MSE | MASTER OF SCIENCE IN ENGINEERING

# Back-Propagation

TSM_DeepLearn

Jean Hennebert
Martin Melchior

# Overview

Recap from last week

Computational Graphs

Chain Rule of Calculus

Back-Propagation

# Recap from Last Week

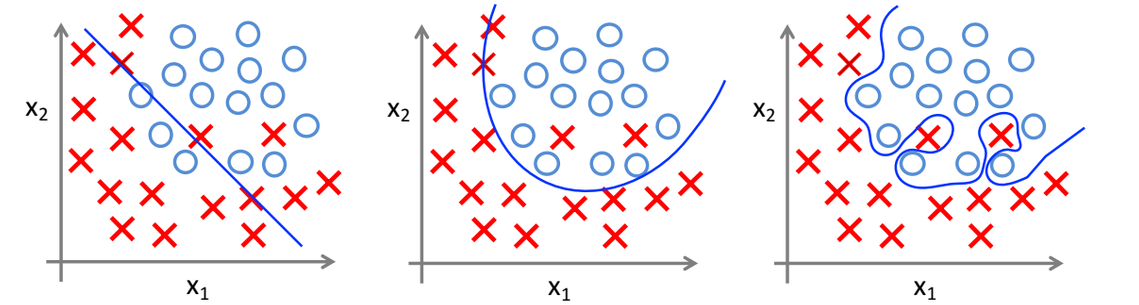
Overfitting

Model Selection

Performance Measures

Curse of Dimensionality

# Overfitting




$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

( $g$ = sigmoid function)

