Summary of Activation Functions, Loss Functions, and Optimizers

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

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 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)

1.6 Backpropagation

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

1.7 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:

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

$$\begin{aligned}
\mathbf{z}_1 &= \mathbf{W}_1\mathbf{a}_0 + \mathbf{b}_1, \quad \mathbf{a}_1 = \phi(\mathbf{z}_1),\\
\mathbf{z}_2 &= \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2, \quad \mathbf{a}_2 = \psi(\mathbf{z}_2),\\
z_3 &= \mathbf{W}_3\mathbf{a}_2 + b_3, \quad \hat{y} = \sigma(z_3),
\end{aligned}$$

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:

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

3.5 Numerical Example with Explicit Jacobians

Let $d_0 = d_1 = d_2 = 2$ and choose:

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

Forward propagation yields:

$$\mathbf{z}_1 = \begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix}, \quad \mathbf{a}_1 = \begin{pmatrix} 4.1 \\ 0.3 \end{pmatrix}.$$

Since both components are positive, $\phi'(\mathbf{z}_1) = (1, 1)$ and

$$\mathbf{J}_1 = \mathbf{W}_1.$$

For layer 2:

$$\mathbf{W}_2 = \begin{pmatrix} 0.7 & -1.2 \\ 1.1 & 0.3 \end{pmatrix}, \quad \mathbf{b}_2 = \begin{pmatrix} 0.05 \\ -0.1 \end{pmatrix}.$$

This gives:

$$\mathbf{z}_2 = \begin{pmatrix} 2.56 \\ 4.50 \end{pmatrix}, \quad \mathbf{a}_2 = \begin{pmatrix} 0.988119 \\ 0.999753 \end{pmatrix},$$

$$\psi'(\mathbf{z}_2) = \begin{pmatrix} 0.023620 \\ 0.000494 \end{pmatrix},$$

$$\mathbf{J}_2 = \begin{pmatrix} 0.016535 & -0.028345 \\ 0.000543 & 0.000148 \end{pmatrix}.$$

For the output layer:

$$\mathbf{W}_3 = \begin{pmatrix} 1.4 & -0.8 \end{pmatrix}, \quad b_3 = 0.2,$$

$$z_3 = 0.783564, \quad \hat{y} = 0.686448, \quad \frac{\partial \mathcal{L}}{\partial \hat{y}} = -0.313552.$$

With $\sigma'(z_3) = 0.215271$,

$$\mathbf{J}_3 = \begin{pmatrix} 0.301332 & -0.172190 \end{pmatrix}.$$

Applying Jacobian backpropagation:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = \begin{pmatrix} -0.094506 & 0.053988 \end{pmatrix},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_1} = \begin{pmatrix} -0.001435 & 0.002267 \end{pmatrix},$$

$$\nabla_{\mathbf{x}} \mathcal{L} = \begin{pmatrix} -0.002876 \\ 0.000963 \end{pmatrix}.$$

This result matches exactly the gradient obtained through standard backpropagation.

3.6 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.

3.7 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.

Chapter 4

Vision Transformer (ViT)

4.1 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.

4.2 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:

$$N_{\text{patch}} = \frac{H}{P} \cdot \frac{W}{P} \tag{4.1}$$

A special learnable classification token (`[CLS]`) is prepended to the patch sequence, yielding the total number of tokens:

$$N = N_{\text{patch}} + 1 \tag{4.2}$$

Each token is represented as a vector in $\mathbb{R}^D$, where D is the embedding dimension.

4.3 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:

$$\text{Number of attention modules} = L \tag{4.3}$$

4.4 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:

$$d_h = \frac{D}{h}, \quad \text{with } D \bmod h = 0 \tag{4.4}$$

For each head, attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_h}}\right)V \tag{4.5}$$

The attention matrix for each head has size:

$$N \times N \tag{4.6}$$

This quadratic dependence on the number of tokens is the primary source of memory and computational cost in ViT.

4.5 Computational Scaling

The dominant computational and memory terms in ViT arise from self-attention and MLP layers.

4.5.1 Attention Cost

Self-attention scales as:

$$\mathcal{O}(N^2 \cdot D) \tag{4.7}$$

This quadratic scaling with respect to the number of tokens makes patch size and input resolution critical design parameters.

4.5.2 MLP Cost

Each Transformer layer contains an MLP with hidden dimension:

$$D_{\text{MLP}} = r \cdot D \tag{4.8}$$

where r is the MLP expansion ratio, typically set to 4.

