Lecture 2: Backpropagation

Peter Bloem

Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

BACKPROPAGATION

Preliminaries:

- neural networks
- the gradient of the loss
- learning by gradient descent

Today:

- backpropagation

VU

THE PLAN

part 1: review

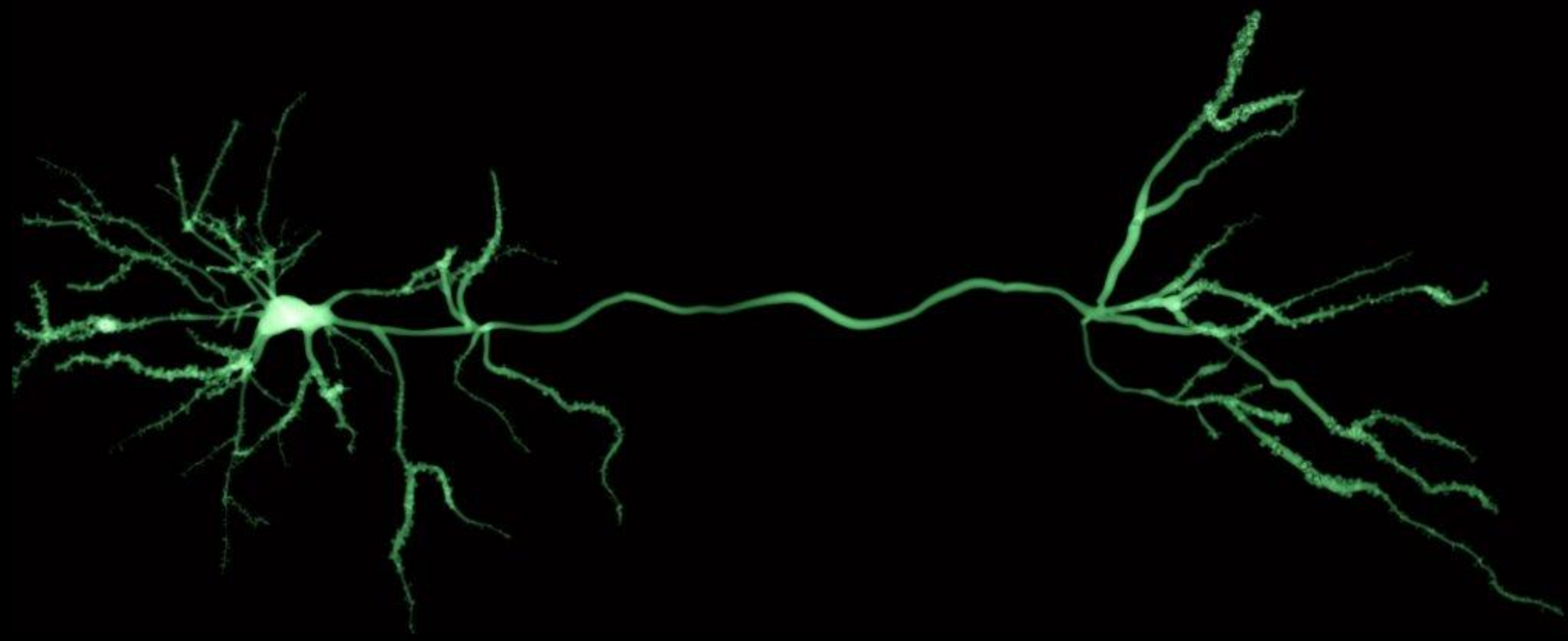
part 2: scalar backpropagation

part 3: tensor backpropagation

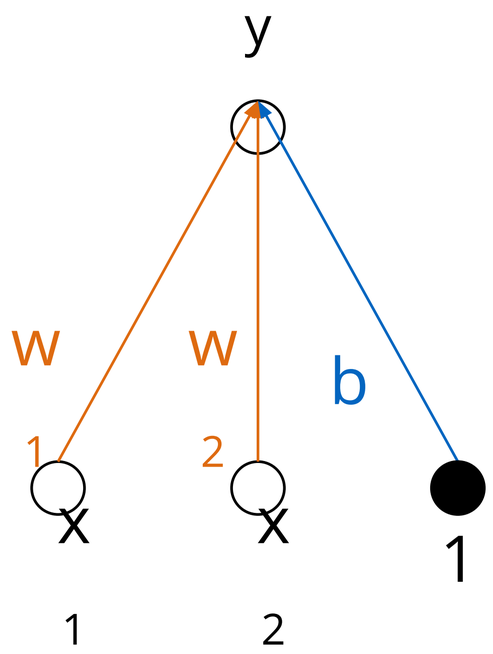
part 4: automatic differentiation

PART ONE: **REVIEW**

RECAP: NEURAL NETWORKS

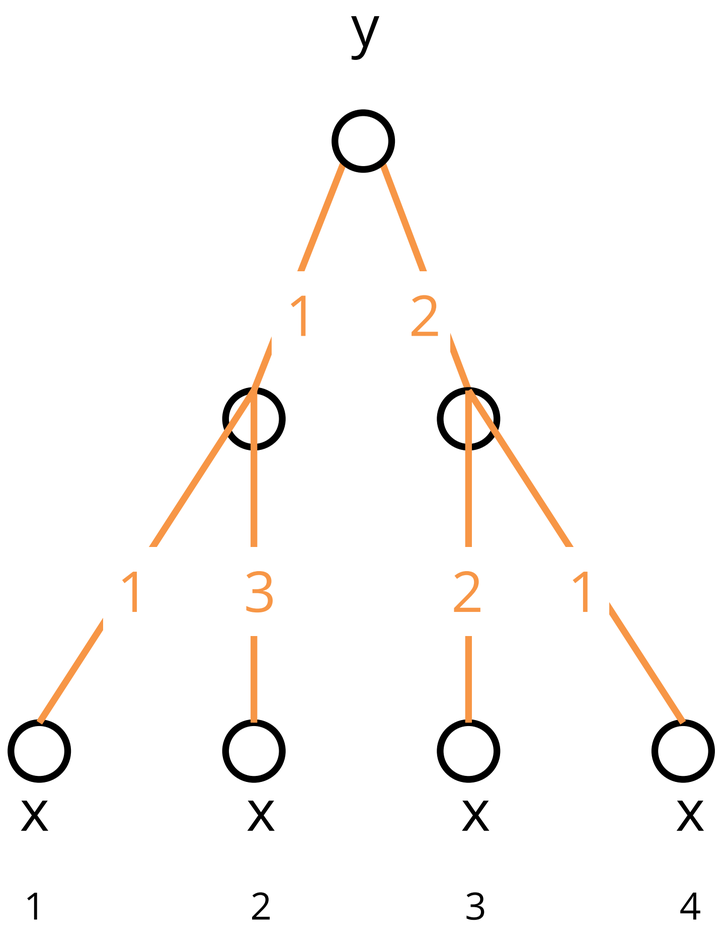


1957: THE PERCEPTRON

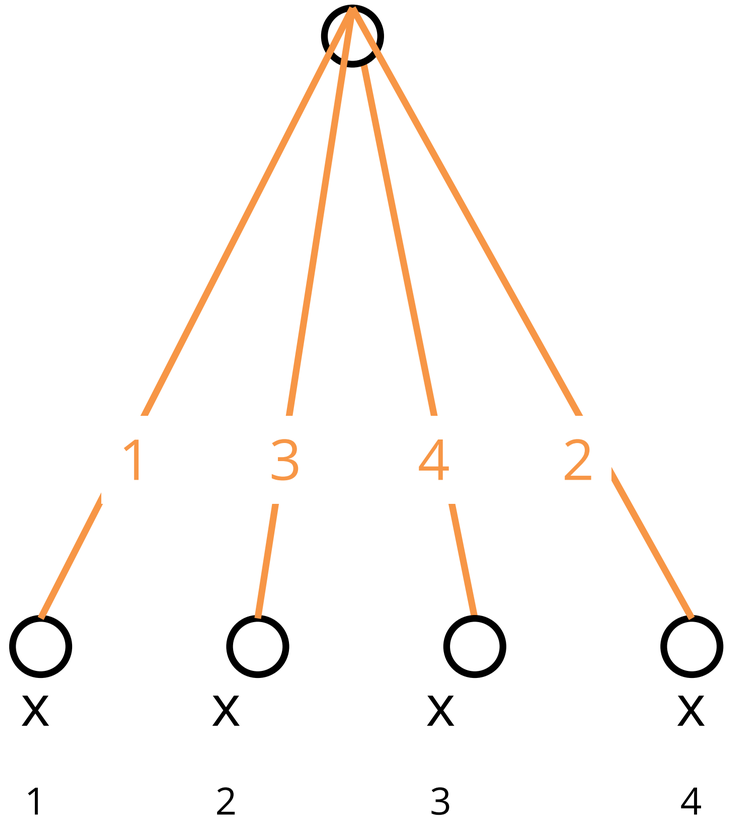


$$y = w_1 x_1 + w_2 x_2 + b$$

PROBLEM: COMPOSING NEURONS

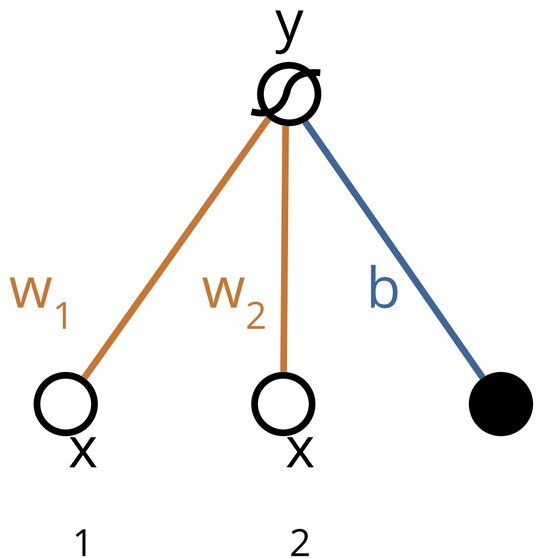


$$y = 1(1x_1 + 3x_2) + 2(2x_3 + 1x_4)$$



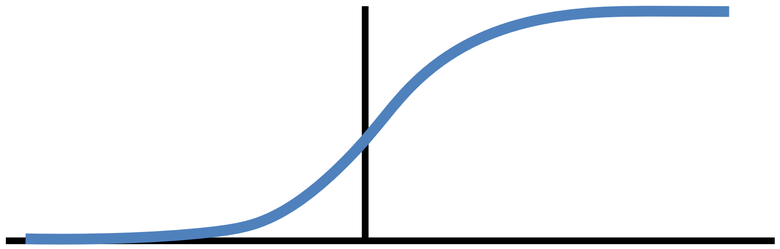
$$y = 1x_1 + 3x_2 + 4x_3 + 2x_4$$

NONLINEARITY



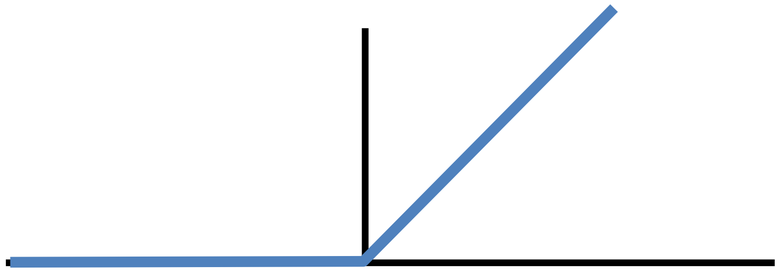
$$y = \sigma(w_1 x_1 + w_2 x_2 + b)$$

sigmoid



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

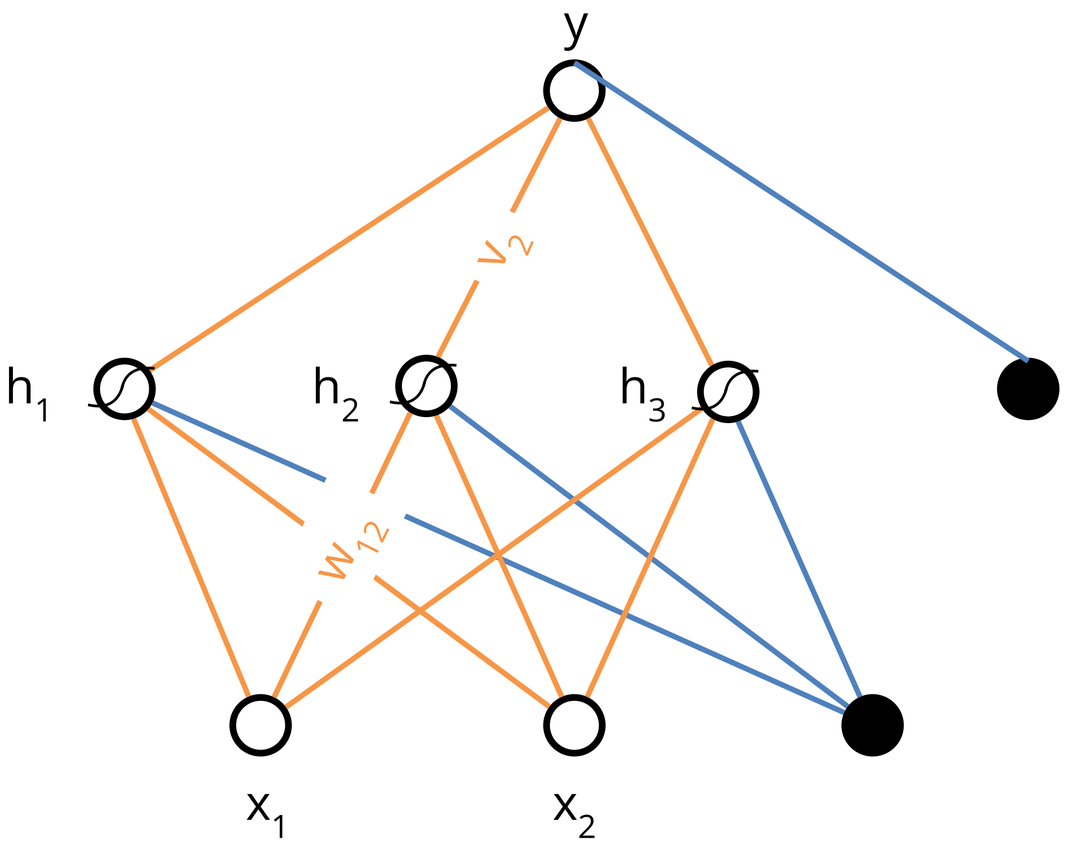
ReLU



$$r(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

OTHER NONLINEARITIES

FEEDFORWARD NETWORK



output layer

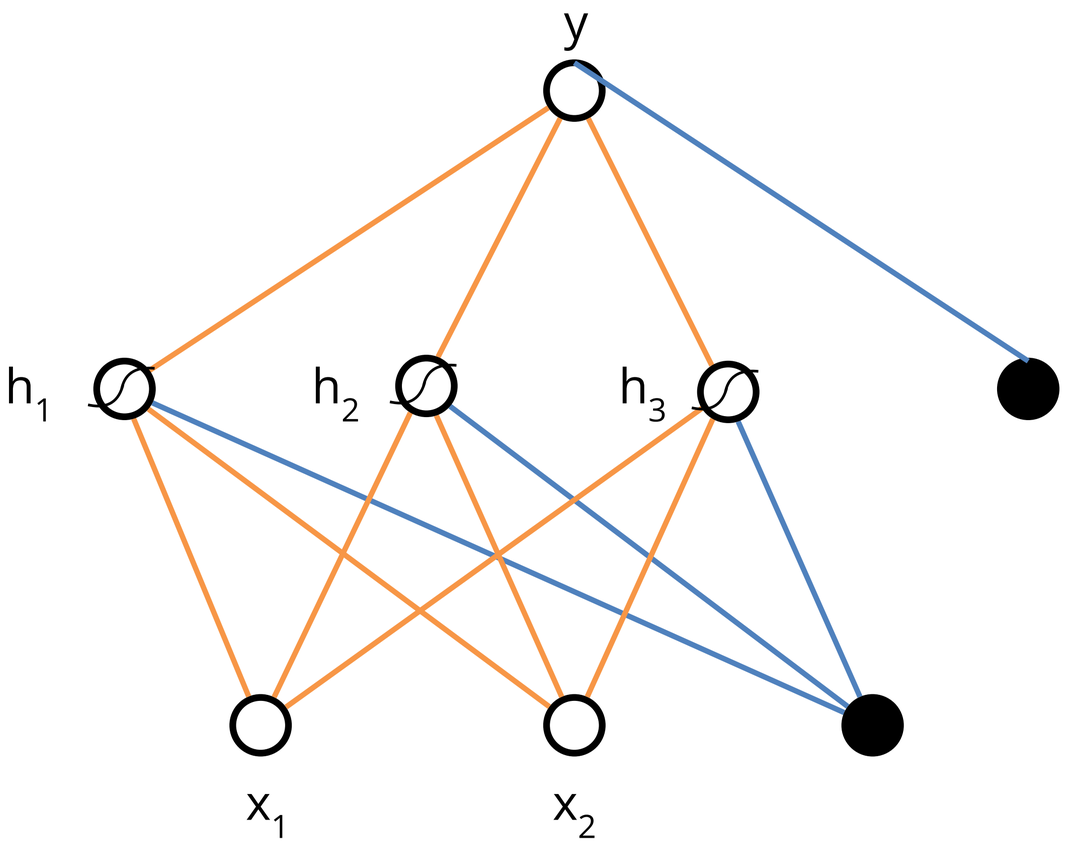
hidden layer

input layer

(features)

aka Multilayer Perceptron (MLP)

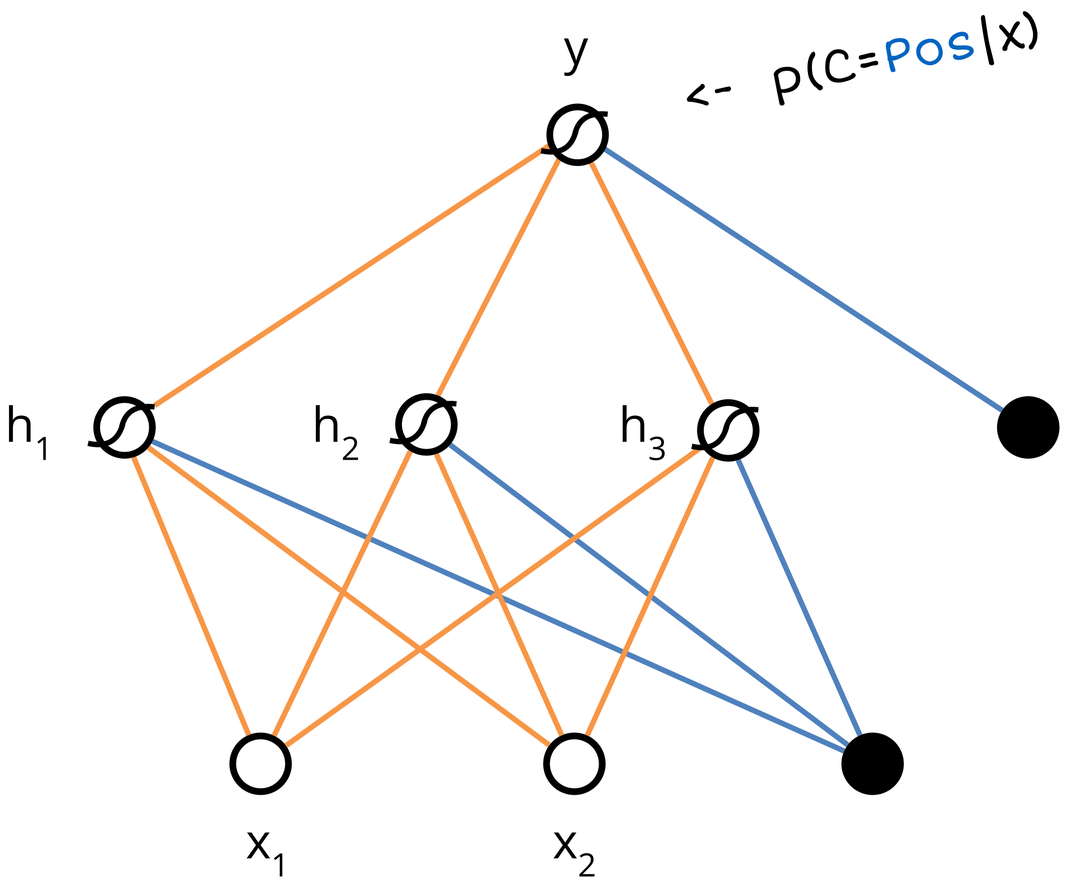
REGRESSION



linear regression

feature extractor

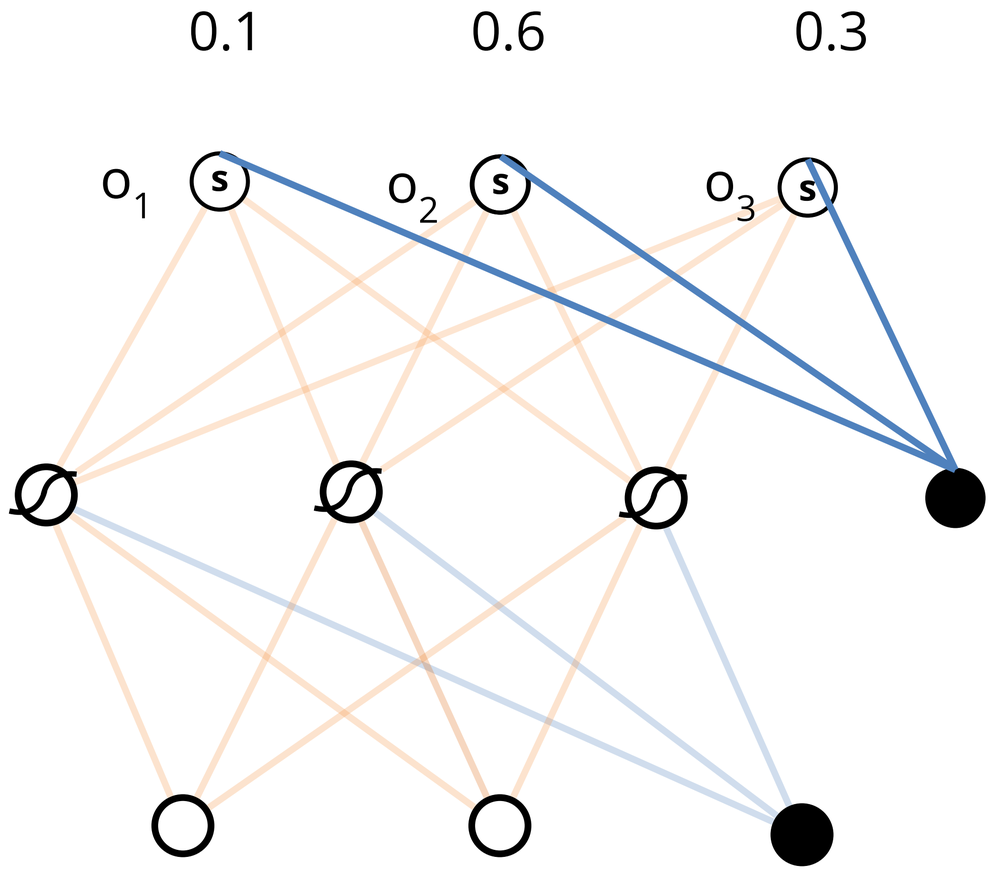
BINARY CLASSIFICATION



logistic regression

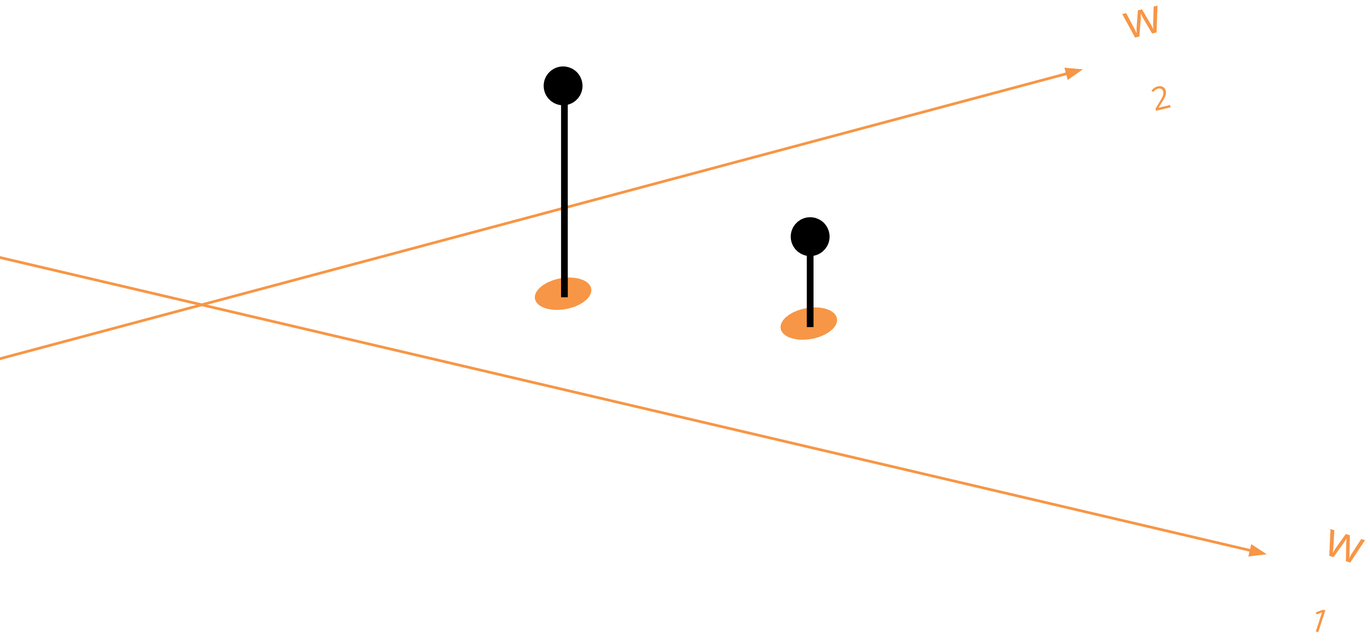
feature extractor

MULTI-CLASS CLASSIFICATION: THE SOFTMAX ACTIVATION



$$y_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

HOW DO WE FIND GOOD WEIGHTS?



