ChatGPT

Backpropagation Unveiled: A Personal Journey from Confusion to Clarity

When I first heard about **backpropagation**, it sounded like some mysterious magic in neural networks. How could machines "learn" by somehow sending errors backwards? It felt overwhelming at first. But step by step, with some analogies and a lot of tinkering, the pieces fell into place. In this post, I'll walk you through the backpropagation algorithm *as if I'm explaining it to a friend* – in simple, honest terms. We'll use real-world analogies, examples, and even cute diagrams to make each concept click.

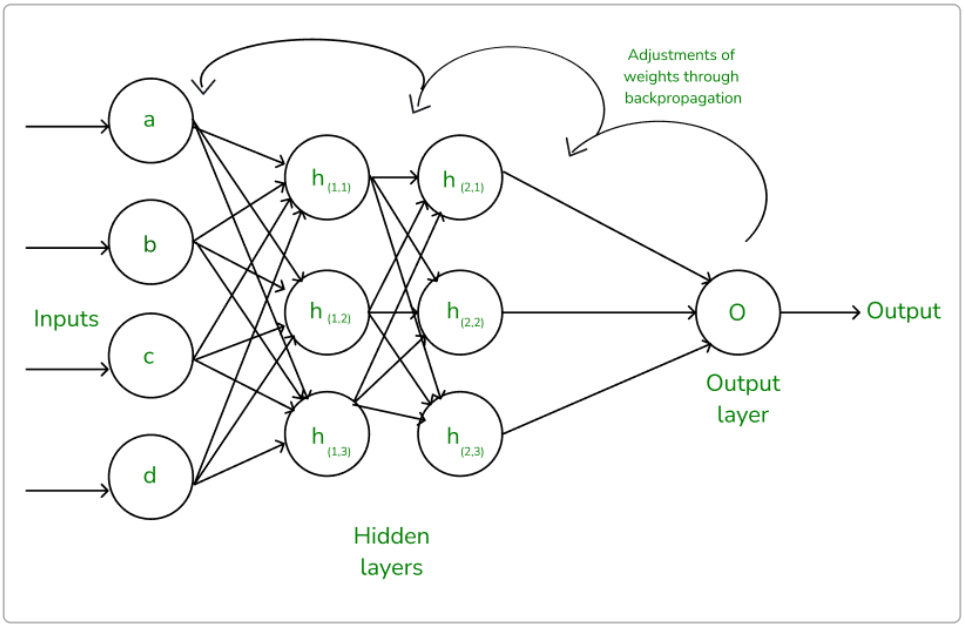

Figure: A simple multi-layer neural network. Inputs (a, b, c, d) feed into hidden neurons (h_{11}, h_{12}, h_{13}, etc.), which then feed into the output O. During training, backpropagation sends feedback (green arrows) from the output back through the network to adjust the weights.

Imagine a neural network like a series of decision layers. Each circle above (like *a, b, c, d*, h_{11}, h_{23}, etc.) is a **neuron** or **node**. The arrows connecting them are **weights**, which are like knobs controlling how strongly one neuron influences the next. When we feed an input through the network (a process called the *forward pass*), each neuron does a little calculation: it takes all its incoming values, multiplies each by its weight, adds a **bias** (like a baseline adjustment), and then runs the sum through an **activation function** (like ReLU or sigmoid) to produce its output.

In other words, each neuron is like a recipe step. The inputs are ingredients, the weights say "how much of each ingredient to use," and the bias is an extra pinch of flavor. The activation function is like the cooking process that transforms the mixture (for example, a threshold that turns the mixture sweet or dull). The

beauty is that a neuron can *nudge* the sum one way or another, and then decide if the signal is “turned on” or “off” (or somewhere in between), depending on the activation.

Here are the key pieces in plain language:

- **Inputs and Weights:** Each input (like *a*, *b*, *c*, *d*) is multiplied by a weight. Think of it as each ingredient in a recipe being scaled by a certain amount. A large weight means that input has a strong effect.
- **Bias:** The bias is like adding a fixed ingredient no matter what, say a spoonful of salt even if you used zero sugar. It allows the neuron to shift its threshold up or down.
- **Summation:** The neuron adds up all the (input × weight) terms *plus* the bias. This is the total “activation” before the non-linear step.
- **Activation Function:** After summing, the neuron applies a function (sigmoid, ReLU, tanh, etc.) to the total. This squashes or transforms the number into a range that the network can use (for example, between 0 and 1 for a sigmoid). It’s like taking the batter of a cake and baking it into something you can taste and evaluate.

