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

—The Full Story—

1 Setup

We consider a shallow neural network $f_{\text{NN}}(\boldsymbol{x};\boldsymbol{\theta})$ trained on a dataset $\boldsymbol{X}$ with two explanatory variables. A single observation is a column vector

$$\boldsymbol{x}_{2\times 1} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

The network parameters are collected in

$$\boldsymbol{\theta} = \left\{ \boldsymbol{W}^{[1]}_{3\times 2},\ \boldsymbol{b}^{[1]}_{3\times 1},\ \boldsymbol{W}^{[2]}_{1\times 3},\ \boldsymbol{b}^{[2]}_{1\times 1} \right\}.$$

The network uses two types of activation functions:

$$\text{Hidden layer: } \text{ReLU}(z) = \max(0, z), \quad \text{Output layer: } \sigma(z) = \frac{1}{1+e^{-z}}.$$

What can this network represent? With three hidden neurons using ReLU activations, the network defines a piecewise-linear surface over the two-dimensional input space, partitioned into at most 7 distinct regions. These regions are then weighted and passed through the sigmoid activation, which maps the output to the interval $(0, 1)$.

Training the model: We initialize the 13 parameters in $\boldsymbol{\theta}$ using He-initialization. Training proceeds in a loop:

forward pass → loss evaluation → backprop → gradient update → forward pass ...

Each training cycle nudges the parameters in a direction that reduces the error, bringing the predictions closer to the data. Repeating this process many many times allows the network to gradually sculpt a function that captures the underlying pattern.

Let us now begin with the forward pass.

2 Forward Pass

The forward pass is straightforward: it means computing the network's prediction for a given input with the current parameters. In this section, we focus on stochastic gradient descent (SGD), where a single observation is processed at a time (for batch gradient descent, multiple observations would be used simultaneously).
In essence, the forward pass consists of taking the input vector, passing it through each layer in sequence, and obtaining the output. It is nothing more than function evaluation: plugging an observation into the parameterized function to produce a prediction.

Formally, our network is nothing more than a nested function:

$$f_{\text{NN}}(\boldsymbol{x};\boldsymbol{\theta}) = \sigma\Big(\boldsymbol{W}^{[2]}\,\text{ReLU}(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}) + \boldsymbol{b}^{[2]}\Big).$$

This compact vector notation is convenient, but it can feel abstract. To see what is really happening, let us write the same function explicitly in scalar form where now each term is just a single number:

$$\begin{aligned} f_{\text{NN}}(x_1, x_2;\boldsymbol{\theta}) = \sigma\Big(& w_1^{[2]}\,\text{ReLU}(w_{11}^{[1]}x_1 + w_{12}^{[1]}x_2 + b_1^{[1]}) \\ &+ w_2^{[2]}\,\text{ReLU}(w_{21}^{[1]}x_1 + w_{22}^{[1]}x_2 + b_2^{[1]}) \\ &+ w_3^{[2]}\,\text{ReLU}(w_{31}^{[1]}x_1 + w_{32}^{[1]}x_2 + b_3^{[1]}) \\ &+ b^{[2]}\Big). \end{aligned}$$

Hence, a neural network is ultimately just a function of the inputs and the parameters. This is directly analogous to the classical linear regression model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2,$$

where the prediction depends only on the inputs x_1, x_2 and the parameters $\boldsymbol{\theta} = \begin{bmatrix}\beta_0 & \beta_1 & \beta_2\end{bmatrix}^T$. In this notation, β_0 acts as a bias term, while β_1 and β_2 serve as weights.
The key difference in neural networks is the introduction of non-linear activation functions and the stacking of multiple such layers. Conceptually, however, the principle remains the same: linear combinations followed by transformations, repeated in sequence. The resulting formulas may look more complex, but at their core neural networks are still just "inputs go in, output comes out."

Because every step of the computation is explicit and differentiable, the whole function is well-defined at every input, and we can systematically compute derivatives with respect to each parameter. Nothing intimidating is happening here. It is simply a larger, nested version of functions you have already seen.

