Lecture 5: Neural Networks

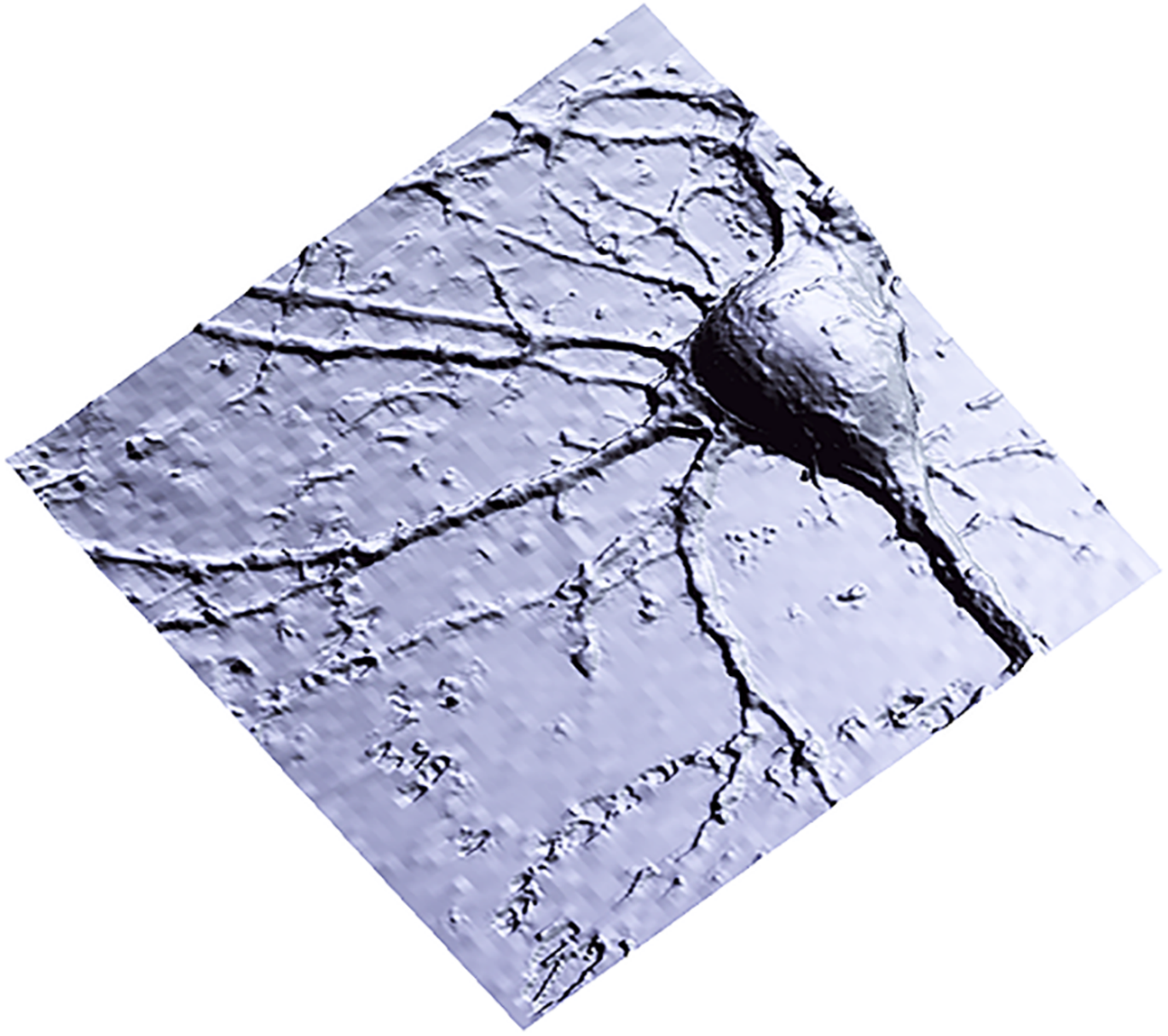
Matthew Caldwell

COMP0088 Introduction to Machine Learning • UCL Computer Science

Contents

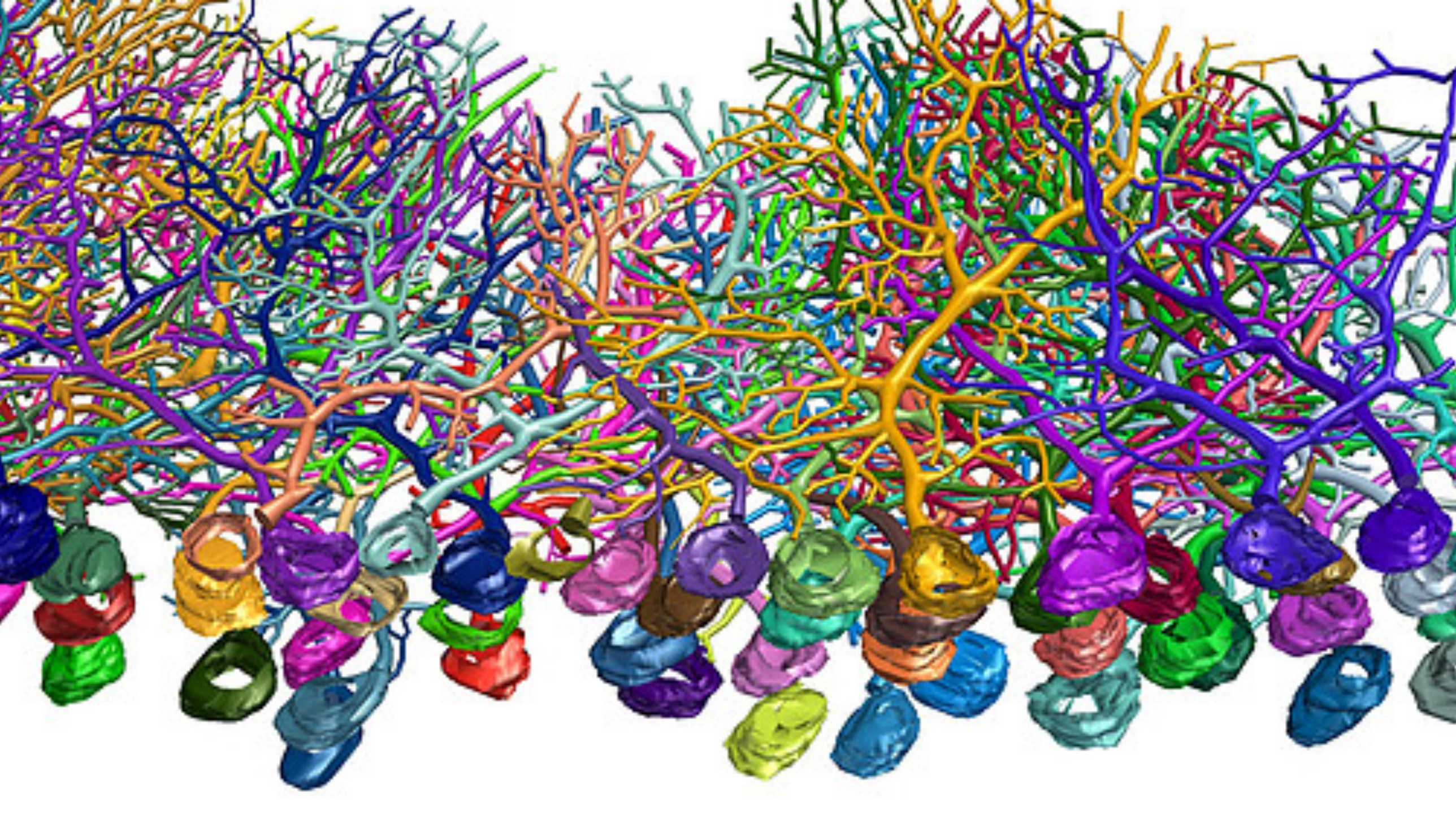
5.1: Neurons for Beginners

COMP0088 Introduction to Machine Learning • UCL Computer Science



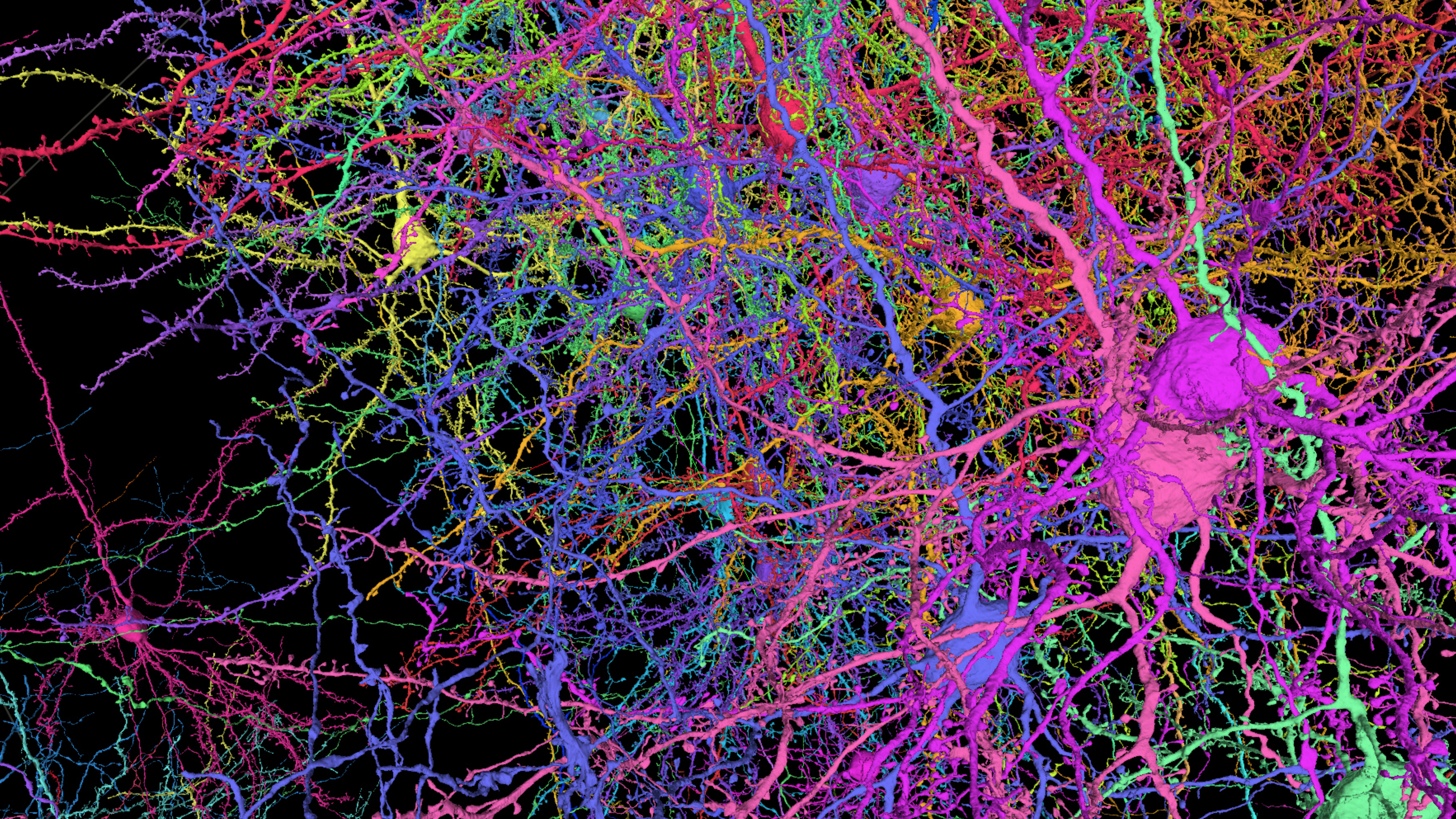
Neurons

- Neurons (nerve cells) are the primary computational elements of our brains and of information processing capacity throughout the animal kingdom
- They underpin complex behaviour even in very simple organisms
- Eg, the nematode worm *C. elegans* has a highly stereotyped central nervous system with just 302 neurons, forming ~7000 synapses (connections), but it is capable of food-seeking, threat avoidance, mating and learning
- The fruit fly *D. melanogaster* has about ~100k, ?(20-100)m synapses, and exhibits a wide range of complex behaviours, including flight
- 100k may sound like quite a lot, but humans have around 100 billion neurons, forming hundreds of trillions of synapses



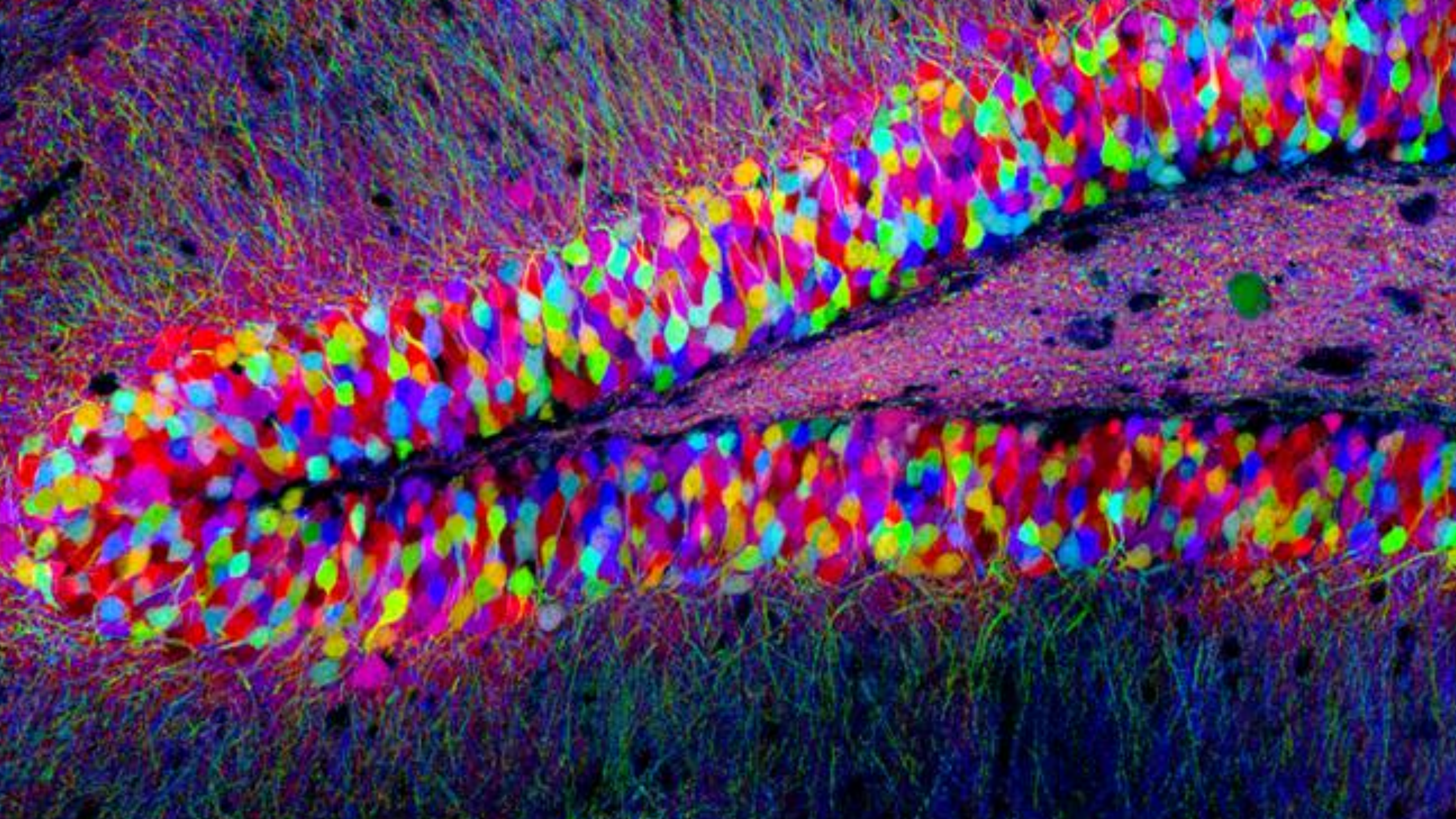
Electrochemical computation

- Neurons are analogue, noisy, highly complicated, highly varied
- A single neuron is a horribly complex dynamic system involving very large numbers of tiny flows of water & charged particles in and out of variably-permeable and morphologically complicated bags of glop
- Flows are modulated by fine-grained chemical signals picked up (mostly) from other neurons, (mostly) across a complex array of receiving tendrils known as dendrites, all operating over multiple overlapping timescales
- Synapses — the points of communication between neurons — come in several different types, but vast majority are “non-contact”, releasing their neurotransmitter into very tiny gaps between the cells



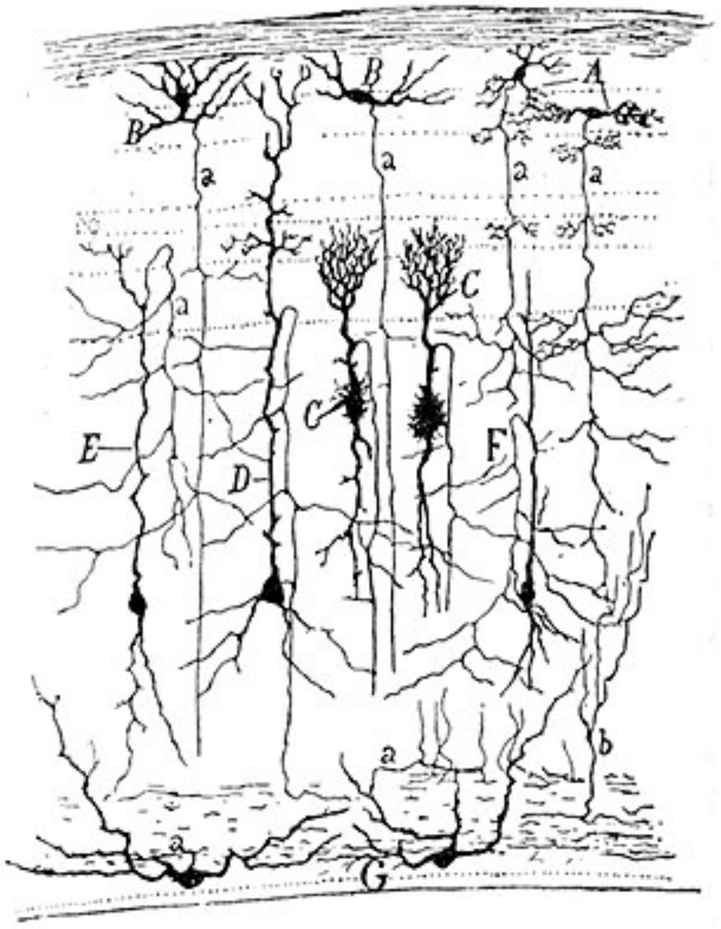
Excitability

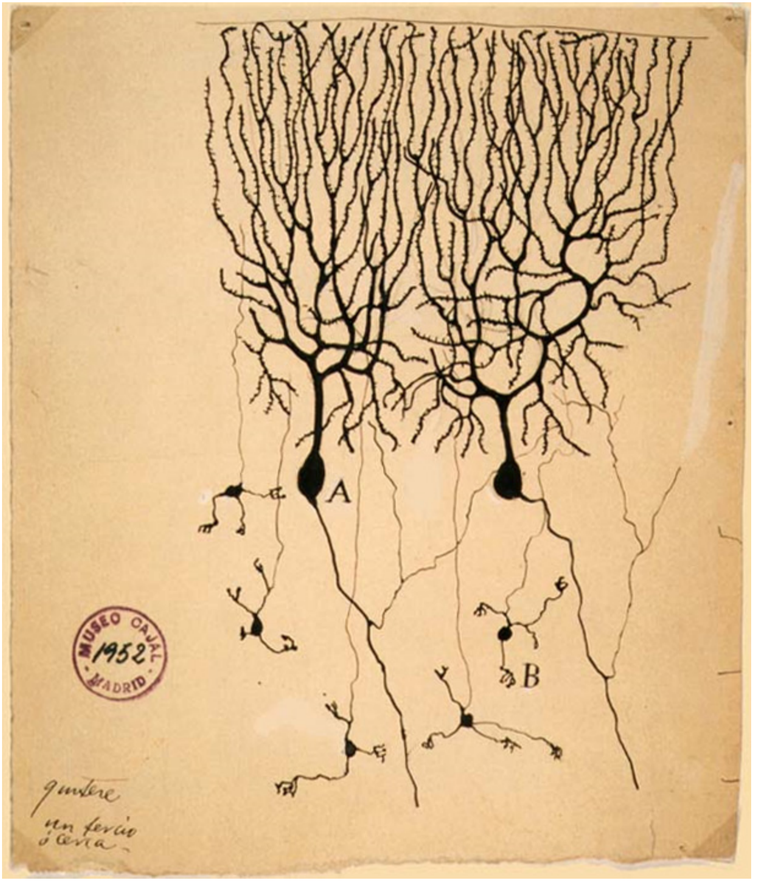
- Neurons are excitable — if the aggregate of the all the chemical flows leads to the intracellular-extracellular ionic balance exceeding some threshold, a rapid electrical pulse (the action potential) is generated
  - → Hodgkin-Huxley model of this remains one of the triumphs of mathematical biology
- Voltage flashes down the axon — the “output channel” of the neuron
- Action potentials trigger neurotransmitter release at synapses which will in turn modulate the ion flows in downstream neurons
- Neurons aren’t the only excitable cells — another big group are muscle cells. Where muscle excitability produces force and movement, neuronal excitability processes information