The MLP cost scales as:

$$\mathcal{O}(N \cdot D \cdot D_{\text{MLP}}) \tag{4.9}$$

4.6 Parameter Impact Summary

Table 4.1 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 4.1: Impact of Vision Transformer architectural parameters.

4.7 Numerical Example

Consider an input image of size 224×224 with patch size $P = 16$.

$$N_{\text{patch}} = (224/16)^2 = 14^2 = 196 \tag{4.10}$$

$$N = 196 + 1 = 197 \tag{4.11}$$

For an embedding dimension $D = 192$ and $h = 3$ heads:

$$d_h = 192/3 = 64 \tag{4.12}$$

The attention matrix size per layer has size:

$$h \cdot N^2 = 3 \times 197^2 = 116{,}427 \tag{4.13}$$

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.

4.8 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.

4.9 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 Representati
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

Table 4.2: 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.

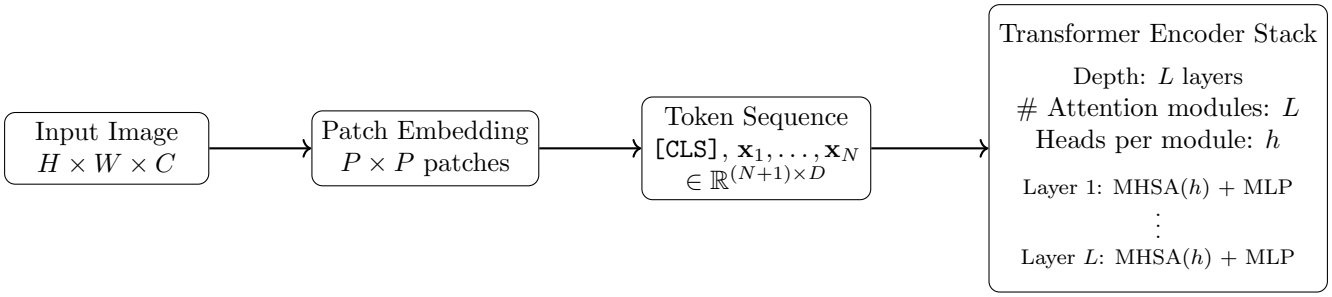

Figure 4.1: High-level architecture of the Vision Transformer (ViT). An input image is split into non-overlapping patches, embedded into a token sequence, and processed by a Transformer encoder stack of depth L. Each encoder layer contains exactly one multi-head self-attention (MHSA) module (so the number of attention modules is L), and each MHSA uses h attention heads.

