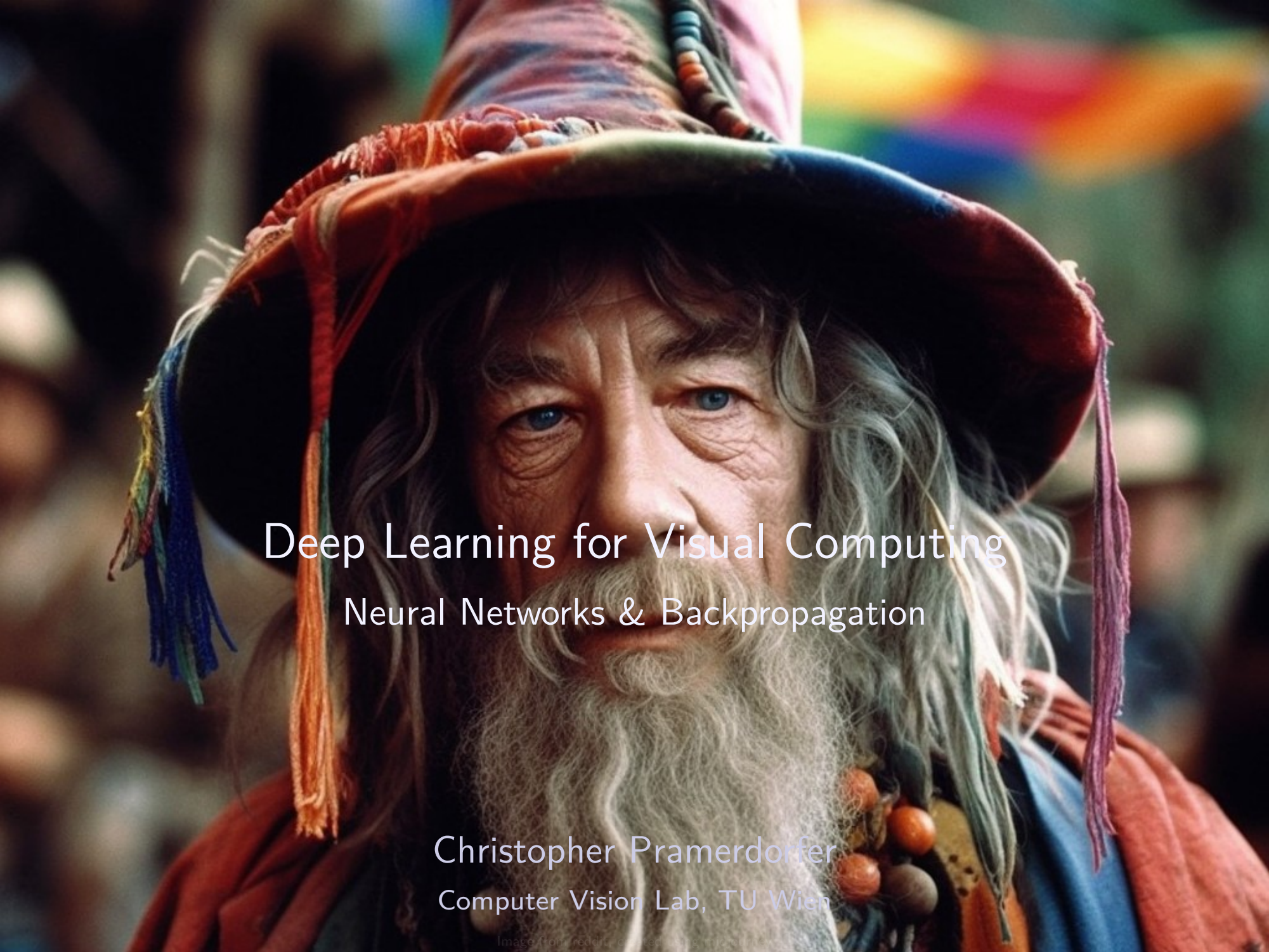


Deep Learning for Visual Computing

Neural Networks & Backpropagation

Christopher Pramerdorfer

Computer Vision Lab, TU Wien

Image from reddit

This Week in AI
Chat-GPT Plugins

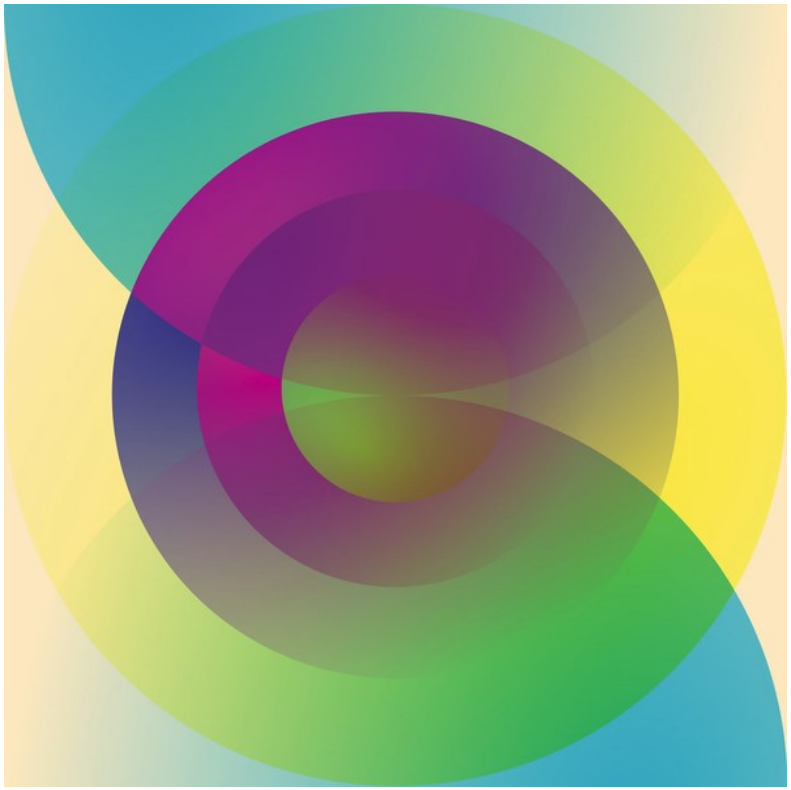


Image from OpenAI

Topics

Neural networks

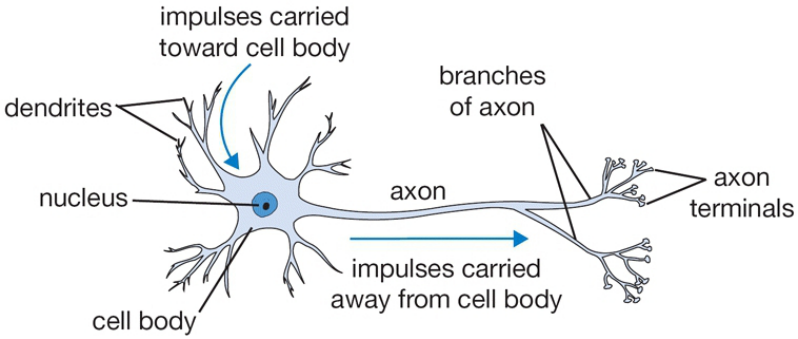
- Motivation
- Computational graphs

Computing function gradients

- Numeric and analytic gradients
- Derivatives in graphs
- Backpropagation algorithm