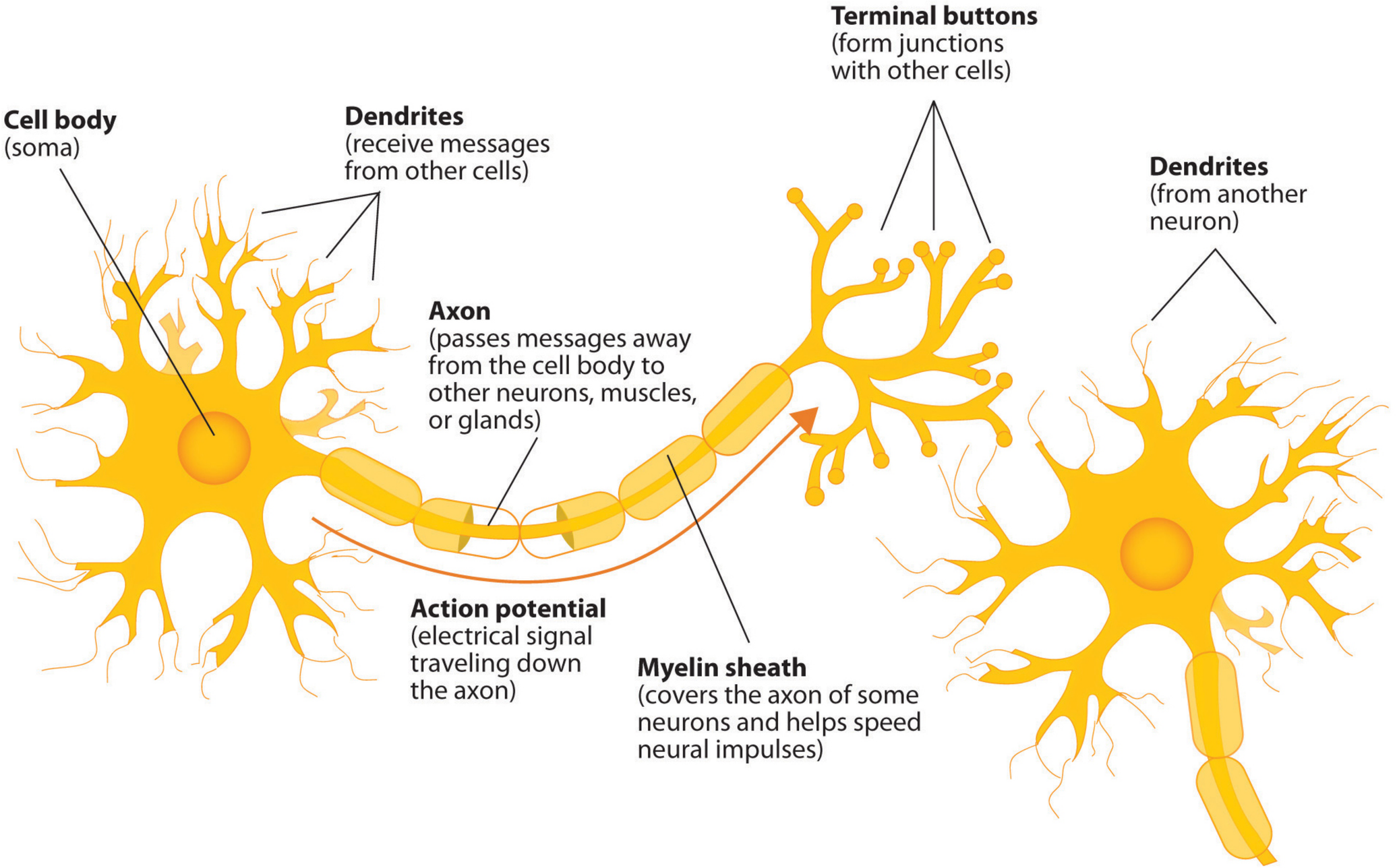


- Much of the underlying functional complexity & processing power of neurons was essentially invisible until very recently — the more we find out about them, the more sophisticated they seem to be
- But at least some aspects of their complexity — notably connectedness — was apparent from late C19 when Cajal and Golgi devised ways of staining them for microscopy and making detailed drawings of what they saw
- Cognitive scientists and AI researchers have often underestimated neurons as very simple processing units — we now think they are closer to CPUs than to simple integrators — but correctly identified connectivity as a defining feature and a great source of power


c
d
a
b

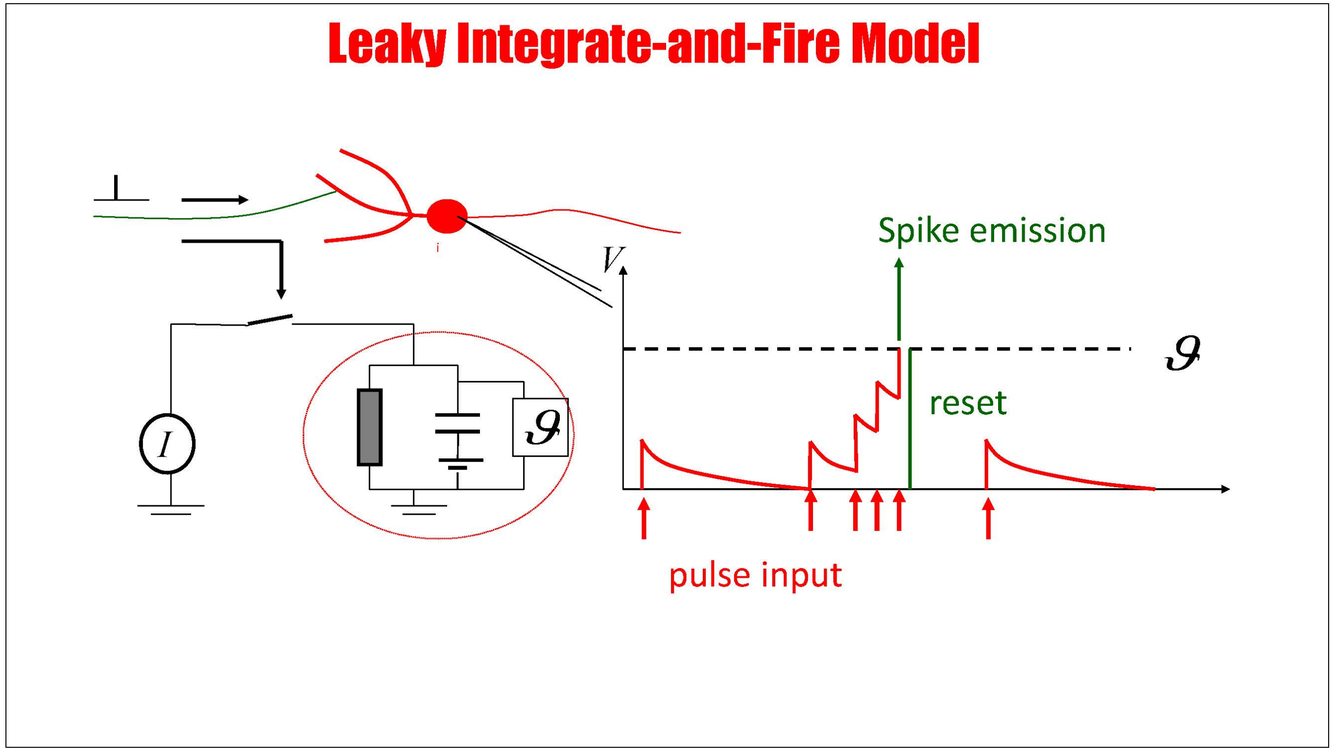

B
B
A
a
a
a
a
a
C
C
E
D
F
a
b
a
G


A
B
MUSEO NAVAL
1952
MADRID


Terminal buttons
(form junctions
with other cells)
Cell body
(soma)
Dendrites
(receive messages
from other cells)
Dendrites
(from another
neuron)
Axon
(passes messages away
from the cell body to
other neurons, muscles,
or glands)
Action potential
(electrical signal
traveling down
the axon)
Myelin sheath
(covers the axon of some
neurons and helps speed
neural impulses)

Integrate and fire

- A common family of models of neuronal behaviour essentially have the cell accumulate inputs until a trigger threshold is reached, causing a spike
- The integrated signal is reflected in the firing rate, which is considered as encoding a numeric output



$$C_m \frac{dV_m(t)}{dt} = I(t) - \frac{V_m(t)}{R_m}$$

Perceptrons as model neurons

- Treating the firing rate as a continuous input value at the synapses, a crude model of the already crude model that is integrate and fire takes a weighted sum of the inputs and applies a threshold to see if the neuron fires:

$$\text{fire} = \begin{cases} 1 & \text{if } \mathbf{x} \cdot \mathbf{w} \geq \text{ threshold} \\ 0 & \text{otherwise} \end{cases}$$

  → weights represent synaptic strength at each of the inputs

- You don't have to be Marvin Minsky to recognise this as just the same old linear classifier model we know and love

All models are wrong

- This linear model is extremely remote from the electrobiochemical realities of neural computation
- The analogy between the integrative behaviour of real neurons and the “weighted sum + non-linearity” of perceptrons etc is loose
- But a lot of the brain’s processing and learning does seem to be done at synapses, lending some credence to the connectionist model that brains get their power from connectivity
- Basically: artificial neural networks bear only a metaphorical resemblance to biological brains
- … but it’s a metaphor that has proved pretty fruitful

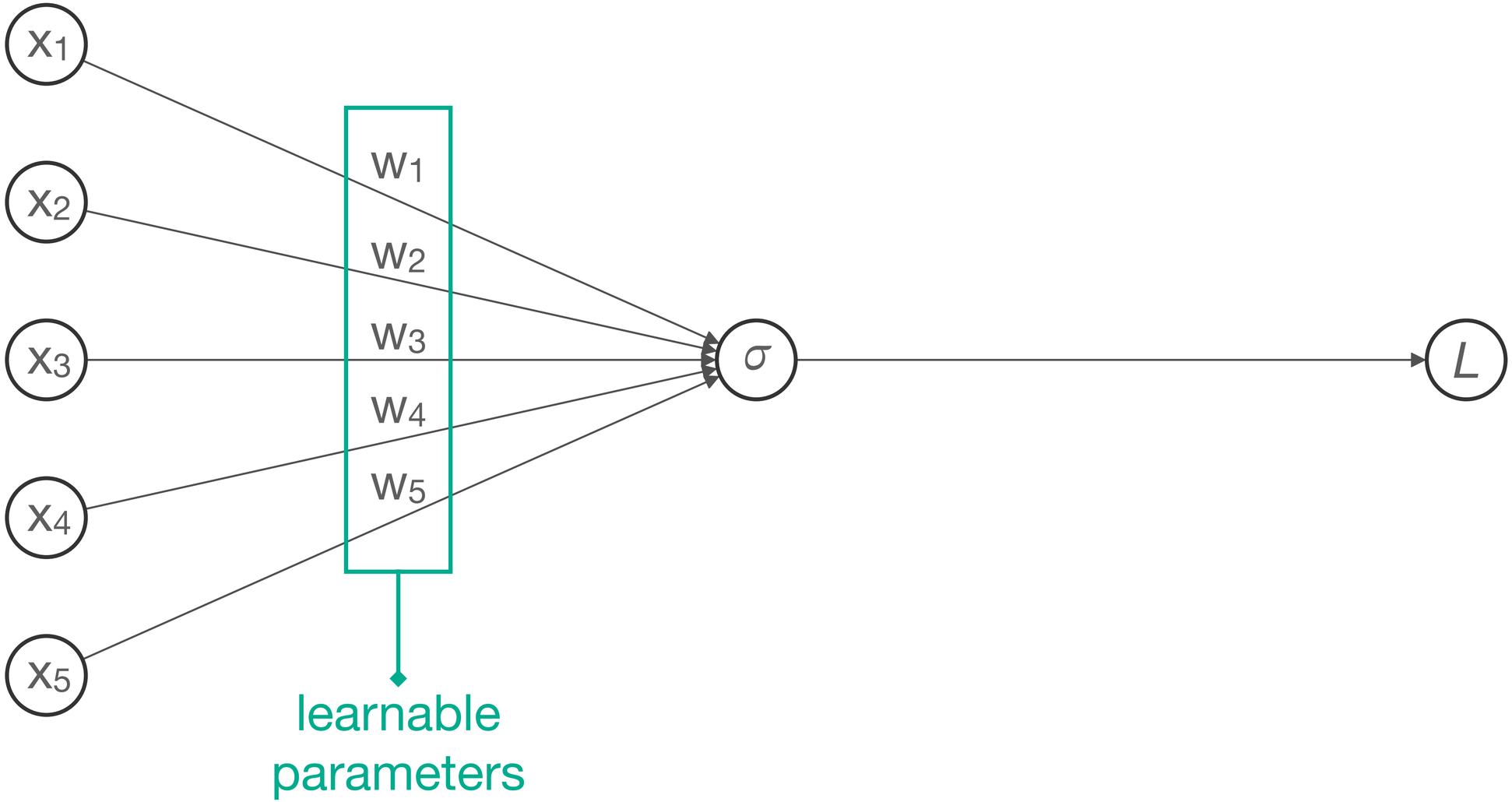
5.2: Multi-Layer Perceptrons

COMP0088 Introduction to Machine Learning • UCL Computer Science

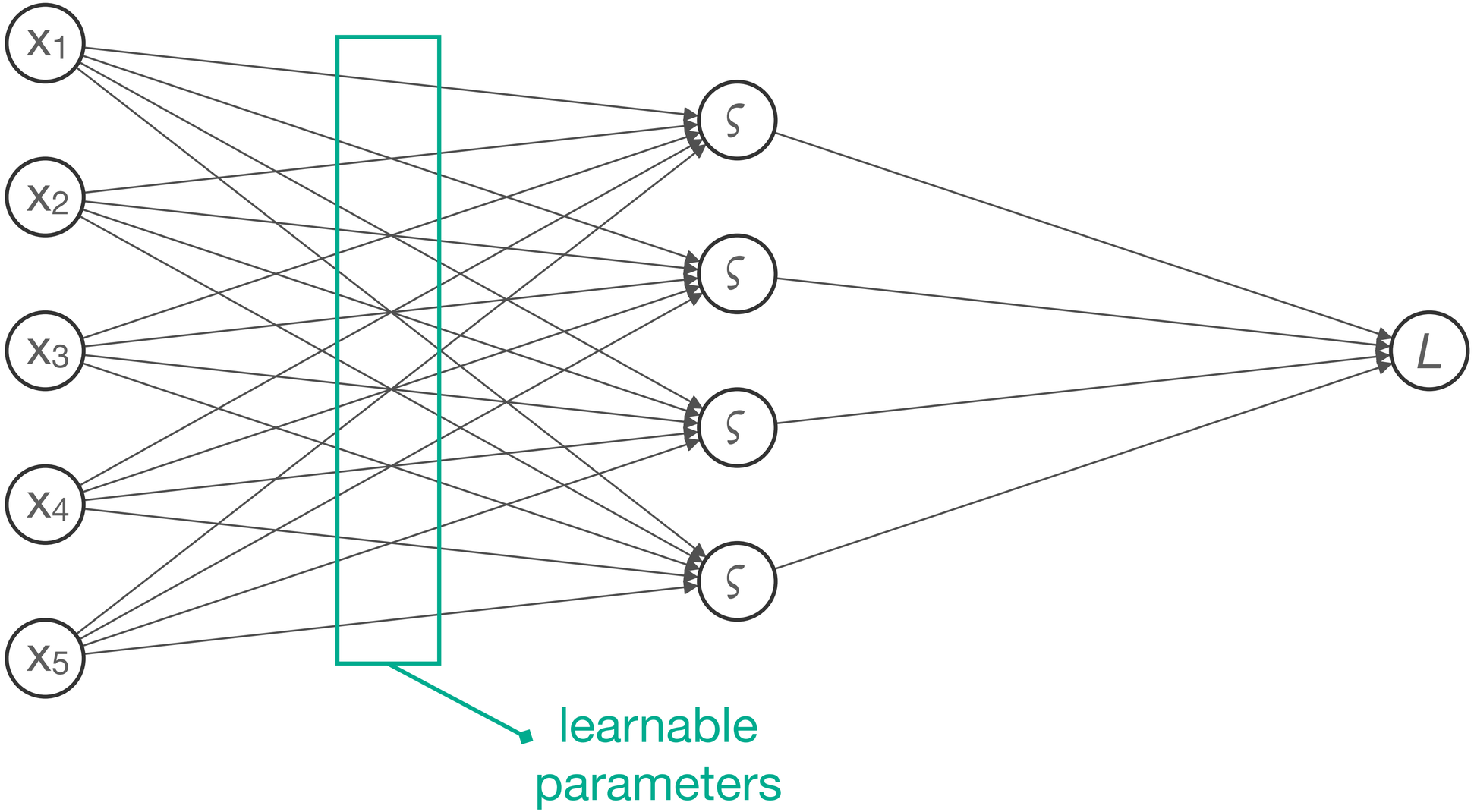
Limitations of the lone perceptron

- Single perceptrons and equivalent linear models fit only a linear boundary
- Incapable of learning more complex boundaries in the original feature space
- This is a shortcoming of the model, not the learning algorithm
- Eg, perceptron algorithm has other issues too, but this one is shared