The forward pass unfolds layer by layer as follows:

$$\boldsymbol{f}^{[0]}_{3\times 1} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]} \qquad \text{(pre-activations of the hidden layer)}$$

$$\boldsymbol{h}^{[1]}_{3\times 1} = \mathrm{ReLU}\left(\boldsymbol{f}^{[0]}\right) \qquad \text{(hidden activations)}$$

$$f^{[1]}_{1\times 1} = \boldsymbol{W}^{[2]}\boldsymbol{h}^{[1]} + b^{[2]} \qquad \text{(pre-activation of the output layer)}$$

$$\hat{y}_{1\times 1} = \sigma\left(f^{[1]}\right) \qquad \text{(final prediction)}$$

Notice that $\boldsymbol{x}$ and $\hat{y}$ are written without a superscript, whereas all other quantities carry one. This is because the superscript indicates the *layer index*. For example, $\boldsymbol{W}^{[k]}$ denotes the weight matrix connecting layer $k-1$ to layer k. Since the input $\boldsymbol{x}$ always comes from the 0-th layer and the prediction $\hat{y}$ is always the final output, it is redundant to assign them a superscript. Only the intermediate quantities inside the network carry layer indices which could easily be increased if we would make the network deep.

Also note the dimensionality of all the objects. The input $\boldsymbol{x}$, the hidden activations $\boldsymbol{h}^{[k]}$, and the output $\hat{y}$ are all represented as column vectors (with $\hat{y}$ being a 1×1 scalar in this example). This choice of column-vector convention is important, because it determines how matrix multiplication works in both the forward and backward passes.
Moreover, each weight matrix has the shape

$$\boldsymbol{W}^{[k]} \in \mathbb{R}^{\,n_k \times n_{k-1}},$$

where n_{k-1} is the number of inputs from the previous layer and n_k is the number of neurons in the current layer.

In linear-algebra terms, the expression $\boldsymbol{W}^{[k]}\boldsymbol{h}^{[k-1]}$ means this: the weight matrix $\boldsymbol{W}^{[k]}$ is a linear transformation acting on the activations from the previous layer. Geometrically, it can stretch, rotate or flip the input vector, re-expressing the same point in a new coordinate system. Those transformed coordinates are the pre-activations of layer k, which the nonlinearity will then reshape further. Therefore, each layer linearly re-expresses and then folds the input space, gradually arranging it so that the initial inputs lie close in high-dimensional space to where the solution lives.

The result of the forward pass is the network's current prediction for a single input,

$$\hat{y} = f_{\mathrm{NN}}(\boldsymbol{x}; \boldsymbol{\theta}^{(t)}),$$

where $\boldsymbol{\theta}^{(t)}$ denotes the parameters at iteration t of training.

To determine how strongly, and in which direction, the parameters should be adjusted, we need a measure of how far our predictions are from the truth. This is precisely the role of the *loss function*, which we discuss next.

3 Loss Evaluation

For a single training example with input $\boldsymbol{x}_i$, the network produces the prediction

$$\hat{y}_i = f_{\text{NN}}(\boldsymbol{x}_i; \boldsymbol{\theta}).$$

To evaluate how well this prediction matches the true label, we introduce a *per-sample loss function*. In regression[1],

a natural choice is the squared error:

$$l_i(\hat{y}_i; y_i) = (\hat{y}_i - y_i)^2.$$

Multiplying with $\frac{1}{2}$ for convenience in taking derivatives, equivalently:

$$l_i(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = \tfrac{1}{2}\big(f_{\text{NN}}(\boldsymbol{x}_i; \boldsymbol{\theta}) - y_i\big)^2.$$

Note the distinction:

- $\hat{y}_i = f(\boldsymbol{x}_i; \boldsymbol{\theta})$ is a function of the input, given fixed parameters.
- $l_i(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)$ is a function of the parameters, given the data $(\boldsymbol{x}_i, y_i)$.