$$\text{model}_\theta(\mathbf{x}) = \mathbf{y}$$

$$\text{loss}_{\mathbf{x},\mathbf{t}}(\theta) = \|\text{model}_\theta(\mathbf{x}) - \mathbf{t}\|$$

$$\arg\min_{\theta} \; \text{loss}_{\text{data}}(\theta)$$

SOME COMMON LOSS FUNCTIONS

regression

squared errors $\|\mathbf{y} - \mathbf{t}\| \quad \text{with } \mathbf{y} = \text{model}_\theta(\mathbf{x})$

absolute errors $\|\mathbf{y} - \mathbf{t}\|_1 = \sum_i \text{abs}(y_i - t_i)$

classification

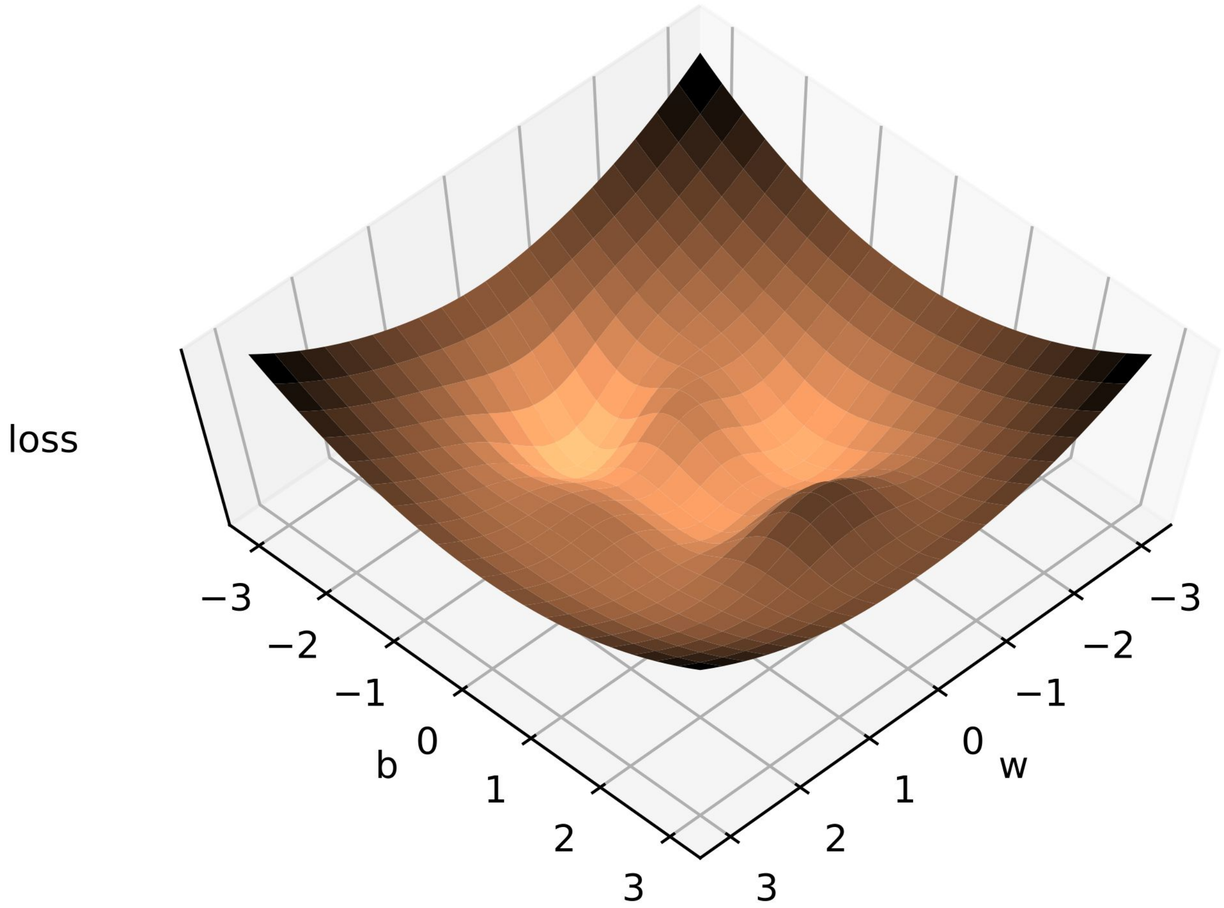
binary cross-entropy $-\log p_\theta(t) \quad \text{with } t \in \{0, 1\}$

cross-entropy $-\log p_\theta(t) \quad \text{with } t \in \{0, \ldots, K\}$

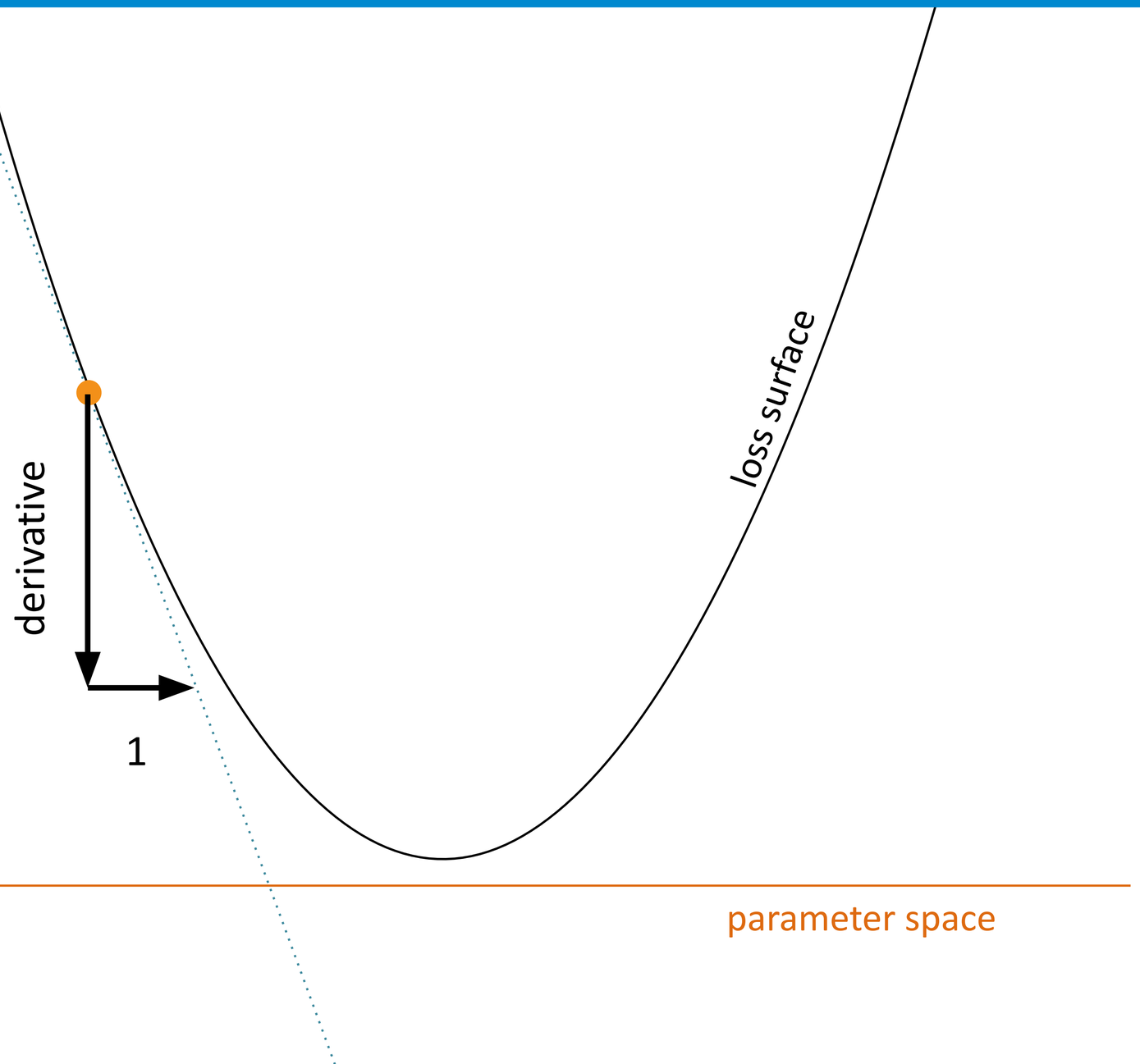
hinge loss $\max(0, 1 - ty) \quad \text{with } t \in \{-1, 1\}$

VU

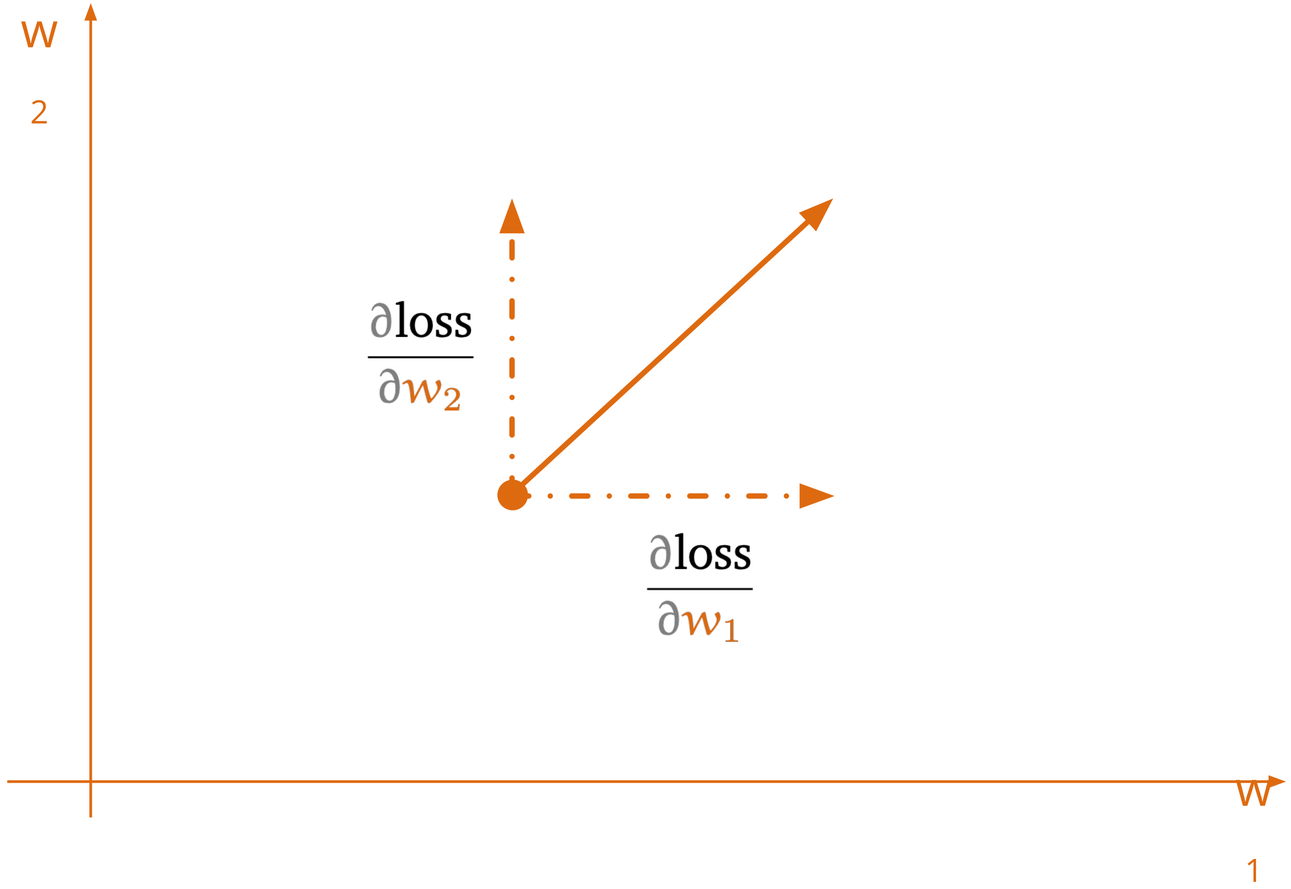
LOSS SURFACE



DERIVATIVE



GRADIENT



$$\nabla_\theta \text{loss}_{x,t}(\theta)$$

STOCHASTIC GRADIENT DESCENT

pick some initial weights θ (for the whole model)

loop:

  for x, t in Data:

    $\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \text{loss}_{\mathbf{x},\mathbf{t}}(\theta)$

stochastic gradient descent: loss over one example per step (loop over data)

minibatch gradient descent: loss over a few examples per step (loop over data)

full-batch gradient descent: loss over the whole data

RECAP

perceptron: linear combination of inputs

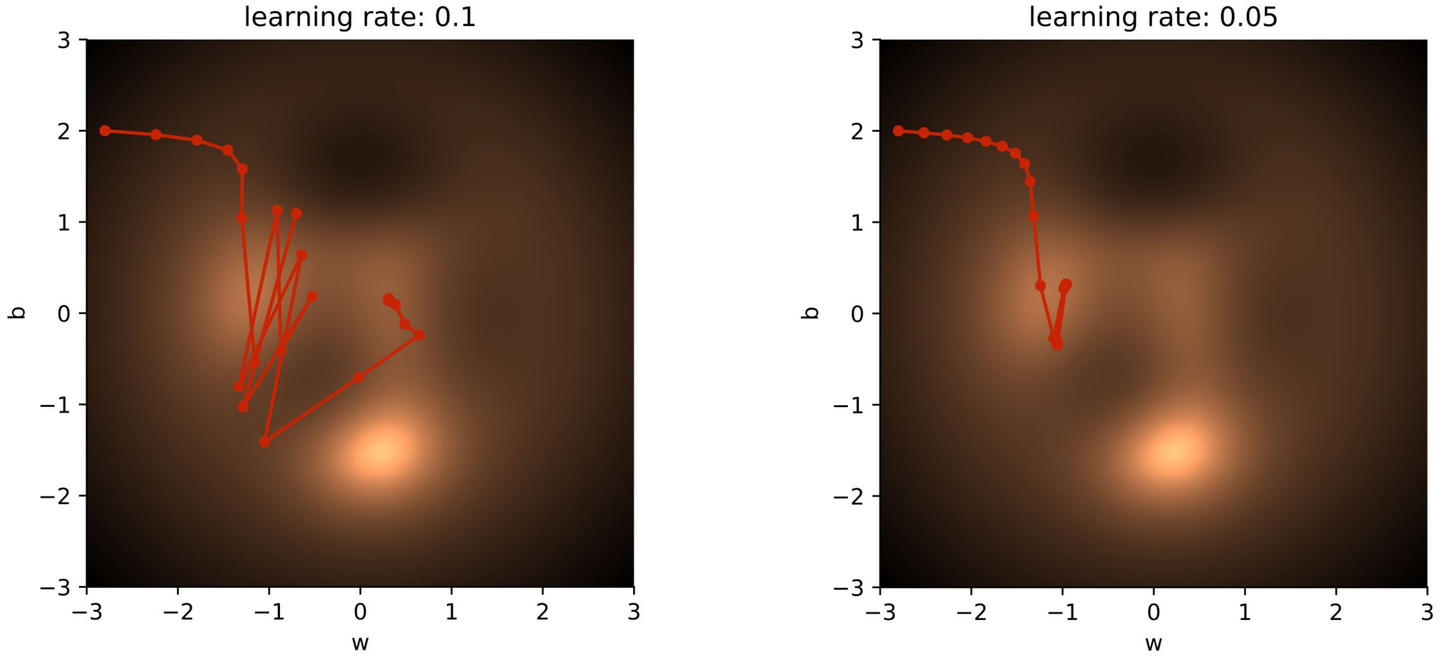
neural network: network of perceptrons, with scalar nonlinearities

training: minibatch gradient descent

But, how do we compute the gradient of a complex neural network?

Next video: **backpropagation**.

VU


learning rate: 0.1
learning rate: 0.05
learning rate: 0.01
learning rate: 0.005
w
b
3
2
1
0
−1
−2
−3

PART ONE

~~Neural Networks: review~~

Scalar backpropagation: definition

Scalar backpropagation on a simple neural network

Lecture 2: Backpropagation

Peter Bloem

Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

PART TWO: SCALAR BACKPROPAGATION

How do we work out the gradient for a neural network?

VU

input
Conv 7x7+2(S)
MaxPool 3x3+2(S)
LocalRespNorm
Conv 1x1+1(V)
Conv 3x3+1(S)
LocalRespNorm
MaxPool 3x3+2(S)
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
Conv 1x1+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
Conv 1x1+1(S)
DepthConcat
MaxPool 3x3+2(S)
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
Conv 1x1+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
AveragePool 5x5+3(V)
Conv 1x1+1(S)
FC
FC
SoftmaxActivation
softmax0
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
Conv 1x1+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
AveragePool 5x5+3(V)
Conv 1x1+1(S)
FC
FC
SoftmaxActivation
softmax1
MaxPool 3x3+2(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
Conv 1x1+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
MaxPool 3x3+1(S)
Conv 1x1+1(S)
DepthConcat
AveragePool 7x7+1(V)
FC
SoftmaxActivation
softmax2

GRADIENT COMPUTATION: THE SYMBOLIC APPROACH

WolframAlpha computational intelligence

derivative of l(w) = (d + z * 1/(1+e^-(c + v*(b + 1/(1+e^-(x*w))))))

Extended Keyboard Upload Examples Random

Derivative:

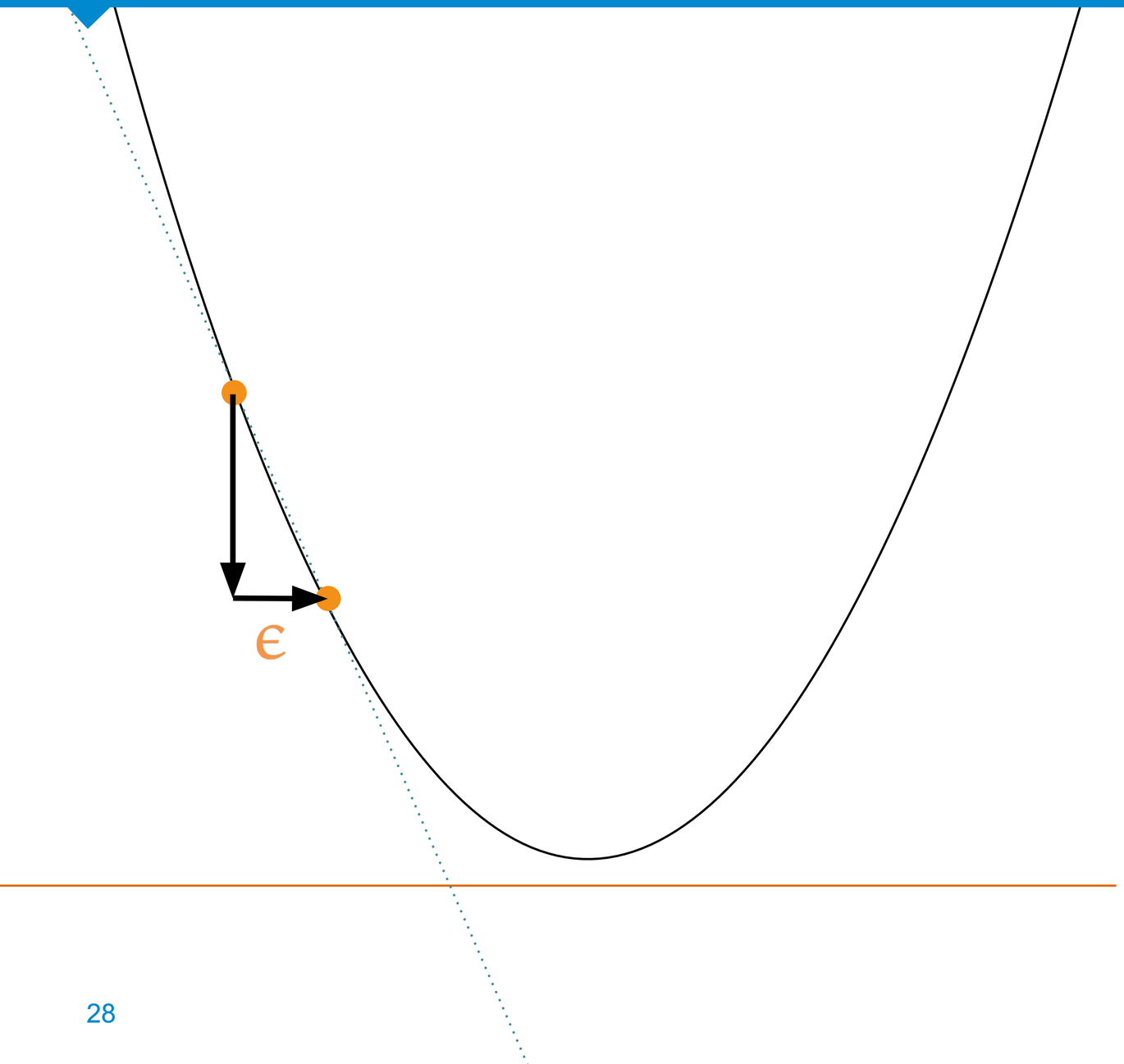
Approximate form Step-by-step solution

$$\frac{\partial}{\partial w}\left(l(w) = d + \frac{z}{1 + e^{-(c+v(b+1/(1+e^{-(x\,w)})))}}\right) = \frac{\partial \text{Hold}\left[\frac{z}{e^{-(v(b+1/(e^{-(w\,x)}+1))+c)}+1} + d\right]}{\partial w}$$

Alternate form:

$$l'(w) = \frac{v\,x\,z\,e^{w\,x}}{(e^{w\,x}+1)^2\left(e^{v(b+1/(e^{-w\,x}+1))+c}+1\right)} - \frac{v\,x\,z\,e^{w\,x}}{(e^{w\,x}+1)^2\left(e^{v(b+1/(e^{-w\,x}+1))+c}+1\right)^2}$$

GRADIENT COMPUTATION: THE NUMERIC APPROACH



$$\text{deriv} \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

the "method of finite differences"

BACKPROPAGATION: THE MIDDLE GROUND

Work out parts of the derivative **symbolically**

chain these together in a **numeric computation**.

secret ingredient: **the chain rule**.

