The Mathematics Behind Deep Learning

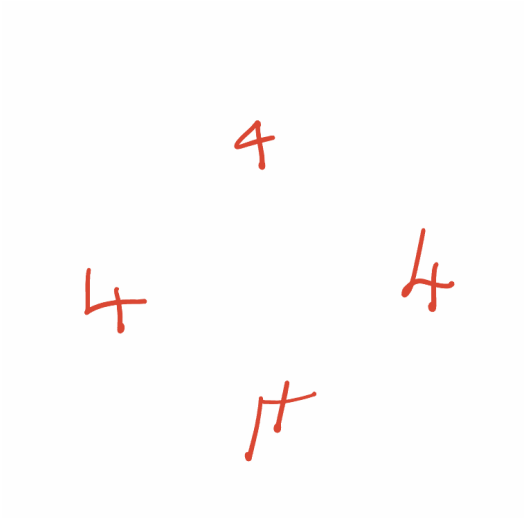
What is a neural network, and how does it learn?

Miraj Samarakkody, Ph.D.

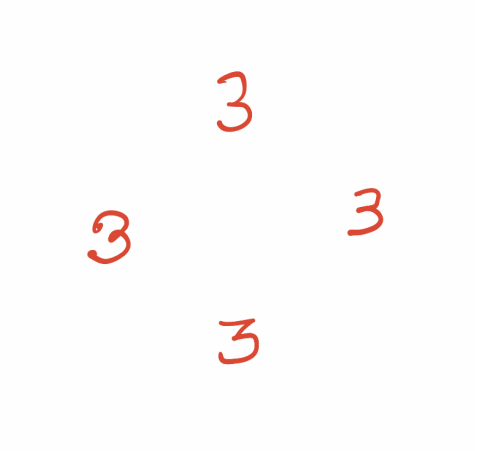
Department of Mathematics & Computer Science
Tougaloo College

October 28, 2026

What do you see?



A problem that is easy for you

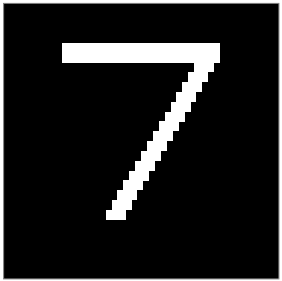


- You instantly read all of these as a **3**.
- But the actual pixels are completely different each time.

Goal of this talk

See that a neural network is just a function built from simple math: multiplying, adding, and "squishing."

What is a computer see?

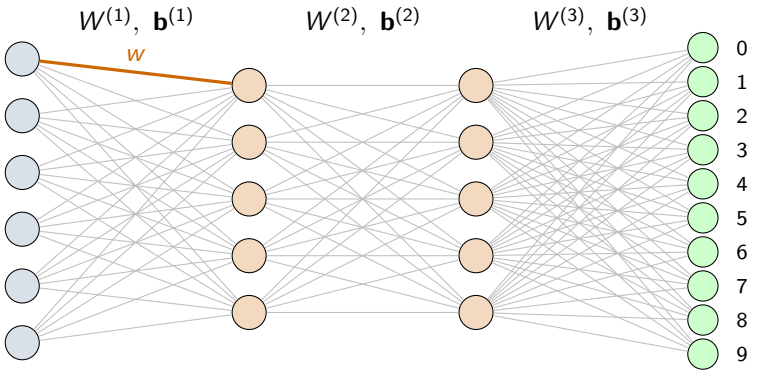


$28 \times 28 = 784$ pixels

- One number per pixel: 0 = black, 1 = white.
- So the image becomes a list of 784 numbers:

$$(a_1, a_2, \ldots, a_{784}).$$

The network: layers of neurons



input $\mathbf{a}^{(0)}$
784 pixels

hidden $\mathbf{a}^{(1)}$
16 neurons

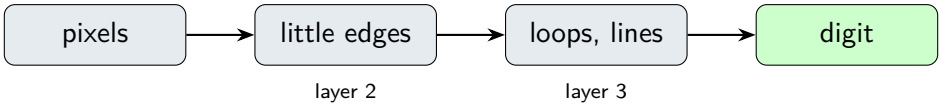
hidden $\mathbf{a}^{(2)}$
16 neurons

output $\mathbf{a}^{(3)}$
10 digits

$\mathbf{a}$ = activations, w = weights (one per line), b = biases (one per neuron).

