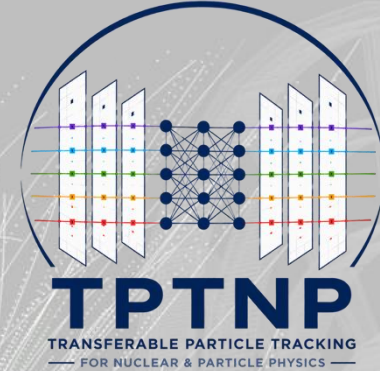




sPHENIX Preliminary
2025-6-14, Run 66641
BCO: 722670579120701
Collisions: Au Au $\sqrt{s_{NN}} = 200$ GeV



Stony Brook
University



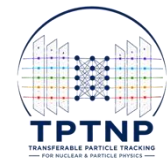
How Neural Networks Actually Work

Curve fitting, CNN, Transformers, Mamba

Mariia Mitrankova
Stony Brook University

October 5th, 2026

Plan for the hour



- 1 What is a neural network?**
fitting curves through points — with many knobs
- 2 Why don't huge networks just memorize?**
billions of knobs, and still no overfitting
- 3 Images: convolutional networks (CNN)**
look at small patches, then patches of patches
- 4 Text: attention and Transformers**
the engine behind ChatGPT
- 5 Mamba: reading with a memory**
fast models for very long texts

Based on four videos by *Algorithmic Simplicity* (YouTube) — links at the end



sPHENIX Preliminary
2025-6-14, Run 66641
BCO: 722670579120701
Collisions: Au+Au @ $\sqrt{s_{NN}} = 200$ GeV

Part I: What is a neural network, *really*?

It all starts with drawing a line through points

It all starts with fitting a line

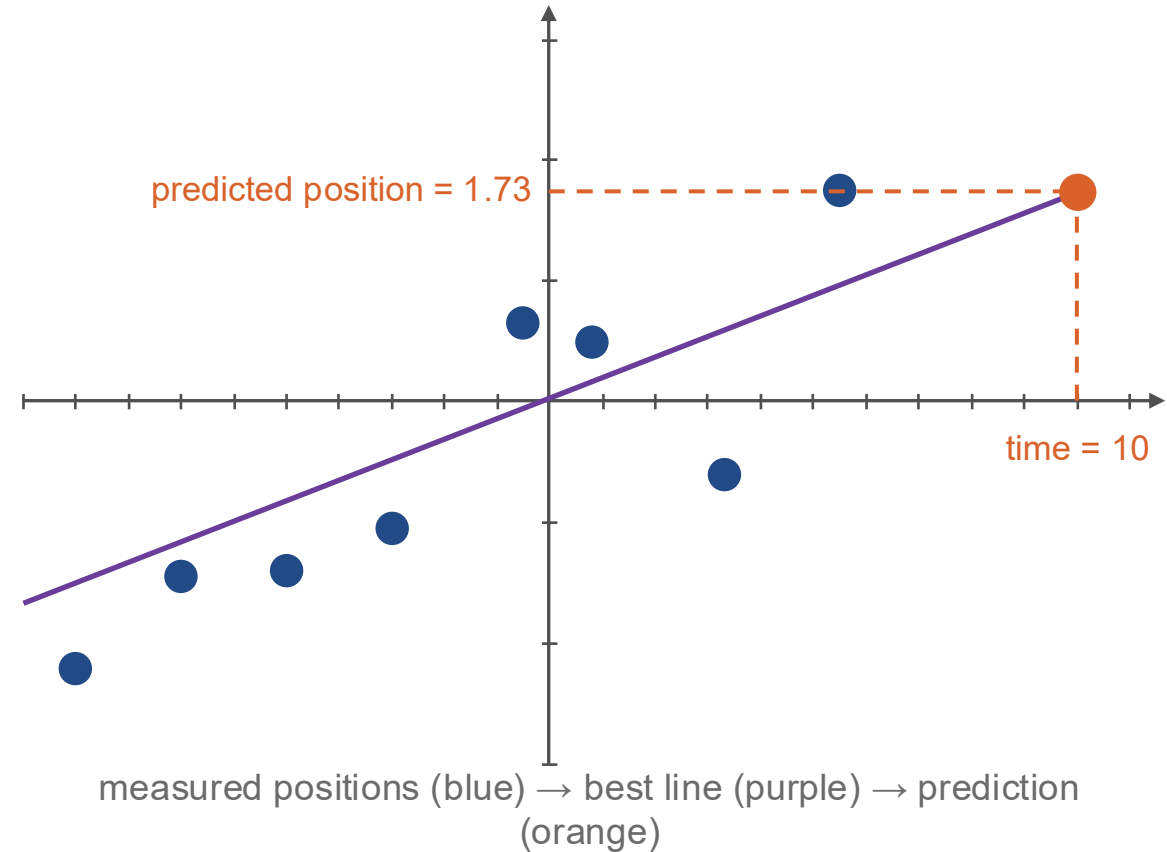


C. F. Gauss (1777–1855)



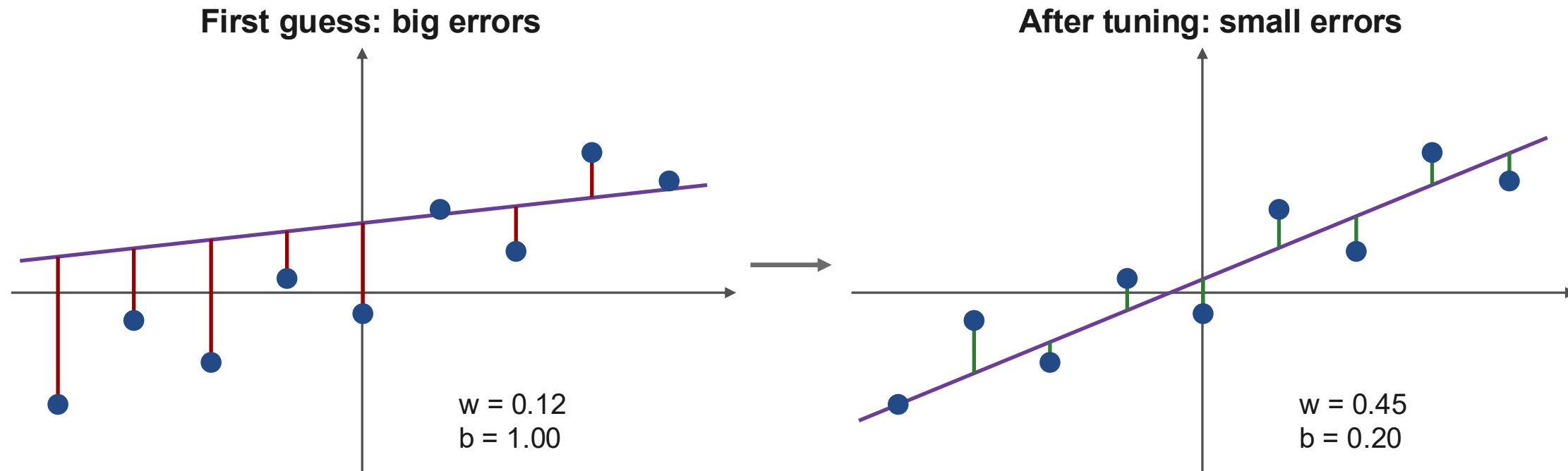
Least squares (Gauss & Legendre, ~1805):

find the line that passes as close as possible to all the points.



In one sentence: measure some points, draw the best line through them, use the line to predict new points.

How do we find the best line? Turn two knobs



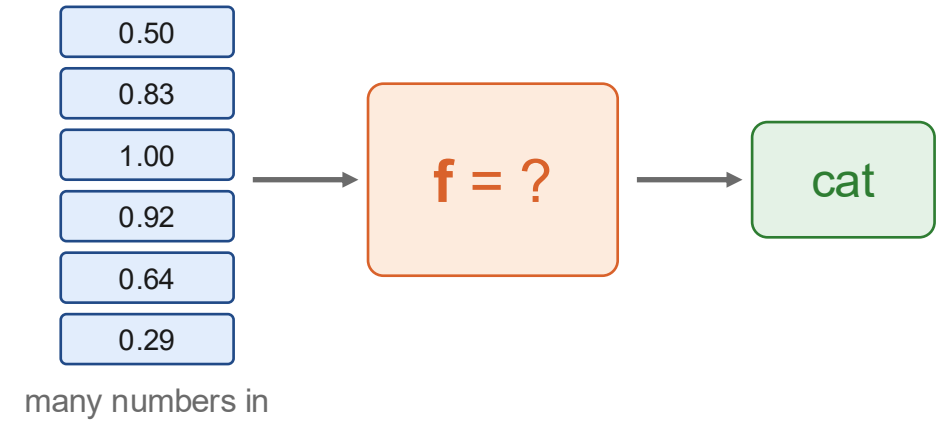
- ▶ The line is $y = w \cdot x + b$: two knobs, the slope w and the shift b .
- ▶ Error = add up the squared lengths of the red/green bars. Turn the knobs until the error is as small as possible.

In one sentence: “training” just means turning the knobs until the total error is as small as possible.

Machine learning = fitting a function

We want a function f : input $\rightarrow$ output, that nobody can write down by hand:

Input	Output	Numbers in
a photo	“cat” or “dog”	thousands – millions
hits in the TPC	track momentum, direction	~ 10,000
the text so far	the next word	thousands



- ▶ What we **do** have: lots of examples — inputs together with the right answers.
- ▶ **Goal**: find an f that agrees with the examples **and** gives good answers for inputs it has never seen.

In one sentence: the same game as the straight line — but the input is thousands of numbers, not one.

A straight line is not enough

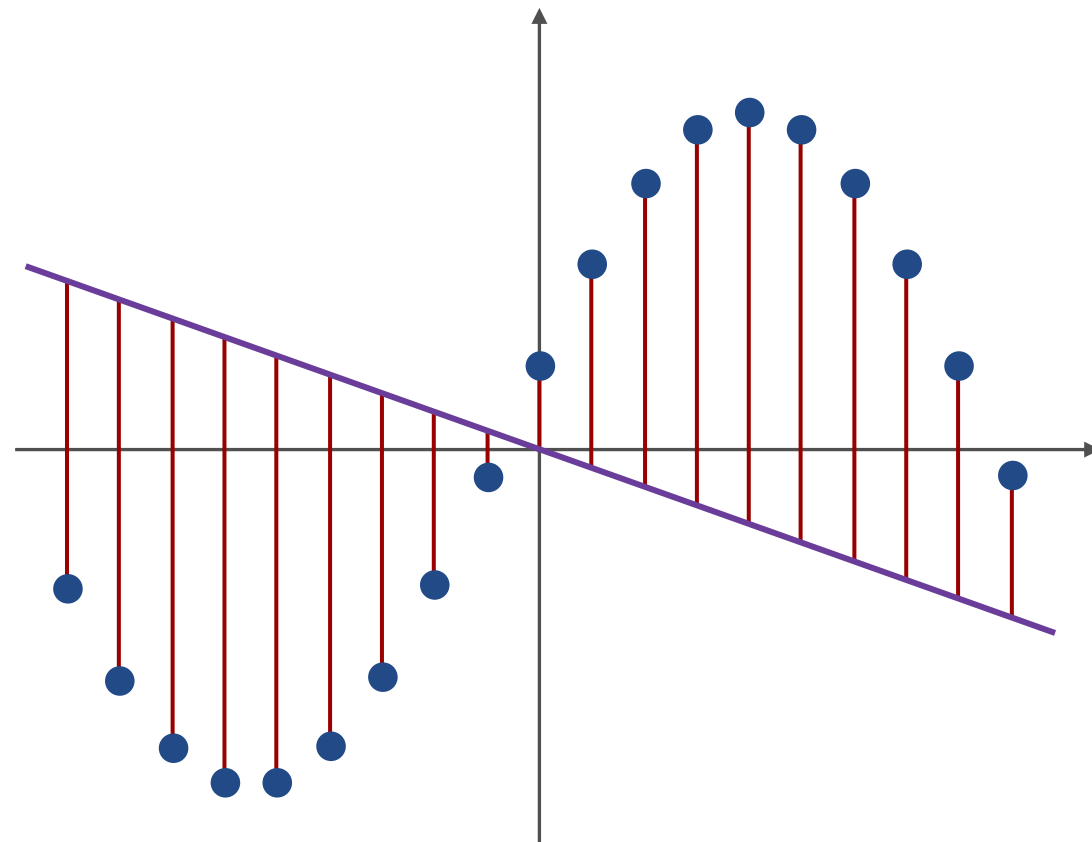
- ▶ Real data is curved: a wave, a circle, a bent track ...
- ▶ Even the **best** straight line misses most points.

Can we stack several lines?

