Deep Learning (2): Backpropagation and Training in Practice

(for high school and first-year university students)

How to find the weights from data

Planned and edited by Yoshio Okamoto / Written by Claude Opus 4.5 (Anthropic)

Ver. 0.2 October 7, 2026

Note: The following content was generated by AI. The editor has checked it, but please be aware that it may still contain errors or mistaken assumptions.

© 2026 Yoshio Okamoto. Free to use for educational purposes (please cite the source).

Introduction

In Part 1, we saw that a neural network is a function that repeats "take a weighted sum, add a bias, pass through a nonlinear activation function." There, a person decided the weights, but real networks have tens of thousands to billions of weights, which no person can set.

> Part 2 deals with **how to learn the weights from data**. The idea is the same as before: "minimize the loss function by gradient descent." The problem is **how to compute the gradient efficiently for an enormous number of weights**. The answer is **backpropagation**.
> In the terms of Part 1, the network has tens of thousands of "knobs." Backpropagation is the method that tells us, all at once, which knob to turn in which direction and by how much, while watching how far off the output is (the loss).

Part	Title
Part 1	Basics of Neural Networks
Part 2	**Backpropagation and Training in Practice**
Part 3	Convolutional Neural Networks
Part 4	Sequence Data and Transformers

How to read this booklet The main text is written so that it can be read with high-school mathematics (differentiation of composite functions, exponential functions, the summation sign $\sum$). In a few places it goes slightly beyond high school (partial derivatives), but an explanation in words is given on the spot. The general notation with matrices and vectors is collected in the **footnotes** and in **Appendix A** at the end. You can follow the main text even if you skip the appendix. All numerical values in the examples were actually computed (programs in Appendix B).

Notation

- $\dfrac{\partial L}{\partial w}$ is a **partial derivative**. It expresses the rate of change of L when only w is moved a little while all the other parameters are held fixed. Think of it as "the slope of L in the w direction." The partial derivatives with respect to all the parameters together are called the **gradient**.

- Numbers written as superscripts **in parentheses**, as in $W^{(1)}$ and $\delta^{(2)}$, are **layer numbers**, not powers.
- $\sigma(z) = 1/(1+e^{-z})$ is the sigmoid function, and its derivative is $\sigma'(z) = \sigma(z)\big(1-\sigma(z)\big)$. η (eta) is the learning rate.

1 Why an efficient way of computing gradients is needed

1.1 The naive method: numerical differentiation

The most naive way to find a gradient is **numerical differentiation**: move each weight a tiny bit and measure the change in the loss. We move a parameter w back and forth by a small amount ε (epsilon) and take

$$\frac{\partial L}{\partial w} \approx \frac{L(w+\varepsilon) - L(w-\varepsilon)}{2\varepsilon}$$

(with the other parameters held fixed). This is the very definition of the derivative. We turn just one knob a little and see how much the loss meter moves.

> With this method, each parameter requires two computations of the loss (forward passes). With P parameters, one gradient computation requires $2P$ forward passes. For the handwritten-digit example of Part 1 ($P \approx 80{,}000$), one update needs 160,000 forward passes; for an LLM with 8 billion parameters, 16 billion. **If we tried turning 80,000 knobs one at a time, we would never finish.**

1.2 The efficiency of backpropagation

With backpropagation, **the gradients with respect to all the parameters are obtained by one backward computation, costing about the same as one forward pass**. Regardless of the number of parameters, the computation is only about 2–3 times that of a forward pass.

> When a company misses its sales target, it would be hard work for the president to ask every employee one by one. Instead, the president tells each department head "your department's share of the responsibility is this much," each department head tells each section chief, each section chief tells each staff member, and so on. **If responsibility is distributed from the top down, everyone learns the size of their own responsibility in a single pass.** Backpropagation distributes the "miss" (the error) of the output in this way, from the output side to the input side.

Below, we look at how this works.

2 The chain rule and computation graphs

2.1 Differentiating composite functions (the chain rule)

When $y = f(u)$ and $u = g(x)$,

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

(differentiation of composite functions). This is called the **chain rule**. It means: "when x changes a little, u changes $\frac{du}{dx}$ times as much, and when u changes a little, y changes $\frac{dy}{du}$ times as much, so overall the factors multiply."

Picture **gears meshed together**. If turning gear A once turns B three times, and turning B once turns C twice, then turning A once turns C $3 \times 2 = 6$ times. Multiplying the ratios of the gears along the way gives the overall ratio: this is the chain rule.

The same thinking applies when there are many variables. If L depends on x through intermediate quantities $u_1, \ldots, u_m$, then

$$\frac{\partial L}{\partial x} = \sum_{k=1}^{m} \frac{\partial L}{\partial u_k} \cdot \frac{\partial u_k}{\partial x}.$$

There are m "paths" from x to L, and the influence travels along each, so we add up the influence of each path. It is like a river that splits into several branches and joins again downstream: the amount of water downstream is the total of the water in the branches.

It is enough to remember: **multiply the derivatives along each path, and add over all paths**.

2.2 Computation graphs

As a simple example, consider $f = (x+y)\,z$. Writing the intermediate value as $q = x+y$, we have $f = q\,z$. A diagram of the flow of this computation is called a **computation graph** (Figure 1).