In stochastic gradient descent (SGD), training updates use l_i from a single observation as an approximation to the full objective. The true training goal, however, is to minimize the *empirical risk* i.e. the average loss across the entire dataset:

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} l_i(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i).$$

Thus, while each per-sample loss is a moving target that depends on which observation is considered at a given step, the global objective is fixed: find a parametrization that minimizes the average loss.

The loss function defines a high-dimensional *landscape*. For a simple linear model with two parameters, e.g. $y = \beta_0 + \beta_1 x$, the MSE surface is convex and bowl-shaped, with a unique minimum (at $\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T y$). In contrast, the loss landscape of a neural network is highly non-convex, with many valleys, ridges, and local minima and does not have a closed-form solution. Training does therefore not seek the global minimum, but rather converges to a sufficiently flat region, over the course of training, where predictions generalize well.

Every weight and bias influences the loss through the nested chain of computations in the forward pass. Since the entire network is just a composition of differentiable functions, we can systematically apply the chain rule to compute the gradients with respect to every parameter. This realization—that the loss can be differentiated all the way back through the network—is the central insight behind backpropagation. Once these gradients are known, we know exactly how to adjust the parameters to reduce the loss.

Let's derive them now!

[1] A sigmoid output activation is most commonly used in *classification* settings, where the appropriate loss is the cross-entropy rather than the mean squared error (MSE). In our tutorial problem, however, the task is to predict a continuous production function: $Y = AK^{\alpha}L^{1-\alpha}$ with $A \sim Beta(a, b)$ and $K, L \in [0, 1]$ making this a regression problem and justify using MSE.

4 Backward Pass

We begin at the very end of the network, which is why the method is called backpropagation. For the output layer, the first step is to compute the derivative of the loss with respect to its pre-activation $f^{[1]}$:

$$\frac{\partial l}{\partial f^{[1]}} = \frac{\partial \hat{y}}{\partial f^{[1]}} \frac{\partial l}{\partial \hat{y}} = \sigma(f^{[1]})(1 - \sigma(f^{[1]})) \odot (\hat{y} - y)$$

This quantity is called the *error signal* of the final layer. It measures how the loss changes when we change $f^{[1]}$ marginally and simply follows from applying the chain-rule. Error signals play a crucial role in backpropagation: once computed for one layer, they are used both to form gradients with respect to that layer's weights and biases, and to propagate the error backward to earlier layers as we will see shortly.

Note how the chain rule is written down from right to left. For example, in the scalar case with $y = f(x)$ and $z = g(y)$ we usually write $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$. Under the column-vector convention, however, matrix calculus gives $\frac{\partial z}{\partial x} = \left(\frac{\partial y}{\partial x}\right)^\top \frac{\partial z}{\partial y}$. That's why we use the right-to-left concention throughout. This is also intuitively appealing, since gradients naturally propagate backward through the sequence of operations from right to left. Finally, because activations are applied element-wise, their derivatives must also be taken element-wise—hence the Hadamard product ($\odot$) appears in the formulas (when calculating activations from pre-activations inside neurons).

Now that we have the error signal of the output layer, we can compute the gradients with respect to the weights and biases that feed into it. Recall that

$$f^{[1]} = \boldsymbol{W}^{[2]} \boldsymbol{h}^{[1]} + b^{[2]}.$$

It follows that

$$\frac{\partial l}{\partial \boldsymbol{b}^{[2]}} = \frac{\partial f^{[1]}}{\partial \boldsymbol{b}^{[2]}} \frac{\partial l}{\partial f^{[1]}} = \frac{\partial l}{\partial f^{[1]}} \qquad \text{and} \qquad \frac{\partial l}{\partial \boldsymbol{W}^{[2]}} = \frac{\partial f^{[1]}}{\partial \boldsymbol{W}^{[2]}} \frac{\partial l}{\partial f^{[1]}} = \frac{\partial l}{\partial f^{[1]}} \boldsymbol{h}^{[1]\top}.$$