A line of a line is still a line:

$$2 \cdot (3x + 1) + 5 = 6x + 7$$

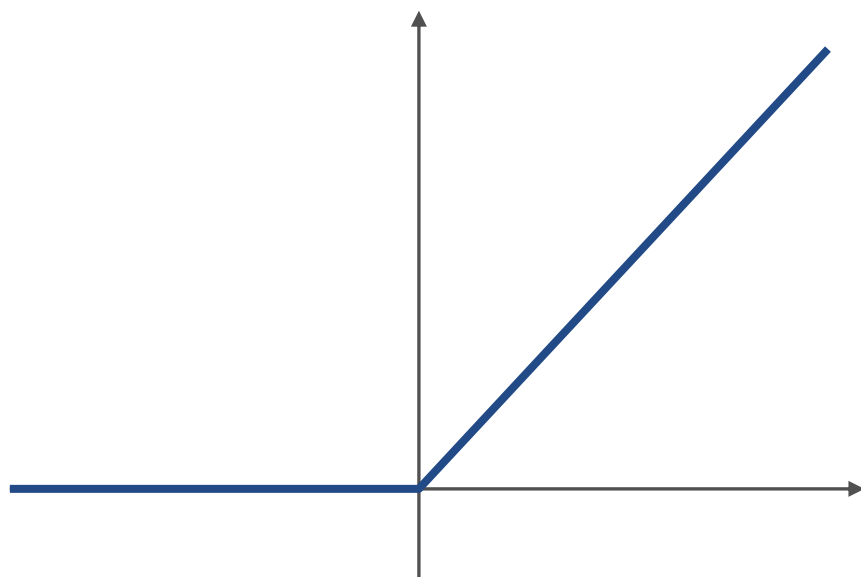
Stacking straight lines gains nothing.



best possible straight line through a wave: still terrible

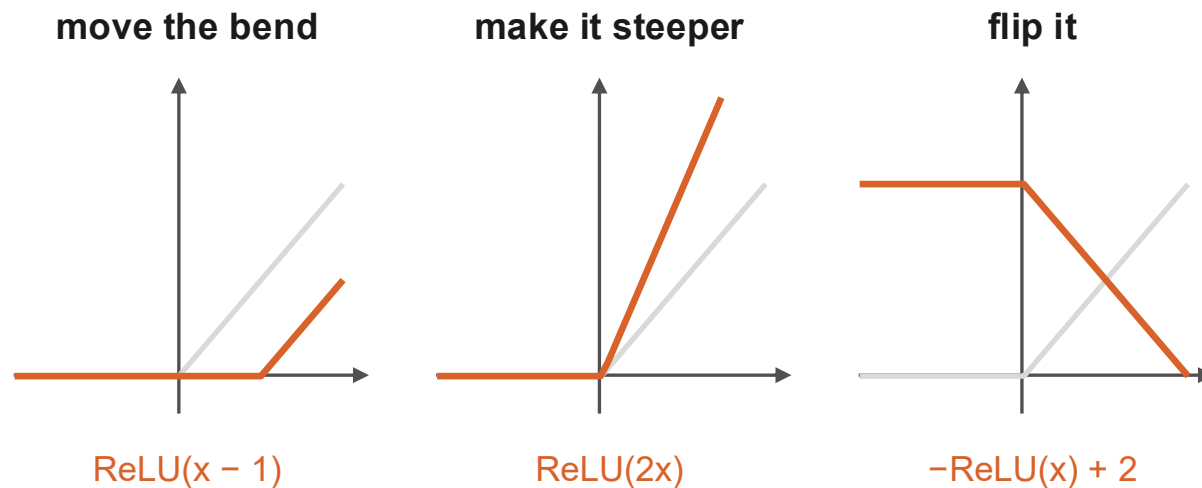
In one sentence: to draw curves we need something that can bend.

A neuron = a line with one bend



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

negative $\rightarrow 0$, positive $\rightarrow$ unchanged



One neuron: $y = w_2 \cdot \text{ReLU}(w_1 \cdot x + b_1) + b_2$

Four knobs decide **where** the bend is, how **steep** it is and which **way** it points.

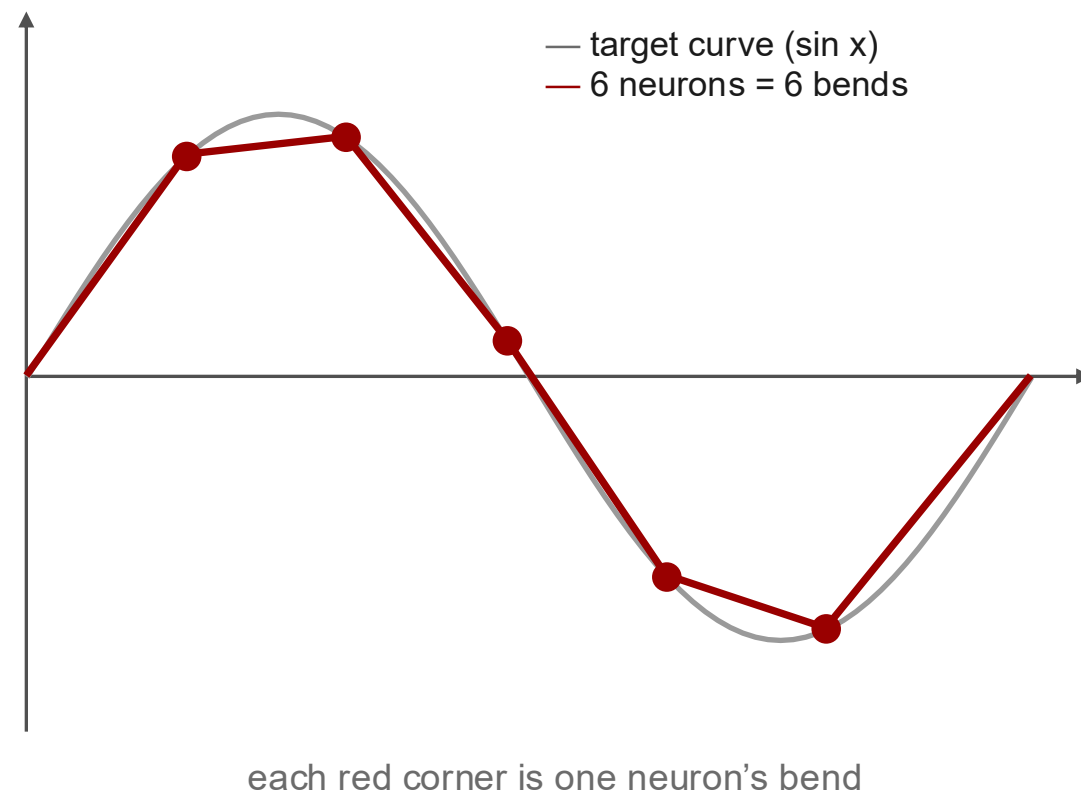
In one sentence: a neuron is a hinge: flat, then a bend, then straight — and its knobs move the bend around.

Many neurons = many bends = any curve

- ▶ Add up many neurons: every neuron adds one bend.
- ▶ More neurons $\rightarrow$ more corners $\rightarrow$ the curve can follow the data more closely.
- ▶ With enough neurons you can draw **any** curve (**universal approximation**).

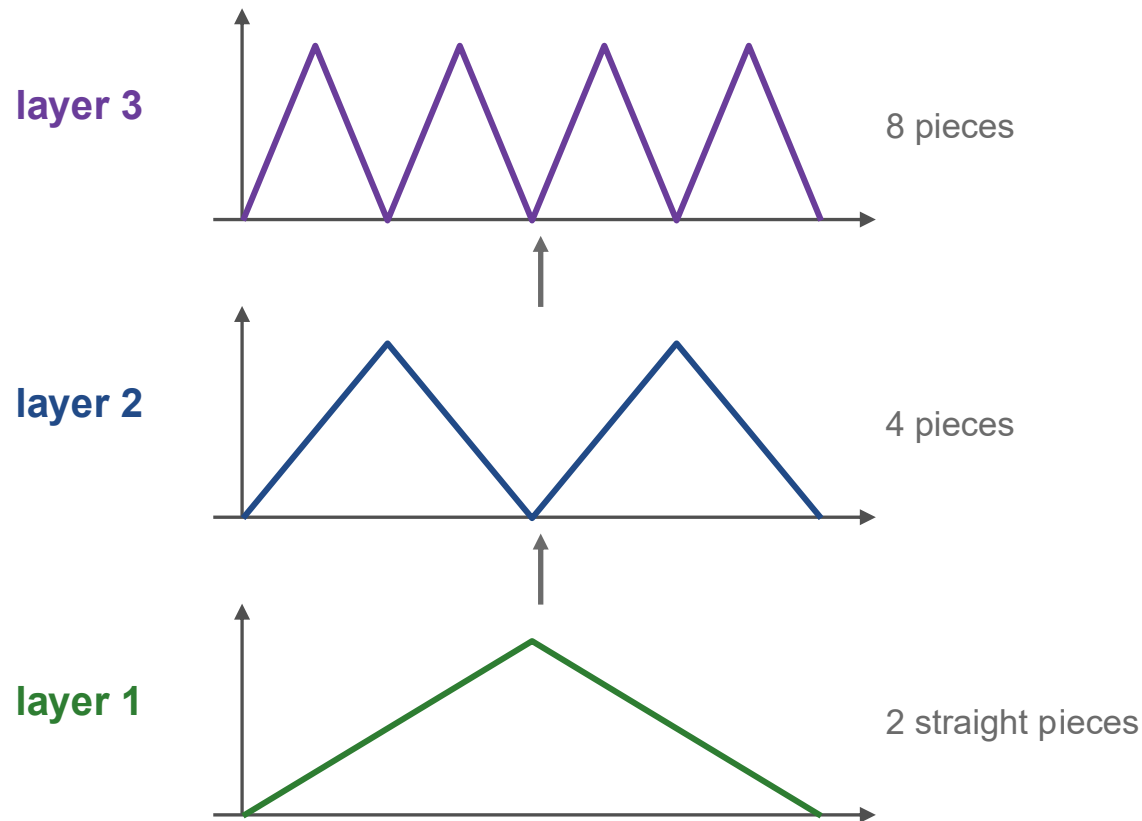
Why this matters

Any set of observations — the next move in chess, the next hit on a track in the TPC — is a bunch of points. With enough neurons, a network can draw a curve through them.



In one sentence: a network with one layer of neurons draws curves made of straight pieces.

Stack layers: bends of bends



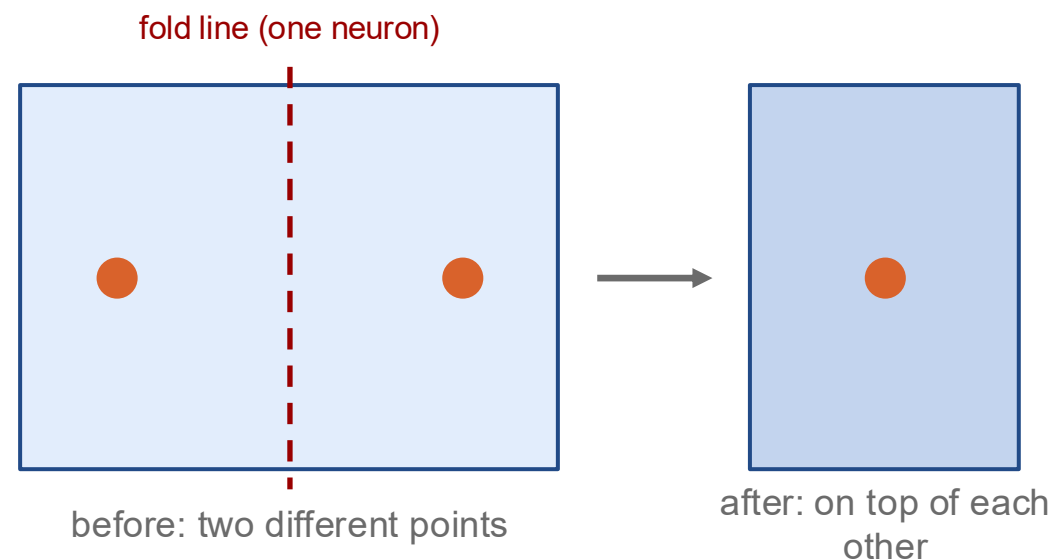
- ▶ Layer 2 does not see the input — it sees the **curves made by layer 1**.
- ▶ It shifts and flips them, adds them up and cuts everything below zero.
- ▶ Every cut can **double** the number of straight pieces.
- ▶ Example above: 2 neurons per layer → 2, 4, 8 pieces.

In general: **L layers of n neurons → up to n^L pieces.**

In one sentence: going deeper multiplies the corners — far more than making one layer wider.

Why depth works: every layer folds the space

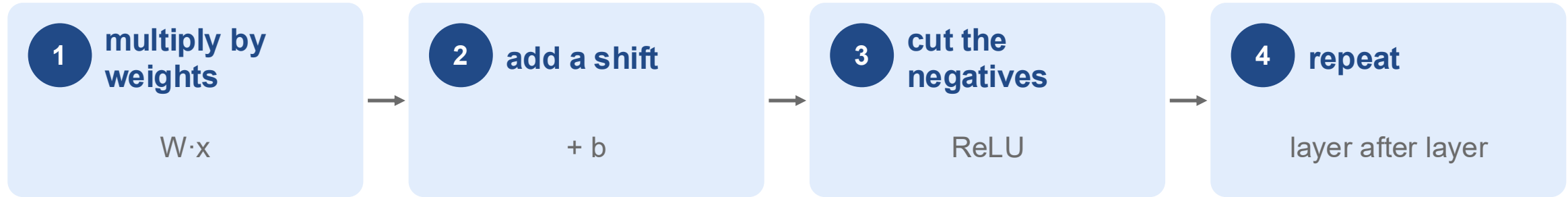
- ▶ Think of a sheet of paper. Each ReLU **folds** it along a line.
- ▶ After the fold, points that were far apart lie **on top of each other**.
- ▶ The next layer cuts the folded paper: one cut = a cut in **every** layer of the stack.
- ▶ So a pattern learned once is reused everywhere it repeats (an edge, a texture, a word).



Montúfar et al. (2014) count these pieces precisely: they grow exponentially with depth.