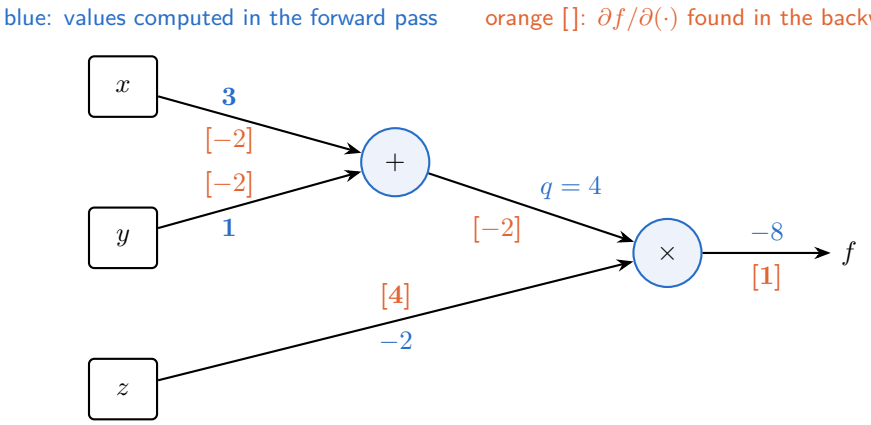


Figure 1 The computation graph of $f = (x+y)z$. For $x = 3$, $y = 1$, $z = -2$, the blue numbers are the values obtained by computing from left to right (forward pass), and the orange [ ] are the partial derivatives $\partial f/\partial(\cdot)$ obtained by computing from right to left (backward pass).

Forward pass (left to right) $q = 3 + 1 = 4$, $f = 4 \times (-2) = -8$.

Backward pass (right to left) Going from the output to the inputs, we find $\partial f/\partial(\cdot)$ at each point.

1. The starting point is $\partial f/\partial f = 1$.
2. For the multiplication $f = qz$, changing q a little changes f by z times as much, so $\partial f/\partial q = z = -2$. Similarly, $\partial f/\partial z = q = 4$.
3. For the addition $q = x + y$, $\partial q/\partial x = \partial q/\partial y = 1$, so by the chain rule $\partial f/\partial x = \partial f/\partial q \times 1 = -2$ and $\partial f/\partial y = -2$.

(Check: increasing x from 3 to 3.01 gives $f = (4.01)(-2) = -8.02$, a change of -2 times 0.01.)

What matters here is that **each node only has to receive "the derivative with respect to its own output" from behind, multiply it by its own local derivative, and pass it forward**.

- An addition node passes the value it receives unchanged to both inputs.
- A multiplication node passes the value it receives multiplied by "the other input's value."

It is like a bucket brigade: each person just takes what the neighbor hands over, processes it a little, and passes it on. Nobody needs to see the whole picture.

However complicated a computation is, it is a combination of simple nodes, so applying this local rule in order from the back gives the derivatives with respect to all the inputs in one backward computation. This is the essence of backpropagation. It is also important that intermediate values (here $q = 4$) are recorded during the forward pass and reused in the backward pass.

3 Backpropagation in a neural network

3.1 A two-layer network

Consider a network with 2 inputs, 2 hidden units and 1 output, the same shape as the XOR network of Part 1. All the activation functions are sigmoid functions.

- Hidden layer ($k = 1, 2$): $z_k^{(1)} = W_{k1}^{(1)} x_1 + W_{k2}^{(1)} x_2 + b_k^{(1)}, \quad h_k = \sigma\big(z_k^{(1)}\big)$
- Output layer: $z^{(2)} = W_1^{(2)} h_1 + W_2^{(2)} h_2 + b^{(2)}, \quad \hat{y} = \sigma\big(z^{(2)}\big)$

$W_{kj}^{(1)}$ is "the weight from input j to hidden unit k," and $W_k^{(2)}$ is "the weight from hidden unit k to the output." There are 9 parameters: 6 weights and 3 biases. The loss is the same cross-entropy as in the logistic regression of the previous document,

$$L = -\big[\, y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \,\big]$$

(y is the correct answer, 0 or 1).

3.2 The backpropagation formulas

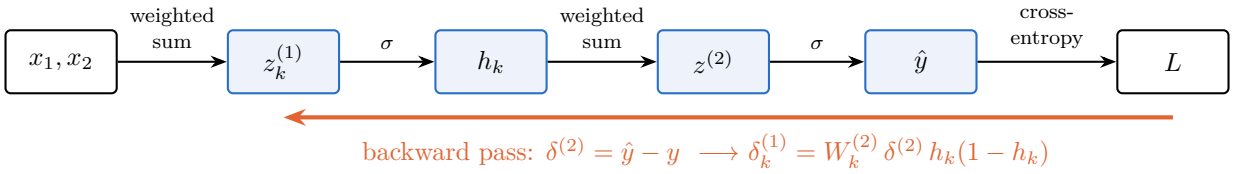


Figure 2 Forward pass (black arrows, left to right) and backward pass (orange arrow, right to left) of the two-layer network.

We write the partial derivative with respect to z of each layer as δ (delta) and call it the **error**. In the analogy above, it corresponds to "the size of the responsibility" assigned to each department. We find them in order from the output side.

Output layer By the same computation as in the previous document (the gradient of logistic regression),[1]

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = \hat{y} - y.$$

[1] Section 4.5 and Appendix B.3 of the previous document, "Supervised and Unsupervised Learning." When the sigmoid function and cross-entropy are combined, $\hat{y}(1-\hat{y})$ cancels in the middle of the chain rule, giving this simple form.