THE CHAIN RULE

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

INTUITION FOR THE CHAIN RULE

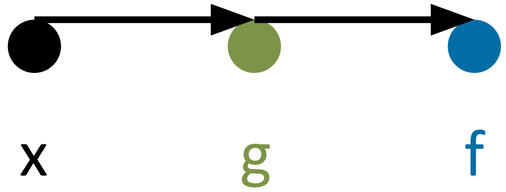
slope of f over g

$$f(g) = s_f g + b_f$$

$$g(x) = s_g x + b_g$$

$$\text{with } s_f = \frac{\partial f}{\partial g}$$

$$s_g = \frac{\partial g}{\partial x}$$



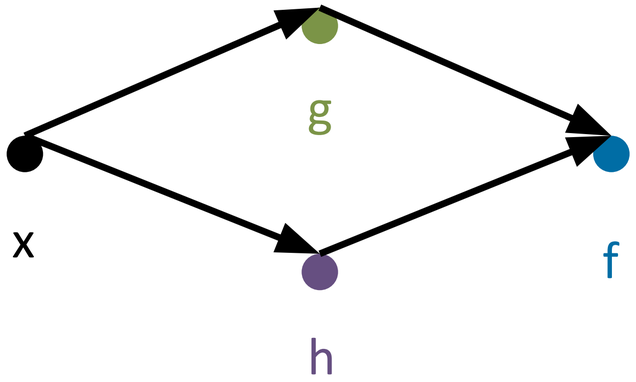
$$f(g(x)) = s_f(s_g x + b_g) + b_f$$

$$= s_f s_g x + s_f b_g + s_g$$

$$\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

VU

MULTIVARIATE CHAIN RULE



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

INTUITION FOR THE MULTIVARIATE CHAIN RULE

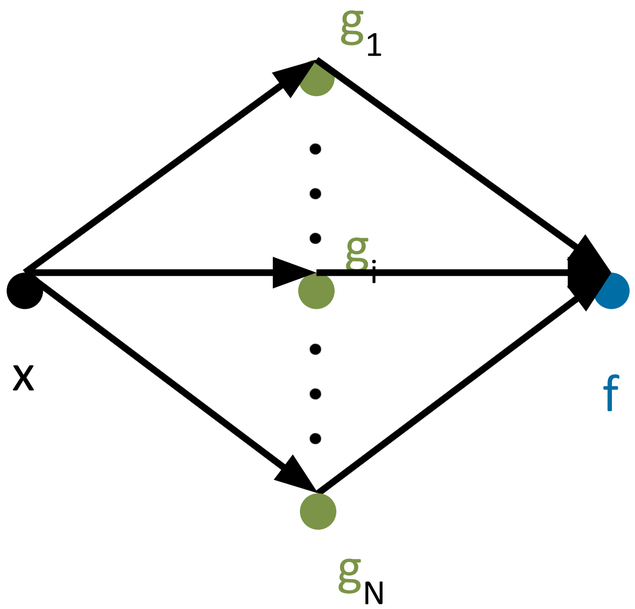
$$f(g,h) = s_1 g + s_2 h + b_f$$
$$g(x) = s_g x + b_g$$
$$h(x) = s_h x + b_h$$

with $s_1 = \frac{\partial f}{\partial g}$ $s_2 = \frac{\partial f}{\partial h}$ $s_g = \frac{\partial g}{\partial x}$ $s_h = \frac{\partial h}{\partial x}$

$$\begin{aligned} f(g(x),h(x)) &= s_1(s_g x + b_g) + s_2(s_f x + b_f) + b_f \\ &= s_1 s_g x + s_1 b_g + s_2 s_f x + s_2 b_f + b_f \\ &= (s_1 s_g + s_1 s_h)x + s_2 b_g + s_2 b_f + b_f \end{aligned}$$

$$\frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

MULTIVARIATE CHAIN RULE



$$\frac{\partial f}{\partial x} = \sum_i \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

EXAMPLE

$$f(x) = \frac{2}{\sin(e^{-x})}$$

ITERATING THE CHAIN RULE

$$f(x) = \frac{2}{\sin(e^{-x})}$$

$$f(x) = d(c(b(a(x))))$$

operations:

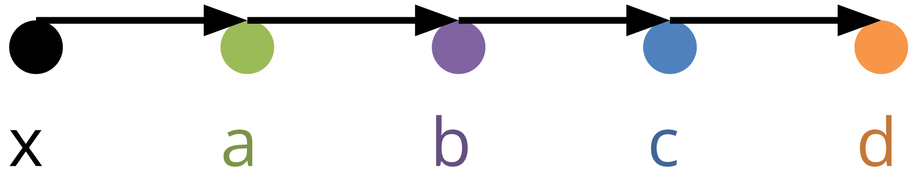
$$d(c) = \frac{2}{c}$$

$$c(b) = \sin b$$

$$b(a) = e^{a}$$

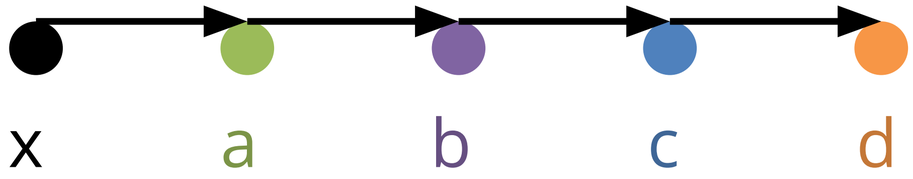
$$a(x) = -x$$

computation graph:



ITERATING THE CHAIN RULE

$$
\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial d}{\partial x} \\
&= \frac{\partial d}{\partial c}\frac{\partial c}{\partial x} \\
&= \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial x} \\
&= \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}
\end{aligned}
$$



GLOBAL AND LOCAL DERIVATIVES

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

global derivative

local derivatives

BACKPROPAGATION

The **BACKPROPAGATION** algorithm:

- break your computation up into a sequence of operations
  what counts as an operation is up to you
- work out the **local derivatives** symbolically.
- compute the **global derivative** numerically
  by computing the local derivatives and multiplying them

VU

WORK OUT THE LOCAL DERIVATIVES SYMBOLICALLY

$$f(x) = \frac{2}{\sin(e^{-x})}$$

operations:

$$d(c) = \frac{2}{c}$$

$$c(b) = \sin b$$

$$b(a) = e^{a}$$

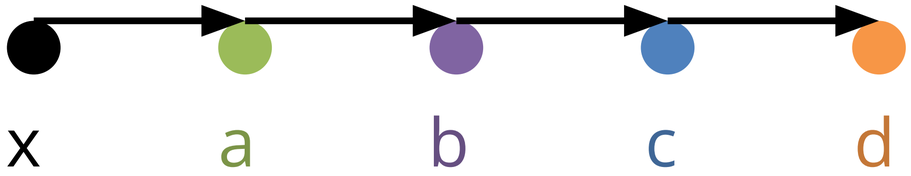
$$a(x) = -x$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

$$= -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$

COMPUTE A FORWARD PASS (X = -4.499)

$$f(-4.499) = 2$$



$$d = \frac{2}{c} = 2$$

$$c = \sin b = 1$$

$$b = e^{a} = 90$$

$$a = -x = 4.499$$

COMPUTE A FORWARD PASS (X = -4.499)

$$f(-4.499) = 2$$

$$d = \frac{2}{c} = 2$$

$$c = \sin b = 1$$

$$b = e^{a} = 90$$

$$a = -x = 4.499$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$

$$= -\frac{2}{1^2} \cdot \cos 90 \cdot e^{4.499} \cdot -1$$

$$= -\frac{1}{2} \cdot 0 \cdot 90 \cdot -1 = 0$$

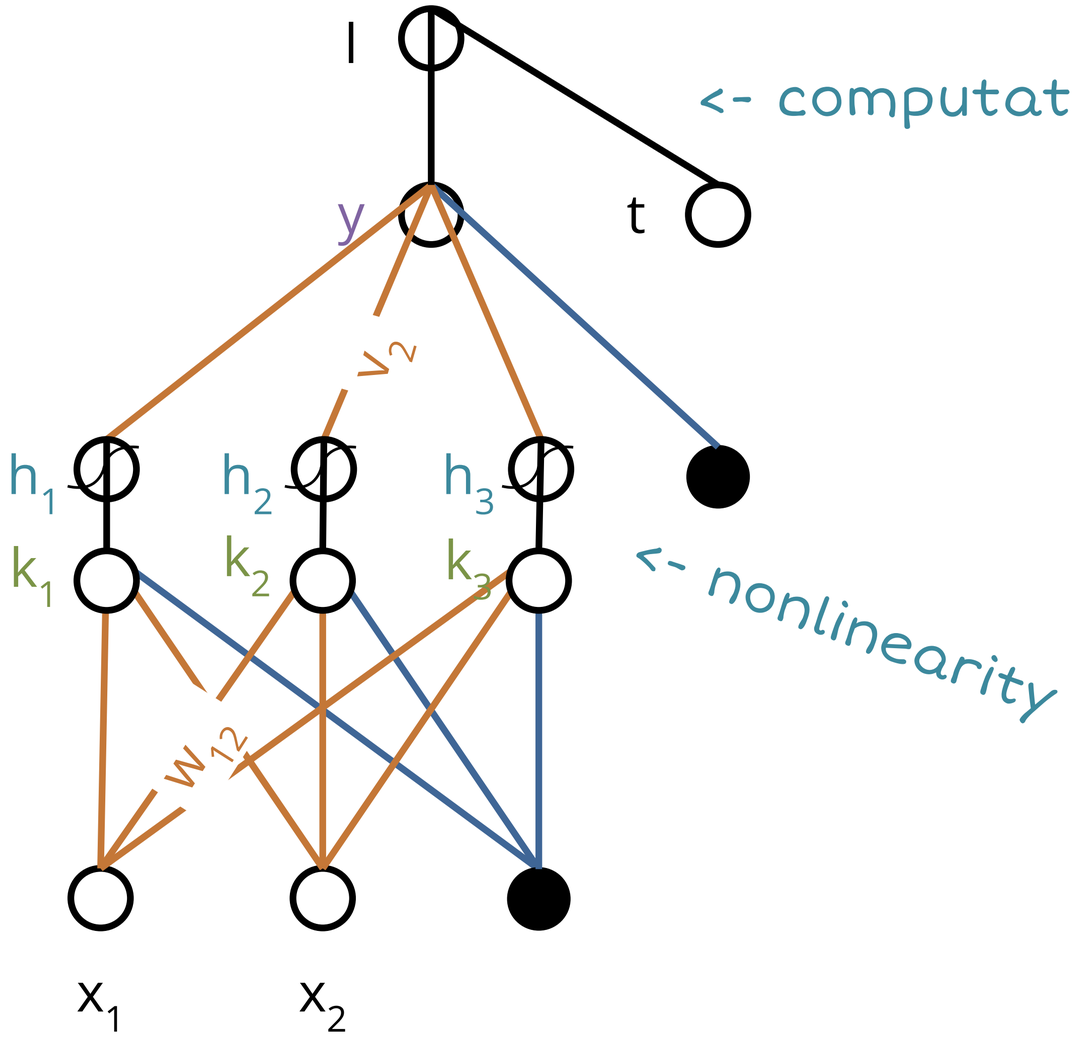
BACKPROPAGATION

The **BACKPROPAGATION** algorithm:

- break your computation up into a sequence of operations
  what counts as an operation is up to you
- work out the **local derivatives** symbolically.
- compute the **global derivative** numerically
  by computing the local derivatives and multiplying them
- Much more accurate than finite differences
  only source of inaccuracy is the numeric computation of the operations.
- Much faster than symbolic differentiation
  The backward pass has (broadly) the sample complexity as the forward.

VU

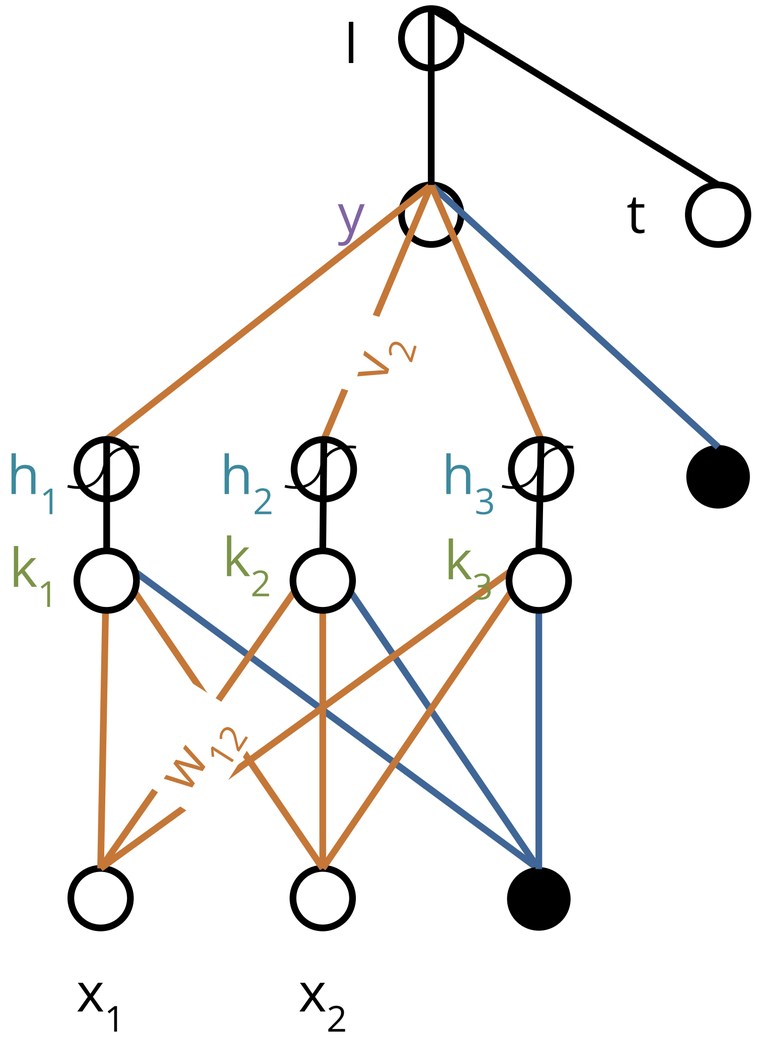
BRACKPROPAGATION IN A FEEDFORWARD NETWORK



<- computation of loss

<- nonlinearity

BRACKPROPAGATION IN A FEEDFORWARD NETWORK



$$\frac{\partial l}{\partial v_2}, \frac{\partial l}{\partial w_{12}}$$

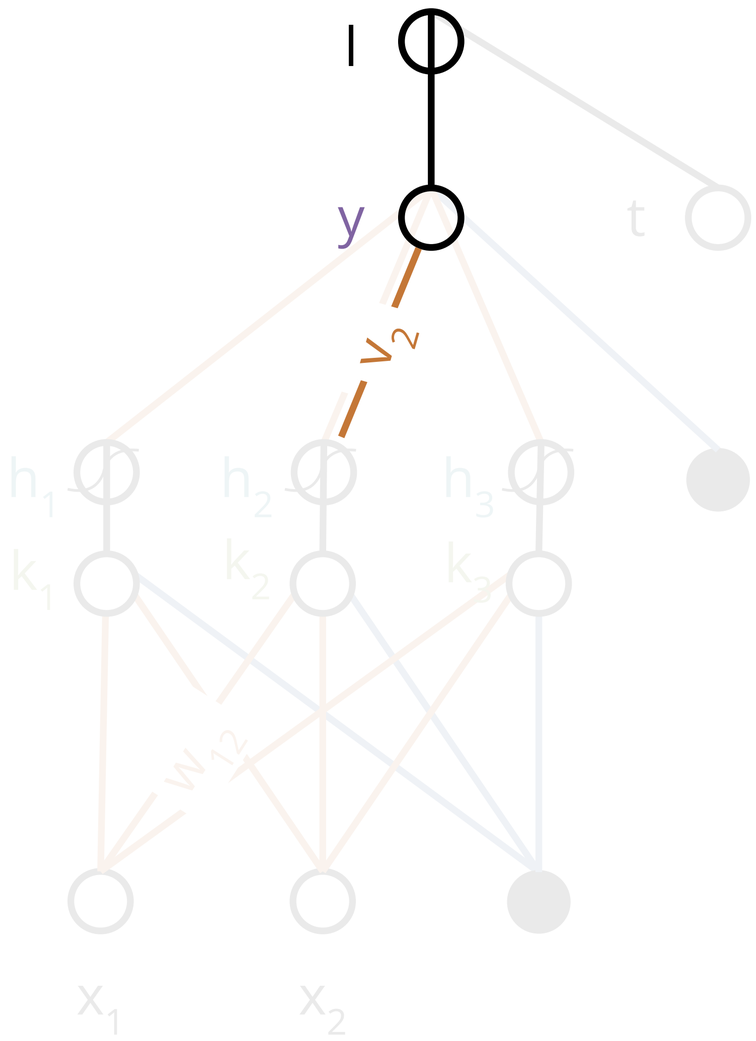
$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

$$h_2 = \frac{1}{1 + \exp(-k_2)}$$

$$k_2 = w_{21} x_1 + w_{22} x_2 + b_x$$

BACKPROPAGATION IN A FEEDFORWARD NETWORK



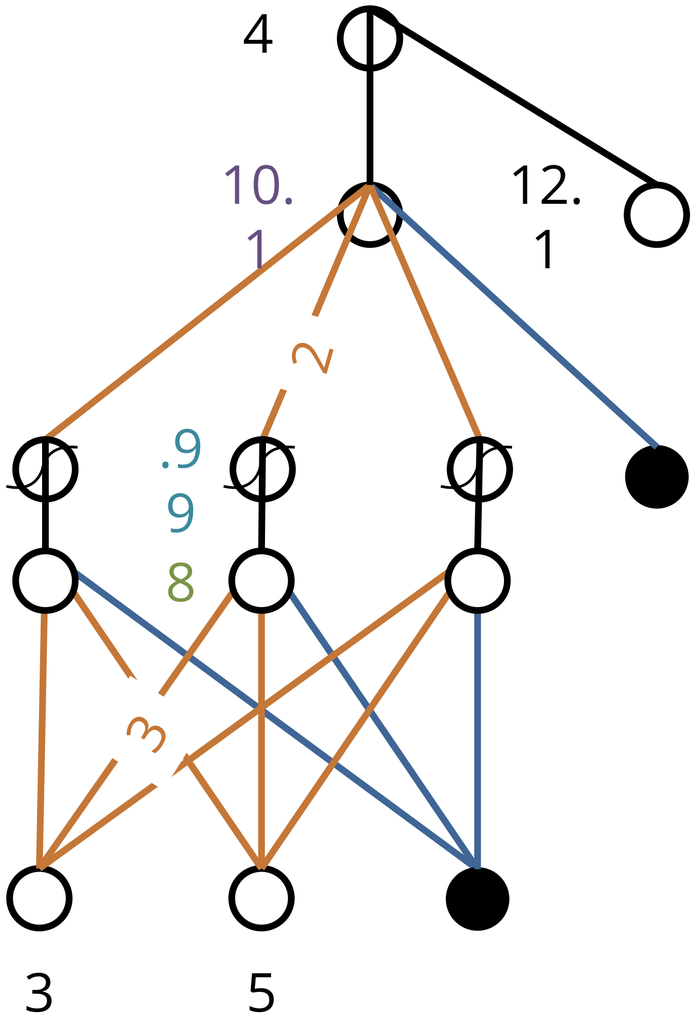
$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial v_2}$$

$$= 2(y - t) \cdot h_2$$

BACKPROPAGATION IN A FEEDFORWARD NETWORK



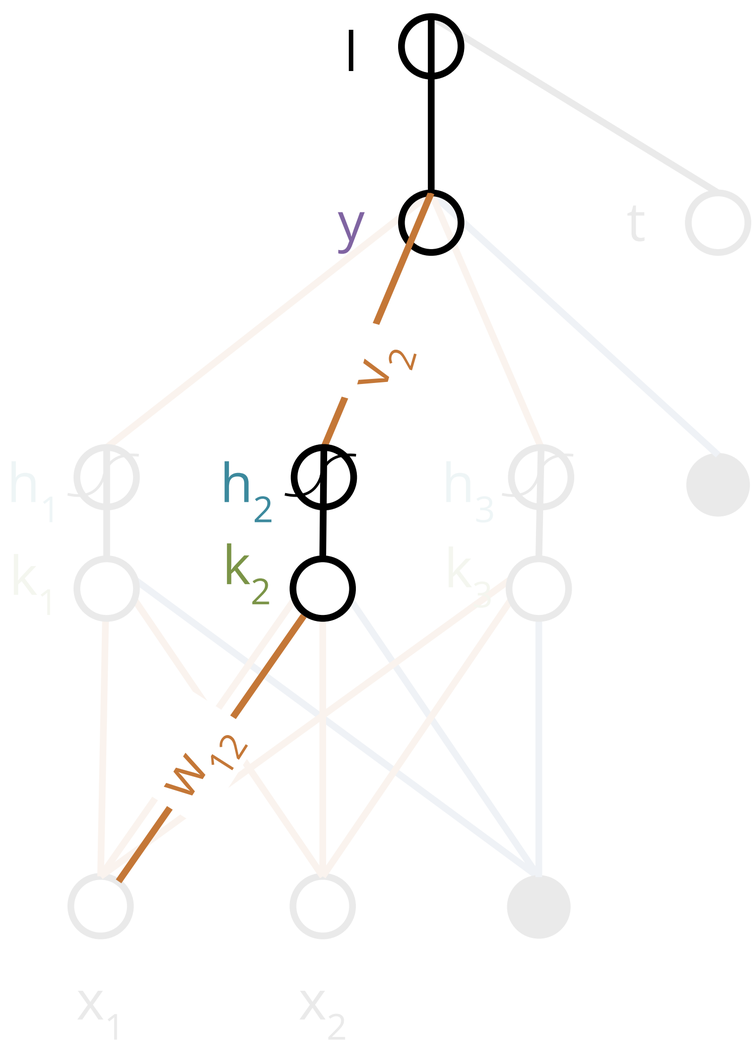
$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial v_2}$$

$$= 2(y - t) \cdot h_2$$

$$= 2(10.1 - 12.1) \cdot .99 = -3.96$$

$$v_2 \leftarrow v_2 - \alpha \cdot -3.96$$

BACKPROPAGATION IN A FEEDFORWARD NETWORK



$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

$$h_2 = \frac{1}{1 + \exp(-k_2)}$$

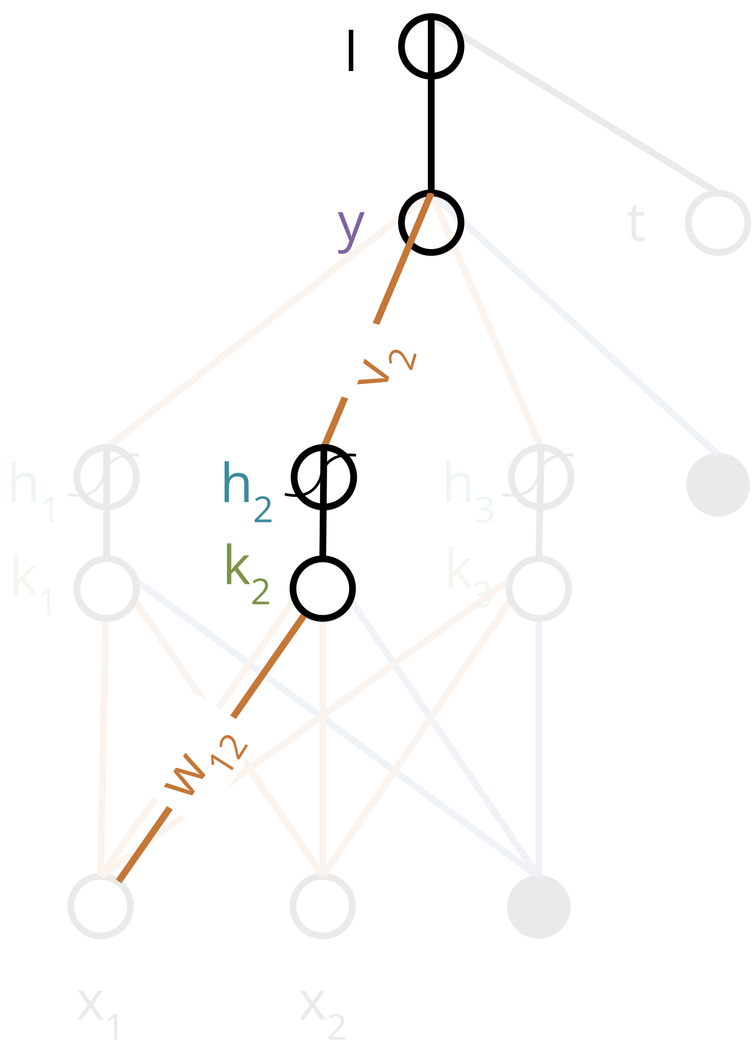
$$k_2 = w_{21} x_1 + w_{22} x_2 + b_x$$

$$\frac{\partial l}{\partial w_{12}} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial k_2} \frac{\partial k_2}{\partial w_{12}}$$

$$= 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

derivative of sigmoid: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

BACKPROPAGATION



$$\frac{\partial l}{\partial y}$$

$$\frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2} = \frac{\partial l}{\partial h_2}$$

$$\frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial k_2} = \frac{\partial l}{\partial k_2}$$

$$\frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial k_2}\frac{\partial k_2}{\partial w_{12}} = \frac{\partial l}{\partial w_{12}}$$

BACKPROPAGATION

The **BACKPROPAGATION** algorithm:

- break your computation up into a sequence of operations
  what counts as an operation is up to you
- work out the **local derivatives** symbolically.
- compute the **global derivative** numerically
  by computing the local derivatives and multiplying them
- Walk backward from the loss, *accumulating* the derivatives.

