Deep Learning Course

Lesson 3 -Backpropagation, and Gradient Descent

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

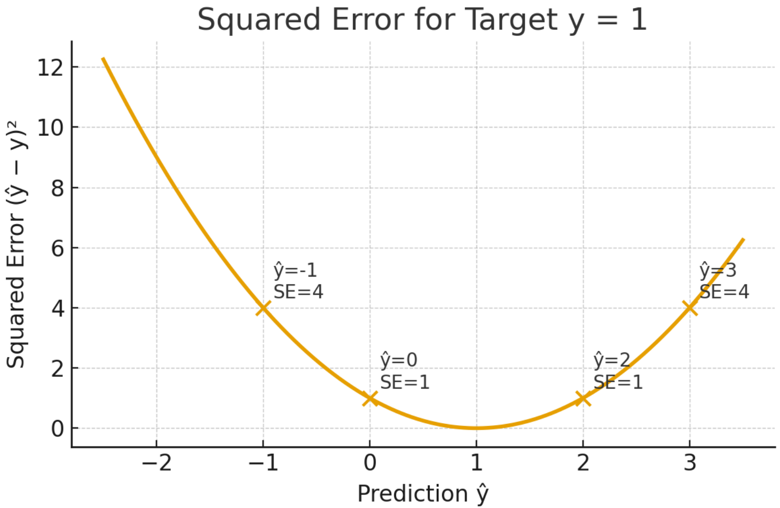
October 29, 2025

What does it mean to "improve" a model?

- In regression, we predict one or more numbers. We "improve" the model when predictions get closer to the true targets.
- The discrepancy is called the **loss**. A common choice is the **squared error**:

$$\mathrm{SE}(\hat{y}, y) = (\hat{y} - y)^2.$$

- Key properties:
  - $\mathrm{SE} \geq 0$: the error is never negative.
  - Larger deviations are penalized more strongly (quadratic growth).
- Example: if the target is $y = 1$
  - Predicting $\hat{y} = 0$ or 2 gives error 1
  - Predicting $\hat{y} = -1$ or 3 gives error 4



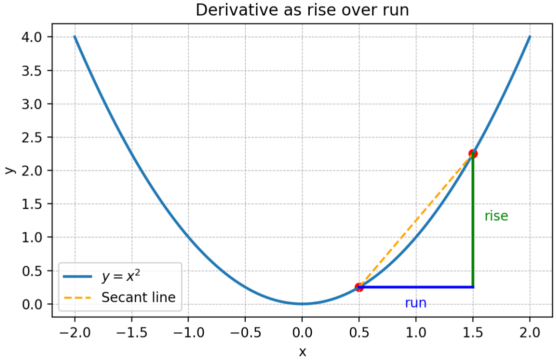
Example with target $y = 1$: predictions $\hat{y} = 0$ or 2 give error 1; predictions $\hat{y} = -1$ or 3 give error 4.

Our goal: reducing the error

- Once we can measure the error (loss), the next step is to **minimize** it.
- Intuitively, we want to **go downhill** in the loss landscape - moving toward smaller and smaller values.
- The parameters of the network (weights and biases) determine where we are on this landscape.
- The question becomes: *In which direction should we move the parameters to make the loss smaller?*

How do we go downhill? Thanks to derivatives!

- The **derivative** tells us how a function changes when its input changes.
- Intuitively, it measures the **slope** or **tilt** of the curve at a given point.
- A positive derivative means the function is increasing; a negative derivative means it is decreasing.
- When we want to **minimize** a function, we move in the direction where the derivative is **negative** - that is, downhill.



Formal definition of the derivative

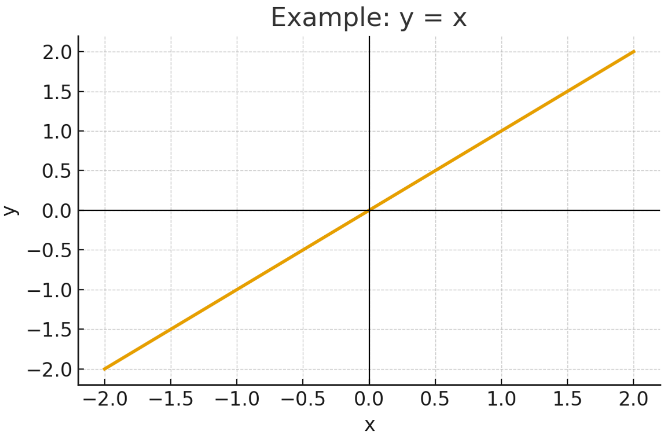
- The derivative of a function $f(x)$ at a point x_0 measures the instantaneous rate of change of f at that point.
- Formally, it is defined as the limit of the average rate of change:

$$f'(x_0) = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

- The smaller the Δx, the closer the secant line gets to the tangent line.
- Geometrically, the derivative corresponds to the slope of the tangent line to the curve $y = f(x)$ at x_0.
- If this limit exists, the function is said to be **differentiable** at x_0.

Example: $y = x$

- Consider the simple function $y = x$.
- It is a straight line passing through the origin.
- Question: **What is the value of the derivative?**



Derivative of $y = x$

- Let $f(x) = x$.
- By definition, the derivative is

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

- Substituting $f(x) = x$:

$$f'(x) = \lim_{\Delta x \to 0} \frac{(x + \Delta x) - x}{\Delta x}.$$

- Simplifying the numerator:

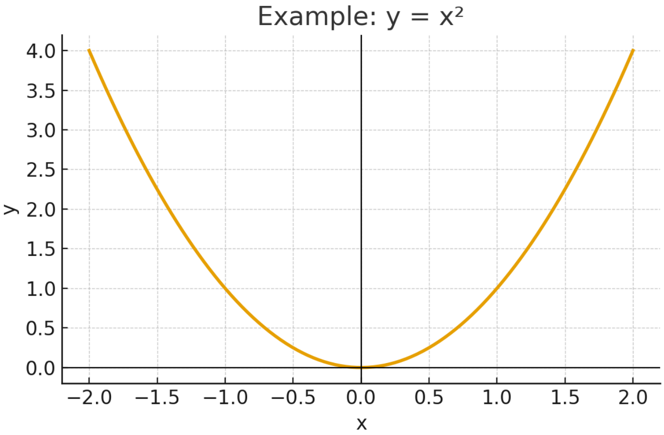
$$f'(x) = \lim_{\Delta x \to 0} \frac{\Delta x}{\Delta x} = \lim_{\Delta x \to 0} 1 = 1.$$

Result

For all x, the derivative of $y = x$ is $f'(x) = 1$.