Logistic regression as a neural net

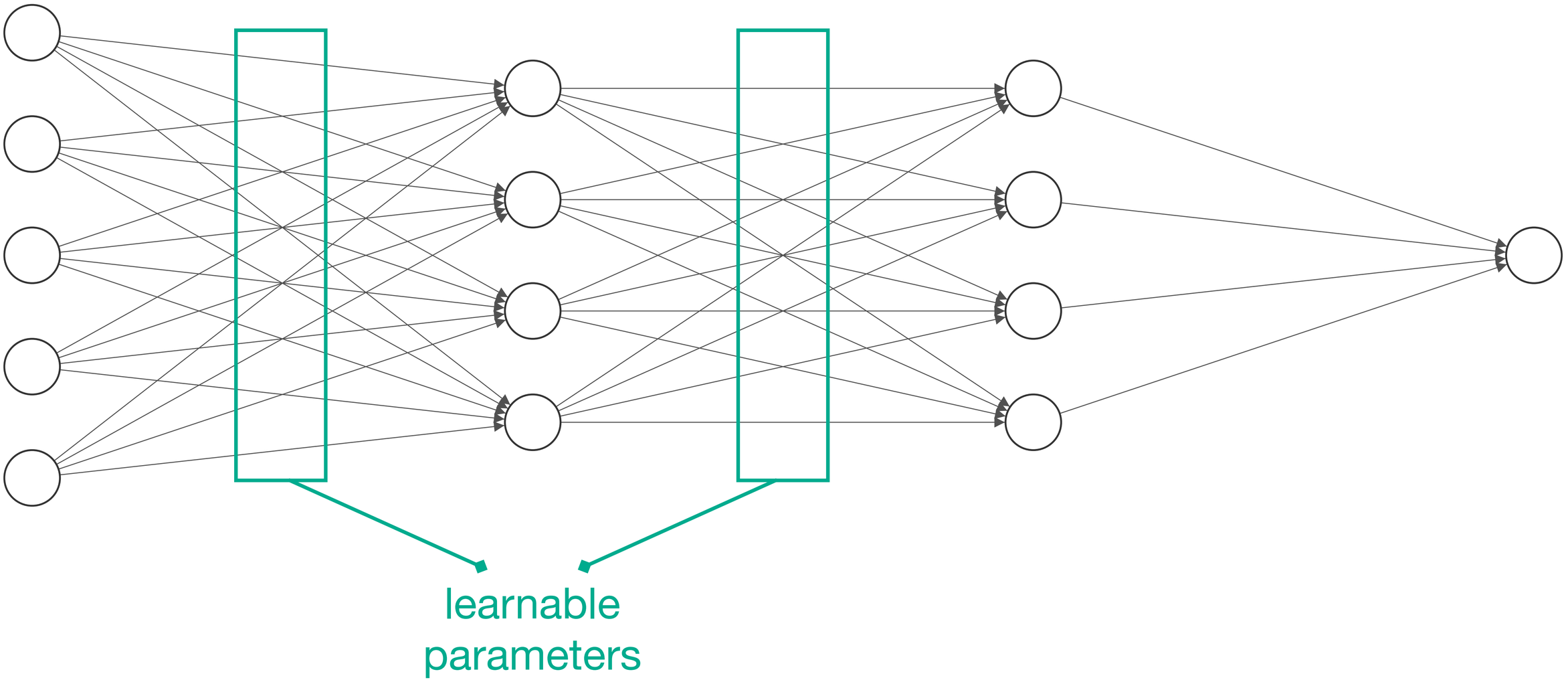


Softmax regression as a neural net



Hidden layers

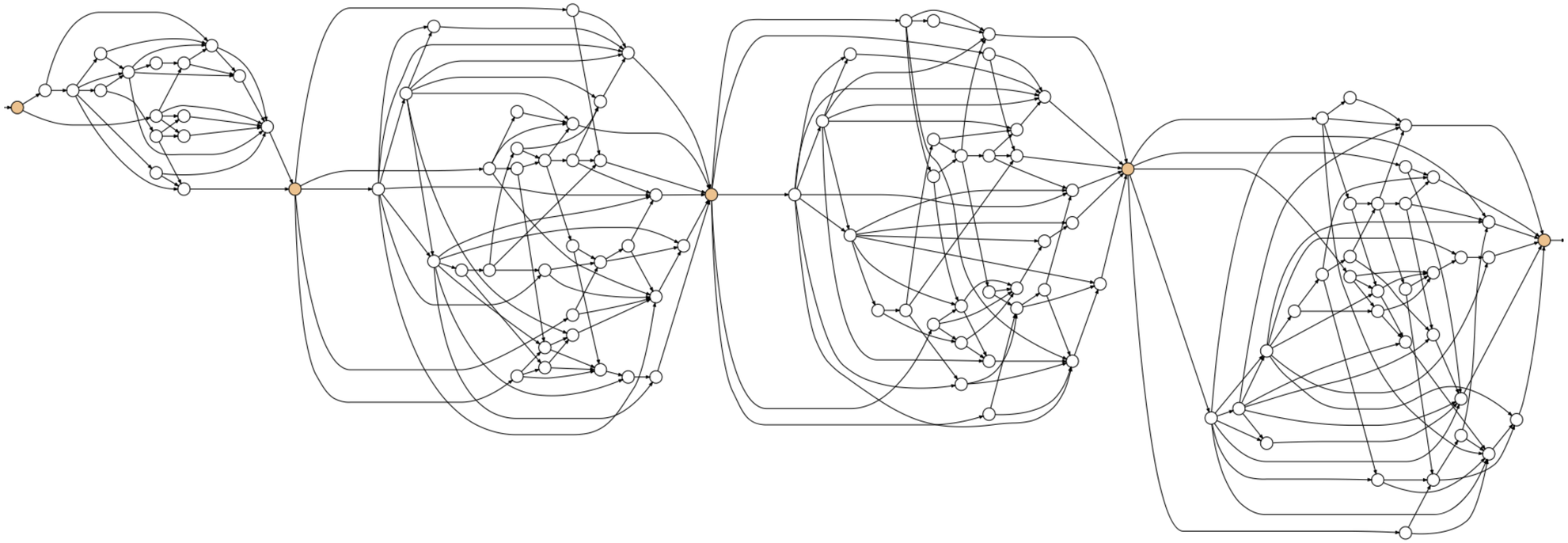
- Single linear decision boundaries constitute inferred “features” that can be aggregated — as we saw in AdaBoost, where a composite score was built as a weighted sum of individual decisions
- Notably, the AdaBoost boundary is not linear, even though the constituent decision boundaries are (at least when using decision stumps as learners)
- What if we add an intermediate “hidden” layer of neurons that take as input the previous layer’s outputs?
- The multi-layer perceptron is born!


learnable
parameters

Layer cake

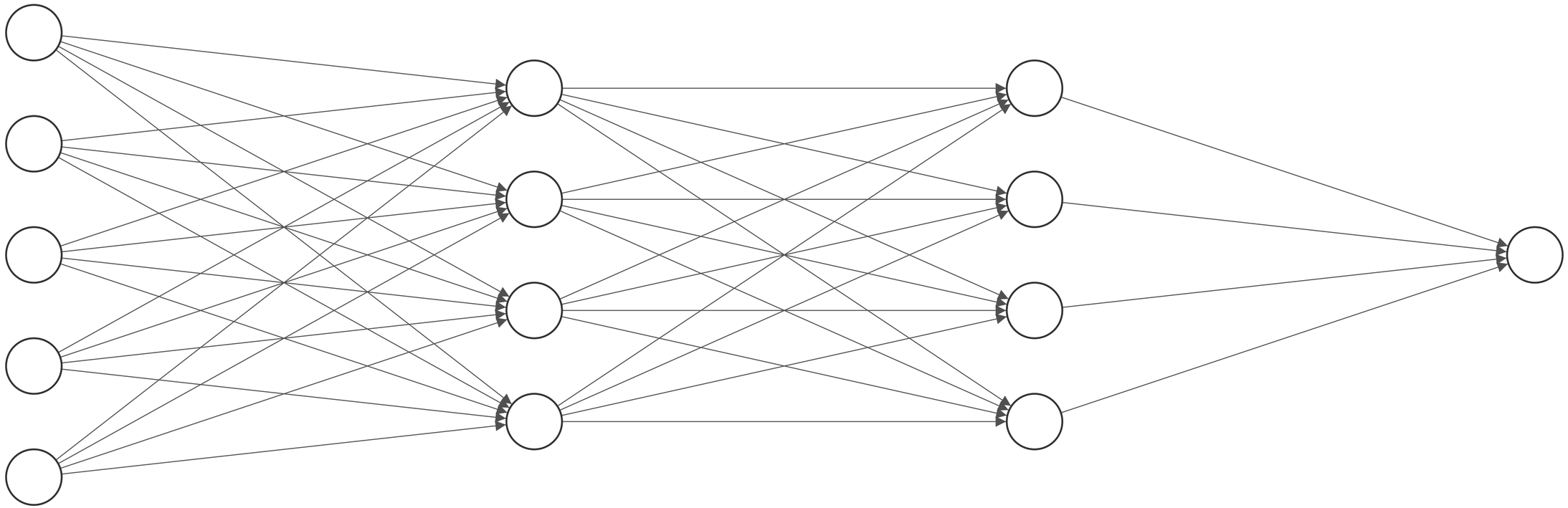
- In principle there's no reason why neural nets can't be wired up in any kind of crazy configuration

  → Indeed, there's some evidence that doing so can give rise to efficient networks



Layer cake

- However, in practice we almost always structure networks in layers



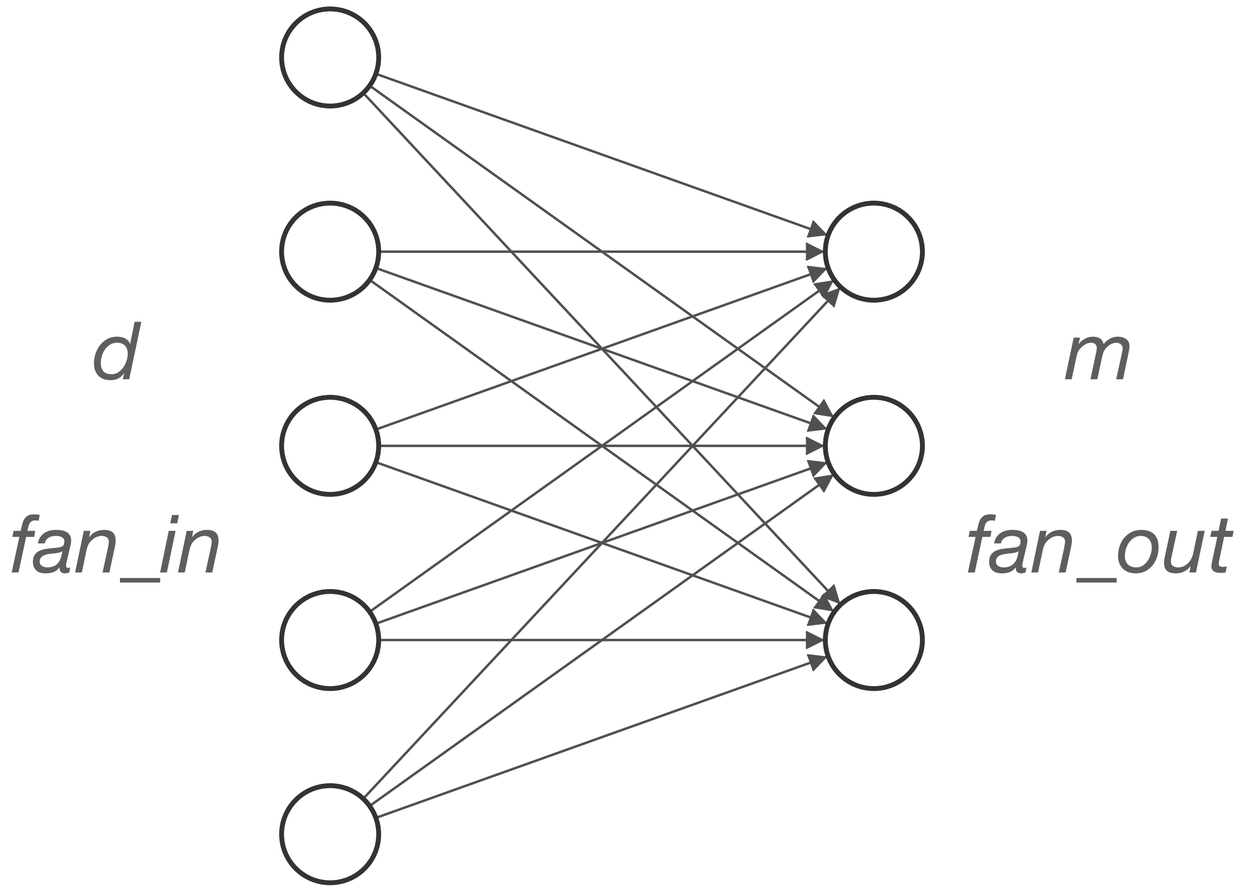
input layer hidden layer 1 hidden layer 2 output layer

Layer cake

- Layers are a useful organising principle, dividing the processing up into functional chunks that are easier to implement and reason about
- What constitutes a layer is somewhat up for grabs, but we often define a layer as “a thing that has weights”
- Input and output layers are special cases that are often but not always distinguished as separate entities
- Layers are also computationally useful, allowing for efficient data structures and implementation in terms of linear algebra

Fully connected layers

- All inputs are connected to all outputs, each with an associated connection weight
- Maps input dimension *d* to output dimension *m*
- Input size is sometimes called *fan_in* and output *fan_out*
- There are *fan_in* × *fan_out* or *dm* weights in the layer



Matrix representation of layers

- A fully connected layer with *d* inputs and *m* outputs can be conveniently represented as a *d*×*m* matrix of weights
  - → Rows correspond to the input features
  - → Columns to the output features
  - → An input row vector of *d* features becomes an output row vector of *m* combinations

$$\mathbf{z} = \mathbf{x}\mathbf{W}$$

  - → An input design matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ becomes an output design matrix $\mathbf{Z} \in \mathbb{R}^{n \times m}$

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

The importance of being non-linear

- But wait: a weighted sum of weighted sums is still just a weighted sum
- In matrix terms:

$$\begin{aligned}\mathbf{x}\mathbf{W}_1\mathbf{W}_2...\mathbf{W}_k &= \mathbf{x}(\mathbf{W}_1\mathbf{W}_2...\mathbf{W}_k) \\ &= \mathbf{x}\mathbf{W} \\ &\text{where } \mathbf{W} = \mathbf{W}_1\mathbf{W}_2...\mathbf{W}_k\end{aligned}$$

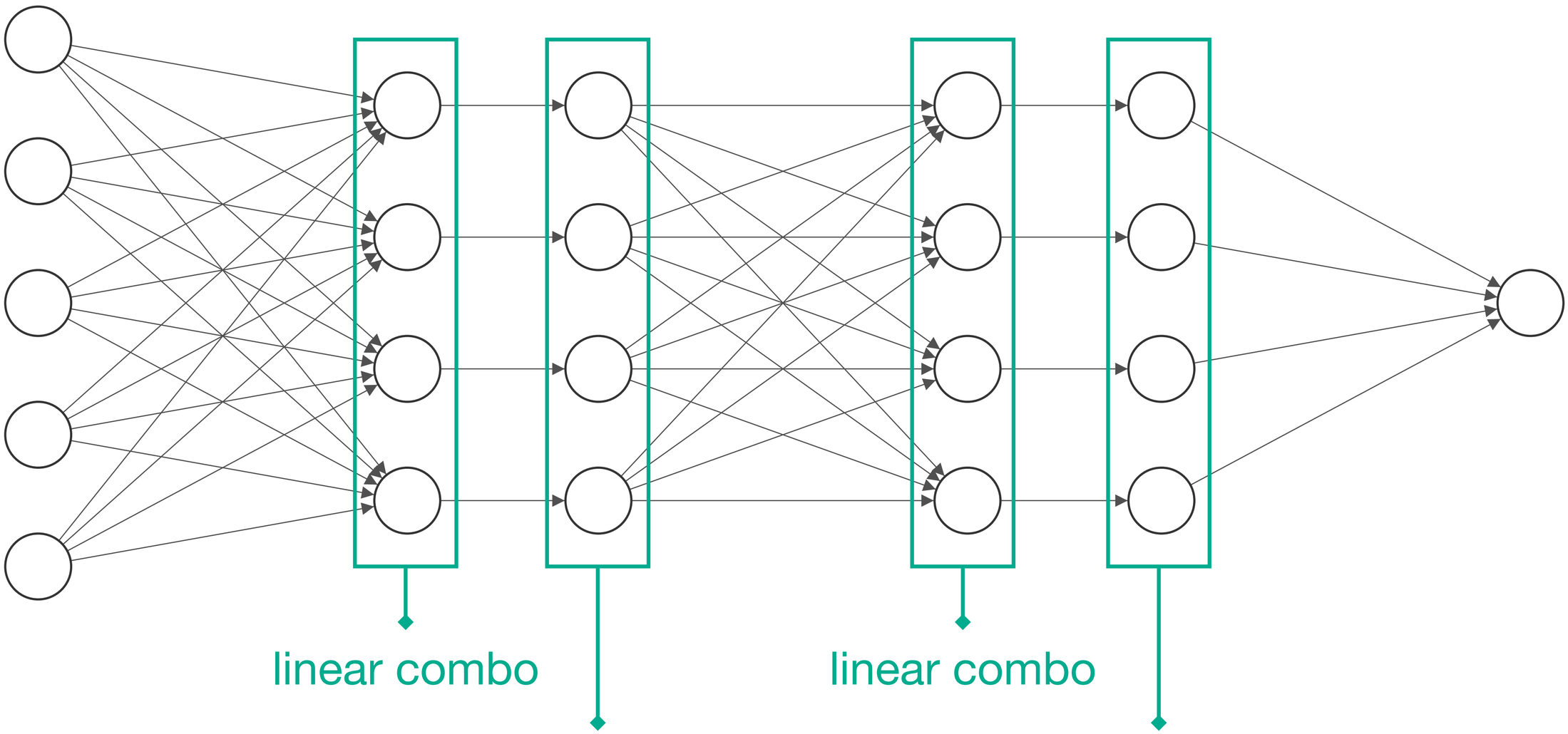
The importance of being non-linear

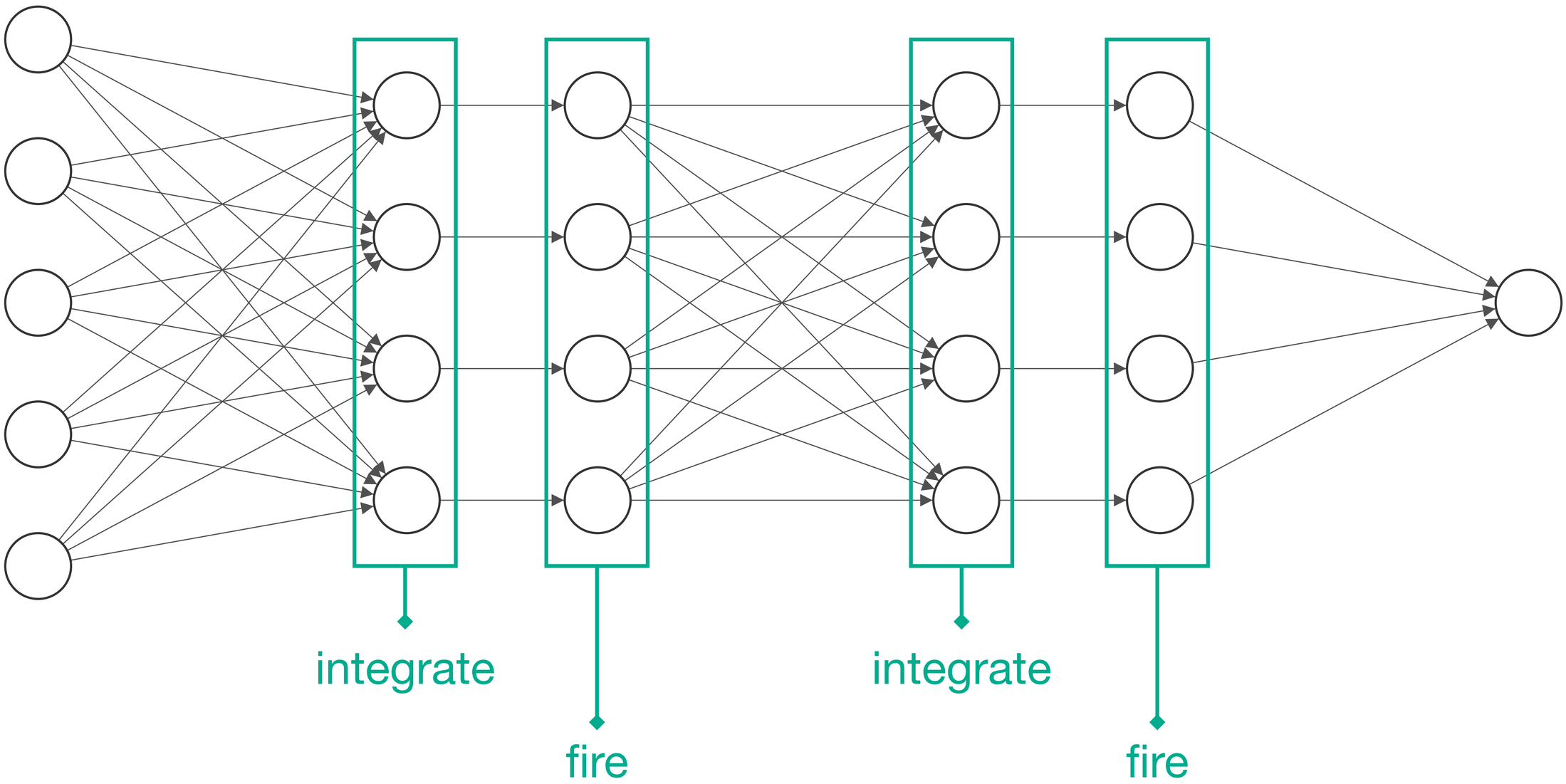
- Why does this not apply to AdaBoost?
- Ta-da! Non-linearities to the rescue

$$\hat{\mathbf{y}} = a_k(...a_2(a_1(\mathbf{x}\mathbf{W}_1)\mathbf{W}_2)...\mathbf{W}_k)$$