In one sentence: a deep network is folded paper: few cuts, but each cut goes through many layers at once.

So: what *is* a neural network?



$$f(x) = W_3 \cdot \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 \cdot x + b_1) + b_2) + b_3$$

- ▶ All the W 's and b 's are the knobs: millions to billions of them.
- ▶ The result is a curve made of an enormous number of straight pieces — built by folding, so the pieces share their knobs.

In one sentence: a neural network is a curve-drawing machine with a huge number of knobs.

Training, step 1: measure how wrong we are (the “loss”)

Predicting a number

loss = average of (prediction – truth)²

Example: predicted 2.5, truth 3

$$\rightarrow (2.5 - 3)^2 = 0.25$$

Exactly the squared bars from the straight-line fit.

Predicting a class (cat / dog)

loss = $-\log(\text{probability given to the right answer})$

Right answer “cat”:

P(cat) = 0.9 $\rightarrow$ loss = 0.1 (good)

P(cat) = 0.1 $\rightarrow$ loss = 2.3 (bad)

- Physicists: this is the $-\ln L$ you minimize in every likelihood fit.
- Training = make this one number as small as possible — a fit with a billion parameters.

In one sentence: the loss is one number that says how wrong the network is on the examples.

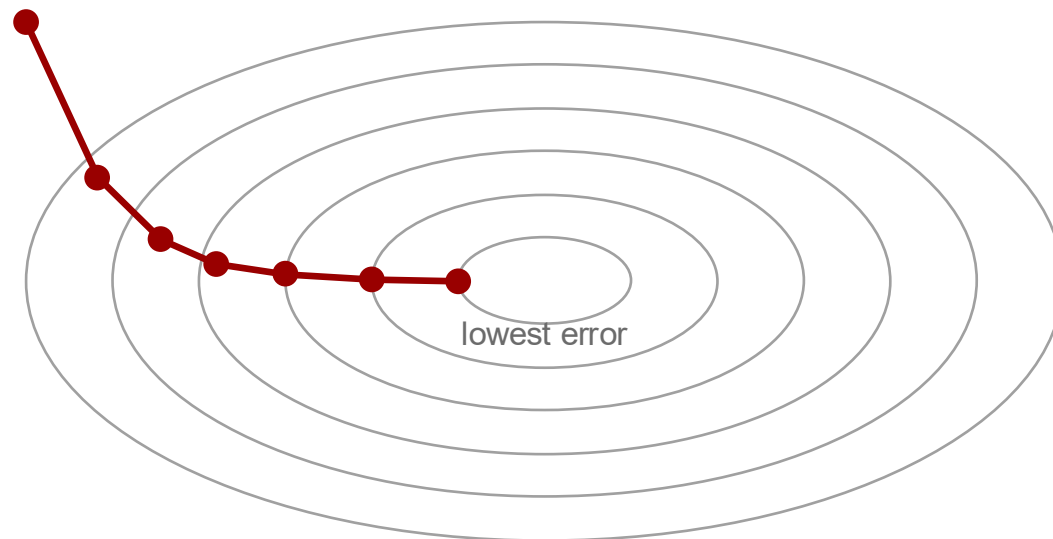
Training, step 2: walk downhill (gradient descent)

- ▶ You stand on a hill in thick fog and want to reach the valley.
- ▶ Feel the slope under your feet, take a small step **downhill**, repeat.

new knobs = old knobs – step size × slope

- ▶ The “slope” (gradient) tells, for every knob, which way makes the error smaller.
- ▶ Billions of knobs → no MINUIT, no Hessians: just follow the slope, many thousands of times.

start (random knobs)



map of the error for two knobs: contour lines like a hiking map

In one sentence: training = feel the slope, step downhill, repeat — for every knob at once.



Part II: Why don't huge networks just memorize?

billions of knobs, and still no overfitting

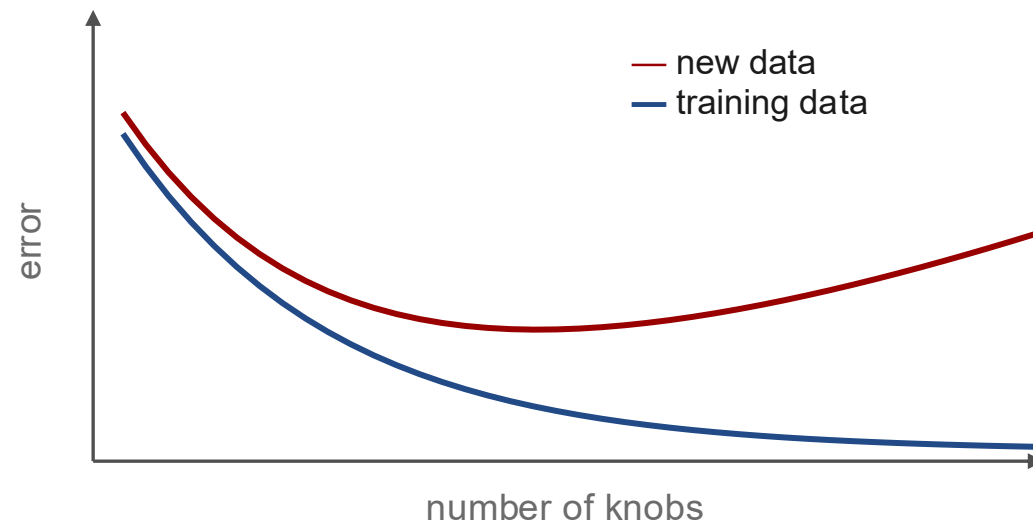
The paradox: too many knobs should be a disaster

Textbook statistics:

- ▶ more knobs than data points $\Rightarrow$ the model can fit the noise $\Rightarrow$ **overfitting**.
- ▶ Physicists: keep $\chi^2/\text{ndf} \approx 1$.

Deep learning:

- ▶ far more knobs than data points, routinely.
- ▶ and bigger models work **better**.



Input

928373 + 360123

412412 + 227492

362842 + 840331

```
if input == 928373 + 360123: output = 1288496
if input == 412412 + 227492: output = 639904
if input == 362842 + 840331: output = 1203173
... (one line per example)
```

Output

1288496

639904

1203173

the fear: a giant lookup table that never learned to add

In one sentence: a network has enough knobs to memorize everything — yet in practice it learns the rule instead.

Experiment: networks can memorize pure nonsense

Zhang et al. (2017): take 50,000 labelled photos (CIFAR-10) and train the same network twice:

Real labels

training photos: **100 %** correct
new photos: **~95 % correct**
it learned what a cat looks like

Labels shuffled at random

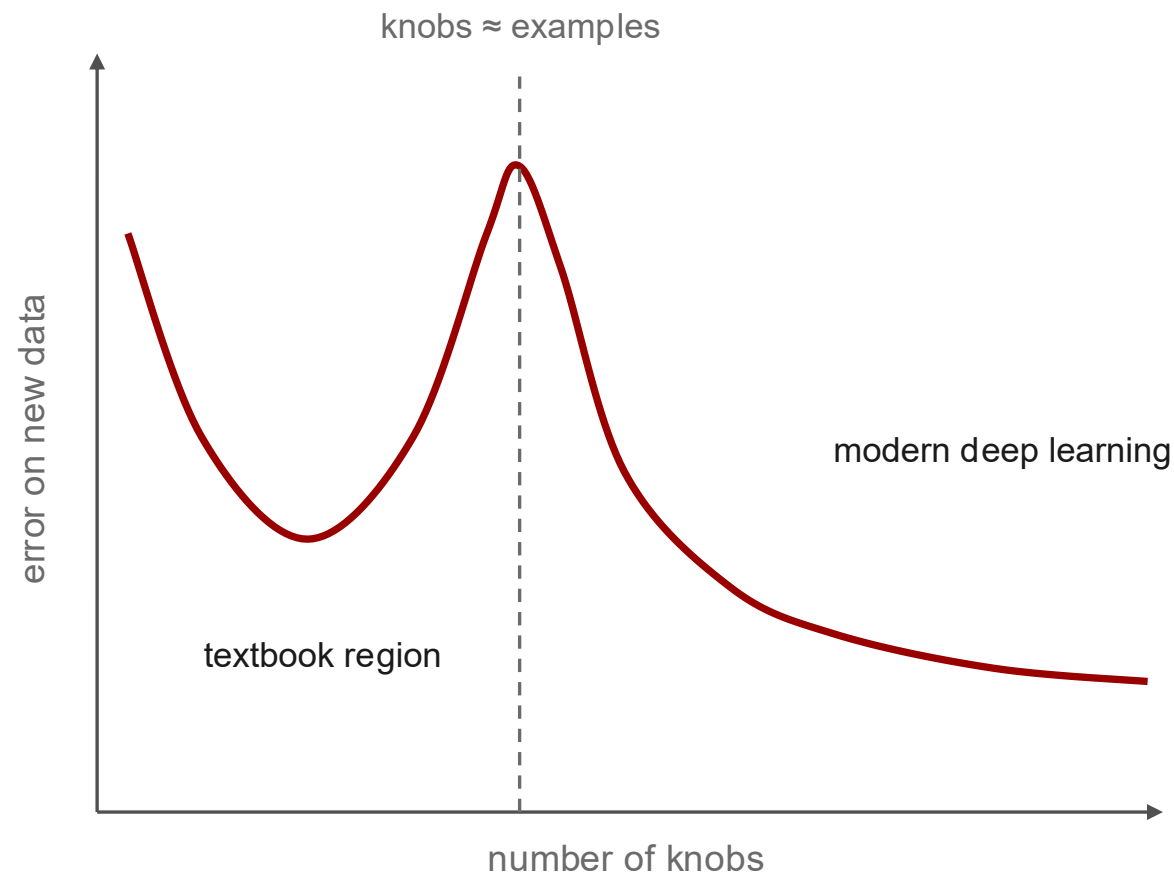
training photos: **100 %** correct (!)
new photos: **10 %** — pure guessing
it memorized 50,000 random answers

- ▶ So the network **can** be a lookup table. With real labels, it **chooses** not to be one.
- ▶ Counting knobs cannot explain this — the answer must be in **which** solution training finds.

In one sentence: the capacity to memorize is there; something in training prefers the real rule.

Double descent: the error goes up — then down again

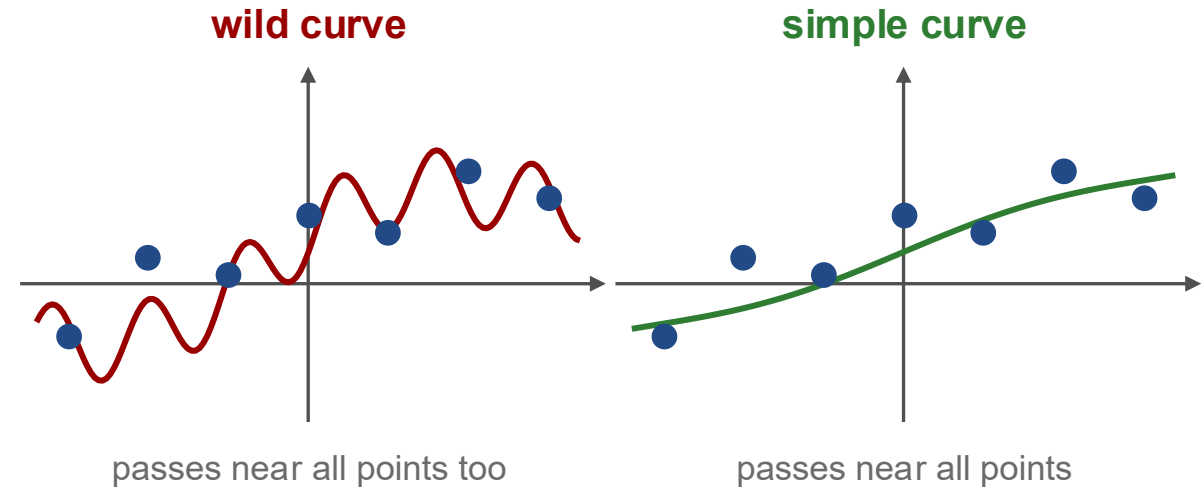
- ▶ Few knobs: the classic U-shape.
- ▶ Knobs $\approx$ examples: the model is **forced** to pass through every noisy point $\rightarrow$ it wiggles wildly $\rightarrow$ worst error.
- ▶ Many more knobs: the error goes **down again** — often below the textbook optimum.
- ▶ Seen everywhere: image networks, Transformers, even plain linear regression.



In one sentence: past a certain size, adding knobs makes the model generalize better, not worse.

Why? Training prefers *simple* answers

- ▶ With huge models, **endless** settings of the knobs fit the training data.
- ▶ Most of them are wild curves; a few are simple and smooth.
- ▶ Random knobs almost always give a **simple** curve — wild curves need very special knob settings.
- ▶ Walking downhill from a random start stops at the first good place → a simple solution.