Example: $y = x^2$

- Consider the quadratic function $y = x^2$.
- Notice that it is a curved shape opening upward.
- Question: **What is the value of the derivative of $y = x^2$?**



Derivative of $y = x^2$

- Let $f(x) = x^2$.
- By definition of the derivative:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

- Substitute $f(x) = x^2$:

$$f'(x) = \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h}.$$

- Expand the square:

$$f'(x) = \lim_{h \to 0} \frac{x^2 + 2xh + h^2 - x^2}{h}.$$

- Simplify the numerator:

$$f'(x) = \lim_{h \to 0} \frac{2xh + h^2}{h} = \lim_{h \to 0} (2x + h).$$

- Taking the limit:

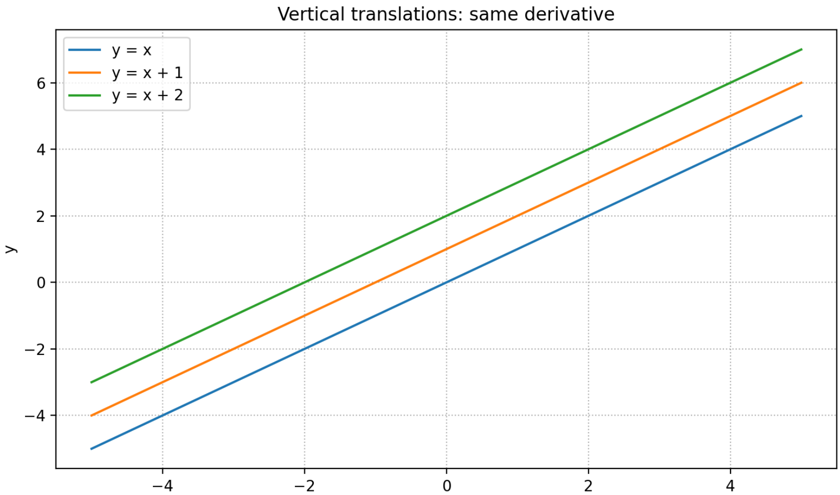
$$f'(x) = 2x.$$

Result

For all x, the derivative of $y = x^2$ is $f'(x) = 2x$.

Vertical translations do not change the derivative

- Functions $y = x$, $y = x + 1$, $y = x + 2$ share the same slope 1.
- Adding a constant shifts the graph but does *not* change the derivative.



Introduction to Partial Derivatives

- When a function depends on more than one variable, such as $f(x, y)$, we can study how the function changes with respect to each variable separately.
- The **partial derivative** measures how the function changes when we vary one variable while keeping the others constant.
- In practice, when taking the derivative with respect to one variable, the other variables are treated as if they were **constants**.
- Formally:
$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h, y) - f(x, y)}{h}, \quad \frac{\partial f}{\partial y} = \lim_{h \to 0} \frac{f(x, y+h) - f(x, y)}{h}.$$
- Example: if $f(x, y) = x^2 + 3y$,
$$\frac{\partial f}{\partial x} = 2x, \quad \frac{\partial f}{\partial y} = 3.$$
- Partial derivatives are fundamental in multivariable calculus and in machine learning, where loss functions often depend on many parameters.

What happens when x is a vector?

- When x is a vector $x = [x_1, x_2, \dots, x_n]$, the function $f(x)$ depends on multiple variables at once.
- Taking the derivative of f with respect to x means computing the **partial derivative with respect to each component**.
- The result is a **vector** called the **gradient**:

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}.$$

- Each component of the gradient tells us how much $f(x)$ changes when we vary one coordinate, keeping the others constant.
- The gradient points in the direction of the **steepest increase** of the function.

When both input and output are vectors

- Sometimes a function takes a vector as input and returns a vector as output:

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

- Example:

$$f(x) = \begin{bmatrix} x_1 + 2x_2 \\ 3x_1 - x_2 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

- In this case, we cannot describe the derivative with a single vector.
- Instead, we use a **matrix of partial derivatives**, called the **Jacobian matrix**:

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}.$$

- Each row corresponds to the gradient of one output component with respect to all input components.
- The Jacobian generalizes the concept of the derivative to vector-valued functions.

Computing the Jacobian from the previous example

- Recall the function:

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} x_1 + 2x_2 \\ 3x_1 - x_2 \end{bmatrix}.$$

- Compute the partial derivatives of each component:

$$\frac{\partial f_1}{\partial x_1} = 1, \quad \frac{\partial f_1}{\partial x_2} = 2,$$

$$\frac{\partial f_2}{\partial x_1} = 3, \quad \frac{\partial f_2}{\partial x_2} = -1.$$

- Therefore, the **Jacobian matrix** is:

$$J_f(x) = \begin{bmatrix} 1 & 2 \\ 3 & -1 \end{bmatrix}.$$

- Each row represents the gradient of one output component with respect to the input vector x.

When x is itself a function: the Chain Rule

- Sometimes the variable x is not a simple number or vector, but rather another function.
- Example: if $y = f(u)$ and $u = g(x)$, then y ultimately depends on x through g.
- To find how y changes with respect to x, we use the **Chain Rule**.
- The Chain Rule connects the rate of change of f with respect to u and the rate of change of u with respect to x:

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

- Intuitively, we are "chaining" together the effects of each transformation.
- This concept is fundamental in backpropagation, where each layer of a neural network applies the Chain Rule to propagate gradients backward.

Returning to our main goal: reducing the error

- We now have all the tools we need to return to our original problem: **reducing the loss**.
- We know how to compute the error between the target value and the predicted value.
- However, what we truly want to know is: **How should we change the weights so that the error decreases?**
- The loss depends on the weights indirectly:

$$\text{Loss} = L(\hat{y}, y) = L(f(x; W), y).$$

- Thanks to derivatives and the Chain Rule, we can measure how small changes in the weights W affect the loss:

$$\frac{\partial L}{\partial W}.$$

- This derivative tells us the direction in which to adjust the weights to reduce the error.
- The process of computing these derivatives layer by layer is called **backpropagation**.

Using the Chain Rule to express $\frac{dL}{dW}$

- In a neural network, the loss L depends on the weights W through several intermediate variables:

$$x \;\rightarrow\; z = xW + b \;\rightarrow\; \hat{y} = f(z) \;\rightarrow\; L(\hat{y}, y)$$

- To compute how the loss changes with respect to the weights, we apply the **Chain Rule**:

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dz} \cdot \frac{dz}{dW}.$$

- Each term in this product captures a different part of the dependency:
  - $\frac{dL}{d\hat{y}}$: how the loss changes with the predicted output,
  - $\frac{d\hat{y}}{dz}$: how the activation output changes with the pre-activation,
  - $\frac{dz}{dW}$: how the pre-activation changes with the weights.
- This decomposition allows us to compute gradients efficiently, step by step, moving backward through the network.