Why layers? The hope

- A digit is built from pieces:
  - 9 = a **loop** on top + a **line** on the right
  - 8 = a **loop** on top + a **loop** on the bottom
- Each loop or line is built from smaller **edges**.



Breaking a hard problem into layers of simpler ones is the idea behind "deep" learning.

Step 1: weights and a weighted sum

How could one neuron detect an edge in a certain spot?

- Give every connection from the previous layer a weight $w_1, w_2, \ldots, w_{784}$ (just numbers).
- Compute the weighted sum:

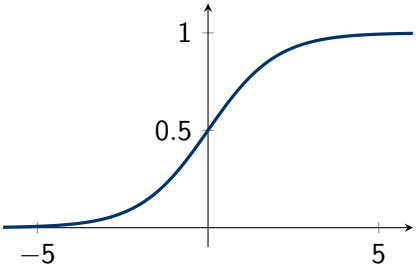
$$w_1 a_1 + w_2 a_2 + w_3 a_3 + \cdots + w_{784} a_{784}.$$

- Positive weights on the pixels in the edge region, negative weights on the pixels around it, zero everywhere else.
- Then the sum is **large** exactly when that region is bright and its surroundings are dark, i.e., when there is an edge there.

Math idea

A weighted sum is a dot product $\mathbf{w} \cdot \mathbf{a}$ from linear algebra. It measures how well the input matches the pattern $\mathbf{w}$.

Step 2: squish it into $[0, 1]$ with the sigmoid



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The weighted sum can be any number.
- We want the neuron's activation a (a number) between 0 and 1.
- The sigmoid σ does this. It is called the activation function (a rule, not a number):
  - very negative $\to$ close to 0
  - very positive $\to$ close to 1
  - around 0 $\to$ in between

Step 3: the bias

Maybe we only want the neuron to light up when the weighted sum is *bigger than 10*. Subtract 10 before squishing:

$$\underbrace{a}_{\substack{\text{activation}\\ \text{(a number)}}} = \underbrace{\sigma}_{\substack{\text{activation}\\ \text{function}}} \left(w_1 a_1 + w_2 a_2 + \cdots + w_{784} a_{784} \underbrace{- 10}_{\text{bias}} \right).$$

Weights

tell the neuron what pattern to look for.

Bias

tells the neuron how strong the match must be before it becomes active.

Every neuron in the hidden and output layers has its own weights and its own bias.

How many numbers is that?

	weights	**biases**
input $\to$ hidden 1	$784 \times 16 = 12{,}544$	16
hidden 1 $\to$ hidden 2	$16 \times 16 = 256$	16
hidden 2 $\to$ output	$16 \times 10 = 160$	10
total	12,960	42

$$12{,}960 + 42 = 13{,}002 \text{ knobs and dials}$$

Learning means having the computer find good values for all 13,002 numbers automatically, from examples.

The compact version: matrices

Put all the activations of a layer in a column vector, and all the weights in a matrix:

$$\mathbf{a}^{(1)} = \sigma\left(\underbrace{\begin{pmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ \vdots & & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{pmatrix}}_{W} \underbrace{\begin{pmatrix} a_0^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix}}_{\mathbf{a}^{(0)}} + \underbrace{\begin{pmatrix} b_0 \\ \vdots \\ b_k \end{pmatrix}}_{\mathbf{b}}\right)$$

$$\boxed{\mathbf{a}^{(1)} = \sigma\big(W\mathbf{a}^{(0)} + \mathbf{b}\big)}$$

- Row j of W holds the weights going into neuron j.
- σ is applied to each entry separately. Computers are very fast at this.

A tiny example you can do by hand

Two inputs, one neuron: $a_1 = 1,\ a_2 = 0.5, \quad w_1 = 2,\ w_2 = -1, \quad b = -1.$

1. Weighted sum: $2(1) + (-1)(0.5) = 1.5$
2. Add bias: $1.5 + (-1) = 0.5$
3. Squish: $\sigma(0.5) = \dfrac{1}{1 + e^{-0.5}} \approx 0.62$

The neuron's activation is about $\mathbf{0.62}$: moderately "on."