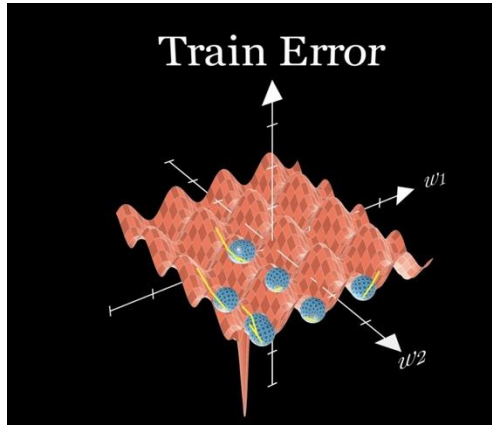


Physicist's version

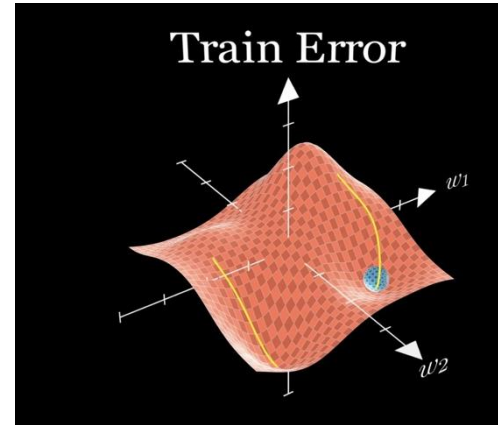
Occam's razor is built in: the network + random start + small steps favour simple functions — and the real world is mostly simple.

In one sentence: more knobs give more ways to be simple, so the simple answer becomes even more likely.

Big networks: many small networks in one

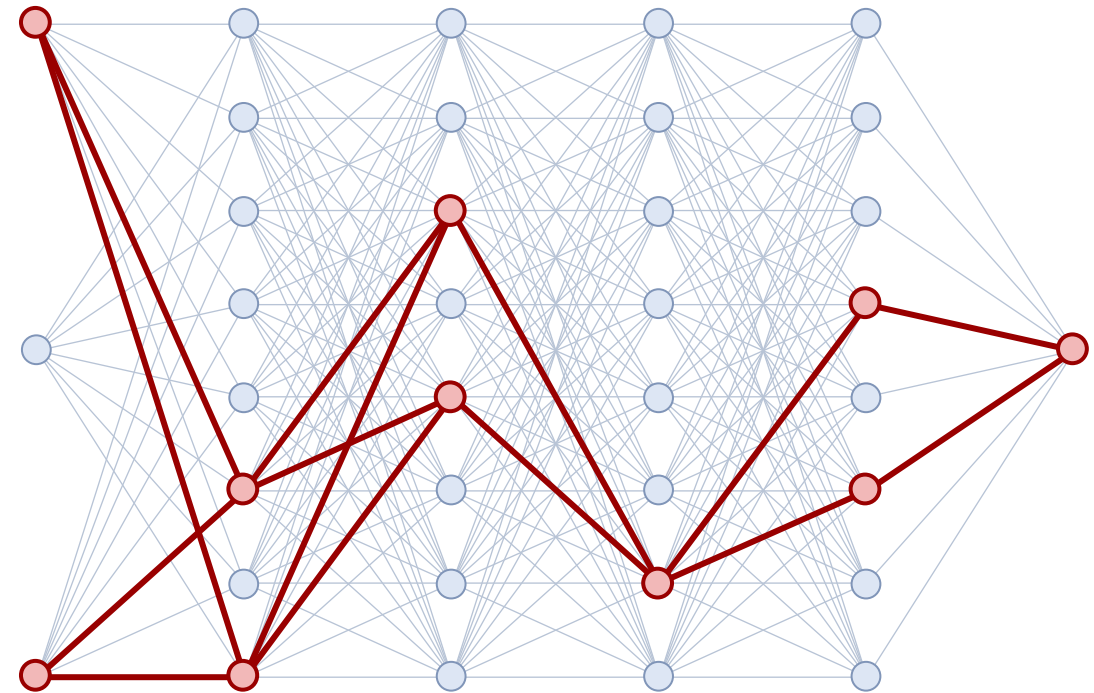
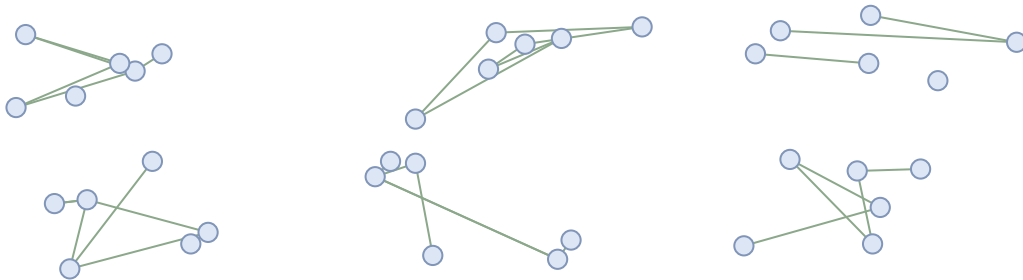


small network: many traps



huge network: a way down

many different small networks ...

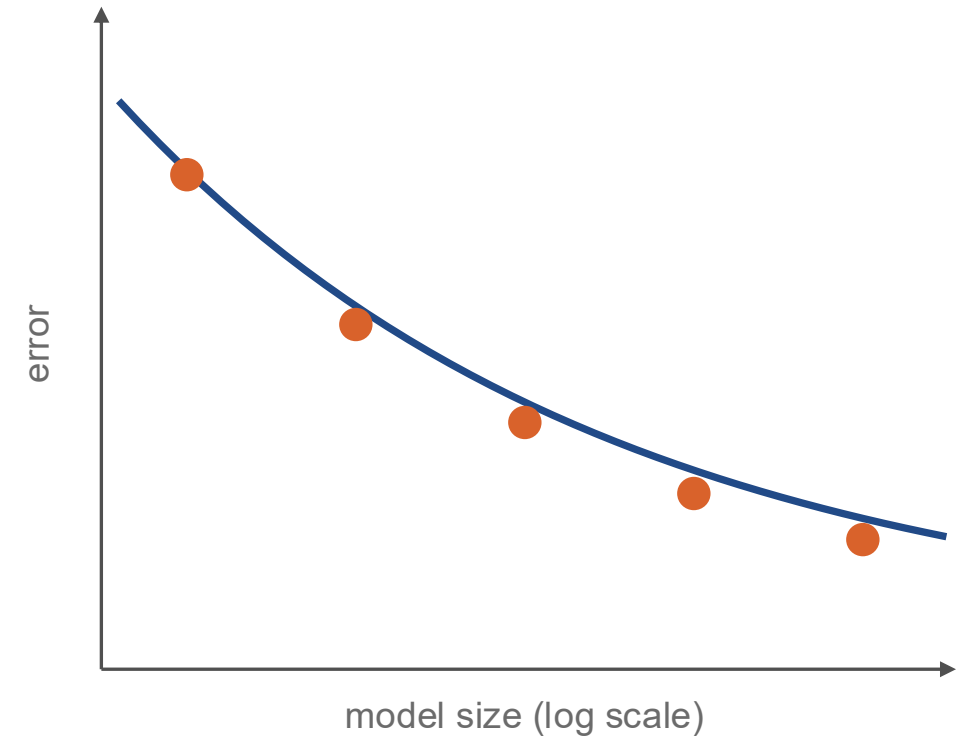


... all hidden inside one big network: training finds a good one (red)

In one sentence: a huge network tries many small networks at once — one of them is bound to work well.

Why this matters for ChatGPT

- ▶ **Scaling laws:** the error falls smoothly and predictably as we add knobs, data and computing power.
- ▶ Around 2018 almost nobody expected “just make it bigger” to keep working.
- ▶ To predict **all** human text well, the simplest solution must contain grammar, facts, arithmetic, some reasoning...



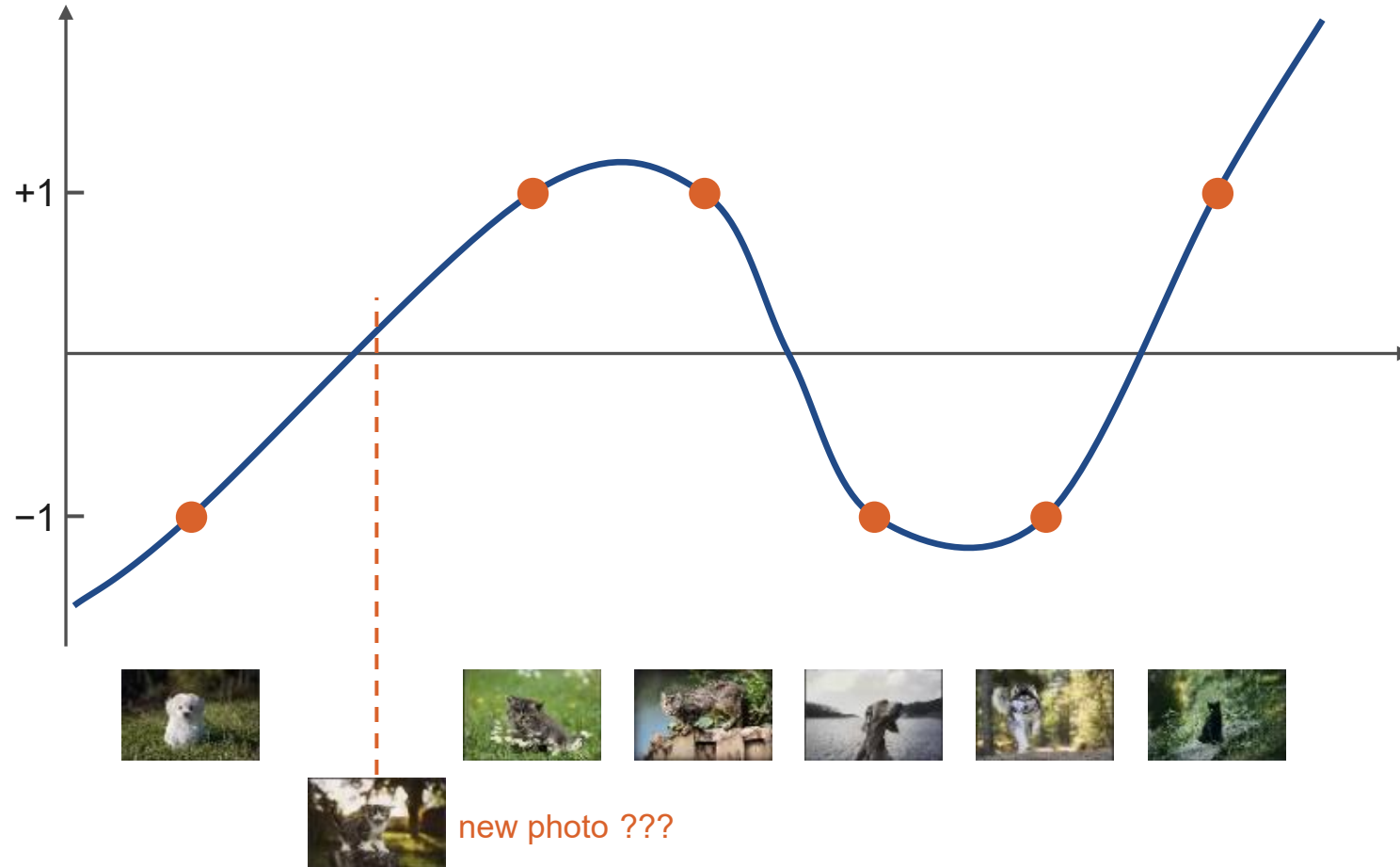
In one sentence: bigger works better — and the simplest way to predict text is to “understand” it a little.



Part III: Images — convolutional networks (CNN)

look at small patches, then patches of patches

Classification is fitting a curve, too



- ▶ Put each photo on the x-axis.
- ▶ Its label on the y-axis: **+1 = cat**, **-1 = dog**.
- ▶ The network draws a curve through the points.
- ▶ New photo → read the curve: above 0 → cat.

In one sentence: “cat or dog?” is the same curve-fitting game — but where on the x-axis does a photo go?