Forward pass with $\hat{y} = f(xW^T)$ - part 1

- Let the input vector be:

$$x = \begin{bmatrix} x_1 & x_2 \end{bmatrix}, \quad W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}.$$

- Since W has shape 3×2, we compute:

$$z = xW^T.$$

- Expanding element by element:

$$\begin{aligned} z_1 &= x_1 w_{11} + x_2 w_{12}, \\ z_2 &= x_1 w_{21} + x_2 w_{22}, \\ z_3 &= x_1 w_{31} + x_2 w_{32}. \end{aligned}$$

- The activation function f is applied elementwise:

$$\hat{y} = f(z) = \begin{bmatrix} f(z_1), & f(z_2), & f(z_3) \end{bmatrix}.$$

- At this stage, we have completed the **forward pass**, producing the predicted output $\hat{y}$.

Forward pass with $\hat{y} = f(xW^T)$ - part 2

- Suppose the true target is:

$$y = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix}.$$

- We define the loss using the **squared error**:

$$L = \frac{1}{2}\sum_{i=1}^{3}(\hat{y}_i - y_i)^2.$$

- The division by 2 is a mathematical convenience: when we later take the derivative, the exponent 2 and the factor $\frac{1}{2}$ cancel out neatly:

$$\frac{d}{d\hat{y}_i}\left(\frac{1}{2}(\hat{y}_i - y_i)^2\right) = (\hat{y}_i - y_i).$$

- This simplification makes the gradient expressions cleaner and easier to interpret.
- Therefore, the forward computation gives:

$$x \xrightarrow{W^T} z \xrightarrow{f} \hat{y} \xrightarrow{L} \text{Loss}.$$

Computing $\frac{dL}{d\hat{y}}$

- Recall the loss function for our example:

$$L = \frac{1}{2}\sum_{i=1}^{3}(\hat{y}_i - y_i)^2.$$

- To find how the loss changes with respect to the predicted outputs $\hat{y}$, we take the derivative with respect to each component $\hat{y}_i$:

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{\partial}{\partial \hat{y}_i}\left(\frac{1}{2}(\hat{y}_i - y_i)^2\right).$$

- Applying the derivative:

$$\frac{\partial L}{\partial \hat{y}_i} = (\hat{y}_i - y_i).$$

- In vector form:

$$\frac{dL}{d\hat{y}} = \left[\hat{y}_1 - y_1, \quad \hat{y}_2 - y_2, \quad \hat{y}_3 - y_3\right].$$

Computing $\frac{d\hat{y}}{dz}$: general case

- Recall that:

$$z_1 = x_1 w_{11} + x_2 w_{12},$$
$$z_2 = x_1 w_{21} + x_2 w_{22},$$
$$z_3 = x_1 w_{31} + x_2 w_{32}.$$

- The activation function f transforms z into:

$$\hat{y} = f(z)$$

- Since both $\hat{y}$ and z are vectors, the derivative of one with respect to the other is a **Jacobian matrix**:

$$\frac{d\hat{y}}{dz} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial z_1} & \frac{\partial \hat{y}_1}{\partial z_2} & \frac{\partial \hat{y}_1}{\partial z_3} \\ \frac{\partial \hat{y}_2}{\partial z_1} & \frac{\partial \hat{y}_2}{\partial z_2} & \frac{\partial \hat{y}_2}{\partial z_3} \\ \frac{\partial \hat{y}_3}{\partial z_1} & \frac{\partial \hat{y}_3}{\partial z_2} & \frac{\partial \hat{y}_3}{\partial z_3} \end{bmatrix}.$$

- Each element represents how a small change in one component of z affects one component of $\hat{y}$.
- In general, this matrix may contain nonzero off-diagonal terms if the activation function couples multiple outputs.

Simplification for pointwise activation functions (e.g. ReLU)

- For most neural network activations, such as ReLU, sigmoid, or tanh, the function acts **independently** on each component of z:

$$\hat{y}_i = f(z_i).$$

- This means that changing z_j affects only $\hat{y}_j$, and not any other component.
- As a result, the Jacobian matrix becomes **diagonal**:

$$\frac{d\hat{y}}{dz} = \begin{bmatrix} f'(z_1) & 0 & 0 \\ 0 & f'(z_2) & 0 \\ 0 & 0 & f'(z_3) \end{bmatrix}.$$

- For the ReLU activation:

$$f(z_i) = \max(0, z_i) \quad \Rightarrow \quad f'(z_i) = \begin{cases} 1 & \text{if } z_i > 0, \\ 0 & \text{if } z_i \leq 0. \end{cases}$$

- This diagonal structure is what makes backpropagation efficient, since each neuron's derivative depends only on its own activation.

Understanding $\frac{dz}{dW}$: step by step

- Recall that:

$$z_1 = x_1 w_{11} + x_2 w_{12}, \quad z_2 = x_1 w_{21} + x_2 w_{22}.$$

- If we take the derivative of z_1 with respect to the weights in the first row $W_1 = [w_{11}, w_{12}]$:

$$\frac{\partial z_1}{\partial W_1} = \begin{bmatrix} \frac{\partial z_1}{\partial w_{11}} & \frac{\partial z_1}{\partial w_{12}} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} = x.$$

- However, if we take the derivative of z_1 with respect to the second row of weights $W_2 = [w_{21}, w_{22}]$:

$$\frac{\partial z_1}{\partial W_2} = \begin{bmatrix} \frac{\partial z_1}{\partial w_{21}} & \frac{\partial z_1}{\partial w_{22}} \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}.$$

- This shows that z_1 depends only on the first row of W, while z_2 depends only on the second row.
- Each neuron is connected only to its own set of weights.

Building the full Jacobian $\frac{dz}{dW}$

- Extending the previous reasoning to all components z_1, z_2, z_3:

$$z_i = x_1 w_{i1} + x_2 w_{i2}, \quad i = 1, 2, 3.$$

- Each z_i depends only on the weights in its own row $W_i = [w_{i1}, w_{i2}]$:

$$\frac{\partial z_i}{\partial W_j} = \begin{cases} x & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

- Therefore, the full Jacobian is a block matrix:

$$\frac{dz}{dW} = \begin{bmatrix} x & 0 & 0 \\ 0 & x & 0 \\ 0 & 0 & x \end{bmatrix},$$

where each block $x = [x_1, x_2]$ represents how one neuron's output changes with its own weights.