This is "the gap between prediction and correct answer" itself. Since $z^{(2)} = W_1^{(2)} h_1 + W_2^{(2)} h_2 + b^{(2)}$, changing $W_k^{(2)}$ a little changes $z^{(2)}$ by h_k times as much. By the chain rule, the gradients of the output-layer weights and bias are

$$\frac{\partial L}{\partial W_k^{(2)}} = \delta^{(2)}\, h_k, \qquad \frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}.$$

Hidden layer We send the error back to the previous layer. Changing h_k a little changes $z^{(2)}$ by $W_k^{(2)}$ times as much, and changing $z_k^{(1)}$ a little changes h_k by $\sigma'(z_k^{(1)}) = h_k(1 - h_k)$ times as much. Multiplying by the chain rule,

$$\delta_k^{(1)} = \frac{\partial L}{\partial z_k^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k^{(1)}} = \delta^{(2)}\, W_k^{(2)}\, h_k(1 - h_k).$$

From this, the gradients of the hidden-layer weights and biases are found in the same form as for the output layer:

$$\frac{\partial L}{\partial W_{kj}^{(1)}} = \delta_k^{(1)}\, x_j, \qquad \frac{\partial L}{\partial b_k^{(1)}} = \delta_k^{(1)}.$$

In every layer, the gradient has the following form:

(gradient of a weight) = (the error δ that arrived from behind at the weight's destination) × (the value that came into the weight).

And the error δ is sent from the output side to the input side in order, by **multiplying the error of the layer behind by the connecting weight and by the derivative of the activation function**.
In the responsibility analogy, **a subordinate connected by a large weight is given a larger share of the responsibility, and a weight that sent in a large value is corrected more**. The more a place influenced the result, the more it is fixed.

3.3 General multilayer networks

With more layers or more hidden units, the form is exactly the same. When a hidden unit connects to several destinations, following "add over all paths" of the chain rule, we add up the "error × weight" arriving from each destination, and then multiply by the derivative of the activation function. A person given responsibility by several bosses takes on the total.

In the forward pass, values are sent from input to output; in the backward pass, the same weights are used to send errors from output to input. What each layer needs is only the values that came into that layer and the values of z in that layer, and these are recorded during the forward pass. At university, this is written concisely with matrices (Appendix A).

3.4 Gradient check

Backpropagation programs are easy to get wrong, so the standard practice is to **check them against numerical differentiation** on a small network (a **gradient check**). It is the same as "checking your answer" by solving a problem a different way and comparing. With the program in Appendix B.2, comparing all 9 parameters of the XOR network gives differences of about 10^{-11}, confirming that the backpropagation formulas are correct.

Parameter	Backpropagation	Numerical	Difference
$W_{11}^{(1)}$	-0.00844318	-0.00844318	1.4×10^{-12}
$W_{1}^{(2)}$	-0.19033744	-0.19033744	9.2×10^{-12}
$b^{(2)}$	-0.38068435	-0.38068435	1.1×10^{-11}

(3 of the 9 parameters)

4 Example: learning XOR

4.1 Setup

This time we let the network learn from data the XOR whose weights a person decided in Part 1 (program in Appendix B.1).

- Network: 2 inputs, 2 hidden units, 1 output; all activation functions are sigmoid functions
- Initial values: weights are uniform random numbers from -1 to 1; biases are 0
- Loss: the sum of the cross-entropies of the 4 data points
- Update: gradient descent, learning rate $\eta = 1.0$, 5000 times

4.2 Results: cases that work and cases that do not

Changing the random-number seed, we trained from two sets of initial values, A and B (Figure 3).

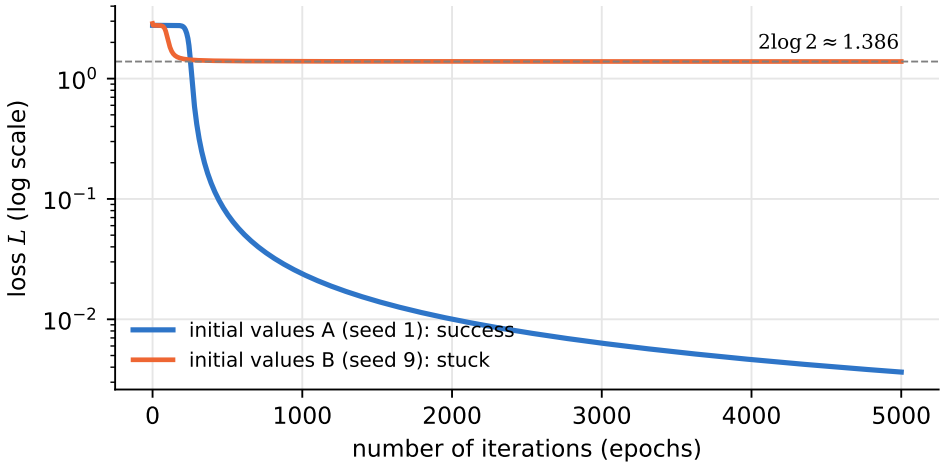


Figure 3 Change of the loss (log-scale vertical axis). With initial values A, the loss keeps decreasing toward 0; with initial values B, it stops near $2\log 2 \approx 1.386$.

Iterations	0	100	500	1000	2000	5000
Loss with initial values A	2.774	2.772	0.075	0.024	0.010	0.004
Loss with initial values B	2.874	2.169	1.402	1.393	1.389	1.387

The outputs after training are as follows.

Input	Correct answer	Output with A	Output with B
(0, 0)	0	0.001	0.000
(0, 1)	1	0.999	1.000
(1, 0)	1	0.999	0.500
(1, 1)	0	0.001	0.500

Figure 4 shows the output of the network at each point of the input plane as shading.