Putting it all together, if you look at one hidden neuron, its output might be:

[\text{neuron output} = \sigma(w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b),]

where (x_i) are inputs, (w_i) are weights, (b) is bias, and (\sigma) is the activation. You don’t need to memorize the formula; just remember the flow: *multiply inputs by weights, add bias, apply activation*.

Forward Pass: Making Predictions

With those pieces in place, the **forward pass** is straightforward (though still fun to visualize). We feed the network some input, and it propagates forward through the layers to produce an output. Here’s an example:

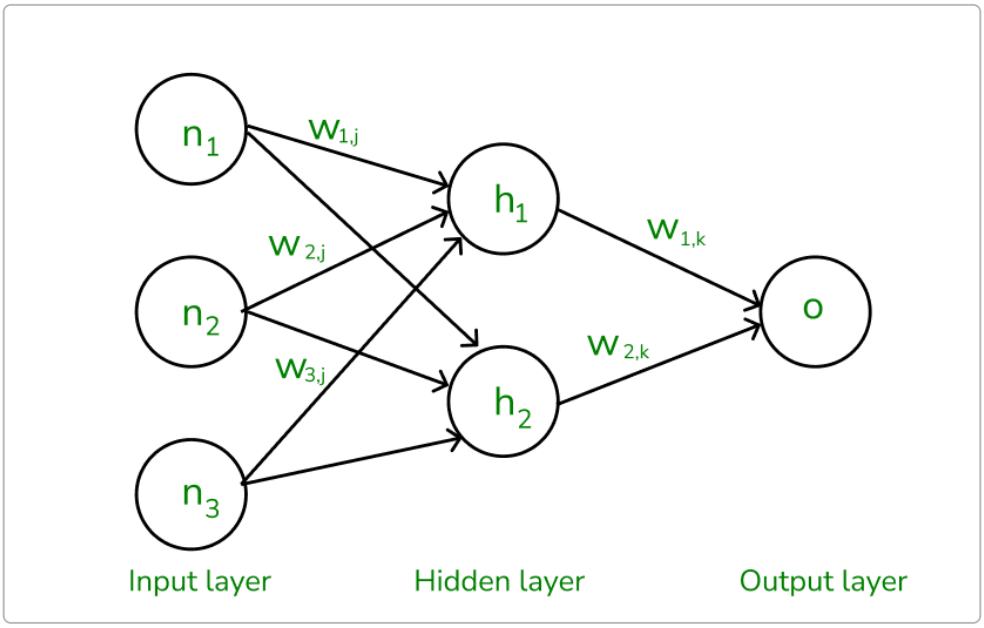

Figure: A tiny neural network with 3 inputs (n_1, n_2, n_3), 2 hidden neurons (h_1, h_2), and 1 output (O). Each arrow has a weight label. During the forward pass, inputs are multiplied by weights, summed, and passed through an activation function to yield each neuron's output.

In the picture above, suppose the inputs are (n_1, n_2, n_3). Each hidden node (h_1 or h_2) sums up (\text{(inputs × their weights)}), adds a bias, and then squashes it through an activation. The outputs of h_1 and h_2 then become inputs to the final output neuron O, which does the same weighted-sum-plus-activation step to produce the final prediction.

Let's do a mini numeric example (don't worry, the numbers will make it feel concrete, but the main takeaway is the process). Consider the small network below with just two inputs, two hidden neurons, and one output. The weights are shown (e.g. (w_{1,1} = 0.2), etc.) and let's say the inputs are (x_1 = 0.35) and (x_2 = 0.7).