Recap: structure of the full derivative

- We expressed the gradient of the loss with respect to the weights using the Chain Rule:

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dz} \cdot \frac{dz}{dW}.$$

- Each component describes a different relationship in the network:
  - $\frac{dL}{d\hat{y}} = \hat{y} - y$ measures how the loss changes with the predicted output.
  - $\frac{d\hat{y}}{dz} = \mathrm{diag}(f'(z_1), f'(z_2), f'(z_3))$ describes how the activation function transforms the signal.
- These two parts together form the gradient that flows backward through the output layer.

Recap: how z depends on the weights

- The third term, $\dfrac{dz}{dW}$, shows how each pre-activation depends on its own weights:

$$z_i = x_1 w_{i1} + x_2 w_{i2} \quad \Rightarrow \quad \frac{\partial z_i}{\partial W_i} = x.$$

- Cross-dependencies are zero:

$$\frac{\partial z_i}{\partial W_j} = \begin{cases} x & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

- Putting all terms together, the gradient for each neuron becomes:

$$\frac{dL}{dW_i} = (\hat{y}_i - y_i)\, f'(z_i)\, x.$$

- Each neuron's update depends on:
  1. The prediction error $(\hat{y}_i - y_i)$,
  2. The local slope $f'(z_i)$,
  3. The input vector x.

Introducing the concept of δ

- To simplify notation, we group the terms that measure the local error for each neuron:

$$\delta_i = (\hat{y}_i - y_i)\, f'(z_i).$$

- The vector form is:

$$\boldsymbol{\delta} = \begin{bmatrix}\delta_1 & \delta_2 & \delta_3\end{bmatrix} = (\hat{y} - y) \odot f'(z),$$

  where $\odot$ denotes elementwise multiplication.
- Intuitively:
  - $(\hat{y} - y)$ tells us the prediction error,
  - $f'(z)$ scales that error according to the local slope of the activation.
- $\boldsymbol{\delta}$ represents the **error signal** that will be propagated backward through the network.

Simplified gradient using δ

- Using the definition of δ_i, the gradient with respect to the weights becomes:

$$\frac{dL}{dW_i} = \delta_i x.$$

- In matrix form, stacking all neurons together:

$$\frac{dL}{dW} = \boldsymbol{\delta}^T x.$$

- This compact expression shows that:
  - the gradient is the **outer product** between the input x and the error signal $\boldsymbol{\delta}$,
  - each neuron's weight update is proportional to the input values that produced the error.
- This formulation is the foundation of **backpropagation**.

From Jacobians to outer product: make the multiplication explicit

We start from the chain rule:

$$\frac{dL}{dW} = \underbrace{\frac{dL}{d\hat{y}}}_{1\times 3} \underbrace{\frac{d\hat{y}}{dz}}_{3\times 3} \underbrace{\frac{dz}{dW}}_{\text{block matrix}} .$$

For a pointwise activation, $\frac{d\hat{y}}{dz} = \mathrm{diag}(f'(z_1), f'(z_2), f'(z_3))$. Define $\delta := \frac{dL}{dz} = (\hat{y} - y) \odot f'(z)$ so that $\frac{dL}{d\hat{y}} \frac{d\hat{y}}{dz} = \delta$ (shape 1×3).
The last factor has block-diagonal structure because each z_i depends only on W_i:

$$\frac{dz}{dW} = \begin{bmatrix} x & 0 & 0 \\ 0 & x & 0 \\ 0 & 0 & x \end{bmatrix}, \quad \text{each block } x = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \text{ has shape } 1 \times 2.$$

Now multiply the vector $\delta = \begin{bmatrix} \delta_1 & \delta_2 & \delta_3 \end{bmatrix}$ by this block matrix:

$$\delta \cdot \frac{dz}{dW} = \left[\, \delta_1 x \;\middle|\; \delta_2 x \;\middle|\; \delta_3 x \,\right],$$

a 1×6 row obtained by concatenating the three 1×2 blocks.

Where the outer product comes from

The row $\left[\, \delta_1 x \mid \delta_2 x \mid \delta_3 x \,\right]$ groups the contributions per neuron. If we reshape these two-wide blocks into rows, we obtain a 3×2 matrix:

$$\frac{dL}{dW} \equiv \begin{bmatrix} \delta_1 x \\ \delta_2 x \\ \delta_3 x \end{bmatrix} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 \end{bmatrix} = \delta^T x.$$

Elementwise this means

$$\frac{\partial L}{\partial w_{ik}} = \delta_i \, x_k, \quad i \in \{1,2,3\},\ k \in \{1,2\}.$$

Why it collapses to an outer product:

1. $\frac{d\hat{y}}{dz}$ is diagonal for pointwise activations, so cross terms vanish when forming δ.
2. Each z_i depends only on the weights in row W_i, making $\frac{dz}{dW}$ block-diagonal with identical x blocks.

These two structural diagonals turn the general Jacobian product into the outer product $\delta^T x$.

What about the b?

- Until now we ignored the bias term b, but it is present in the full expression:

$$z = xW^T + b.$$

- The bias affects each neuron additively, so the derivative of z_i with respect to b_i is simply:

$$\frac{\partial z_i}{\partial b_i} = 1.$$

- Therefore, using the Chain Rule:

$$\frac{dL}{db_i} = \frac{dL}{dz_i} \cdot \frac{dz_i}{db_i} = \delta_i.$$

- In vector form:

$$\frac{dL}{db} = \boldsymbol{\delta}.$$

- Each bias term has the same gradient as its corresponding neuron's δ_i.

Embedding the bias inside the input vector

- To simplify notation, we can include the bias in the weight matrix by augmenting the input:

$$\tilde{x} = \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix}, \quad \tilde{W} = \begin{bmatrix} w_{11} & w_{12} & b_1 \\ w_{21} & w_{22} & b_2 \\ w_{31} & w_{32} & b_3 \end{bmatrix}.$$

- The forward pass becomes:

$$z = \tilde{x}\,\tilde{W}^T.$$

- The gradient keeps the same form:

$$\frac{dL}{d\tilde{W}} = \boldsymbol{\delta}^T\tilde{x},$$

  and the last column of $\frac{dL}{d\tilde{W}}$ corresponds exactly to $\frac{dL}{db} = \boldsymbol{\delta}$.
- This trick allows us to treat the bias as just another weight connected to a constant input 1.