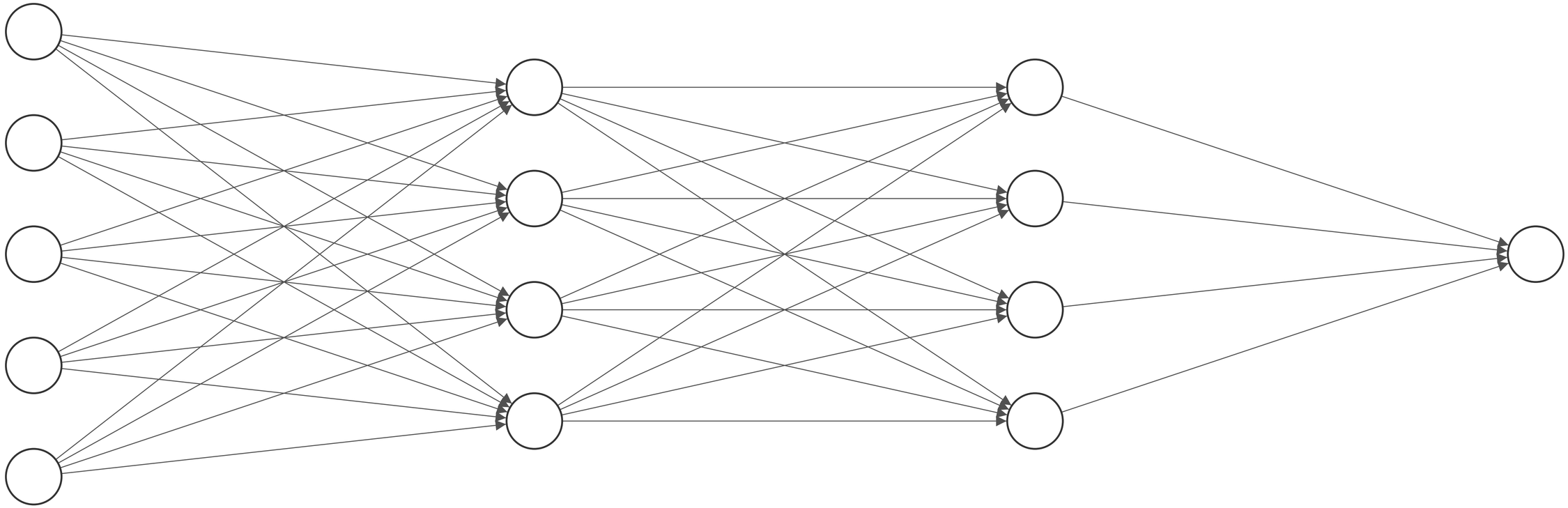
- The a_i are known as “activation functions” (see §5.4)

- Cf. biological neural nets viewed as “integrate and fire”
  - → Again, this is only a loose analogy
  - → Biological neurons are non-linear all the way down!


linear combo
non-linear activation
linear combo
non-linear activation


integrate
fire
integrate
fire

- In practice, we usually just merge activations into the layers for more compact representation

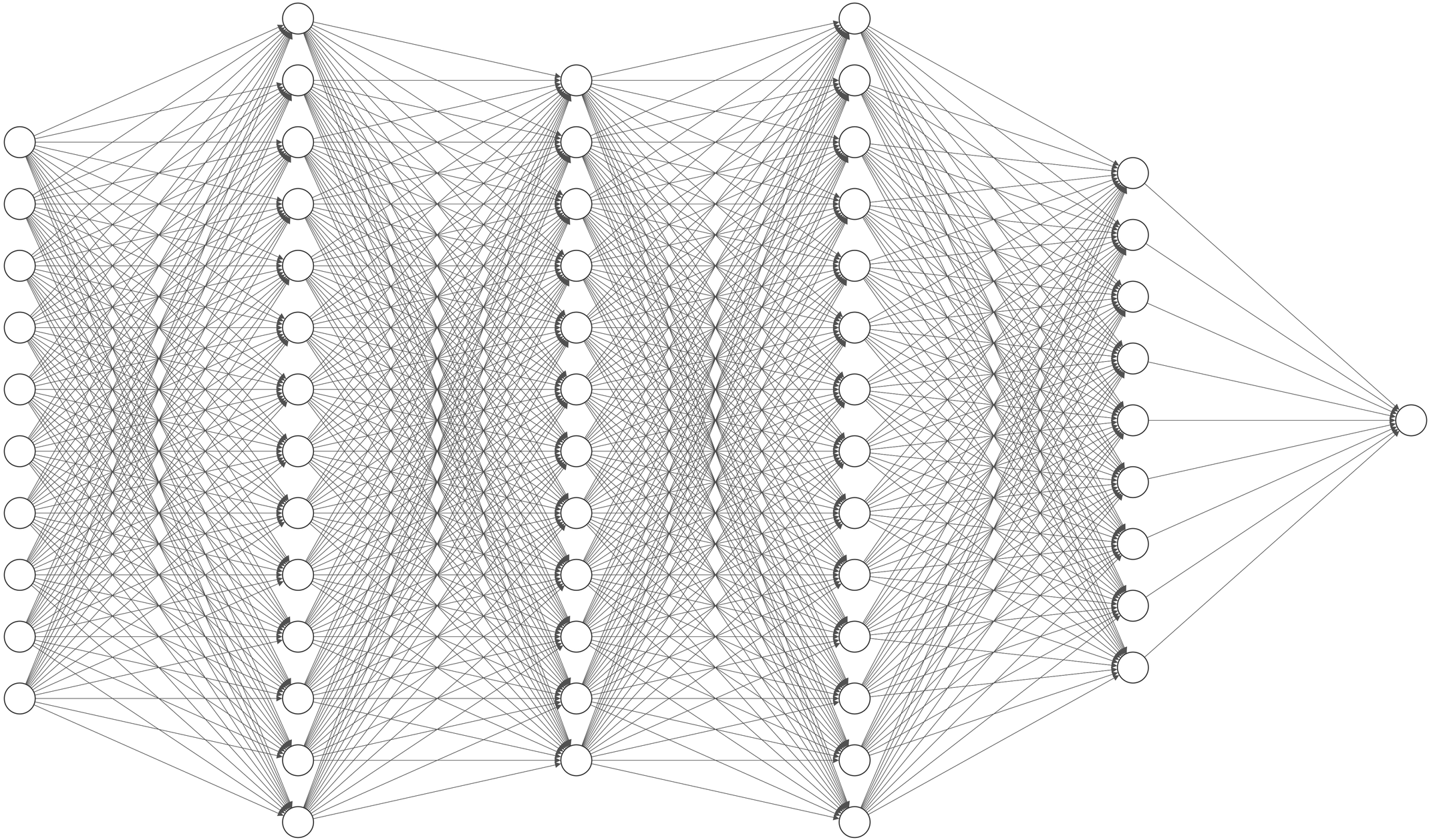


Layers as aggregators

- It’s useful to think about what a hidden layer is doing
- Each neuron is (roughly) finding a distinct linear boundary (or regression manifold) in the space generated by the previous layer
  - → Glossing over the effect of activation for the moment
- First hidden layer captures useful hyperplanes in the raw feature space
- Next layer finds useful combinations of those first hyperplanes, and so on
- This allows the network to build up increasingly complex classification boundaries or regression landscapes

Wide and deep

- We can make bigger networks — with richer processing capacity — by having more neurons in a layer (width) and also by having more layers (depth)
  - → Wide layers generate a bigger range of candidate feature
  - → Deep layers produce more complex nested aggregates
- We could really go to town on this — and we will!
- But remember how painful it was deriving the logistic regression gradient?
- How on Earth are we supposed to fit a model like this?

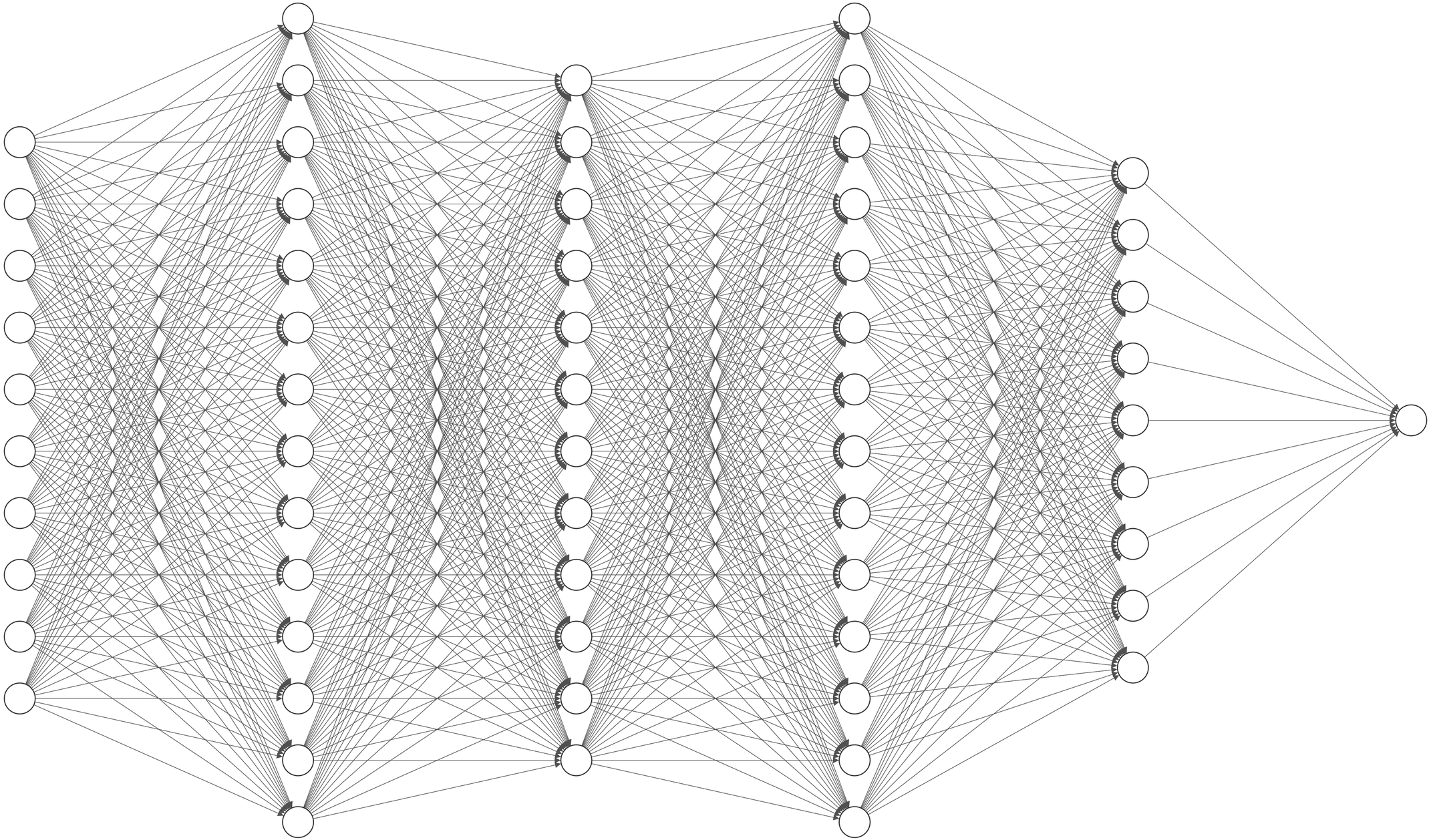


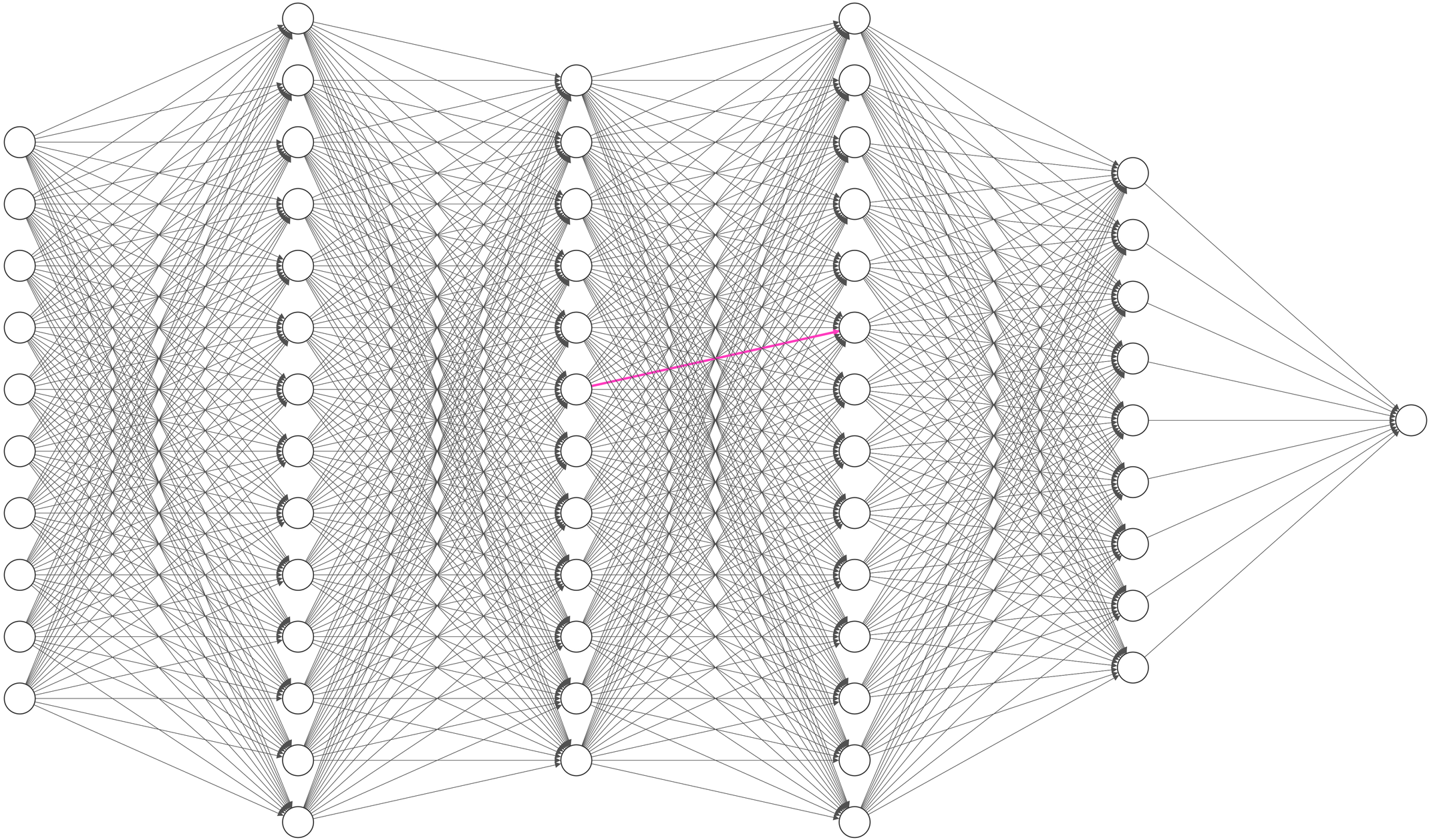
5.3: Backpropagation

COMP0088 Introduction to Machine Learning · UCL Computer Science

Problem statement

- We have a neural network of some complexity
  - → It doesn't have to be huge, but it's a bit more than a simple logistic regression, with a bunch of weights, hidden layers and non-linear activation functions
- How do we train it?
- Gradient descent worked for logistic regression, lasso, linear SVMs
- But those things have (relatively) simple (sub)gradients
- For a big compound function like a neural network, how are we supposed to figure out what effect some random weight parameter buried in the middle is having on our loss right at the end?





Numerical gradients

- Recalling limit definition of the derivative

$$\frac{\partial f}{\partial x} = \lim_{\delta \to 0} \frac{f(x+\delta, \ldots) - f(x, \ldots)}{\delta}$$

- We can approximate numerically at some concrete small δ

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\delta, \ldots) - f(x, \ldots)}{\delta}$$

Numerical gradients

- Easy to understand
- Easy to implement
- Almost guaranteed to work
- Places few constraints on model or function, other than being vaguely coherent — must be defined at x and $x+\delta$
- But: extremely slow, since you have to twiddle each parameter individually and then for each one run the whole model to check change in output

Analytic gradients

- In previous cases we have derived gradients analytically
- Where feasible this is smart, principled and efficient
- But fiddly at the best of times and completely intractable at worst
- Also very very easy to get wrong

- And yet: as mentioned way back in week 2, the chain rule is actually very straightforward and mechanical
- Surely there's some way to avoid — or at least automate — that algebra?

Unpicking the chain rule

- Just boils down to: gradients multiply
- There are several ways of notating this (not always clarifying)
- Leibniz notation

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

- Function composition

$$(f \circ g)' = (f' \circ g) \times g'$$

Where is each derivative evaluated?