BUILDING SOME INTUITION

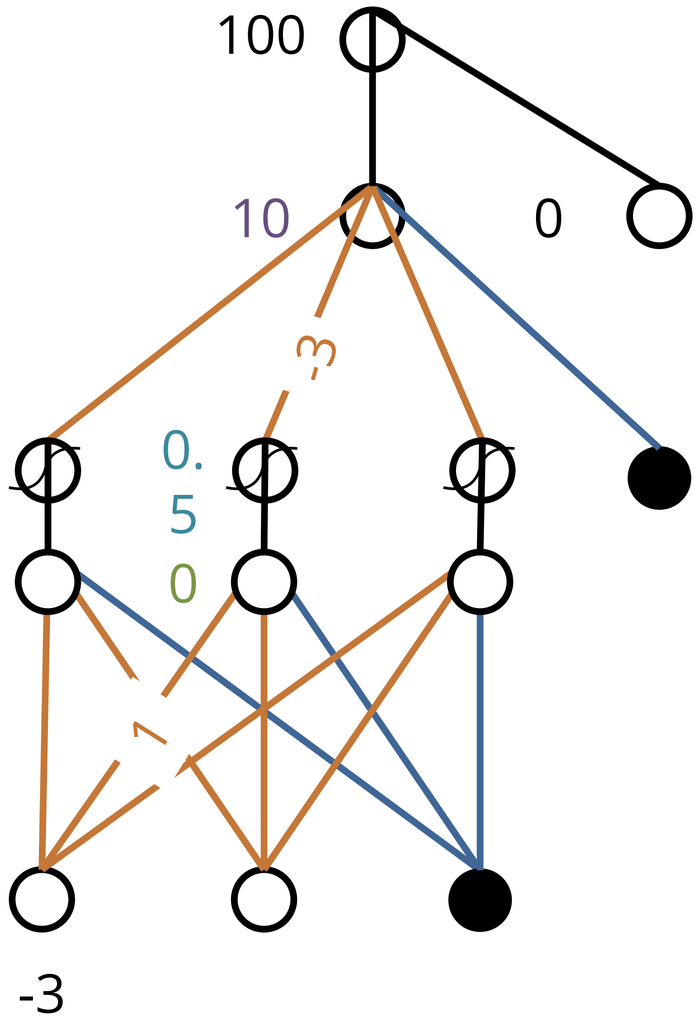
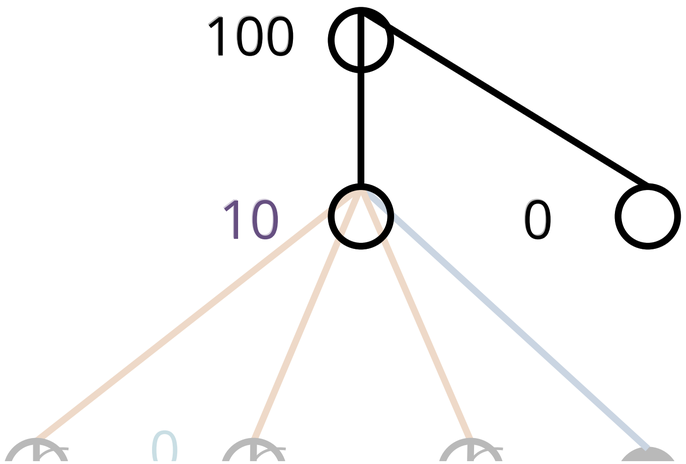
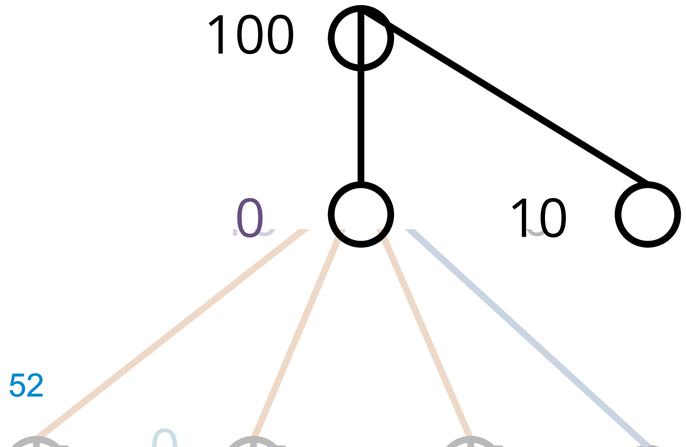


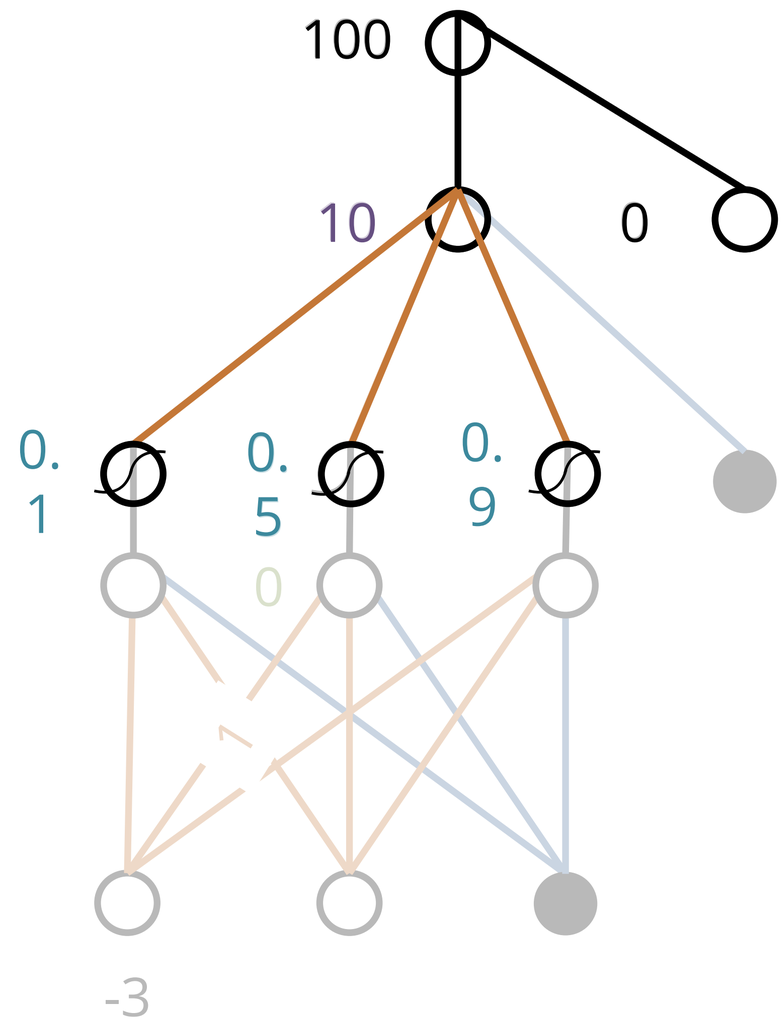
IMAGE THAT y IS A PARAMETER



$$y \leftarrow y - \alpha \cdot 2(y - t)$$
$$= y - \alpha \cdot 20$$



$$y \leftarrow y - \alpha \cdot 2(y - t)$$
$$= y + \alpha \cdot 20$$



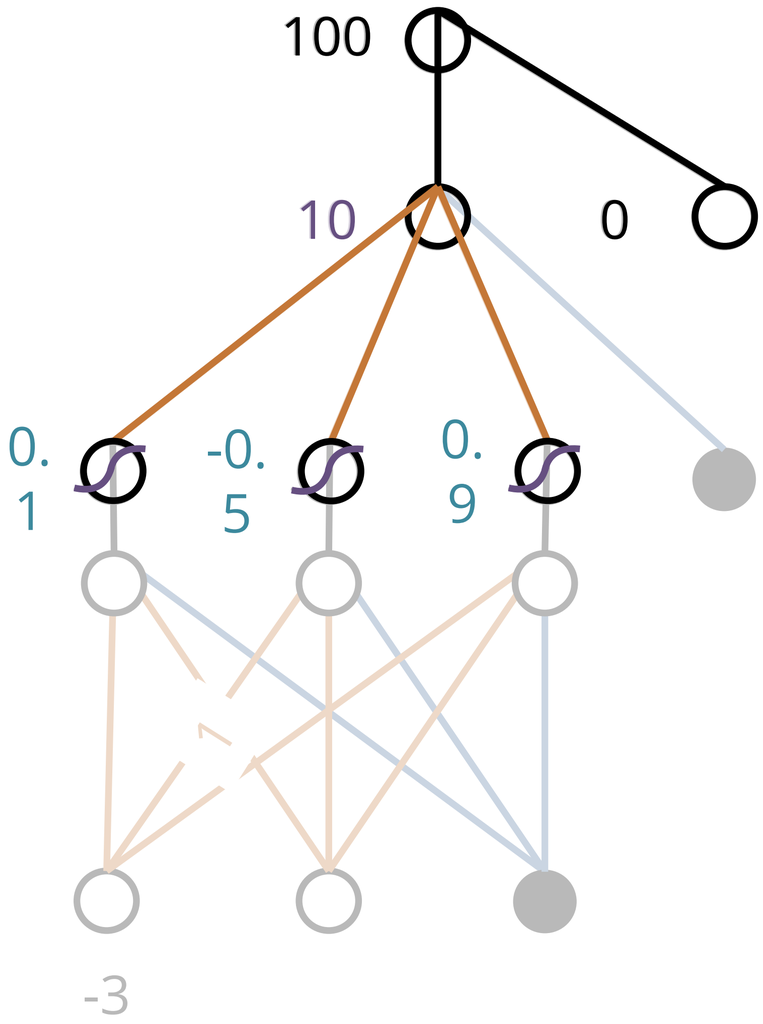
$$v_2 \leftarrow v_2 - \alpha \cdot 2(y - t)h_2$$

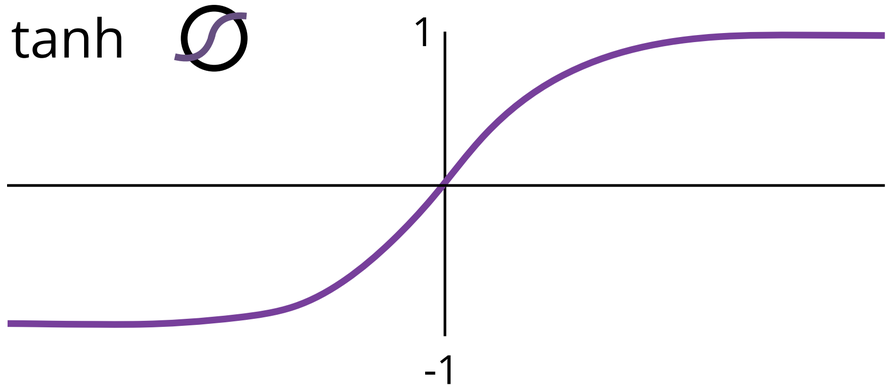
$$v_1 \leftarrow v_1 - \alpha \cdot 20 \cdot 0.1$$

$$v_2 \leftarrow v_2 - \alpha \cdot 20 \cdot 0.5$$

$$v_3 \leftarrow v_3 - \alpha \cdot 20 \cdot 0.9$$

NEGATIVE ACTIVATION

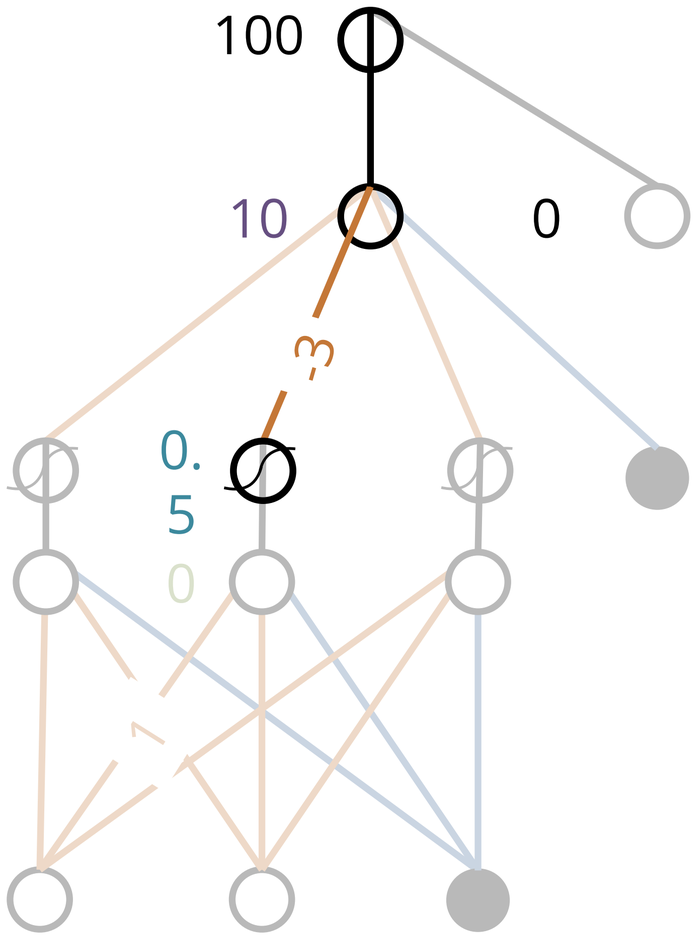




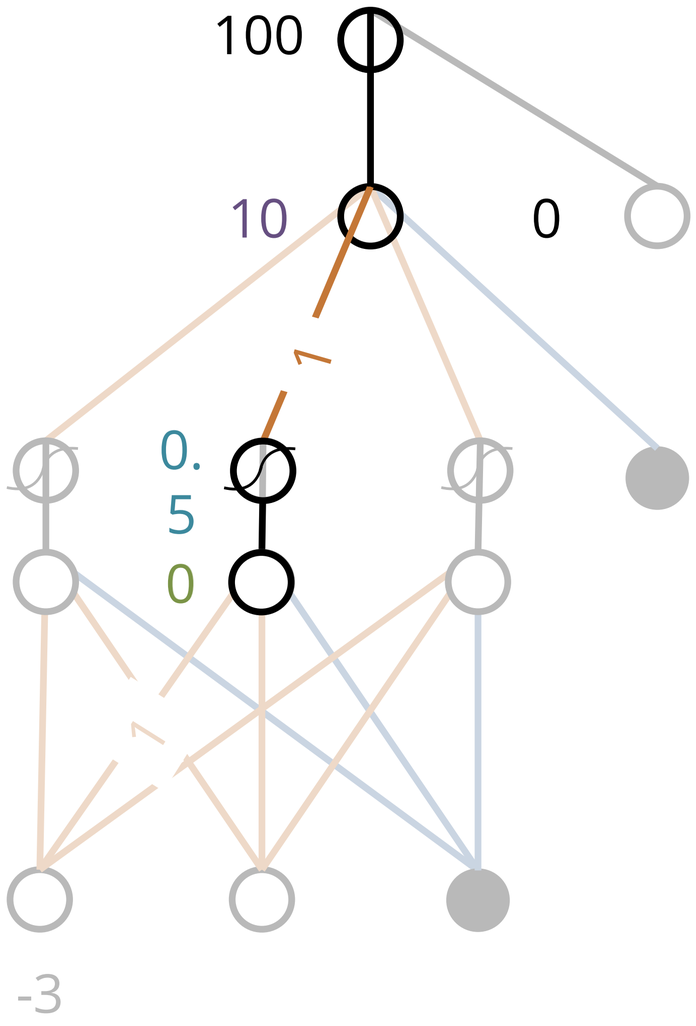
$$\nu_1 \leftarrow \nu_1 - \alpha \cdot 20 \cdot 0.1$$

$$\nu_2 \leftarrow \nu_2 + \alpha \cdot 20 \cdot 0.5$$

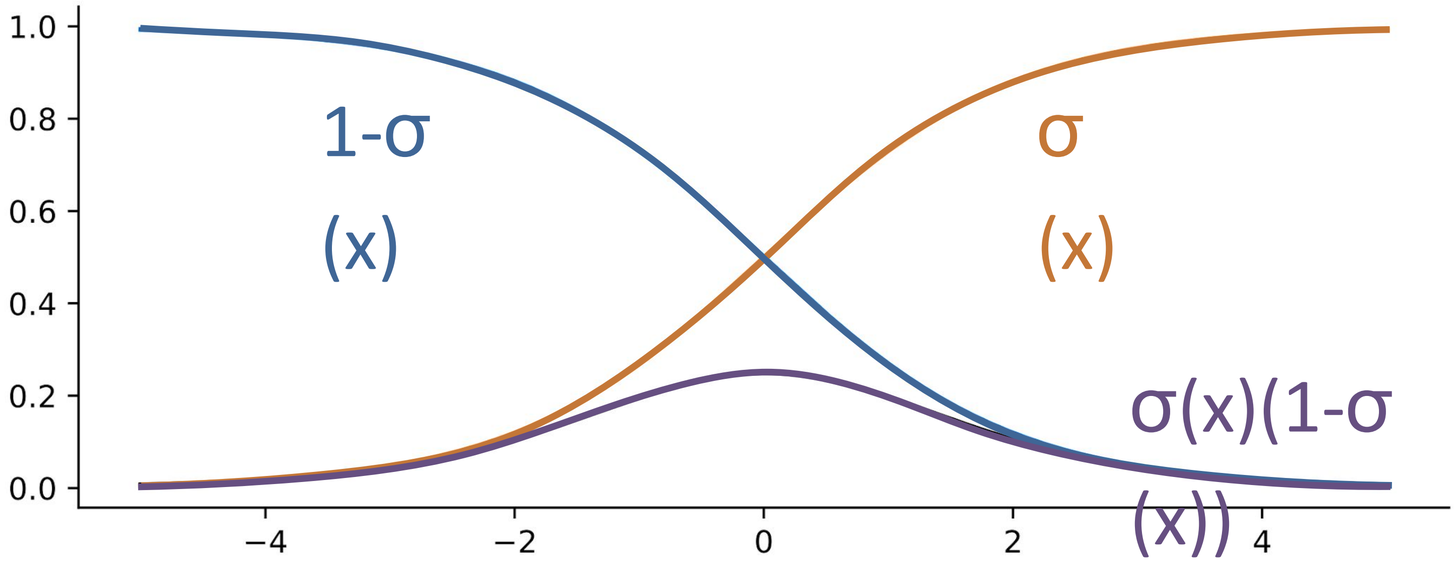
$$\nu_3 \leftarrow \nu_3 - \alpha \cdot 20 \cdot 0.9$$

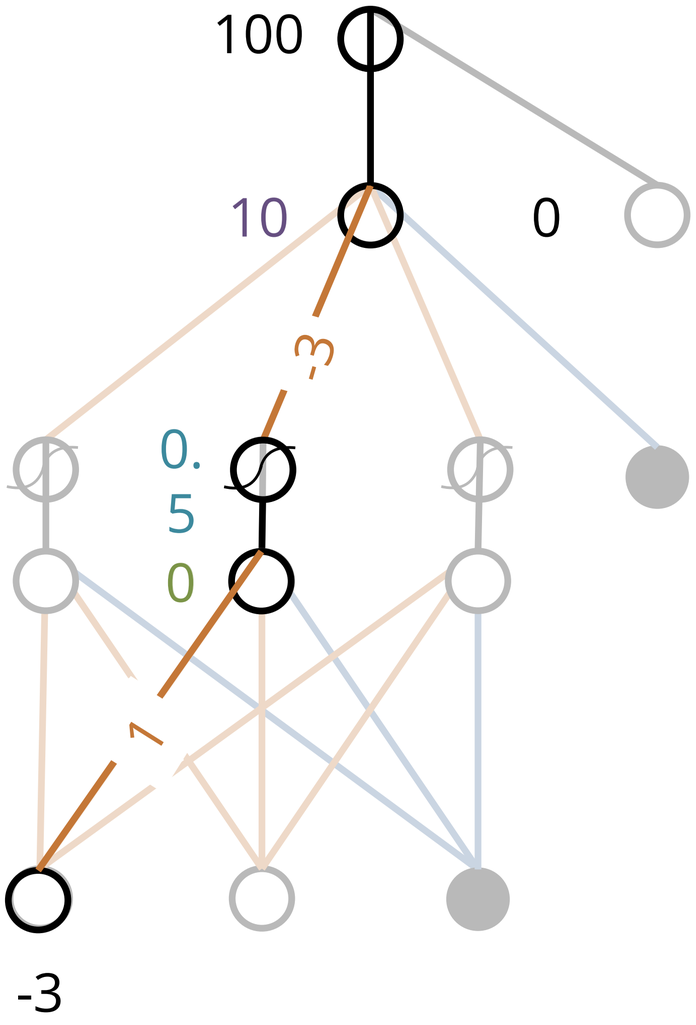


$$\mathbf{h}_2 \leftarrow \mathbf{h}_2 - \alpha \cdot 2(\mathbf{y} - \mathbf{t})\mathbf{v}_2$$
$$= \mathbf{h}_2 + \alpha \cdot 20 \cdot 3$$



$$k_2 \leftarrow k_2 - \alpha \cdot 2(y - t)v_2 h_2(1 - h_2)$$





$$w_{12} \leftarrow w_{12} - \alpha \cdot 2(y - t) v_2 h_2 (1 - h_2) x_1$$

$$= w_{12} - \alpha \cdot 20 \cdot -3 \cdot \frac{1}{4} \cdot -3$$

$$= w_{12} - \alpha \cdot 20 \cdot 3 \cdot \frac{1}{4} \cdot 3$$

GLOBAL AND LOCAL AND ACCUMULATED DERIVATIVES

$$\frac{\partial l}{\partial w_{12}} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial k_2}\frac{\partial k_2}{\partial w_{12}}$$

global derivative

local derivatives

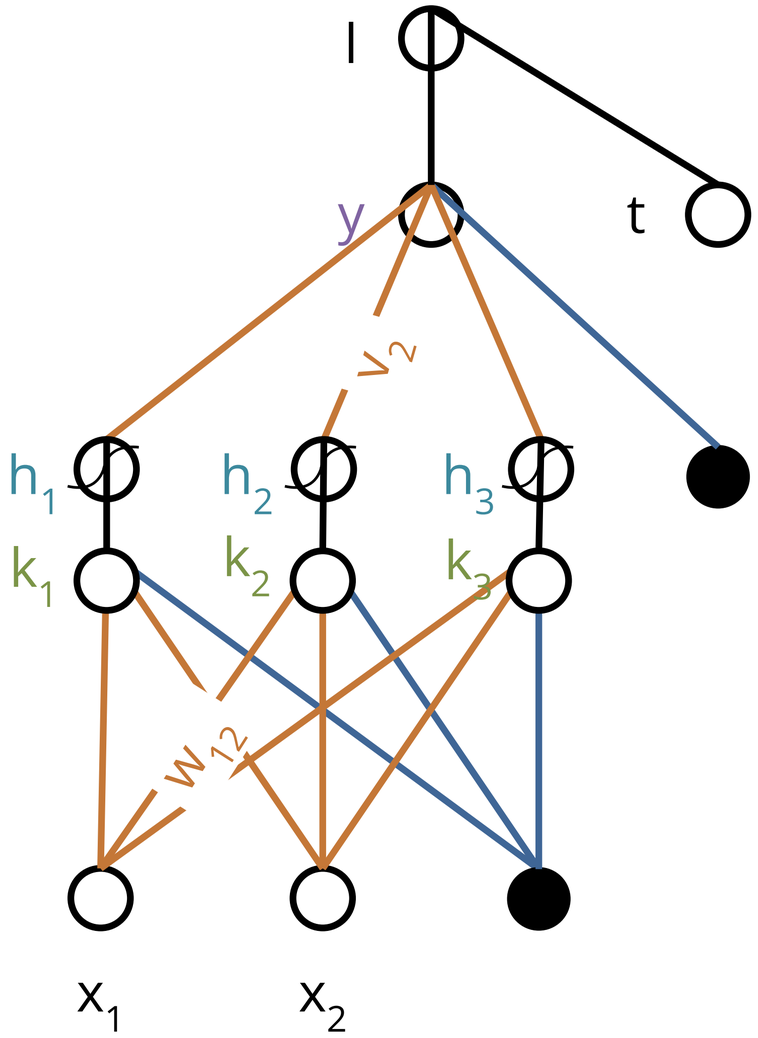
$\frac{\partial l}{\partial k_2}$

$\frac{\partial l}{\partial h_2}$

cumulative derivatives

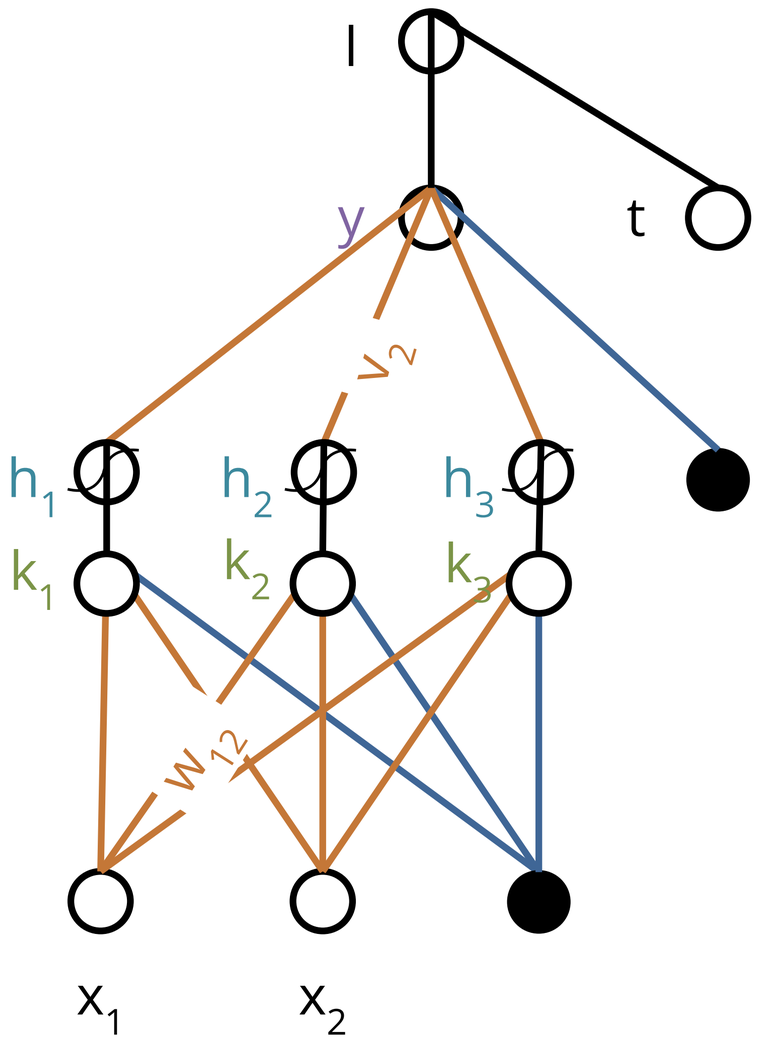
$\frac{\partial l}{\partial y}$

FORWARD PASS IN PSEUDOCODE



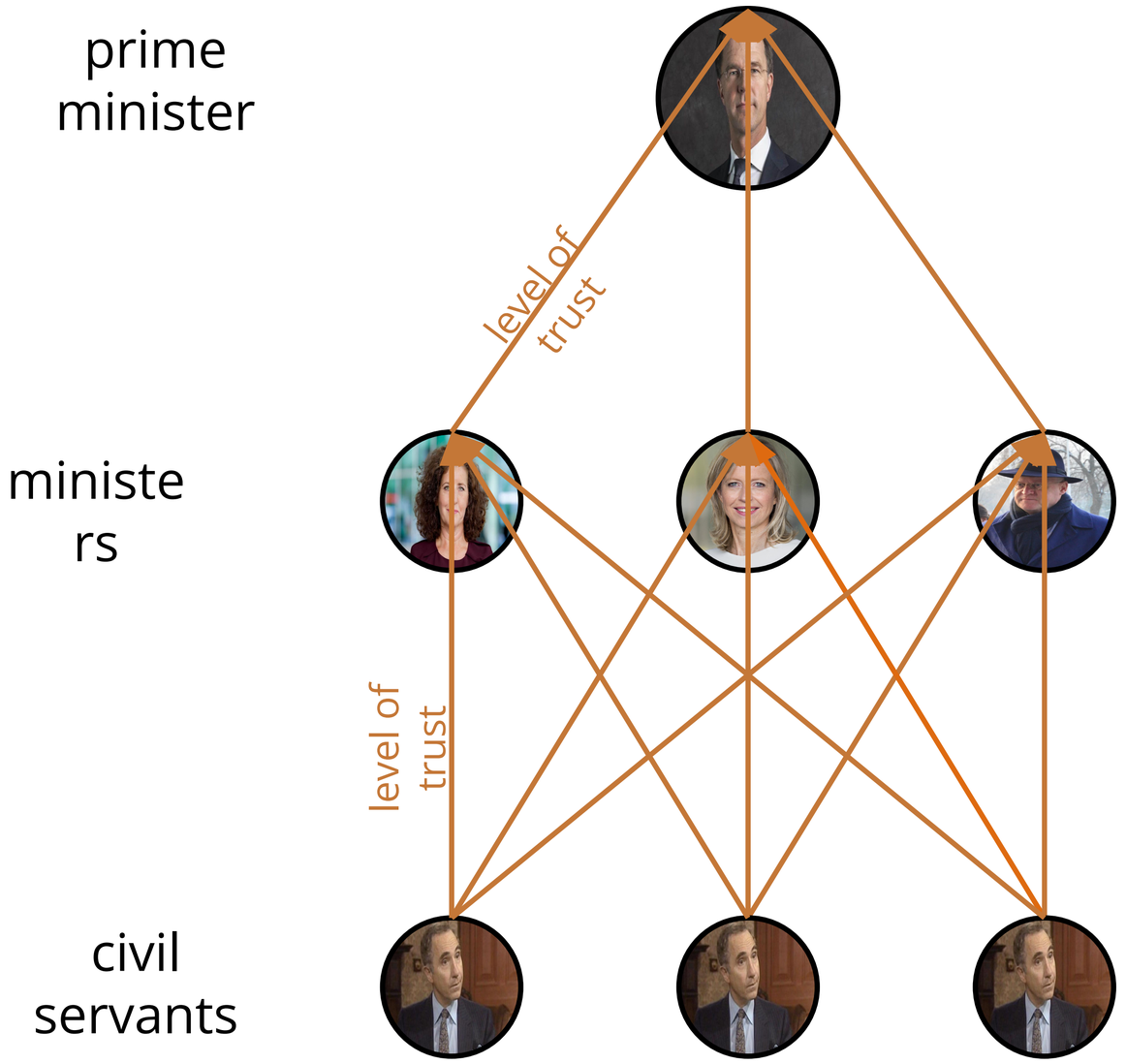

```
for i in [1 … 2]:
    for j in [1 … 3]:
        k[i] += w[i,j] * x[i]
    k[i] += b[i]

for i in [1 … 3]:
    h[i] = sigmoid(k[i]))

for i in [1 … 3]:
```