Neural Networks
Motivation

Original idea was to roughly model neuron behavior



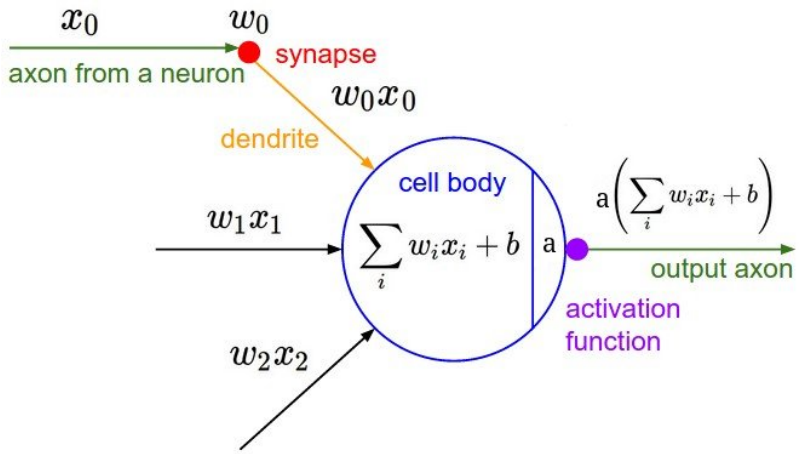


Image from cs231n.github.io

Neural Networks
Motivation

In very simplified terms, a neuron (nerve cell)

- Has multiple input connections
- Weights each input signal by some factor
- Fires output signal if sum of inputs exceeds a threshold
- Firing rate depends on sum

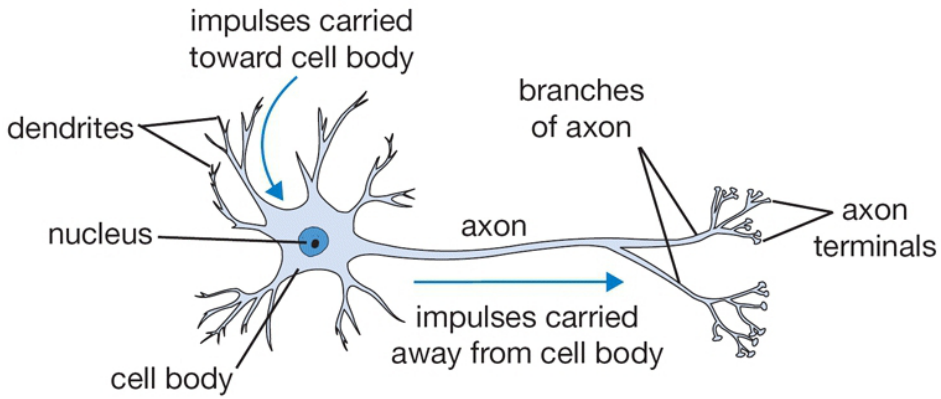


Image adapted from cs231n.github.io

We can model this behavior as $o = a(\mathbf{w} \cdot \mathbf{x} + b)$

- $\mathbf{x}$ and $\mathbf{w}$ are all inputs and weights
- b corresponds to the threshold
- $a(\cdot)$ is an activation function that models firing rate

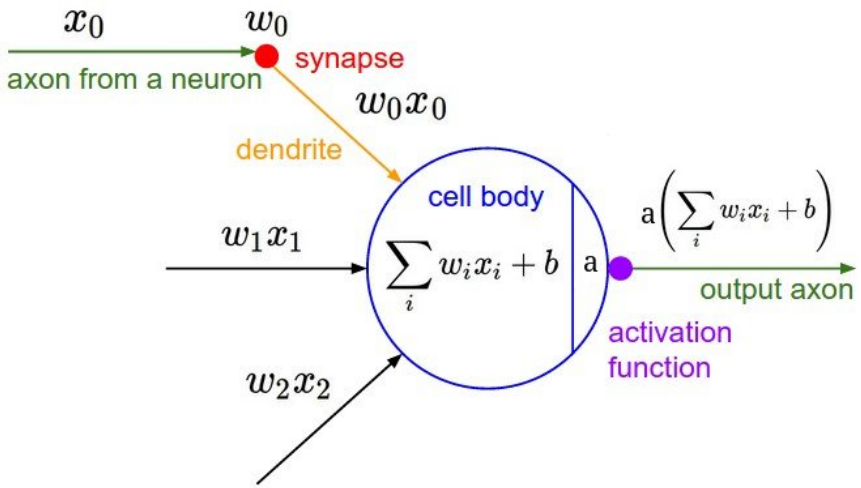


Image adapted from cs231n.github.io

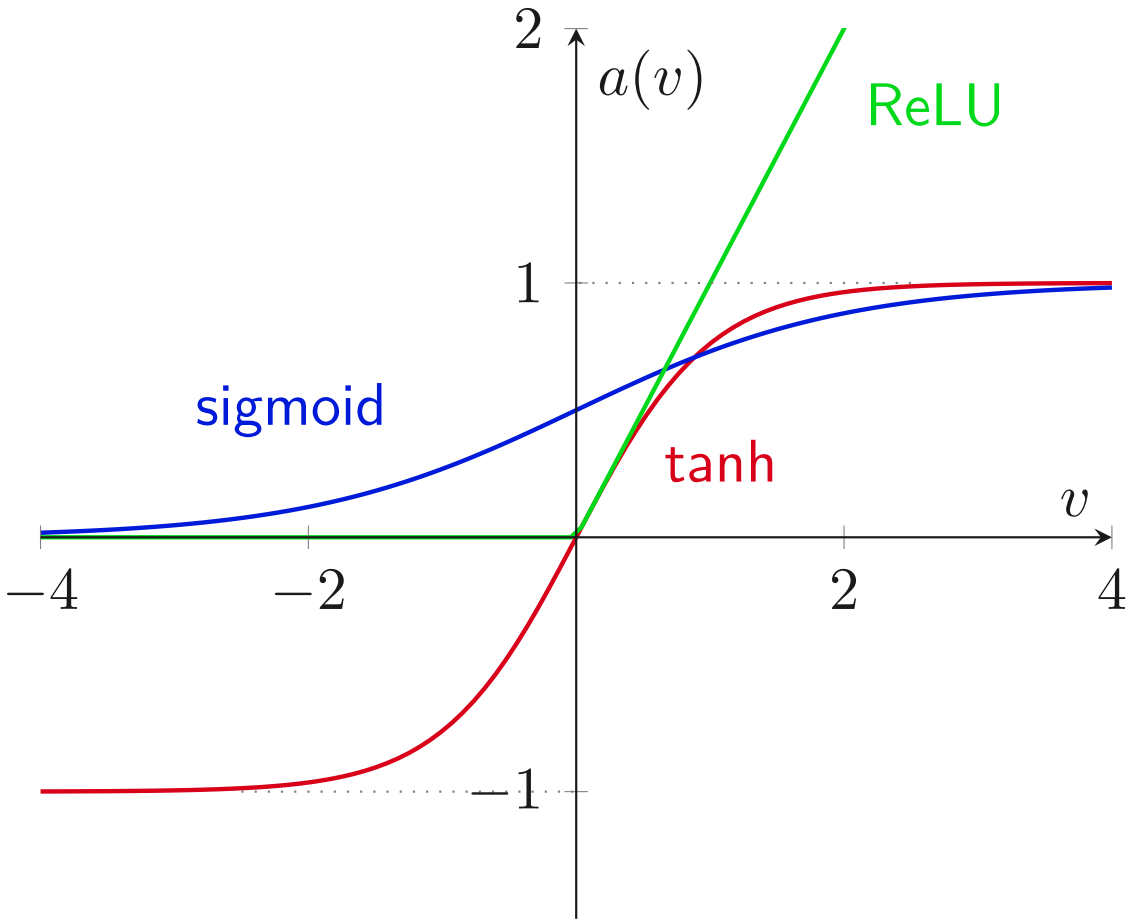
We can select $a(\cdot)$ freely

- Sigmoid function $\mathrm{sgn}(v)$ was popular early on
- Hyperbolic tangent $\tanh(v)$ is better and sometimes used
- Rectified linear unit (ReLU) $\max(0, v)$ most popular now

