer-specific sensitivity (loss-independent)
Parameter gradient	$\dfrac{\partial \mathcal{L}}{\partial W}$	**Gradient**	Quantity used for parameter updates
Linear layer relation	$\dfrac{\partial \mathcal{L}}{\partial W} = \dfrac{\partial \mathcal{L}}{\partial z}\dfrac{\partial z}{\partial W}$	Chain rule	Connects delta to gradient

3.13 Linear Algebra Operations in Forward and Backward Propagation

3.14 Commutativity of Linear Algebra Operations

Not all linear algebra operations used in forward and backward propagation are commutative. This section clarifies which operations commute, which do not, and how this impacts backpropagation formulas.

Key identities. For vectors $a, b \in \mathbb{R}^n$,

$$a \mathbin{@} b = a^\top b = b^\top a = b \mathbin{@} a,$$

so the dot product is commutative.

Table 3.6: Commutativity properties of common linear algebra operations

Operation	**Inputs**	**Expression**	**Commutative?**	**Notes**
Dot product	vector, vector	$a \mathbin{@} b$	Yes	$a \mathbin{@} b = b \mathbin{@} a$
Outer product	vector, vector	$\text{outer}(a, b)$	No	$\text{outer}(b, a) = \text{outer}(a, b)^\top$
Matrix–vector	matrix, vector	$W \mathbin{@} a$	No	Order defines the linear map
Matrix–matrix	matrix, matrix	$A \mathbin{@} B$	No	In general $AB \neq BA$

In contrast, the outer product satisfies

$$\text{outer}(a, b) = ab^\top, \qquad \text{outer}(b, a) = ba^\top,$$

and therefore

$$\text{outer}(b, a) = \text{outer}(a, b)^\top,$$

which shows that outer products are *not* commutative.

Implication for backpropagation.

- Forward propagation uses matrix–vector products ($W@a$).
- Backward signal propagation uses transpose maps ($W^\top @ dz$).
- Weight gradients use outer products ($dW = dz\, a^\top$).

The lack of commutativity explains why the ordering of operands in backpropagation formulas is essential and cannot be rearranged arbitrarily.

Summary.

Dot products commute. Outer products transpose. Matrix multiplication does not commute.

Table 3.7: Linear algebra operations used in forward and backward propagation

Context	**Inputs**	**Operation**	**Result**	**Code**
Forward pass	matrix, vector	Linear map	vector	`z = W @ a`
Backward pass (input gradient)	matrix, vector	Transpose linear map	vector	`da = W.T @ dz`
Backward pass (weight gradient)	vector, vector	Outer product	matrix	`dW = np.outer(dz, a)`
Dot product	vector, vector	Inner product (summation)	scalar	`u @ v`
Outer product	vector, vector	No summation	matrix	`np.outer(u, v)`
Matrix product	matrix, matrix	Matrix multiplication	matrix	`A @ B`

The `@` operator propagates signals through summation, whereas the outer product constructs parameter gradients without summation.

For completeness and reproducibility, we list the core Python functions used in all numerical experiments. These functions implement the forward computation, the backward pass via the chain rule, and a single gradient descent update. Only the initialization of the upstream gradient dv_7 differs across loss functions.

Objective function. The scalar output of the computational graph is

$$v_7(x_1, x_2) = x_1 e^{x_2} + \cos(x_1 x_2).$$

```
import math

def f(x1, x2):
    return x1 * math.exp(x2) + math.cos(x1 * x2)
```

Direct minimization of v_7 ($dv_7 = 1$). This function corresponds to the loss $\mathcal{L} = v_7$.

```
def compute_deltas(x1, x2, lr=1e-2, prnt=False, prnt_delta=False):
    # ----- forward pass -----
    v1 = x1
    v2 = x2
    v3 = math.exp(v2)
    v4 = v1 * v3
    v5 = v1 * v2
    v6 = math.cos(v5)
    v7 = v4 + v6
    f_old = v7

    # ----- backward pass -----
    dv7 = 1.0

    dv4 = dv7
    dv6 = dv7
    dv5 = dv6 * (-math.sin(v5))

    dv1 = dv4 * v3 + dv5 * v2
    dv2 = dv5 * v1 + (dv4 * v1) * math.exp(v2)

    # ----- gradient descent update -----
    x1_new = x1 - lr * dv1
    x2_new = x2 - lr * dv2

    if prnt_delta:
        f_new = f(x1_new, x2_new)
        print(f"delta_f␣=␣{f_new␣-␣f_old}")
        print(f"f_new={f_new},␣f_old={f_old}")
        print("-" * 25)

    return x1_new, x2_new
```

Highlighted line. The blue assignment `dv7 = 1.0` encodes the loss $\mathcal{L} = v_7$. Changing the loss function modifies only this line via $dv_7 = \partial\mathcal{L}/\partial v_7$.

Targeted optimization ($v_7 \to t$). This function minimizes the squared loss

$$\mathcal{L} = \tfrac{1}{2}(v_7 - t)^2, \qquad dv_7 = v_7 - t.$$

```
def compute_deltas_make_v7_zero(
    x1, x2, v_target=0, lr=1e-3,
    prnt=False, prnt_delta=False
):
    # ----- forward pass -----
```

```
    v1 = x1
    v2 = x2
    v3 = math.exp(v2)
    v4 = v1 * v3
    v5 = v1 * v2
    v6 = math.cos(v5)
    v7 = v4 + v6

    # ----- loss and initial gradient -----
    dv7 = v7 - v_target
    L_old = 0.5 * (v7 - v_target)**2


    # ----- backward pass -----
    dv4 = dv7
    dv6 = dv7
    dv5 = dv6 * (-math.sin(v5))

    dv1 = dv4 * v3 + dv5 * v2
    dv2 = dv5 * v1 + (dv4 * v1) * math.exp(v2)

    # ----- gradient descent update -----
    x1_new = x1 - lr * dv1
    x2_new = x2 - lr * dv2

    if prnt_delta:
        v7_new = f(x1_new, x2_new)
        L_new = 0.5 * (v7_new - v_target)**2
        print(f"v7_old={v7}, v7_new={v7_new}")
        print(f"L_old={L_old}, L_new={L_new}, delta_L={L_new - L_old}")
        print("-" * 25)

    return x1_new, x2_new
```

Highlighted line. The blue assignment `dv7 = v7 - v_target` corresponds to the squared loss $\mathcal{L} = \frac{1}{2}(v_7 - v_{\text{target}})^2$. The target value enters the optimization exclusively through this initial backpropagation signal.

Remark. The forward graph and all chain-rule computations are identical in both functions. The optimization objective enters the algorithm exclusively through the choice of

$$dv_7 = \frac{\partial \mathcal{L}}{\partial v_7}.$$

3.15 Numerical Experiments: Effect of the Initial Backprop Signal dv_7

This section presents concrete numerical experiments using the scalar computational graph

$$v_7(x_1, x_2) = x_1 e^{x_2} + \cos(x_1 x_2),$$

and illustrates how *changing only the initial backprop signal*

$$dv_7 = \frac{\partial \mathcal{L}}{\partial v_7}$$

alters the qualitative behavior of gradient descent.

All experiments were executed using the Python implementation shown below :contentReference[oaicite:0]index=0.

3.15.1 Direct minimization of v_7 ($dv_7 = 1$)

We first minimize the output directly by choosing

$$\mathcal{L} = v_7, \qquad dv_7 = 1.$$

Code.

```
x1, x2 = 0.6, 0.6
for i in range(10):
    x1, x2 = compute_deltas(
        x1, x2,
        lr=1e-2,
        prnt=False,
        prnt_delta=True
    )
print(f"x1={x1}, x2={x2}")
```

Observed behavior.

- The function value v_7 decreases monotonically.
- The parameters drift toward a region that reduces $x_1 e^{x_2}$.
- No target value is enforced; the algorithm only seeks smaller output values.

A representative outcome after a few steps is

$$x_1 \approx 0.441, \qquad x_2 \approx 0.523,$$

with v_7 steadily decreasing.

3.15.2 Forcing $v_7 \to 0$ (squared loss, $dv_7 = v_7$)

We now change only the loss:

$$\mathcal{L} = \tfrac{1}{2} v_7^2, \qquad dv_7 = v_7.$$

Code.

```
x1, x2 = 0.5, 0.5
for i in range(5000):
    x1, x2 = compute_deltas_make_v7_zero(
        x1, x2,
        v_target=0,
        lr=1e-3,
        prnt=False,
        prnt_delta=False
    )
print(f"x1={x1}, x2={x2}")
```

Observed behavior.

- The loss $\frac{1}{2} v_7^2$ decreases toward zero.
- The parameters converge to a finite solution.
- The gradient magnitude vanishes as $v_7 \to 0$.

Typical convergence:

$$x_1 \approx -0.555, \qquad x_2 \approx 0.543, \qquad v_7 \approx 0.$$

3.15.3 Forcing $v_7 \to 1$ (shifted squared loss, $dv_7 = v_7 - 1$)

Next, we enforce a nonzero target by choosing

$$\mathcal{L} = \tfrac{1}{2}(v_7 - 1)^2, \qquad dv_7 = v_7 - 1.$$

Code.

```
x1, x2 = 1.5, 1.5
for i in range(5000):
    x1, x2 = compute_deltas_make_v7_zero(
        x1, x2,
        v_target=1,
        lr=1e-3,
        prnt=False,
        prnt_delta=False
    )
print(f"x1={x1}, x2={x2}")
print(f(x1, x2))
```

Observed behavior.

- The parameters adjust until $v_7 \approx 1$.
- Multiple parameter configurations can satisfy the target.
- Large parameter magnitudes may appear if the solution manifold is poorly conditioned.

  One observed outcome:

$$x_1 \approx -42.54, \qquad x_2 \approx -82.86, \qquad v_7 \approx 0.99999.$$

3.15.4 Interpretation

These experiments demonstrate that:

- The forward computational graph is unchanged.
- The backward mechanics are unchanged.
- *Only the initial gradient*

$$dv_7 = \frac{\partial \mathcal{L}}{\partial v_7}$$

  determines the optimization objective.

Thus:

$$\begin{aligned} \mathcal{L} = v_7 &\;\Rightarrow\; dv_7 = 1, \\ \mathcal{L} = \tfrac{1}{2}v_7^2 &\;\Rightarrow\; dv_7 = v_7, \\ \mathcal{L} = \tfrac{1}{2}(v_7 - 1)^2 &\;\Rightarrow\; dv_7 = v_7 - 1. \end{aligned}$$

Key takeaway.