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots)$

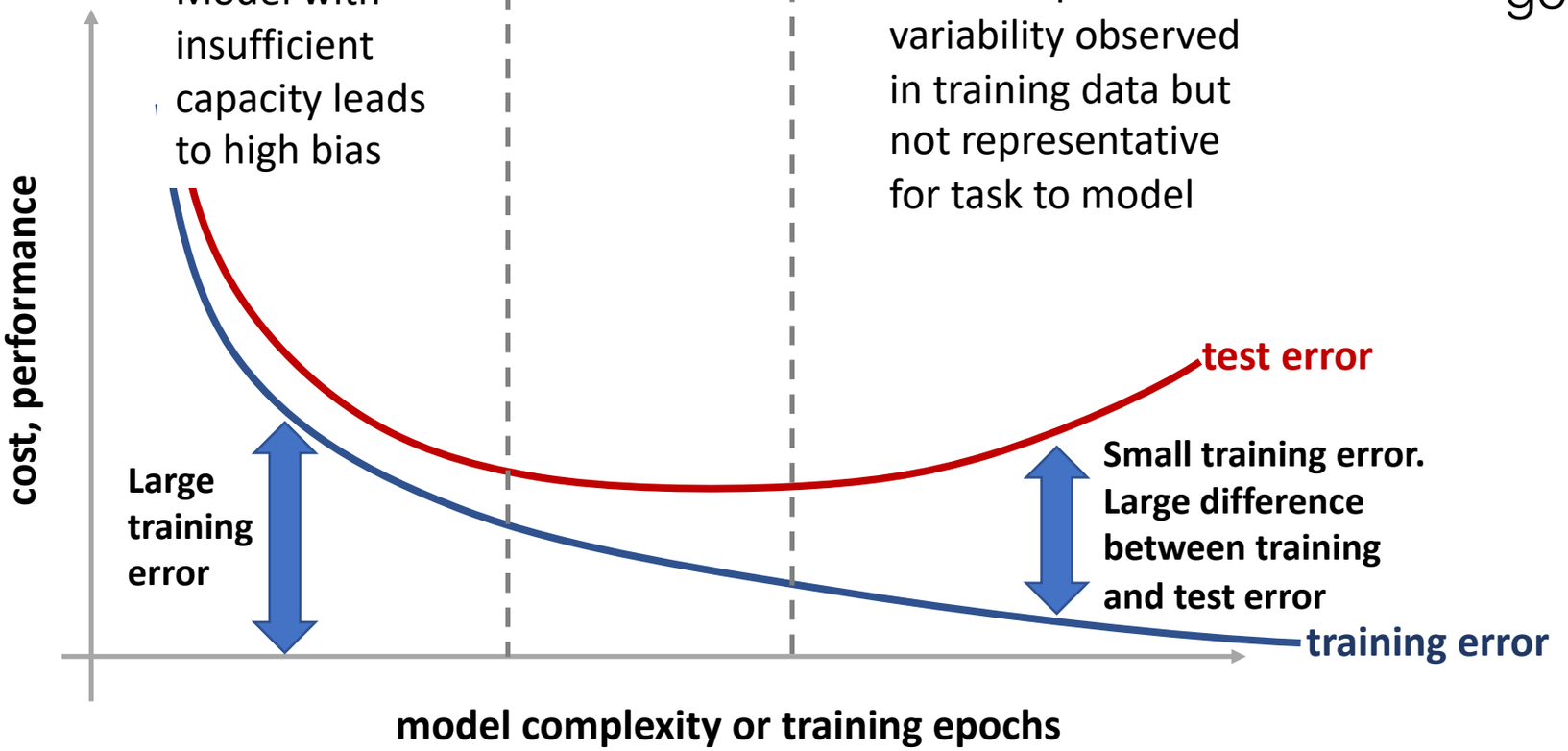
**Underfitting**

Model with insufficient capacity leads to high bias

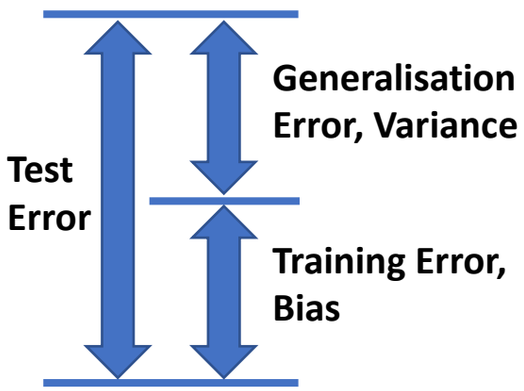
**Good Fit**

**Overfitting**

Model captures variability observed in training data but not representative for task to model




Overfitting occurs when the learned hypothesis (trained model) fits the training data set very well - but fails to generalise to new examples.

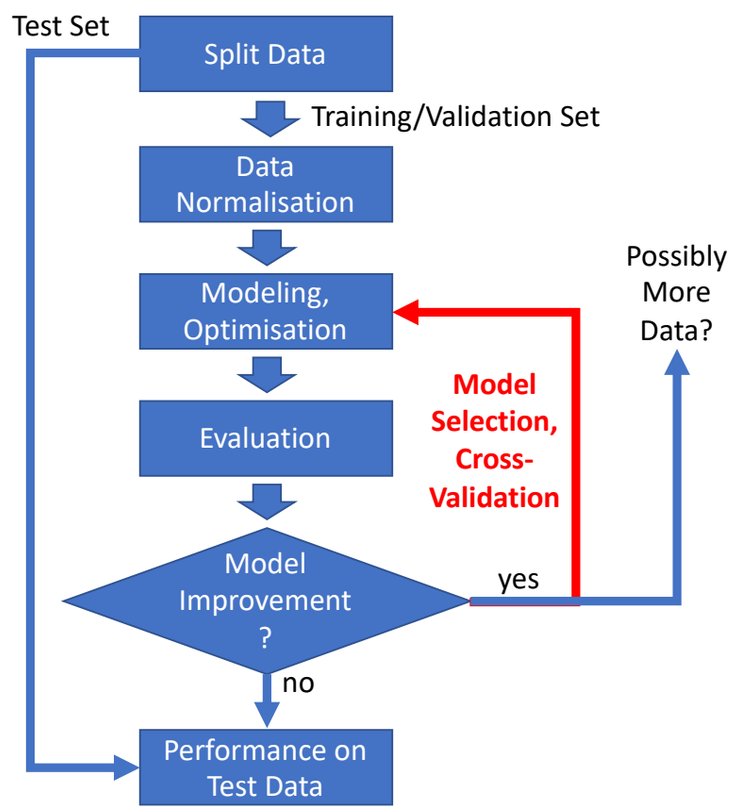


# Model Selection / Performance Measures

| | |
|---|---|
| training set | for tuning model parameters |
| validation set | for tuning model hyper-parameters |
| test set | for measuring performance |