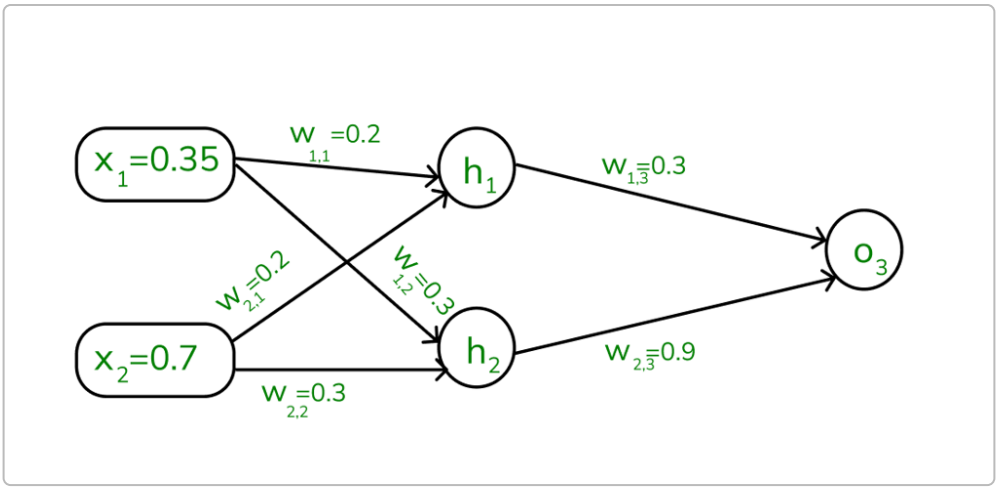

Figure: Our example network before computing. Inputs (x_1=0.35), (x_2=0.7) feed into two hidden neurons (h_1, h_2) with initial weights (green). The final output is O. (We'll show the computed outputs on the next diagram.)

Starting the forward pass, each hidden neuron does:
- **Hidden neuron h_1:** Compute weighted sum (a_1 = 0.2 \times 0.35 + 0.2 \times 0.7 + b_{h1}). (Assume its bias (b_{h1}) is included; let's say it results in (a_1=0.21).) Then apply the activation (sigmoid) to get (y_3 = \sigma(a_1) \approx 0.56).
- **Hidden neuron h_2:** Compute (a_2 = 0.3 \times 0.35 + 0.3 \times 0.7 + b_{h2}). (Say it gives (a_2=0.315).) After sigmoid, (y_4 = \sigma(a_2) \approx 0.59).

Now these hidden outputs (y_3, y_4) go to the output neuron O:
- **Output neuron O:** Compute (a_3 = 0.3 \times y_3 + 0.9 \times y_4 + b_O). (This might equal roughly (0.702).) After activation, the final output is (y_5 = \sigma(a_3) \approx 0.67).

We can see all these computed outputs in the diagram:

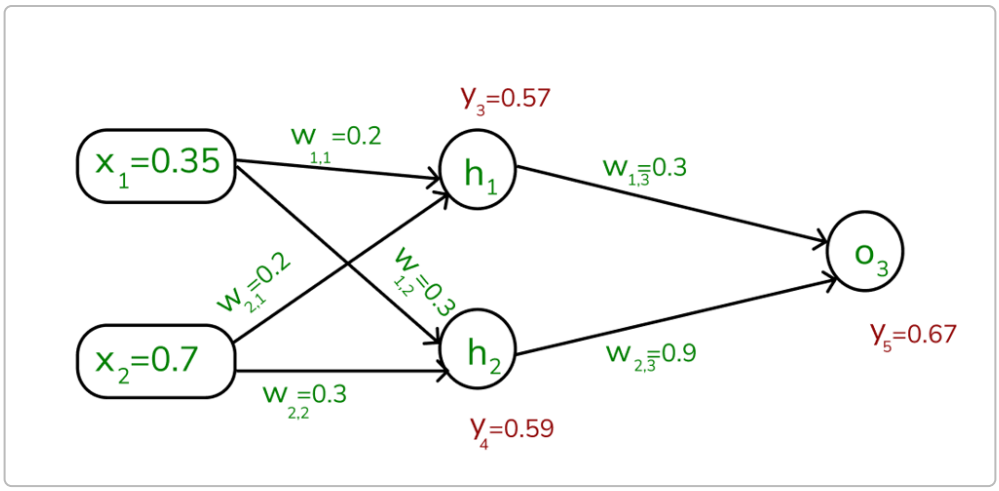

Figure: After the forward pass: the hidden neurons produced outputs (y_3 \approx 0.57) and (y_4 \approx 0.59) (red), and the final output is (y_5 \approx 0.67). These results come from summing the inputs × weights and applying the activation.

According to the math above (see [1]), (y_3 \approx 0.56), (y_4 \approx 0.59), and the network's prediction is (y_5 \approx 0.67). To give some context, let's say our target (desired) value for this example is (0.5). That means our network overshot a bit: it predicted (0.67) instead of (0.5).