> Backpropagation does not encode the goal. The loss function does. The goal enters the algorithm *only* through dv_7.

Chapter 4

The Curse of Dimensionality and Fixed Basis Functions

4.1 Polynomial Basis Functions and Parameter Growth

Consider a polynomial model of order M in D input dimensions. The number of coefficients required grows on the order of

$$\mathcal{O}(D^M). \tag{4.1}$$

This growth is combinatorial and quickly becomes infeasible as the input dimension increases.

Example

Dimension D	Order M	Approx. Parameters
5	3	$\sim 10^2$
50	3	$\sim 10^5$
100	3	$\sim 10^6$
1024 (image)	3	$\sim 10^9$

Table 4.1: Growth of polynomial coefficients with dimensionality

Even for moderate values of D, the number of parameters exceeds what can be reliably estimated from data. This phenomenon is known as the *curse of dimensionality*.

4.2 Grid-Based Basis Functions

An alternative approach is to partition the input space into a regular grid. If each dimension is divided into K intervals, the total number of grid cells is

$$\mathcal{O}(K^D). \tag{4.2}$$

Each grid cell can be interpreted as defining a separate basis function.

Example

Consequences

- Data become sparse, with most grid cells empty.

Dimensions D	Bins per Dimension K	Grid Cells
2	10	10^2
5	10	10^5
10	10	10^{10}

Table 4.2: Exponential growth of grid cells

- Predictions rely on very few samples.
- Distance-based intuition becomes unreliable.
- Coverage of the input space requires exponentially many basis functions.

4.3 High-Dimensional Geometry

Geometric intuition developed in low-dimensional spaces fails in high dimensions.

4.3.1 Volume Concentration

The volume of a D-dimensional hypersphere of radius r is given by

$$V_D(r) = K_D r^D, \tag{4.3}$$

where K_D depends only on the dimensionality D.

Consider a unit hypersphere ($r = 1$). The fraction of volume lying between radii $1 - \varepsilon$ and 1 is

$$1 - (1 - \varepsilon)^D. \tag{4.4}$$

As D increases, this fraction rapidly approaches one, indicating that most of the volume concentrates near the boundary.

Implications

- Most points lie near the boundary of the space.
- Distances between points become increasingly similar.
- Euclidean distance loses discriminative power.

4.4 Data Manifolds

Although the ambient dimensionality of data may be large, real-world data typically lie on low-dimensional manifolds embedded within the high-dimensional space.

Examples

- Handwritten digits vary in stroke shape and orientation.
- Natural images are constrained by object geometry and lighting.
- Speech signals are governed by vocal tract dynamics.

4.4.1 Natural Images

A 64×64 grayscale image corresponds to a point in a 12,288-dimensional space. However, natural images occupy only a small fraction of this space due to strong correlations between neighboring pixels.

Image Type	Structure	Pixel Correlation
Natural images	Structured	High
Random noise	Unstructured	None

Table 4.3: Natural images versus random noise

4.5 Radial Basis Functions

Radial basis functions (RBFs) associate a basis function with each training sample:

$$\phi_n(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_n\|^2}{s^2}\right), \tag{4.5}$$

where s controls the width of the basis function.

Advantages

- Data-dependent basis functions
- Local similarity modeling

Limitations

- Poor scalability with dataset size
- Sensitivity to distance concentration
- High computational cost
- Careful tuning of s required

4.6 Motivation for Deep Learning

The primary limitations of fixed basis function methods are:

- Basis functions are chosen independently of the data.
- The number of required functions grows exponentially with dimension.
- High-dimensional geometry undermines low-dimensional intuition.

Deep neural networks address these challenges by learning:

- Data-dependent representations
- Hierarchical and compositional features
- Scalable models for high-dimensional inputs

Chapter Summary

Method	Scaling with D	High-D Feasible
Polynomial bases	$\mathcal{O}(D^M)$	No
Grid-based methods	$\mathcal{O}(K^D)$	No
RBF networks	$\mathcal{O}(N)$	Limited
Deep networks	Data-adaptive	Yes

Chapter 5

Gradients and Optimization Foundations

This chapter introduces the mathematical foundations underlying gradient-based learning in neural networks. We begin with differential operators and subgradients, then develop a geometric view of optimization through error surfaces, curvature, stochastic methods, and convergence behavior.

5.1 Gradient and Subgradient

Function

$$f : \mathbb{R}^n \to \mathbb{R}$$

Nabla (del) operator

$$\nabla \equiv \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_n} \right)$$

Gradient

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{bmatrix} \in \mathbb{R}^n$$

Subgradient A vector $g \in \mathbb{R}^n$ is a subgradient of f at x_0 if

$$f(x) \geq f(x_0) + g^\top (x - x_0), \quad \forall x \in \mathbb{R}^n$$

Meaning of g

$g \in \mathbb{R}^n$ is a fixed slope vector defining a supporting affine function

$$x \mapsto f(x_0) + g^\top (x - x_0)$$

g is constant with respect to x and is not a function of x

Subdifferential

$$\partial f(x_0) = \left\{ g \in \mathbb{R}^n : f(x) \geq f(x_0) + g^\top (x - x_0),\ \forall x \right\}$$

Smooth case If f is differentiable at x_0,

$$\partial f(x_0) = \{\nabla f(x_0)\}$$

Non-smooth case If f is not differentiable at x_0,

$$|\partial f(x_0)| > 1$$

One-dimensional case

$$\nabla f(x) = \frac{df}{dx}(x)$$

Optimization update

$$g_k \in \partial f(x_k)$$

$$x_{k+1} = x_k - \eta g_k$$

Key distinction

$$\nabla = \text{operator}, \quad \nabla f = \text{gradient vector (unique if exists)}, \quad \partial f = \text{subgradient set}$$

One-line interpretation

$$\nabla f(x_0) = g \text{ (smooth case)}, \qquad g \in \partial f(x_0) \text{ (non-smooth case)}$$

Example 1 (Quadratic, smooth)

$$f(x) = \tfrac{1}{2}\|x\|_2^2$$

$$\nabla f(x) = x, \qquad \partial f(x) = \{x\}$$

Example 2 (Absolute value, non-smooth)

$$f(x) = |x|$$

$$\partial f(x) = \begin{cases} \{1\}, & x > 0 \\ [-1, 1], & x = 0 \\ \{-1\}, & x < 0 \end{cases}$$

Example 3 (ReLU)

$$f(x) = \max(0, x)$$

$$\partial f(x) = \begin{cases} \{1\}, & x > 0 \\ [0, 1], & x = 0 \\ \{0\}, & x < 0 \end{cases}$$

Example 4 (Linear function)

$$f(x) = a^\top x + b$$

$$\nabla f(x) = a, \qquad \partial f(x) = \{a\}$$

5.2 Error Surfaces and Gradient-Based Optimization

5.2.1 Error Surfaces

Let

$$\mathbf{w} \in \mathbb{R}^W$$

denote the vector of all learnable parameters (weights and biases) of a neural network, and let

$$E(\mathbf{w}) : \mathbb{R}^W \to \mathbb{R}$$

denote the associated error (loss) function.

Geometrically, $E(\mathbf{w})$ defines a scalar-valued surface over the W-dimensional parameter space. Training a neural network can therefore be interpreted as navigating this surface in search of parameter values that yield small error.

5.2.2 Local Change in Error

Consider a small perturbation of the parameters

$$\mathbf{w} \to \mathbf{w} + \delta\mathbf{w}.$$

Using a first-order Taylor approximation, the change in the error function is

$$E(\mathbf{w} + \delta\mathbf{w}) \;\approx\; E(\mathbf{w}) + \nabla E(\mathbf{w})^\top \delta\mathbf{w}.$$

Thus, the incremental change in error is approximately

$$\delta E \;\approx\; \delta\mathbf{w}^\top \nabla E(\mathbf{w}),$$

where

$$\nabla E(\mathbf{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_W} \end{bmatrix}$$

is the gradient of the error function with respect to the parameters.

5.2.3 Geometric Meaning of the Gradient

The gradient vector $\nabla E(\mathbf{w})$ points in the direction of the steepest local increase of the error function. Formally,

$$\max_{\|\delta\mathbf{w}\|=1} \delta\mathbf{w}^\top \nabla E(\mathbf{w}) = \|\nabla E(\mathbf{w})\|.$$

Consequently, moving in the direction

$$-\nabla E(\mathbf{w})$$

produces the greatest local decrease in the error.

5.2.4 Gradient Descent Update

A gradient-based optimization step is therefore given by

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla E(\mathbf{w}_k),$$

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

Substituting into the first-order approximation yields

$$\delta E \approx -\eta \|\nabla E(\mathbf{w}_k)\|^2 \leq 0,$$

indicating a local decrease in the error.

5.2.5 Stationary Points

A necessary condition for a stationary point of the error surface is

$$\nabla E(\mathbf{w}) = \mathbf{0}.$$

At such points, no first-order change in the parameters produces a change in the error. Stationary points may correspond to **local minima, local maxima, or saddle points**.

Second-order characterization Let $\nabla^2 E(\mathbf{w})$ denote the Hessian matrix of second derivatives.

Table 5.1: Classification of stationary points using first- and second-order derivatives

Condition	Second-order behavior	Point type
$\nabla E(\mathbf{w}) = \mathbf{0}$	$\nabla^2 E(\mathbf{w}) \succ 0$	Local minimum
$\nabla E(\mathbf{w}) = \mathbf{0}$	$\nabla^2 E(\mathbf{w}) \prec 0$	Local maximum
$\nabla E(\mathbf{w}) = \mathbf{0}$	Indefinite $\nabla^2 E(\mathbf{w})$	Saddle point
$\nabla E(\mathbf{w}) = \mathbf{0}$	$\nabla^2 E(\mathbf{w}) = 0$ or singular	Higher-order analysis required

In high-dimensional neural network models, saddle points are particularly common.

5.2.6 Interpretation

Training a neural network consists of iteratively computing the gradient of the error surface and updating the parameters in a direction that locally reduces the error. Backpropagation provides an efficient mechanism for evaluating $\nabla E(\mathbf{w})$, enabling gradient-based optimization in high-dimensional parameter spaces.