Try it

What happens if the bias is $b = -5$ instead? (Answer: $\sigma(-3.5) \approx 0.03$, nearly "off.")

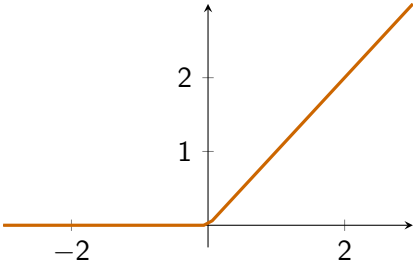
The whole network is just a function

$$\underbrace{\mathbf{a}^{(0)}}_{\text{784 numbers in}} \longmapsto \underbrace{\mathbf{a}^{(3)}}_{\text{10 numbers out}}$$

$$\mathbf{a}^{(3)} = \sigma\Big(W_3\,\sigma\big(W_2\,\sigma\big(W_1\mathbf{a}^{(0)} + \mathbf{b}_1\big) + \mathbf{b}_2\big) + \mathbf{b}_3\Big)$$

- A very complicated function with 13,002 parameters...
- ...but it is built only from multiply, add, and squish.
- It is reassuring that it is complicated: recognizing digits is a hard task!

A modern note: ReLU



$$\mathrm{ReLU}(x) = \max(0, x)$$

- The sigmoid was inspired by biological neurons being "on" or "off."
- Today most networks use the ReLU (rectified linear unit) as the activation function instead.
- It is simpler, and networks using it are much easier to train.

How does the network learn?

Cost functions and gradient descent

Learning from examples

- We have training data: tens of thousands of images of handwritten digits, each with a label saying which digit it is (the MNIST database).
- Start with completely random weights and biases.
- Of course the network does terribly at first: its output is garbage.
- Plan:
  1. Measure *how bad* the network is with a single number.
  2. Nudge the 13,002 weights and biases to make that number smaller.
  3. Repeat many, many times.
- Then check it on new images it has never seen. Doing well there is what "learning" really means.

The cost of one example

Show the untrained network an image of a **3**:

digit	0	1	2	3	4	5	6	7	8	9
network output $a_j^{(3)}$	0.43	0.28	0.19	0.88	0.72	0.01	0.64	0.86	0.99	0.63
what we want y_j	0	0	0	1	0	0	0	0	0	0

Add up the squares of the differences:

$$C = (0.43-0)^2 + (0.28-0)^2 + \cdots + (0.88-1)^2 + \cdots + (0.63-0)^2 = \sum_{j=0}^{9} \left(a_j^{(3)} - y_j\right)^2 \approx 3.36$$

- C is **small** when the network confidently gets it right.
- C is **large** when the network has no idea.

The cost function

Average the cost over *all* the training examples:

$$C(w_1, w_2, \ldots, w_{13,002}) = \frac{1}{N} \sum_{\text{examples}} \sum_{j=0}^{9} \left(a_j^{(3)} - y_j\right)^2.$$

The network is a function

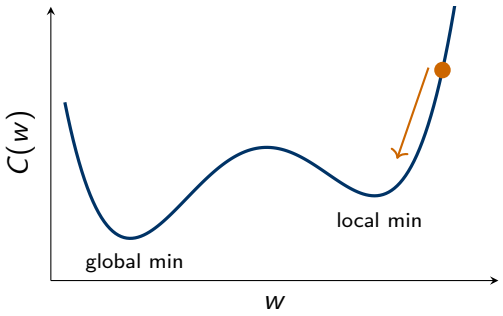
Input: 784 pixels
Output: 10 numbers
Parameters: 13,002 weights and biases

The cost is a function

Input: 13,002 weights and biases
Output: 1 number (how bad)
Parameters: the training data

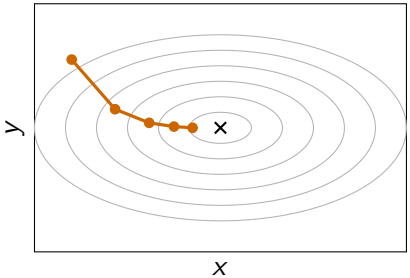
Learning = finding the minimum of the cost function.