Loss Calculation: Measuring Our Mistake

Now that we have a prediction ((y_5 = 0.67)) and a target ((0.5)), we need to quantify *how far off* we are. This is the **loss** or **error**. A simple way is the **Mean Squared Error** (MSE):

[\text{MSE} = (\text{predicted} - \text{actual})^2.]

In our example, that's ((0.67 - 0.5)^2). The network made an error of (0.17) (actually (0.5 - 0.67 = -0.17) if we do target minus output), and squaring it emphasizes bigger mistakes. Indeed, geometrically you can think of this as how far you missed the bullseye on a dartboard. Here, our "miss distance" is (0.17) (the network predicted higher than it should have).

The key is: **the loss tells us how bad the output is**. For backpropagation, we usually use something like MSE. As GeeksforGeeks notes, *"One common method for error calculation is the Mean Squared Error (MSE) given by: MSE = (Predicted Output – Actual Output)²"* [2]. We could use other loss functions too (like cross-entropy), but the idea is the same: get a number that says how wrong we are.

Analogy: Imagine shooting basketball free throws. The loss is like how far away from the hoop your last shot was. If you made the perfect shot, the loss is zero. If you miss, the loss is the distance or "how off" your shot was. Here our shot was a little off by 0.17.

Backward Pass: Learning from Mistakes

Okay, so we know our prediction was off. What do we do with that information? This is where **backpropagation** (the backward pass) comes in. The goal of backpropagation is to send that error back through the network and figure out *which weights need adjusting and by how much*.

Think of the backward pass like peeling an onion: you start from the final error at the output and peel backwards through each layer. Concretely, we compute an **error term** (often called delta, (\delta)) for each neuron. For the output neuron, it's simple: its delta is basically the derivative of the loss with respect to its input. For example, with MSE and sigmoid activation, the output delta (\delta_O) might be ((y_{\text{target}} - y_5) \times \sigma'(a_3)), which numerically was about (-0.0376) in our example.

Then we go one layer back: each hidden neuron's error is found by looking at how much that neuron contributed to the output's error. In math, each hidden delta is something like (\delta_{h_i} = \sigma'(a_i) \times \sum (\delta_O \times w_{h_i,O})). In words, the hidden neuron takes the error from the output, multiplies by its connecting weight (because that weight controlled how much *its* output influenced the final error), and also factors in the derivative of its own activation. The chain rule from calculus is what justifies these steps.

GeeksforGeeks nicely summarizes: *"Once the error is calculated the network adjusts weights using gradients which are computed with the chain rule... These gradients indicate how much each weight and bias should be adjusted to minimize the error..."* [3] . Basically, the math is just bookkeeping with derivatives. You end up with a gradient (slope) for each weight, telling us which way to nudge it to reduce the error.

Analogy: I like to think of the backward pass like a teacher grading a multi-part problem. The student got a final score, and the teacher asks "Which steps led to the mistakes?" The teacher traces back: if the final answer is too big, maybe the step before multiplied by a wrong factor, etc. Backpropagation is doing that mechanically: passing "blame" backwards.

For our example network, the backward pass calculated how much each weight contributed to the 0.17 error and in which direction. (We won't bore you with the exact numbers here, but you can see the idea: each weight's update is proportional to its role in the error.) After running backpropagation, we figure out small adjustments (\Delta w) for each weight.

Weight Updates: Putting It Into Practice

Once we have the gradients from the backward pass, we *update* the weights. Typically, we do something like:

[w_{\text{new}} = w_{\text{old}} - \eta \, (\text{gradient}),]

where (\eta) is the **learning rate** (a small number like 0.1 or 0.01) that prevents us from making huge jumps. In essence, we move each weight a little in the direction that reduces the loss.

Continuing our example, after one round of backpropagation we got new weights. The changes were tiny – in fact, one weight (w_{1,1}) changed from 0.20 to about 0.200945 (a change of +0.000945) [4]. That's almost imperceptible, but it's the right direction. (You can see the updated values in the next diagram.)