5.2.7 Local Quadratic Approximation

Insight into the local geometry of the error surface can be obtained via a second-order Taylor expansion of the error function $E(\mathbf{w})$ about a reference point $\tilde{\mathbf{w}}$:

$$E(\mathbf{w}) \;\approx\; E(\tilde{\mathbf{w}}) + (\mathbf{w} - \tilde{\mathbf{w}})^\top \mathbf{b} + \frac{1}{2}(\mathbf{w} - \tilde{\mathbf{w}})^\top \mathbf{H}(\mathbf{w} - \tilde{\mathbf{w}}).$$

Here,

$$\mathbf{b} = \nabla E(\mathbf{w})\big|_{\mathbf{w}=\tilde{\mathbf{w}}}$$

is the gradient of the error function evaluated at $\tilde{\mathbf{w}}$, and

$$\mathbf{H} = \nabla^2 E(\mathbf{w})\big|_{\mathbf{w}=\tilde{\mathbf{w}}}$$

is the Hessian matrix of second-order partial derivatives.

Interpretation of the terms

- $E(\tilde{\mathbf{w}})$: error value at the reference point,
- $(\mathbf{w} - \tilde{\mathbf{w}})^\top \mathbf{b}$: first-order (linear) variation describing local slope,
- $\frac{1}{2}(\mathbf{w} - \tilde{\mathbf{w}})^\top \mathbf{H}(\mathbf{w} - \tilde{\mathbf{w}})$: second-order variation describing local curvature.

Stationary point If $\mathbf{w}^*$ is a stationary point of the error surface, then

$$\nabla E(\mathbf{w}^*) = \mathbf{0}.$$

In this case, the linear term vanishes and the approximation reduces to

$$E(\mathbf{w}) \approx E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*).$$

Second-order characterization The local behavior of the error surface near $\mathbf{w}^*$ is determined by the Hessian $\mathbf{H}$:

- $\mathbf{H} \succ 0$: local minimum,
- $\mathbf{H} \prec 0$: local maximum,
- $\mathbf{H}$ indefinite: saddle point.

Geometric view Near a stationary point, the error surface behaves locally as a quadratic form. The eigenvalues of the Hessian determine the curvature along different directions in parameter space, explaining the prevalence of saddle points in high-dimensional neural networks.

5.2.8 Eigenvalues, Curvature, and Stationary Points

Let $\mathbf{H} = \nabla^2 E(\mathbf{w}^*)$ denote the Hessian matrix of the error function evaluated at a stationary point $\mathbf{w}^*$. Let $\{\lambda_i, \mathbf{u}_i\}$ denote the eigenvalue–eigenvector pairs of $\mathbf{H}$, satisfying

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i.$$

Since the Hessian is symmetric, its eigenvectors form a complete orthonormal basis:

$$\mathbf{u}_i^\top \mathbf{u}_j = \delta_{ij},$$

where δ_{ij} is the Kronecker delta defined by

$$\delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Thus, $\|\mathbf{u}_i\| = 1$ and $\mathbf{u}_i \perp \mathbf{u}_j$ for $i \neq j$.

Expansion in the Eigenbasis

Any displacement from the stationary point can be expressed in the eigenbasis of the Hessian as

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{u}_i.$$

Substituting this expansion into the local quadratic approximation,

$$E(\mathbf{w}) \approx E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*),$$

yields

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2.$$

Each eigen-direction contributes independently to the change in error.

Classification of Stationary Points

From the quadratic form

$$E(\mathbf{w}) - E(\mathbf{w}^*) = \frac{1}{2} \sum_i \lambda_i \alpha_i^2,$$

the nature of the stationary point follows directly:

- If $\lambda_i > 0$ for all i, the point is a local minimum.
- If $\lambda_i < 0$ for all i, the point is a local maximum.
- If the eigenvalues have mixed signs, the point is a saddle point.

Positive Definiteness

The Hessian $\mathbf{H}$ is positive definite if

$$\mathbf{v}^\top \mathbf{H} \mathbf{v} > 0 \quad \forall \mathbf{v} \neq \mathbf{0},$$

which is equivalent to

$$\lambda_i > 0 \quad \forall i.$$

Positive definiteness guarantees that $\mathbf{w}^*$ is a strict local minimum.

Geometric Interpretation

Contours of constant error near $\mathbf{w}^*$ satisfy

$$\sum_i \lambda_i \alpha_i^2 = \text{const.}$$

These contours form ellipses (or ellipsoids in higher dimensions) whose principal axes are aligned with the eigenvectors $\mathbf{u}_i$. The lengths of the axes are proportional to $\lambda_i^{-1/2}$, indicating steep directions for large eigenvalues and flat directions for small eigenvalues.

This local geometry explains the behavior of gradient-based optimization methods near stationary points.

5.2.9 Stochastic and Mini-Batch Gradient Descent

Assume the error function decomposes over individual data points:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}),$$

where $E_n(\mathbf{w})$ denotes the loss associated with the n-th sample.

The full gradient is

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} \nabla E_n(\mathbf{w}),$$

which becomes computationally expensive for large N.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent updates the parameters using a single randomly selected data point:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla E_n(\mathbf{w}^{(t)}),$$

where n is sampled uniformly from $\{1, \ldots, N\}$.

The stochastic gradient satisfies

$$\mathbb{E}[\nabla E_n(\mathbf{w})] = \nabla E(\mathbf{w}),$$

so the update direction is an unbiased but noisy estimate of the true gradient.

The noise introduced by SGD can help escape shallow local minima and saddle points.

Mini-Batch Gradient Descent

Mini-batch gradient descent uses a subset $B \subset \{1, \ldots, N\}$ of size $|B|$:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \sum_{n \in B} \nabla E_n(\mathbf{w}^{(t)}).$$

This approach balances computational efficiency and variance reduction by averaging gradients over multiple samples.

Gradient Noise Scaling

Assuming the per-sample gradient has standard deviation σ, the standard deviation of the mini-batch gradient estimator scales as

$$\frac{\sigma}{\sqrt{|B|}}.$$

Thus, increasing the batch size yields diminishing returns: increasing $|B|$ by a factor of 100 reduces the gradient noise by only a factor of 10.

Shuffling and Randomness

Mini-batches should be formed by random sampling. Datasets are typically shuffled between epochs to avoid correlations arising from ordered data and to ensure that each mini-batch is a representative sample of the full dataset.

Terminology

Even when mini-batches are used, the optimization method is commonly referred to as stochastic gradient descent due to the inherent randomness in gradient estimation.

Table 5.2: Comparison of batch, stochastic, and mini-batch gradient descent

Method	**Batch size**	**Gradient noise**	**Computation per step**
Batch Gradient Descent	N	None	High
Stochastic Gradient Descent	1	High	Low
Mini-Batch Gradient Descent	$\|B\| \ll N$	Moderate	Moderate

Practical considerations Mini-batch sizes are often chosen to maximize hardware efficiency, particularly on GPUs. Powers of two such as $|B| \in \{32, 64, 128, 256\}$ are commonly used in practice.

Notation The superscript (t) denotes the iteration index of the optimization algorithm. For example, $\mathbf{w}^{(t)}$ represents the parameter vector after t gradient descent updates.

5.2.10 Convergence Behavior

Near a local minimum $\mathbf{w}^*$, the error surface can be approximated by a quadratic function. Let $\mathbf{H}$ denote the Hessian matrix at $\mathbf{w}^*$, with eigenpairs $(\lambda_i, \mathbf{u}_i)$.

Any displacement from the minimum can be written as

$$\mathbf{w}^{(t)} - \mathbf{w}^* = \sum_i \alpha_i^{(t)} \mathbf{u}_i.$$

For this quadratic approximation, the gradient is

$$\nabla E(\mathbf{w}^{(t)}) = \mathbf{H}(\mathbf{w}^{(t)} - \mathbf{w}^*).$$

Applying the gradient descent update

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla E(\mathbf{w}^{(t)}),$$

and projecting onto the eigenvector direction $\mathbf{u}_i$, yields the scalar recursion

$$\alpha_i^{(t+1)} = (1 - \eta\lambda_i)\, \alpha_i^{(t)}.$$

This equation shows that gradient descent converges *linearly* and *independently* along each eigenvector direction of the Hessian.

Convergence condition For convergence along direction i, the magnitude of the update factor must be less than one:

$$|1 - \eta\lambda_i| < 1.$$

This implies the learning rate must satisfy

$$0 < \eta < \frac{2}{\lambda_{\max}},$$

where $\lambda_{\max} = \max_i \lambda_i$.

Thus, the largest curvature direction determines the maximum stable learning rate.

Example 1: One-dimensional quadratic Consider

$$E(w) = \tfrac{1}{2}\lambda w^2, \quad \lambda > 0.$$

Gradient descent gives

$$w^{(t+1)} = (1 - \eta\lambda) w^{(t)}.$$

Convergence occurs if

$$|1 - \eta\lambda| < 1,$$

which yields $0 < \eta < \frac{2}{\lambda}$. If η is too large, the iterates oscillate or diverge.

Example 2: Narrow valley (two dimensions) Let

$$E(w_1, w_2) = \tfrac{1}{2}(\lambda_1 w_1^2 + \lambda_2 w_2^2), \quad \lambda_1 \gg \lambda_2 > 0.$$

The updates become

$$w_1^{(t+1)} = (1 - \eta\lambda_1) w_1^{(t)}, \qquad w_2^{(t+1)} = (1 - \eta\lambda_2) w_2^{(t)}.$$

To ensure stability in the steep direction w_1, the learning rate must satisfy

$$\eta < \frac{2}{\lambda_1}.$$

However, since $\lambda_2 \ll \lambda_1$,

$$1 - \eta\lambda_2 \approx 1,$$

and convergence along w_2 is extremely slow.

This mismatch causes the iterates to bounce across the steep direction while making slow progress along the flat direction, producing the characteristic *zig-zag* trajectory.

Interpretation

- Each eigenvalue λ_i controls the convergence rate along direction $\mathbf{u}_i$.
- Large eigenvalues enforce small learning rates.
- Small eigenvalues cause slow convergence.
- Ill-conditioned problems ($\lambda_{\max} \gg \lambda_{\min}$) lead to zig-zag behavior.

Summary Gradient descent converges linearly near a local minimum, with the slowest convergence governed by the smallest eigenvalues of the Hessian and the maximum allowable learning rate governed by the largest eigenvalue.