BACKWARD PASS IN PSEUDOCODE

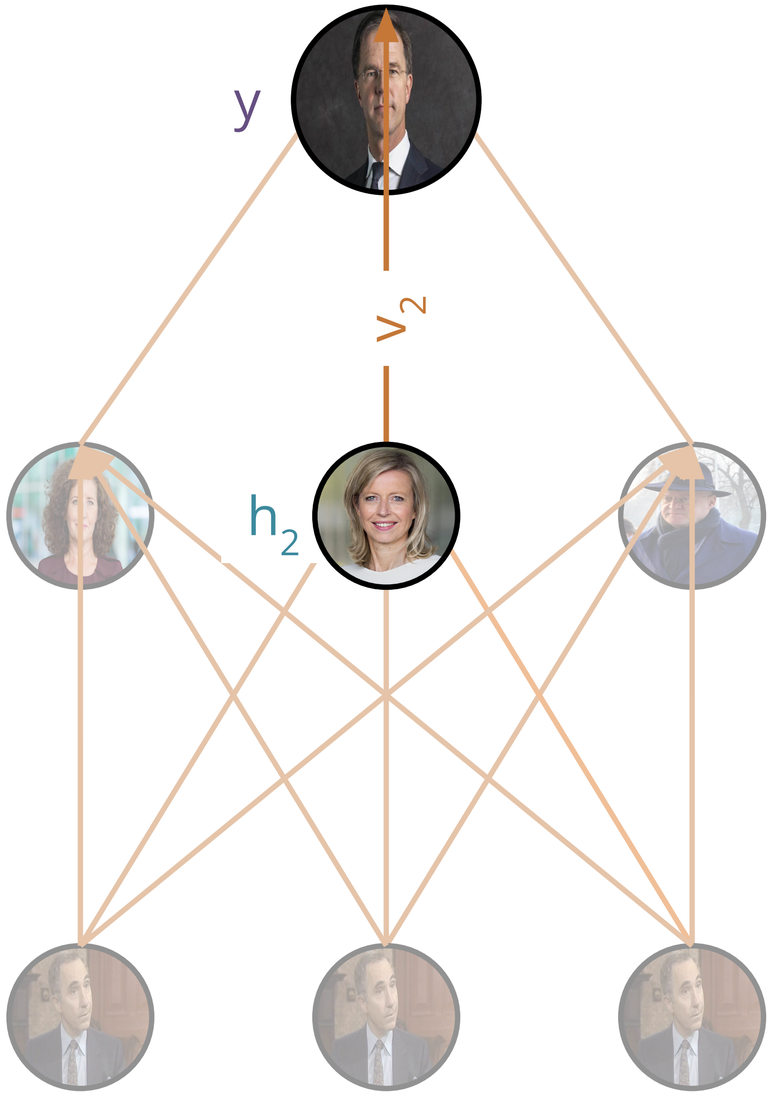



```
y' = 2 * (y - t) # the error

for i in [1 … 3]:

    v'[i] = y' * h[i]

    h'[i] = y' * v[i]

c' = y'


for i in [1 … 3]:

    k'[i] = sig(h'[i])) * sig(1-h'[i]))


for i in [1 … 2]:
```

AN ANALOGY

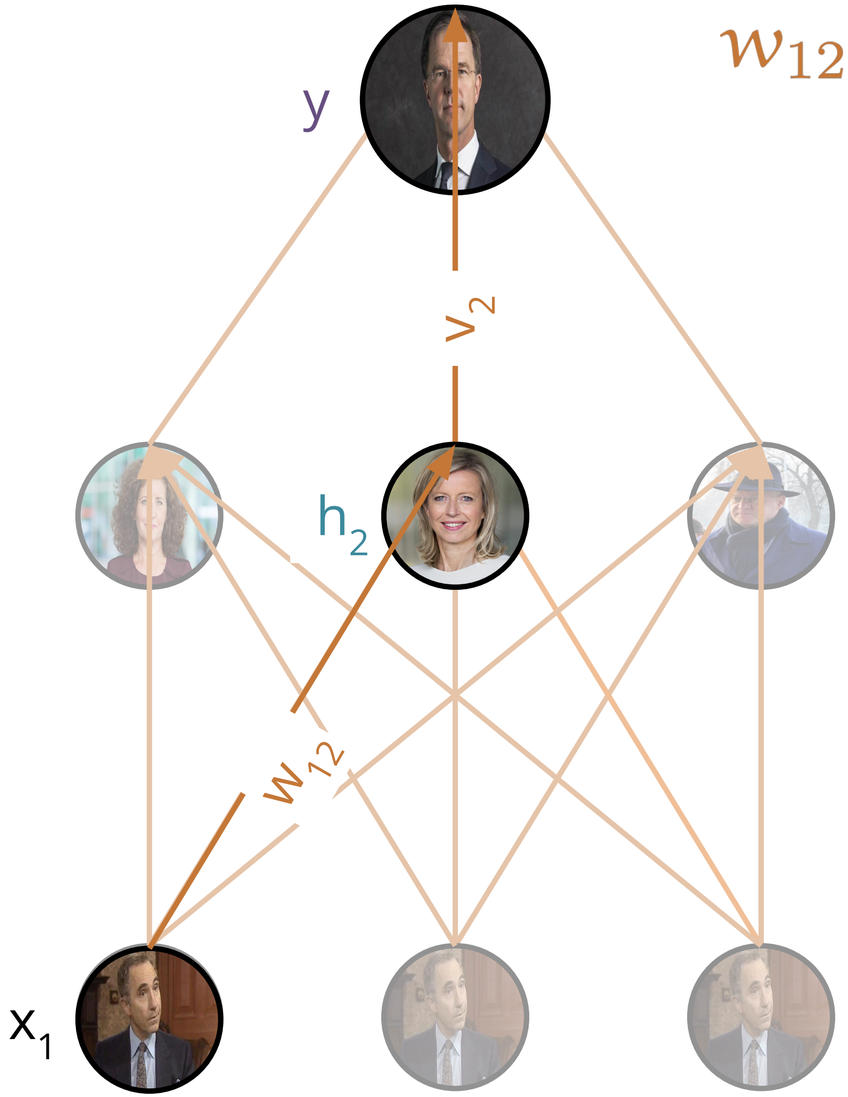

prime minister
ministers
civil servants
level of trust
level of trust

AN ANALOGY



$$v_2 \leftarrow v_2 - \alpha \cdot 2(y - t) \cdot h_2$$

AN ANALOGY



$$w_{12} \leftarrow w_{12} - \alpha \cdot 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

global error

h_2's contribution

h_2 's contribution, considering the activation function

x_1 's contribution

RECAP

Backpropagation: method for computing derivatives.

Combined with gradient descent to train NNs.

Middle ground between symbolic and numeric computation.

- Break computation up into operations.
- Work out local derivatives symbolically.
- Work out global derivative numerically.

Lecture 2: Backpropagation

Peter Bloem

Deep Learning 2020

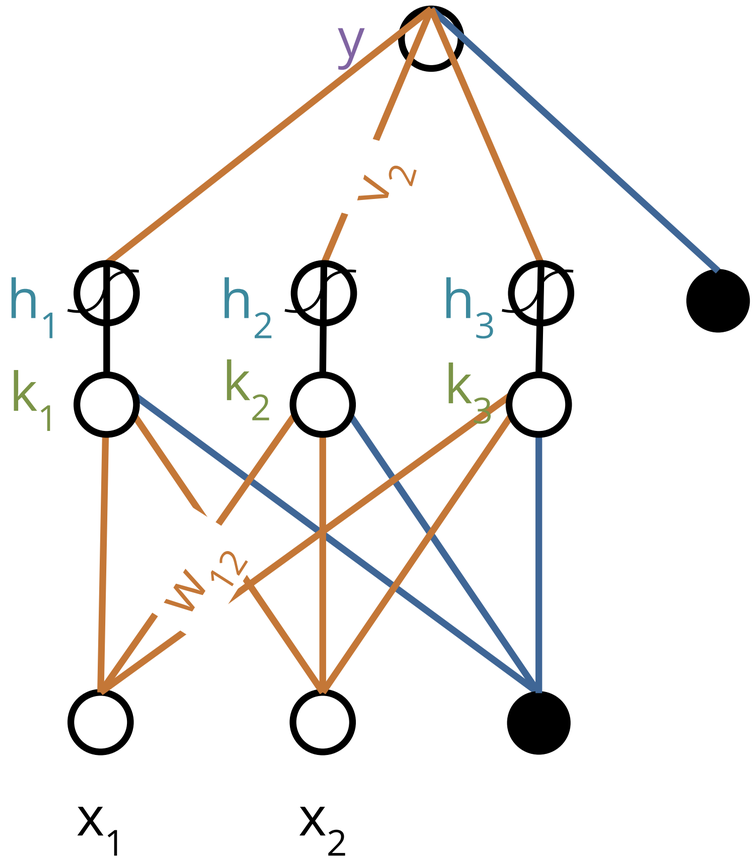
dlvu.github.io

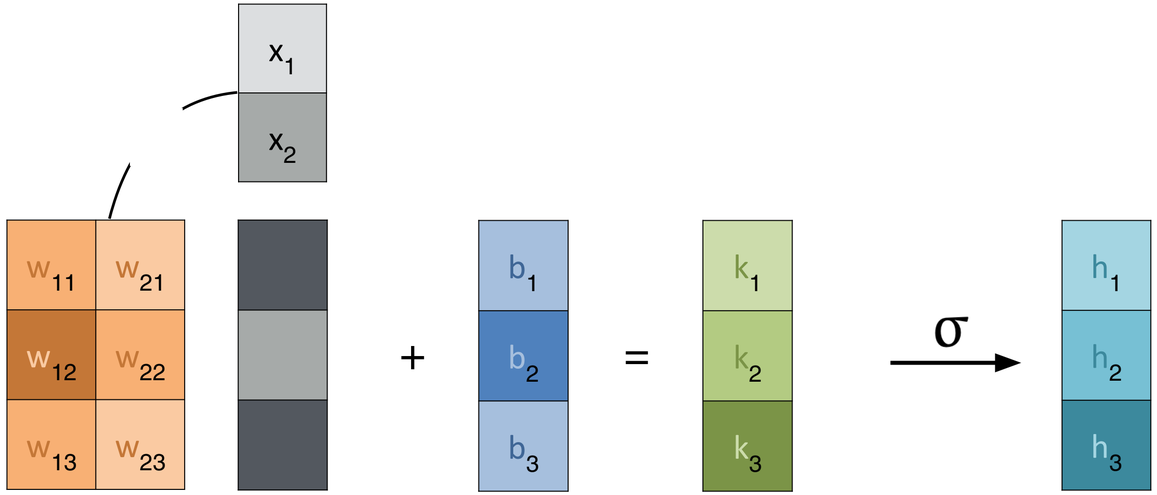
VU VRIJE UNIVERSITEIT AMSTERDAM

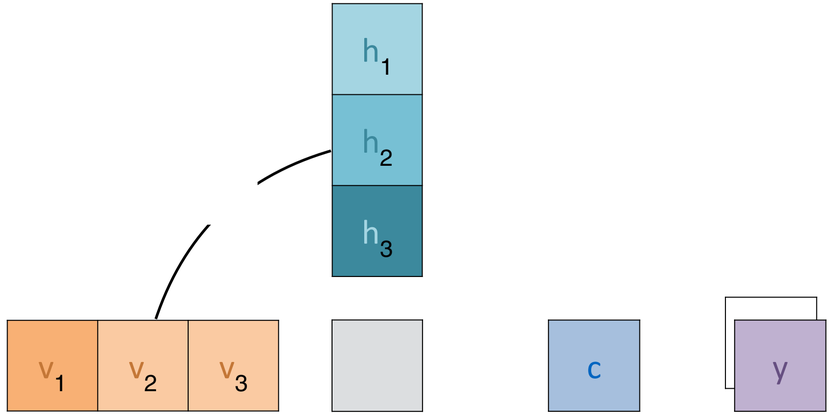
PART THREE: TENSOR BACKPROPAGATION

Expressing backpropagation in vector, matrix and tensor operations.

IT'S ALL JUST LINEAR ALGEBRA







$$\mathbf{f}(\mathbf{x}) = \mathbf{V}\,\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

MATRIX MULTIPLICATION


```
# k, w, k, h, etc are zero'd arrays

for i in [1 … 3]:

    for j in [1 … 3]:

        k[i] += w[i,j] * x[i]

    k[i] += b[i]


for i in [1 … 3]:

    h[i] = sigmoid(k[i]))


for i in [1 … 3]:
```

VECTORIZING

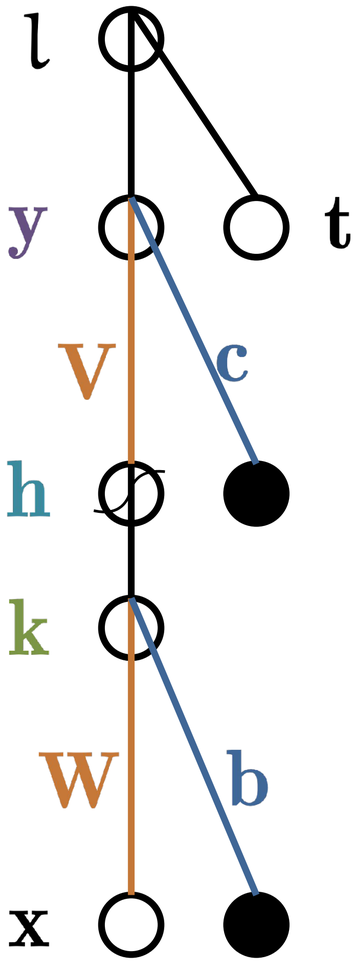
Express everything as operations on vectors, matrices and tensors.

Get rid of all the loops

Makes the notation *simpler*.

Makes the execution *faster*.

FORWARD



$$l = \sum (y - t)^2$$

$$y = Vh + c$$

$$h = \sigma(k)$$

$$k = Wx + b$$

k = w.dot(x) + b

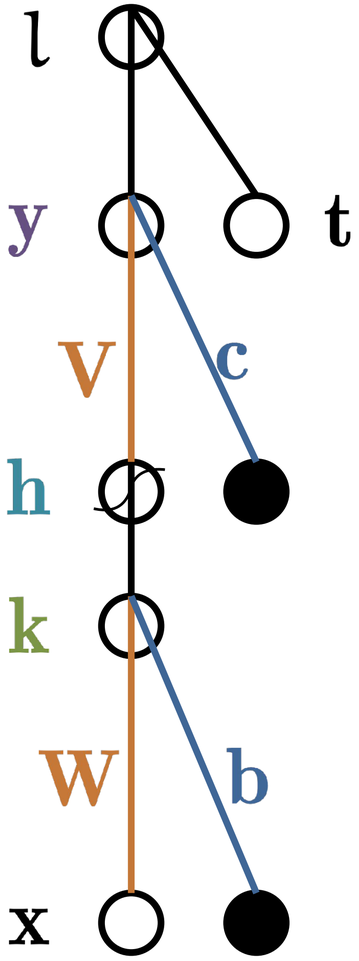
h = sigmoid(k)

y = v.dot(h) + c

l = (y - t) ** 2

VU

BUT WHAT ABOUT THE BACKWARD PASS?



$$l = \sum (y - t)^2$$

$$y = Vh + c$$

$$h = \sigma(k)$$

$$k = Wx + b$$

Can we do something like this?

$$\frac{\partial l}{\partial W} \stackrel{?}{=} \frac{\partial l}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial k}\frac{\partial k}{\partial W}$$

VU

GRADIENTS, JACOBIANS, ETC

$\mathbf{f}(\mathbf{A}) = \mathbf{B}, \quad \frac{\partial \mathbf{B}}{\partial \mathbf{A}}$: derivatives of every element of **A** over every element of **B**

for instance:

$$\mathbf{f}\left(\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}\right) = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\mathbf{J_f} = \begin{pmatrix} \partial b_1/\partial a_1 & \partial b_2/\partial a_1 \\ \partial b_1/\partial a_2 & \partial b_2/\partial a_2 \\ \partial b_1/\partial a_3 & \partial b_2/\partial a_3 \end{pmatrix}$$

input is a ↓ / function returns a →	scalar	vector	matrix
scalar	scalar	vector	matrix
vector	vector	matrix	?
matrix	matrix	?	?

arrange derivatives in a:

$$\frac{\partial l}{\partial w} \stackrel{?}{=} \frac{\partial l}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial k}\frac{\partial k}{\partial w}$$

uh
oh

SIMPLIFYING ASSUMPTIONS

1) The computation graph *always* has a single, scalar output: l

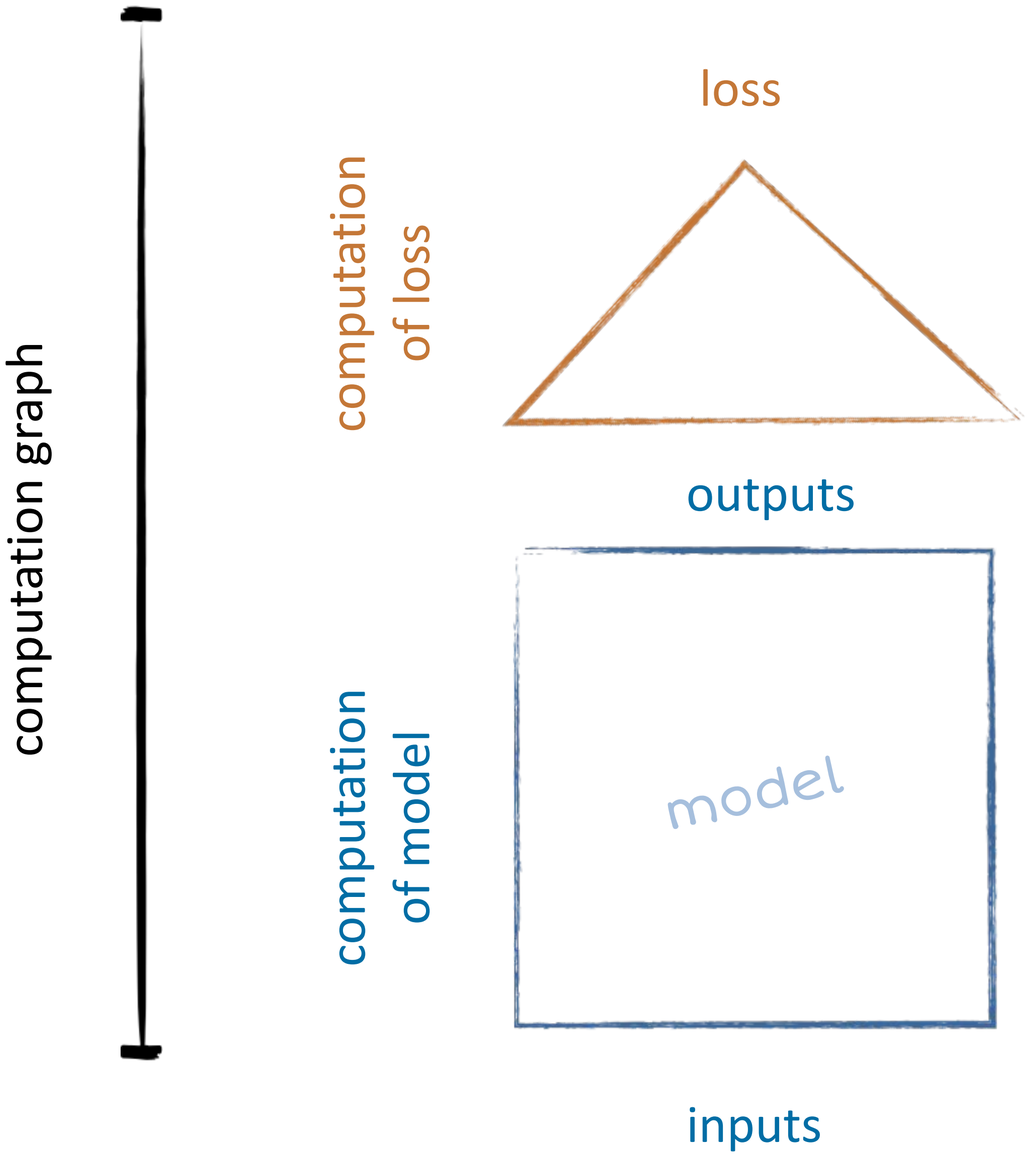
2) We are only ever interested in the derivative of l.

$$\frac{\partial l}{\partial \mathbf{W}}$$

<- scalar

<- tensor of any dimension

	scalar	vector	matrix
scalar	scalar	vector	matrix
vector	vector	matrix	?
matrix	matrix	?	?
3-tensor	3-tensor	?	?


computation graph
loss
computation of loss
outputs
computation of model
model
inputs

THE GRADIENT

We will call $\frac{\partial l}{\partial \mathbf{W}}$ the gradient of l (with respect to $\mathbf{W}$)

Commonly written as $\nabla_{\mathbf{W}} l$

Nonstandard assumption: $\nabla_{\mathbf{W}} l$ has the same *shape* as $\mathbf{W}$

$$[\nabla_{\mathbf{W}} l]_{ijk} = \frac{\partial l}{\partial W_{ijk}}$$

VU

NEW NOTATION: THE GRADIENT *FOR* W

$$\nabla_{\mathbf{W}} l = \mathbf{W}^{\nabla}$$

<- always the same

<- actual object of interest

$$\mathbf{b}^{\nabla} = \nabla_{\mathbf{b}} l = \left[\frac{\partial l}{\partial b_1} \cdots \frac{\partial l}{\partial b_n} \right]^{\mathsf{T}}$$

$$y^{\nabla} = \nabla_{y} l = \frac{\partial l}{\partial y}$$

VU

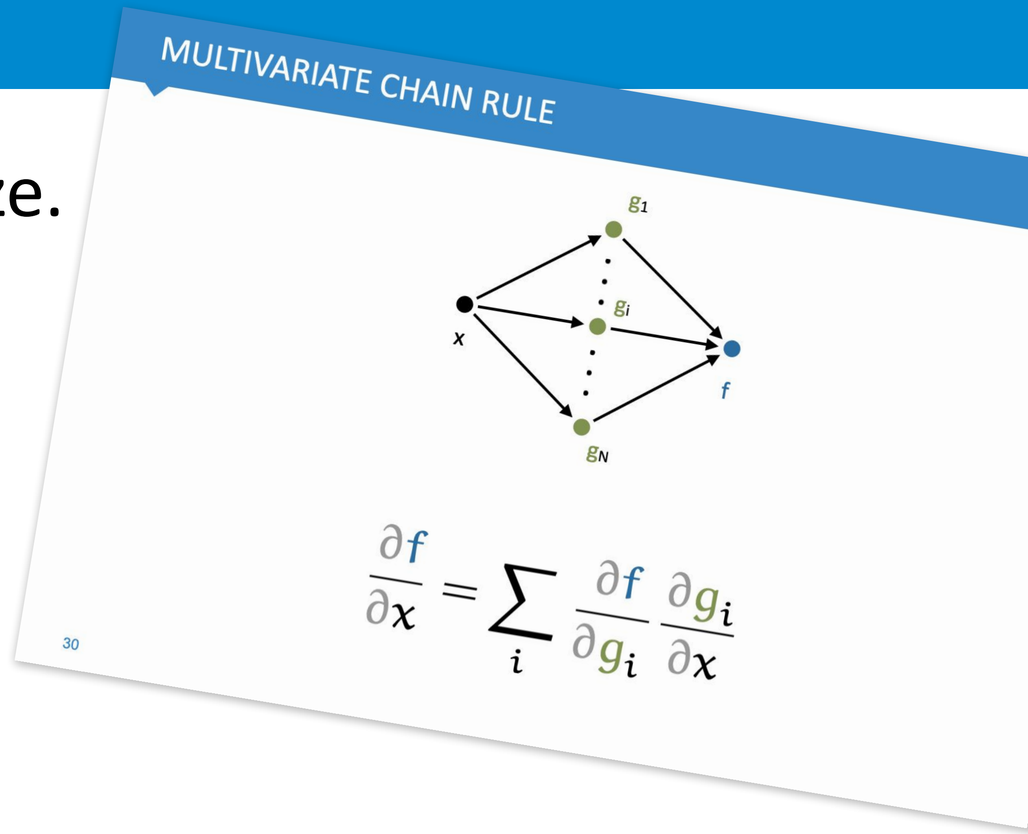
$$[\mathbf{W}^{\nabla}]_{ij} = [W_{ij}]^{\nabla} = \frac{\partial l}{\partial W_{ij}}$$

$$= W_{ij}^{\nabla}$$

TENSOR BACKPROPAGATION

Work out *scalar* derivatives first, then vectorize.