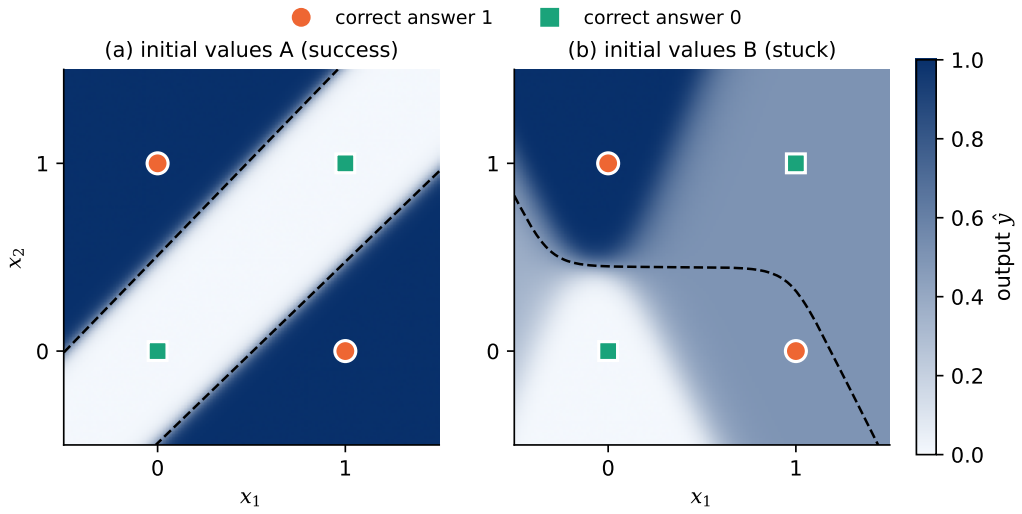


Figure 4 Output $\hat{y}$ of the trained networks (darker is closer to 1). The dashed lines are the boundary $\hat{y} = 0.5$. In (a), XOR is correctly separated by two boundary lines. In (b), the two points on the right cannot be separated, and 0.5 is output for both.

Initial values A For the first 200 or so iterations there is a "plateau" where the loss hardly decreases, but then it drops suddenly and the network learns XOR correctly. As in Figure 3 of Part 1, two straight lines divide the plane into three bands.

Looking at the hidden-layer outputs (h_1, h_2), inputs $(0, 0)$ and $(1, 1)$ are both moved to (0.99, 0.99), input $(0, 1)$ to (1.00, 0.01), and input $(1, 0)$ to (0.01, 1.00).

This transformation differs from the OR and NAND designed by a person in Part 1, but it too **gathers the points whose correct answer is 0 into one place, giving an arrangement that a straight line can separate**. In the analogy of Part 1, the paper was folded in a way different from the person's. The network found its own "solution," different from a person's.

Initial values B The loss stops near $2 \log 2 \approx 1.386$. This is the state in which 0.5 is output for two of the inputs (a loss of $\log 2$ each) and the other two are predicted correctly.[2] The network is caught at a place where the gradient is almost 0 and cannot get out.

4.3 The landscape of the loss function

Gradient descent is like **walking down a mountain in thick fog**. You cannot see far; all you know is the slope at your feet, so you just keep taking steps downhill.
The loss of linear regression or logistic regression had a bowl-like shape (a convex function) with a single valley bottom. Wherever you start, you reach the same bottom. But the loss of a network with hidden layers is not convex: **there may be several hollows (valley bottoms) and flat places (passes and plateaus)**. Where you end up depends on where you start.

[2] Giving probability 0.5 to the correct answer makes the loss of that data point $-\log 0.5 = \log 2 \approx 0.693$. This happens for two data points, giving $2 \log 2$.

A **saddle point** is a point that, like the shape of a horse's saddle, is a valley bottom seen in one direction and a summit seen in another. In landscape terms, it corresponds to a **mountain pass**. A pass is the lowest point seen along the ridge and the highest point seen along the valley. The slope there is 0, so gradient descent tends to stop moving.

In this example, increasing the hidden units from 2 to 3 gave 0 failures out of 100 different initial values (with 2, there were 14 failures out of 100). It is known from experience that adding units makes the loss landscape gentler and makes it easier to reach a good solution. In real deep learning, it is known that by making the network large enough and combining the techniques described later, this problem is not very serious.

5 Stochastic gradient descent

5.1 Mini-batches

So far, we computed the gradient with the loss taken as **the sum (mean) over all data**. But with millions of data points, each update requires forward and backward passes over all the data, which takes time.

So each time, a small number of data points (for example, 32) are chosen at random, and the gradient is computed and the update made with only this **mini-batch**. For each parameter w,

$$w \leftarrow w - \eta \times \big(\text{the mean of the gradients } \partial \ell_i / \partial w \text{ over the data in the mini-batch}\big)$$

(where ℓ_i is the loss of a single data point i).[3] This is called **stochastic gradient descent** (SGD). Going through all the data once is called **one epoch**.

> This is the same idea as an **opinion poll**. Without asking the whole nation, asking a randomly chosen part gives a rough idea of the overall trend. The mini-batch gradient is an "estimate" of the gradient over all the data; it is a little off each time, but on average it points in the right direction.

So the update direction wobbles each time, but overall it approaches the minimum (Figure 5).

As in Figure 5(b), the path of stochastic gradient descent zigzags and keeps wandering even near the minimum.

> Even so, **since each update needs little computation, far more updates can be made in the same computing time**. It is also thought that this wandering helps the parameters escape from shallow hollows and passes.

5.2 Improved update methods

Several improvements on basic SGD are in use. Here are typical ones.

- **Momentum**: the update uses a moving average v of past gradients. For each parameter w,

  $$v \leftarrow \beta v + \frac{\partial L}{\partial w}, \qquad w \leftarrow w - \eta\, v \qquad (\beta \approx 0.9).$$

  Like the inertia of a ball rolling down a slope, it cancels zigzags and speeds up movement in a steady direction. Small dips can even be passed over by momentum.

[3] University textbooks write this, with $\boldsymbol{\theta}$ the vector of all parameters, B the mini-batch and $|B|$ its size, as $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{1}{|B|} \sum_{i \in B} \nabla_{\boldsymbol{\theta}} \ell_i$. $\nabla_{\boldsymbol{\theta}} \ell_i$ is the gradient of ℓ_i (the vector of partial derivatives with respect to each parameter).