5.2.11 Vanishing and Exploding Gradients

Training deep neural networks using gradient-based optimization relies on the efficient propagation of error signals from the output layer back to the early layers of the network. As network depth increases, however, gradients may become exponentially small or exponentially large.

These phenomena are known as *vanishing gradients* and *exploding gradients*, respectively.

5.2.12 Gradient Propagation via the Chain Rule

Consider a feedforward neural network with K layers. Let

$$\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}, \qquad \mathbf{a}^{(k)} = \sigma(\mathbf{z}^{(k)}),$$

where $\mathbf{a}^{(k)}$ denotes the activation vector at layer k, and E denotes the loss function.

The gradient of the loss with respect to parameters in early layers is obtained by repeated application of the chain rule. For example, the gradient with respect to the activations of the first layer satisfies

$$\frac{\partial E}{\partial \mathbf{a}^{(1)}} = \left(\prod_{k=2}^{K} \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{a}^{(k-1)}} \right) \frac{\partial E}{\partial \mathbf{a}^{(K)}}.$$

Layer-wise Jacobian The Jacobian matrix of layer k is given by

$$\mathbf{J}^{(k)} = \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{a}^{(k-1)}} = \operatorname{diag}\Big(\sigma'(\mathbf{z}^{(k)})\Big)\, \mathbf{W}^{(k)}.$$

Thus, backpropagation is governed by the product of Jacobians across layers:

$$\frac{\partial E}{\partial \mathbf{a}^{(1)}} = \left(\prod_{k=2}^{K} \mathbf{J}^{(k)}\right) \frac{\partial E}{\partial \mathbf{a}^{(K)}}.$$

5.2.13 Vanishing Gradients

If the spectral norms (or singular values) of the Jacobian matrices satisfy

$$\|\mathbf{J}^{(k)}\| < 1 \quad \text{for most layers,}$$

then repeated multiplication yields

$$\left\| \prod_{k=2}^{K} \mathbf{J}^{(k)} \right\| \xrightarrow[K\to\infty]{} 0.$$

Consequently, gradients shrink exponentially with network depth:

$$\left\| \frac{\partial E}{\partial \mathbf{a}^{(1)}} \right\| \ll \left\| \frac{\partial E}{\partial \mathbf{a}^{(K)}} \right\|.$$

Implications

- Early layers learn extremely slowly.
- Gradient updates become numerically insignificant.
- Feature extraction in early layers is impaired.

Example: Sigmoid activation For the sigmoid function,

$$\sigma(z) = \frac{1}{1+e^{-z}}, \qquad \sigma'(z) = \sigma(z)(1-\sigma(z)) \le \tfrac{1}{4}.$$

Thus, each Jacobian contains factors strictly less than one. Repeated multiplication across layers guarantees exponential decay of gradients, explaining why deep sigmoid networks suffer from vanishing gradients.

5.2.14 Exploding Gradients

Conversely, if the Jacobian norms satisfy

$$\|\mathbf{J}^{(k)}\| > 1 \quad \text{for many layers,}$$

then

$$\left\| \prod_{k=2}^{K} \mathbf{J}^{(k)} \right\| \xrightarrow[K\to\infty]{} \infty.$$

This leads to exploding gradients, causing unstable parameter updates and numerical overflow.

Example: Large weight matrices If weight matrices are initialized with large spectral norms, then even moderate activation derivatives can cause Jacobian norms greater than one, leading to exponential growth of gradients during backpropagation.

Summary

- Backpropagation is governed by products of Jacobians of the form

$$\operatorname{diag}(\sigma'(\mathbf{z}^{(k)}))\mathbf{W}^{(k)}.$$

- Jacobian norms less than one cause vanishing gradients.
- Jacobian norms greater than one cause exploding gradients.
- Activation functions, weight initialization, and network depth jointly determine gradient stability.

5.3 Normalization Methods

Normalization rescales variables in a neural network to improve numerical stability and optimization efficiency. Table 5.3 compares common normalization techniques based on where and how normalization is applied.

Table 5.3: Comparison of normalization methods in neural networks

Method	Normalization equation	Axis of normalization	Typical use / example
Input (data) normalization	$\tilde{\mathbf{x}} = \frac{\mathbf{x}-\boldsymbol{\mu}_{\text{data}}}{\boldsymbol{\sigma}_{\text{data}}}$	Across dataset samples (per feature)	Image pixel normalization or feature standardization before training
Batch normalization	$\hat{\mathbf{z}}^{(k)} = \frac{\mathbf{z}^{(k)}-\boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2+\epsilon}}$, $\mathbf{a}^{(k)} = \gamma\hat{\mathbf{z}}^{(k)} + \beta$	Across mini-batch samples (per feature)	Deep CNNs and feedforward networks with large batch sizes
Layer normalization	$\hat{\mathbf{z}}^{(k)} = \frac{\mathbf{z}^{(k)}-\mu_L}{\sqrt{\sigma_L^2+\epsilon}}$, $\mathbf{a}^{(k)} = \gamma\hat{\mathbf{z}}^{(k)} + \beta$	Across features within a single sample	Transformers and recurrent networks with variable batch sizes

Explanation Input normalization standardizes raw data and remains fixed throughout training. Batch normalization normalizes intermediate pre-activations using mini-batch statistics, stabilizing gradient propagation but introducing dependence on batch size. Layer normalization operates independently on each sample, making it suitable for sequence models and architectures with small or varying batch sizes.

Example

- Input normalization scales image pixels to zero mean and unit variance before training a convolutional network.

- Batch normalization enables stable training of very deep CNNs by controlling the distribution of hidden-layer activations.
- Layer normalization is used in Transformer models to stabilize training across time steps and sequence lengths.

Table 5.4: Comparison of Batch Normalization and Layer Normalization

Aspect	**Batch Normalization (BN)**	**Layer Normalization (LN)**
What is normalized?	Pre-activations $\mathbf{z}^{(k)}$ for each feature (neuron) separately	Pre-activations $\mathbf{z}^{(k)}$ across all features within one sample
Mean and variance computed over	Mini-batch dimension (across samples)	Feature dimension (within a single sample)
Mean	$\mu_{B,j} = \frac{1}{B}\sum_{i=1}^{B} z_{i,j}$	$\mu_{L,i} = \frac{1}{d}\sum_{j=1}^{d} z_{i,j}$
Variance	$\sigma^2_{B,j} = \frac{1}{B}\sum_{i=1}^{B}(z_{i,j}-\mu_{B,j})^2$	$\sigma^2_{L,i} = \frac{1}{d}\sum_{j=1}^{d}(z_{i,j}-\mu_{L,i})^2$
Normalization step	$\hat{z}_{i,j} = \frac{z_{i,j}-\mu_{B,j}}{\sqrt{\sigma^2_{B,j}+\epsilon}}$	$\hat{z}_{i,j} = \frac{z_{i,j}-\mu_{L,i}}{\sqrt{\sigma^2_{L,i}+\epsilon}}$
Dependence on batch size	Strong dependence; unstable for small batches	Independent of batch size
Training vs inference	Uses batch statistics during training and running averages during inference	Same computation during training and inference
Typical architectures	Convolutional and feedforward networks with large batches	Transformers and recurrent networks
Key intuition	"Normalize across samples for each neuron"	"Normalize across neurons for each sample"

Chapter 6

Regularization Principles: Tikhonov, Bayesian Priors, and Modern Deep Networks

6.1 Tikhonov Regularization: The Classical Perspective

Tikhonov regularization arises as a principled solution to ill-posed inverse problems. Consider a linear operator

$$A : \mathbb{R}^n \to \mathbb{R}^m$$

and the linear system

$$Ax = b,$$

where $b \in \mathbb{R}^m$ is observed data and $x \in \mathbb{R}^n$ is unknown.

In practice, one or more of the following occur:

- A is ill-conditioned (small singular values),
- the system is underdetermined ($m < n$),
- the data b contains noise.

Direct inversion is unstable. Small perturbations in b may produce arbitrarily large changes in x.

Tikhonov proposed solving instead the penalized problem

$$\min_{x \in \mathbb{R}^n} \left\{ \|Ax - b\|_2^2 + \lambda \|x\|_2^2 \right\}, \qquad \lambda > 0.$$

The quadratic penalty enforces stability by controlling the norm of the solution.

6.1.1 Closed-Form Solution

The minimizer satisfies the normal equation

$$(A^\top A + \lambda I) x_\lambda = A^\top b,$$

hence

$$x_\lambda = (A^\top A + \lambda I)^{-1} A^\top b.$$

Since $A^\top A + \lambda I$ is strictly positive definite for $\lambda > 0$, the solution exists uniquely and depends continuously on b.

Thus,

$$\text{Tikhonov regularization} = \text{L2 regularization} = \text{Ridge regression}.$$

They are mathematically identical.

6.2 Statistical Learning Theory View

Consider empirical risk minimization over a hypothesis space $\mathcal{H}$:

$$\min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i).$$

With finite data, a model may interpolate the training set while generalizing poorly.

Generalization bounds typically take the form

$$\mathcal{E}_{\text{test}}(f) \leq \mathcal{E}_{\text{train}}(f) + \mathcal{C}(\mathcal{H}),$$

where $\mathcal{C}(\mathcal{H})$ is a complexity term depending on

- VC dimension,
- Rademacher complexity,
- norm constraints.

Imposing a constraint such as

$$\|w\|_2 \leq C$$

shrinks the effective capacity of the hypothesis class.

Therefore, L2 regularization reduces statistical complexity in addition to stabilizing numerical inversion.

6.3 Functional Analysis View: Ill-Posed Operators

Let

$$A : \mathcal{X} \to \mathcal{Y}$$

be a bounded linear operator between Hilbert spaces.

A problem is well-posed if:

1. A solution exists,
2. The solution is unique,
3. The solution depends continuously on the data.

If A has small singular values σ_i, then inversion involves division by σ_i, and instability arises as $\sigma_i \to 0$.

Using the singular value decomposition

$$A = U \Sigma V^\top,$$

Tikhonov replaces the inverse

$$A^{-1}$$

with the regularized inverse

$$(A^\top A + \lambda I)^{-1} A^\top.$$

In spectral form,

$$\frac{\sigma_i}{\sigma_i^2 + \lambda}$$

damps directions corresponding to small σ_i.