Use ReLU unless you have good reason not to

- We will talk more about transfer functions later

Neural Networks
Activation Functions



Use ReLU unless you have good reason not to

- Performs close to optimal in wide range of tasks
- Helps preserve signal throughout network (more later)
- Results in sparse networks

Several tweaks to ReLU exist (e.g. Leaky ReLU, ELU)

- Work (marginally) better in certain situations

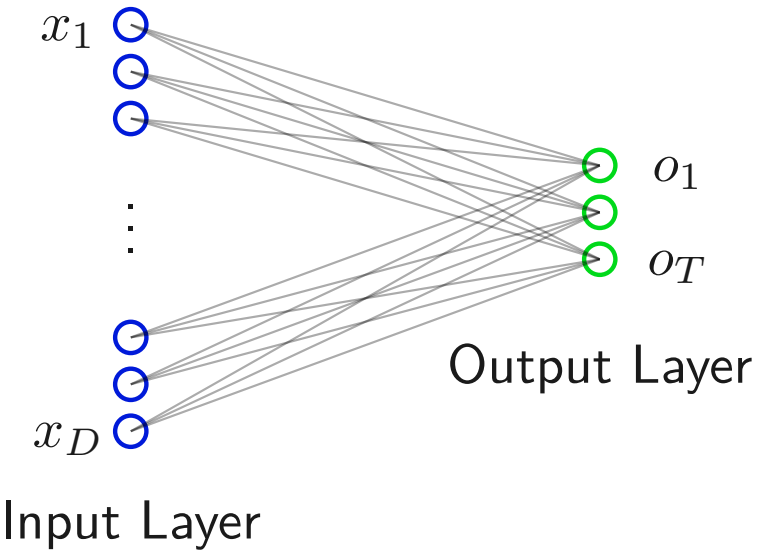
Neurons compute $o = a(\mathbf{w} \cdot \mathbf{x} + b)$

So T neurons form a linear model with T outputs

- Simply set $a(v) = v$ such that $o_t = \mathbf{w}_t \cdot \mathbf{x} + b_t$
- Result is identical to definition in previous lecture

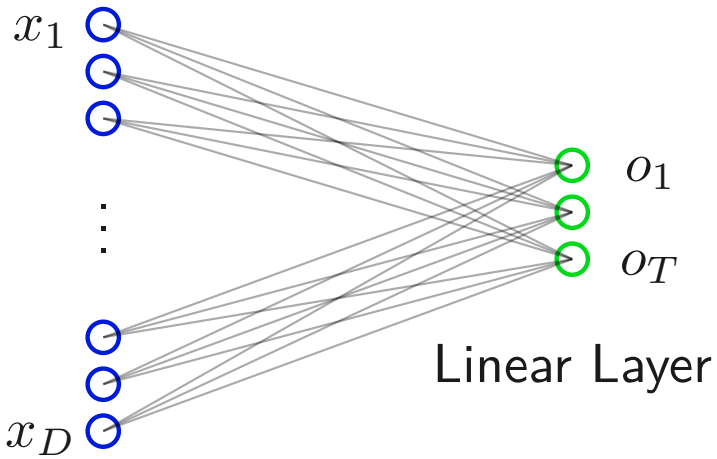
Neurons form layers

- Neurons in same layer see same inputs
- Inputs are often also represented as layer



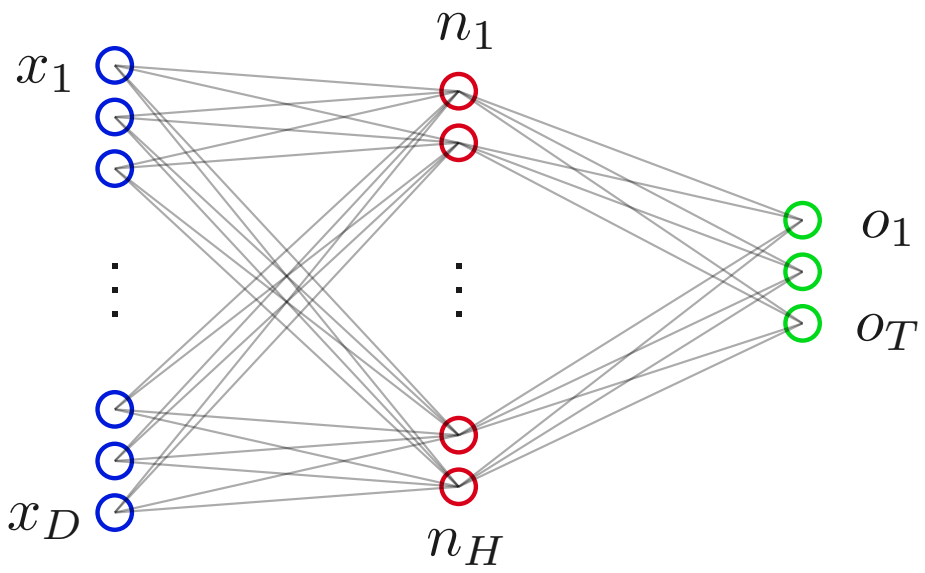
Layers are often given names based on what they do

- Layer with above neurons ($a(v) = v$) called linear
- Activation functions often considered own layer



Can add hidden layers to increase model capacity

- Such networks are called multi-layer perceptrons (MLPs)
- Deep neural networks (DNNs) have several such layers



Assuming $D = 1$ and $a(v) = \text{ReLU}$

- Model is T piecewise linear functions with H kinks
- Same for $D > 1$ but hard to visualize

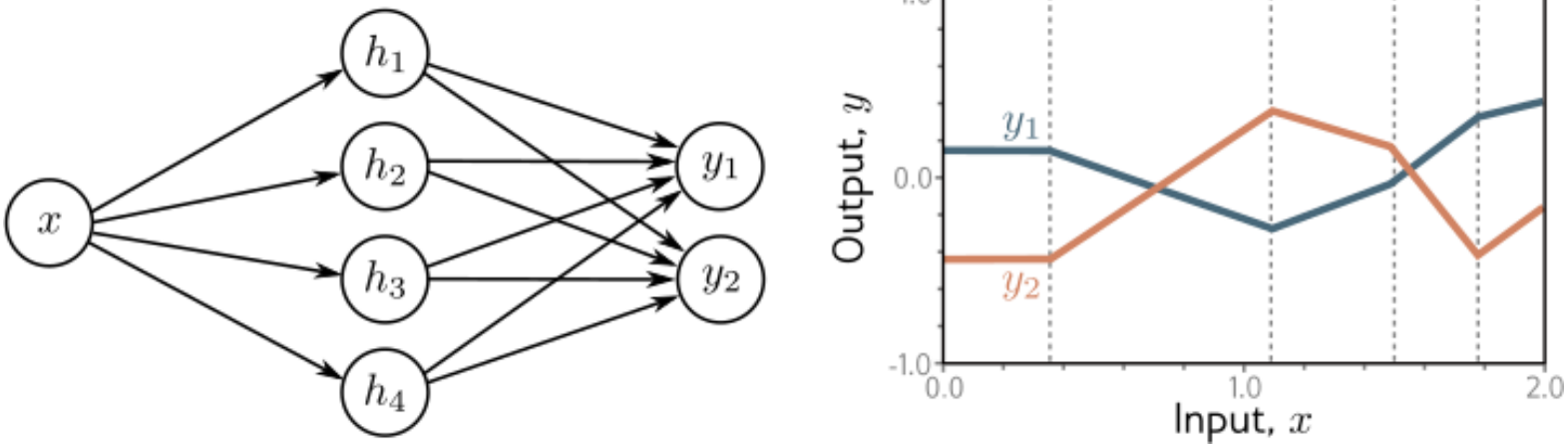


Image from [3]