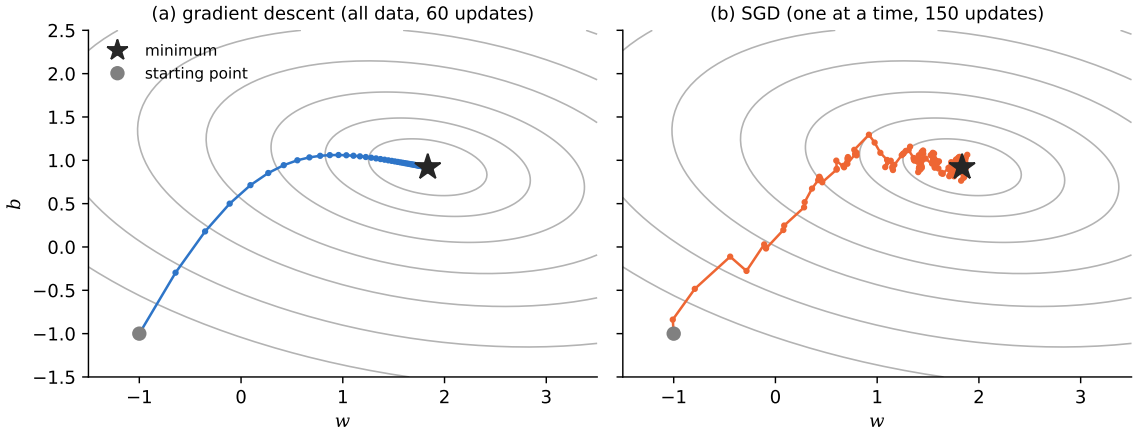


Figure 5 Movement of the parameters in fitting a straight line ($y = wx + b$, 50 data points) with (a) gradient descent and (b) stochastic gradient descent (one data point at a time, 3 epochs). The gray lines are contours of the loss, and the star is the minimum.

- **Adam**: in addition to the moving average of the gradient, it records the moving average of the squared gradient and automatically adjusts the learning rate for each parameter. It is one of the most widely used methods today.[4]

6 The vanishing gradient problem

6.1 What happens in deep networks

Consider a long, thin network with only one unit in each layer:

$$x \xrightarrow{w_1} h_1 \xrightarrow{w_2} h_2 \xrightarrow{w_3} \cdots \xrightarrow{w_L} h_L \longrightarrow L.$$

Each layer computes $h_l = f(z_l)$, $z_l = w_l h_{l-1} + b_l$. Tracing backward with the chain rule as in Section 3.2, the error in the layer nearest the input is

$$\delta_1 = f'(z_1) \cdot w_2 \, f'(z_2) \cdot w_3 \, f'(z_3) \cdots w_L \, f'(z_L) \cdot \frac{\partial L}{\partial h_L},$$

a product in which the derivative f' of each layer's activation function is multiplied many times (for the general case with many units, see Appendix A.4). In the gear analogy, if you connect many gears each with a ratio less than 1, the last gear hardly turns.

> The maximum of the derivative of the sigmoid function, $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$, is 0.25 at $z = 0$ (Figure 6(a)).[5] Multiplying this 10 times gives $0.25^{10} \approx 10^{-6}$, so almost no error reaches the layers near the input. As a result, **the weights of the layers near the input are hardly updated, and training does not progress**. This is called the **vanishing gradient problem**.
>
> In a game of **telephone** (whispering a message down a line), if the voice became one quarter as loud with each person, the tenth person would hear nothing at all. The layers near the input cannot receive the news of "how to fix things."

[4] How Adam works and its formulas are explained in Section 4.4 and Appendix A.6 of the separate document "Reading microgpt."

[5] Writing $s = \sigma(z)$, we have $0 < s < 1$ and $s(1-s) = \frac{1}{4} - \big(s - \frac{1}{2}\big)^2 \le \frac{1}{4}$, with equality when $s = \frac{1}{2}$, that is, $z = 0$.

Conversely, if the factors are larger than 1, the gradient can grow explosively (**exploding gradients**). This is the telephone game in which the voice keeps getting louder until it becomes a shout.

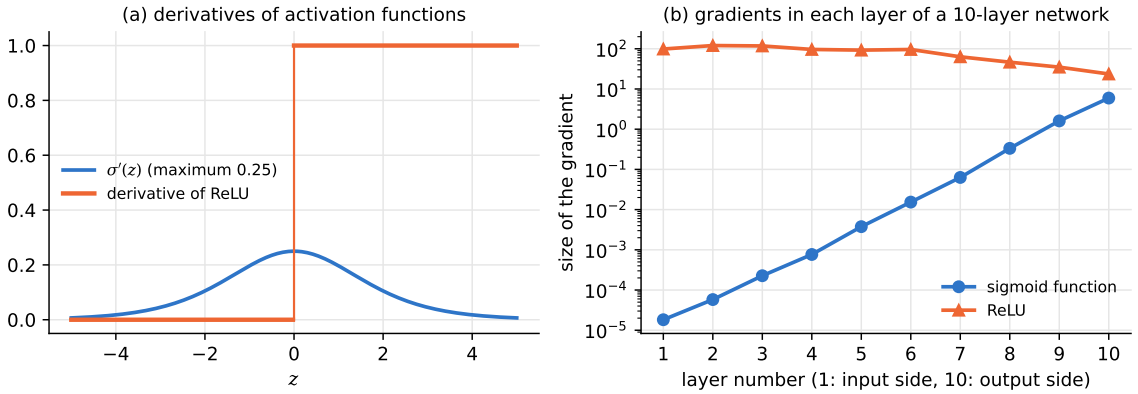


Figure 6 (a) Derivatives of the sigmoid function and ReLU. (b) Size of the gradient in each layer of a network with 10 layers (50 units each), when backpropagation is done with random weights (log-scale vertical axis).