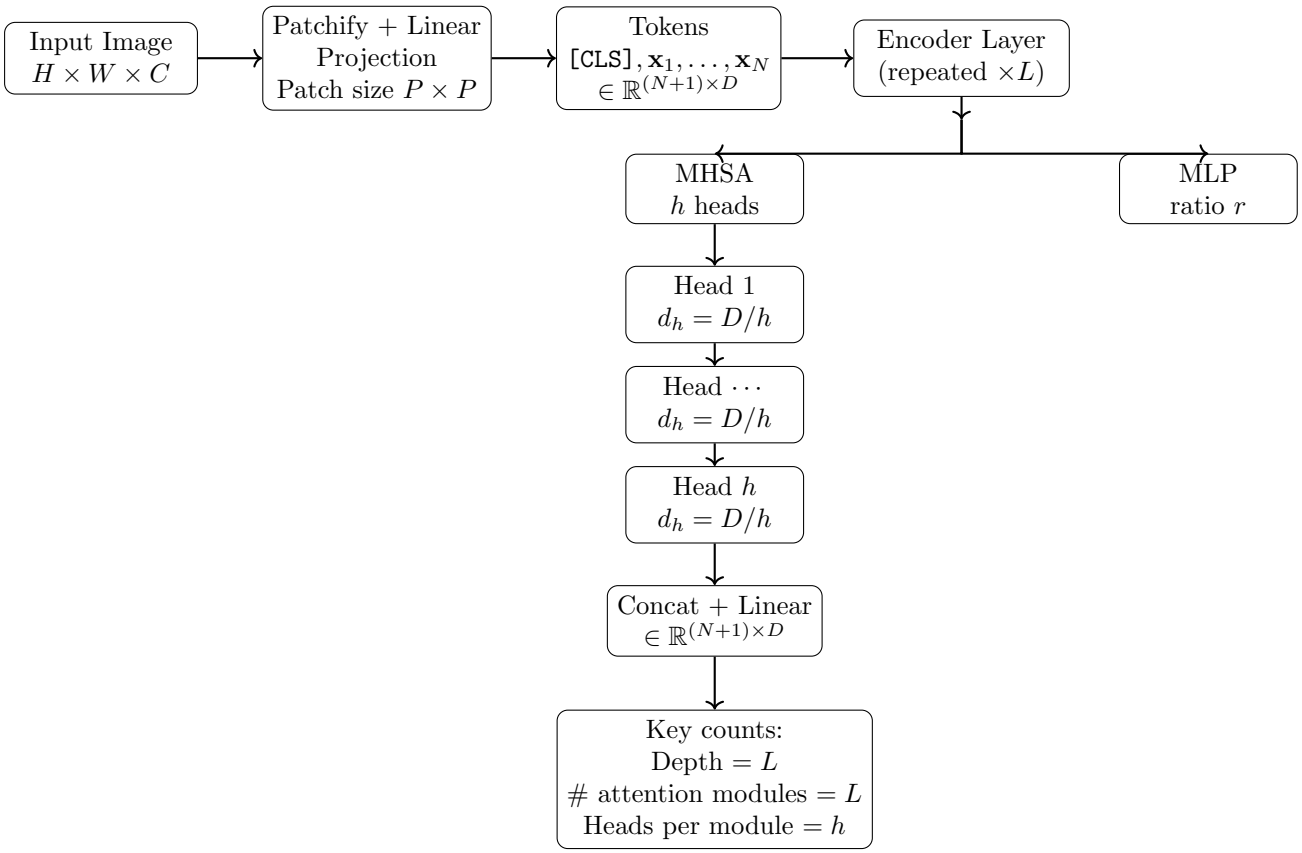

Figure 4.2: Hydra/tree-style view of ViT without overflow. Tokens pass through an encoder layer repeated $\times L$. Each layer contains exactly one MHSA (so the number of attention modules is L) with h heads and one MLP block. The MHSA splits into h heads and merges back via concatenation to dimension D.

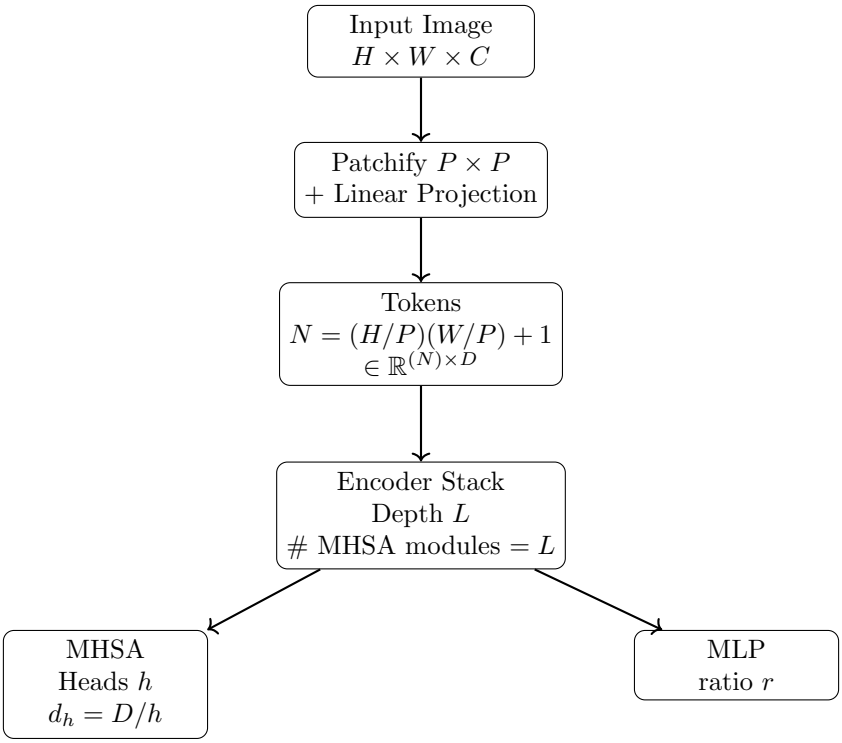

Figure 4.3: Tree-style view of ViT. The encoder stack has depth L; each layer contains one MHSA (therefore total attention modules $= L$) with h heads and an MLP block.

4.10 General Structure of Forward Pass and Backpropagation

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)**).

4.10.1 Forward Pass

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 (or the outputs of the previous layer), adds a bias, and applies an activation function (such as sigmoid, ReLU, or tanh) to produce its output.