Two quick sanity checks make these formulas intuitive:

- **Bias:** Nudging $b^{[2]}$ by ε shifts $f^{[1]}$ by exactly ε, so l changes by $\left(\frac{\partial l}{\partial f^{[1]}}\right)\varepsilon$. Hence the bias gradient equals the error signal.
- **Weights:** Nudging only the j-th weight by ε changes $f^{[1]}$ by $h_j^{[1]}\varepsilon$, so l changes by $\left(\frac{\partial l}{\partial f^{[1]}}\right) h_j^{[1]} \varepsilon$. Column-wise, this is the outer product "error signal $\times$ activation," i.e. $\frac{\partial l}{\partial \boldsymbol{W}^{[2]}} = \left(\frac{\partial l}{\partial f^{[1]}}\right) \boldsymbol{h}^{[1]\top}$. Inactive units ($h_j^{[1]} \approx 0$) contribute almost no gradient.

In short: biases shift the output baseline (they learn the offset), while weights transmit the error in proportion to the signal they carried (they learn the scaling of each hidden activation).

With the error signal $\frac{\partial l}{\partial f^{[1]}}$ in hand, the next step is to push it one layer back in order to compute the gradients with respect to $\boldsymbol{W}^{[1]}$ and $\boldsymbol{b}^{[1]}$ as well.

To do so, we first need the derivative of the loss with respect to the hidden layer's pre-activations $\boldsymbol{f}^{[0]}$, so the error signal of the hidden layer. Applying the chain rule gives:

$$\frac{\partial l}{\partial \boldsymbol{f}^{[0]}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{f}^{[0]}} \frac{\partial f^{[1]}}{\partial \boldsymbol{h}} \frac{\partial l}{\partial f^{[1]}} = \mathbb{I}[\, \boldsymbol{f}^{[0]} > 0 \,] \odot \left({\boldsymbol{W}^{[2]}}^{\top} \frac{\partial l}{\partial f^{[1]}} \right).$$

Here, $\mathbb{I}[\, \boldsymbol{f}^{[0]} > 0 \,]$ is the derivative of the ReLU, again applied element-wise. This formula illustrates the central efficiency of backpropagation: instead of recomputing everything from scratch, we simply reuse the error signal $\frac{\partial l}{\partial f^{[1]}}$ from the output layer, pass it backward using ${\boldsymbol{W}^{[2]}}^{\top}$, and modulate it by the derivative of the activation function, where have made use of:

$$\frac{\partial f^{[1]}}{\partial \boldsymbol{h}} \frac{\partial l}{\partial f^{[1]}} = {\boldsymbol{W}^{[2]}}^{\top} \frac{\partial l}{\partial f^{[1]}},$$

which makes intuitive sense again: the error signal is distributed back to the hidden units in proportion to how strongly each of them contributed to $f^{[1]}$.

Finally, we can then compute the gradients with respect to $\boldsymbol{W}^{[1]}$ and $\boldsymbol{b}^{[1]}$ as:

$$\frac{\partial l}{\partial \boldsymbol{b}^{[1]}} = \frac{\partial \boldsymbol{f}^{[0]}}{\partial \boldsymbol{b}^{[1]}} \frac{\partial l}{\partial \boldsymbol{f}^{[0]}} = \frac{\partial l}{\partial \boldsymbol{f}^{[0]}} \qquad \text{and} \qquad \frac{\partial l}{\partial \boldsymbol{W}^{[1]}} = \frac{\partial \boldsymbol{f}^{[0]}}{\partial \boldsymbol{W}^{[1]}} \frac{\partial l}{\partial \boldsymbol{f}^{[0]}} = \frac{\partial l}{\partial \boldsymbol{f}^{[0]}} \boldsymbol{x}^{\top}.$$

Summarizing, the crucial step in backpropagation is obtaining the derivative of the loss with respect to each layer's pre-activations. Once this error signal is known, everything follows almost automatically:

- The gradient with respect to the biases is just the error signal itself.
- The gradient with respect to the weights is the error signal post-multiplied by the activations of the previous layer (reflecting the linear weighting).
- To propagate the error signal back one layer, we pre-multiply it by the transpose of the weight matrix and then apply the activation derivative element-wise.