Neural Networks
Demo

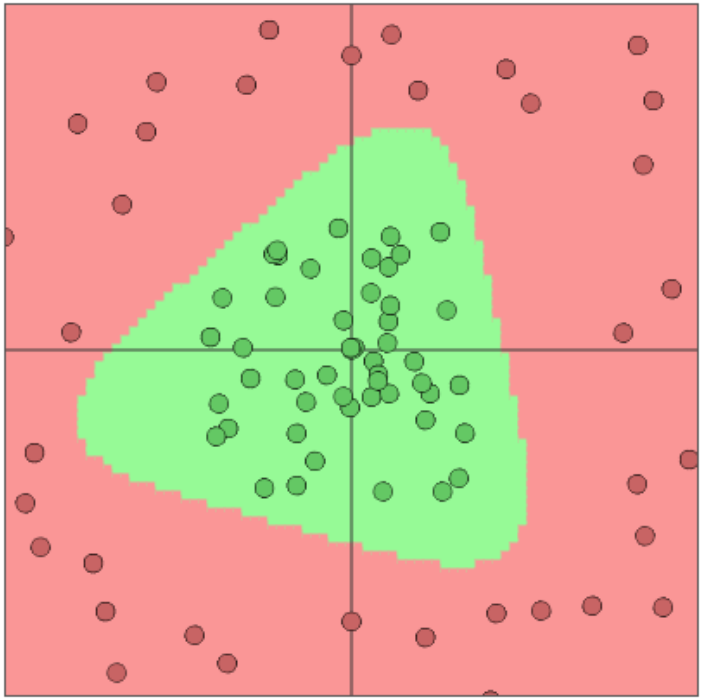


Image from cs.stanford.edu

MLPs are universal approximators (depending on H and $a(\cdot)$)

- Can represent any decision boundary (classification)
- Can approximate any function to any degree (regression)

Does this solve our image classification problems?

- No because we have bad (low-level) feature vectors
- Garbage in, garbage out

What if we use the raw images instead?

- Squeeze $32 \times 32 \times 3$ images to vectors $\mathbf{x}$ with $\dim(\mathbf{x}) = 3072$

Also does not work

- Shallow MLPs perform poorly
- Neither do deep MLPs (also poor scaling)

MLP-based designs that work well exist though

- More in a later lecture

The term "neural network" is misleading

- Biological neurons are much more complex [1]
- Modern (deep) neural networks have evolved a lot

Better to think of modern neural networks as

- Compositions of scalar sub-functions (units)
- That form a directed acyclic computational graph
- And grouped into layers depending on distance from input

Neural Networks
Computational Graphs – Example

$$o = f(x) = f(g_1(h_1(x), h_2(x)), g_2(h_2(x), h_3(x)))$$

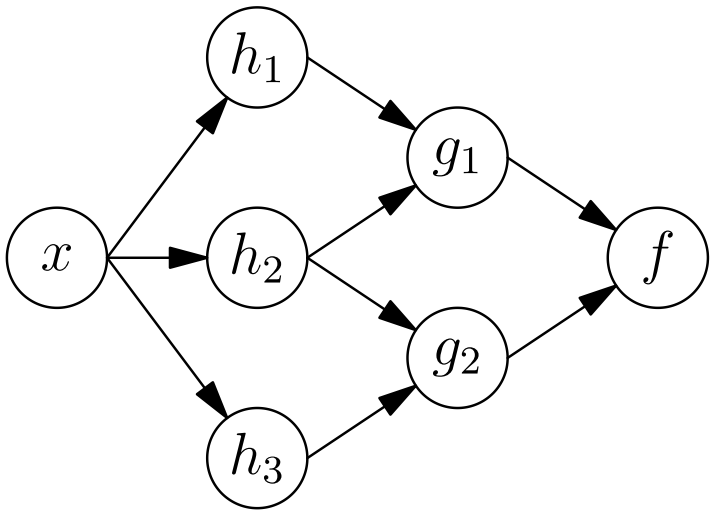


Image from wikipedia

Granularity can be chosen freely

- We might define $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$
- Or $g(\mathbf{x}) = h_1(x_1) + \cdots + h_D(x_D) + b$ with $h_d(x_d) = w_d x_d$
- Or whatever we want really

We will vary granularity depending on context

- More granularity helps understanding training process
- Implementations are less granular (parallel computations)

blank page

Gradient Computation

Training involves computing $\nabla L(\boldsymbol{\theta})$ in every iteration

- Many iterations required to train networks
- Computation must be exact and efficient

Gradient Computation

Parameter Initialization

Got to initialize all parameters first

- Multiplicative weights $\mathbf{w}_1 \cdots \mathbf{w}_H$ and $\mathbf{w}_1 \cdots \mathbf{w}_T$
- Additive biases $b_1 \cdots b_H$ and $b_1 \cdots b_T$

Want to preserve signal strength throughout network

- Avoid numerical issues (floating point math)
- Prevent exploding or vanishing gradients

Biases are not critical

- Simply set to 0 initially

Gradient Computation

Parameter Initialization

Weights are critical (multiplicative effect on output)

- Signal will vanish/explode if weights too small/large
- The deeper the network, the bigger the problem