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.

4.10.2 Backpropagation

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.

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).

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.

4.10.3 Universality of the Process

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

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

4.10.4 Variations

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

- **Activation Functions:** Modern networks often use ReLU (Rectified Linear Unit) 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 like Adam, RMSProp, or Adagrad, which adaptively adjust the learning rate.

4.10.5 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.
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.

Table 4.3: 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 4.4: 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

Table 4.5: 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

4.10.6 Three-Layer Forward and Backprop Example

Network definition (scalar).

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

$$z_2 = w_2 a_1 + b_2, \quad a_2 = \tanh(z_2),$$

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

Activation derivatives.

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

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 4.6: 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

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 $\mathbb{1}[z_1 > 0]$.

Table 4.7: 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 4.8: 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 4.9: 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

4.10.7 Three-Layer Forward and Backprop Example

Network definition (scalar).

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

$$z_2 = w_2 a_1 + b_2, \quad a_2 = \tanh(z_2),$$

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

Activation derivatives.

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

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 4.10: 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

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 $\mathbb{1}[z_1 > 0]$.

Table 4.11: 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 4.12: 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 4.13: 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

4.10.8 Small Chain-Rule Example and Gradient Accumulation

Computation graph. Consider the following scalar computation:

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

Here, the intermediate variable v_3 feeds into two downstream variables, v_5 and v_6.

Forward pass (numerical example). Let $x = 3$. Then:

$$\begin{aligned} v_3 &= 9, \\ v_5 &= 18, \\ v_6 &= 10, \\ f &= 180. \end{aligned}$$

Backward pass: chain rule with accumulation. To compute the gradient with respect to v_3, note that f depends on v_3 through two distinct paths. The multivariable chain rule gives:

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

Evaluate each term.

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

Gradient accumulation. Substituting:

$$\frac{\partial f}{\partial v_3} = (18)(1) + (10)(2) = 18 + 20 = \boxed{38}.$$

Interpretation. The summation in the chain rule corresponds directly to gradient accumulation in reverse-mode automatic differentiation. When a node feeds multiple children, each child contributes a partial gradient, and the total gradient is obtained by summing all contributions:

$$\frac{\partial f}{\partial v_3} \longleftrightarrow \texttt{grad[v3] += contribution from each child}.$$

Key point. The "+" operator in the chain rule corresponds to the "`+=`" operation in backpropagation implementations: gradients are accumulated whenever a variable influences the output through multiple paths.

4.10.9 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. The symbol $\odot$ denotes the *Hadamard (element-wise) product* and is *not* the Khatri–Rao product.

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. The network employs different activation functions at each layer:

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

For a single training sample, the loss is

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

Activation derivatives (element-wise). All activation derivatives are applied element-wise:

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

For vectors, we use the Hadamard product $\odot$ and element-wise powers such as $\mathbf{a}^{\odot 2} = \mathbf{a} \odot \mathbf{a}$.

Forward propagation (matrix form).

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1, \qquad \mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1), \tag{4.14}$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2, \qquad \mathbf{a}_2 = \tanh(\mathbf{z}_2), \tag{4.15}$$

$$\mathbf{z}_3 = \mathbf{W}_3\mathbf{a}_2 + \mathbf{b}_3, \qquad \hat{\mathbf{y}} = \sigma(\mathbf{z}_3). \tag{4.16}$$

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).

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

where $\odot$ is a **Hadamard (element-wise) product**.

Layer 2 (tanh).

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

Here, $\mathbf{W}_3^\top \boldsymbol{\delta}_3$ is a **matrix–vector product**.

Layer 1 (ReLU).

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

where $\mathbb{1}[\mathbf{z}_1 > 0]$ is an element-wise binary mask.

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**, yielding a matrix in $\mathbb{R}^{d_k \times d_{k-1}}$. It is **not** a Hadamard, Kronecker, or Khatri–Rao product.

Forward-pass memory (saved tensors). During the forward pass, certain intermediate quantities must be retained in memory because they are required to compute gradients during backpropagation. Automatic differentiation frameworks store these values as part of the computational graph.

Table 4.14: 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	$\hat{\mathbf{y}} - \mathbf{y}$

Table 4.15: 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. Remark. Standard backpropagation for feedforward neural networks requires only matrix–vector products, Hadamard products, and outer products. Higher-order tensor products such as the Kronecker or Khatri–Rao products do not appear.