This mechanism is known as spectral filtering.

In deep learning, weight decay performs the same stabilization in parameter space.

6.4 Bayesian Interpretation

Assume a linear Gaussian model:

$$b = Ax + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2 I).$$

The likelihood is

$$p(b \mid x) \propto \exp\left(-\frac{1}{2\sigma^2}\|Ax - b\|_2^2\right).$$

Assume a Gaussian prior:

$$x \sim \mathcal{N}(0, \tau^2 I).$$

The posterior maximizer (MAP estimator) solves

$$\min_x \left\{\|Ax - b\|_2^2 + \lambda\|x\|_2^2\right\}, \qquad \lambda = \frac{\sigma^2}{\tau^2}.$$

Thus,

$$\text{L2 regularization} = \text{Gaussian prior}.$$

6.5 Connections to Modern Deep Learning

6.5.1 Weight Decay

Neural network training solves

$$\min_\theta \mathcal{L}(\theta) + \lambda\|\theta\|_2^2.$$

This is precisely Tikhonov regularization applied in high-dimensional parameter space.

6.5.2 Dropout

Dropout introduces multiplicative Bernoulli noise,

$$h \mapsto m \odot h, \quad m_i \sim \text{Bernoulli}(p),$$

which approximates Bayesian model averaging and induces implicit regularization.

6.5.3 Early Stopping

For least squares, gradient descent yields

$$x^{(k)} = \sum_i \left(1 - (1 - \eta\sigma_i^2)^k\right) \frac{u_i^\top b}{\sigma_i} v_i.$$

Stopping early suppresses small singular directions.
Thus, early stopping is implicit spectral regularization.

6.5.4 Normalization Layers

Batch normalization and layer normalization do not directly penalize norms but modify the geometry of the optimization landscape, often leading to flatter minima and improved generalization.

6.6 Conceptual Synthesis

Regularization mechanisms all serve to:

1. Restrict the effective hypothesis space,
2. Stabilize inverse problems,
3. Introduce inductive bias,
4. Improve generalization.

Without such bias, learning from finite data is fundamentally ill-posed, consistent with the No Free Lunch theorem.

6.7 Regularization in Deep Networks

Training a deep model solves a nonlinear inverse problem:

$$f_\theta(x_i) \approx y_i.$$

Infinitely many parameters θ interpolate the data.
Regularization selects among them by enforcing structure:

- Small parameter norm,
- Smoothness,
- Stability,
- Simplicity.

In this sense, Tikhonov regularization extends naturally to modern deep learning as a high-dimensional stabilization principle.

6.8 Symmetry, Invariance, and Equivariance

In many machine learning applications, predictions should behave consistently under certain transformations of the input. These transformations may represent symmetries of the underlying task. Understanding how a model responds to such transformations leads to the notions of invariance and equivariance.

Let the input be an image I, and let T denote a transformation acting on the input (e.g., translation, rotation, scaling). Let S denote a neural network or function mapping inputs to outputs.

6.8.1 Symmetry

A transformation T represents a symmetry of a task if applying T to the input does not change the essential property being predicted.

Formally, a task exhibits symmetry under transformation T if

$$\text{Task}(T(I)) = \text{Task}(I).$$

Example 1: Object presence In image classification, the question

"Is there a cat in the image?"

is invariant to translations of the cat inside the image. Thus translation is a symmetry of the task.

Example 2: Physical laws In physics-informed learning, many systems are translation invariant:

$$f(x + c) = f(x),$$

for constant shift c. This reflects translational symmetry of the governing equations.

Transformations such as translation, rotation, or scaling that leave relevant properties unchanged are examples of symmetries.

6.8.2 Invariance

A function f is invariant to a transformation T if

$$f(T(I)) = f(I).$$

Invariance means the output remains unchanged when the input is transformed.

Example 1: Image classification Suppose $f(I)$ predicts a class label. If the image is shifted:

$$f(T_{\text{translate}}(I)) = f(I).$$

Similarly, many classification models aim for rotation invariance:

$$f(T_{\text{rotate}}(I)) = f(I).$$

Example 2: Norm function Consider the Euclidean norm:

$$f(\mathbf{x}) = \|\mathbf{x}\|_2.$$

This function is invariant to orthogonal rotations R:

$$f(R\mathbf{x}) = \|R\mathbf{x}\|_2 = \|\mathbf{x}\|_2.$$

Thus the norm is rotation invariant.

Example 3: Set pooling If a model aggregates features using summation,

$$f(\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}) = \sum_i \mathbf{x}_i,$$

the function is invariant to permutation of inputs. This is crucial in graph neural networks.

6.8.3 Equivariance

A function S is equivariant to transformation T if transforming the input results in a predictable transformation of the output:

$$S(T(I)) = \tilde{T}(S(I)),$$

where $\tilde{T}$ is the corresponding transformation acting on the output space.

Equivariance means the output changes in the same structured way as the input.

Example 1: Image segmentation Suppose $S(I)$ produces a pixel-wise segmentation mask. If the input image is translated by 10 pixels to the right, the segmentation mask should also translate by 10 pixels:

$$S(T_{\text{translate}}(I)) = T_{\text{translate}}(S(I)).$$

Example 2: Convolution Let $\mathcal{C}$ denote convolution with kernel K. Then

$$\mathcal{C}(T_{\text{translate}}(I)) = T_{\text{translate}}(\mathcal{C}(I)).$$

Thus convolution is translation equivariant.

Example 3: Linear maps Let

$$S(\mathbf{x}) = A\mathbf{x},$$

where A is a matrix. If $T(\mathbf{x}) = R\mathbf{x}$ is a rotation and A commutes with R, then

$$S(R\mathbf{x}) = AR\mathbf{x} = RA\mathbf{x} = RS(\mathbf{x}),$$

which shows equivariance.

6.8.4 Vision Tasks

Different tasks require different transformation behavior:

- **Classification:** translation invariant
- **Segmentation:** translation equivariant
- **Object detection:** partially equivariant (bounding boxes shift)
- **Pose estimation:** rotation equivariant

For classification, translation equivariant feature maps are typically followed by pooling operations, producing approximate translation invariance:

$$\text{Pooling}(\mathcal{C}(T(I))) = \text{Pooling}(\mathcal{C}(I)).$$

6.8.5 Summary Table

Property	Mathematical Form	Example
Symmetry	$\text{Task}(T(I)) = \text{Task}(I)$	Object presence under translation
Invariance	$f(T(I)) = f(I)$	Classification, norm function
Equivariance	$S(T(I)) = \tilde{T}(S(I))$	Segmentation, convolution

6.8.6 Conceptual Distinction

- **Symmetry** is a property of the task.
- **Invariance** is a property of the model output remaining unchanged.
- **Equivariance** is a property of the model output transforming predictably.

Invariance is appropriate for global prediction tasks such as classification, while equivariance is essential for structured output tasks such as segmentation, detection, and localization.

6.9 Enforcing Invariance and Regularization

6.9.1 Methods for Enforcing Invariance

Invariance can be encouraged through several complementary strategies:

1. **Pre-processing:** Construct invariant features.
2. **Regularized error functions:** Penalize sensitivity to transformations.
3. **Data augmentation:** Expand training data using transformed samples.
4. **Network architecture:** Encode invariance structurally.

Example 1: Pre-processing In image classification, instead of using raw pixel coordinates, one may use

$$f(I) = \text{histogram of gradients}(I),$$

which is more robust to small translations and lighting variations.

Example 2: Data augmentation If T is a translation operator, training on

$$\{I, T(I), T^2(I), \dots\}$$

encourages the model to satisfy approximately

$$f(T(I)) \approx f(I).$$

Example 3: Architectural enforcement Convolution satisfies

$$\mathcal{C}(T(I)) = T(\mathcal{C}(I)),$$

which enforces translation equivariance structurally.

Pooling then yields approximate invariance.

6.10 Error Function Regularization

A common approach to regularization is to modify the error function by adding a penalty discouraging large parameter values:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w}.$$

Here:

- $\mathbf{w}$ is the parameter vector,
- $E(\mathbf{w})$ is the original loss,
- $\lambda > 0$ controls regularization strength.

This is known as **L2 regularization** or **weight decay**.

6.10.1 Interpretation

The added term is

$$\frac{\lambda}{2}\|\mathbf{w}\|_2^2.$$

It penalizes large parameter magnitudes and encourages solutions near the origin.

Effect of λ

- $\lambda = 0$: no regularization.
- Large λ: strong shrinkage toward zero.

6.10.2 Gradients of Regularized Error Functions

Regularization modifies the objective function:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\,\Omega(\mathbf{w}),$$

where $\Omega(\mathbf{w})$ is a penalty term.

L2 Regularization (Weight Decay)

Penalty:

$$\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2 = \frac{1}{2}\mathbf{w}^\top\mathbf{w}.$$

Regularized objective:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w}.$$

Gradient:

$$\nabla\tilde{E}(\mathbf{w}) = \nabla E(\mathbf{w}) + \lambda\mathbf{w}.$$

Gradient descent update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\left(\nabla E(\mathbf{w}_t) + \lambda\mathbf{w}_t\right).$$

Rewriting,

$$\mathbf{w}_{t+1} = (1 - \eta\lambda)\mathbf{w}_t - \eta\nabla E(\mathbf{w}_t).$$

Thus, even if $\nabla E = 0$,

$$\mathbf{w}_{t+1} = (1 - \eta\lambda)\mathbf{w}_t,$$

showing exponential shrinkage toward zero.
This is why L2 regularization is called **weight decay**.

L1 Regularization

Penalty:

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_i |w_i|.$$

Regularized objective:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\|\mathbf{w}\|_1.$$

The L1 norm is not differentiable at $w_i = 0$. Its subgradient is:

$$\frac{\partial}{\partial w_i}|w_i| = \begin{cases} 1, & w_i > 0 \\ -1, & w_i < 0 \\ \in [-1, 1], & w_i = 0 \end{cases}$$

Thus, the subgradient of the regularized objective is:

$$\nabla \tilde{E}(\mathbf{w}) = \nabla E(\mathbf{w}) + \lambda \operatorname{sign}(\mathbf{w}),$$

where

$$\operatorname{sign}(w_i) = \begin{cases} 1, & w_i > 0 \\ -1, & w_i < 0 \\ \in [-1, 1], & w_i = 0 \end{cases}$$

Update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left(\nabla E(\mathbf{w}_t) + \lambda \operatorname{sign}(\mathbf{w}_t) \right).$$

Key property Unlike L2, L1 does not scale shrinkage with magnitude. Instead, it applies a constant pull toward zero.