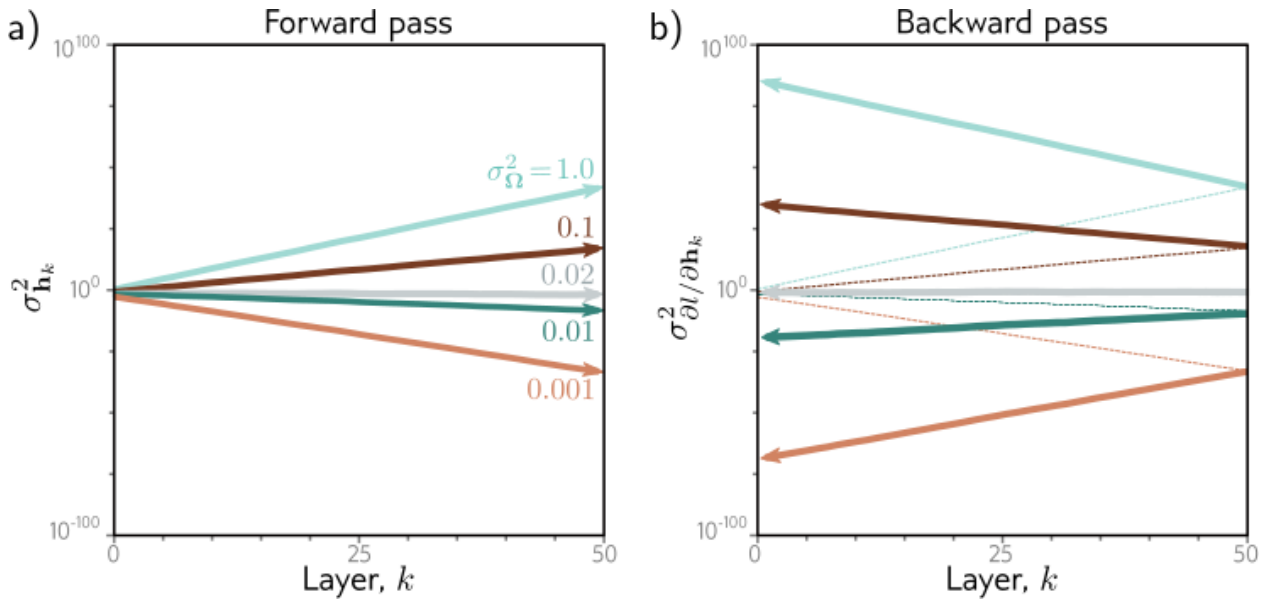


Image from [3].

Gradient Computation
Parameter Initialization

Sample weights from $\mathcal{N}(0, \sigma)$ to avoid issues

- Optimal σ depends on number of connections and $a(\cdot)$
- See [3] for details if interested

Note that this preserves signal only initially

- Weights change during training
- Why normalization layers are important (more later)

Gradient Computation

Numerical Gradients

One way to obtain $\nabla L(\boldsymbol{\theta})$ is numerical differentiation

- $\nabla L_p(\boldsymbol{\theta}) = (L(\boldsymbol{\theta} + \mathbf{1}_p\epsilon) - L(\boldsymbol{\theta}))/\epsilon$ for $p \in [1, \dim(\boldsymbol{\theta})]$
- Vector $\mathbf{1}_p$ is 1 at position p and 0 otherwise
- Follows directly from definition of the derivative

Practical considerations

- ϵ should be close to 0 while avoiding numerical issues
- $\nabla L_p(\boldsymbol{\theta}) = (L(\boldsymbol{\theta} + \mathbf{1}_p\epsilon) - L(\boldsymbol{\theta} - \mathbf{1}_p\epsilon))/2\epsilon$ preferable

Gradient Computation
Numerical Gradients

Trivial to implement

Only an approximation (ϵ cannot be arbitrarily small)

Too inefficient in practice

- Must evaluate $L \dim(\boldsymbol{\theta}) + 1$ times
- Complex networks have millions of parameters

Gradient Computation
Analytic Gradients

We thus would prefer the analytic gradient

- Obtain ∇L analytically using calculus

Can compute $\nabla L(\boldsymbol{\theta})$ directly

- Accurate (no approximation)
- Potentially much more efficient (single evaluation)

Gradient Computation
Derivatives in Graphs

Neural networks are computational graphs

- As are loss function defined for them

Derivatives in such graphs can be computed iteratively

- Recursive application of the chain rule
- Recall that if $F(x) = f(g(x))$ then $F'(x) = f'(g(x))g'(x)$

To compute gradients in such graphs we

- Evaluate the graph and store local results (forward pass)
- Aggregate local gradients (backward pass)