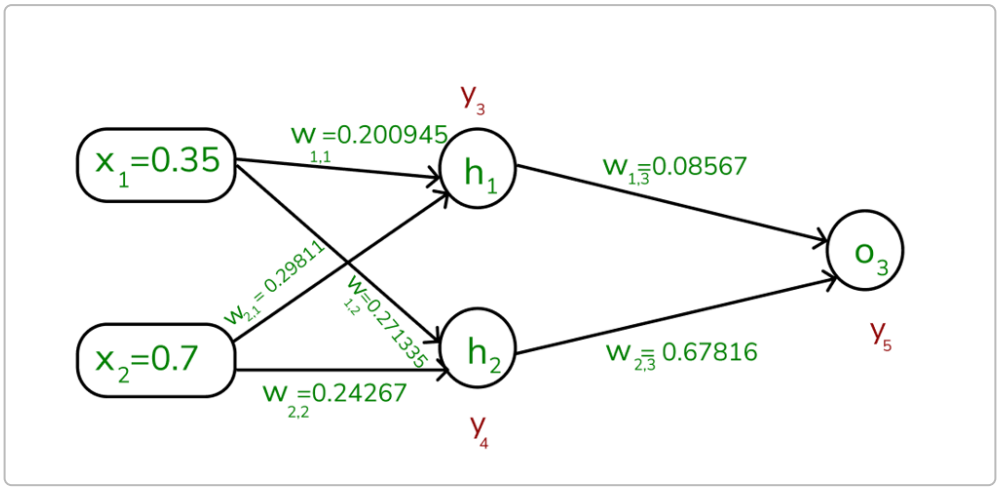

Figure: After the backward pass, the weights have been updated slightly (new values in green). For instance, (w_{1,1}) changed from 0.20 to 0.200945. These small adjustments will push the output closer to the target on the next forward pass.

After updating, we would do another forward pass with the new weights. In our example, the output moves a bit nearer the target (now around 0.61 instead of 0.67, as GeeksforGeeks notes), then we would compute error again and repeat. This loop of forward pass → loss → backward pass → weight update happens many times (many **epochs**) until the network is predicting well. Over time, the loss shrinks and the model "learns."

Analogy: Think of tuning a guitar by ear. You pluck a string (forward pass) and maybe it's sharp by a tiny amount. You then adjust the tuning peg just a hair (backward pass telling you how much to adjust). Next time you pluck, it's closer to in-tune. You repeat until it sounds right. The machine does the same with weights and errors.

At first, the idea of tweaking dozens or thousands of weights slightly seemed crazy to me. But it makes sense: each weight is like a dial. When the output was too high, backprop nudged the dials so the next try will be a bit lower. Eventually the network finds a combination where it "always" (hopefully) hits near the bullseye.

Why Backpropagation Is Efficient

One thing that surprised me is how *efficient* backpropagation is. Naively, you might think "If I have many layers, won't computing all those gradients be slow?" But backprop uses the **chain rule** cleverly to save work. It reuses computations from the forward pass and propagates derivatives layer by layer, rather than re-deriving everything from scratch.

GeeksforGeeks highlights this: backprop *"computes the gradient of the loss function with respect to each weight using the chain rule making it possible to update weights efficiently"* [5]. And importantly, it *"scales well to*

networks with multiple layers and complex architectures" [6] . In other words, the math of chaining derivatives means that one pass backward gives you *all* the partial derivatives you need, without explosive cost. This is why deep learning is even feasible: backprop handles hundreds of layers and millions of weights in practice.

Analogy: It's like doing an assembly line in reverse. If you know your final product is wrong, you don't re-create everything from scratch to find the issue. You trace back step by step using the work you already did, rather than throwing everything out. In fact, the gradient "flows back" through the network almost like electricity or water, adjusting each connection proportionally.

I remember being amazed: what seemed like overwhelming calculus turns out to be one neat backwards pass. When I first studied it, the math was baffling, but thinking of it as just passing the error back through a chain (and multiplying by each weight and derivative along the way) helped a ton. It felt *satisfying* to realize this logical flow – it's not magic, just careful application of derivatives.