**See ML Course**

## k-fold cross validation

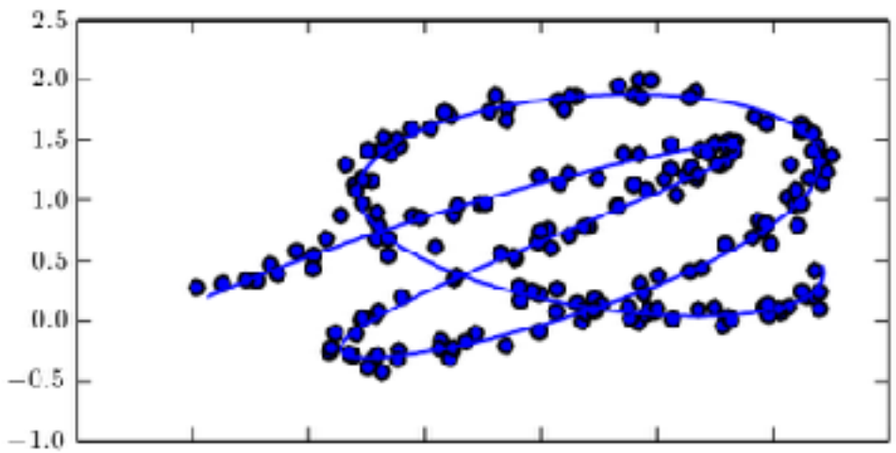



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 902 | 0 | 3 | 3 | 3 | 11 | 5 | 4 | 3 | 2 |
| **1** | 0 | 1119 | 10 | 4 | 1 | 8 | 1 | 1 | 14 | 5 |
| **2** | 3 | 12 | 872 | 15 | 13 | 5 | 15 | 13 | 27 | 7 |
| **3** | 3 | 9 | 25 | 896 | 0 | 50 | 2 | 13 | 24 | 16 |
| **4** | 3 | 0 | 6 | 1 | 858 | 3 | 10 | 5 | 8 | 54 |
| **5** | 8 | 5 | 10 | 20 | 9 | 809 | 16 | 6 | 29 | 9 |
| **6** | 1 | 1 | 6 | 1 | 4 | 14 | 978 | 1 | 4 | 3 |
| **7** | 6 | 2 | 11 | 3 | 6 | 1 | 1 | 957 | 0 | 42 |
| **8** | 3 | 18 | 11 | 16 | 6 | 27 | 9 | 4 | 867 | 17 |
| **9** | 5 | 4 | 3 | 8 | 22 | 9 | 0 | 27 | 2 | 912 |

