Article Link: Learn to Build a Neural Network From Scratch — Yes, Really (by Aadil). Other Resources include “I Built a Neural Network from Scratch,” “Neural Networks Explained from Scratch using Python,” “Building a Neural Network FROM SCRATCH (no TensorFlow/PyTorch, just NumPy & math),” “How to Create a Neural Network (and Train it to Identify Doodles),” the QuickDraw dataset (https://github.com/googlecreativelab/quickdraw-dataset), and the Manim repository (https://github.com/3b1b/manim).

Cost measures how wrong our machine’s predictions are. A scalar is a single number; a vector is an array of numbers; a matrix is a two-dimensional table of numbers, described by its rows and columns. Taking the dot product between two vectors returns a single number. Transposing a matrix simply swaps its rows and columns, which allows us to multiply matrices together without changing the underlying data.

In calculus, the power rule states that the derivative of x^n is $n \cdot x^{n-1}$ for any real number n. A derivative measures how one quantity changes with respect to another; the derivative of a constant is zero. The product rule lets us differentiate a product $h(x) = f(x) \cdot g(x)$ by computing $h'(x) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$. The sum rule says that the derivative of a sum is the sum of the derivatives. The chain rule allows us to differentiate a composite function $f(g(x))$ by computing $f'(g(x)) \cdot g'(x)$. When multiple variables are involved, we compute partial derivatives—treating all other variables as constants—to see which inputs most affect the cost function. We denote a partial derivative of f with respect to x as $\partial f / \partial x$ (“del f del x”).

Gradient descent uses the gradient of the cost function, denoted ∇C, to update model parameters (weights and biases) in the direction that most decreases cost. By combining partial derivatives and the chain rule, we can figure out exactly how each weight and bias should change to reduce error.

A neural network consists of layers of nodes (neurons) connected by weights and biases. Each node in one layer connects to every node in the next layer via a weight. Only the input layer holds real data values; all other layers compute their node values by summing the products of previous-layer activations and their connecting weights, then adding a bias. If layer l has i nodes and layer l–1 has j nodes, it has i·j weights. Each non-input node also has its own bias. We adjust these weights and biases—the “dials” of the network—to minimize the cost function via gradient descent.

To make layer-to-layer computations efficient, we represent each layer’s activations as a vector (or matrix, once vectorized over many training examples) and the weights between layers as a matrix. For any layer l, the weight matrix W^[l] has dimensions n^[l] × n^[l–1], where n^[l] is the number of nodes in layer l. The bias for layer l is a matrix b^[l] of dimensions n^[l] × 1. We perform feed-forward by computing z = W^[l]·a^[l–1] + b^[l] and then passing z through an activation function g(z), which must be nonlinear (so it cannot simply be mz + b) and typically compresses its input into a fixed range, such as [0,1]. A common choice is the sigmoid function g(z) = 1/(1 + e^(–z)).

Vectorization lets us process m training samples at once by arranging input data X as an n × m matrix, where n is the number of features and m is the number of examples. We then compute each layer's activations A^[l] as matrices with n^[l] rows and m columns. Numpy supports efficient matrix operations: use the @ operator for matrix multiplication, .shape() to inspect dimensions, and .reshape() to reformat arrays. Remember that numpy arrays are contiguous C arrays, not Python lists, and you can use np.exp() for exponentials. Use assert statements to check your assumptions when debugging.

Backpropagation is the algorithm that trains the network by computing the gradients $\partial C/\partial W^{[l]}$ and $\partial C/\partial b^{[l]}$ via the chain rule, then updating each parameter in the opposite direction of its partial derivative. We repeat the cycle of feed-forward, cost evaluation, backpropagation, and parameter update (using a learning rate α, often 0.01) many times—sometimes thousands—to converge toward a minimum of the cost function.

Common cost functions include mean squared error (MSE), root mean squared error (RMSE), and mean absolute error (MAE). For binary outcomes, binary cross-entropy is often preferred. Convex cost functions (shaped like bowls) have a single global minimum, while non-convex functions can have local minima. Gradient descent follows the negative gradient to find a minimum; the learning rate α controls how large each update step is.

In practice, building a neural network from scratch involves these steps: initialize weights and biases randomly; repeat for each training sample or batch: perform feed-forward to compute predictions $\hat{y}$, compute the cost $C(\hat{y},y)$, save C to track convergence, perform backpropagation to compute ∇C, update weights and biases; monitor the cost over many iterations until it stabilizes.

How a neural network works in a nutshell: feed forward to calculate the network output $\hat{y}$, calculate the cost C from $\hat{y}$ and the true labels y, run backpropagation to compute ∇C for each layer's weights and biases, update parameters using learning rate α, and repeat until the cost minimally decreases. To get started, gather lots of high-quality data, choose your model architecture, and train the model.