4.10.10 Hessian–Vector Products and the Role of the Nabla Operator

Let $f(\mathbf{w})$ denote a scalar-valued objective function, where $\mathbf{w} \in \mathbb{R}^W$ is the vector of model parameters (e.g., weights and biases of a neural network). The *gradient* of f with respect to $\mathbf{w}$ is denoted by $\nabla f(\mathbf{w}) \in \mathbb{R}^W$, and the *Hessian* is the matrix of second-order partial derivatives

$$\mathbf{H} = \nabla^2 f(\mathbf{w}) \in \mathbb{R}^{W \times W}, \qquad H_{ij} = \frac{\partial^2 f}{\partial w_i \, \partial w_j}.$$

Forming the full Hessian matrix explicitly is typically computationally prohibitive for large-scale models. However, many second-order methods require only the *Hessian–vector product*, defined as

$$\mathbf{H}\mathbf{u} = \nabla^2 f(\mathbf{w})\, \mathbf{u},$$

where $\mathbf{u} \in \mathbb{R}^W$ is an arbitrary vector in parameter space. The result $\mathbf{H}\mathbf{u}$ is itself a vector in $\mathbb{R}^W$ and represents the action of the curvature of f along the direction $\mathbf{u}$.

A key identity enables efficient computation of this product without explicitly constructing the Hessian:

$$\mathbf{H}\mathbf{u} = \nabla_{\mathbf{w}} \Big(\nabla f(\mathbf{w})^\top \mathbf{u} \Big).$$

This identity can be interpreted as follows:

1. Compute the gradient $\nabla f(\mathbf{w})$ with respect to the parameters.
2. Form a scalar by taking its inner product with $\mathbf{u}$.
3. Differentiate this scalar once more with respect to $\mathbf{w}$.

The nabla operator $\nabla_{\mathbf{w}}$ therefore always denotes partial derivatives with respect to the parameter vector $\mathbf{w}$. In this context, the first application of $\nabla_{\mathbf{w}}$ produces the gradient, while the second application produces second-order curvature information. Automatic differentiation systems can evaluate the above expression efficiently, yielding Hessian–vector products in time comparable to a small constant multiple of a gradient computation.

This capability is fundamental to Hessian-free optimization, conjugate gradient methods, and other second-order–inspired algorithms used in large-scale machine learning.

Worked Examples and Interpretation

To clarify the meaning of the Hessian–vector product $\mathbf{H}\mathbf{u} = \nabla^2 f(\mathbf{w})\, \mathbf{u}$ and the role of the nabla operator, we present concrete examples.

Example 1: Simple quadratic function. Consider the scalar function

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

The gradient of f with respect to $\mathbf{w} = (w_1, w_2)^\top$ is

$$\nabla f(\mathbf{w}) = \begin{pmatrix} 2w_1 + 3w_2 \\ 3w_1 + 4w_2 \end{pmatrix}.$$

Applying the nabla operator once more yields the Hessian

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

Let $\mathbf{u} = (1, -1)^\top$ be an arbitrary direction in parameter space. The Hessian–vector product is then

$$\mathbf{Hu} = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \end{pmatrix}.$$

This result represents how the gradient of f changes when the parameters are perturbed in the direction $\mathbf{u}$.

Verification via the nabla identity. Using the identity

$$\mathbf{Hu} = \nabla_{\mathbf{w}} \Big(\nabla f(\mathbf{w})^\top \mathbf{u} \Big),$$

we first compute the scalar

$$\nabla f(\mathbf{w})^\top \mathbf{u} = (2w_1 + 3w_2,\, 3w_1 + 4w_2) \begin{pmatrix} 1 \\ -1 \end{pmatrix} = -(w_1 + w_2).$$

Differentiating this scalar with respect to $\mathbf{w}$ gives

$$\nabla_{\mathbf{w}}(-(w_1 + w_2)) = \begin{pmatrix} -1 \\ -1 \end{pmatrix},$$

which matches the Hessian–vector product obtained above.

Example 2: Interpretation in neural networks. In neural networks, the parameter vector $\mathbf{w}$ collects all weights and biases, and $f(\mathbf{w}) = \mathcal{L}$ denotes the loss. The gradient $\nabla f(\mathbf{w})$ is computed via backpropagation. For any chosen direction $\mathbf{u}$ in parameter space, the Hessian–vector product $\mathbf{Hu}$ measures how this gradient changes along $\mathbf{u}$.