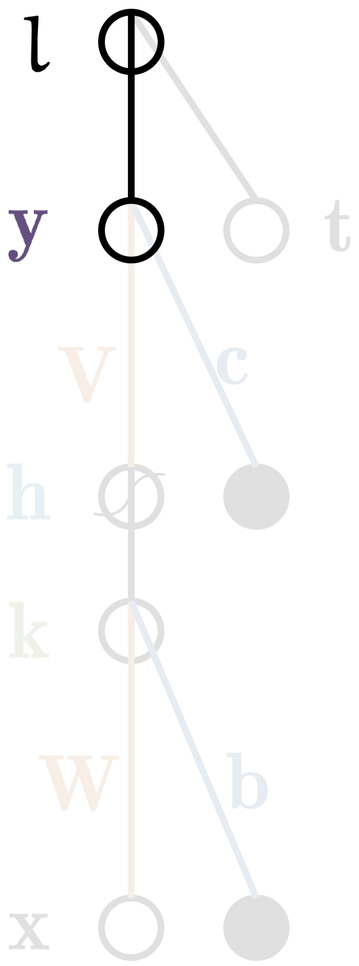
Use the **multivariate chain rule** to derive the scalar derivative.



Apply the chain rule step by step.

Start at the loss and work backward, *accumulating* the gradients.

GRADIENT FOR y



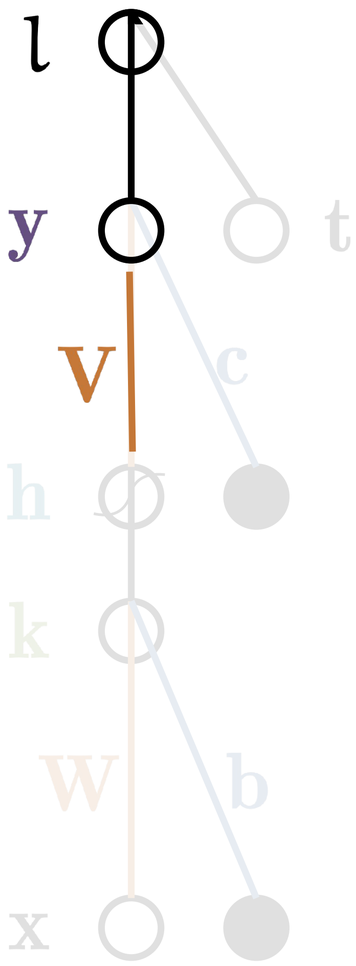
$$\mathbf{y}^{\nabla}$$

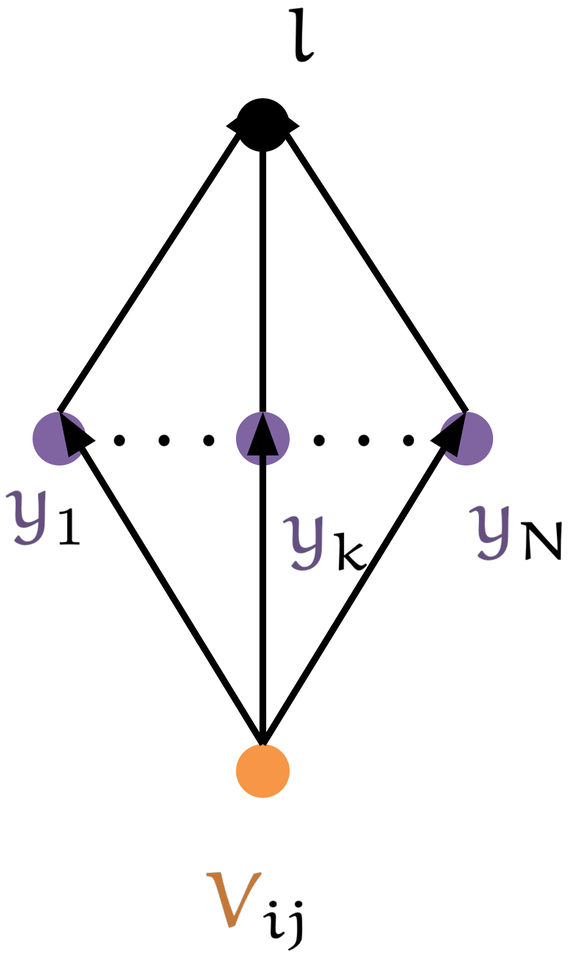
$$\begin{aligned} y_i^{\nabla} &= \frac{\partial l}{\partial y_i} \\ &= \frac{\partial \sum_k (y_k - t_k)^2}{\partial y_i} = \sum_k \frac{\partial (y_k - t_k)^2}{\partial y_i} \\ &= \sum_k 2(y_k - t_k) \frac{\partial y_k - t_k}{\partial y_i} = 2(y_i - t_i) \end{aligned}$$

$$\mathbf{y}^{\nabla} = 2 \begin{pmatrix} y_1 - t_1 \\ \vdots \\ y_n - t_n \end{pmatrix} = 2 \cdot (\mathbf{y} - \mathbf{t})$$

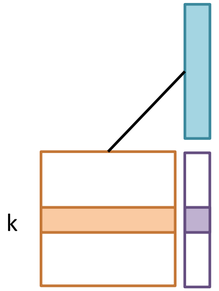
VU

GRADIENT FOR V

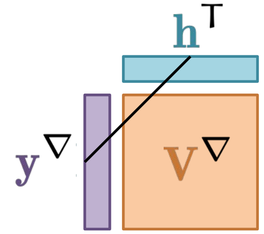




$$
\begin{aligned}
V^{\nabla}_{ij} &= \frac{\partial l}{\partial V_{ij}} \\
&= \sum_k \frac{\partial l}{\partial y_k}\frac{\partial y_k}{\partial V_{ij}} \quad = \sum_k y^{\nabla}_k \frac{\partial y_k}{\partial V_{ij}} \\
&= \sum_k y^{\nabla}_k \frac{\partial [\mathbf{V}\mathbf{h}+\mathbf{c}]_k}{\partial V_{ij}} = \sum_k y^{\nabla}_k \frac{\partial [\mathbf{V}_{k:}\mathbf{h}]_k + c_k}{\partial V_{ij}} \\
&= \sum_k y^{\nabla}_k \frac{\partial \sum_l V_{kl} h_l + c_k}{\partial V_{ij}} \\
&= \sum_{kl} y^{\nabla}_k \frac{\partial V_{kl} h_l}{\partial V_{ij}} \quad = y^{\nabla}_i \frac{\partial V_{ij} h_j}{\partial V_{ij}} \\
&= y^{\nabla}_i h_j
\end{aligned}
$$



$$
\mathbf{V}^{\nabla} = \begin{pmatrix} \vdots \\ \cdots \quad y^{\nabla}_i h_j \quad \cdots \\ \vdots \end{pmatrix} = \mathbf{y}^{\nabla}\mathbf{h}^{\mathsf{T}}
$$

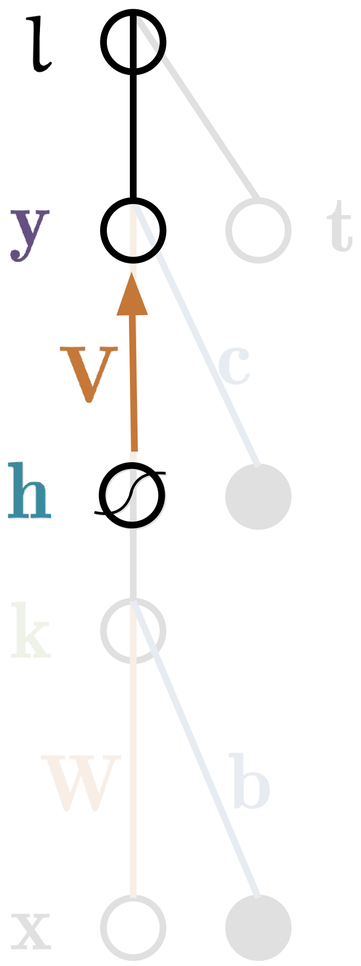


RECIPE

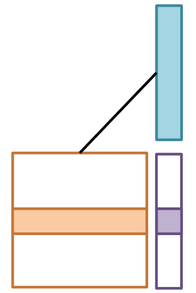
To work out a gradient $\mathbf{X}^{\nabla}$ for some $\mathbf{X}$:

- Write down a **scalar derivative** of the loss over *one element* of **X.**
- Use the **multivariate chain** rule to sum over all outputs.
- Work out the scalar derivative.
- **Vectorize** the computation of $\mathbf{X}^{\nabla}$ in terms of the original inputs.

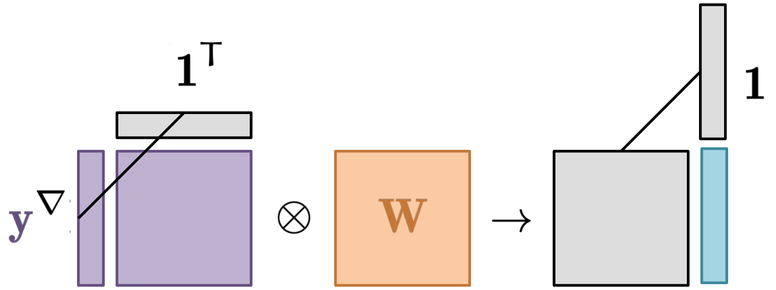
GRADIENT FOR h



$$
\begin{aligned}
h_i^\nabla &= \frac{\partial l}{\partial h_i} \\
&= \sum_k \frac{\partial l}{\partial y_k}\frac{\partial y_k}{\partial h_i} = \sum_k y_k^\nabla \frac{\partial y_k}{\partial h_i} \\
&= \sum_k y_k^\nabla \frac{\partial [\mathbf{W}\mathbf{h}+\mathbf{c}]_k}{\partial h_i} \qquad = \sum_k y_k^\nabla \frac{\partial \sum_l W_{kl}h_l + c_k}{\partial h_i} \\
&= \sum_{kl} y_k^\nabla \frac{\partial W_{kl}h_l + c_k}{\partial h_i} \qquad = \sum_k y_k^\nabla \frac{\partial W_{ki}h_i}{\partial h_i} \\
&= \sum_k y_k^\nabla W_{ki}
\end{aligned}
$$



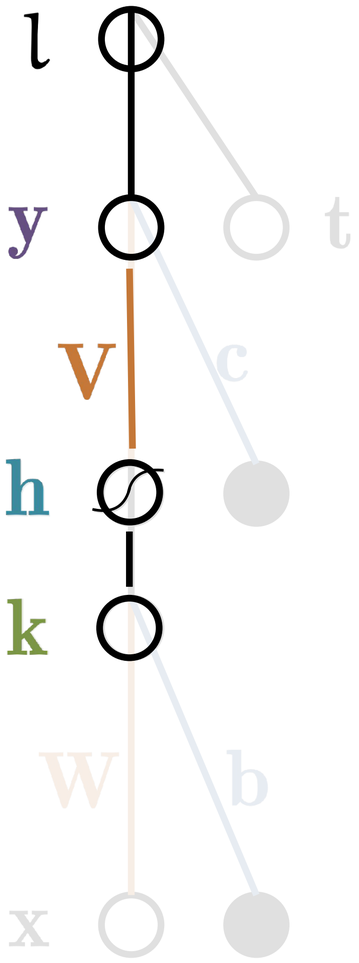
$$
\mathbf{h}^\nabla = \begin{pmatrix} \vdots \\ \sum_k y_k^\nabla W_{ki} \\ \vdots \end{pmatrix} = (\mathbf{y}^\nabla \mathbf{1}^\mathsf{T} \otimes \mathbf{W})\mathbf{1}
$$

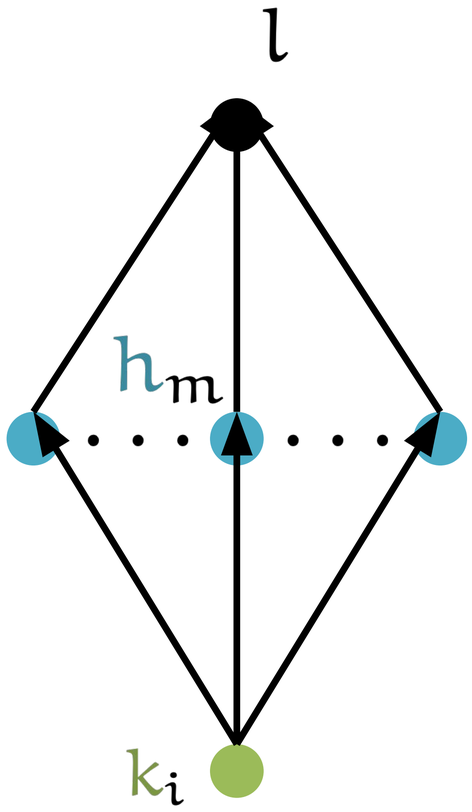


or in numpy:

```
(gy[None, :] * w).sum(axis=1)
```

GRADIENT FOR k





$$k_i^\nabla = \frac{\partial l}{\partial k_i}$$

$$= \sum_m h_m^\nabla \frac{\partial h_m}{\partial k_i} = \sum_m h_m^\nabla \frac{\partial \sigma(k_m)}{\partial k_i}$$

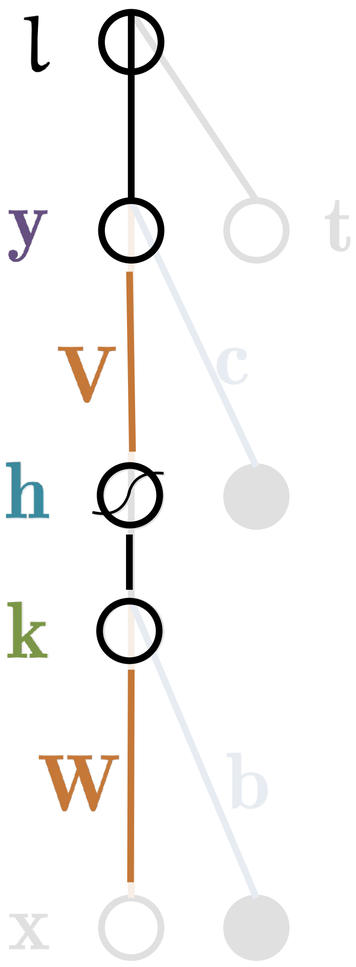
$$= h_i^\nabla \frac{\partial \sigma(k_i)}{\partial k_i}$$

$$= h_i^\nabla \sigma'(k_i) = h_i^\nabla \sigma(k_i)(1 - \sigma(k_i))$$

$$= h_i^\nabla h_i (1 - h_i)$$

$$\mathbf{k}^\nabla = \mathbf{h}^\nabla \otimes \mathbf{h} \otimes (1 - \mathbf{h})$$

GRADIENT FOR W

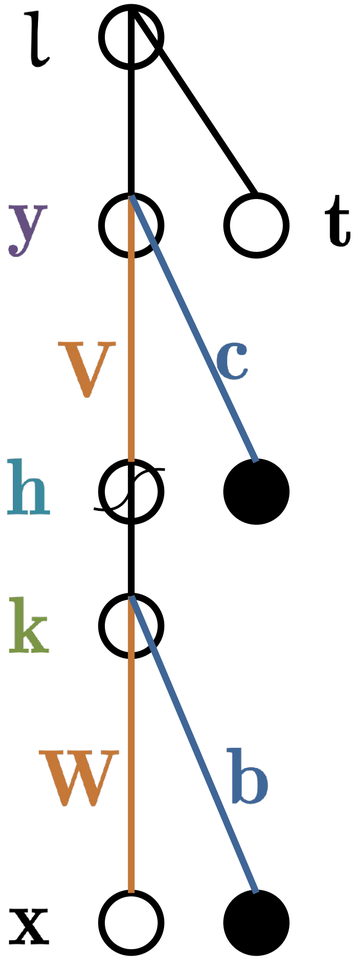


$$
\begin{aligned}
W_{ij}^{\nabla} &= \frac{\partial l}{\partial W_{ij}} \\
&= \sum_m \frac{\partial l}{\partial k_m} \frac{\partial k_m}{\partial W_{ij}} = \sum_m k_m^{\nabla} \frac{\partial k_m}{\partial W_{ij}} \\
&= \sum_m k_m^{\nabla} \frac{\partial [\mathbf{W}\mathbf{x} + \mathbf{b}]_m}{\partial W_{ij}} = \sum_m k_m^{\nabla} \frac{\partial [\mathbf{W}_{m:}\mathbf{x}]_m + b_m}{\partial W_{ij}} \\
&= \sum_m k_m^{\nabla} \frac{\partial \sum_l W_{ml} x_l + b_m}{\partial W_{ij}} \\
&= \sum_{ml} k_m^{\nabla} \frac{\partial W_{ml} x_l}{\partial W_{ij}} = k_i^{\nabla} \frac{\partial W_{ij} x_j}{\partial W_{ij}} \\
&= k_i^{\nabla} x_j
\end{aligned}
$$

$$
\mathbf{W}^{\nabla} = \begin{pmatrix} & \vdots & \\ \dots & k_i^{\nabla} x_j & \dots \\ & \vdots & \end{pmatrix} = \mathbf{k}^{\nabla} \mathbf{x}^{\mathsf{T}}
$$

VU

PSEUDOCODE: BACKWARD



y’ = 2 * (y - t)

v’ = y’.mm(h.T)

h’ = y’[None, :] * w).sum(axis=1)

k’ = sigmoid(k) * sigmoid(1 - k)

w’ = k’.mm(x.T)

RECAP

Vectorizing makes notation simpler, and computation faster.

Vectorizing the **forward pass** is usually easy.

Backward pass: work out the scalar derivatives, accumulate, then vectorize.

Lecture 2: Backpropagation

Peter Bloem

Deep Learning 2020

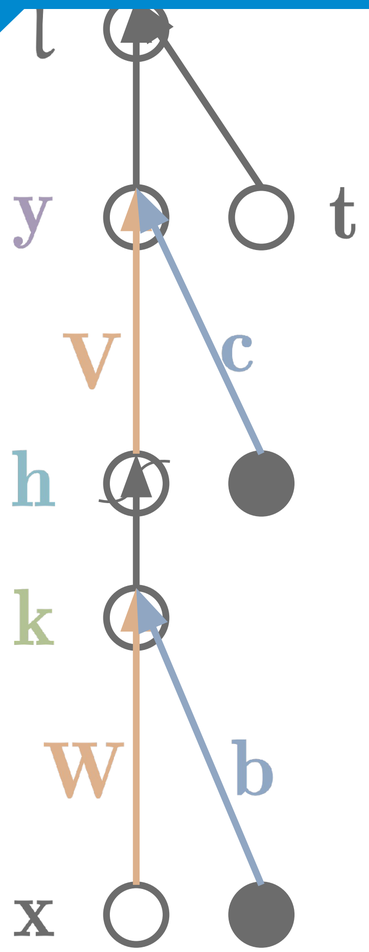
dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

PART FOUR: AUTOMATIC DIFFERENTIATION

Letting the computer do all the work.

THE STORY SO FAR



v' = y'.mm(h.T)

h' = y'[None, :] * w).sum(axis=1)

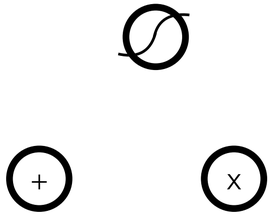
k' = sigmoid(k) * sigmoid(1 - k)

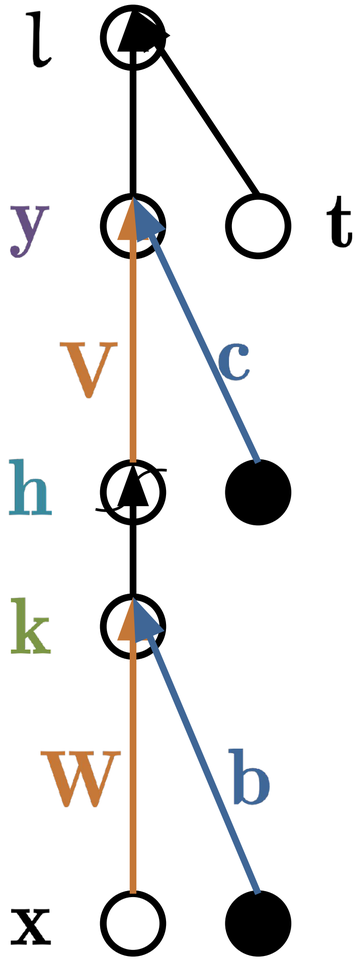
w' = k'.mm(x.T)

pen and paper

in the computer

THE IDEAL





$k = w.dot(x) + b$

$h = sigmoid(k)$

$y = v.dot(h) + c$

$l = (y - t) ** 2$

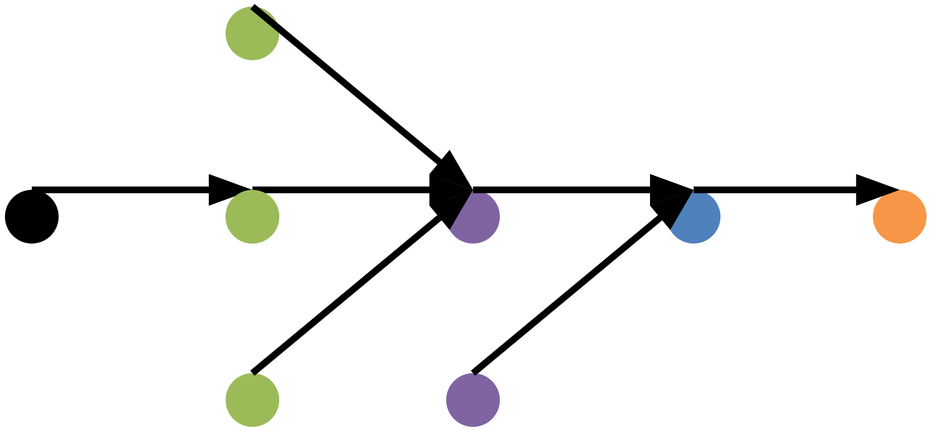
pen and paper

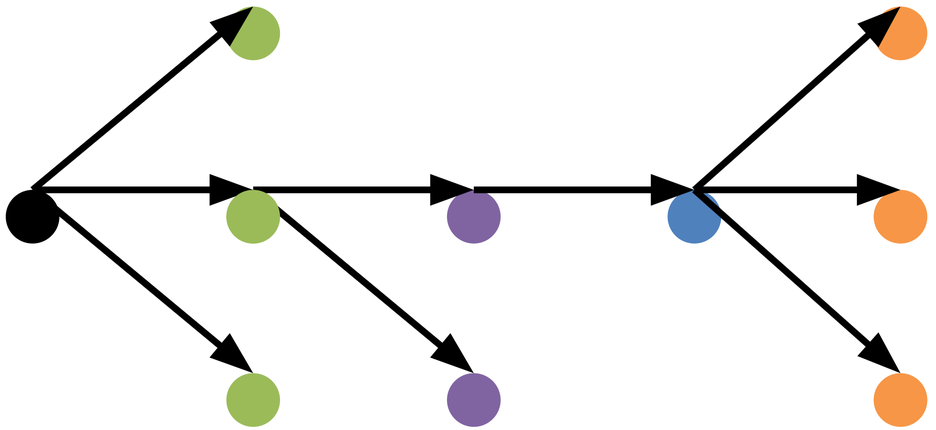
in the computer

l.backward() *# start backprop*

VU

AUTOMATIC DIFFERENTIATION





many inputs, few outputs, work *backward*

few inputs, many outputs, work *forward*

<- deep learning

THE PLAN

Tensors

Building computation graphs

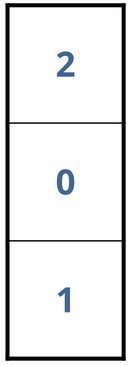
Working out backward functions

TENSORS (AKA MULTIDIMENSIONAL ARRAYS)

scalar

0-tensor

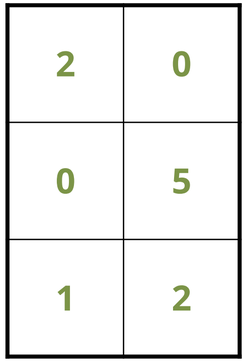
vector



1-tensor

shape: 3

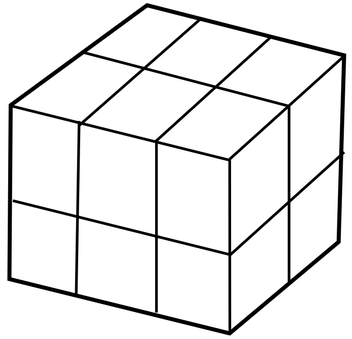
matrix



2-tensor

shape:

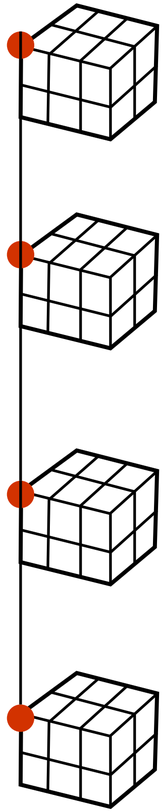
3x2



3-tensor

shape:

2x3x2



4-tensor

shape:

2x3x2x4

A CLASSIFICATION TASK AS TWO TENSORS

181	46	male
181	50	male
166	44	female
171	38	female
152	36	female
156	40	female
167	40	female
170	45	male
178	50	male
191	50	male

X =

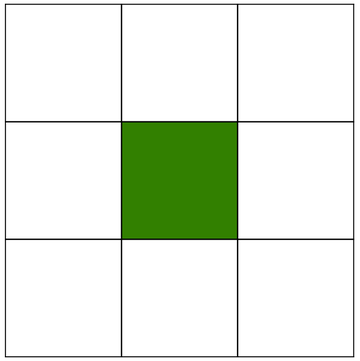
181	46
181	50
166	44
171	38
152	36
156	40
167	40
170	45
178	50
191	50
166	38