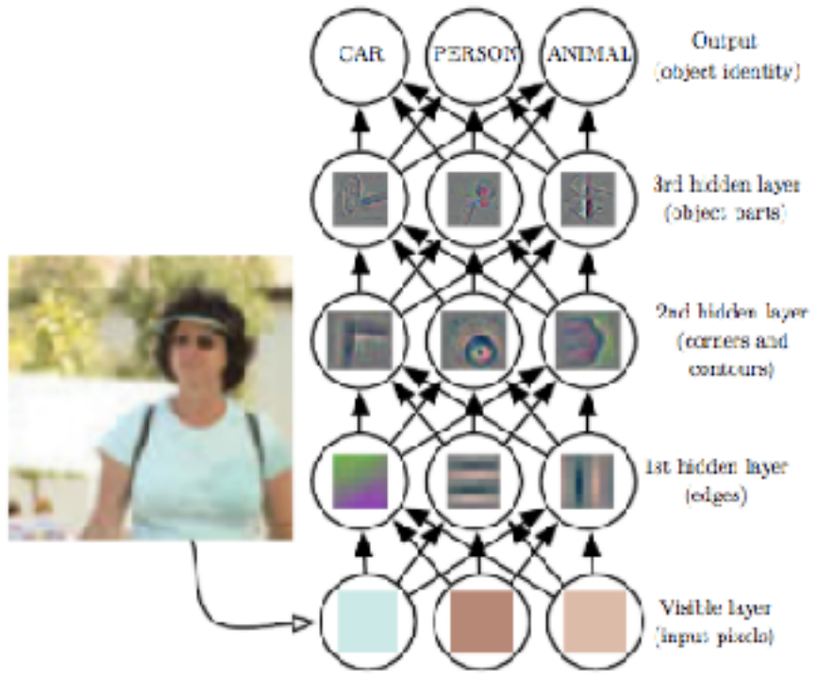
$$
\begin{aligned}
\text{overall accuracy} &= \frac{\#\text{ correctly classified}}{\#\text{ samples}} \\
\text{error rate} &= 1 - \text{overall accuracy} \\
\text{per class accuracy} &= \frac{TP+TN}{\#\text{ samples}} \\
\text{class recall} &= \frac{TP}{TP+FN} \\
\text{class precision} &= \frac{TP}{TP+FP} \\
\text{F-score} &= \ldots
\end{aligned}
$$