From backpropagation to gradient descent

- After computing all gradients through backpropagation, we know how the loss changes with respect to each parameter:

$$\frac{dL}{dW}, \quad \frac{dL}{db}.$$

- The next step is to actually **update the parameters** in order to reduce the loss.
- The idea of **gradient descent** is simple:

$$\text{new parameter} = \text{old parameter} - \eta \cdot \text{gradient},$$

  where η is the **learning rate**.
- Applied to our case:

$$W \leftarrow W - \eta \frac{dL}{dW}, \quad b \leftarrow b - \eta \frac{dL}{db}.$$

- The learning rate controls the size of each update:
  - too large $\rightarrow$ overshoot the minimum,
  - too small $\rightarrow$ very slow convergence.
- This iterative process gradually moves the network's parameters downhill in the loss landscape.

Lab time (1/3) - Forward pass exercise

- Let's practice the full forward computation with a simple example.
- Given:

$$x = [1, 2], \quad W = \begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 5 & 2 \end{bmatrix}, \quad y = [1, 4, 4].$$

- Definitions:

$$z = xW^T, \quad \hat{y} = \mathrm{ReLU}(z), \quad L = \frac{1}{2}(\hat{y} - y)^2.$$

- **Tasks:**
  1. Compute the vector z.
  2. Apply the ReLU activation to get $\hat{y}$.
  3. Compute the loss L.
- Hint: Remember that ReLU is defined as $\mathrm{ReLU}(z_i) = \max(0, z_i)$.

Question: What are the values of z, $\hat{y}$, and L?

Lab time (1/3) - Solution

- Given:

$$x = [1, 2], \quad W = \begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 5 & 2 \end{bmatrix}, \quad y = [1, 4, 4].$$

- Compute $z = xW^T$:

$$\begin{aligned} z_1 &= 1 \cdot 3 + 2 \cdot 2 = 7, \\ z_2 &= 1 \cdot 4 + 2 \cdot 1 = 6, \\ z_3 &= 1 \cdot 5 + 2 \cdot 2 = 9. \end{aligned}$$

- Apply ReLU activation:

$$\hat{y} = \mathrm{ReLU}(z) = [7, 6, 9].$$

- Compute the loss:

$$L = \frac{1}{2} \sum_i (\hat{y}_i - y_i)^2 = \frac{1}{2} \left[(7-1)^2 + (6-4)^2 + (9-4)^2 \right] = \frac{1}{2}(36 + 4 + 25) = \frac{65}{2} = 32.5.$$

- **Result:**

$$z = [7, 6, 9], \quad \hat{y} = [7, 6, 9], \quad L = 32.5.$$

Lab time (2/3): compute the gradients (task)

Given the same setup:

$$x = [1, 2], \quad W = \begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 5 & 2 \end{bmatrix}, \quad y = [1, 4, 4], \quad z = xW^T, \quad \hat{y} = \mathrm{ReLU}(z), \quad L = \frac{1}{2}\sum_i (\hat{y}_i - y_i)^2.$$

Tasks

1. Compute $\frac{dL}{d\hat{y}}$.
2. Compute $\frac{d\hat{y}}{dz}$ for ReLU and then $\frac{dL}{dz}$.
3. Using $z_i = x_1 w_{i1} + x_2 w_{i2}$, compute $\frac{dz}{dW}$.
4. Combine the parts to obtain $\frac{dL}{dW}$.

Hints:

$$\mathrm{ReLU}(u) = \max(0, u), \quad \mathrm{ReLU}'(u) = \begin{cases} 1 & u > 0 \\ 0 & u \leq 0 \end{cases}$$

Lab time (2/3): compute the gradients (solution)

From the forward pass:

$$z = [7, 6, 9], \quad \hat{y} = [7, 6, 9].$$

1. $\dfrac{dL}{d\hat{y}} = \hat{y} - y = [6, 2, 5].$
2. For ReLU at these $z_i > 0$: $\dfrac{d\hat{y}}{dz} = \mathrm{diag}(1, 1, 1)$. Hence $\dfrac{dL}{dz} = \dfrac{dL}{d\hat{y}} \dfrac{d\hat{y}}{dz} = [6, 2, 5]$. Define $\delta := \dfrac{dL}{dz} = [6, 2, 5]$.
3. Since $z_i = x_1 w_{i1} + x_2 w_{i2}$, $\dfrac{\partial z_i}{\partial W_i} = x = [1, 2]$, $\dfrac{\partial z_i}{\partial W_j} = 0$ for $i \neq j$.
4. Row-wise gradient:

$$\frac{dL}{dW_i} = \delta_i \, x \quad \Rightarrow \quad \frac{dL}{dW} = \begin{bmatrix} 6 \\ 2 \\ 5 \end{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 12 \\ 2 & 4 \\ 5 & 10 \end{bmatrix}.$$

Lab time (3/3) - Applying gradient descent (question)

We now use the results from the previous exercise. Given:

$$x = [1, 2], \quad W = \begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 5 & 2 \end{bmatrix}, \quad y = [1, 4, 4],$$

and we already computed:

$$\frac{dL}{dW} = \begin{bmatrix} 6 & 12 \\ 2 & 4 \\ 5 & 10 \end{bmatrix}.$$

We will now apply **gradient descent** to update the weights:

$$W_{\text{new}} = W - \eta \frac{dL}{dW}.$$

Tasks:

1. Choose a learning rate $\eta = 0.1$.
2. Compute the updated weights W_{new}.
3. Interpret what happens to each row of W.

Question: What are the new values of W after one gradient descent step?

Lab time (3/3) - Applying gradient descent (solution)

Using $\eta = 0.1$:

$$W_{\text{new}} = W - \eta \frac{dL}{dW} = \begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 5 & 2 \end{bmatrix} - 0.1 \times \begin{bmatrix} 6 & 12 \\ 2 & 4 \\ 5 & 10 \end{bmatrix}.$$

Compute the update:

$$0.1 \times \begin{bmatrix} 6 & 12 \\ 2 & 4 \\ 5 & 10 \end{bmatrix} = \begin{bmatrix} 0.6 & 1.2 \\ 0.2 & 0.4 \\ 0.5 & 1.0 \end{bmatrix}.$$

Subtract from W:

$$W_{\text{new}} = \begin{bmatrix} 3 - 0.6 & 2 - 1.2 \\ 4 - 0.2 & 1 - 0.4 \\ 5 - 0.5 & 2 - 1.0 \end{bmatrix} = \begin{bmatrix} 2.4 & 0.8 \\ 3.8 & 0.6 \\ 4.5 & 1.0 \end{bmatrix}.$$

Result:

$$W_{\text{new}} = \begin{bmatrix} 2.4 & 0.8 \\ 3.8 & 0.6 \\ 4.5 & 1.0 \end{bmatrix}.$$

Each weight has moved slightly **downhill** in the direction opposite to its gradient.