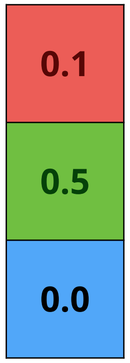
y =

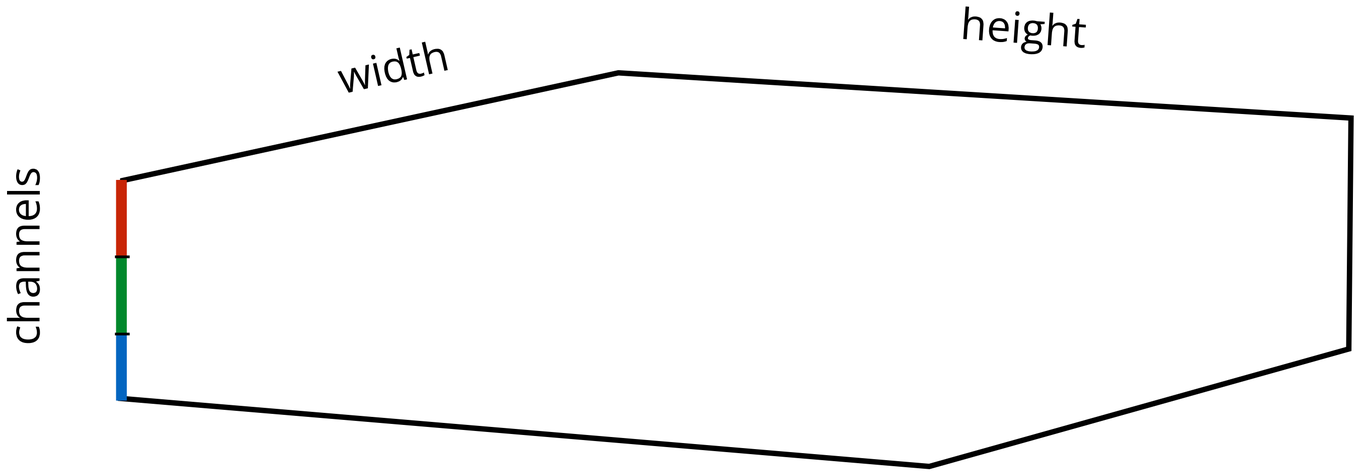
0
0
1
1
1
1
1
0
0
0
1

AN IMAGE AS A 3-TENSOR

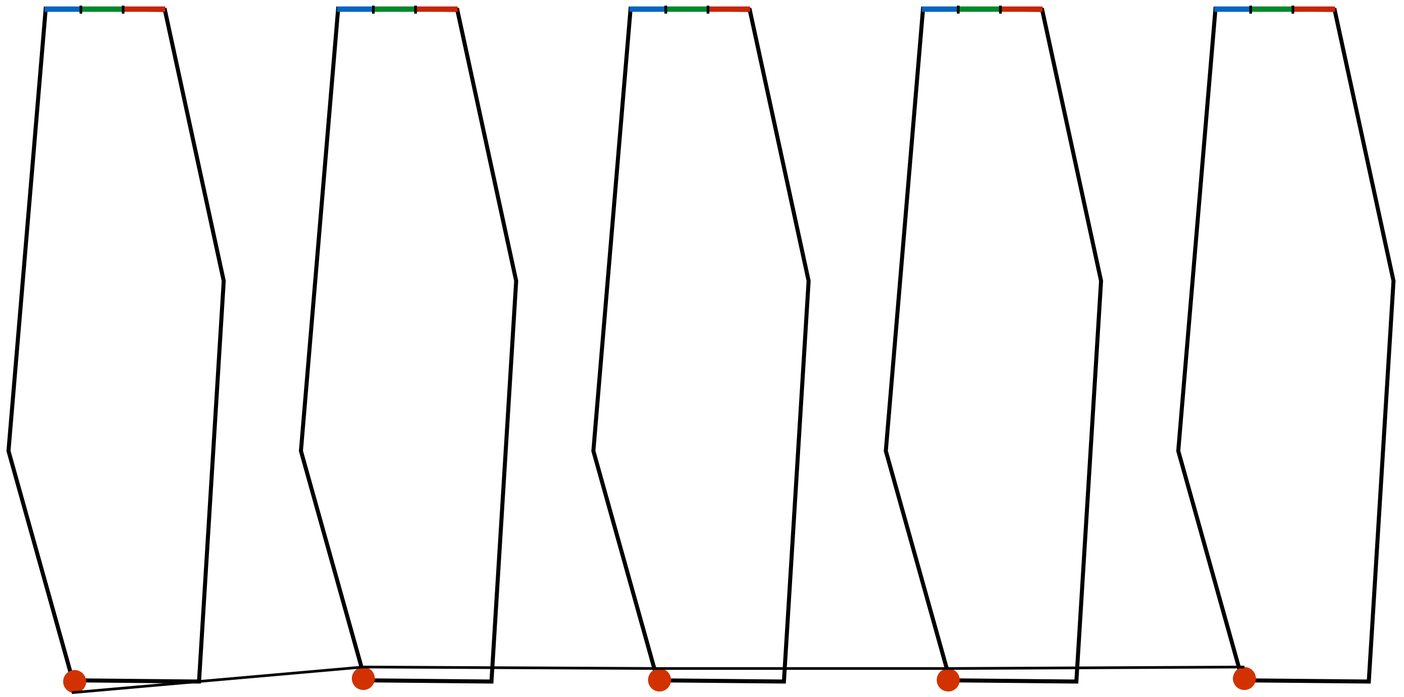
pixel







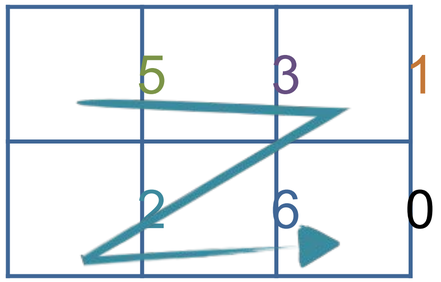
A DATASET OF IMAGES



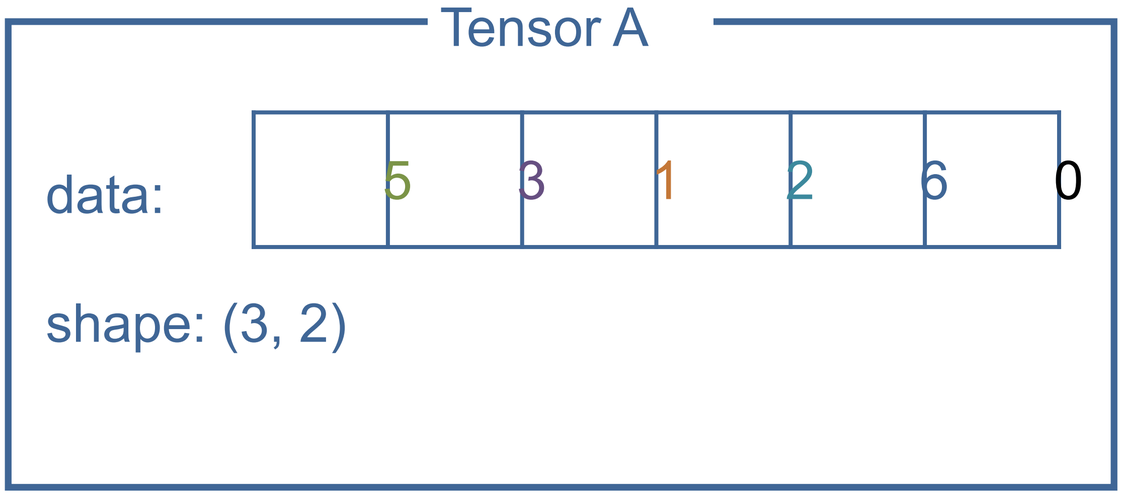
```
In [5]: from keras.datasets import cifar10

        (x_train, y_train), (x_test, y_test) = cifar10.load_data()

        x_train.shape

Out[5]: (50000, 32, 32, 3)
```

TENSORS IN MEMORY: ROW MAJOR ORDERING

A =



row-major ordering



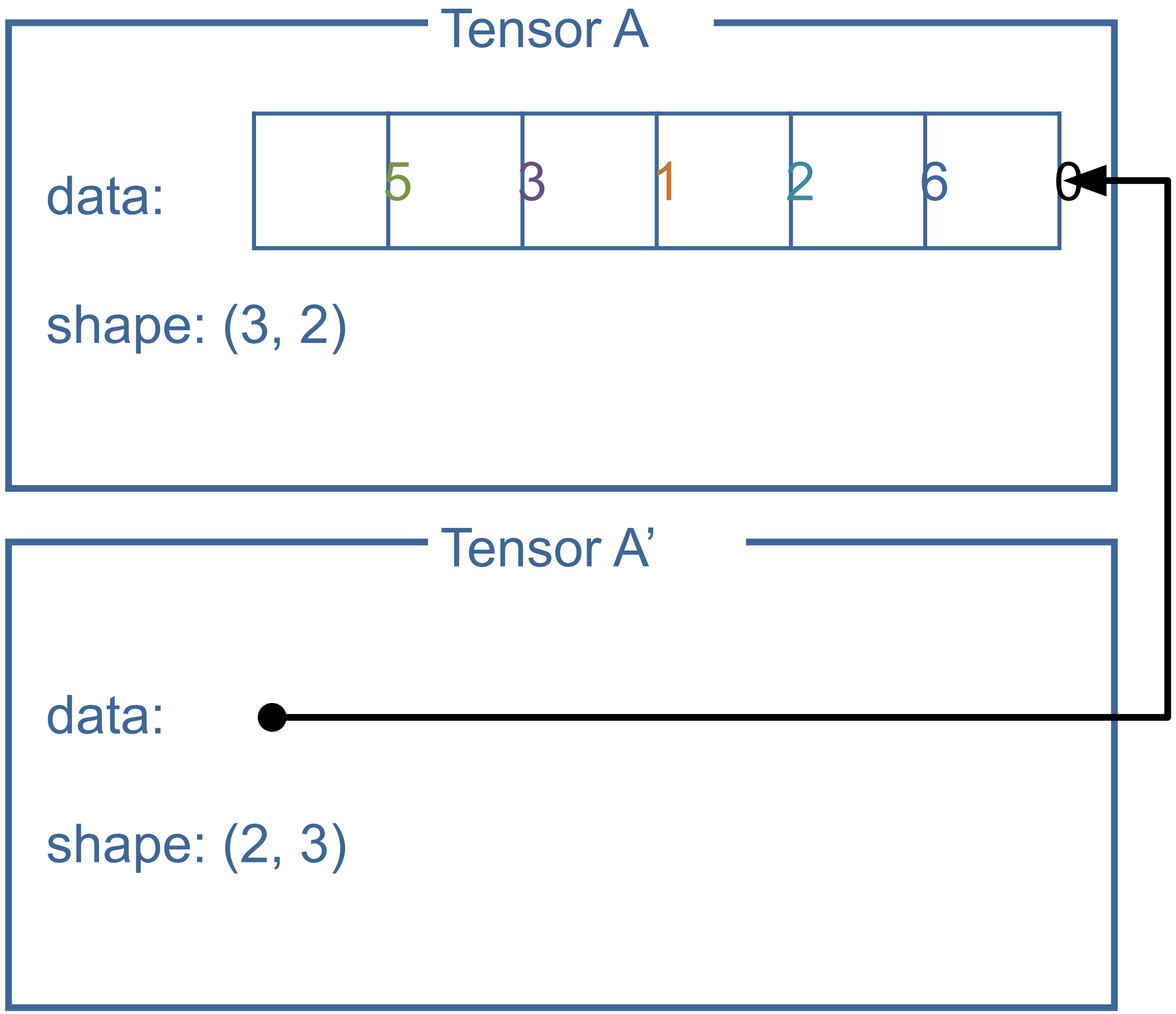
RESHAPE/VIEW

A =

5	3	1
2	6	0

A’ =

5	3
1	2
6	0



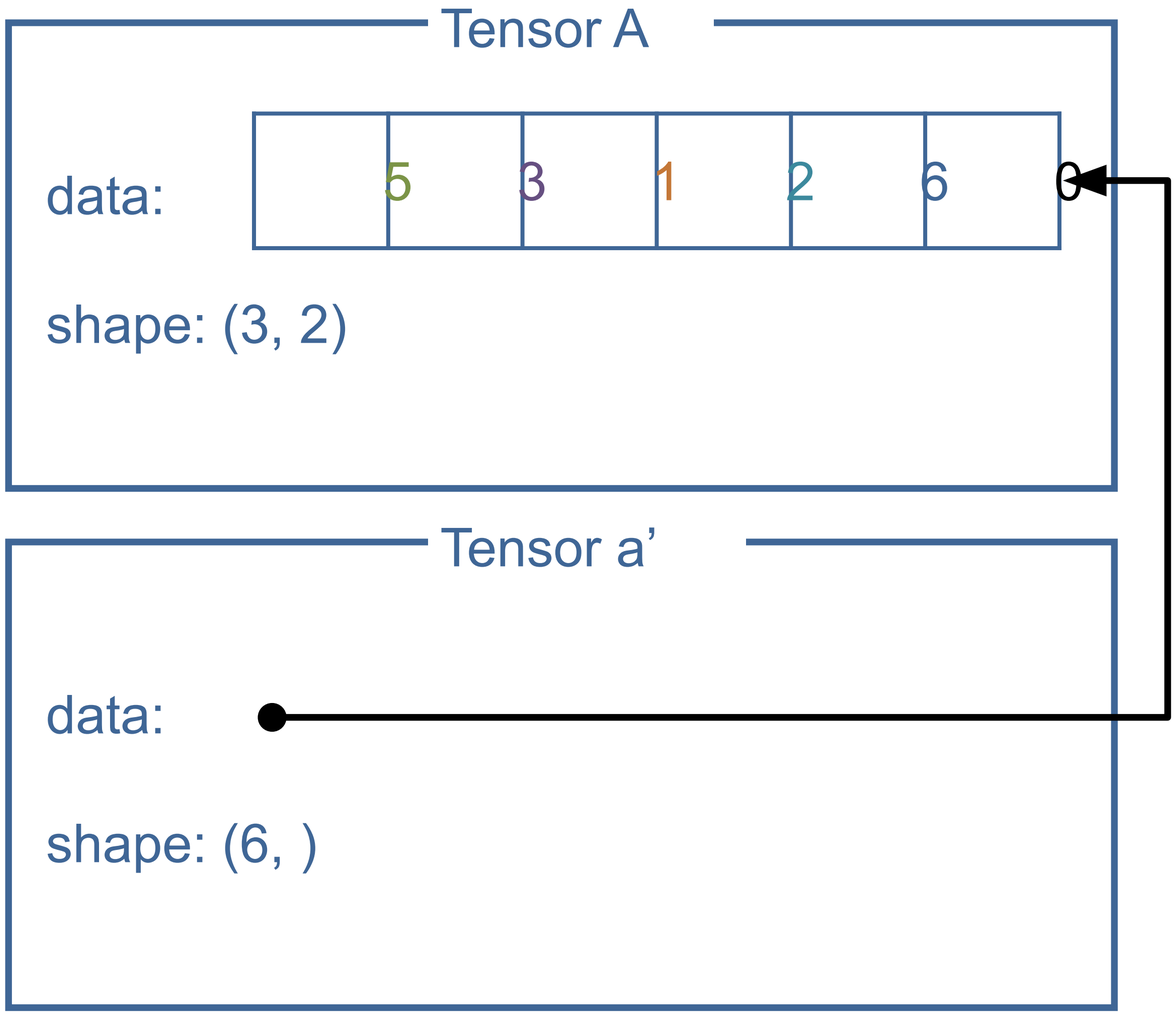
RESHAPE/VIEW

A =

5	3	1
2	6	0

a’ =

5	3	1	2	6	0



EXAMPLE: NORMALIZE COLORS

```
images = load_images(…)

n, h, w, c = images.size()



images = images.reshape(n*h*w, c)

images = images / images.max(dim=0)
# this makes no sense, but it is allowed.

images = images.reshape(n*h, w*c)
images = images.reshape(n, h, w, c)

images = images.reshape(h, n, c, w)
```

TRANSPOSE

A =

5	3	1
2	6	0

A^T=

5	2
3	6
1	0

Tensor A

data: 5 3 1 2 6 0

shape: (3, 2)

Tensor A^T

parent:

shape: reverse(parent.shape)

```
def get(i, j):
    return parent.get(j, i)
```

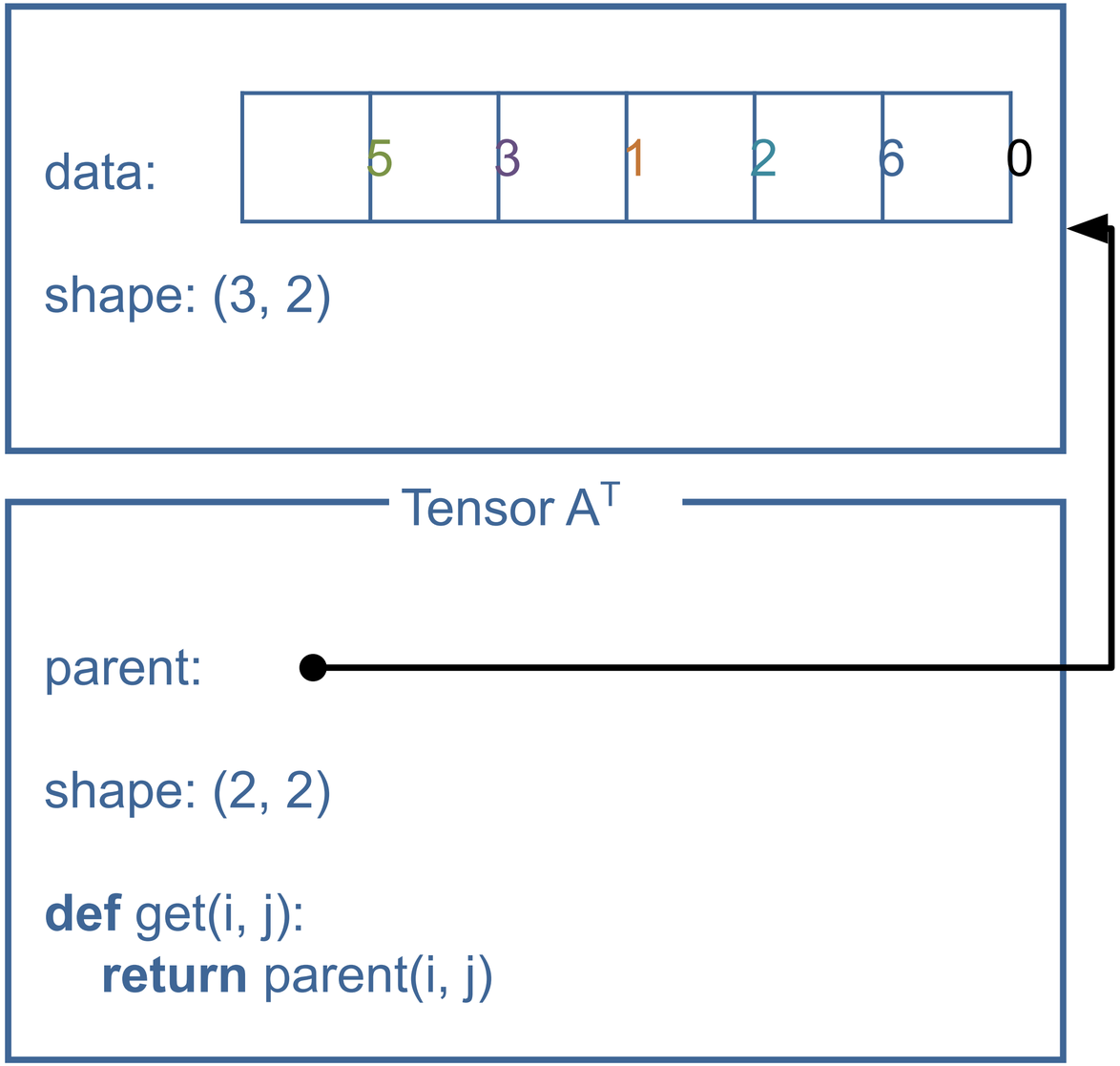
SLICE

A =

5	3	1
2	6	0

A[:, :2]=

5	3
2	6



NON-CONTIGUOUS

Contiguous tensor: the data are directly laid out in row-major ordering for the shape with no gaps or shuffling required.

```
x = torch.randn(2, 3)
x = x[:, 1:]
x.view(6)
Traceback (most recent call last):
…
RuntimeError: view size is not compatible with input tensor's size and stride (at least one dimension spans across two contiguous subspaces). Use .reshape(...) instead.
```

x.contiguous(): make contiguous (by copying the data).

x.view(): reshape if possible without making contiguous.

x.reshape(): reshape, call contiguous() if necessary.

VU

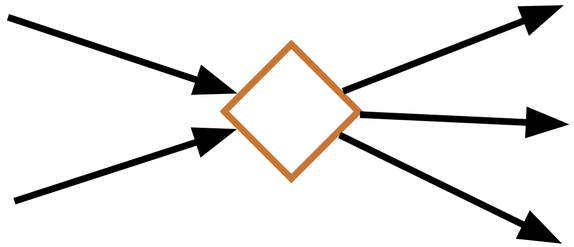
~~Tensors~~

Building computation graphs

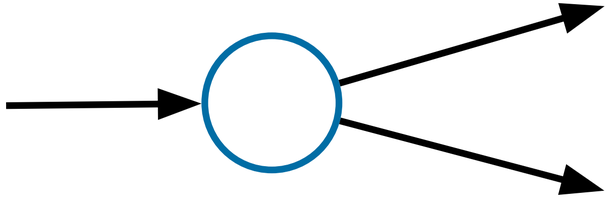
Working out backward functions

COMPUTATION GRAPHS

Operation nodes



Tensor nodes

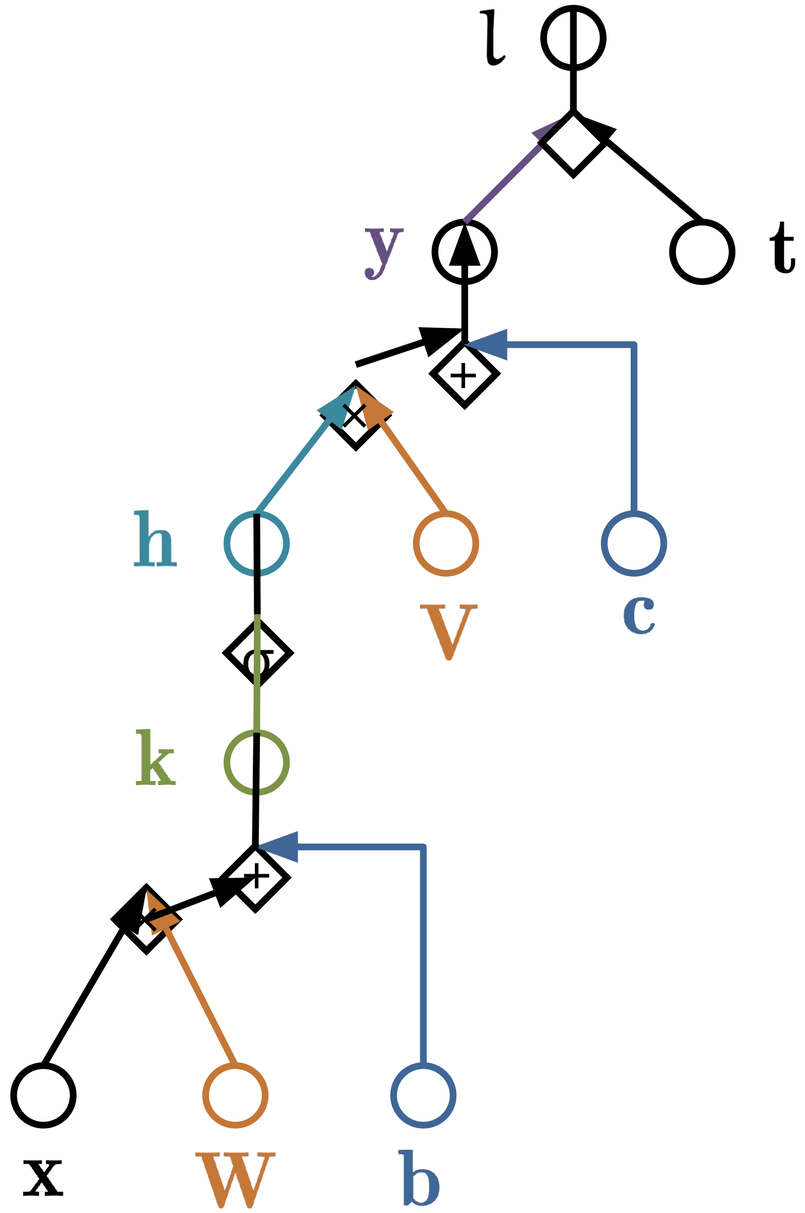


bipartite: only op-to-tensor or tensor-to-op edges

tensor nodes: one input, multiple outputs

operation nodes: multiple inputs and outputs

FEEDFORWARD NETWORK

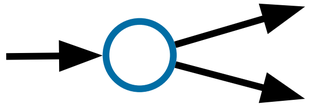


RESTATING OUR ASSUMPTIONS

One output l node with a scalar value.

All gradients are of l, with respect to the tensors on the tensor nodes.