- Output of one function is input to the next
- So derivatives are taken at those points too

$$(f \circ g)'(x) = f'(g(x)) \times g'(x)$$

$$\Big[f(g(x))\Big]' = f'(g(x)) \times g'(x)$$

Sequences of functions

- Any number of function calls can be chained this way

$$\Big[f(g(h(x)))\Big]' = f'(g(h(x))) \times g'(h(x)) \times h'(x)$$

$$\frac{df_1}{dx} = \frac{df_1}{df_2}\frac{df_2}{df_3}\cdots\frac{df_n}{dx}$$

Sequences of functions

- Computationally, we might think of making nested function calls

```
d = h(g(f(a)))
```

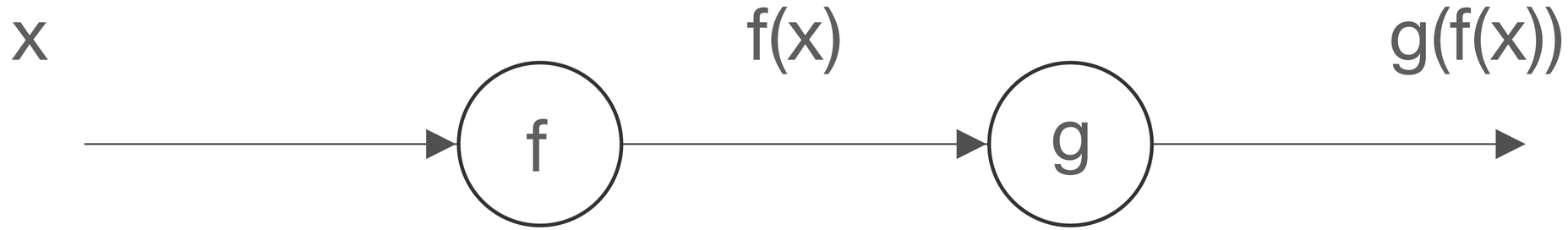
- Or, broken down into steps

```
b = f(a)
c = g(b)
d = h(c)
```

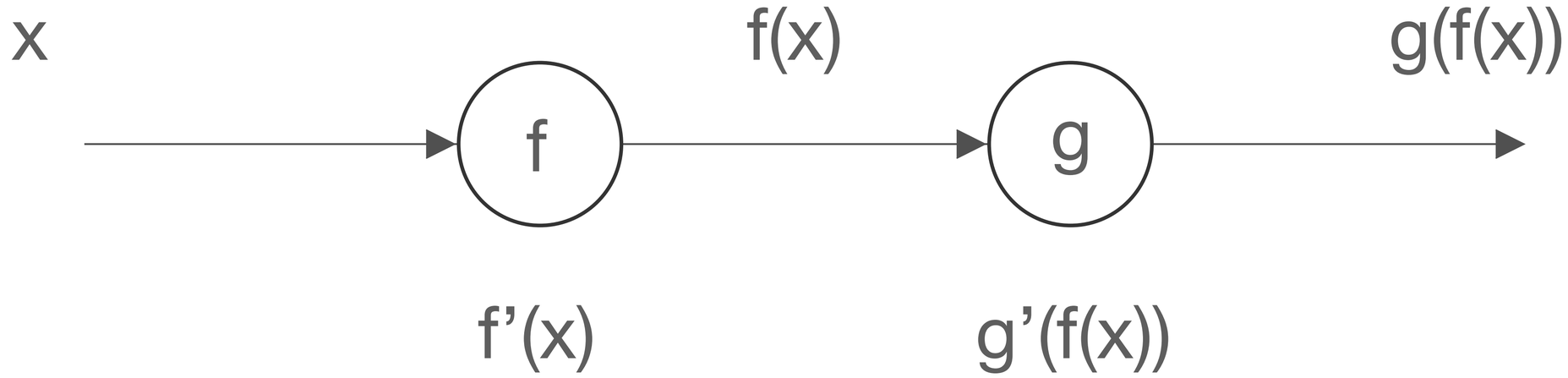
- Output of each function becomes input of the next

Represent operation sequence as a graph

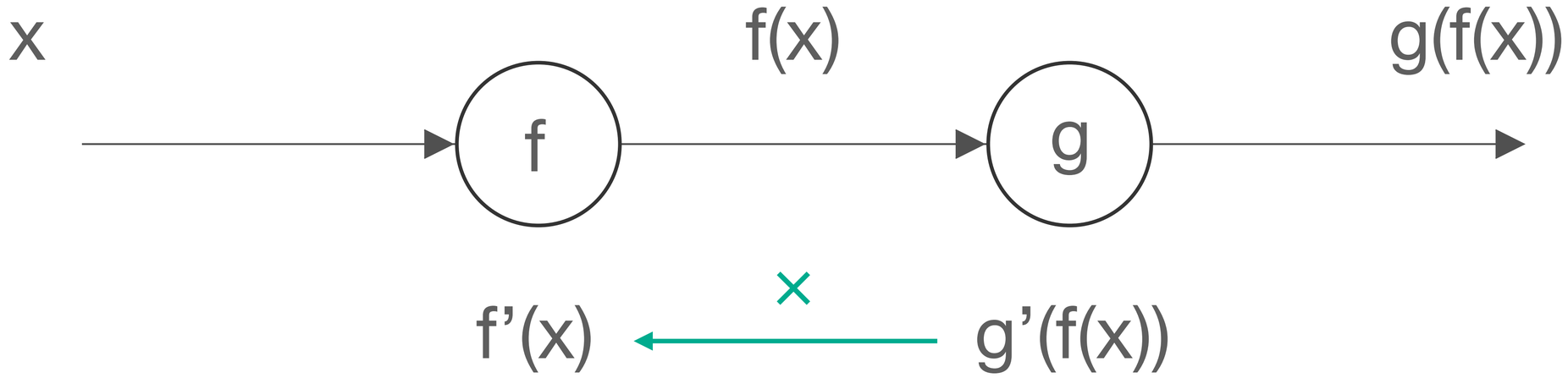
Data flows through the graph



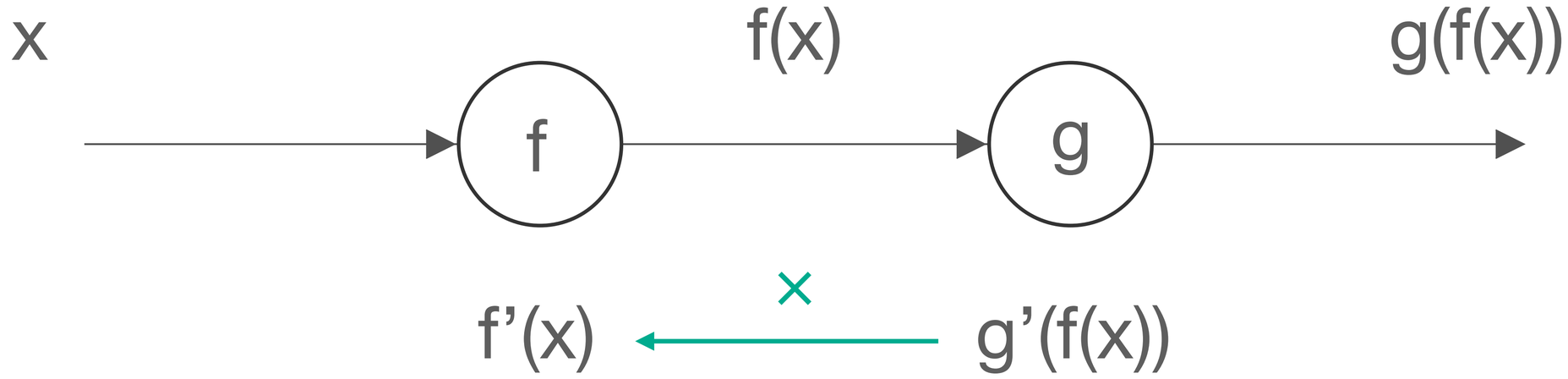
Each operation has a gradient



Gradient also flows through graph

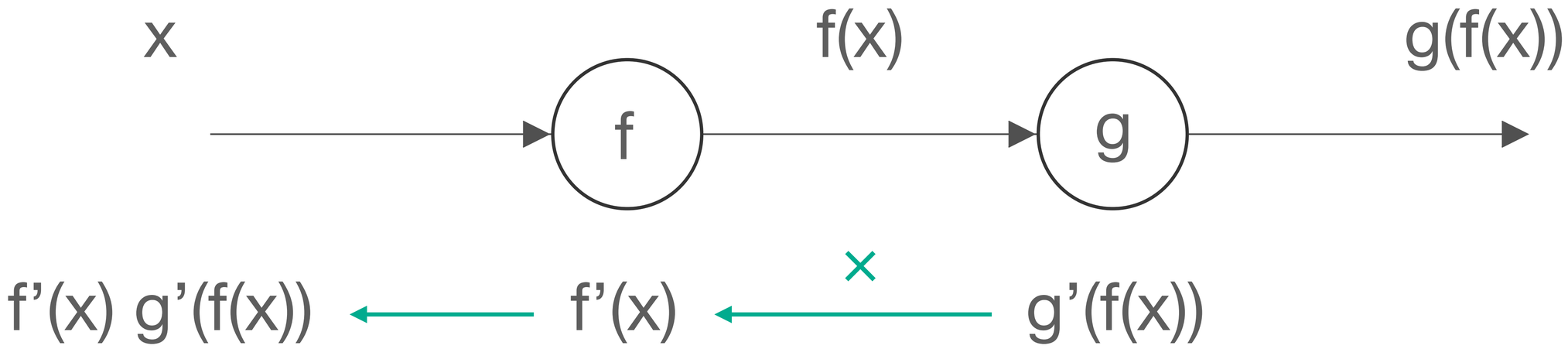


Gradient also flows through graph

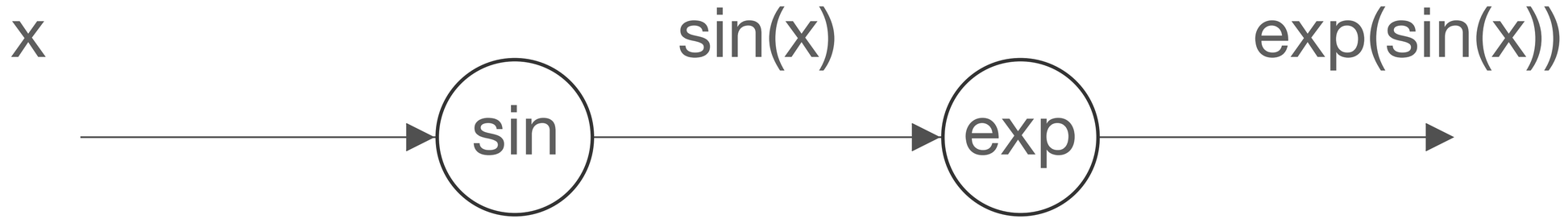


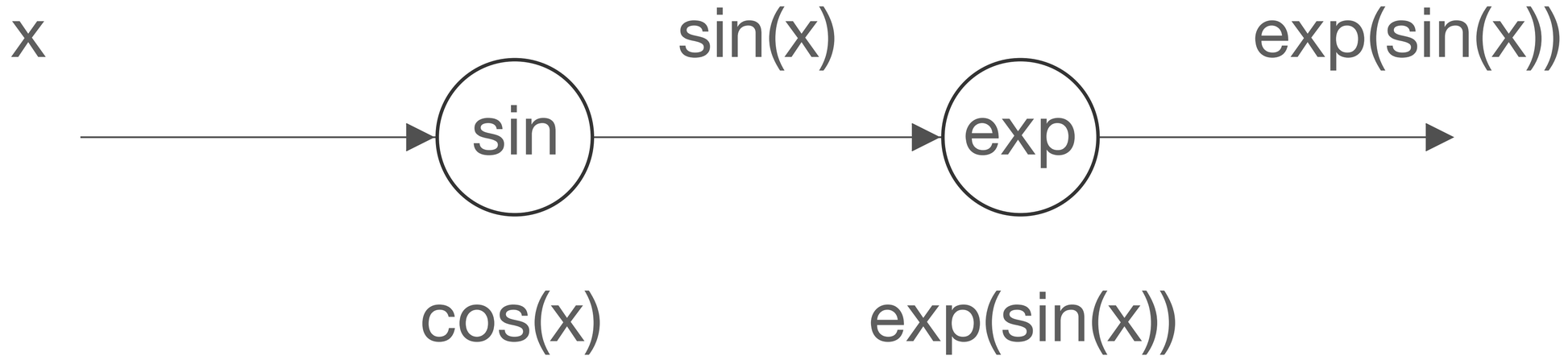
Multiplication is commutative, so this could go in either direction, but it will be convenient shortly to consider the gradient as flowing backwards

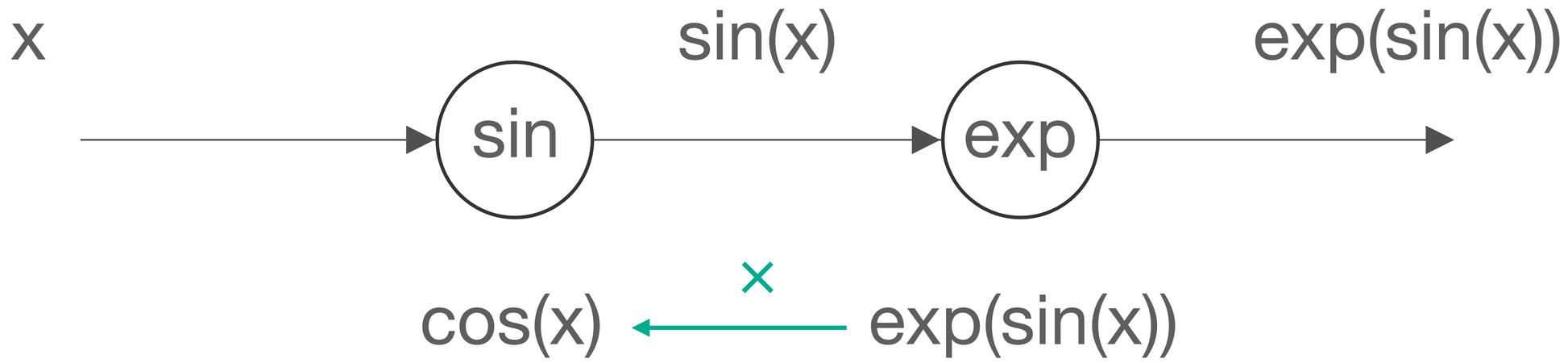
Overall gradient is product of stages

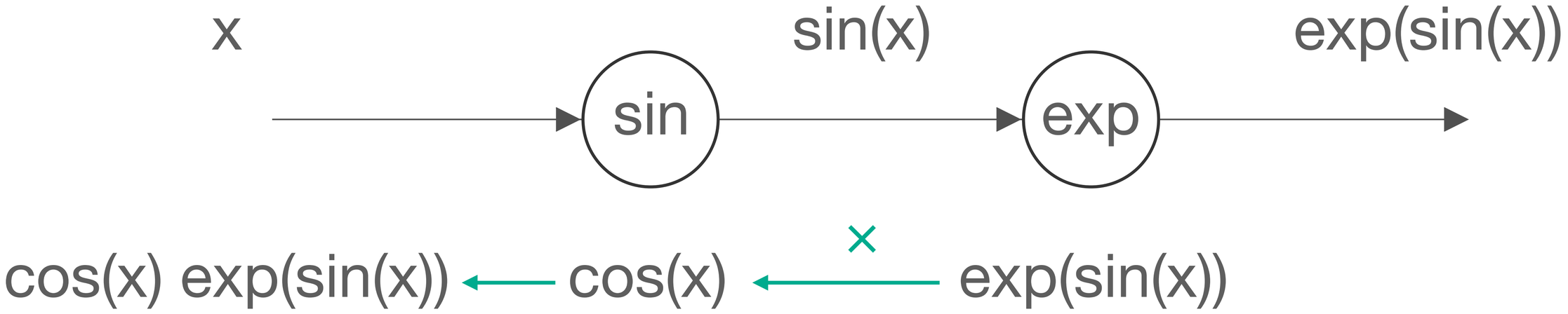


Simple operation graph examples

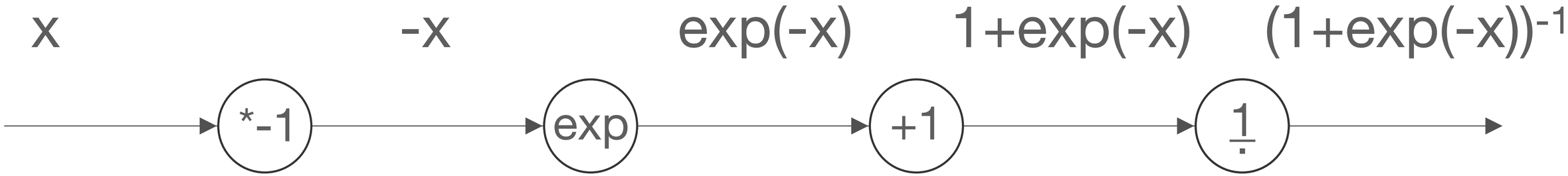

x
sin(x)
exp(sin(x))
sin
exp

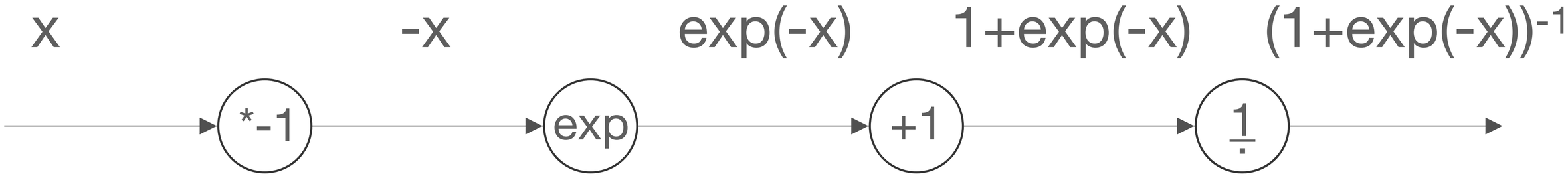

x
sin(x)
exp(sin(x))
sin
exp
cos(x)
exp(sin(x))


x
sin(x)
exp(sin(x))
sin
exp
×
cos(x)
exp(sin(x))

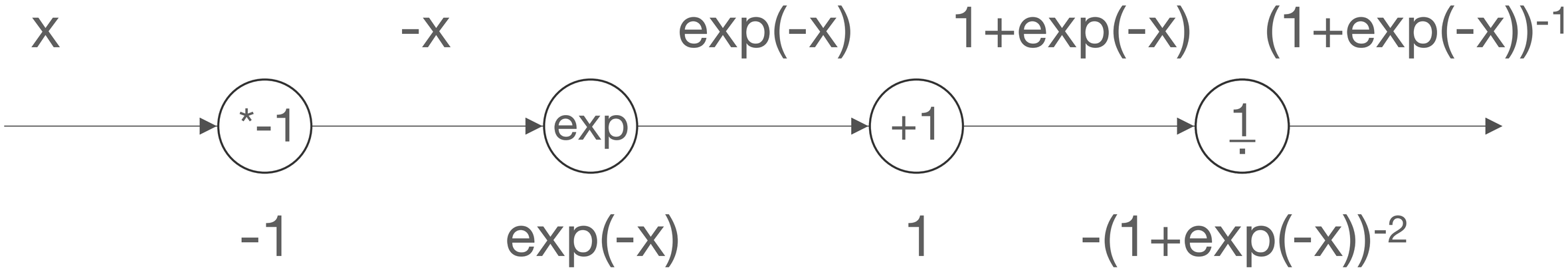

x
sin(x)
exp(sin(x))
sin
exp
cos(x) exp(sin(x))
cos(x)
×
exp(sin(x))