New loss after weight update

After applying the gradient descent step, the new weights are:

$$W_{\text{new}} = \begin{bmatrix} 2.4 & 0.8 \\ 3.8 & 0.6 \\ 4.5 & 1.0 \end{bmatrix}.$$

Now recompute the forward pass:

$$z_{\text{new}} = x W_{\text{new}}^T, \quad \hat{y}_{\text{new}} = \text{ReLU}(z_{\text{new}}), \quad L_{\text{new}} = \frac{1}{2} \sum_i (\hat{y}_{\text{new},i} - y_i)^2.$$

Step 1: Compute z_{new}

$$z_{\text{new}} = [1, 2] \begin{bmatrix} 2.4 & 3.8 & 4.5 \\ 0.8 & 0.6 & 1.0 \end{bmatrix} = [4.0, 5.0, 6.5].$$

Step 2: Apply ReLU:

$$\hat{y}_{\text{new}} = [4.0, 5.0, 6.5].$$

New loss after weight update

Step 3: Compute new loss:

$$L_{\text{new}} = \frac{1}{2}[(4-1)^2 + (5-4)^2 + (6.5-4)^2] = \frac{1}{2}(9+1+6.25) = 8.125.$$

Result: The new loss is

$$L_{\text{new}} = 8.125,$$

a significant reduction from the previous $L = 32.5$.

Two Layers Network, forward setup & assumptions (single sample)

$$
\begin{aligned}
&\text{Input: } x \in \mathbb{R}^{1\times d} \\
&z_1 = xW_1^\top + b_1 \quad (z_1 \in \mathbb{R}^{1\times h}), \\
&h = f_1(z_1) \quad (\text{pointwise}) \quad (h \in \mathbb{R}^{1\times h}), \\
&z_2 = hW_2^\top + b_2 \quad (z_2 \in \mathbb{R}^{1\times k}), \\
&\hat{y} = f_2(z_2) \quad (\text{pointwise}), \\
&\text{Squared Error (SE): } L(\hat{y}, y) = \tfrac{1}{2}\|\hat{y} - y\|_2^2.
\end{aligned}
$$

Shapes: $W_1 \in \mathbb{R}^{h\times d}$, $b_1 \in \mathbb{R}^{h}$, $W_2 \in \mathbb{R}^{k\times h}$, $b_2 \in \mathbb{R}^{k}$.

Full chain first (no shortcuts)

Goal: gradients w.r.t. parameters of both layers.

Second layer

$$\frac{\partial L}{\partial W_2} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{1\times k} \underbrace{\frac{\partial \hat{y}}{\partial z_2}}_{k\times k} \underbrace{\frac{\partial z_2}{\partial W_2}}_{k\times (kh)}, \qquad \frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial b_2}.$$

First layer

$$\frac{\partial L}{\partial W_1} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{1\times k} \underbrace{\frac{\partial \hat{y}}{\partial z_2}}_{k\times k} \underbrace{\frac{\partial z_2}{\partial h}}_{k\times h} \underbrace{\frac{\partial h}{\partial z_1}}_{h\times h} \underbrace{\frac{\partial z_1}{\partial W_1}}_{h\times (hd)}$$

(and analogously for $\partial L/\partial b_1$).

Compute the 5 Pieces (Pointwise Activations)

1) $\dfrac{\partial L}{\partial \hat{y}} = \hat{y} - y \in \mathbb{R}^{1\times k}$ (for MSE; replace accordingly for a generic loss).

2) $\dfrac{\partial \hat{y}}{\partial z_2} = \mathrm{diag}\big(f_2'(z_2)\big) \in \mathbb{R}^{k\times k}$.

3) $\dfrac{\partial z_2}{\partial h} = W_2 \in \mathbb{R}^{k\times h}$ since $z_2 = hW_2^\top$.

4) $\dfrac{\partial h}{\partial z_1} = \mathrm{diag}\big(f_1'(z_1)\big) \in \mathbb{R}^{h\times h}$.

5) $\dfrac{\partial z_1}{\partial W_1}$: as a Jacobian wrt $\mathrm{vec}(W_1) \in \mathbb{R}^{hd}$ (row-major):

$$\boxed{\frac{\partial z_1}{\partial \mathrm{vec}(W_1)} = I_h \otimes x^\top} \quad (h \times hd).$$

Introduce Deltas (δ_2, δ_1)

Define the layer-2 and layer-1 *pre-activation* gradients:

$$\delta_2 \;:=\; \frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}}\,\frac{\partial \hat{y}}{\partial z_2} = (\hat{y} - y) \;\odot\; f_2'(z_2) \;\in \mathbb{R}^{1\times k}.$$

$$\delta_1 \;:=\; \frac{\partial L}{\partial z_1} = \Big(\delta_2\, W_2\Big) \;\odot\; f_1'(z_1) \;\in \mathbb{R}^{1\times h}.$$

Assembling the chain for the first layer

Chain:

$$\frac{\partial L}{\partial \mathrm{vec}(W_1)} = \underbrace{\delta_2}_{1\times k}\,\underbrace{W_2}_{k\times h}\,\underbrace{\mathrm{diag}\big(f'(z_1)\big)}_{h\times h}\,\underbrace{(I_h \otimes x^\top)}_{h\times hd}.$$

Define the hidden-layer delta

$$\boxed{\delta_1 := (\delta_2\, W_2)\ \circ\ f'(z_1)} \quad (1 \times h),$$

then

$$\boxed{\frac{\partial L}{\partial W_1} = \delta_1^\top\, x} \quad (h \times d), \qquad \boxed{\frac{\partial L}{\partial b_1} = \delta_1} \quad (h).$$

Generalization: L Layers, Pointwise Activations

Forward for $l = 1, \dots, L$:

$$z^{(l)} = h^{(l-1)} {W^{(l)}}^\top, \qquad h^{(l)} = f^{(l)}\big(z^{(l)}\big), \quad h^{(0)} = x.$$

Define pre-activation deltas:

$$\delta^{(L)} = \frac{\partial L}{\partial h^{(L)}} \odot {f^{(L)}}'\big(z^{(L)}\big), \qquad \delta^{(l)} = \big(\delta^{(l+1)} W^{(l+1)}\big) \odot {f^{(l)}}'\big(z^{(l)}\big), \;\; l = L-1, \dots, 1.$$