# Curse of Dimensionality

Representation on sub-manifolds
in very high dimensional spaces



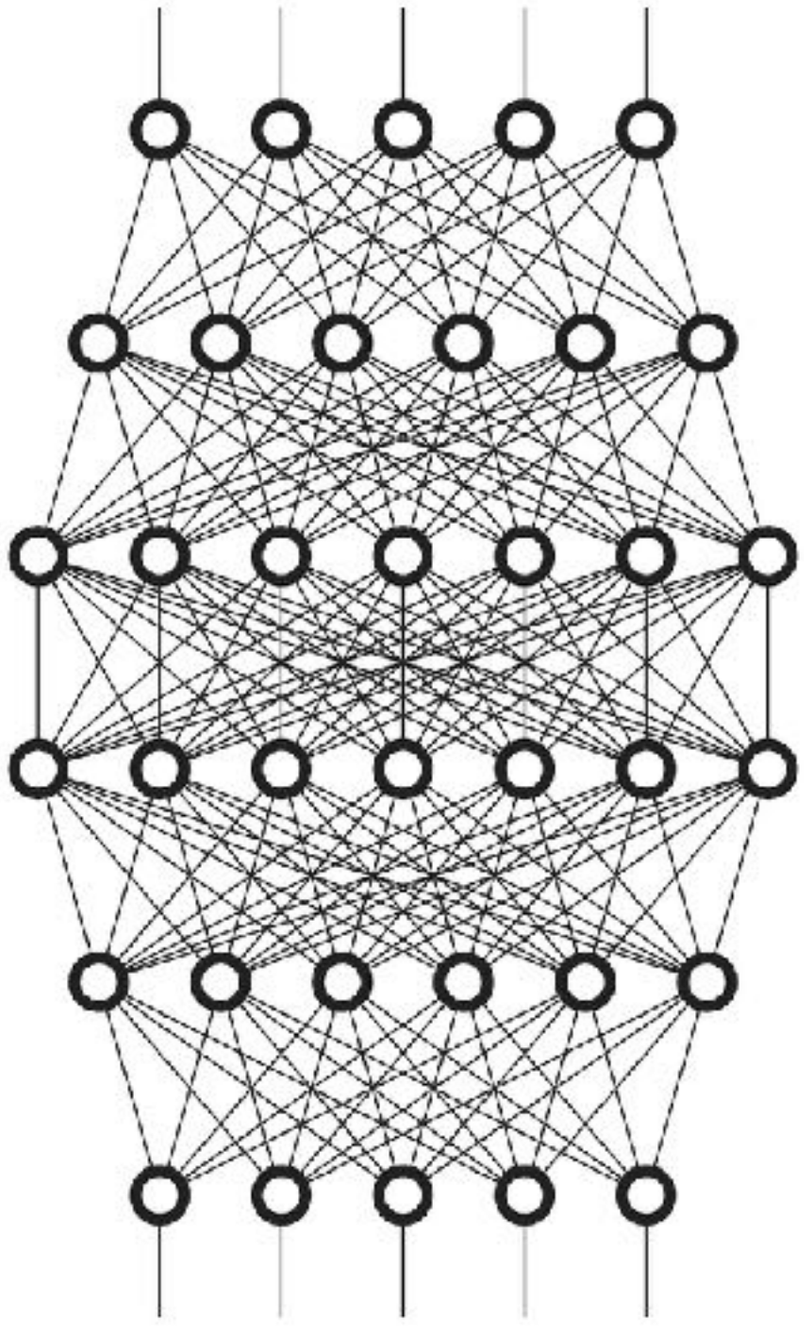

Concept Hierarchy



# Plan for this Week

Implement gradient descent for deep networks.

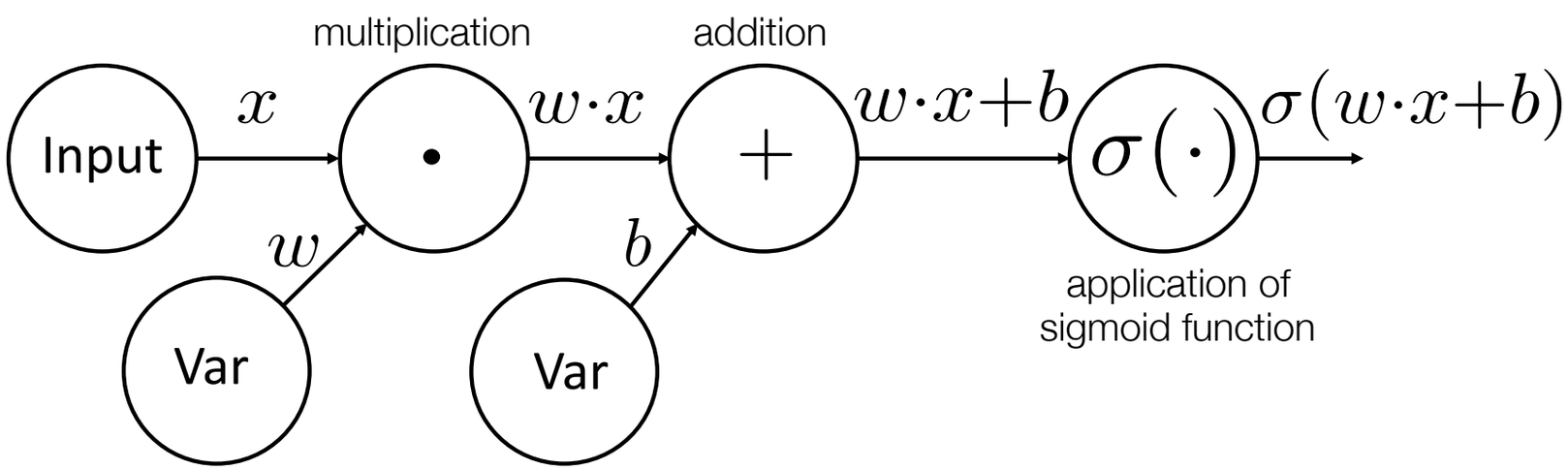
# Computational Graphs



# Computational Graphs

A computational graph is a directed graph where the nodes correspond to operations or variables. Variables can feed their value into operations, and operations can feed their output into other operations. This way, every node in the graph defines a function of the variables.