The most important quantity in backpropagation is therefore the *error signal*:

$$\boldsymbol{\delta}^{[k]} = \frac{\partial l}{\partial \boldsymbol{f}^{[k]}},$$

It acts as the messenger that carries information about how much each neuron's pre-activation contributed to the final loss. Backpropagation is nothing more than computing, reusing, and propagating this error signal from the output all the way back to the input. If you remember one thing, remember this:
backprop is the journey of the error signal through the network.

Algorithm 1 Backpropagation for a general L-layer neural network

Require: Stored pre-activations $\{\boldsymbol{f}^{[k]}\}_{k=0}^{L-1}$ and activations $\{\boldsymbol{h}^{[k]}\}_{k=1}^{L}$ from the forward pass loss $l(\hat{y}, y)$ and activation derivatives $g'^{[k]}$.

$\boldsymbol{\delta}^{[L-1]} \leftarrow \dfrac{\partial \hat{y}}{\partial \boldsymbol{f}^{[L-1]}} \odot \dfrac{\partial l}{\partial \hat{y}}$ ▷ Initialize output-layer error signal

for $k = L-1$ down to 1 **do**

  $\dfrac{\partial l}{\partial \boldsymbol{W}^{[k+1]}} \leftarrow \boldsymbol{\delta}^{[k]}\, \boldsymbol{h}^{[k]\top}$ ▷ Calculate weights gradient

  $\dfrac{\partial l}{\partial \boldsymbol{b}^{[k+1]}} \leftarrow \boldsymbol{\delta}^{[k]}$ ▷ Calculate bias gradient

  $\boldsymbol{\delta}^{[k-1]} \leftarrow g'^{[k]}(\boldsymbol{f}^{[k-1]}) \odot \left(\boldsymbol{W}^{[k+1]\top} \boldsymbol{\delta}^{[k]}\right)$ ▷ Propagate error signal

end for

$\dfrac{\partial l}{\partial \boldsymbol{W}^{[1]}} \leftarrow \boldsymbol{\delta}^{[0]}\, \boldsymbol{x}^{\top}, \quad \dfrac{\partial l}{\partial \boldsymbol{b}^{[1]}} \leftarrow \boldsymbol{\delta}^{[0]}$ ▷ First layer gradients

We now know how a small change in the network parameters would affect the loss. This is the essential purpose of backpropagation: to translate the global training objective (minimizing the loss) into precise, local adjustment rules for each weight and bias. In other words, backprop tells us the direction of steepest ascent of the loss in parameter space. To actually train the network, we must now take the opposite direction: a downhill step, so that the parameters move in a way that reduces the error. This is the role of gradient descent.

5 Gradient Descent Update

Formally, for each parameter $\theta \in \boldsymbol{\theta}$ the update rule at iteration t is

$$\theta^{(t+1)} = \theta^{(t)} - \eta \, \frac{\partial l}{\partial \theta^{(t)}},$$

where $\eta > 0$ is the learning rate, a hyperparameter that controls the step size. A large η makes fast but potentially unstable progress; a small η makes slow but steady progress.