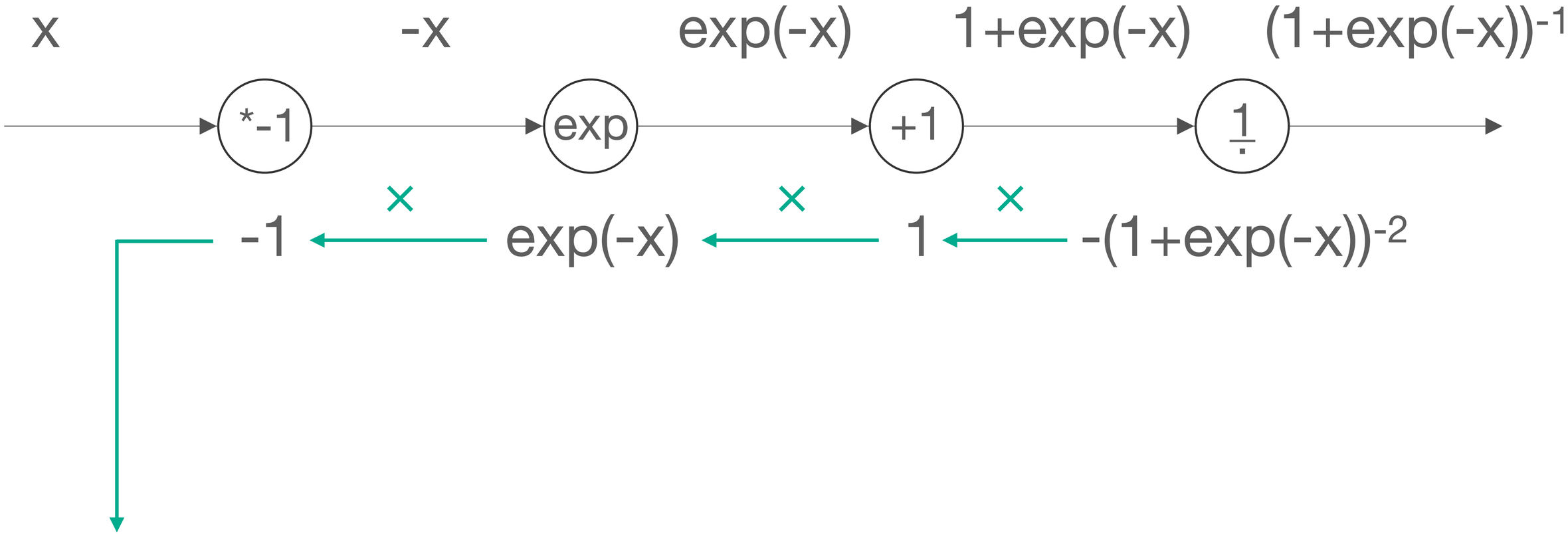
Sequence of multiple nodes


x
-x
exp(-x)
1+exp(-x)
(1+exp(-x))-1
*-1
exp
+1
1



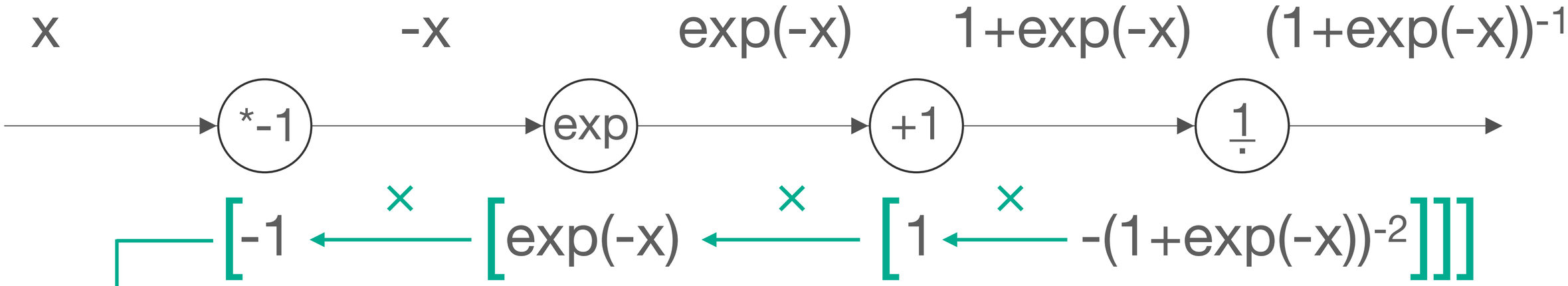
This is actually the sigmoid (logistic) function we used in week 2


x
-x
exp(-x)
1+exp(-x)
(1+exp(-x))^-1
*-1
exp
+1
1
-1
exp(-x)
1
-(1+exp(-x))^-2



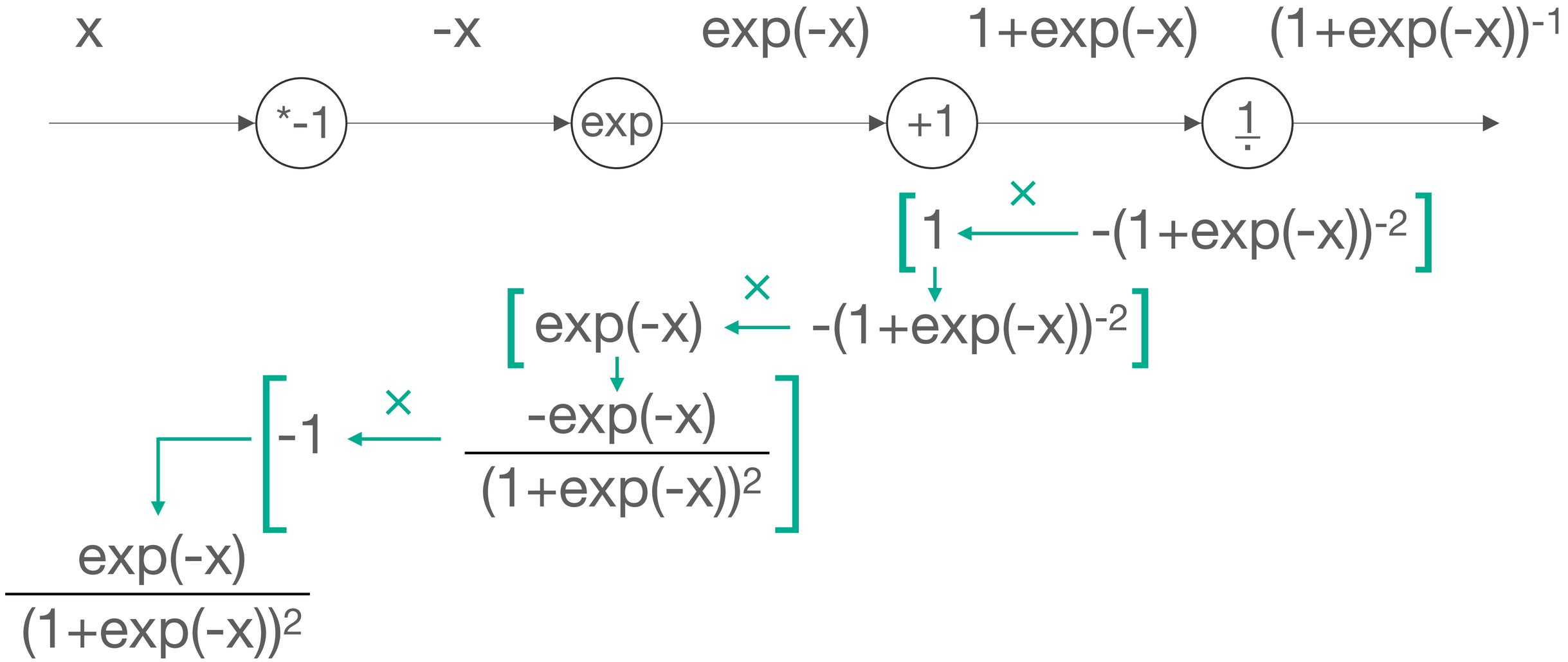
$$\frac{\exp(-x)}{(1+\exp(-x))^2}$$

We previously simplified this further, but here we don't care about neat algebra, we just want an answer

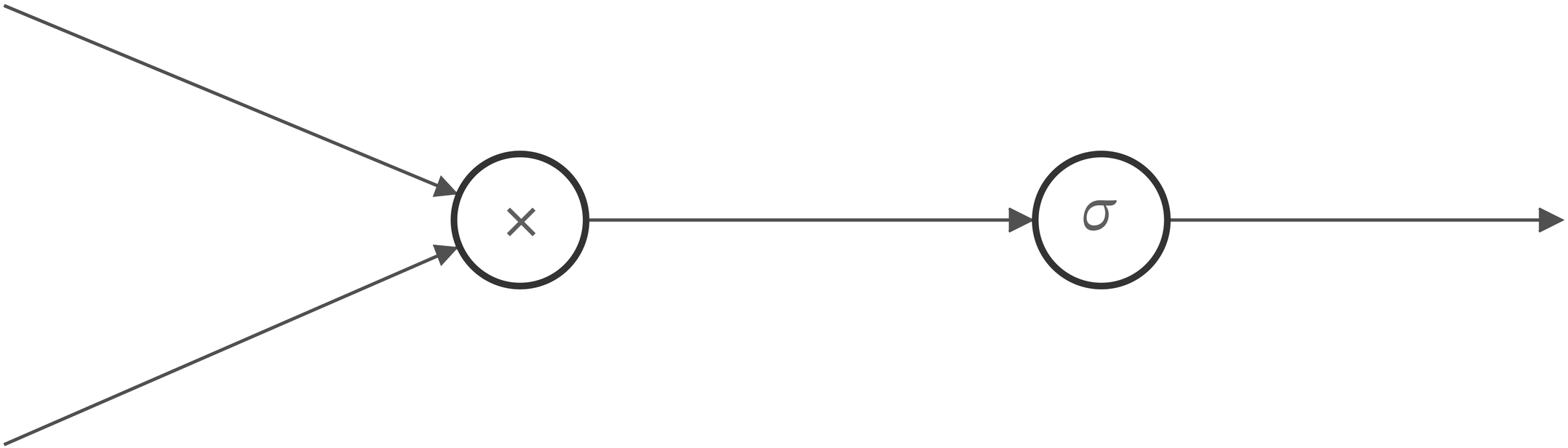


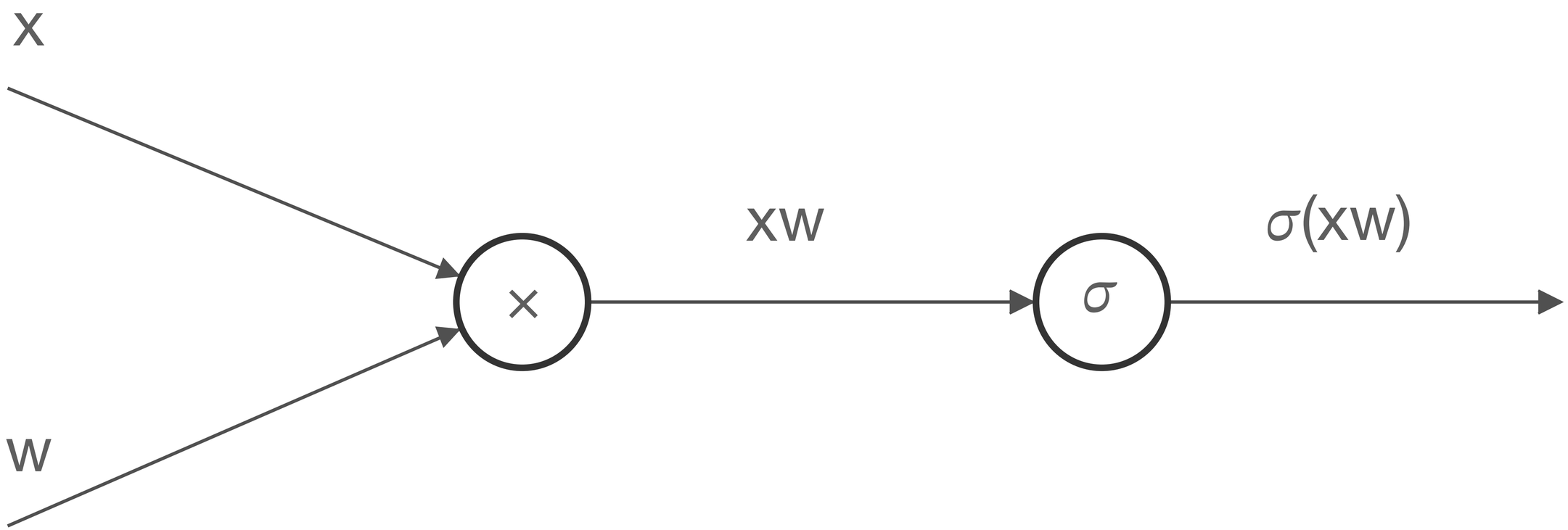
Multiplication is associative, so we can bracket this whole expression as a chain of pairwise products

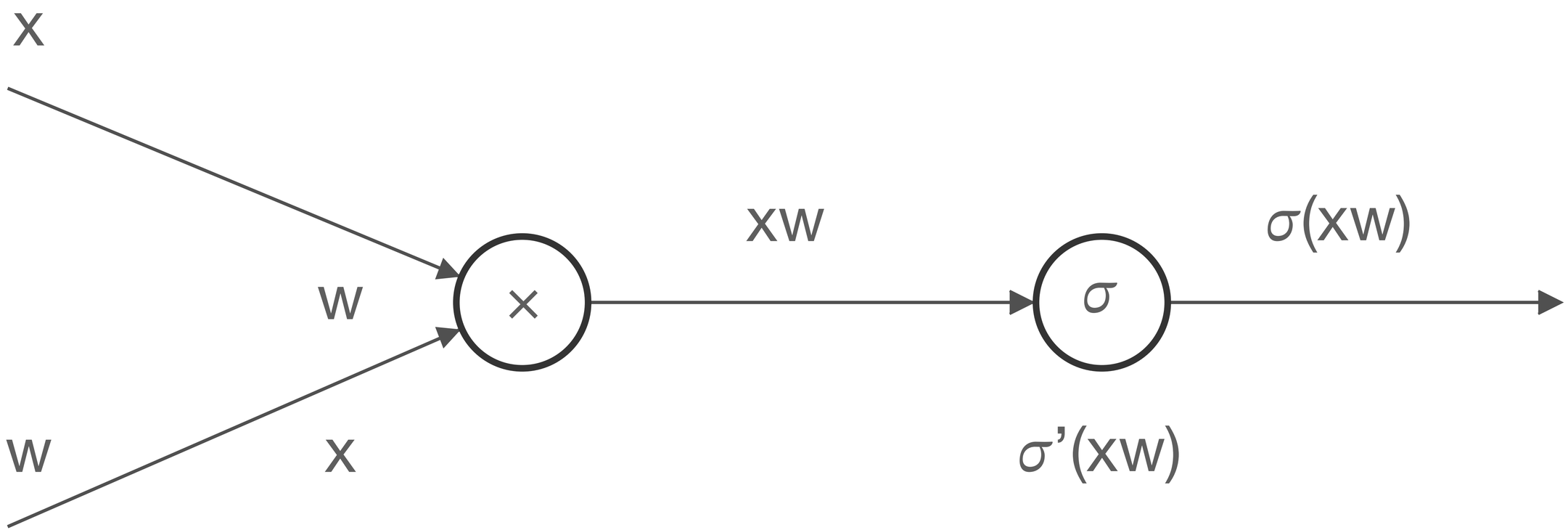
$$\frac{\exp(-x)}{(1+\exp(-x))^2}$$

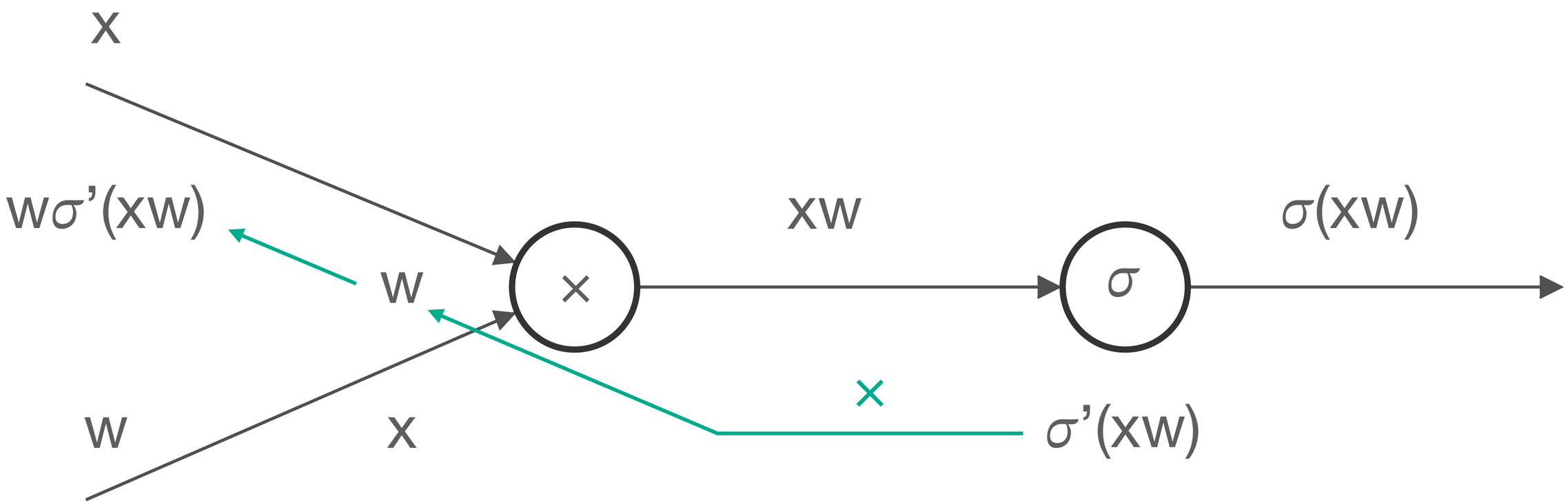

x
-x
exp(-x)
1+exp(-x)
(1+exp(-x))^-1
*-1
exp
+1
1/·
×
1
-(1+exp(-x))^-2
×
exp(-x)
-(1+exp(-x))^-2
×
-1
-exp(-x)
(1+exp(-x))^2
exp(-x)
(1+exp(-x))^2

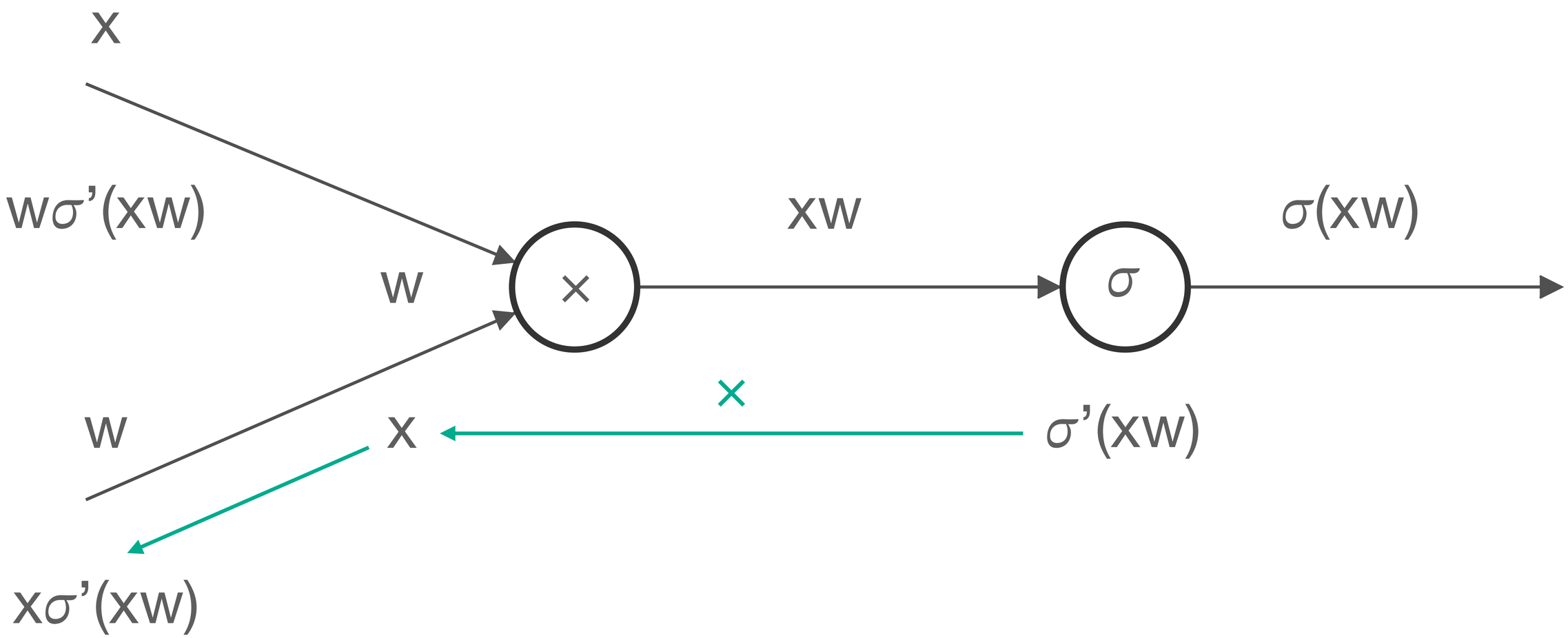
Node with multiple inputs




x
w
×
xw
σ
σ(xw)


x
w
×
xw
σ
σ(xw)
w
x
σ’(xw)

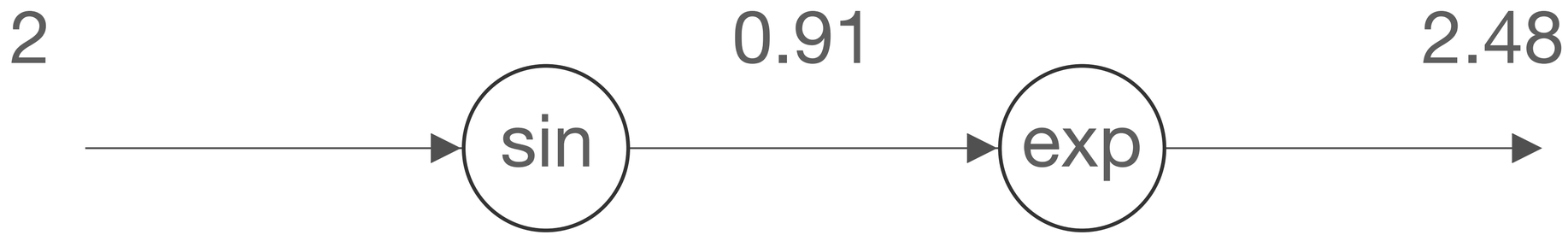

x
wσ'(xw)
w
xw
×
σ
σ(xw)
×
w
x
σ'(xw)