This encourages **sparsity**, meaning many weights become exactly zero.

General L_p Regularization

Penalty:

$$\Omega(\mathbf{w}) = \frac{1}{p} \|\mathbf{w}\|_p^p = \frac{1}{p} \sum_i |w_i|^p.$$

Regularized objective:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{p} \sum_i |w_i|^p.$$

For $p > 1$, the gradient is:

$$\frac{\partial}{\partial w_i} \left(\frac{1}{p} |w_i|^p \right) = |w_i|^{p-2} w_i.$$

Thus,

$$\nabla \tilde{E}(\mathbf{w}) = \nabla E(\mathbf{w}) + \lambda \, |\mathbf{w}|^{p-2} \odot \mathbf{w},$$

where $\odot$ denotes elementwise multiplication.

Special cases:

$$p = 2 \Rightarrow \lambda \mathbf{w}$$

$$p = 1 \Rightarrow \lambda \operatorname{sign}(\mathbf{w})$$

Behavior comparison

- $p = 2$: smooth shrinkage, no sparsity.
- $p = 1$: promotes sparsity.
- $p > 2$: strongly penalizes large weights.
- $p < 1$: non-convex, strongly sparse but harder to optimize.

Summary

$$\text{L2:} \quad + \lambda \mathbf{w}$$

$$\text{L1:} \quad + \lambda \, \text{sign}(\mathbf{w})$$

$$\text{Lp:} \quad + \lambda \, |\mathbf{w}|^{p-2} \odot \mathbf{w}$$

L2 yields smooth weight decay, L1 yields sparse solutions, general L_p interpolates between these behaviors.

6.10.3 Geometric Interpretation

Suppose the original loss near its minimum is quadratic:

$$E(\mathbf{w}) = \frac{1}{2} \mathbf{w}^\top H \mathbf{w}.$$

Contours of $E(\mathbf{w})$ are ellipses.
The regularization term has contours:

$$\|\mathbf{w}\|_2^2 = \text{constant},$$

which are circles.
The minimum of

$$E(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

occurs where elliptical contours intersect circular contours.

Example: Two-dimensional case Let

$$E(w_1, w_2) = \frac{1}{2}(4w_1^2 + w_2^2).$$

Unregularized minimum: $(0, 0)$.
Suppose instead the minimum without regularization were at $(2, 1)$.
Adding regularization yields

$$\tilde{E}(w_1, w_2) = \frac{1}{2}(4w_1^2 + w_2^2) + \frac{\lambda}{2}(w_1^2 + w_2^2).$$

The effective curvature becomes

$$(4 + \lambda) \text{ in direction } w_1, \quad (1 + \lambda) \text{ in direction } w_2.$$

The solution shifts toward the origin.

Directional effect Regularization shrinks more strongly in directions where:

$$\text{data curvature is small.}$$

This suppresses parameters that contribute weakly to reducing training error.

6.10.4 Practical Example: Linear Regression

Consider linear regression:

$$E(\mathbf{w}) = \frac{1}{2N}\|X\mathbf{w} - \mathbf{y}\|^2.$$

The L2-regularized solution satisfies:

$$(X^\top X + \lambda I)\mathbf{w} = X^\top \mathbf{y}.$$

Thus,

$$\mathbf{w} = (X^\top X + \lambda I)^{-1} X^\top \mathbf{y}.$$

Adding λI stabilizes inversion and reduces variance.
This is precisely Tikhonov regularization.

6.10.5 Summary

- Regularization modifies the objective:

$$E(\mathbf{w}) \to E(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2.$$

- The gradient becomes:

$$\nabla E + \lambda \mathbf{w}.$$

- Geometrically, solutions move toward the origin.
- Regularization reduces overfitting by suppressing weakly supported parameter directions.

6.11 L_p Regularization in PyTorch

Consider the regularized objective

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}_{\text{data}}(\theta) + \frac{\lambda}{p}\sum_{\ell} \|W_\ell\|_p^p, \qquad p > 1.$$

Here only weight matrices W_ℓ are regularized (biases excluded).

6.11.1 PyTorch Implementation

```
p = 3.0
lam = 1e-4

def lp_weight_penalty(model, p: float):
    reg = 0.0
    for name, param in model.named_parameters():
        if not param.requires_grad:
            continue
        if name.endswith("weight"):  # weights only
            reg = reg + param.abs().pow(p).sum()
    return reg

for x, y in train_loader:
    optimizer.zero_grad(set_to_none=True)
```

```
    yhat = model(x)
    data_loss = criterion(yhat, y)

    reg = lp_weight_penalty(model, p)
    loss = data_loss + (lam / p) * reg

    loss.backward()
    optimizer.step()
```

6.11.2 Gradient Behavior

Because the regularizer depends only on weights,

$$\tilde{\mathcal{L}} = \mathcal{L} + R(W),$$

we obtain

$$\frac{\partial \tilde{\mathcal{L}}}{\partial a} = \frac{\partial \mathcal{L}}{\partial a}, \qquad \frac{\partial \tilde{\mathcal{L}}}{\partial z} = \frac{\partial \mathcal{L}}{\partial z}.$$

Thus, backpropagation deltas are unchanged.

Only parameter gradients are modified:

$$\frac{\partial \tilde{\mathcal{L}}}{\partial W} = \frac{\partial \mathcal{L}}{\partial W} + \lambda |W|^{p-2} \odot W.$$

For special cases:

$$p = 2 \Rightarrow +\lambda W, \qquad p = 3 \Rightarrow +\lambda |W| W.$$

Bias gradients remain unchanged when biases are not regularized.

6.12 Inspecting the Autograd Graph

PyTorch builds a dynamic computation graph. The loss

$$\texttt{loss} = \texttt{data\_loss} + \frac{\lambda}{p}\texttt{reg}$$

creates two separate computational branches:

$$\tilde{\mathcal{L}} = \mathcal{L}_{\text{data}} + R(W).$$

6.12.1 Manual Graph Inspection

Each tensor stores a `grad_fn` describing how it was created:

```
loss = data_loss + (lam / p) * reg
print(loss.grad_fn)
```

One can traverse backward:

```
fn = loss.grad_fn
while fn is not None:
    print(fn)
    next_fns = fn.next_functions
    if not next_fns:
        break
    fn = next_fns[0][0]
```

Typical nodes include:

```
AddBackward0
MseLossBackward0
SigmoidBackward0
PowBackward0
AbsBackward0
```

The `AddBackward0` node merges:

- the data loss branch,
- the regularization branch.

6.12.2 Graph Visualization with torchviz

```
# install once
pip install torchviz
```

```
from torchviz import make_dot

loss = data_loss + (lam / p) * reg
graph = make_dot(loss, params=dict(model.named_parameters()))
graph.render("autograd_graph", format="png")
```

This produces a visualization where:

- one branch flows through activations to the data loss,
- one branch flows directly from weights to the regularizer,
- both merge at the final loss node.

6.12.3 Key Principle

Autograd does not label gradients by source. It applies the chain rule over the computation graph. If a term does not depend on a tensor, its derivative with respect to that tensor is structurally zero:

$$\frac{\partial R(W)}{\partial a} = 0, \qquad \frac{\partial R(W)}{\partial z} = 0.$$

Therefore, regularization affects only the gradients of parameters appearing in the regularizer.

6.13 Consistency of Regularizers and Scaling Symmetries

A desirable property of regularization is *consistency*: if two parameter sets represent the same function, the regularizer should assign them the same penalty.

Neural networks often admit multiple equivalent parameterizations due to internal scaling symmetries. Standard weight decay does not always respect these symmetries.

6.13.1 Two-Layer Network Formulation

Consider a two-layer neural network.

First layer

$$z_j = \sum_i w_{ji} x_i + w_{j0}, \qquad a_j = \sigma(z_j),$$

where:

- $W_1 = \{w_{ji}\}$ are first-layer weights,
- w_{j0} are first-layer biases,
- $\sigma(\cdot)$ is a nonlinear activation function.

Second layer (linear output)

$$y_k = \sum_j v_{kj} a_j + v_{k0},$$

where:

- $W_2 = \{v_{kj}\}$ are second-layer weights,
- v_{k0} are output biases.

We now analyze invariances of this model.

6.13.2 Invariance Under Input Linear Transformations

Suppose inputs undergo a linear transformation:

$$x_i \to \tilde{x}_i = a x_i + b.$$

The first-layer pre-activation becomes:

$$\tilde{z}_j = \sum_i w_{ji} \tilde{x}_i + w_{j0} = \sum_i w_{ji}(a x_i + b) + w_{j0}.$$

Expanding,

$$\tilde{z}_j = a \sum_i w_{ji} x_i + b \sum_i w_{ji} + w_{j0}.$$

To preserve the original mapping, define transformed parameters:

$$\tilde{w}_{ji} = \frac{1}{a} w_{ji}, \qquad \tilde{w}_{j0} = w_{j0} - \frac{b}{a} \sum_i w_{ji}.$$

Substituting,

$$\tilde{z}_j = \sum_i \tilde{w}_{ji} \tilde{x}_i + \tilde{w}_{j0} = z_j.$$

Therefore,

$$\tilde{a}_j = \sigma(\tilde{z}_j) = \sigma(z_j) = a_j.$$

The overall function remains unchanged.

6.13.3 Hidden Unit Rescaling Symmetry

A more fundamental symmetry arises from rescaling hidden activations.

Suppose we rescale hidden units:

$$a_j \to \tilde{a}_j = ca_j.$$

To preserve the output,

$$y_k = \sum_j v_{kj} a_j,$$

we compensate by transforming second-layer weights:

$$v_{kj} \to \tilde{v}_{kj} = \frac{1}{c} v_{kj}.$$

Then:

$$\sum_j \tilde{v}_{kj} \tilde{a}_j = \sum_j \frac{1}{c} v_{kj} (ca_j) = \sum_j v_{kj} a_j.$$

Thus,

$$(W_1, W_2) \quad \text{and} \quad (cW_1, \tfrac{1}{c} W_2)$$

represent identical functions.

This symmetry exists even without changing the inputs.