For our two-layer network, the updates are

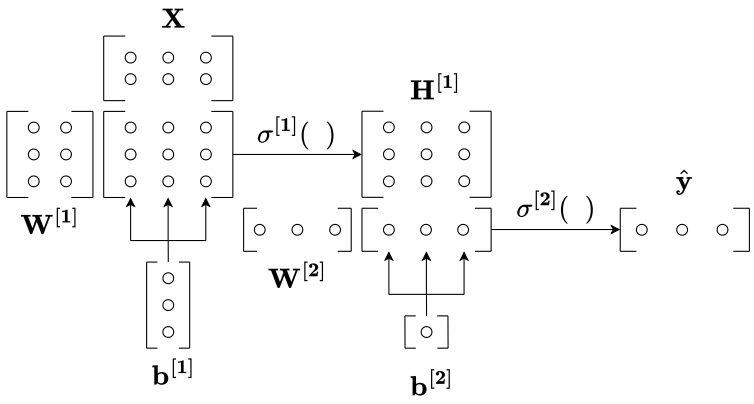
$$\boldsymbol{W}^{[2]} \leftarrow \boldsymbol{W}^{[2]} - \eta \, \frac{\partial l}{\partial \boldsymbol{W}^{[2]}}, \quad \boldsymbol{b}^{[2]} \leftarrow \boldsymbol{b}^{[2]} - \eta \, \frac{\partial l}{\partial \boldsymbol{b}^{[2]}},$$

$$\boldsymbol{W}^{[1]} \leftarrow \boldsymbol{W}^{[1]} - \eta \, \frac{\partial l}{\partial \boldsymbol{W}^{[1]}}, \quad \boldsymbol{b}^{[1]} \leftarrow \boldsymbol{b}^{[1]} - \eta \, \frac{\partial l}{\partial \boldsymbol{b}^{[1]}}.$$

Over many epochs, this iterative cycle gradually adjusts the parameters $\boldsymbol{\theta}$ so that the network's predictions come increasingly close to the training data. Once the parameters have converged to a useful configuration, we say the network is *trained*. At that point, it can be applied to new, unseen inputs to generate predictions, relying on the patterns it has extracted from the training set.

6 Batch Gradient Descent

Up to this point, we have only considered training with a single observation at a time. But in practice, it is far more efficient to process multiple examples simultaneously. We can exploit the power of linear algebra to handle a whole batch of B observations in parallel. This is called batch gradient descent.



The figure above illustrates the flow of data in our two-layer network under batch gradient descent. Here the input matrix $\boldsymbol{X}$ stacks B observations column-wise, so that both the hidden activations and the outputs naturally carry B entries in their second dimension.

The forward pass yields hidden activations $\boldsymbol{H}^{[1]}$ and predictions $\hat{\boldsymbol{y}}$ for all training examples simultaneously:

$$\boldsymbol{H}^{[1]} = \sigma^{[1]}(\boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{b}^{[1]}), \qquad \hat{\boldsymbol{y}} = \sigma^{[2]}(\boldsymbol{W}^{[2]}\boldsymbol{H}^{[1]} + \boldsymbol{b}^{[2]}).$$

Gradients with respect to $\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}, \boldsymbol{W}^{[2]}, \boldsymbol{b}^{[2]}$ are then computed by *averaging across the batch*, ensuring that updates reflect the overall trend in the data rather than single-sample noise.

Formally, the loss gradient with respect to the predictions is

$$\frac{\partial l}{\partial \hat{\boldsymbol{y}}}_{1\times B} = \tfrac{1}{B}\left[(\hat{y}_1 - y_1)\ (\hat{y}_2 - y_2)\ \ldots\ (\hat{y}_B - y_B)\right].$$

From here, backpropagation works just as in the single-sample case:

- $\frac{\partial l}{\partial \boldsymbol{W}^{[2]}}$ is the error signal times the hidden activations $\boldsymbol{H}^{[1]^T}$.
- $\frac{\partial l}{\partial \boldsymbol{b}^{[2]}}$ requires averaging, which is achieved by:

$$\frac{\partial l}{\partial \boldsymbol{b}^{[2]}} = \tfrac{1}{B}\,\boldsymbol{\delta}^{[2]}\mathbf{1}_B.$$

- Error propagation to the hidden layer again uses pre-multiplying by $\boldsymbol{W}^{[2]\top}$.