Crucially, the identity

$$\mathbf{Hu} = \nabla_{\mathbf{w}} \Big(\nabla f(\mathbf{w})^\top \mathbf{u} \Big)$$

allows the Hessian–vector product to be computed using automatic differentiation without explicitly forming the Hessian matrix. This enables curvature-aware optimization methods to scale to large models.

Practical Example: Hessian–Vector Product via Forward and Backpropagation

We now present a practical example demonstrating how a Hessian–vector product is computed using standard forward and backward propagation in a neural network, without explicitly forming the Hessian matrix.

Model and loss. Consider a single-neuron model with one input:

$$z = wx, \qquad \hat{y} = z, \qquad \mathcal{L} = \tfrac{1}{2}(\hat{y} - y)^2,$$

where w is the trainable parameter, x is the input, and y is the target value. The parameter vector is $\mathbf{w} = (w)$.

Forward pass. The forward pass computes:

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

First backward pass (gradient computation). Backpropagation computes the gradient of the loss with respect to w:

$$\nabla f(w) = \frac{\partial \mathcal{L}}{\partial w} = (wx - y)x.$$

This is the standard gradient used in first-order optimization methods such as gradient descent.

Forming the scalar for the Hessian–vector product. Let u be a chosen direction in parameter space (a scalar in this example). We form the scalar

$$s(w) = \nabla f(w)^\top u = (wx - y)x\, u.$$

Second backward pass (Hessian–vector product). We now differentiate $s(w)$ with respect to w:

$$\nabla_w s(w) = \frac{d}{dw}\big((wx - y)x\, u\big) = x^2 u.$$

Since the second derivative of the loss is

$$\nabla^2 f(w) = x^2,$$

the above result is exactly the Hessian–vector product:

$$\nabla_w s(w) = \nabla^2 f(w)\, u.$$

Interpretation in terms of backpropagation. This computation uses:

- one forward pass to compute $\mathcal{L}$,
- one backward pass to compute $\nabla f(w)$,
- a second backward pass to compute $\nabla^2 f(w)\, u$.

At no point is the Hessian matrix explicitly constructed. Instead, the Hessian–vector product is obtained by reusing the standard backpropagation machinery on a scalar quantity. This is precisely how modern deep learning frameworks compute second-order information in practice.

Extension to deep networks. In multilayer neural networks, the same procedure applies with $\mathbf{w}$ representing all weights and biases. The two backward passes correspond to backpropagation through the original loss and backpropagation through a scalar derived from the gradient. This enables Hessian-free optimization and curvature-aware methods to scale to large models.

Practical Example: Hessian–Vector Product in a Three-Layer Network

We now present a practical example demonstrating how Hessian–vector products are computed in a three-layer neural network using standard forward and backward propagation, including both weights and biases.

Model and loss. Consider a scalar three-layer network

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

where x is the input, y is the target, and the parameter vector is

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

Identity activations are used to isolate the propagation mechanism.

—

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

$$\nabla f(\boldsymbol{\theta}) = (\hat{y} - y) \begin{pmatrix} w_3w_2x \\ w_3w_2 \\ w_3a_1 \\ w_3 \\ a_2 \\ 1 \end{pmatrix}.$$

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

$$s(\boldsymbol{\theta}) = \nabla f(\boldsymbol{\theta})^\top \mathbf{u} = (\hat{y} - y)\Big(u_{w_1}w_3w_2x + u_{b_1}w_3w_2 + u_{w_2}w_3a_1 + u_{b_2}w_3 + u_{w_3}a_2 + u_{b_3}\Big).$$

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 $\mathbf{H}\mathbf{u} = \nabla^2 f(\boldsymbol{\theta})\, \mathbf{u}$.
  - The direction $\mathbf{u}$ is chosen externally (e.g., the gradient direction or a random vector).
  - Quantities such as $\mathbf{u}^\top \mathbf{H}\mathbf{u}$ 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
    $$\mathbf{Hp} = -\nabla f(\boldsymbol{\theta}).$$
  - *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 $\mathbf{Hu}$.
  - Computing $\mathbf{Hu}$ 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

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

    using conjugate gradient iterations.
  - Each CG iteration requires Hessian–vector products $\mathbf{Hu}$ 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.

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 in the graph stores the local derivative information required to apply the chain rule during backpropagation, along with references to its parent tensors. During the backward

Table 4.16: 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()`

pass, PyTorch traverses this graph in reverse to compute gradients of the loss with respect to model parameters.

During inference, the computational graph is not constructed. Tensor operations are executed without storing dependency information or intermediate activations, which reduces memory usage and improves runtime performance.

Index