Figure 6(b) compares the sizes of the weight gradients in each layer when backpropagation is actually carried out in a 10-layer network. With the sigmoid function, the gradient becomes a fraction of its size with each layer from the output side (layer 10) toward the input side (layer 1), and in layer 1 it is about one three-millionth of that in layer 10.

6.2 Countermeasures

- **Use ReLU**: the derivative of ReLU is always 1 for $z > 0$, so it does not shrink when multiplied (a message whose loudness does not change). In Figure 6(b), too, with ReLU the gradients of all the layers stay at about the same size. This is a major reason why ReLU came to be widely used.
- **Careful initial values**: so that the spread of each layer's output neither grows nor shrinks as it passes through the layers, the variance of the initial weights is chosen according to the number n of inputs to the unit. For ReLU, variance $2/n$ (He initialization), and for the sigmoid function and tanh, variance about $1/n$ (Xavier initialization), are commonly used.[6] It is like adjusting each person's voice to just the right loudness at the start of the message.
- **Residual connections**: we add a "shortcut" that adds the input of a layer directly to its output, as in $h^{(l)} = h^{(l-1)} + F\big(h^{(l-1)}\big)$. In backpropagation, the error passes along this shortcut directly to the input side,[7] so networks of more than 100 layers became trainable (they are also used in the Transformer of Part 4). It is like running **a telephone line directly from the first person to the last** alongside the line of people in the telephone game.
- **Normalization layers**: layers that bring each layer's outputs to a mean of 0 and a variance of about 1 (batch normalization, layer normalization) are inserted, preventing the size of

[6] With n inputs, the weighted sum is a sum of n terms, so its spread (variance) is roughly "variance of the weights $\times$ n $\times$ mean of the squared inputs." Making the variance of the weights about $1/n$ keeps the spread of the output the same regardless of n. With ReLU, the negative half becomes 0, so the variance is doubled to make up for it.

[7] Differentiating $h + F(h)$ with respect to h gives $1 + F'(h)$. Thanks to the "1+" part, the product does not approach 0 even if F' is small.

the values from drifting from layer to layer. It is like resetting the loudness of the voice to a standard level now and then along the way.

7 Countermeasures against overfitting

Neural networks have a very large number of parameters, so the **overfitting** we saw in the previous document (fitting the training data well but not predicting unseen data well) happens easily.
Think of a student who memorized the answers to past exam papers. They get full marks on the past papers, but cannot solve a new problem with slightly different numbers. A network with too many knobs easily memorizes the "answers" of the training data.

Typical countermeasures are as follows.

- **Regularization (weight decay)**: add $\lambda \times$ (the sum of the squares of the weights) to the loss to keep the weights from becoming too large. This is the same idea as the ridge regression of the previous document.
- **Early stopping**: during training, record the loss on validation data not used for training, and stop training when it starts to increase. If the scores on the past papers keep rising but the scores on mock exams (the validation data) start to fall, it is a sign that memorization has begun.
- **Dropout**: during training, a fixed proportion (for example, 50%) of the hidden units is randomly switched off each time. It is like a sports team in which a few players are randomly rested at each practice: the team learns to play without relying on a particular star, and everyone can play. It encourages learning robust representations that do not rely too much on particular units. All units are used at prediction time.
- **Data augmentation**: artificially increase the training data, for example by slightly rotating, shifting or flipping images. It is like practicing with variants of the same problem with the numbers changed.
- **More data**: in the end, this is the most reliable countermeasure.

8 Summary

1. Finding the gradient by numerical differentiation requires computation proportional to the number of parameters, which is impractical for large networks.
2. Backpropagation applies the chain rule in order from the output side and finds the gradients of all the parameters with about the same amount of computation as a forward pass.
3. The gradient of a weight is "the error δ that arrived from behind $\times$ the value that came into that weight," and the error δ is sent from the output side to the input side as "the error of the layer behind $\times$ weight $\times$ derivative of the activation function."
4. The loss of a neural network is not convex, and depending on the initial values, training can stall. Making the network larger makes this problem less likely.
5. In actual training, stochastic gradient descent with mini-batches, avoiding vanishing gradients with ReLU and careful initial values, and suppressing overfitting with regularization and the like are combined.

Coming next The networks so far have been "fully connected," with every component of the input connected to every unit. Part 3 deals with **convolutional neural networks**, which are suited to data with spatial structure, such as images.

Appendix A Backpropagation written with matrices and vectors

This appendix uses partial derivatives and matrices (first-year university linear algebra).

Notation Vectors are written in bold lowercase ($\mathbf{x}$, $\mathbf{h}$) and matrices in uppercase (W); vectors are column vectors. The superscript T is the **transpose** (swapping rows and columns), not a power. $\mathbf{a} \odot \mathbf{b}$ is the **componentwise product** (Hadamard product): $(a_1, a_2)^\mathsf{T} \odot (b_1, b_2)^\mathsf{T} = (a_1 b_1,\ a_2 b_2)^\mathsf{T}$. $\dfrac{\partial L}{\partial W}$ is the matrix of the partial derivatives of L with respect to each entry of W, arranged in the same shape as W. When a function f acts on a vector, it acts on each component separately.