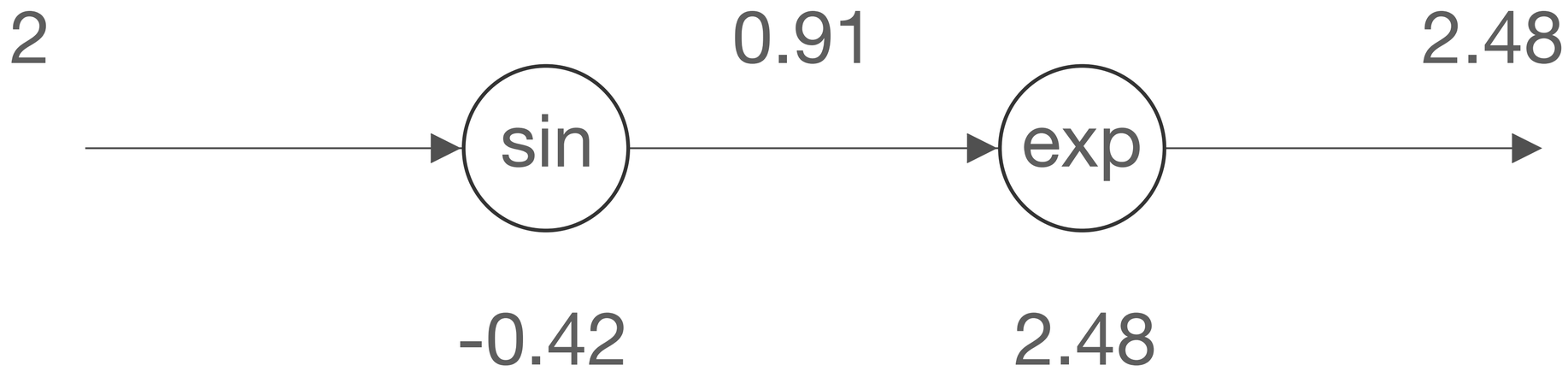

x
wσ'(xw)
w
×
xw
σ
σ(xw)
×
w
x
σ'(xw)
xσ'(xw)

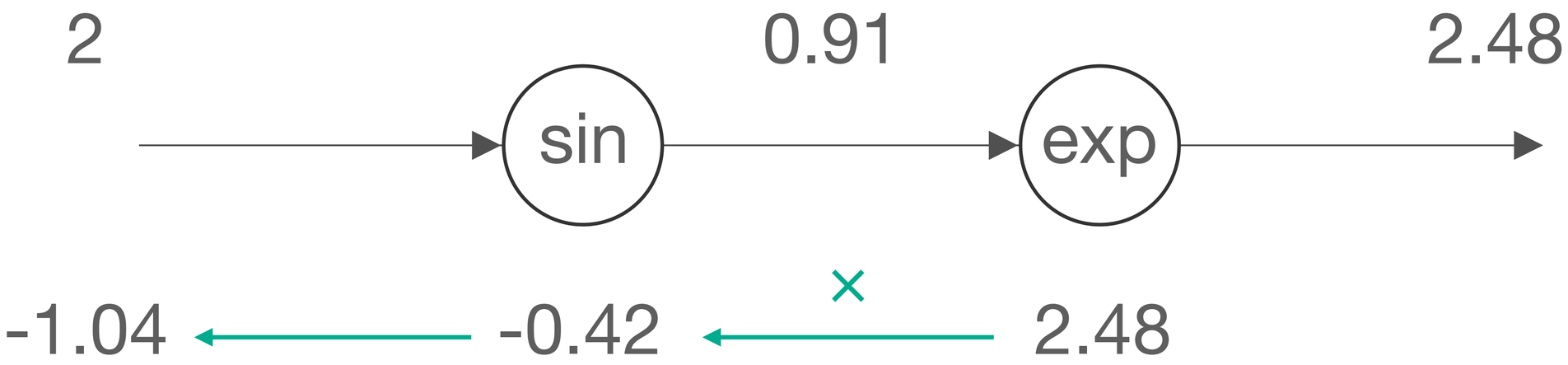
Gradients are numbers

- So far we've been treating this symbolically and it's a bit clunky
  - → We're still carrying a bunch of nested terms around
- But we don't need the whole analytic expression for the gradient
- We just need its specific numeric value in any particular case
  - → In order to subtract <learning rate amount> of it from the specific numeric value that is our weight
- If we just use the numbers it all becomes a lot cleaner