Backpropagation Steps (Summary)

To recap, here's a simpler step-by-step picture of backpropagation, with analogies:

1. **Forward Pass:** Give the network some inputs and compute outputs layer by layer (neurons sum weighted inputs + bias, apply activation) [7] . Like mixing ingredients in a recipe and tasting the final dish.
2. **Compute Loss (Error):** Compare the network's final output to the target. For example, use MSE (=(y_{\text{pred}}-y_{\text{true}})^2) [2] . This is the measure of how "off" we are. Like seeing how far a darts throw is from the bullseye.
3. **Backward Pass (Backpropagate Error):** Starting from the output layer, propagate the error back to each weight using the chain rule [3] . Compute each neuron's "blame" ((\delta) value) for the final error. It's similar to a teacher giving feedback: "Your final answer was wrong by X, so the earlier steps must be adjusted by these proportions."
4. **Update Weights:** Adjust each weight slightly in the direction that reduces error (usually subtract gradient × learning_rate). For instance, (w_{\text{new}} = w_{\text{old}} - \eta \, \delta \times (\text{input to that weight})). This is like tweaking each recipe ingredient a tiny bit to get closer to the desired taste next time.

We repeat these steps many times. Each iteration, the network's prediction gets a bit better. Soon the outputs will stay closer to targets for all training examples. It's iterative learning.

Personal Reflections

I have to admit: initially, I found backprop **mystical**. I was especially confused about "how can we compute derivatives through so many layers, and why does doing it in reverse make sense?" But thinking of it as everyday scenarios helped:

- **Teacher Analogy:** At first it clicked when I imagined a math student getting a test back. If the final grade is off, the teacher points out which steps were wrong. The student can then adjust those steps next time. Similarly, in backprop each weight "learns" from the output error and adjusts itself.

- **Recipe Analogy:** I thought of cooking. If my soup is too salty, I trace back: did I add too much salt or not enough water? I adjust those exactly like the network adjusts weights when the flavor (output) is off.
- **Puzzle Analogy:** It was also helpful to think backwards: say I'm lost on a trail (the error). Backprop is like retracing my steps, checking each fork backwards to see where the wrong turn was made. This mental image made the "backward" part less scary.

Each time I understood one small piece, I felt that *aha!* moment. For example, realizing the chain rule is just a way to see how a small change in an earlier weight changes the final outcome (and thus the error) was very satisfying. And seeing how beautifully this simple idea of "follow the error backward" powers all modern AI feels inspirational.

Conclusion: Learning Like Humans?

In the end, backpropagation turned out not to be a black box, but a mirror of how we ourselves might refine our knowledge. We make a guess (forward pass), check how far off we are (compute loss), analyze what went wrong step by step (backward pass), and then tweak our thinking (update weights) before trying again.

Backprop is iterative and self-correcting, much like learning a new skill or concept. If I miss an answer, I look at where I went wrong and try a new approach next time. Neural networks do the same math-wise. While the brain doesn't literally do backprop (neurons are different), the *philosophy* of learning from errors is very human.

When I finally understood backprop, I remember feeling both humbled and excited. It was humbling because something that once seemed arcane now made intuitive sense, and exciting because now I could appreciate the power behind neural networks: they *learn* by a process that's not too alien from our own learning curve.

If you've read this far and still feel lost at any point, know that's totally normal. Backpropagation took me a few days of fiddling and analogies to really click. But once it does, it's like unlocking how the brains of AI tick. Keep experimenting, draw out diagrams, tell yourself little stories (like teacher, recipe, or puzzle), and bit by bit it will become clear.

In summary: Backprop is just a structured way to adjust a network's "knobs" by the amount of the error it caused. It may sound complicated, but at its heart, it's just math *serving the story of learning*. And remember – **everyone** finds this tricky at first. The fact that neural nets can learn from mistakes, step by step, is frankly amazing. It's something any learner can find inspiring once the clouds of confusion pass.

Key Takeaways:
- Neural networks learn by trial-and-error: a forward pass makes a prediction, we measure error, and then we backtrack to fix it.
- Each neuron's output is "input × weights + bias," passed through an activation (introducing nonlinearity).
- Backpropagation efficiently computes gradients of the error w.r.t. every weight (using the chain rule) [8] [5], allowing us to update all weights correctly.
- We update weights a little at a time (learning rate) so the network gradually improves.
- This whole algorithm *mirrors* human learning in spirit: make a guess, get feedback, adjust, and repeat.

Hope this journey through backprop was helpful and that you feel a bit more confident next time you hear someone mention gradient descent or chain rule in neural nets. Keep experimenting and learning – your future self will thank you for persisting!

---

1 2 3 4 5 6 7 8 **Backpropagation in Neural Network | GeeksforGeeks**
https://www.geeksforgeeks.org/backpropagation-in-neural-network/