Example: $g(x; w, b) = \sigma(w \cdot x + b)$



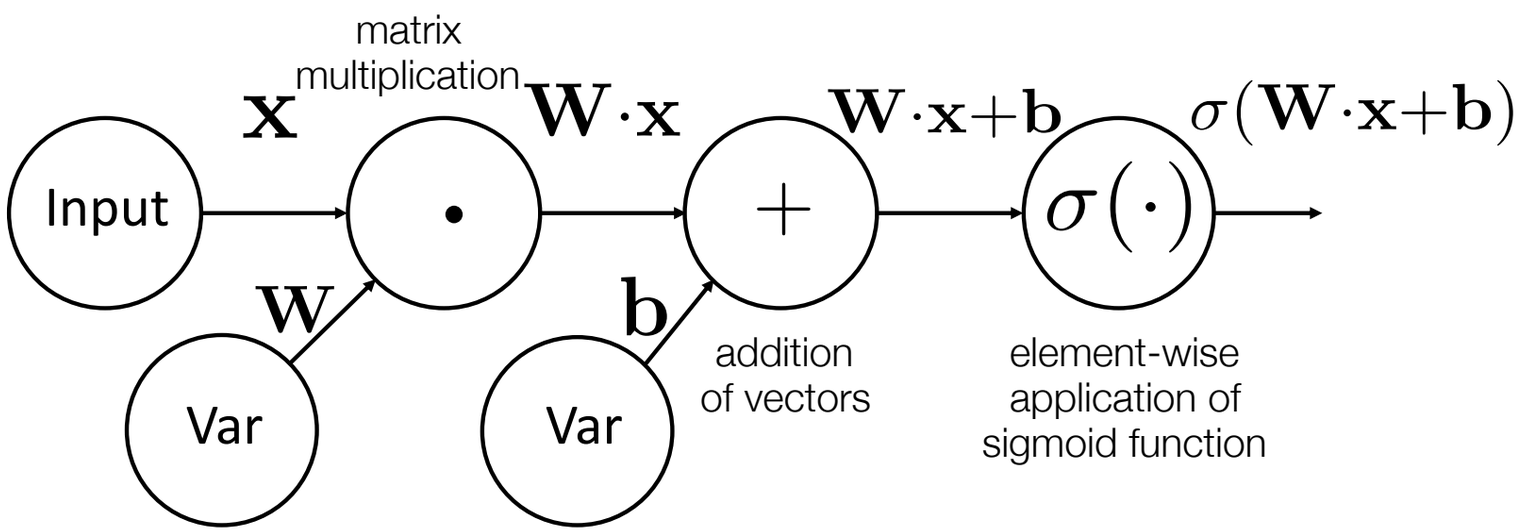

Level of function performed at the nodes is not precisely defined: Simple addition, sigmoid function vector operations, …



See http://www.deepideas.net/deep-learning-from-scratch-i-computational-graphs/

# Computational Graphs of Neural Nets (1)

- Neural networks can be considered as a special form of computational graphs.
- In this context, inputs into and outputs out of the nodes are multi-dimensional arrays (called “tensors”): scalars, vectors, matrices and higher rank objects.
- Example: $g(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$