6.13.4 Why Standard L_2 Regularization Breaks This Symmetry

Standard weight decay is

$$\Omega(W) = \frac{\lambda}{2} \left(\sum_{w \in W_1} w^2 + \sum_{v \in W_2} v^2 \right).$$

Under hidden scaling:

$$W_1 \to cW_1, \qquad W_2 \to \frac{1}{c} W_2,$$

the penalty becomes:

$$\Omega = \frac{\lambda}{2} \left(c^2 \sum_{W_1} w^2 + \frac{1}{c^2} \sum_{W_2} v^2 \right).$$

The penalty changes with c, even though the network function is unchanged.

Therefore:

> Standard weight decay penalizes parameterization, not function complexity.

6.13.5 Concrete Numeric Example

Let

$$W_1 = 2, \qquad W_2 = 3.$$

Then

$$\Omega = \frac{\lambda}{2}(2^2 + 3^2) = \frac{13\lambda}{2}.$$

Rescale hidden units with $c = 2$:

$$W_1 \to 4, \qquad W_2 \to \frac{3}{2}.$$

New penalty:

$$\frac{\lambda}{2}(16 + 2.25) = \frac{\lambda}{2}(18.25).$$

Different penalty. Same function.
This illustrates the inconsistency.

6.13.6 Layer-Wise Regularization

One improvement is to use layer-specific penalties:

$$\Omega(W) = \frac{\lambda_1}{2} \sum_{w \in W_1} w^2 + \frac{\lambda_2}{2} \sum_{v \in W_2} v^2.$$

Under hidden rescaling by c, invariance can be preserved if

$$\lambda_1 \to c^{-2}\lambda_1, \qquad \lambda_2 \to c^2\lambda_2.$$

This aligns regularization with network symmetries.

6.13.7 Bayesian Interpretation

Quadratic regularization corresponds to Gaussian priors:

$$p(W) \propto \exp\left(-\frac{\alpha_1}{2} \sum_{W_1} w^2 - \frac{\alpha_2}{2} \sum_{W_2} v^2 \right).$$

Equivalently,

$$W_1 \sim \mathcal{N}(0, \alpha_1^{-1}), \qquad W_2 \sim \mathcal{N}(0, \alpha_2^{-1}).$$

Larger precision parameters α imply stronger shrinkage.

6.13.8 Summary

Neural networks possess internal scaling symmetries:

$$(W_1, W_2) \sim (cW_1, \tfrac{1}{c}W_2).$$

Standard weight decay breaks this symmetry. Invariant regularization seeks to penalize functions rather than arbitrary parameter scalings.

6.14 Classical Bias–Variance Trade-Off and the Modern Over-parameterized Regime

6.14.1 Classical Bias–Variance Trade-Off

Consider the standard supervised learning model

$$y = f^\star(x) + \varepsilon, \qquad \mathbb{E}[\varepsilon] = 0, \qquad \mathrm{Var}(\varepsilon) = \sigma^2,$$

where $f^\star$ is the true function and ε represents irreducible noise.

For a learned predictor $\hat{f}(x)$, the expected squared test error decomposes as

$$\mathbb{E}\big[(\hat{f}(x) - y)^2\big] = \underbrace{\big(\mathbb{E}[\hat{f}(x)] - f^\star(x)\big)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}\big[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2\big]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible error}}.$$

Bias. Bias measures systematic error:

$$\text{Bias}(x) = \mathbb{E}[\hat{f}(x)] - f^\star(x).$$

High bias corresponds to underfitting. For example, fitting a linear model to quadratic data results in both high training and test errors.

Variance. Variance measures sensitivity to fluctuations in the training data:

$$\text{Var}(x) = \mathbb{E}\left[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2\right].$$

High variance corresponds to overfitting. For example, a high-degree polynomial may interpolate noisy data, achieving near-zero training error but large test error.

Classical complexity curve. As model complexity increases:

- Bias decreases,
- Variance increases.

This yields the classical U-shaped test error curve:

- Low complexity: underfitting (high bias),
- Intermediate complexity: optimal trade-off,
- High complexity: overfitting (high variance).

In classical statistics, increasing complexity beyond the interpolation threshold was expected to degrade generalization.

6.14.2 Modern Overparameterized Regime

Modern deep neural networks often operate in a regime where

$$\text{Number of parameters} \gg \text{Number of training samples}.$$

Such models can interpolate the training data:

$$\text{Training error} = 0.$$

According to the classical view, this should imply extreme variance and poor generalization. However, empirical evidence shows that in many cases:

$$\text{Test error decreases even beyond interpolation}.$$

This phenomenon leads to what is known as *double descent*.

6.14.3 Double Descent Phenomenon

As model size increases, test error often behaves as follows:

1. In the classical regime:
   - Test error decreases,
   - Then increases near the interpolation threshold.
2. In the overparameterized regime:
   - Test error decreases again.

Thus, the curve transitions from the classical U-shape to a *double descent* curve:

$$\text{U-shape} \longrightarrow \text{Peak at interpolation} \longrightarrow \text{Second descent}.$$

6.14.4 Why Does This Happen?

Several mechanisms help explain this behavior.

Implicit regularization of SGD. Stochastic gradient descent tends to prefer:

- Low-norm solutions,
- Flat minima,
- Structured representations.

Even in large parameter spaces, optimization dynamics restrict the effective solution complexity.

Parameterization versus function complexity. Increasing parameter count does not necessarily increase effective function complexity. Many parameter configurations represent the same function.

Flat minima. Overparameterized models often admit wide, flat valleys in the loss landscape. Solutions in flat regions tend to generalize better than sharp minima.

6.14.5 Illustrative Example

Suppose:

- Dataset size: 100 samples,
- Model A: 50 parameters,
- Model B: 10,000 parameters.

Both models may interpolate the data. However:

- Model A has limited representation flexibility,
- Model B has many equivalent interpolating solutions,
- SGD often converges to the minimum-norm interpolating solution.

Minimum-norm interpolators frequently generalize well, linking this behavior to:

- Ridge regression,
- Kernel methods,
- Neural tangent kernel (NTK) theory.

6.14.6 Key Contrast

Classical view.

$$\text{Complexity} \uparrow \Rightarrow \text{ Bias} \downarrow \Rightarrow \text{ Variance} \uparrow \Rightarrow \text{ Test error} \uparrow .$$

Modern deep learning view.

$$\text{Complexity} \uparrow \Rightarrow \text{ Bias} \downarrow,$$

$$\text{Variance initially} \uparrow, \quad \text{then implicit regularization dominates,}$$

$$\text{Test error} \downarrow \text{ again.}$$

6.14.7 Implications for Deep Architectures

Convolutional neural networks and transformers:

- Are highly overparameterized,
- Are trained with SGD,
- Possess strong architectural inductive biases.

Despite perfectly fitting training data, they often generalize well. Thus, generalization in deep learning depends not only on parameter count but also on:

- Optimization dynamics,
- Architectural structure,
- Data geometry,
- Implicit biases.

Conclusion. The classical bias–variance framework assumes a fixed hypothesis class and explicit regularization. Modern deep learning operates in highly overparameterized regimes with strong implicit regularization. A complete theory of generalization must therefore extend beyond the classical bias–variance trade-off.

Chapter 7

Vision Transformer (ViT)

7.1 Overview

The Vision Transformer (ViT) is a neural architecture that applies the Transformer model (originally developed for natural language processing) to image understanding tasks. Instead of relying on convolutional operations, ViT represents an image as a sequence of embedded patches and processes this sequence using stacked self-attention layers.

Unlike convolutional neural networks (which impose locality and translation equivariance through kernel design), ViT relies on global self-attention to model long-range spatial relationships. This design choice introduces different scaling behaviors in memory usage and computational cost, which must be carefully considered when ViT is used as a backbone in detection or segmentation frameworks such as Mask R-CNN.

7.2 Patch Embedding and Tokenization

Given an input image of size $H \times W \times C$ (height $\times$ width $\times$ channels), ViT first partitions the image into non-overlapping square patches of size $P \times P$. Each patch is flattened and projected into a fixed-dimensional embedding space.

The number of patch tokens is given by:

$$N_{\text{patch}} = \frac{H}{P} \cdot \frac{W}{P} \tag{7.1}$$

A special learnable classification token (`[CLS]`) is prepended to the patch sequence, yielding the total number of tokens:

$$N = N_{\text{patch}} + 1 \tag{7.2}$$

Each token is represented as a vector in $\mathbb{R}^D$, where D is the embedding dimension.

7.3 Transformer Depth and Attention Modules

A Vision Transformer consists of a stack of identical Transformer layers. Each layer contains exactly one multi-head self-attention (MHSA) module followed by a feed-forward multilayer perceptron (MLP).

- The number of **attention modules** is equal to the Transformer depth L.
- Each attention module processes the full token sequence.
- Depth is a user-defined architectural hyperparameter and is independent of token count, embedding dimension, and number of attention heads.

Thus:

$$\text{Number of attention modules} = L \tag{7.3}$$

7.4 Multi-Head Self-Attention

Within a single attention module, multi-head self-attention projects the token embeddings into queries, keys, and values, which are then split across multiple attention heads.

Let:

- D be the embedding dimension
- h be the number of attention heads

The per-head dimension is:

$$d_h = \frac{D}{h}, \quad \text{with } D \bmod h = 0 \tag{7.4}$$

For each head, attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_h}}\right)V \tag{7.5}$$

The attention matrix for each head has size:

$$N \times N \tag{7.6}$$

This quadratic dependence on the number of tokens is the primary source of memory and computational cost in ViT.

7.5 Computational Scaling

The dominant computational and memory terms in ViT arise from self-attention and MLP layers.

7.5.1 Attention Cost

Self-attention scales as:

$$\mathcal{O}(N^2 \cdot D) \tag{7.7}$$

This quadratic scaling with respect to the number of tokens makes patch size and input resolution critical design parameters.

7.5.2 MLP Cost

Each Transformer layer contains an MLP with hidden dimension:

$$D_{\text{MLP}} = r \cdot D \tag{7.8}$$

where r is the MLP expansion ratio, typically set to 4.

The MLP cost scales as:

$$\mathcal{O}(N \cdot D \cdot D_{\text{MLP}}) \tag{7.9}$$

7.6 Parameter Impact Summary

Table 7.1 summarizes the role and impact of key Vision Transformer parameters.