Summary of all gradients for multilayer gradient descent

Quantity	Meaning	Formula
$\delta^{(L)}$	Error at output layer	$(\hat{y} - y) \odot f'(z^{(L)})$
$\delta^{(l)}$	Error at hidden layer l	$(\delta^{(l+1)} W^{(l+1)}) \odot f'(z^{(l)})$
$\dfrac{\partial L}{\partial W^{(l)}}$	Gradient w.r.t. weights	${\delta^{(l)}}^\top h^{(l-1)}$
$\dfrac{\partial L}{\partial b^{(l)}}$	Gradient w.r.t. biases	$\delta^{(l)}$
$\dfrac{\partial L}{\partial h^{(l)}}$	Gradient passed to previous layer	$\delta^{(l+1)} W^{(l+1)}$

Update rule for each layer:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}, \qquad b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial L}{\partial b^{(l)}}.$$

Note: $h^{(0)} = x$ (the input), and the recursion for δ continues backward until the first layer.

Lab time: Implement backpropagation in a two-layer network

- **Goal:** implement a simple two-layer neural network from scratch in NumPy.
- The network should have:

  Input: 2 $\rightarrow$ Hidden: 3 $\rightarrow$ Output: 3

- Use ReLU as activation for the hidden layer and a **linear output** for regression.
- Implement two main functions:
  - `forward(x)` - computes $\hat{y}$ and stores intermediate values for backward.
  - `backward(y, cache)` - computes all gradients using the chain rule.
  - `steps(grads, lr)` - update the weights using the calculated gradients.
- Perform the update step inside a simple loop to observe how the loss decreases over epochs.
- Suggested structure:
  1. Initialize W_1, b_1, W_2, b_2 with random values.
  2. Compute $z_1, h, z_2, \hat{y}$.
  3. Compute the loss $L = \frac{1}{2}\|\hat{y} - y\|^2$.
  4. Backpropagate and update parameters with learning rate η.
- **Challenge:** print the loss at each iteration and verify that it decreases.

My trivial implementation

```python
import numpy as np

def relu(u):
    return np.maximum(0.0, u)

def relu_grad(u):
    return (u > 0).astype(u.dtype)

class TwoLayerNet:
    def __init__(self, in_dim=2, hidden_dim=3, out_dim=3, seed=0):
        np.random.seed(seed)
        self.W1 = np.random.randn(hidden_dim, in_dim)
        self.b1 = np.zeros(hidden_dim)
        self.W2 = np.random.randn(out_dim, hidden_dim)
        self.b2 = np.zeros(out_dim)

    def forward(self, x):
        z1 = x @ self.W1.T + self.b1
        h  = relu(z1)
        z2 = h @ self.W2.T + self.b2
        y_hat = z2
        cache = {"x": x, "z1": z1, "h": h, "z2": z2, "y_hat": y_hat, "W2": self.W2.copy()}
        return y_hat, cache
```

My trivial implementation

```python
def backward(self, y, cache):
    x, z1, h, z2, y_hat, W2 = (
        cache["x"], cache["z1"], cache["h"], cache["z2"], cache["y_hat"], cache["W2"]
    )
    diff = y_hat - y
    loss = 0.5 * np.sum(diff**2)

    # Output layer
    delta2 = diff
    dW2 = np.outer(delta2, h)
    db2 = delta2

    # Hidden layer
    dh = delta2 @ W2
    delta1 = dh * relu_grad(z1)
    dW1 = np.outer(delta1, x)
    db1 = delta1

    grads = {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2}
    return grads, loss
```

My trivial implementation

```python
    def step(self, grads, lr=0.01):
        self.W1 -= lr * grads["W1"]
        self.b1 -= lr * grads["b1"]
        self.W2 -= lr * grads["W2"]
        self.b2 -= lr * grads["b2"]

# ---- training loop ----
net = TwoLayerNet(seed=0)
x = np.array([1.0, 2.0])
y = np.array([1.0, 4.0, 4.0])
lr = 0.1

for epoch in range(20):
    y_hat, cache = net.forward(x)
    grads, loss = net.backward(y, cache)
    net.step(grads, lr=lr)
    print(f"Epoch {epoch+1:02d} | Loss: {loss:.4f}")
```

From single sample to batch processing

- So far, we computed gradients for a **single training example**:

$$x \in \mathbb{R}^{d_{\text{in}}}, \qquad y \in \mathbb{R}^{d_{\text{out}}}.$$

- In practice, we process many samples at once using a **batch**:

$$X \in \mathbb{R}^{m \times d_{\text{in}}}, \qquad Y \in \mathbb{R}^{m \times d_{\text{out}}},$$

  where m is the batch size.

- Forward pass (for the whole batch):

$$Z^{(1)} = X W^{(1)^\top} + b^{(1)}, \qquad H = f(Z^{(1)}),$$

$$Z^{(2)} = H W^{(2)^\top} + b^{(2)}, \qquad \hat{Y} = f(Z^{(2)}).$$

- Loss (mean squared error for the batch):

$$L = \frac{1}{2m} \|\hat{Y} - Y\|_F^2.$$

Backpropagation over batches

- Compute the gradients for all samples in the batch:

$$\delta^{(2)} = (\hat{Y} - Y) \odot f'(Z^{(2)}),$$

$$\frac{\partial L}{\partial W^{(2)}} = \frac{1}{m}\delta^{(2)\top} H, \qquad \frac{\partial L}{\partial b^{(2)}} = \frac{1}{m}\sum_{i=1}^{m}\delta_i^{(2)},$$

$$\delta^{(1)} = (\delta^{(2)} W^{(2)}) \odot f'(Z^{(1)}), \qquad \frac{\partial L}{\partial W^{(1)}} = \frac{1}{m}\delta^{(1)\top} X.$$

- The gradient descent update remains identical:

$$W^{(l)} \leftarrow W^{(l)} - \eta\frac{\partial L}{\partial W^{(l)}}, \qquad b^{(l)} \leftarrow b^{(l)} - \eta\frac{\partial L}{\partial b^{(l)}}.$$

- Each row of $\delta^{(l)}$ corresponds to one sample in the batch.