Gradient Computation
Derivatives in Graphs

Simple example with $e(a, b) = (a + b)(b + 1)$

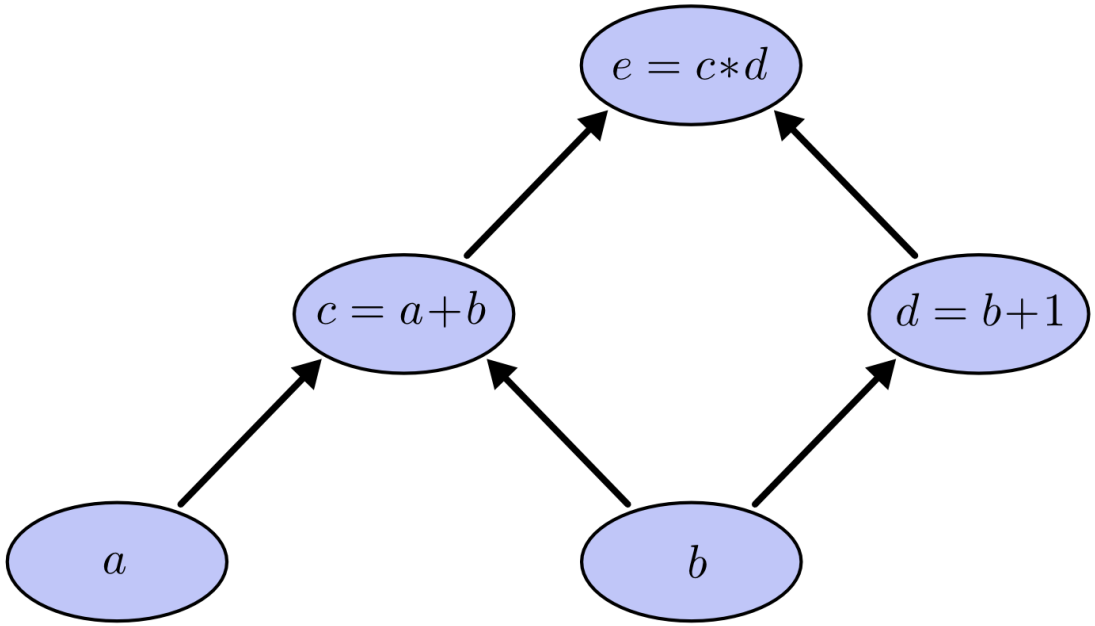


Image from colah.github.org

Gradient Computation
Derivatives in Graphs

Forward pass with $a = 2$ and $b = 1$

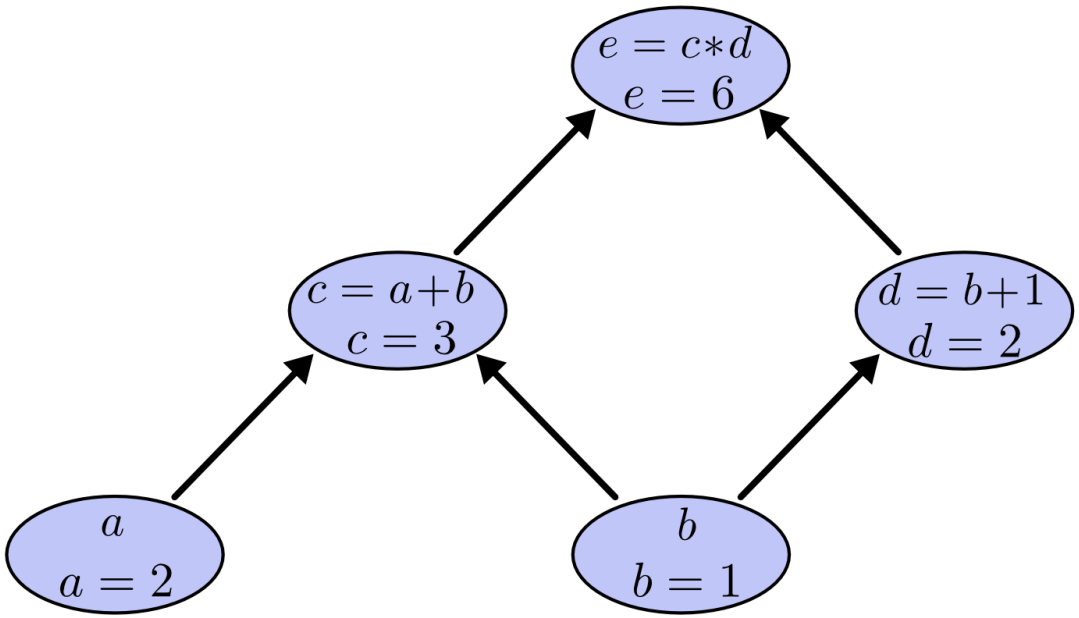


Image from colah.github.org

Gradient Computation
Derivatives in Graphs

Every node can compute local gradients independently

- $\partial f / \partial x$ means partial derivative f_x

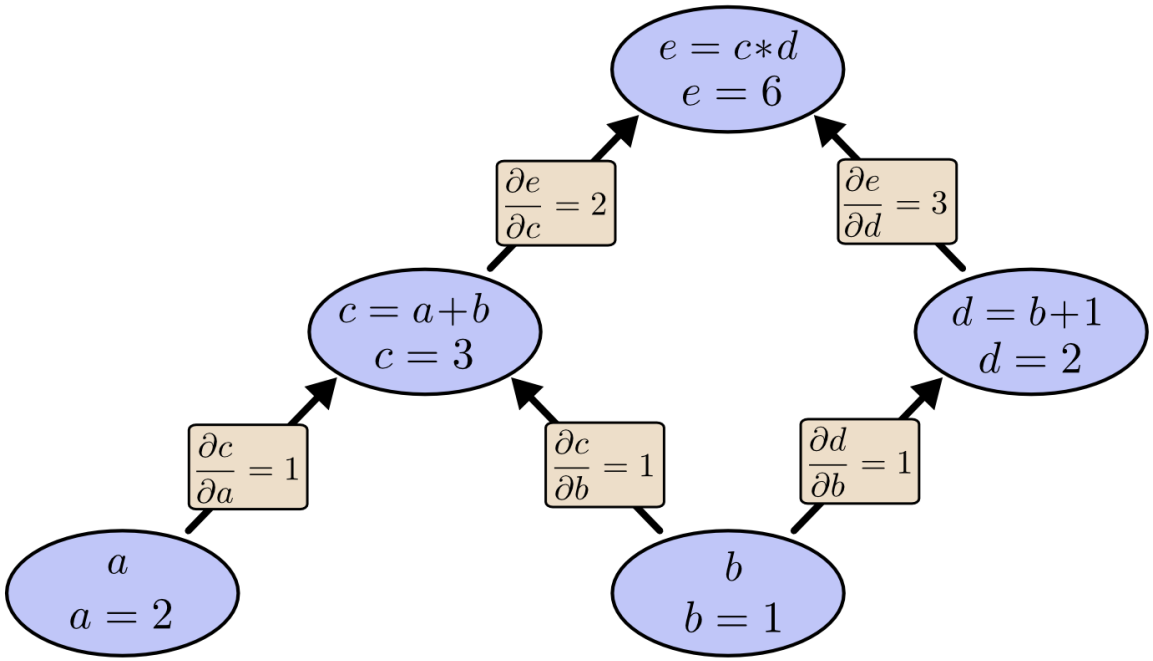


Image from colah.github.org

To obtain $\nabla e(2,1)$ we use the multivariate chain rule

We calculate $\nabla e_a(2,1)$ by

- Multiplying local gradients along every path from a to e
- Summing over all resulting values

Same for e_b (and all other variables in general)