A photo is a point in a gigantic space



red: 0.2
green: 0.9
blue: 0.3

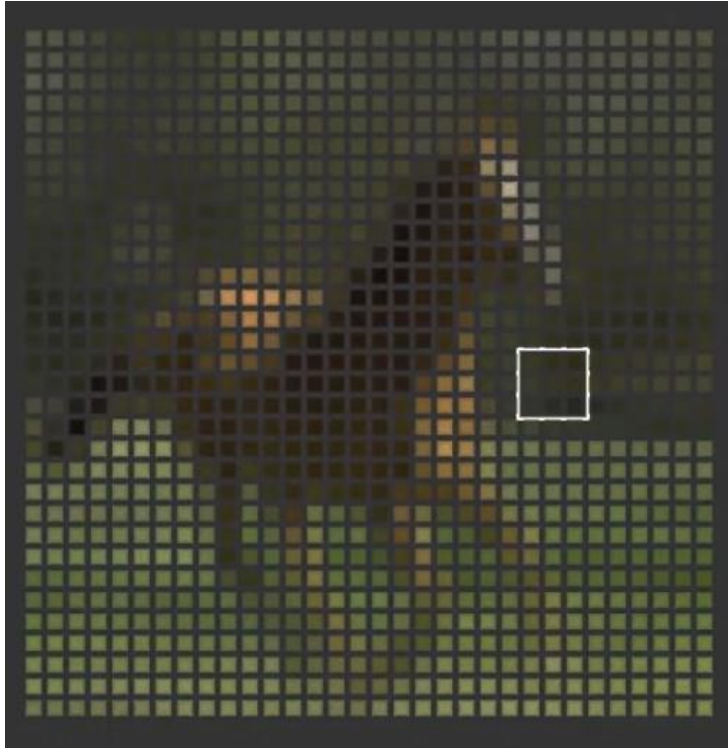
every pixel = 3 numbers

red: 0.4
green: 0.7
blue: 0.2

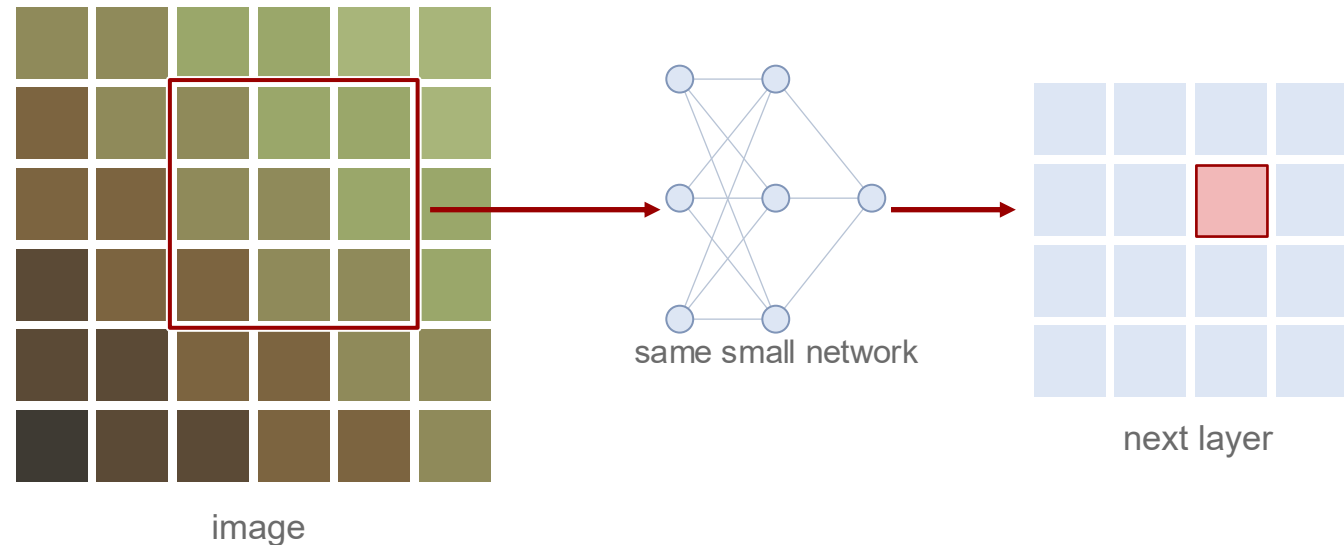
- ▶ 32×32 pixels = 1024 pixels
- ▶ $\times 3$ colours = **3072 numbers**
- ▶ one photo = **one point** in a 3072-dimensional space
- ▶ to fill this space with 9 points per direction:
 $9^{3072} \approx 10^{2931}$ photos
- ▶ atoms in the universe: about 10^{80}

In one sentence: we can never have enough photos to fill this space — we must use what we know about images.

Idea: look at small patches (convolution)



what is in the little square: horse or grass?



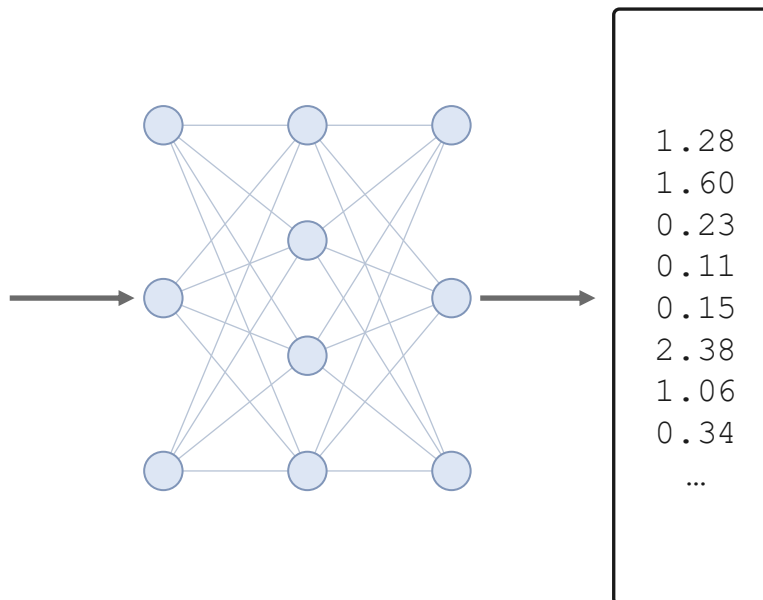
- ▶ The **same** small network slides over every 3×3 patch.
- ▶ It learns simple things: edges, fur, grass. The next layer looks at patches of patches $\rightarrow$ legs, heads, horses.

In one sentence: nearby pixels belong together: learn small local patterns once, reuse them everywhere.

A patch becomes a list of numbers

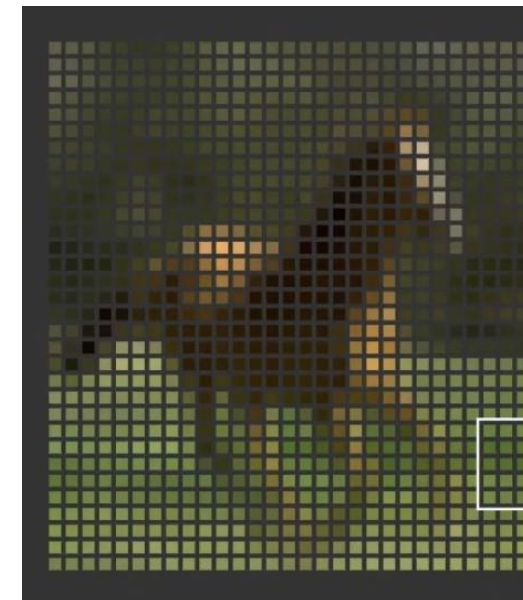
0.32 0.47 0.15	0.33 0.48 0.16	0.34 0.48 0.18
0.34 0.48 0.17	0.33 0.47 0.16	0.32 0.45 0.16
0.44 0.57 0.25	0.44 0.54 0.25	0.41 0.53 0.23

3×3 patch: 9 pixels × 3 numbers



many numbers out

1.28
1.60
0.23
0.11
0.15
2.38
1.06
0.34
...



⇒ grass?

- ▶ The output is not just “grass”: it is a **list of numbers** (features) describing the patch — how green, how striped, how bright ...
- ▶ Information is not thrown away: each layer re-describes the image in more useful numbers.

In one sentence: each layer turns patches into richer descriptions — and passes them on.

Does it work? Yes — because neighbours belong together

Model	Accuracy
Plain neural network	74.3 %
Convolutional network	95.3 %
Human	99.5 %

CIFAR-10 photos (10 classes), numbers from the video

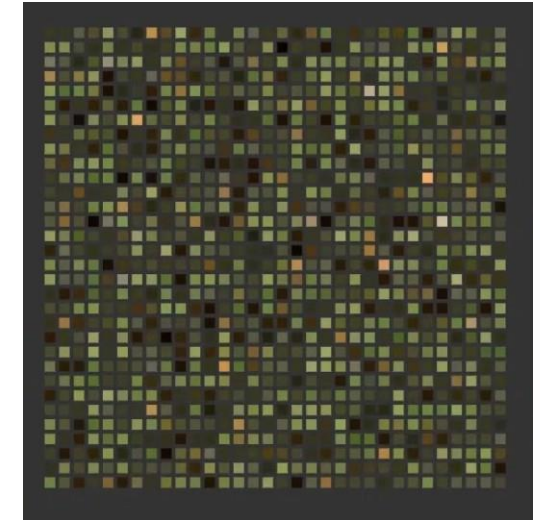
- ▶ Shuffle the pixels in every photo the same way: a plain network doesn't care — a CNN drops to 62.9 %.
- ▶ A CNN works **because** nearby pixels belong together. That is the knowledge we built in.

95.3 %



normal photo

62.9 %



pixels shuffled (always the same way)

In one sentence: build in what you know about the data (here: locality) and you need far fewer examples.

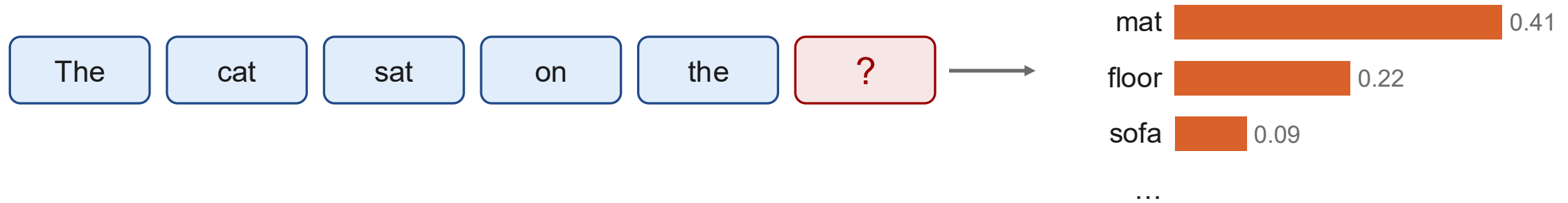


sPHENIX Preliminary
2025-6-14, Run 66641
BCO: 722670579120701
Collisions: Au+Au @ $\sqrt{s_{NN}} = 200$ GeV

Part IV: Text — attention and Transformers

the engine behind ChatGPT

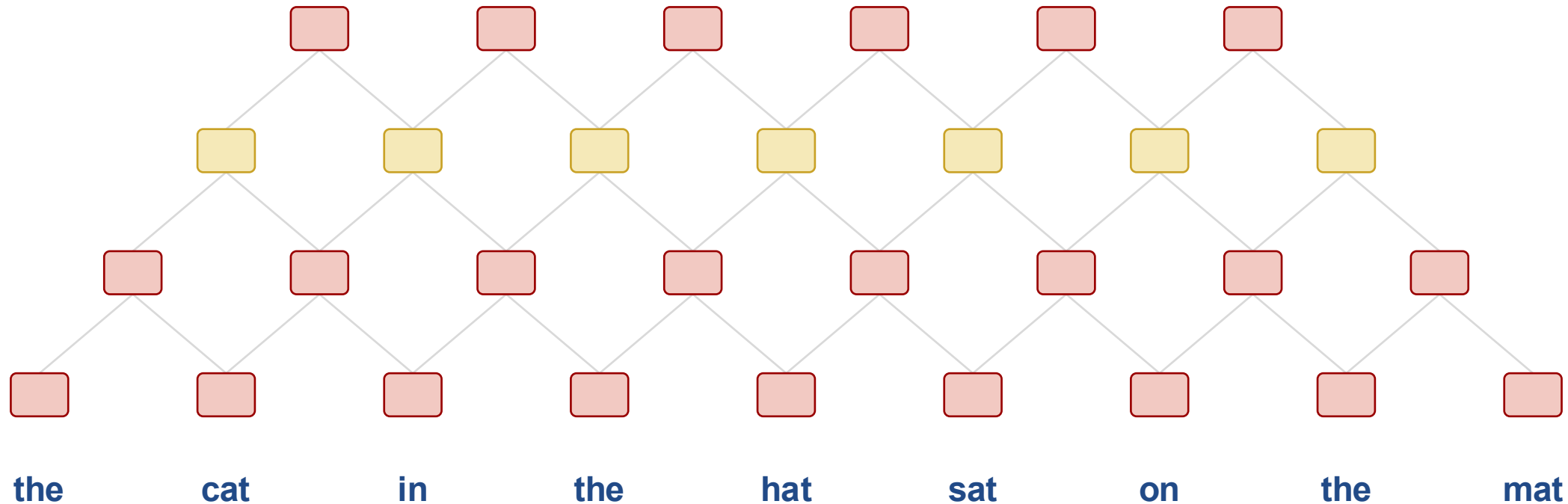
The task: predict the next word



- ▶ Each word becomes a list of ~1000 numbers (its “embedding”) — learned like all other knobs.
- ▶ Output: a probability for every word in the vocabulary (~50,000 words and word pieces).
- ▶ Free training data: in any text, the next word is always known.
- ▶ Difficulty: texts have different lengths, and the important word can be anywhere earlier.

In one sentence: a language model is a curve-fitter whose answer is “which word comes next?”.

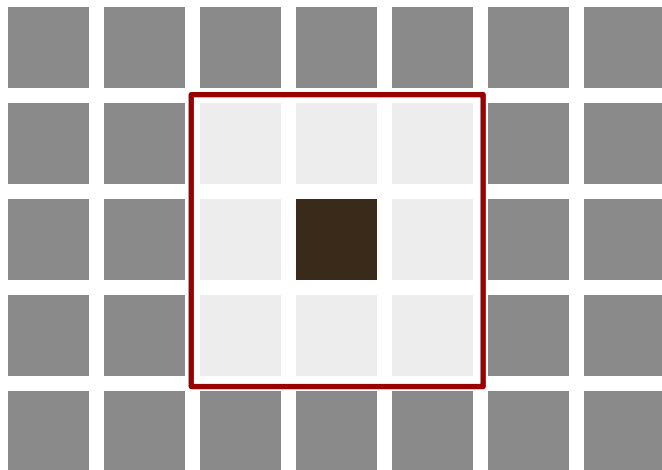
First try: a CNN for text



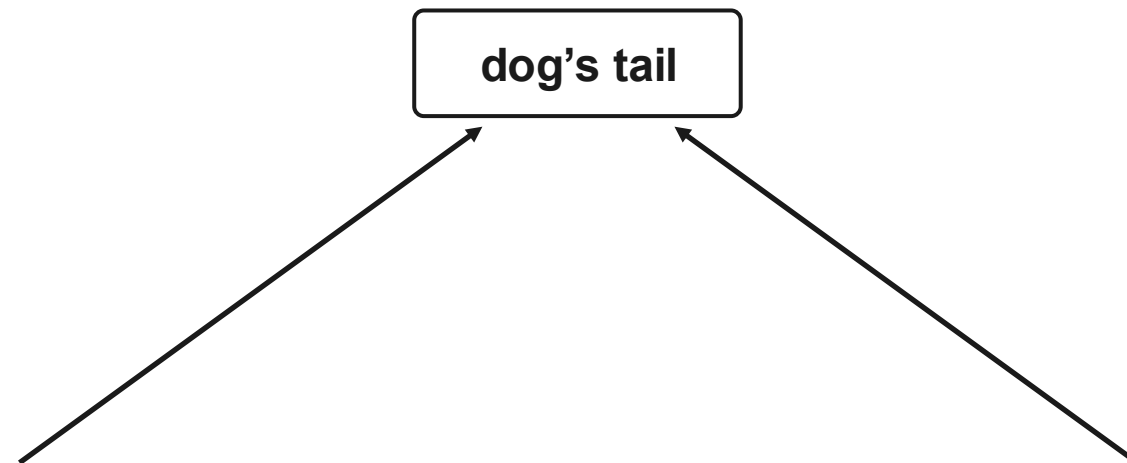
Combine neighbouring words, then neighbours of neighbours ... until the top of the pyramid sees the whole sentence.