Summary of all gradients for multilayer gradient descent (batch version)

Quantity	Meaning	Formula (batch size m)
$\Delta^{(L)}$	Error at output layer	$(\hat{Y} - Y) \odot f'(Z^{(L)})$
$\Delta^{(l)}$	Error at hidden layer l	$(\Delta^{(l+1)} W^{(l+1)}) \odot f'(Z^{(l)})$
$\dfrac{\partial L}{\partial W^{(l)}}$	Gradient w.r.t. weights	$\dfrac{1}{m} \Delta^{(l)\top} H^{(l-1)}$
$\dfrac{\partial L}{\partial b^{(l)}}$	Gradient w.r.t. biases	$\dfrac{1}{m} \sum_{i=1}^{m} \Delta_{i:}^{(l)}$
$\dfrac{\partial L}{\partial H^{(l)}}$	Gradient passed to previous layer	$\Delta^{(l+1)} W^{(l+1)}$

Update rule for each layer:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}, \qquad b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial L}{\partial b^{(l)}}.$$

Note:

- $X = H^{(0)}$ represents the batch input.
- Each row of $\Delta^{(l)}$ corresponds to one training sample.
- The factor $\frac{1}{m}$ ensures averaging over the batch.

Lab time: extend the code to support batches

Goal: modify your two-layer (2–3–3) network to process a **batch** of m samples instead of a single input.

Tasks

1. Update `forward()` to accept a batch X and return $\hat{Y}$ and the cached variables.
2. Update `backward()` to use the batched derivatives and average over m.

My trivial implementation

```python
class TwoLayerNet:

    def __init__(self, in_dim=2, hidden_dim=3, out_dim=3, seed=0):
        np.random.seed(seed)
        self.W1 = np.random.randn(hidden_dim, in_dim)
        self.b1 = np.zeros(hidden_dim)
        self.W2 = np.random.randn(out_dim, hidden_dim)
        self.b2 = np.zeros(out_dim)

    def forward(self, X):
        Z1 = X @ self.W1.T + self.b1
        H  = relu(Z1)
        Z2 = H @ self.W2.T + self.b2
        Y_hat = Z2

        cache = {
            "X": X, "Z1": Z1, "H": H, "Z2": Z2, "Y_hat": Y_hat,
            "W2": self.W2.copy()
        }
        return Y_hat, cache
```

My trivial implementation

```python
def backward(self, Y, cache):
    X, Z1, H, Z2, Y_hat, W2 = (
        cache["X"], cache["Z1"], cache["H"], cache["Z2"], cache["Y_hat"], cache["W2"]
    )
    m = X.shape[0]

    Diff = Y_hat - Y                            # (m, out_dim)
    loss = 0.5 / m * np.sum(Diff**2)

    # Output layer (linear): dY_hat/dZ2 = 1
    Delta2 = Diff                               # (m, out_dim)
    dW2 = (Delta2.T @ H) / m                    # (out_dim, hidden_dim)
    db2 = Delta2.mean(axis=0)                   # (out_dim,)

    # Hidden layer
    dH = Delta2 @ W2                            # (m, hidden_dim)
    Delta1 = dH * relu_grad(Z1)                 # (m, hidden_dim)
    dW1 = (Delta1.T @ X) / m                    # (hidden_dim, in_dim)
    db1 = Delta1.mean(axis=0)                   # (hidden_dim,)

    grads = {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2}
    return grads, loss
```

My trivial implementation

```
if __name__ == "__main__":
    net = TwoLayerNet(seed=0, hidden_dim=1000)
    n = 1000
    X = np.random.uniform(-10, 10, size=(n, 2))

    # Y: [x1**2, x2**2, 2*x1*x2]
    Y = np.column_stack((
        X[:, 0]**2,
        X[:, 1]**2,
        2 * X[:, 0] * X[:, 1]
    ))

    for epoch in range(10000):
        Y_hat, cache = net.forward(X/20)
        grads, loss = net.backward(Y, cache)
        net.step(grads, lr=0.01)
        print(f"Epoch {epoch+1:02d} | Loss: {loss:.4f}")

Y_hat,_ = net.forward([[3/20,-1/20]]) #Extepcted [9, 1, -6]
print(Y_hat)
```

Lab time: train your TwoLayerNet on MNIST

Goal: use your simple `TwoLayerNet` class (with linear output and MSE loss on one-hot labels) to train a neural network on the MNIST dataset.

Instructions: Load the MNIST dataset:

- Each image is 28×28, flatten it into a vector of 784 values.
- Normalize pixel values to the range $[0, 1]$.
- Convert integer labels (0–9) into one-hot vectors of length 10.

Create your model:

```
net = TwoLayerNet(in_dim=784, hidden_dim=128, out_dim=10)
```

Lab time: train your TwoLayerNet on MNIST

Train using the **entire dataset as a single batch**:

- Perform one `forward` pass on all samples.
- Compute gradients with `backward`.
- Apply the update step with `step`.
- Repeat for several epochs.

After each epoch:

- Print the average loss.
- Compute accuracy using `np.argmax(y_hat, axis=1)`.

Note: if the dataset does not fit in memory, you can split it into smaller mini-batches and repeat the same steps for each batch.

Challenge: Observe how the network learns over time - monitor how the loss decreases and accuracy increases, even with this simple setup.

My trivial implementation

```python
#...
def normalize_fit(X):
    mean = X.mean(axis=0, keepdims=True).astype(np.float32)
    std = X.std(axis=0, keepdims=True).astype(np.float32)
    std[std < 1e-6] = 1e-6
    return mean, std

def normalize_apply(X, mean, std):
    return (X - mean) / std
```

My trivial implementation

```python
if __name__ == "__main__":
    Xtr = np.load("train_images.npy").astype(np.float32)
    ytr = np.load("train_labels.npy").astype(np.int64)
    Xte = np.load("test_images.npy").astype(np.float32)
    yte = np.load("test_labels.npy").astype(np.int64)

    Xtr /= 255.0; Xte /= 255.0
    mean, std = normalize_fit(Xtr)
    Xtr = normalize_apply(Xtr, mean, std)
    Xte = normalize_apply(Xte, mean, std)
    ytr = np.eye(10)[ytr]
    model = TwoLayerNet(in_dim=784, hidden_dim=32, out_dim=10)
    for epoch in range(1000):
        Y_hat, cache = model.forward(Xtr)
        grads, loss = model.backward(ytr, cache)
        model.step(grads, lr=0.0005)
        print(f"Epoch {epoch+1:02d} | Loss: {loss:.4f}")

    Y_pred,_ = model.forward(Xte)
    Y_pred = np.argmax(Y_pred,axis=1)
    accuracy = np.sum(Y_pred==yte)/Y_pred.shape[0]
    print(f"Accuracy: {accuracy:.4f}")
```

Thanks!