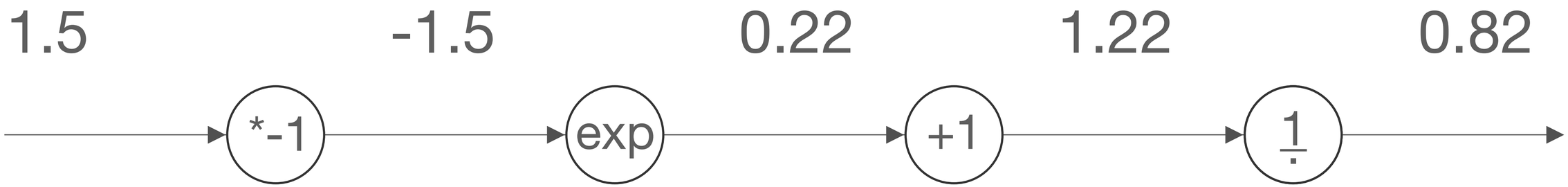
Numeric gradient flows

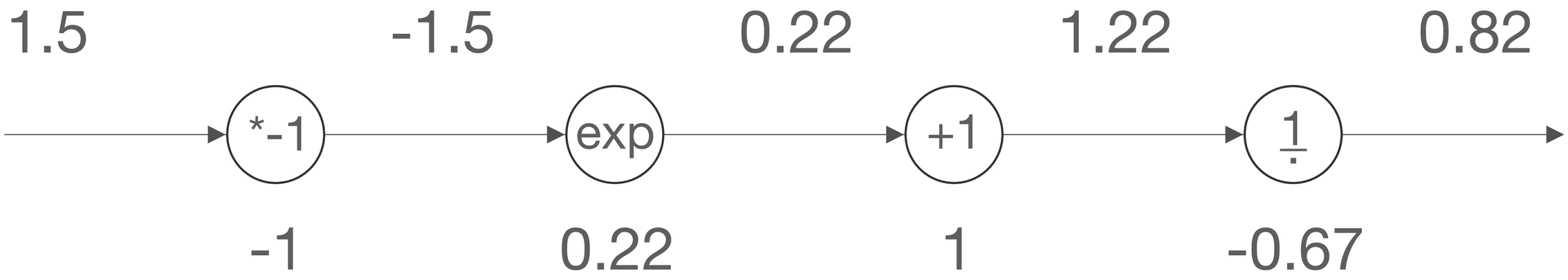

2
0.91
2.48
sin
exp

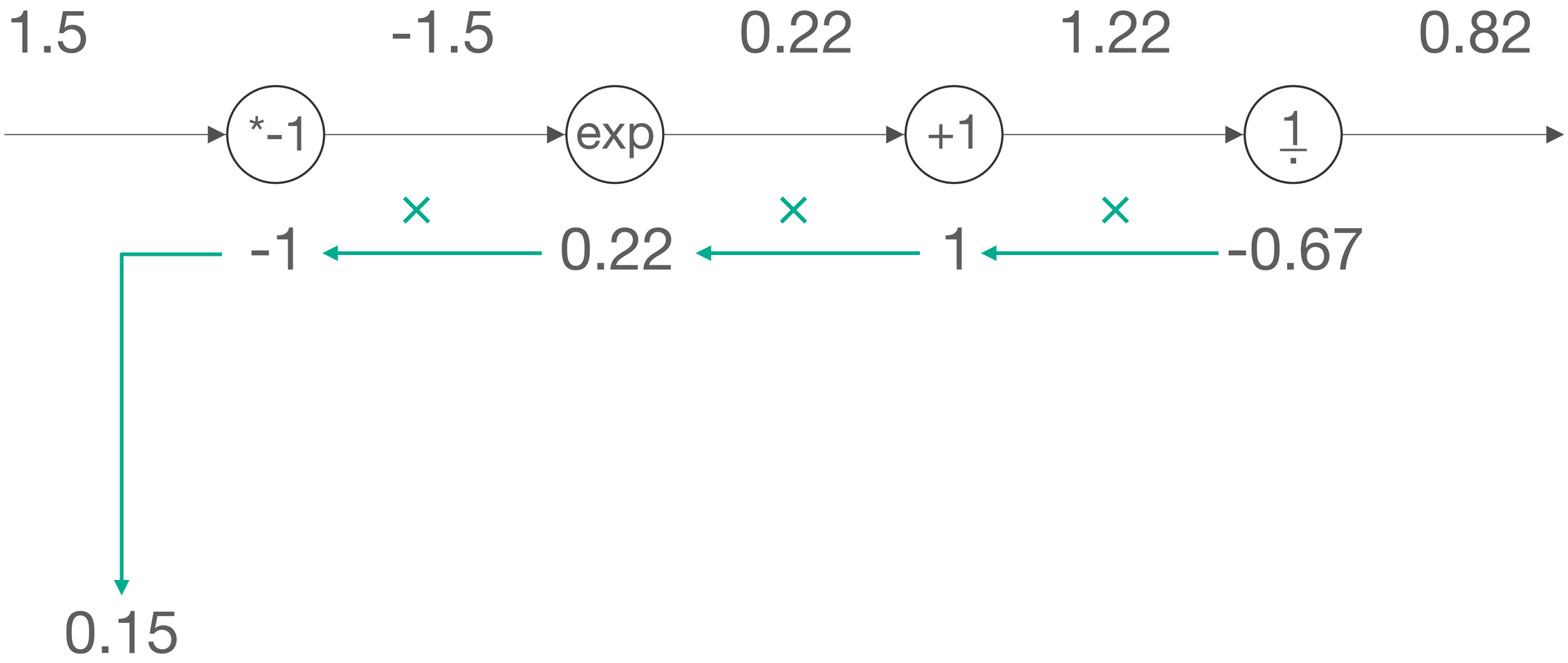

2
0.91
2.48
sin
exp
-0.42
2.48

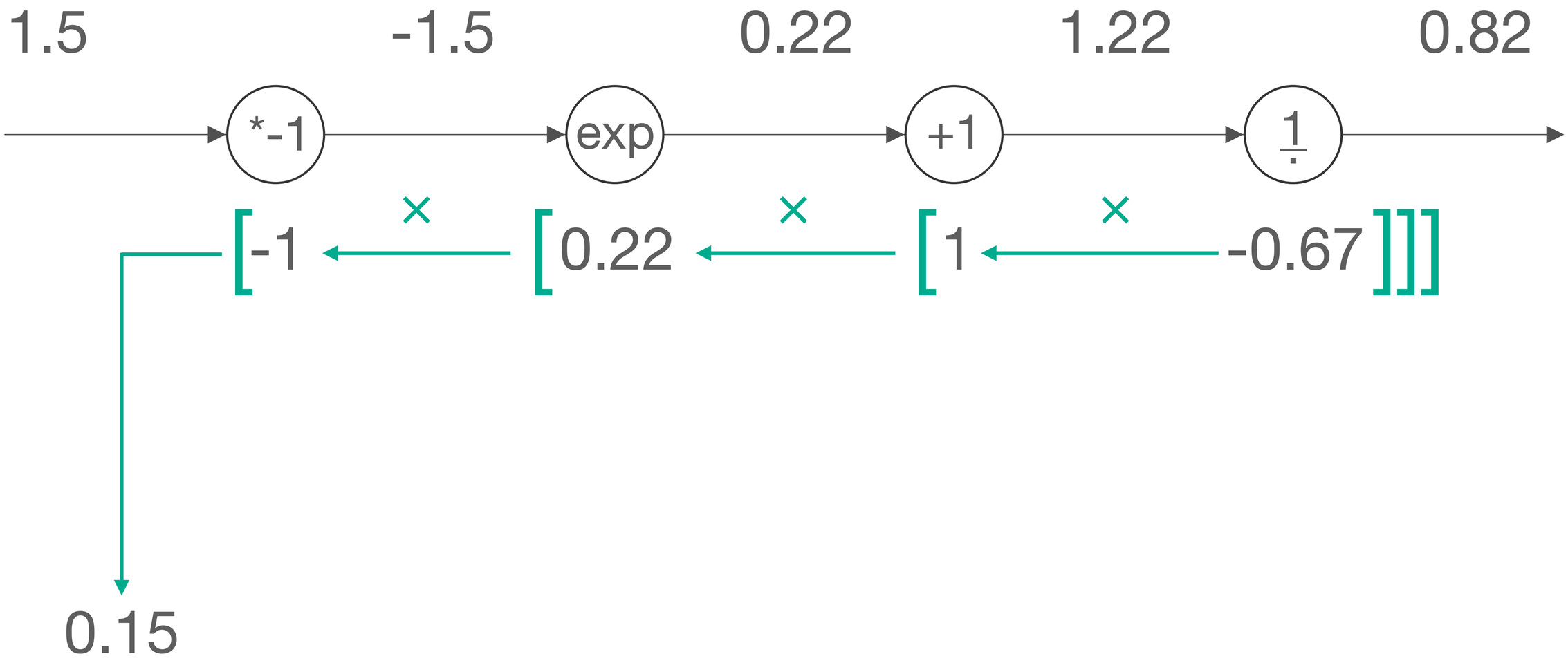

2
0.91
2.48
sin
exp
-1.04
-0.42
×
2.48

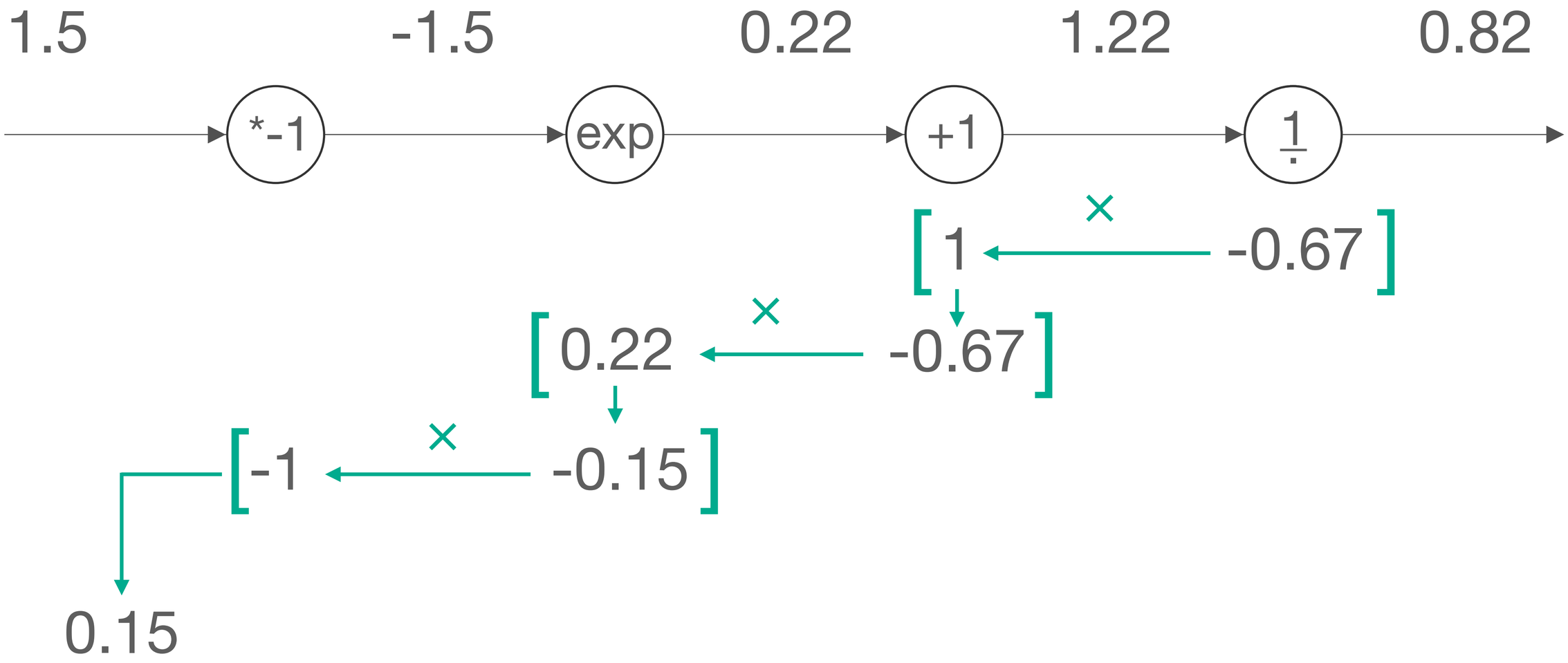
Sequence of multiple nodes


1.5
-1.5
0.22
1.22
0.82
*-1
exp
+1
1

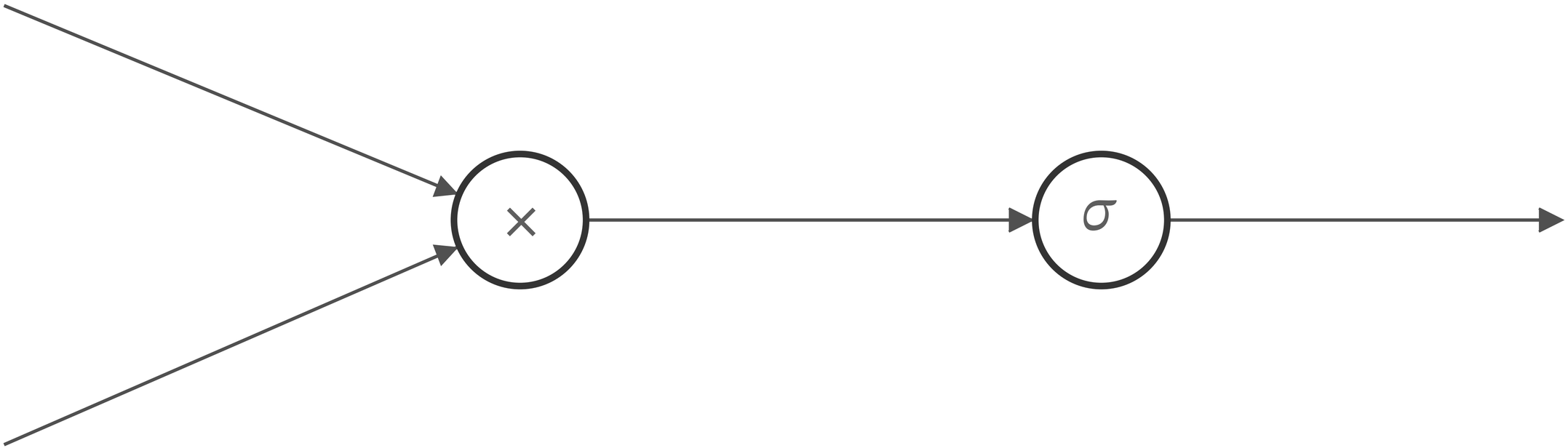

1.5
-1.5
0.22
1.22
0.82
*-1
exp
+1
1
-1
0.22
1
-0.67

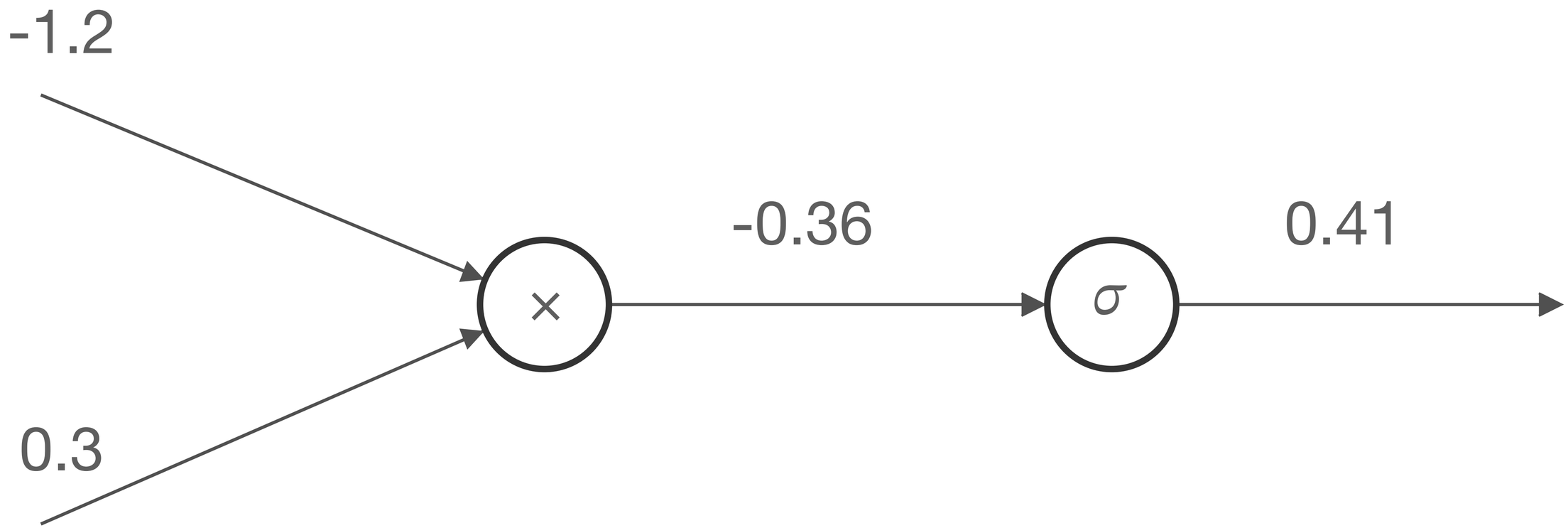

1.5
-1.5
0.22
1.22
0.82
*-1
exp
+1
1
-1
0.22
1
-0.67
0.15

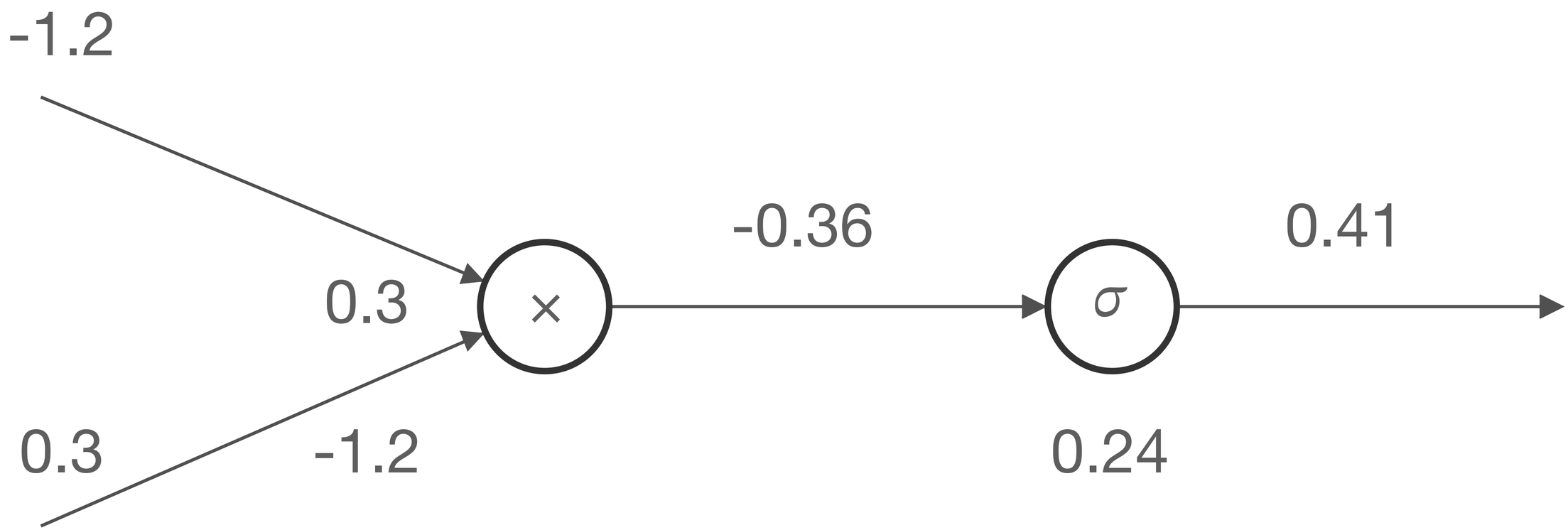

1.5
-1.5
0.22
1.22
0.82
*-1
exp
+1
1
-1
0.22
1
-0.67
0.15

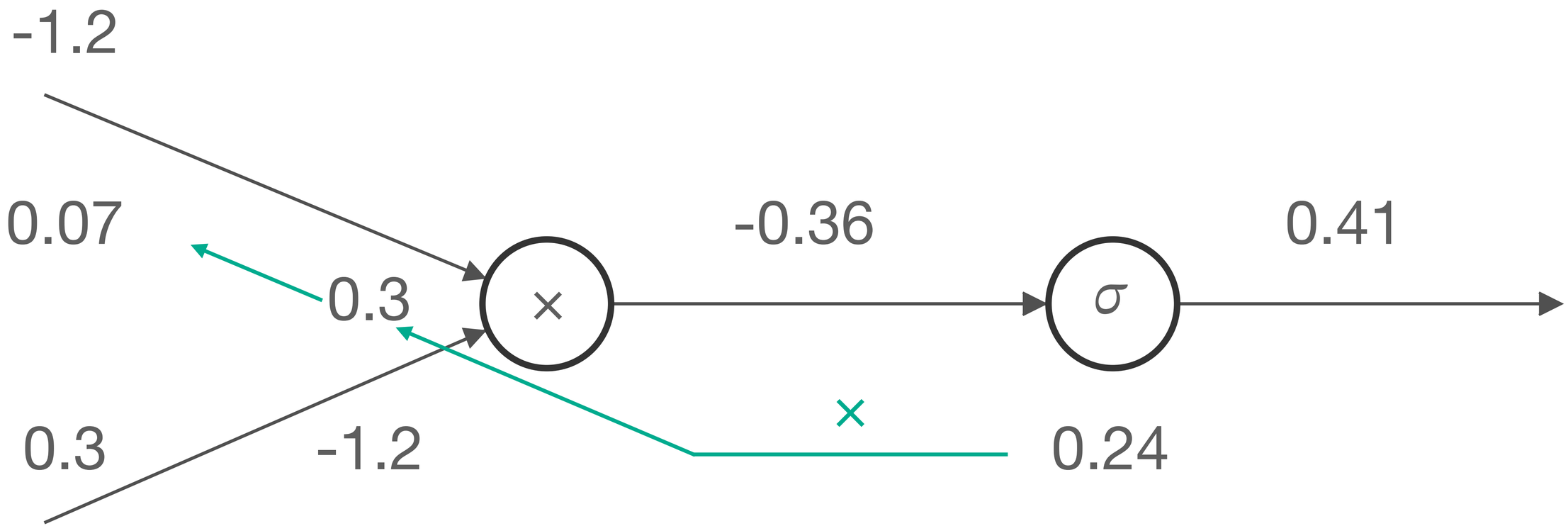

1.5
-1.5
0.22
1.22
0.82
*-1
exp
+1
1
÷
×
1
-0.67
×
0.22
-0.67
×
-1
-0.15
0.15

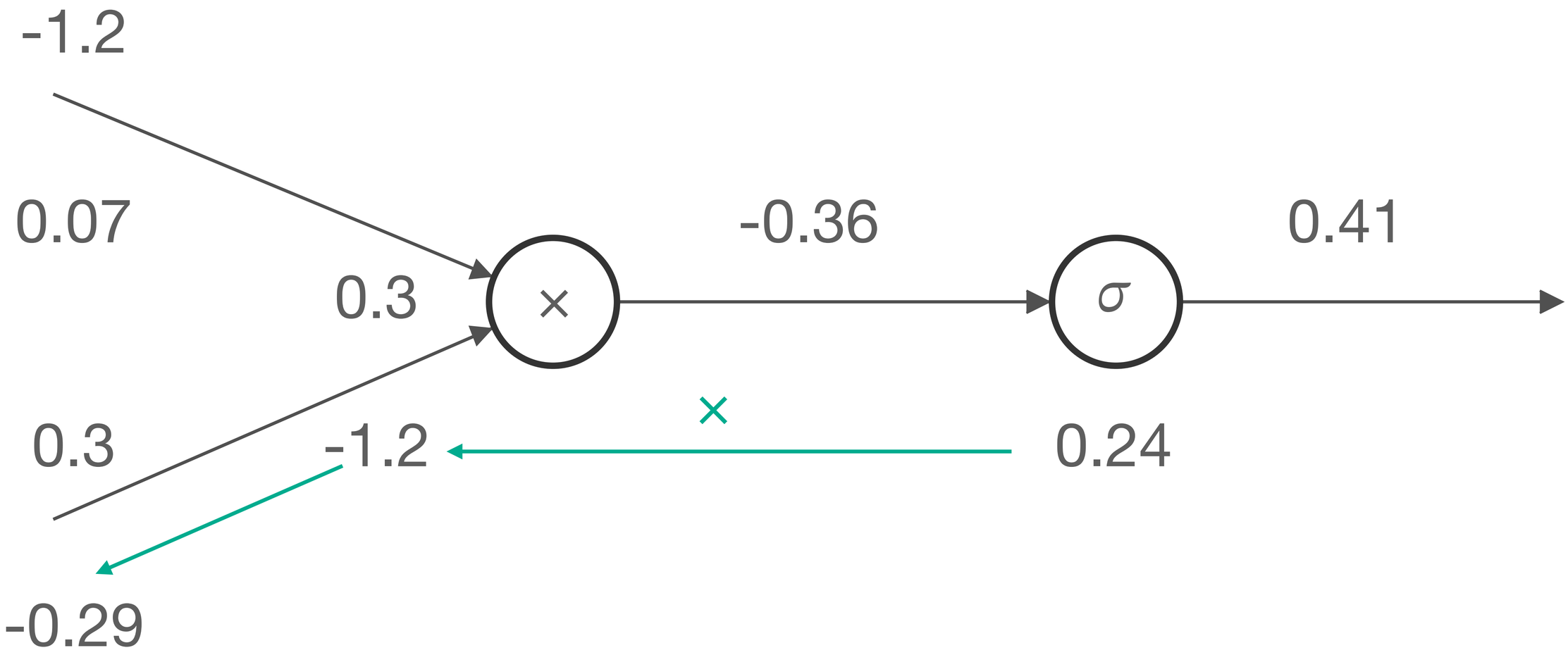
Node with multiple inputs




-1.2
0.3
×
-0.36
σ
0.41


-1.2
0.3
×
-0.36
σ
0.41
0.3
-1.2
0.24


-1.2
0.07
0.3
-0.36
0.41
×
σ
×
0.3
-1.2
0.24


-1.2
0.07
0.3
×
-0.36
σ
0.41
×
0.3
-1.2
0.24
-0.29

```
out1 = f1(x)
out2 = f2(out1)
out3 = f3(out2)
out4 = f4(out3)
loss = f5(out4)

dloss_out4 = df5(out4)
dloss_out3 = df4(out3) * dloss_out4
dloss_out2 = df3(out2) * dloss_out3
dloss_out1 = df2(out1) * dloss_out2
dloss_x = df1(x) * dloss_out1
```

```
out1 = f1(x)
out2 = f2(out1)
out3 = f3(out2)
out4 = f4(out3)
loss = f5(out4)

dloss_out4 = df5(out4)
dloss_out3 = df4(out3) * dloss_out4
dloss_out2 = df3(out2) * dloss_out3
dloss_out1 = df2(out1) * dloss_out2
dloss_x = df1(x) * dloss_out1
```

A local node for local people

- A node lives in its own little bubble
- It keeps track of its own inputs and outputs, and its own local gradient(s)
- It receives and propagates data forward
- It receives and propagates gradients backward
- It saves its own global gradient
- It never needs to know the bigger picture

What counts as a node?

- Free to choose the node boundaries according to what makes sense
- Nodes can be as simple or complex as you like
- If a compound function has a nice simple known derivative — like the sigmoid in the 2-input example — by all means call it a node

Vector inputs

- Broadly the same as the scalar case, but you need to think about what's where
- Gradients are basically always passed as vectors: the backwards gradient is a vector of partials the same size as the input vector
- Two important cases
  - → Inner product of two vectors (ie, linear network layer)
  - → Elementwise operation (eg, non-linear activation function)

Inner product (linear layer)

$$z = \mathbf{x} \cdot \mathbf{w}$$

$$\nabla_{\mathbf{x}} z = \mathbf{w}$$

$$\nabla_{\mathbf{w}} z = \mathbf{x}$$

We need this for propagating back to previous layers

Inner product (linear layer)

$$z = \mathbf{x} \cdot \mathbf{w}$$

$$\nabla_{\mathbf{x}} z = \mathbf{w}$$

$$\nabla_{\mathbf{w}} z = \mathbf{x}$$

We need this for updating the weights in the layer

Elementwise function (activation)

$$a(\mathbf{x}) = [f(x_1), f(x_2), \ldots, f(x_d)]$$

$$\nabla_{\mathbf{x}} a = [f'(x_1), f'(x_2), \ldots, f'(x_d)]$$

Vectors and matrices

- We try to define layer boundaries to hide any undue complexity (cf the Jacobian of the softmax function vs its reduction to a vector when used in combo with cross-entropy)
- For multiple output layers (matrix W) we need all the weight-by-weight terms, and so we need a gradient matrix
- For (mini-)batch inputs (matrix X): can take mean or sum over samples, in which case we only need to propagate a single vector back

  → Mean is more stable and makes gradients more comparable across batch sizes

Forwards and backwards

- Nodes (or layers) are local, but something — some object or data structure — is needed to keep track of the network graph
- Traversing this data structure populates the data and gradients
- This occurs in two passes: a forward pass and a backwards pass

Forward pass

- Forward pass propagates data through nodes in dependency order
- All a node’s inputs must be calculated before the node itself can be called
- Final network output is the loss (or losses)

Backward pass

- Loss gradient is propagated backwards through graph in reverse dependency order
- Last layer calculates the loss gradient wrt its immediate inputs
- Loss is passed back to previous layers, which calculate their own gradients and propagate those back to their own inputs
- This continues right back to the input data

Learning

- At this point, all gradients throughout the whole network are known and we can think about learning
  - → Backpropagation is not a learning procedure, it just calculates gradients
- What happens then is up to the learning algorithm — or “optimiser”
- This may employ vanilla gradient descent or any of the variants previously discussed
- Weights are updated using whatever optimisation policy has been chosen
- Then the whole process repeats until some stopping condition is reached

MLP optimisation is non-convex

- Networks without hidden layers can be convex — eg, we've seen that logistic and softmax regression have a natural network representation
- But adding hidden layers (probably) breaks convexity
- So there is definitely the possibility of getting stuck in local minima
- How serious a problem this is depends on the task, and the training data, and the network structure
- In practice, there is evidence that local minima are not a significant problem for many of the tasks for which deep neural networks are applied, such as computer vision (eg Kawaguchi & Bengio 2019)

5.4: Activation & Initialisation

COMP0088 Introduction to Machine Learning · UCL Computer Science

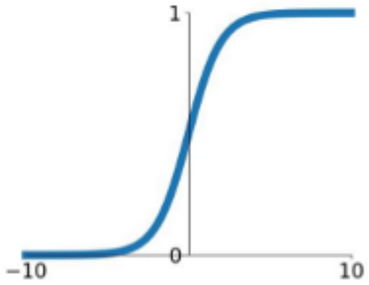
Failure to converge

- Neural nets have been around a long time
- Suffered a setback due to the disappointments of the perceptron, but were back in business in the 1980s
- Backprop as we know it was published in 1986
- CNNs for handwritten digit recognition in 1989 and CNN precursors in 1980
- But often had problems with convergence, especially for deeper networks
- Two of the reasons for this were activation and initialisation

Activation functions

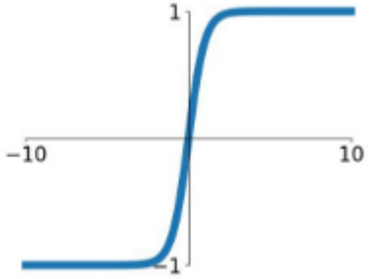
Sigmoid

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



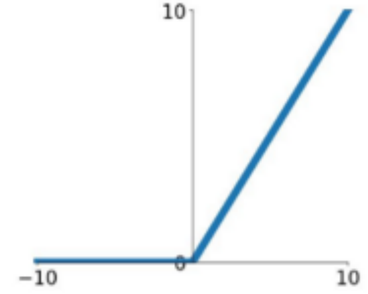
tanh

$$\tanh(x)$$



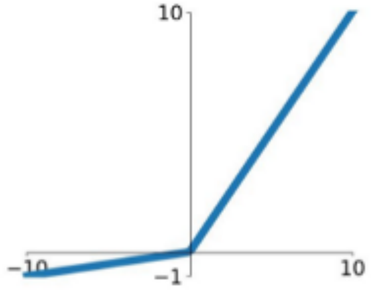
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

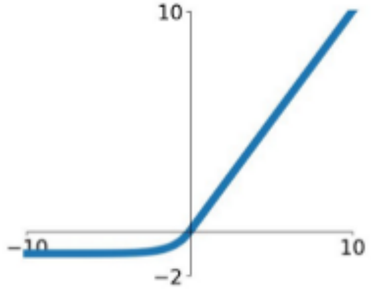


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Vanishing & exploding

- Downsides to the chain rule:
  - → Zeros are deadly
  - → Small numbers get very small
  - → Big numbers get very big
- Activation functions are often prey to this

Activation functions

- sigmoid - vanishing, asymmetric
- tanh - vanishing, symmetric
- relu (rectified linear unit) - asymmetric, non-vanishing, but subject to death
- leaky relu - asymmetric, non-vanishing, undying
- maxout - asymmetric, non-vanishing, double params
- elu (exponential linear unit) - asymmetric, non-vanishing, smooth

- basically: use ReLU

Dead neurons

- If a neuron never fires, it will never learn
- Random initialisation may lead to neurons with zero output for all training samples
- Or learning updates may push the neuron to a point where it has zero output — “falling off the data manifold”
- ReLU is prone to this, although anything with vanishing gradients can come close as well

Symmetry breaking

- With a single perceptron we could get away with initialising to zero
- With multiple neurons in each layer the weights have to be initialised to different values or all parallel neurons will learn the same thing, defeating the whole purpose
- So we randomly initialise from some chosen distribution — uniform and Gaussian are both commonly used
- But it turns out we need to keep an eye on the variance

Preserving variance

- We want to maintain an active and varied population of neurons throughout training
  - → Both in terms of forward pass outputs and backward pass gradients
- If variance is too high, activations and/or gradients can explode or saturate
- If variance is too low, activations and/or gradients can collapse
- Either way, the network becomes untrainable
- Initialisation variance is linked to neuron population size — ie, number of weights in layer
- Because both directions matter, ideally it should account for both fan_in and fan_out

Initialisation and activation are linked

- Optimal variance is different for symmetric tanh and asymmetric relu
  - → tanh: Glorot (aka Xavier) initialisation:

$$w \sim N\left(0, \frac{2}{\text{fan}_{in} + \text{fan}_{out}}\right) \qquad w \sim U\left(-\sqrt{\frac{6}{\text{fan}_{in} + \text{fan}_{out}}}, \sqrt{\frac{6}{\text{fan}_{in} + \text{fan}_{out}}}\right)$$

  - → ReLU: He initialisation:

$$w \sim N\left(0, \frac{2}{\text{fan}_{in}}\right) \qquad w \sim U\left(-\sqrt{\frac{6}{\text{fan}_{in}}}, \sqrt{\frac{6}{\text{fan}_{in}}}\right)$$

Initialisation matters at scale

- Initialisation is less important for smaller, shallower networks
- Effect of multiplying up gradients increases with depth
- Which is one of the reasons early shallow networks worked fine but deeper ones became hard to train

Neural nets roundup

- NNs are very versatile and capable
- Can learn arbitrarily complex decision functions and regression landscapes
- Tend to be black box, implicit and uninterpretable
  - → we'll come back to this
- Need care to train
- Most of the machinery for deep learning has been in place since the 1980s
- Recent renaissance due to computational speed, data availability, and subtleties such as initialisation and activation
- Next time: more!