In one sentence: it works — but two words only “meet” when the pyramid is tall enough to cover both.

Why CNNs work for images but not for text



Images: things that belong together are **close** (the pupil and the eye). A 3×3 patch sees them together.



the **dog** spun around so fast it caught its own **tail**

Text: words that belong together can be **far apart**. A CNN needs many layers before “dog” and “tail” meet — and the pyramid gets huge.

In one sentence: for text we need a way to connect any two words directly, however far apart they are.

Other first ideas that don't quite work

Fixed window

Feed the last k words into one network.

Problem: “cat” in position 3 and in position 7 look completely different → everything must be learned for every position.

Bag of words

Add up all word vectors, then a network.

Problem: Order and word pairs are lost: “dog bites man” = “man bites dog”.

Read word by word

Keep a memory, update it after each word (RNN).

Problem: Slow — one word after another, no parallel computing — and it forgets.

Wish list: any length · knows the order · connects any two words · fast on GPUs

In one sentence: we want every word to be able to look at every other word — directly and in parallel.

Idea: look at every *pair* of words

-2.8
-3.5

the the

-1.6
-2.2

the cat

-3.3
1.4

the sat

0.7
-4.1

cat the

-2.2
2.6

cat cat

1.3
-0.9

cat sat

2.4
0.4

sat the

-0.6
-0.8

sat cat

5.8
2.6

sat sat

Meaning often comes from **pairs**:

- ▶ “bank” + “river” → the side of a river
- ▶ “bank” + “loan” → a place with money

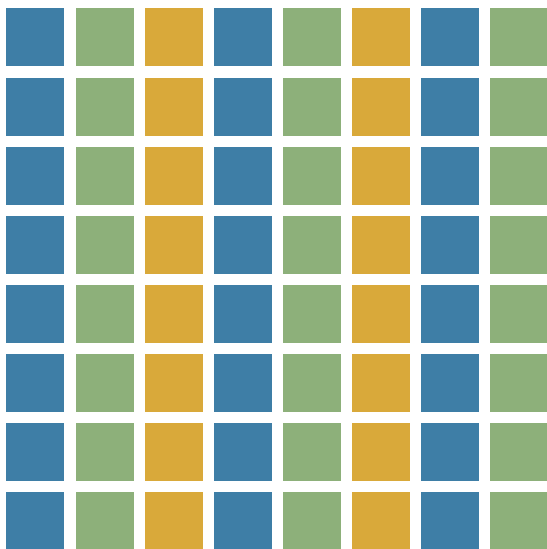
So: run one small network on **every pair** of words.

Any distance works — “dog” and “tail” are just another pair.

every pair → the same small network → a list of numbers

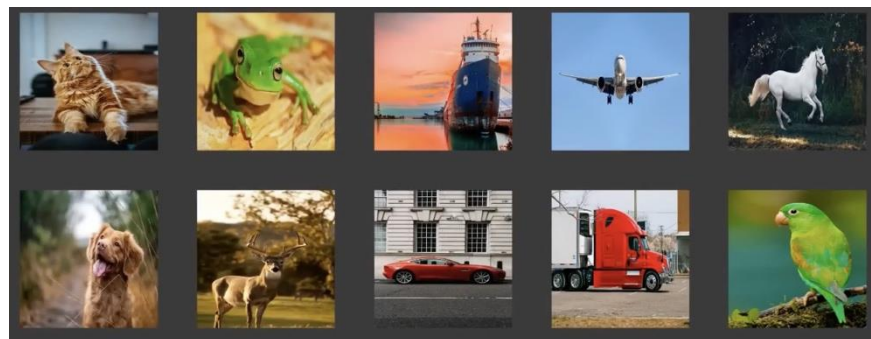
In one sentence: pairs connect any two words directly, no matter how far apart they are.

Problem: n words give n^2 pairs — how to combine them?



8 words $\rightarrow$ 64 pairs

Before the next layer we need **one** list of numbers per word again: $n^2 \rightarrow n$.



average

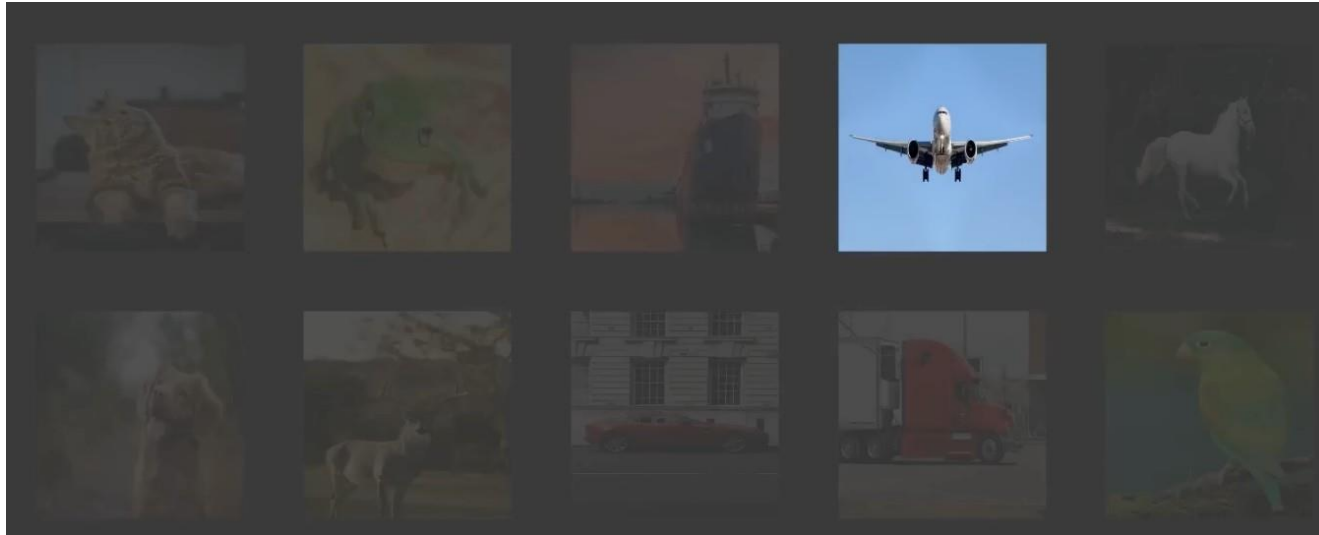


a blurry mess — looks like Monet 😊

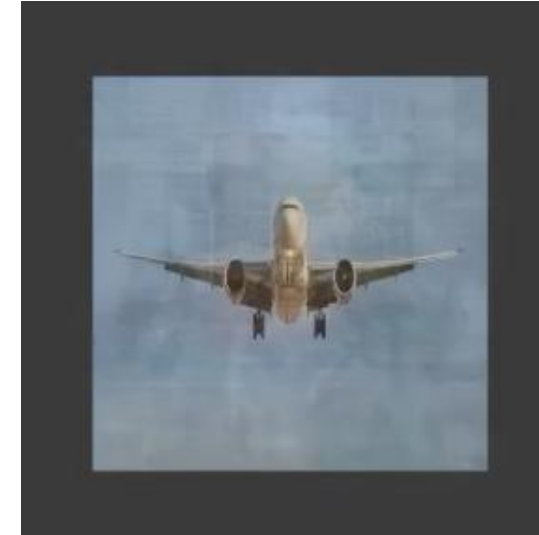
- ▶ Simplest idea: just add (average) the pairs.
- ▶ Like averaging 10 photos: everything gets mixed and the useful detail disappears.

In one sentence: a plain average throws the important information away.

Better: a *weighted* average — focus on what matters



weight ≈ 1 for the airplane, ≈ 0 for everything else



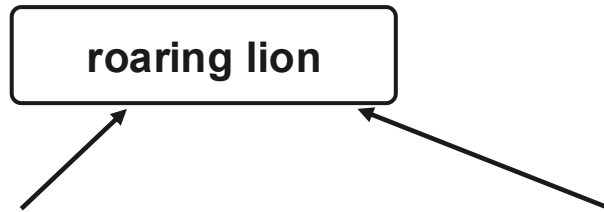
weighted average: sharp

- ▶ Give every item a **weight**: important $\rightarrow$ large, unimportant $\rightarrow$ almost 0.
- ▶ The weights add up to 1, so the result is still an average — but a sharp one.

In one sentence: attention is just a weighted average — with cleverly chosen weights.

Example: what is “roaring”?

there was a tree on **fire** and it was **roaring**



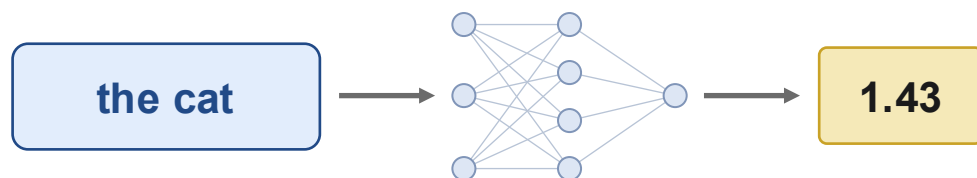
there was a **lion** on fire and it was **roaring**

- ▶ In the first sentence “roaring” belongs with **fire**; in the second with **lion**.
- ▶ Plain average: every pair gets 0.1. Attention: almost all weight on the one pair that matters.

“roaring” +	plain	tree	lion
there	0.1	0	0
was	0.1	0	0
a	0.1	0	0
tree / lion	0.1	0	0.9
on	0.1	0	0
fire	0.1	1.0	0.1
and	0.1	0	0
it	0.1	0	0
was	0.1	0	0
roaring	0.1	0	0

In one sentence: the weights must depend on the words themselves — the same word needs different partners in different sentences.

Who decides the weights? Another small network



importance network: one score for every pair

- 1 **score**
a small network gives every pair a score: “how relevant is this word for that one?”
- 2 **normalize**
turn the scores into weights that are positive and add up to 1 (softmax)
- 3 **combine**
weighted average of the pairs → one new list of numbers per word

In one sentence: attention = score every pair, turn the scores into weights, take the weighted average.

In the papers: queries, keys and values

Query

what am I looking for?

Key

what do I contain?

Value

what do I pass on?

- ▶ Every word makes three lists of numbers.
- ▶ Score of a pair = how well the **query** of one word matches the **key** of the other.
- ▶ Output = weighted average of the **values**.

Example

“The animal didn’t cross the street because **it** was tired.”

query of “it”: *a noun that can be tired*

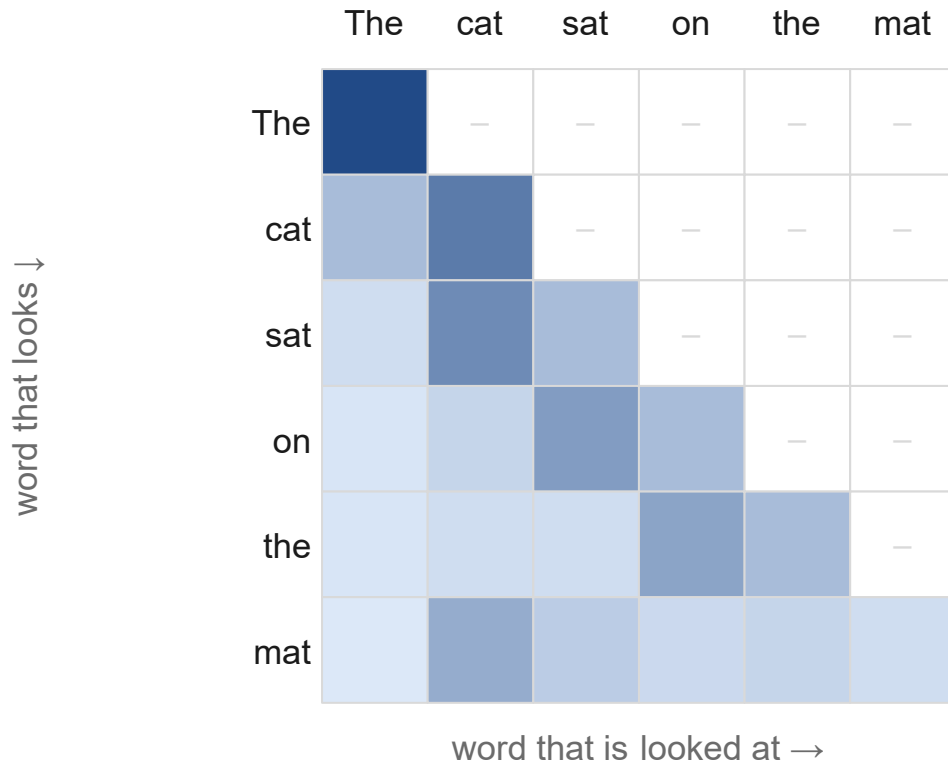
key of “animal” matches better than key of “street”

$$\text{Attention}(Q, K, V) = \text{softmax}(Q \cdot K^T / \sqrt{d}) \cdot V$$

score · normalize · combine — all words at once, one big matrix product

In one sentence: query asks, key answers, value is what gets passed along.