Parameter	Controls	Primary Impact
Patch size P	Number of tokens N	Quadratic attention memory and compute
Embedding dim D	Token feature width	Linear memory, quadratic MLP cost
Depth L	Number of attention modules	Linear scaling in memory and compute
Heads h	Attention parallelism	Minor overhead, $D \bmod h = 0$
MLP ratio r	Feed-forward width	Significant compute increase
Image resolution $H \times W$	Token count	Quadratic attention scaling

Table 7.1: Impact of Vision Transformer architectural parameters.

7.7 Numerical Example

Consider an input image of size 224×224 with patch size $P = 16$.

$$N_{\text{patch}} = (224/16)^2 = 14^2 = 196 \tag{7.10}$$

$$N = 196 + 1 = 197 \tag{7.11}$$

For an embedding dimension $D = 192$ and $h = 3$ heads:

$$d_h = 192/3 = 64 \tag{7.12}$$

The attention matrix size per layer has size:

$$h \cdot N^2 = 3 \times 197^2 = 116{,}427 \tag{7.13}$$

If the patch size is reduced to $P = 8$, the number of tokens increases to $N = 785$, and the attention matrix size increases to approximately 1.85 million entries per layer, illustrating the quadratic scaling effect.

7.8 Implications for Detection and Segmentation

When used as a backbone in detection or segmentation frameworks, ViT must often process images at higher resolutions than standard classification benchmarks. As a result, token count and attention memory become the dominant constraints.

Practical deployments typically rely on:

- Larger patch sizes
- Reduced depth
- Windowed or local attention mechanisms
- Hierarchical token merging

These modifications preserve the representational benefits of self-attention while keeping memory usage within feasible limits.

7.9 Summary

Vision Transformers replace convolutional inductive biases with global self-attention over patch tokens. The number of attention modules is determined solely by the Transformer depth, while computational cost is dominated by the quadratic dependence on the number of tokens.

Parameter	**Controls**	**Compute / Memory Scaling**	**Effect on Representation**
MLP ratio r	Hidden width of MLP (rD)	$\mathcal{O}(N \cdot rD^2)$ compute, $\mathcal{O}(N \cdot rD)$ activations	Channel-wise capacity (no token mixing)

Table 7.2: Impact of the MLP ratio in Vision Transformers.

Understanding these scaling relationships is essential when integrating ViT into large-scale vision systems such as Mask R-CNN.

The MLP ratio determines the width of the channel-wise feed-forward network applied independently to each token, increasing parameter count, compute, and activation memory linearly while leaving token–token interactions unchanged, which are governed exclusively by self-attention. In contrast to self-attention, which mixes information across tokens, the MLP operates only along the feature dimension and therefore controls expressive capacity rather than spatial interaction.

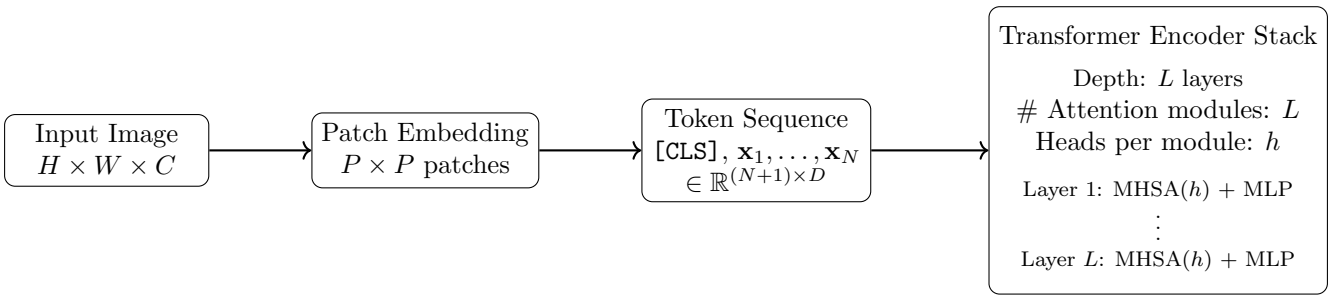

Figure 7.1: High-level architecture of the Vision Transformer (ViT). An input image is split into non-overlapping patches, embedded into a token sequence, and processed by a Transformer encoder stack of depth L. Each encoder layer contains exactly one multi-head self-attention (MHSA) module (so the number of attention modules is L), and each MHSA uses h attention heads.

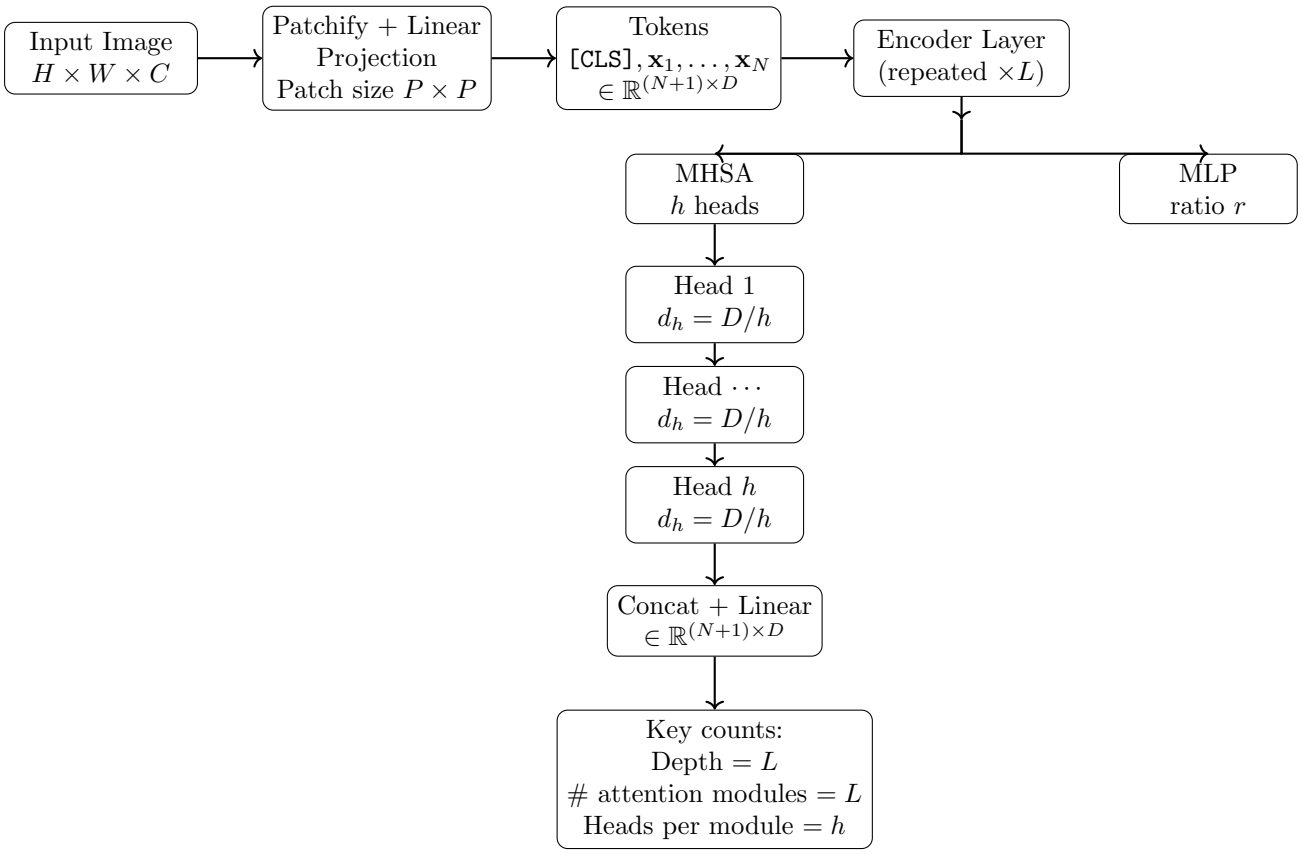

Figure 7.2: Hydra/tree-style view of ViT without overflow. Tokens pass through an encoder layer repeated $\times L$. Each layer contains exactly one MHSA (so the number of attention modules is L) with h heads and one MLP block. The MHSA splits into h heads and merges back via concatenation to dimension D.

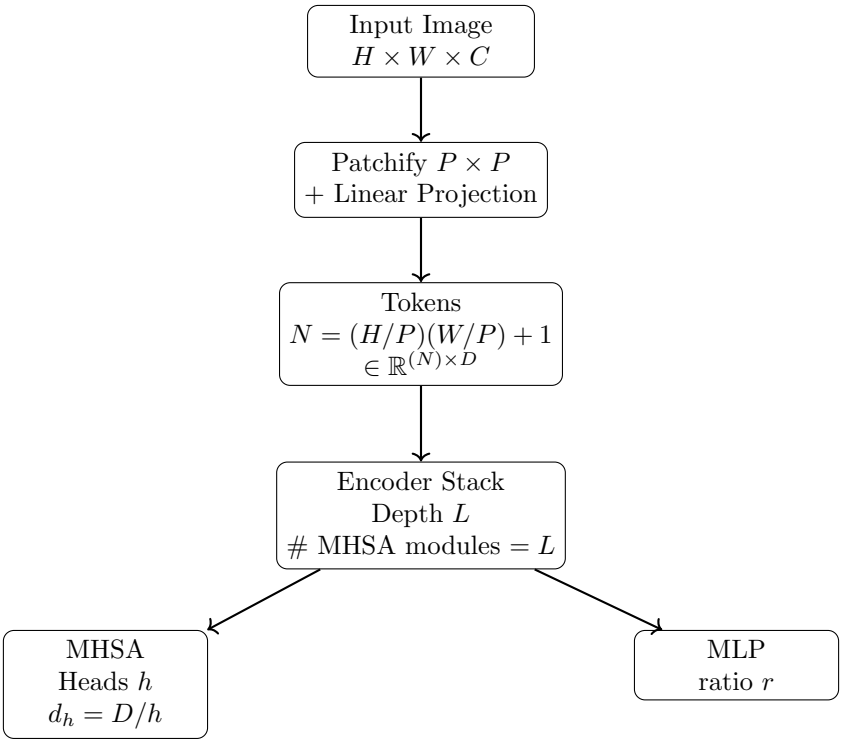

Figure 7.3: Tree-style view of ViT. The encoder stack has depth L; each layer contains one MHSA (therefore total attention modules $= L$) with h heads and an MLP block.

Index