Minimizing a function of one variable



- With 13,002 inputs we cannot just solve $C' = 0$.
- Instead, start somewhere and look at the slope $C'(w)$:
  - slope positive $\Rightarrow$ step **left**
  - slope negative $\Rightarrow$ step **right**
- Like a ball rolling downhill.
- Take smaller steps as the slope flattens, so we do not overshoot.
- We may land in a local minimum, not the best one.

Two variables: the gradient



Contour lines of $C(x, y)$; each orange step moves along $-\nabla C$.

- The gradient collects the slopes in every direction:

$$\nabla C = \left(\frac{\partial C}{\partial x}, \frac{\partial C}{\partial y} \right)$$

- From multivariable calculus: ∇C points in the direction of **steepest increase**.
- So $-\nabla C$ points **downhill** the fastest way.
- Its length tells us *how steep* the hill is.

Gradient descent

Stack all 13,002 weights and biases into one long vector $\vec{W}$. Repeat:

$$\boxed{\vec{W}_{\text{new}} = \vec{W}_{\text{old}} - \eta \nabla C(\vec{W}_{\text{old}})}$$

η ("eta") is a small number called the learning rate.

$$\vec{W} = \begin{pmatrix} 2.25 \\ -1.57 \\ 1.98 \\ \vdots \\ 3.82 \end{pmatrix} \qquad -\nabla C(\vec{W}) = \begin{pmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ -0.32 \end{pmatrix}$$

$\leftarrow$ increase w_1 a little
$\leftarrow$ increase w_2 a lot
$\leftarrow$ decrease w_3 a lot

Each entry says which way to nudge one weight, and how much that weight matters for the cost.

A tiny gradient descent by hand

Minimize $C(w) = (w-3)^2$. Slope: $C'(w) = 2(w-3)$. Start at $w_0 = 0$, learning rate $\eta = 0.25$.

step	w	slope $C'(w)$	new $w = w - 0.25\,C'(w)$
0	0	-6	$0 + 1.5 = 1.5$
1	1.5	-3	$1.5 + 0.75 = 2.25$
2	2.25	-1.5	$2.25 + 0.375 = 2.625$
3	2.625	-0.75	2.8125

- w creeps toward the minimum at $w = 3$, and the steps shrink as the slope flattens.
- The network does exactly this, but in 13,002 dimensions at once.

Try it

What happens with $\eta = 1$? (Answer: w jumps $0 \to 6 \to 0 \to 6 \ldots$ and never settles.)

Two important details

Why the activation function must be smooth

Gradient descent needs slopes. Real neurons are simply "on" or "off," but a jump has no useful slope. That is why artificial neurons use smooth activation functions like $\sigma(x)$: the cost changes *continuously* as we nudge the weights.

Backpropagation

The algorithm for computing ∇C efficiently for all 13,002 weights at once is called backpropagation. It is the chain rule from calculus, applied layer by layer from the output back to the input.

How well does it work?

- After training, our small network correctly classifies about 96% of new images it has never seen; with some tweaks, about 98%.
- But did the hidden layers learn "edges" and "loops"? **Not really.** Their weight patterns look mostly random.
- Feed it random noise and it confidently answers some digit: it has never been taught to say "I don't know."

Lesson

The network found *a* good local minimum of the cost function, not necessarily the one we imagined. Understanding what networks actually learn is an open area of research, and a lot of it is mathematics.

Summary: the math you already know

Idea in the network	**Math behind it**
neuron's activation a	a number between 0 and 1
weighted sum	dot product
activation function (squishing)	a function like $\sigma(x)$ or $\max(0, x)$
a whole layer	matrix times vector, plus vector
the network	composition of functions
how bad it is	cost function (sum of squares)
learning	minimizing the cost: gradient descent
which way to nudge	the gradient $-\nabla C$ (calculus)

Further reading and watching

- 3Blue1Brown, *But what is a neural network?* (Deep learning, Chapter 1), `https://www.youtube.com/watch?v=aircAruvnKk`
- 3Blue1Brown, *Gradient descent, how neural networks learn* (Chapter 2), `https://www.youtube.com/watch?v=IHZwWFHWa-w`
- Michael Nielsen, *Neural Networks and Deep Learning* (free online book), `http://neuralnetworksanddeeplearning.com`
- The MNIST database of handwritten digits.

Thank you! Questions?