What the attention weights look like



- ▶ Row = one word; dark = it pays a lot of attention there.
- ▶ “mat” looks mostly at “cat” and “sat” (who sat where?).
- ▶ Empty upper half: a word may not look at **future** words — otherwise predicting the next word would be cheating.
- ▶ One pass over a text trains **every** next-word prediction in it at once.

In one sentence: each row is a word’s “focus”: a weighted average over the words before it.

Two fixes: word order and many “heads”

Word order

Pairs alone don't know the order:

the fat cat = the cat fat

Fix: add the position (1st, 2nd, 3rd ...) to each word's numbers before attention:

the+1 fat+2 cat+3

Many heads

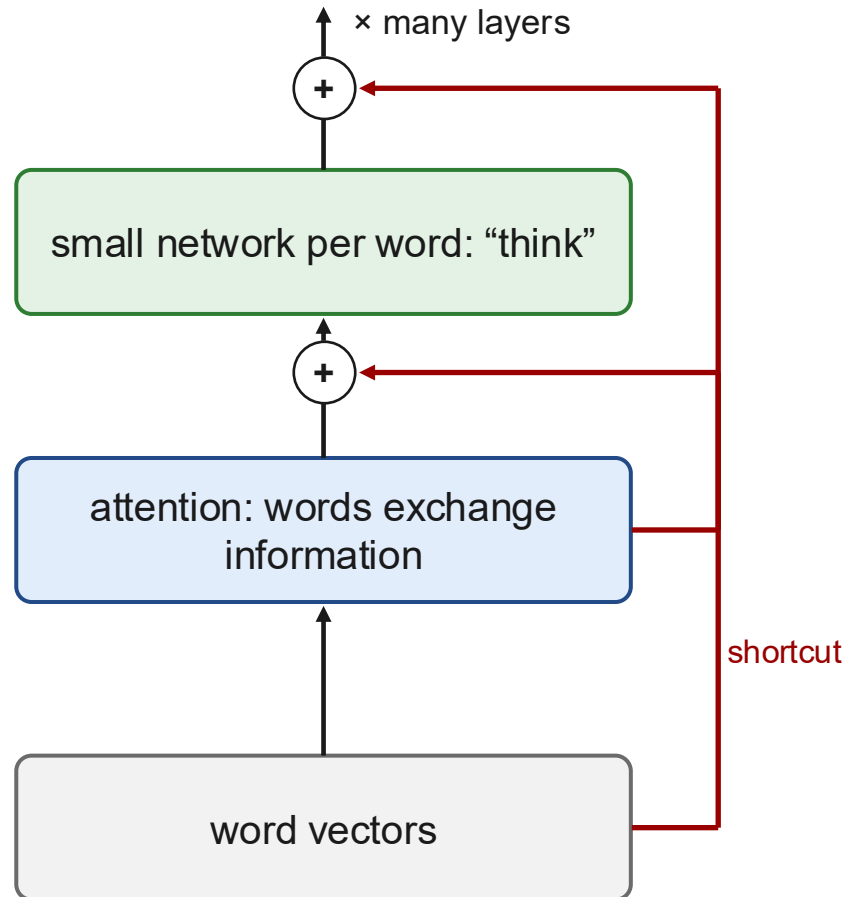
One weighted average picks roughly one kind of relation.

So run several attentions side by side — “heads” — each with its own knobs.

They specialise: “previous word”, “who does the verb”, “which bracket closes this one” ...

In one sentence: add positions so order matters; use several heads so many relations are tracked at once.

The Transformer block

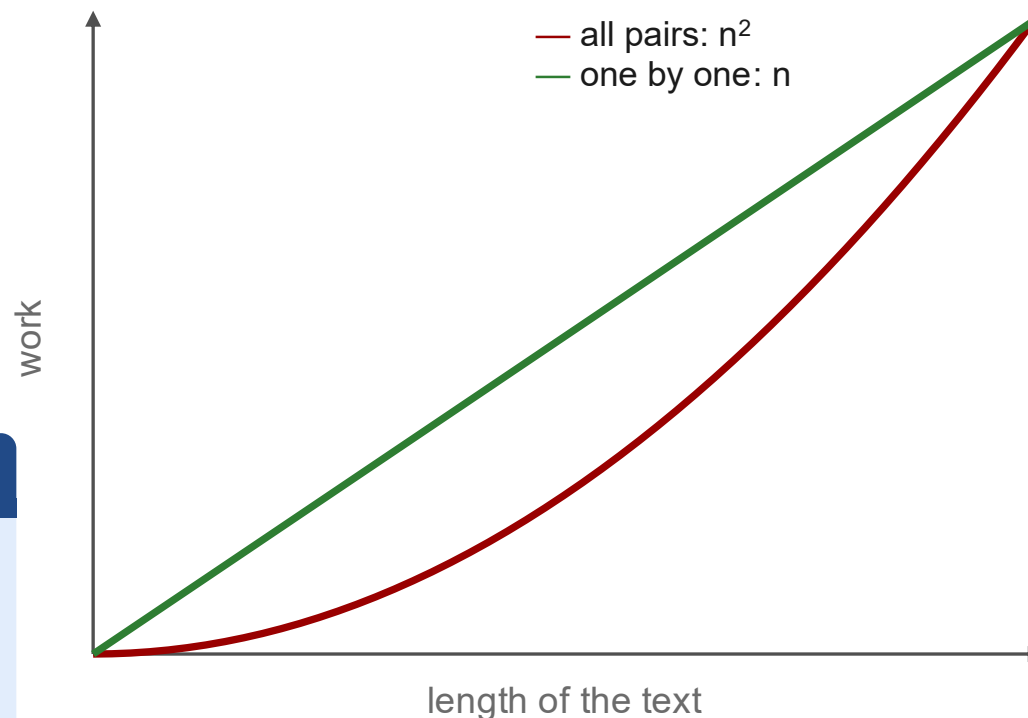


- ▶ **Attention:** words look at each other and exchange information.
- ▶ **Small network per word:** each word “thinks” about what it collected (Part I: folding). Most knobs live here.
- ▶ **Shortcuts (+):** each block only *adds a correction* — this keeps training stable through 100+ layers.
- ▶ GPT = many of these blocks + “no peeking ahead” + next-word prediction.

In one sentence: a Transformer alternates “talk to the other words” and “think on your own”, many times.

The price: the work grows like n^2

- ▶ Every word looks at every word: $n \times n$ pairs.
- ▶ 1,000 words $\rightarrow$ 1 million pairs; 100,000 words $\rightarrow$ 10 billion pairs.
- ▶ While writing, each new word must look back at **all** previous words.



Transformers in short

- ✓ fast to train (everything in parallel)
- ✓ any word can reach any word
- ✓ scale beautifully
- ✗ long texts are expensive

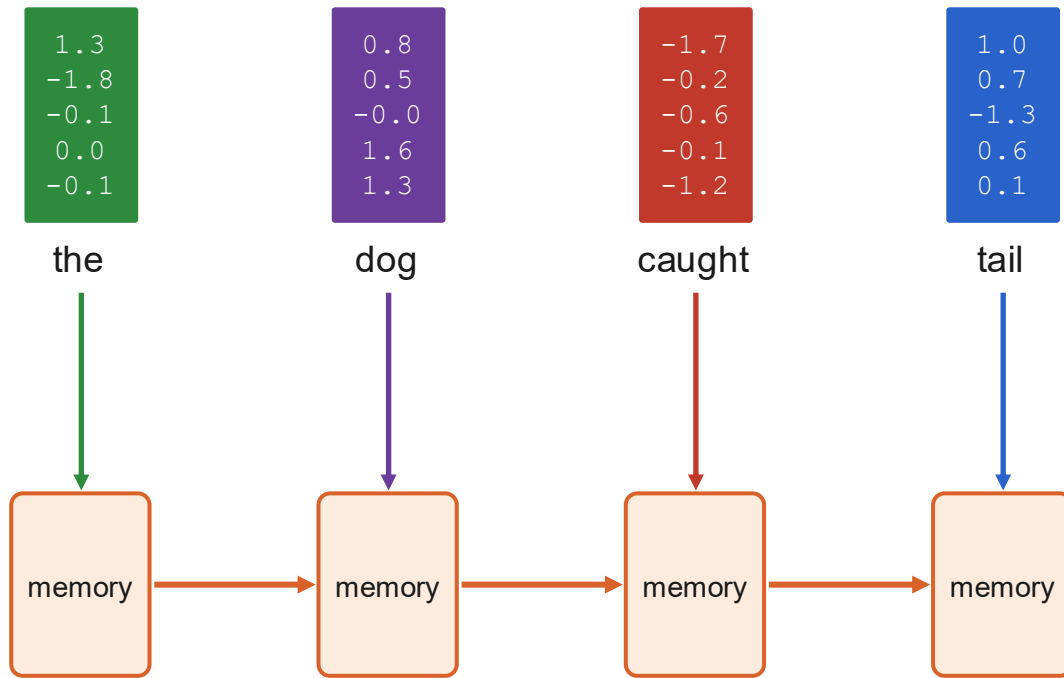
In one sentence: 10× longer text $\rightarrow$ 100× more work. Can we get the benefits at linear cost?



Part V: Mamba — reading with a memory

fast models for very long texts

Recurrent network: read one word at a time

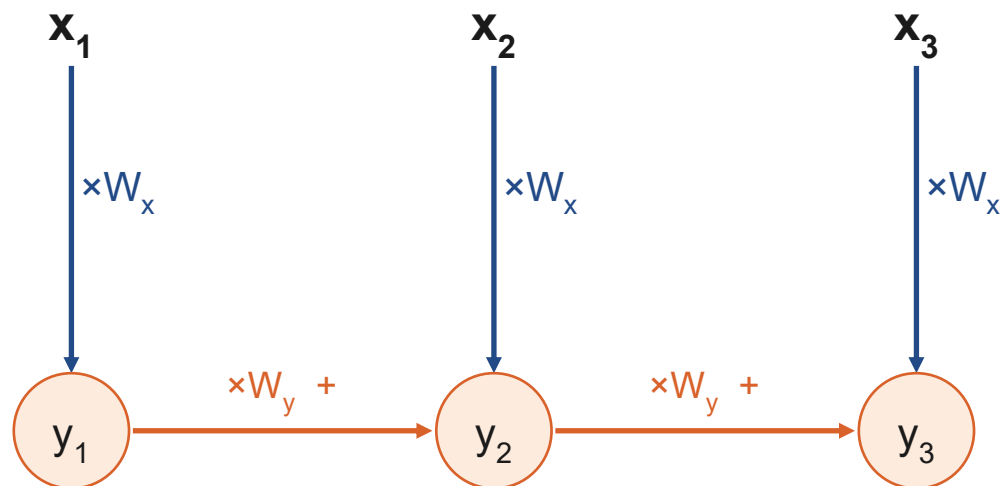


the memory is updated after every word by the same small network

- + Any length; long-range information for free (it is in the memory).
- + Same amount of work per word as a CNN; total work $\propto n$, not n^2 .
- Cannot be parallelized: word 1000 must wait for word 999 $\rightarrow$ slow on GPUs.
- Very hard to train: the memory fades or explodes.

In one sentence: a recurrent network is cheap per word — but slow and forgetful in practice.

What exactly goes wrong? Multiplying again and again



$$y_i = W_y \cdot y_{i-1} + W_x \cdot x_i$$

1. **Sequential:** y_3 needs y_2 , which needs y_1 ... the GPU waits.

2. **Unstable memory:** information from word 1 is multiplied by W_y at every step:

$1.1^{100} \approx 14,000$ (explodes)

$0.9^{100} \approx 0.00003$ (fades away)

But: fixed-size memory $\rightarrow$ constant cost per word, cost $\propto n$ in total.

In one sentence: repeat a multiplication 100 times and almost any number explodes or vanishes.

Linear RNN: take the network out of the loop

Classic RNN

$$y_i = W_3 \cdot \text{ReLU}(W_1 \cdot x_i + W_2 \cdot y_{i-1})$$

a full network **inside** the loop: bends inside the memory update



Linear RNN

$$y_i = W_1 \cdot x_i + W_2 \cdot y_{i-1}$$

only multiply and add inside the loop; apply a full network to each output **afterwards**

- Surprise: removing the bend from the loop loses almost nothing — the network after it still bends.
- And a linear loop can be computed **in parallel** (next slide).

Orvieto, Smith, Gu, Fernando, Gulcehre, Pascanu, De — “Resurrecting Recurrent Neural Networks for Long Sequences”, ICML 2023

In one sentence: keep the memory update simple (linear) and put the complicated part outside the loop.