Gradient Computation
Derivatives in Graphs

$$e_a(2,1) = c_a(2,1) \cdot e_c(2,1) = 1 \cdot 2 = 2$$

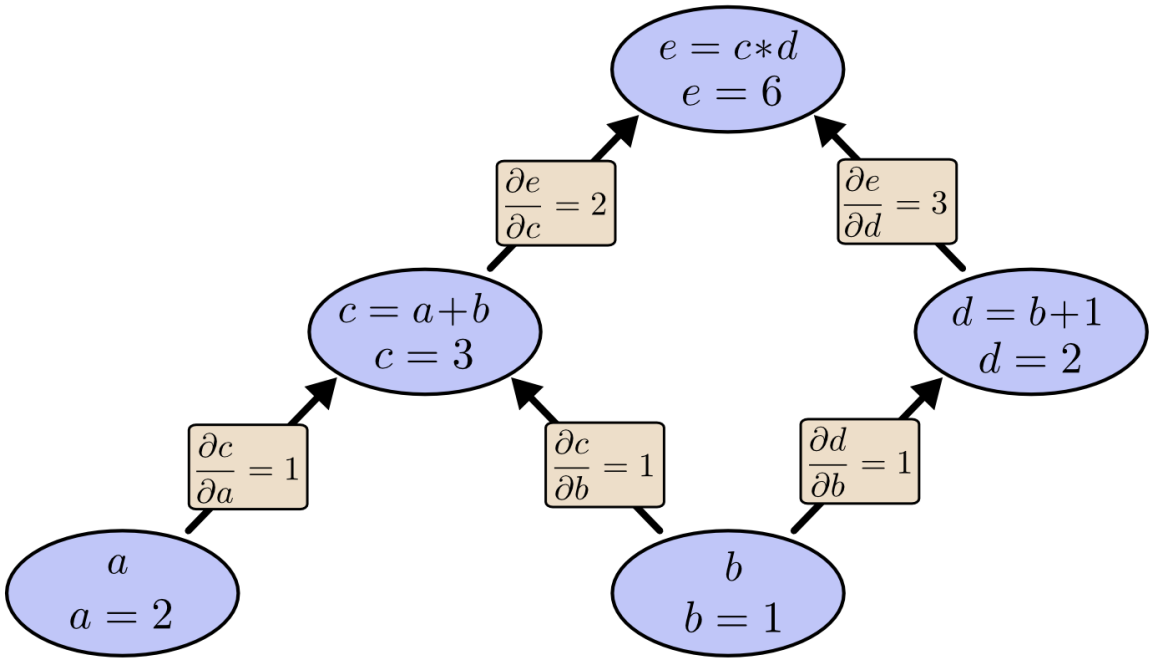


Image from colah.github.org

Gradient Computation
Derivatives in Graphs

$$e_b(2,1) = c_b(2,1) \cdot e_c(2,1) + d_b(2,1) \cdot e_d(2,1) = 2 + 3 = 5$$

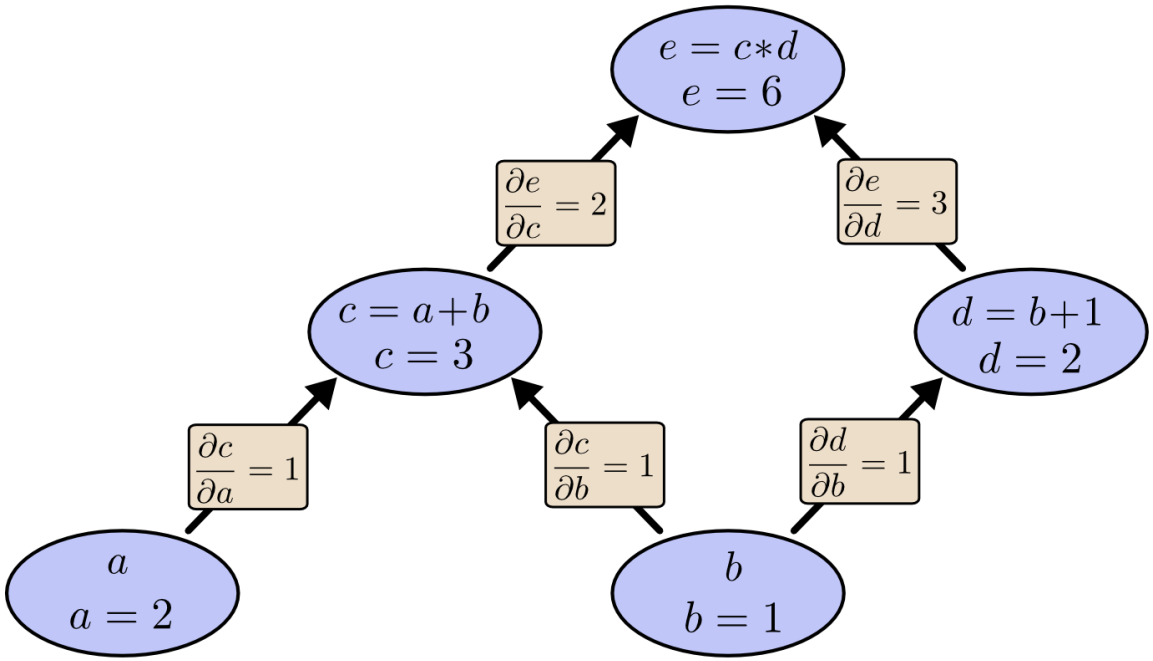


Image from colah.github.org

Can use the same algorithm to compute $L(\boldsymbol{\theta})$

Recall that the loss is an average over S samples

- $L(\boldsymbol{\theta}) = 1/S \cdot \sum_s H(\mathbf{o}^s, \text{softmax}(f(\mathbf{x}^s; \boldsymbol{\theta})))$

So $\nabla L(\boldsymbol{\theta})$ is average of individual $\nabla H(\boldsymbol{\theta})$

- Compute $\nabla H(\boldsymbol{\theta})$ for all s and average
- Of course this applies to loss functions in general

Gradient Computation
Derivatives in Graphs

To calculate $\nabla H(\boldsymbol{\theta})$ we

- Decompose the NN to simple functions
- Do the same for the softmax and cross-entropy
- Stack both to obtain a combined graph
- Use the same algorithm as above

Neural networks are graphs of simple functions

- Can decompose the inner products

f	f'
$x_1 + x_2$	1
$x_1 x_2$	x_2 and x_1
$\max(0, x)$	0 or 1

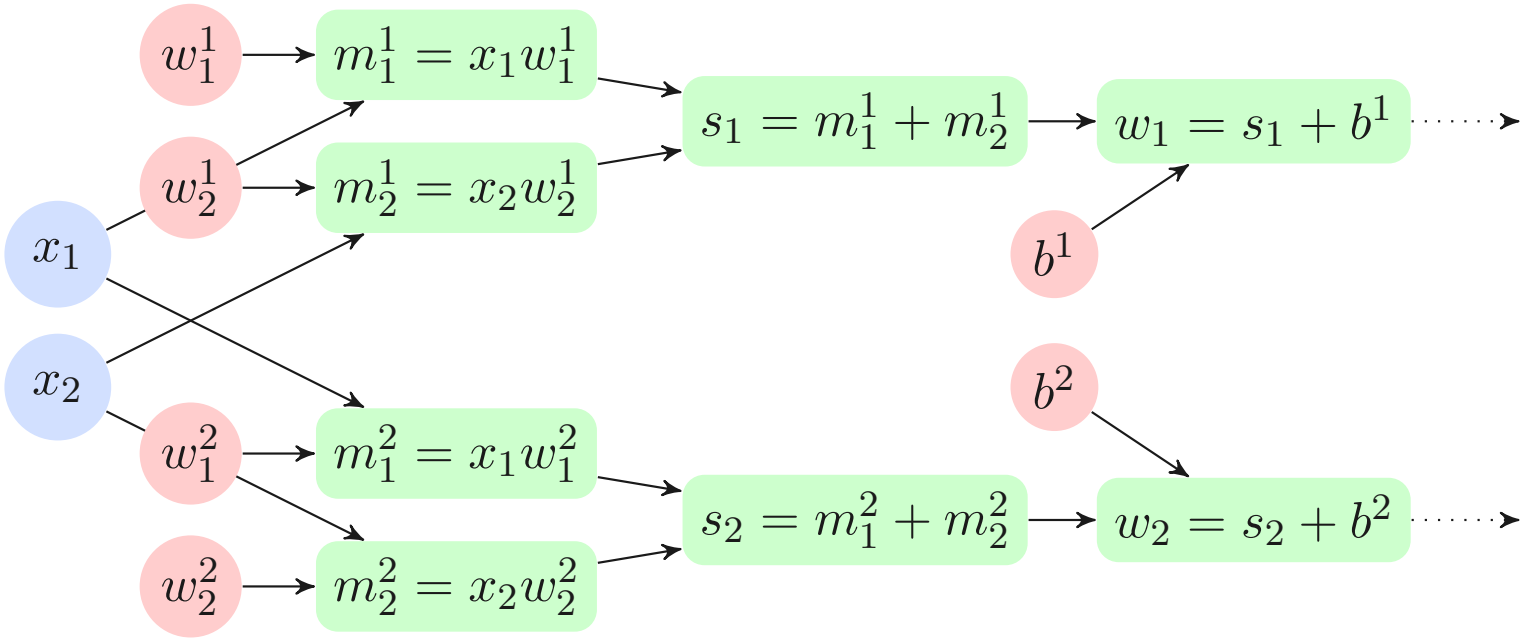
As are the cross-entropy and softmax functions