A.1 A two-layer network

The network of Section 3.1 can be written

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \quad \mathbf{h} = \sigma\big(\mathbf{z}^{(1)}\big), \quad z^{(2)} = W^{(2)}\mathbf{h} + b^{(2)}, \quad \hat{y} = \sigma\big(z^{(2)}\big).$$

$W^{(1)}$ is a 2×2 matrix and $W^{(2)}$ is a 1×2 matrix. The results of Section 3.2 can be summarized as

$$\delta^{(2)} = \hat{y} - y, \qquad \frac{\partial L}{\partial W^{(2)}} = \delta^{(2)}\,\mathbf{h}^\mathsf{T}, \qquad \frac{\partial L}{\partial b^{(2)}} = \delta^{(2)},$$

$$\boldsymbol{\delta}^{(1)} = \frac{\partial L}{\partial \mathbf{z}^{(1)}} = \big(W^{(2)\mathsf{T}}\delta^{(2)}\big) \odot \sigma'\big(\mathbf{z}^{(1)}\big), \qquad \frac{\partial L}{\partial W^{(1)}} = \boldsymbol{\delta}^{(1)}\,\mathbf{x}^\mathsf{T}, \qquad \frac{\partial L}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}^{(1)}.$$

A.2 Derivation of the formulas

Error of the output layer With $L = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$ and $\hat{y} = \sigma(z^{(2)})$, using $\sigma'(z) = \sigma(z)(1-\sigma(z))$,

$$\delta^{(2)} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dz^{(2)}} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\hat{y}(1-\hat{y}) = -y(1-\hat{y}) + (1-y)\hat{y} = \hat{y} - y.$$

Gradients of the weights and the error of the hidden layer Since $z^{(2)} = \sum_k W_k^{(2)} h_k + b^{(2)}$, $\partial z^{(2)}/\partial W_k^{(2)} = h_k$. By the chain rule, $\partial L/\partial W_k^{(2)} = \delta^{(2)} h_k$. Arranged as a row vector, this is $\delta^{(2)}\mathbf{h}^\mathsf{T}$.

Next, $z^{(2)}$ depends on h_k with weight $W_k^{(2)}$, so $\partial L/\partial h_k = \delta^{(2)} W_k^{(2)}$. Furthermore, $h_k = \sigma(z_k^{(1)})$, so

$$\delta_k^{(1)} = \frac{\partial L}{\partial z_k^{(1)}} = \frac{\partial L}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k^{(1)}} = W_k^{(2)}\,\delta^{(2)}\,\sigma'\big(z_k^{(1)}\big).$$

Arranged over k, this is $\big(W^{(2)\mathsf{T}}\delta^{(2)}\big) \odot \sigma'(\mathbf{z}^{(1)})$. Finally, since $z_k^{(1)} = \sum_j W_{kj}^{(1)} x_j + b_k^{(1)}$, $\partial L/\partial W_{kj}^{(1)} = \delta_k^{(1)} x_j$. The matrix with these in row k, column j is $\boldsymbol{\delta}^{(1)}\mathbf{x}^\mathsf{T}$ (column vector $\times$ row vector).

A.3 General multilayer networks

For a network with L layers, $\mathbf{z}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$, $\mathbf{h}^{(l)} = f\big(\mathbf{z}^{(l)}\big)$ ($\mathbf{h}^{(0)} = \mathbf{x}$), the form is exactly the same:

$$\boldsymbol{\delta}^{(l)} = \left(W^{(l+1)\mathsf{T}}\boldsymbol{\delta}^{(l+1)}\right) \odot f'\left(\mathbf{z}^{(l)}\right), \qquad \frac{\partial L}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)}\,\mathbf{h}^{(l-1)\mathsf{T}}, \qquad \frac{\partial L}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}.$$

In the forward pass, multiplying by $W^{(l)}$ moves from input to output; in the backward pass, the error is sent from output to input by **multiplying by the transpose** $W^{(l)\mathsf{T}}$. The k-th component of $W^{(l+1)\mathsf{T}}\boldsymbol{\delta}^{(l+1)}$ is $\sum_m W^{(l+1)}_{mk}\delta^{(l+1)}_m$, which corresponds to "adding up the error $\times$ weight arriving from each destination" described in Section 3.3.

A.4 Vanishing gradients written with matrices

Applying the general formula repeatedly, the error in the layer nearest the input takes the form

$$\boldsymbol{\delta}^{(1)} \sim W^{(2)\mathsf{T}} f'(\cdot)\, W^{(3)\mathsf{T}} f'(\cdot) \cdots W^{(L)\mathsf{T}} f'(\cdot)\, \boldsymbol{\delta}^{(L)},$$

alternately multiplying transposed weight matrices and derivatives of the activation function ($\sim$ means "roughly of this form," and $f'(\cdot)$ is applied as a componentwise product). The long, thin network of Section 6.1 is the case in which all the matrices are 1×1.

Appendix B Python code (standard library only)

These programs compute the examples in the main text. They use no external libraries, only the standard features of Python 3.