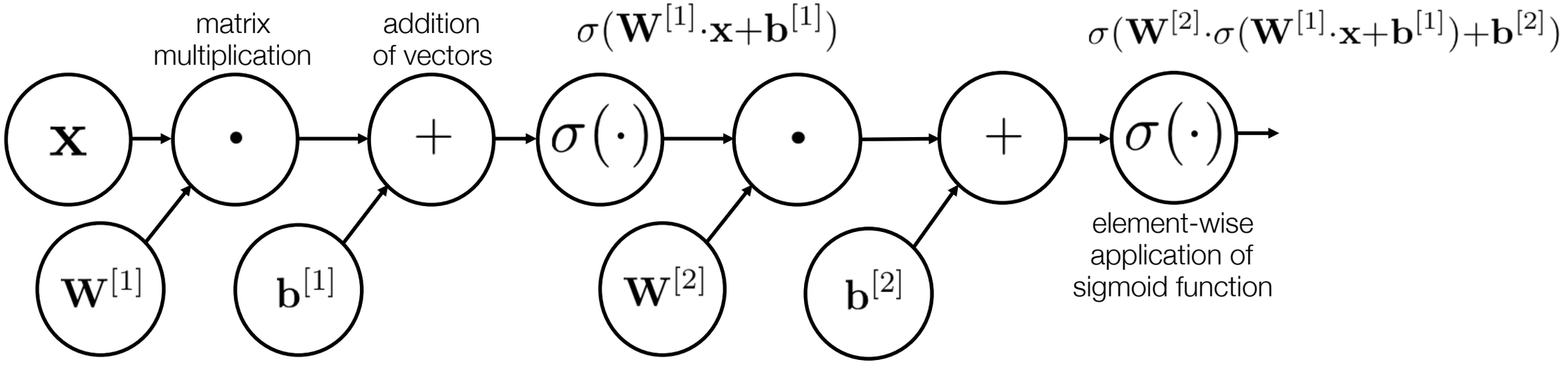



**network consisting just of an output layer**

# Computational Graphs of Neural Nets (2)

Network consisting of one hidden layer and an output layer:

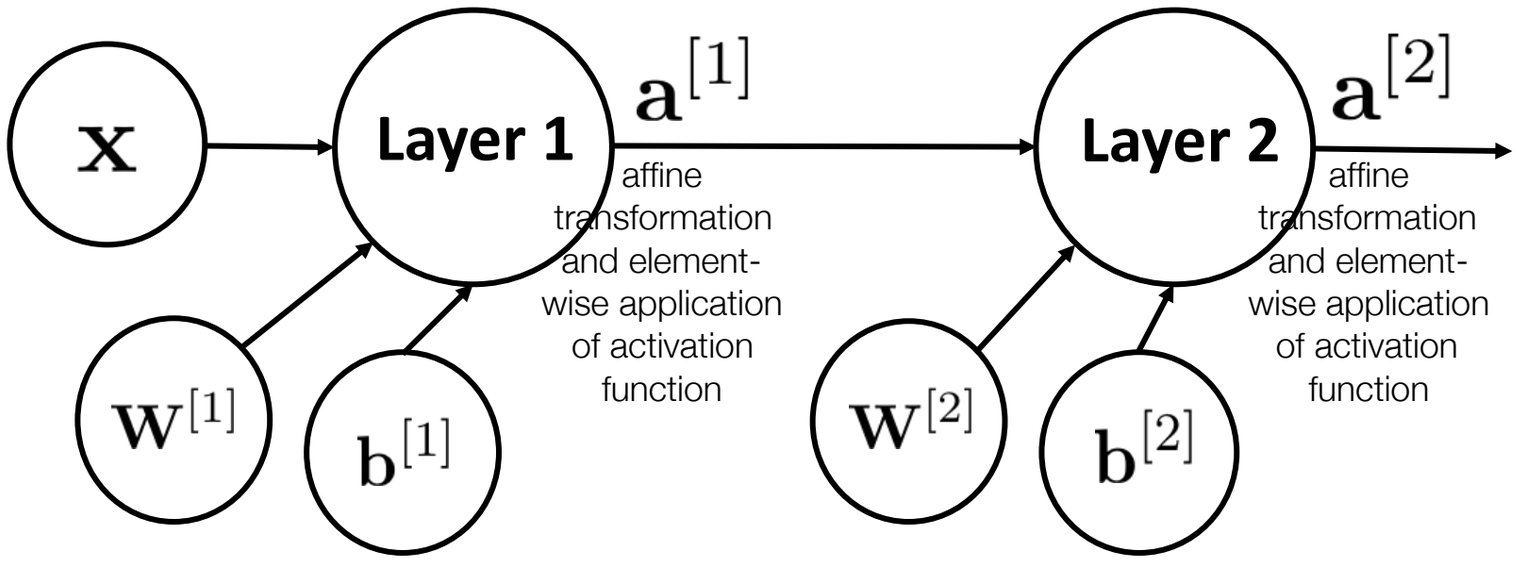
$$g(\mathbf{x}; \mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \mathbf{b}^{[2]}) = \sigma(\mathbf{W}^{[2]} \cdot \sigma(\mathbf{W}^{[1]} \cdot \mathbf{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]})$$




Here, the nodes represent operations on vectors/tensors.