f	f'
$\exp(x)$	$\exp(x)$
$\ln(x)$	$1/x$
x_1/x_2	$1/x_2$ and $-x_1/x_2^2$

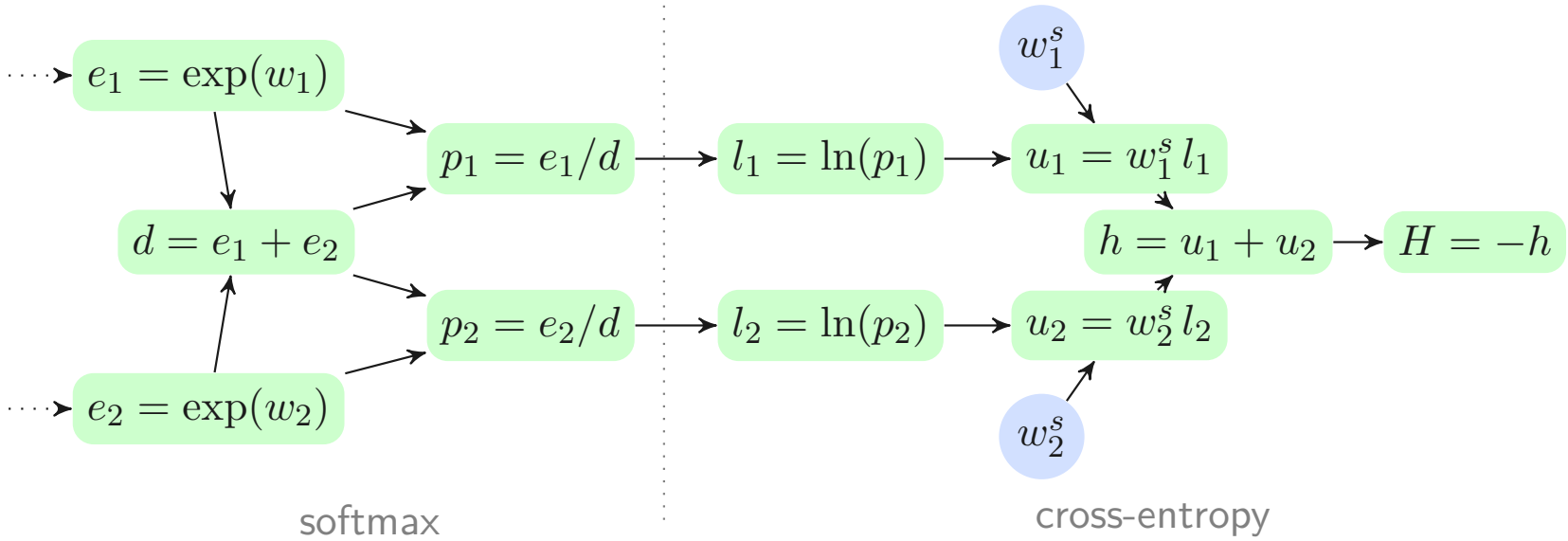
Gradient Computation

Derivatives in Graphs

Linear classifier with $D = 2$ and $T = 2$



"Attached" softmax and cross-entropy



Gradient Computation
Derivatives in Graphs

Graph is dense

- Must sum over several paths per partial derivative
- Number of paths grows exponentially with graph complexity
- Above algorithm not efficient enough for large networks

Reverse-mode differentiation solves this problem

- Computes derivatives of output node wrt. all other nodes
- Efficiently by touching every edge only once
- Called backpropagation in neural network community

Achieved by

- Starting at the output (loss) node
- Propagating local gradients backwards to input nodes
- Storing intermediate results for efficiency

Gradient Computation
Derivatives in Graphs

Start at output node e and move towards inputs

At every node n

- For every child c, compute local gradient $l_c = \partial n / \partial c$
- For every child c, compute $m_c = l_c \cdot \partial e / \partial n$ $(\partial e / \partial e = 1)$
- Compute $\partial e / \partial c$ as sum over all m_c

Gradient Computation

Derivatives in Graphs

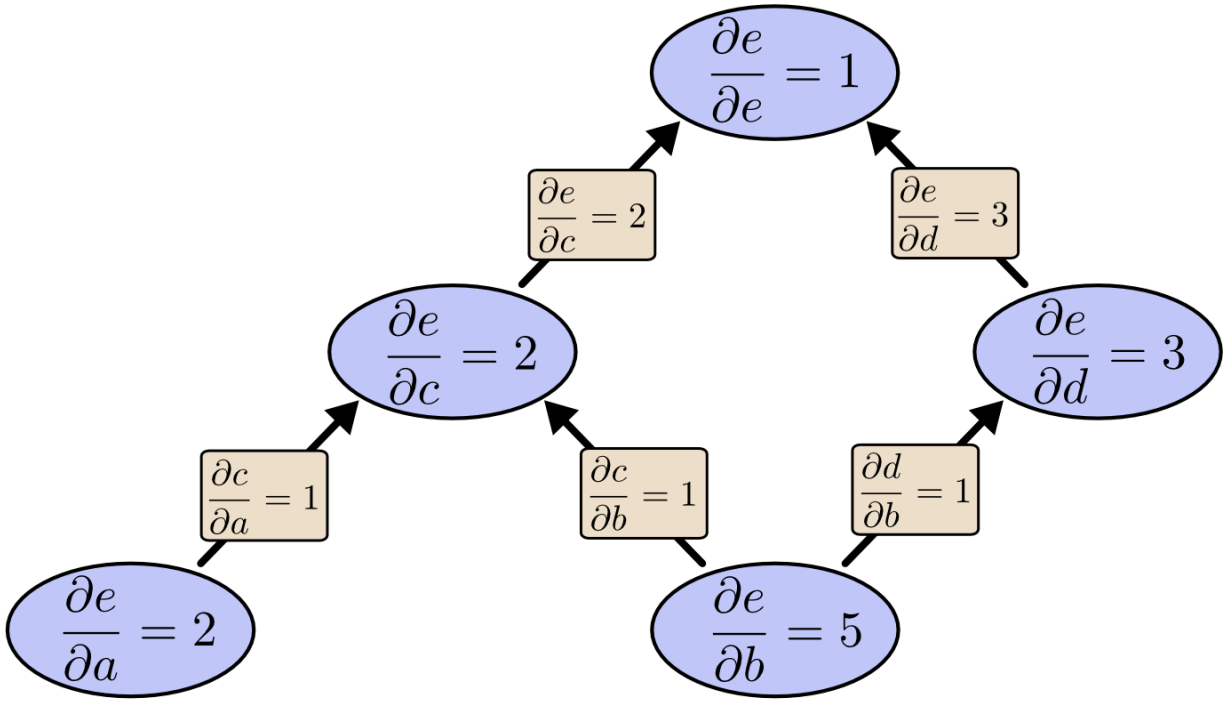


Image from colah.github.org

Gradient Computation
Derivatives in Graphs

Resulting patterns in gradient flow

- `Add` distributes gradients unchanged to all inputs
- `Max` routes gradient unchanged to largest input
- `Multiply` multiplies with switched inputs

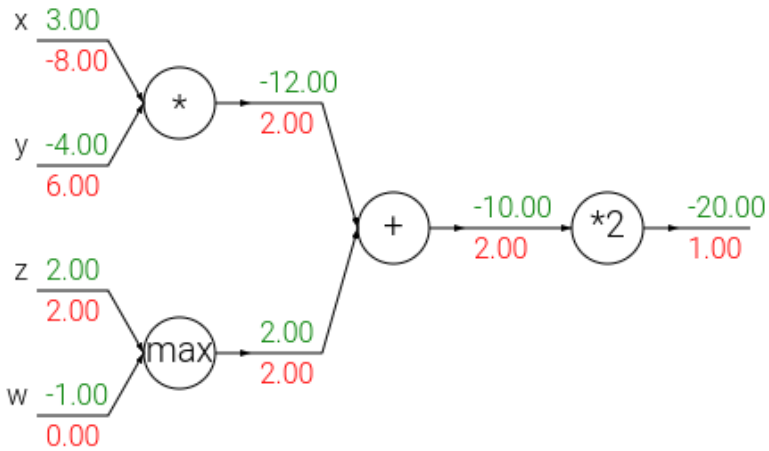


Image from cs231n.github.io

Gradient Computation
Derivatives in Graphs

Neural networks are always trained using backpropagation

- Can increase efficiency by many magnitudes
- Makes training huge (deep) neural networks feasible

In practice

- Graph composition not as fine (vectorization)
- $\nabla H(\boldsymbol{\theta})$ computed in parallel for all s (data parallelism)

Bibliography

[1] Brunel et al. Single neuron dynamics and computation. 2014.

[2] Prince. Computer Vision Models. 2012.

[3] Prince. Understanding Deep Learning. 2023.