B.1 Learning XOR by backpropagation (Section 4)

```
# Learn XOR with a 2-2-1 network (backpropagation)
import math, random

X = [(0, 0), (0, 1), (1, 0), (1, 1)]
Y = [0, 1, 1, 0]

def sigmoid(z):
    return 1.0 / (1.0 + math.exp(-z))

def train(seed, eta=1.0, epochs=5000, log=()):
    random.seed(seed)
    H = 2                                          # number of hidden units
    W1 = [[random.uniform(-1, 1) for _ in range(2)] for _ in range(H)]
    b1 = [0.0] * H
    W2 = [random.uniform(-1, 1) for _ in range(H)]
    b2 = 0.0
    history = []
    for ep in range(epochs + 1):
        # reset the gradients to 0
        gW1 = [[0.0, 0.0] for _ in range(H)]; gb1 = [0.0] * H
        gW2 = [0.0] * H; gb2 = 0.0; L = 0.0
        for x, y in zip(X, Y):
            # --- forward propagation ---
            z1 = [W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k] for k in range(H)]
            h = [sigmoid(z) for z in z1]
            z2 = sum(W2[k]*h[k] for k in range(H)) + b2
            p = sigmoid(z2)
            L += -(y*math.log(p) + (1-y)*math.log(1-p))
            # --- backpropagation ---
            d2 = p - y                             # delta(2) = dL/dz2
            for k in range(H):
```

```
                gW2[k] += d2 * h[k]                 # dL/dW2_k = delta(2) h_k
                d1 = W2[k] * d2 * h[k] * (1 - h[k]) # delta(1)_k = W2_k delta(2) sigma'(z1_k)
                gW1[k][0] += d1 * x[0]              # dL/dW1_kj = delta(1)_k x_j
                gW1[k][1] += d1 * x[1]
                gb1[k] += d1
            gb2 += d2
        history.append(L)
        if ep in log:
            print(f"  epoch {ep:5d}: Loss = {L:.4f}")
        # --- gradient descent ---
        for k in range(H):
            W2[k] -= eta * gW2[k]
            b1[k] -= eta * gb1[k]
            for j in range(2):
                W1[k][j] -= eta * gW1[k][j]
        b2 -= eta * gb2
    return (W1, b1, W2, b2), history

def predict(params, x):
    W1, b1, W2, b2 = params
    h = [sigmoid(W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k]) for k in range(len(b1))]
    return sigmoid(sum(W2[k]*h[k] for k in range(len(b1))) + b2), h

if __name__ == "__main__":
    for seed in (1, 9):
        print(f"seed = {seed}")
        params, hist = train(seed, log=(0, 100, 500, 1000, 2000, 5000))
        for x in X:
            p, h = predict(params, x)
            print(f"  input {x} -> output {p:.3f}   hidden layer ({h[0]:.2f}, {h[1]:.2f})")
```

Output (seed 1 is initial values A in the main text, seed 9 is initial values B):

```
seed = 1
  epoch     0: Loss = 2.7741
  epoch   100: Loss = 2.7721
  epoch   500: Loss = 0.0746
  epoch  1000: Loss = 0.0239
  epoch  2000: Loss = 0.0100
  epoch  5000: Loss = 0.0036
  input (0, 0) -> output 0.001   hidden layer (0.99, 0.99)
  input (0, 1) -> output 0.999   hidden layer (1.00, 0.01)
  input (1, 0) -> output 0.999   hidden layer (0.01, 1.00)
  input (1, 1) -> output 0.001   hidden layer (0.99, 0.99)
seed = 9
  epoch     0: Loss = 2.8736
  epoch   100: Loss = 2.1686
  epoch   500: Loss = 1.4020
  epoch  1000: Loss = 1.3928
  epoch  2000: Loss = 1.3892
  epoch  5000: Loss = 1.3874
  input (0, 0) -> output 0.000   hidden layer (0.86, 0.05)
  input (0, 1) -> output 1.000   hidden layer (0.02, 0.86)
  input (1, 0) -> output 0.500   hidden layer (0.00, 0.00)
  input (1, 1) -> output 0.500   hidden layer (0.00, 0.00)
```

B.2 Gradient check (Section 3.4)

```
# Check the gradients found by backpropagation against numerical differentiation (gradient check)
import math, random
```

```
X = [(0, 0), (0, 1), (1, 0), (1, 1)]
Y = [0, 1, 1, 0]
sig = lambda z: 1.0 / (1.0 + math.exp(-z))

# Put all the parameters in one list: W1 (4), b1 (2), W2 (2), b2 (1)
random.seed(1)
theta = [random.uniform(-1, 1) for _ in range(9)]

def unpack(t):
    return [[t[0], t[1]], [t[2], t[3]]], t[4:6], t[6:8], t[8]

def loss(t):
    W1, b1, W2, b2 = unpack(t)
    L = 0.0
    for x, y in zip(X, Y):
        h = [sig(W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k]) for k in range(2)]
        p = sig(W2[0]*h[0] + W2[1]*h[1] + b2)
        L += -(y*math.log(p) + (1-y)*math.log(1-p))
    return L

def backprop(t):
    W1, b1, W2, b2 = unpack(t)
    g = [0.0] * 9
    for x, y in zip(X, Y):
        h = [sig(W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k]) for k in range(2)]
        p = sig(W2[0]*h[0] + W2[1]*h[1] + b2)
        d2 = p - y
        for k in range(2):
            d1 = W2[k] * d2 * h[k] * (1 - h[k])
            g[2*k] += d1 * x[0]; g[2*k+1] += d1 * x[1]; g[4+k] += d1
            g[6+k] += d2 * h[k]
        g[8] += d2
    return g

def numerical(t, eps=1e-5):
    g = []
    for i in range(len(t)):
        tp = t[:]; tp[i] += eps
        tm = t[:]; tm[i] -= eps
        g.append((loss(tp) - loss(tm)) / (2 * eps))   # central difference
    return g

names = ["W1_11", "W1_12", "W1_21", "W1_22", "b1_1", "b1_2", "W2_1", "W2_2", "b2"]
for n, a, b in zip(names, backprop(theta), numerical(theta)):
    print(f"{n:6s}  backprop {a:+.8f}   numerical {b:+.8f}   difference {abs(a-b):.1e}")
```

Output:

```
W1_11   backprop -0.00844318   numerical -0.00844318   difference 1.4e-12
W1_12   backprop -0.00983020   numerical -0.00983020   difference 2.1e-11
W1_21   backprop -0.02303636   numerical -0.02303636   difference 1.6e-11
W1_22   backprop -0.02104372   numerical -0.02104372   difference 9.9e-12
b1_1    backprop -0.01830691   numerical -0.01830691   difference 6.8e-12
b1_2    backprop -0.04426312   numerical -0.04426312   difference 1.9e-11
W2_1    backprop -0.19033744   numerical -0.19033744   difference 9.2e-12
W2_2    backprop -0.18284084   numerical -0.18284084   difference 1.4e-12
b2      backprop -0.38068435   numerical -0.38068435   difference 1.1e-11
```