Linear means: no need to wait

Write out the loop $y_i = W \cdot y_{i-1} + x_i$ (with W_1 absorbed into x):

$$y_1 = x_1$$

$$y_2 = W \cdot x_1 + x_2$$

$$y_3 = W^2 \cdot x_1 + W \cdot x_2 + x_3$$

$$y_4 = W^3 \cdot x_1 + W^2 \cdot x_2 + W \cdot x_3 + x_4$$

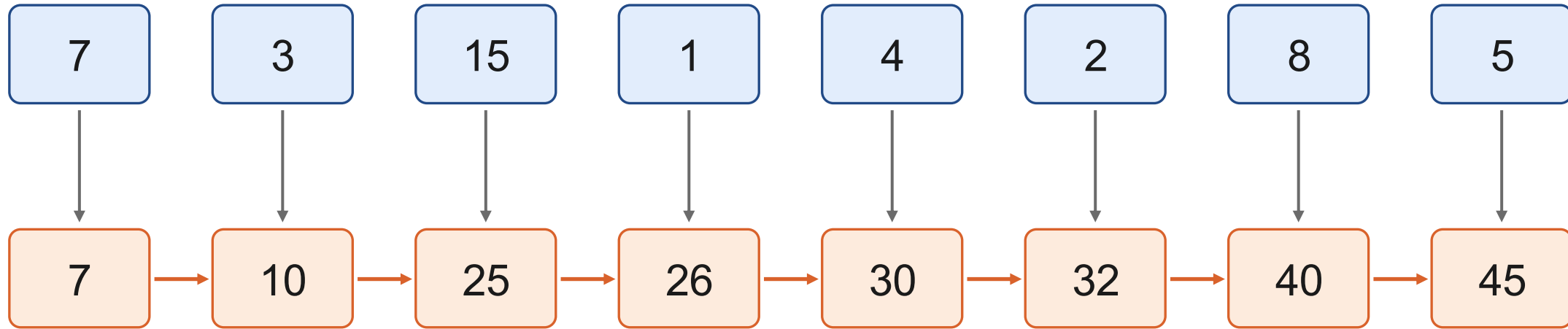
- ▶ Every output is just a **weighted sum of the inputs**.
- ▶ Two steps can be merged into one step:
 (W_2, b_2) after $(W_1, b_1) = (W_2 \cdot W_1, W_2 \cdot b_1 + b_2)$

- ▶ Merging is **associative**: like addition, it does not matter how you group the steps.
- ▶ So we can merge pairs of steps in parallel — like computing running totals fast.

In one sentence: a linear memory can be computed like running totals — in parallel.

Running totals: one by one

numbers

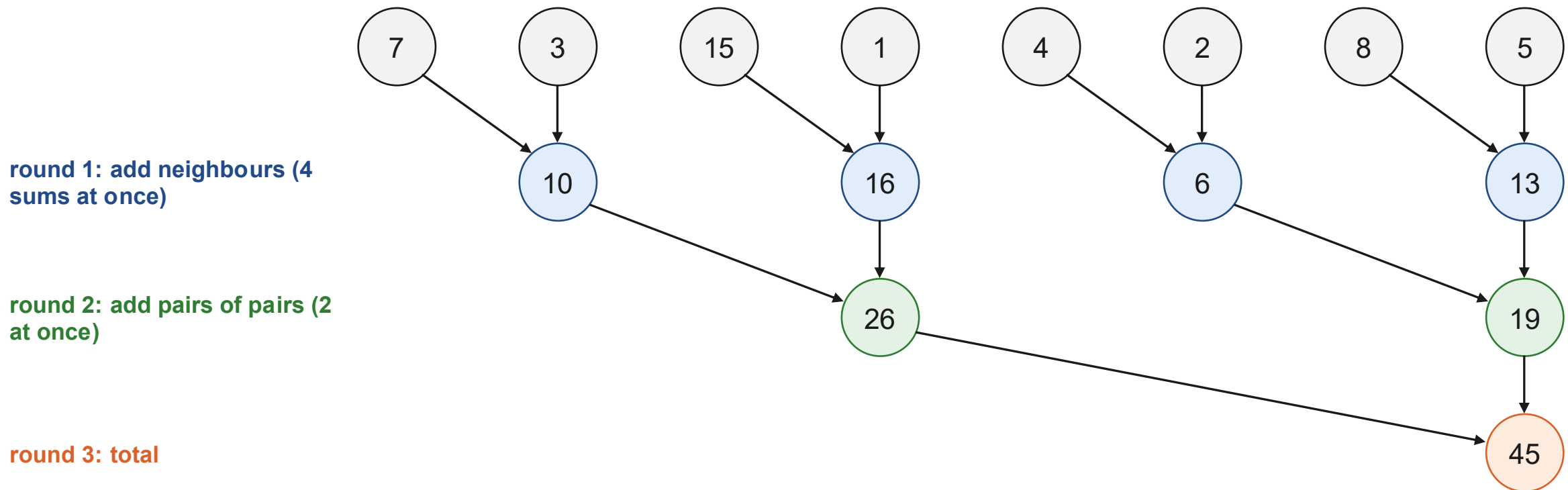


running totals

- One by one: 7, then $7+3 = 10$, then $10+15 = 25$, then $25+1 = 26$...
- n numbers $\rightarrow n$ steps, each waiting for the one before.

In one sentence: this is exactly how a recurrent network runs: step after step after step.

Running totals in parallel: a few rounds instead of n steps

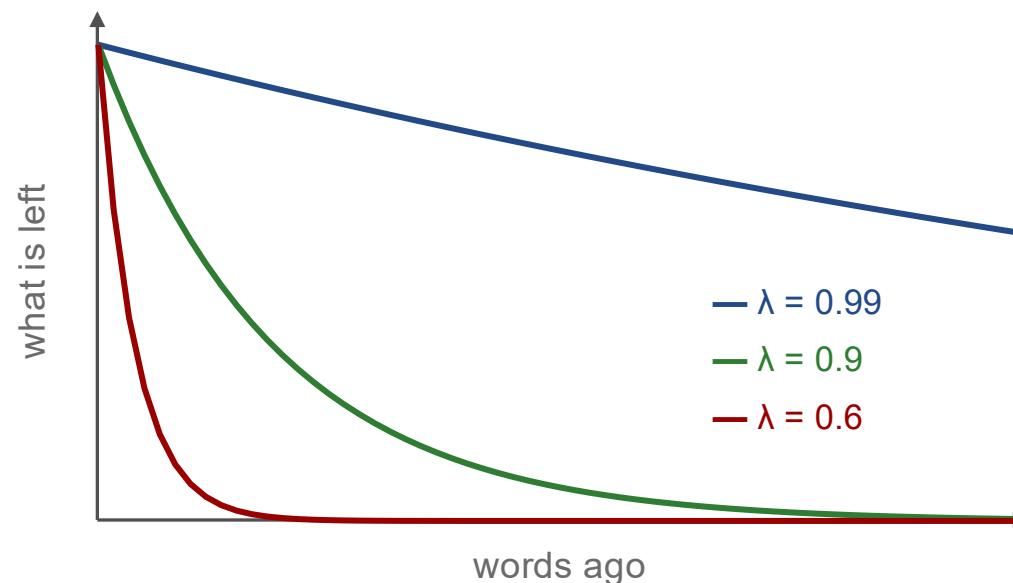


- ▶ A second sweep back down fills in all the other running totals (10, 25, 26, 30 ...).
- ▶ 8 numbers: 3 rounds. A million numbers: ~20 rounds. Works for any associative merge — including the memory update.

In one sentence: a linear recurrent network trains in parallel, like a Transformer, but with work $\propto n$.

Fixing the memory: one number per memory slot

- ▶ Multiplying by a big matrix W again and again is slow and unstable.
- ▶ Trick: write $W = P \cdot D \cdot P^{-1}$ with D **diagonal**. Then $W^2 = P \cdot D \cdot P^{-1} \cdot P \cdot D \cdot P^{-1} = P \cdot D^2 \cdot P^{-1}$
- ▶ Powers of a diagonal matrix = powers of single numbers: each memory slot just gets multiplied by its own λ .
- ▶ Cost per step: $m^3 \rightarrow m$.

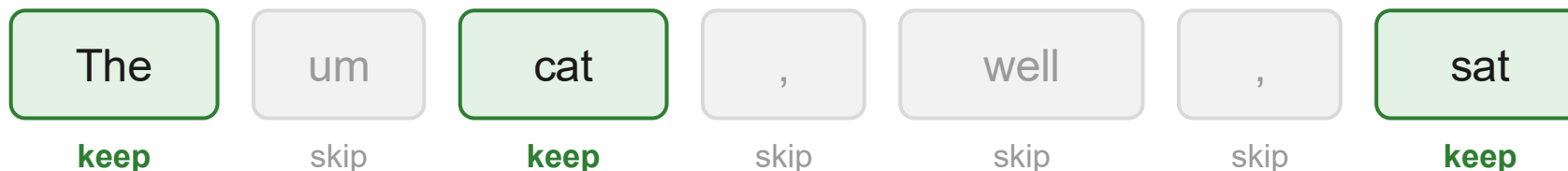


Physicist's view

Keep every $|\lambda| < 1$ (no explosions), start close to 1 (long memory). A bank of damped oscillators driven by the input.

In one sentence: each memory slot fades at its own speed λ — and the network learns the speeds.

Mamba's idea: let each word decide what to remember



- ▶ With fixed speeds λ , every word is remembered the same way — filler words too.
- ▶ Mamba: the forgetting speed is computed **from the current word** (a knob Δ):
 - Δ large $\rightarrow$ forget the old, store this word
 - $\Delta \approx 0 \rightarrow$ ignore this word, keep the memory
- ▶ Still linear in the memory $\rightarrow$ still the fast parallel trick.

Why it matters

This is what attention does well: choose what is important.

Mamba gets it with a fixed-size memory $\rightarrow$ constant cost per word.

(Gu & Dao, 2023: “selective state spaces”)

In one sentence: Mamba = fast linear memory + each word chooses how much to remember.

Mamba in practice

$\sim 2\times$

smaller than a Transformer of the same quality

$\sim 5\times$

faster when writing text

10^6

words of context are feasible

Gu & Dao (2023), language modelling results

- ▶ Clever engineering: the memory is kept in the GPU's small, very fast on-chip memory.
- ▶ Trade-off: a fixed-size memory must **compress** the past — exact recall (“copy the phone number from page 1”) is easier for attention.
- ▶ Many new models are **hybrids**: mostly Mamba-style layers plus a few attention layers.

In one sentence: Mamba is cheap for long texts; attention is best at exact recall — so people mix them.



sPHENIX Preliminary
2025-6-14, Run 66641
BCO: 722670579120701
Collisions: Au+Au @ $\sqrt{s_{NN}} = 200$ GeV

Wrap-up

The big picture

	Plain network	CNN	Transformer	Mamba
Main idea	fold space	small patches	every pair of words	memory with choice
Good for	fixed-size input	images	text, anything	very long sequences
Work for long inputs	—	$\propto n$	$\propto n^2$	$\propto n$
Memory of the past	—	—	exact (all words)	compressed

- ▶ All are trained the same way: loss $\rightarrow$ slopes (backprop) $\rightarrow$ small steps downhill.
- ▶ All generalize for the same reason: training prefers simple solutions.

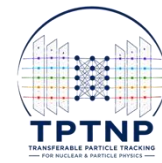
In one sentence: the architecture is how we tell the network what kind of data it will see.

Take-home messages

1. A neural network is a **curve-drawing machine**: multiply, add, bend, repeat. Depth folds space and gives very many pieces cheaply.
2. **Training** = measure the error, follow the slope downhill — a fit with billions of parameters.
3. Huge networks **don't just memorize**: training prefers simple answers.
4. **CNNs** work for images because nearby pixels belong together.
5. **Attention** = a weighted average over all pairs of words; powerful, but work $\propto n^2$.
6. **Mamba** = a fast linear memory where each word decides what to keep; work $\propto n$.

In one sentence: for us: hits $\rightarrow$ tracks is a set/sequence problem too — attention-based and graph-network tracking are active R&D.

Videos and references



Videos — Algorithmic Simplicity (YouTube)

- ▶ But what is a neural network REALLY? <https://youtu.be/FBpPjjhJGhk>
- ▶ Transformers / attention from scratch <https://youtu.be/kWLed8o5M2Y>
- ▶ MAMBA from scratch <https://youtu.be/N6Piou4oYx8>
- ▶ Why deep learning generalizes https://youtu.be/UKcWu1I_UNw

Papers

- ▶ Vaswani et al., *Attention Is All You Need*, NeurIPS 2017
- ▶ Gu & Dao, *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*, 2023
- ▶ Orvieto et al., *Resurrecting Recurrent Neural Networks for Long Sequences*, ICML 2023
- ▶ Zhang et al., *Understanding deep learning requires rethinking generalization*, ICLR 2017
- ▶ Belkin et al., *Reconciling modern machine-learning practice and the bias–variance trade-off*, PNAS 2019
- ▶ Montúfar et al., *On the Number of Linear Regions of Deep Neural Networks*, NeurIPS 2014



sPHENIX Preliminary
2025-6-14, Run 66641
BCO: 722670579120701
Collisions: Au+Au @ $\sqrt{s_{NN}} = 200$ GeV

Questions & discussion

A detailed 3D visualization of a particle collision event. A central point of interaction is shown with a dense, multi-colored burst of particles (red, orange, yellow, green, blue, and purple) radiating outwards. The particles are represented as thin lines and dots, creating a complex, star-like pattern. The background is a light gray, and the overall scene is set within a 3D coordinate system.