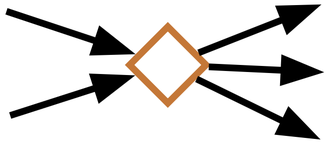
IMPLEMENTATIONS

TensorNode: 

value: <Tensor>

gradient: <Tensor>

source: <OpNode>

OpNode: 

inputs: List<TensorNode>

outputs: List<TensorNode>

op: Op

DEFINING AN OPERATION

Op:

<-- store anything we need for the backward pass

$$f(\mathbf{A}) = \mathbf{B}$$

```
def forward(context, inputs):

    # given the inputs, compute the outputs

    ...
```

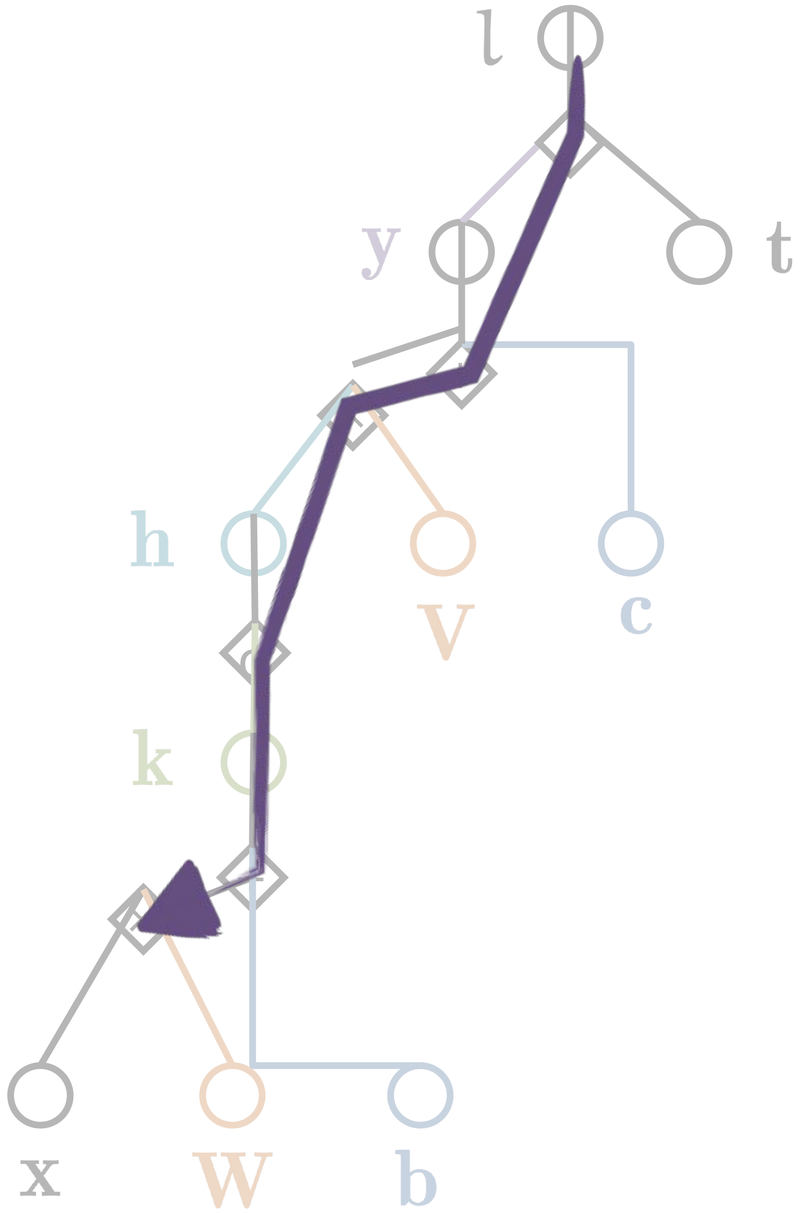
$$\text{forward}_f : \mathbf{A} \mapsto \mathbf{B}$$

$$\text{backward}_f : \mathbf{B}^{\nabla} \mapsto \mathbf{A}^{\nabla}$$

```
def backward(context, outputs_gradient):

    # given the gradient of the loss wrt to the ouputs

    # compute the gradient of the loss wrt to the inputs

    ...
```


VU

BACKPROPAGATION



build a computation graph, perform forward pass

traverse the tree backward breadth-first

call backward for each OpNode

breadth-first ensures that the output gradients are known

add the computed gradients to the tensornodes

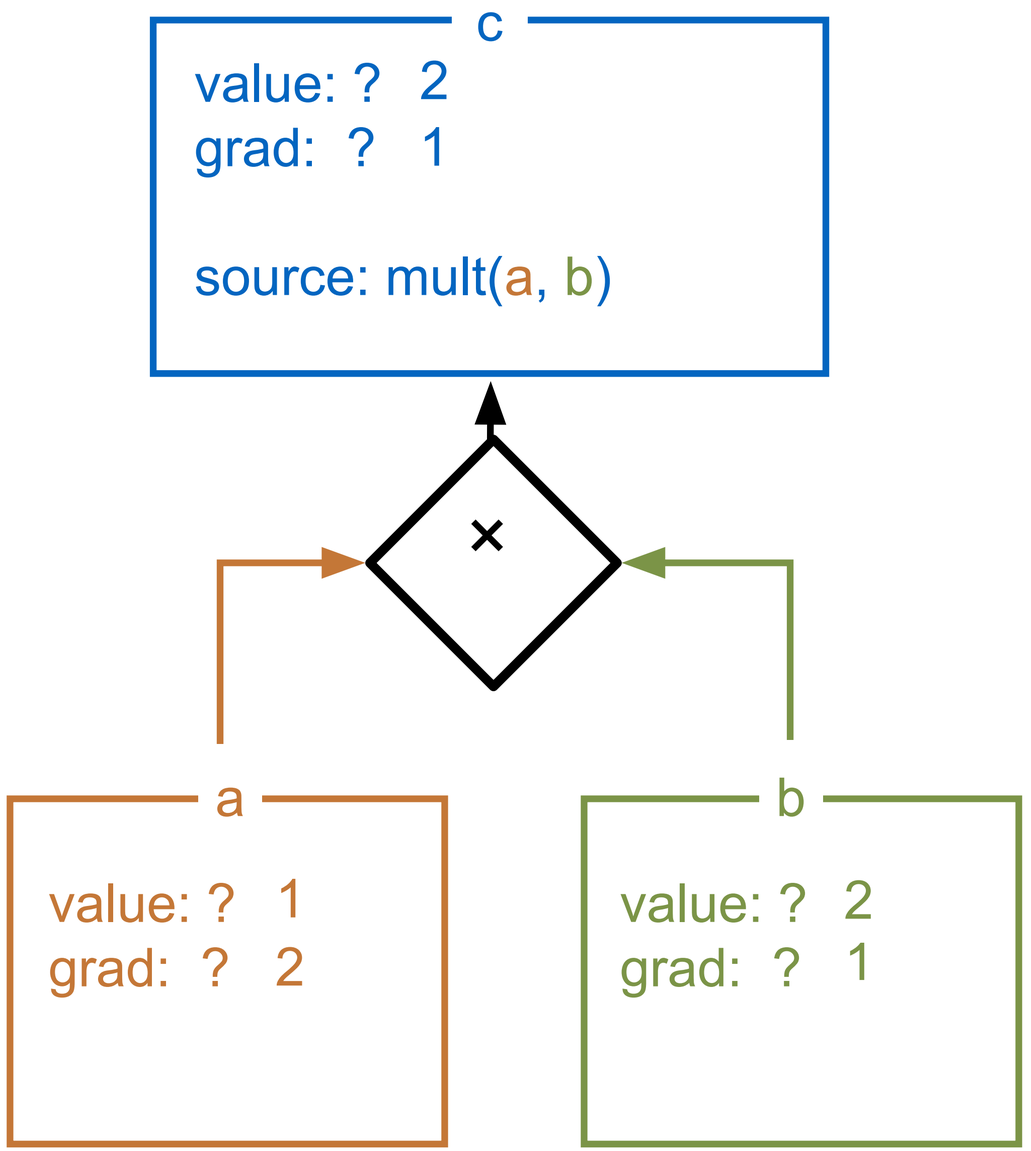
BUILDING COMPUTATION GRAPHS IN CODE

Lazy execution: build your graph, compile it, feed data through.

Eager execution: perform forward pass, keep track of computation graph.

EXAMPLE: LAZY EXECUTION

```
a = TensorNode()
b = TensorNode()


c = a * b

m = Model(
  in=(a, b),
  loss=c)

m.train((1, 2))
```



BUILDING THE COMPUTATION GRAPH: LAZY EXECUTION

Tensorflow 1.x default, Keras default

Define the computation graph.

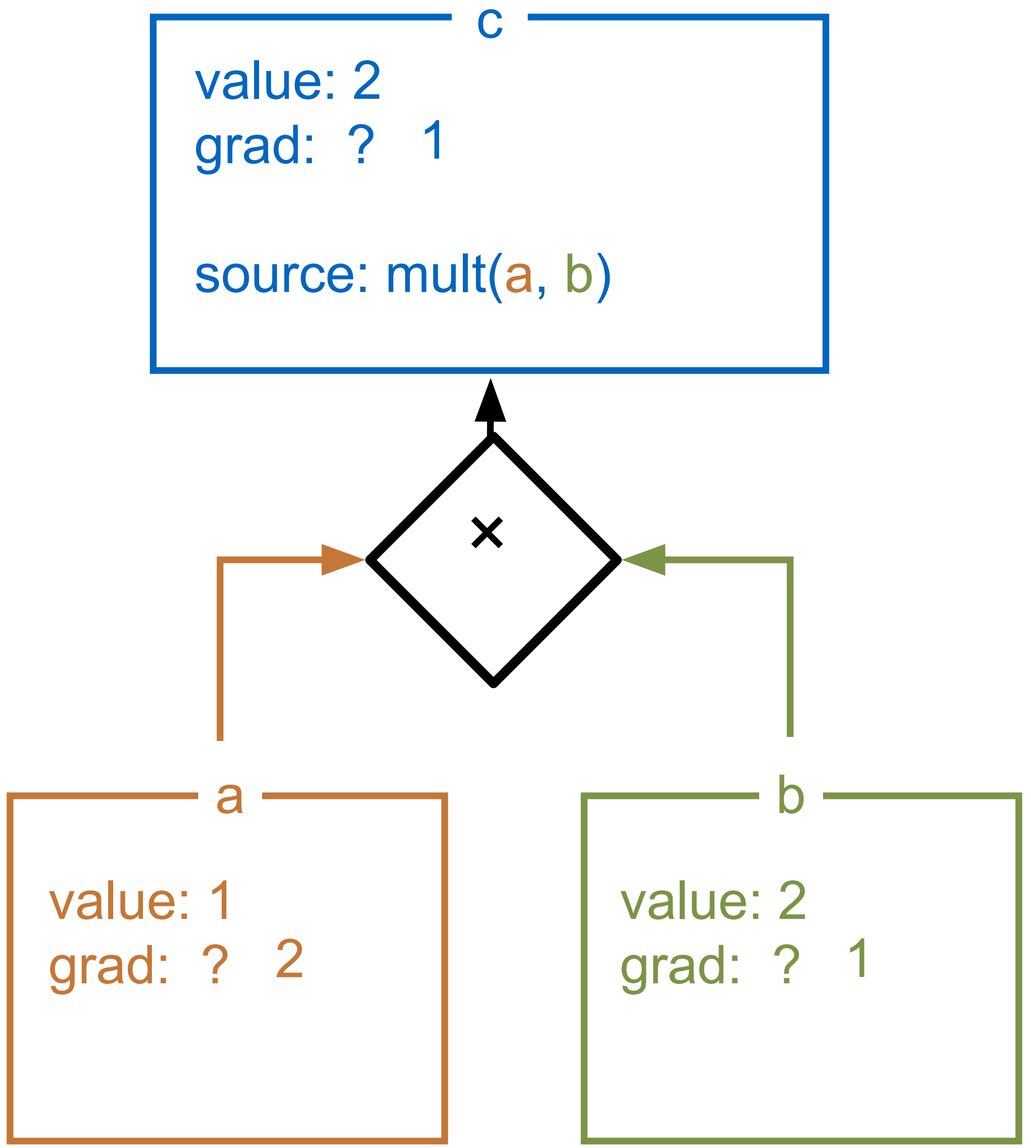
- Compile it.
- Iterate backward/forward over the data

Fast. Many possibilities for optimization. Easy to serialise models. Easy to make training parallel.

Difficult to debug. Model must remain static during training.

EXAMPLE: EAGER EXECUTION

```
a = TensorNode(value=1)
b = TensorNode(value=2)

c = a * b

c.backward()
```



BUILDING THE COMPUTATION GRAPH: EAGER EXECUTION

PyTorch, Tensorflow 2.0 default, Keras option

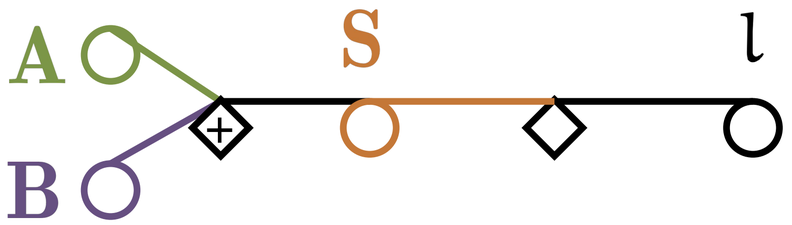
- Build the computation graph on the fly during the forward pass.

Easy to debug, problems in the model occur as the module is executing.
Flexible: the model can be entirely different from one forward to the next.

More difficult to optimize. A little more difficult to serialize.

WORKING OUT THE BACKWARD FUNCTION

```
class Plus(Op):

  def forward(context, a, b):

    # a, b are matrices of the same size

    return a + b

    return goutput, goutput

  def backward(context, goutput):

    ???
```



$$A^{\nabla}_{ij} = \frac{\partial l}{\partial A_{ij}}$$

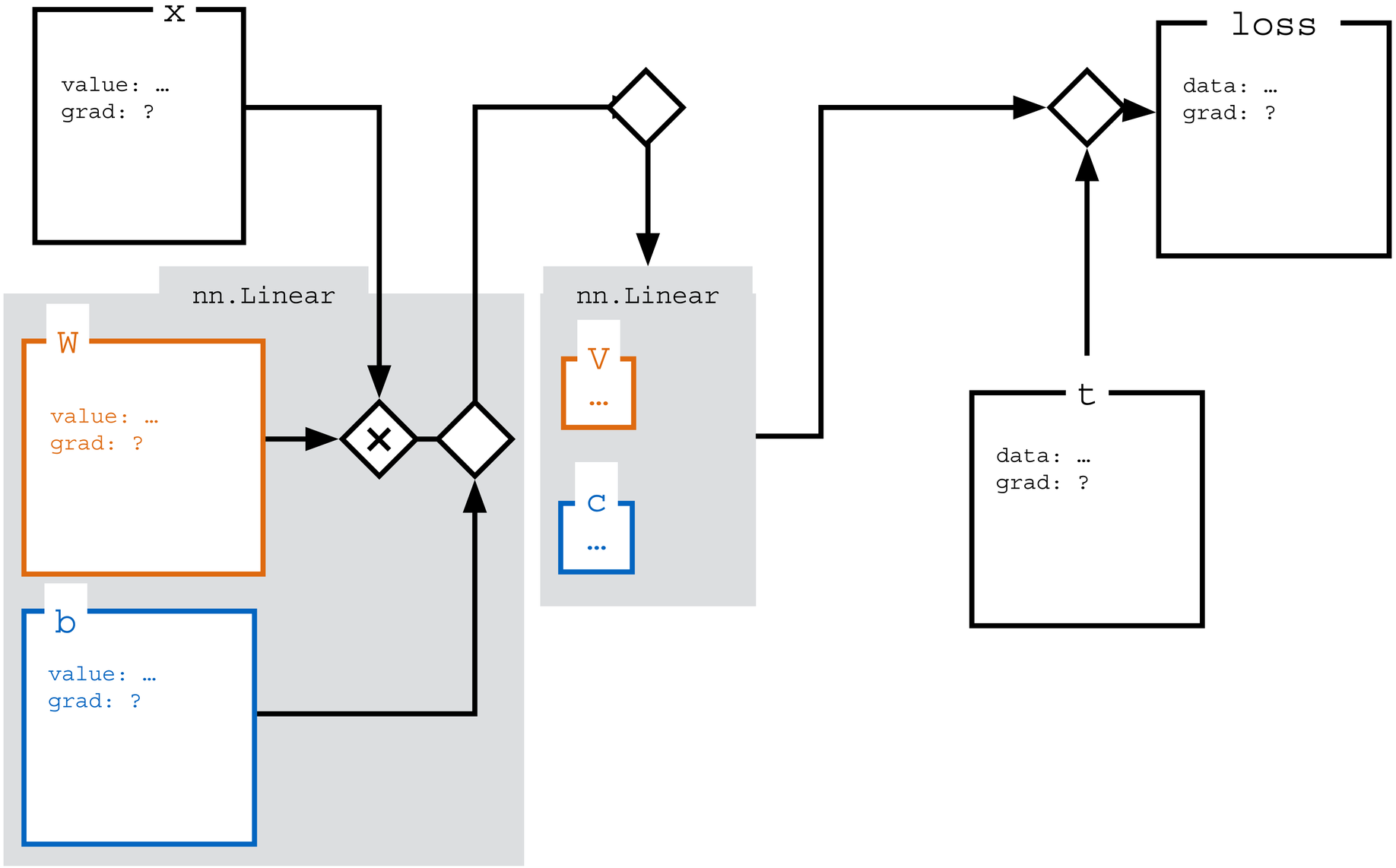
$$= \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial A_{ij}} = \sum_{kl} S^{\nabla}_{kl} \frac{\partial S_{kl}}{\partial A_{ij}}$$

$$= \sum_{kl} S^{\nabla}_{kl} \frac{\partial [\mathbf{A} + \mathbf{B}]_{kl}}{\partial A_{ij}} = \sum_{kl} S^{\nabla}_{kl} \frac{\partial A_{kl} + B_{kl}}{\partial A_{ij}}$$

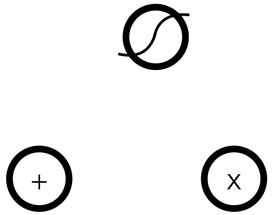
$$= S^{\nabla}_{ij} \frac{\partial A_{ij}}{\partial A_{ij}} = S^{\nabla}_{ij}$$

$$\mathbf{A}^{\nabla} = \mathbf{S}^{\nabla} \qquad \mathbf{B}^{\nabla} = \mathbf{S}^{\nabla}$$

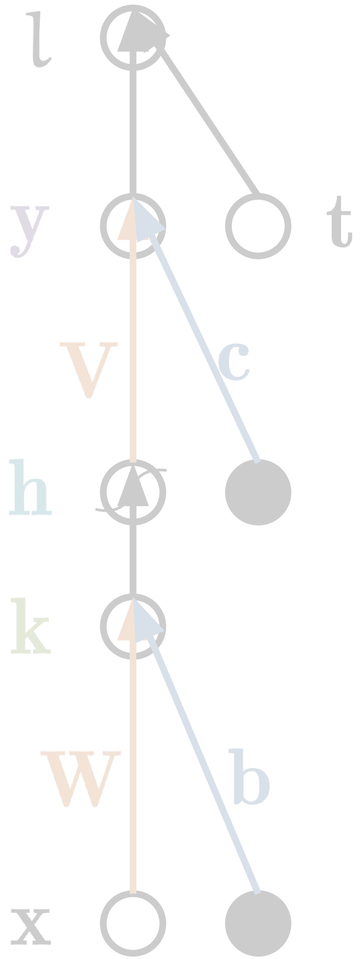
MODULES OR LAYERS



THE IDEAL



pen and paper



$k = w.dot(x) + b$

$h = sigmoid(k)$

$y = v.dot(h) + c$

$l = (y - t) ** 2$

compute forward pass
and
build computation graph

in the computer

```
l.backward() # start backprop
```

VU

~~Tensors~~

Building computation graphs

Working out backward functions

BACKWARD: A FEW MORE EXAMPLES

Sigmoid

Row-wise sum

Expand

SIGMOID

```
class Sigmoid(Op):

  def forward(context, x):

    # x is a tensor of any shape
    sigx = 1 / (1 + (- x).exp())
    context['sigx'] = sigx

  def backward(context, goutput):
    sigx = context['sigx']
    return goutput * sigx * (1 - sigx)
```

$$X^{\nabla}_{ijk} = \sum_{abc} Y^{\nabla}_{abc} \frac{\partial Y_{abc}}{\partial X_{ijk}}$$

$$= \sum_{abc} Y^{\nabla}_{abc} \frac{\partial \sigma(X_{abc})}{\partial X_{ijk}}$$

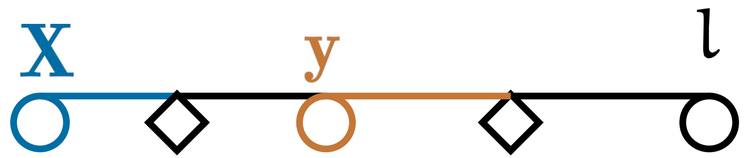
$$= Y^{\nabla}_{ijk} \frac{\partial \sigma(X_{ijk})}{\partial X_{ijk}}$$

$$= Y^{\nabla}_{ijk} \sigma(X_{ijk}) (1 - \sigma(X_{ijk}))$$

$$\mathbf{X}^{\nabla} = \mathbf{Y}^{\nabla} \otimes \sigma(\mathbf{X}) \otimes (1 - \sigma(\mathbf{X}))$$

ROW-WISE SUM

```
class RowSum(Op):

    def forward(context, x):

        # x is a matrix
        context['m'] = x.shape[1]

        sumd = x.sum(axis=1)

        n, m = gy.shape[0], context['m']
        return gy[None, :].expand(n, m)

    def backward(context, gy):

        ???
```



$$X_{ij}^{\nabla} = \sum_{k} y_k^{\nabla} \frac{\partial y_k}{\partial X_{ij}} = \sum_{k} y_k^{\nabla} \frac{\partial \sum_{l} X_{kl}}{\partial X_{ij}}$$

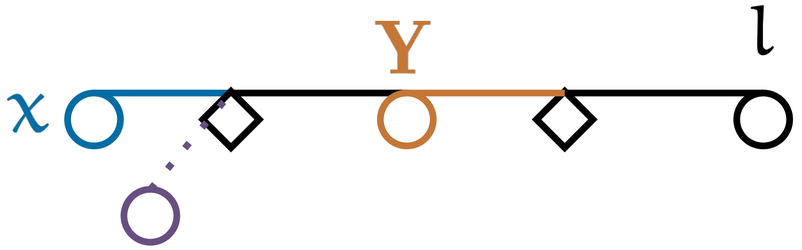
$$= \sum_{kl} y_k^{\nabla} \frac{\partial X_{kl}}{\partial X_{ij}} = y_i^{\nabla} \frac{\partial X_{ij}}{\partial X_{ij}}$$

$$= y_i^{\nabla}$$

$$\mathbf{X}^{\nabla} = \mathbf{y}^{\nabla} \mathbf{1}^{\mathsf{T}}$$

EXPAND

```
class Expand(Op):

  def forward(context, x, size):

    # x is a scalar

    return np.full(x. size=size)
    return gy.sum(), None

  def backward(context, gy):

    ???
```

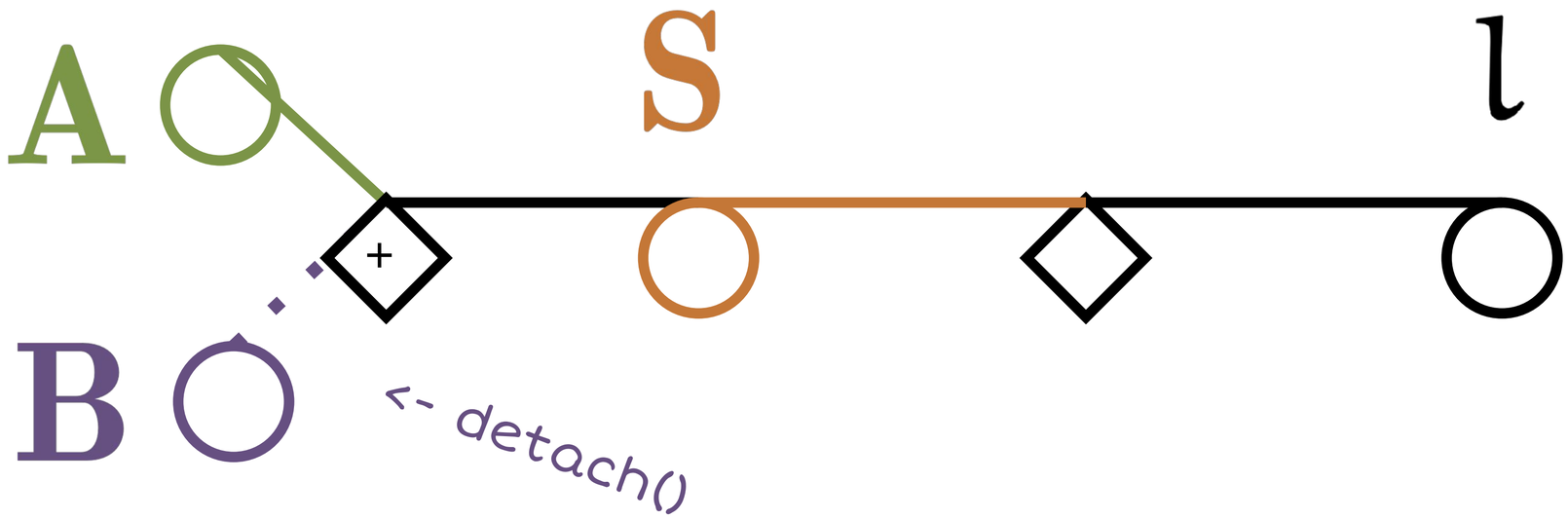


$$x^{\nabla} = \sum_{ab} Y^{\nabla}_{ab} \frac{\partial Y_{ab}}{\partial x}$$

$$= \sum_{ab} Y^{\nabla}_{ab} \frac{\partial x}{\partial x}$$

$$= \sum_{ab} Y^{\nabla}_{ab}$$

DETACH()



~~Tensors~~

Building computation graphs

Working out backward functions

LECTURE RECAP

neural networks: historically inspired by brain structure, but really just a pile of differentiable linear algebra.

backpropagation: work out local gradients symbolically, global gradients numerically. Work from back to front, applying the chain rule.

tensor backpropagation: assume scalar loss l, always compute the gradient of the loss.

automatic differentiation: Build computation graph on the fly. Backpropagation as a *breadth-first* walk down from the loss.

THANK YOU FOR YOUR ATTENTION

dlvu@peterbloem.nl

target class 1 0

softmax activation

sigmoid activation