# Graphs Composed of Layers

The operations of a single layer (affine transformation and application of activation function) can be collapsed into a single node:




Layers are the abstraction level most frameworks use to define the architecture of a DL model.

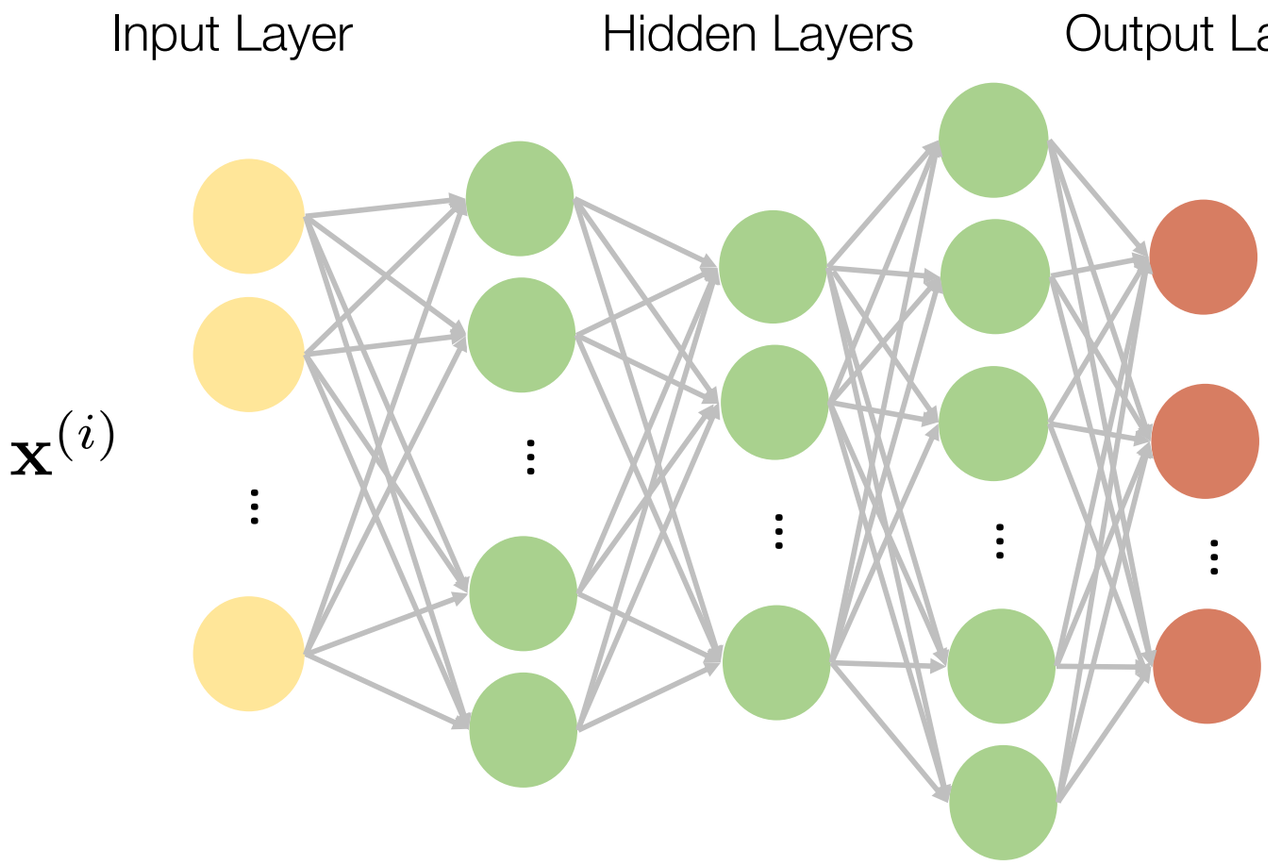
**Example `keras`**

Fully connected perceptron layer instantiated as `Dense`

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
model.add(Dense(64, activation='sigmoid', input_dim=784))
model.add(Dense(64, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))
```

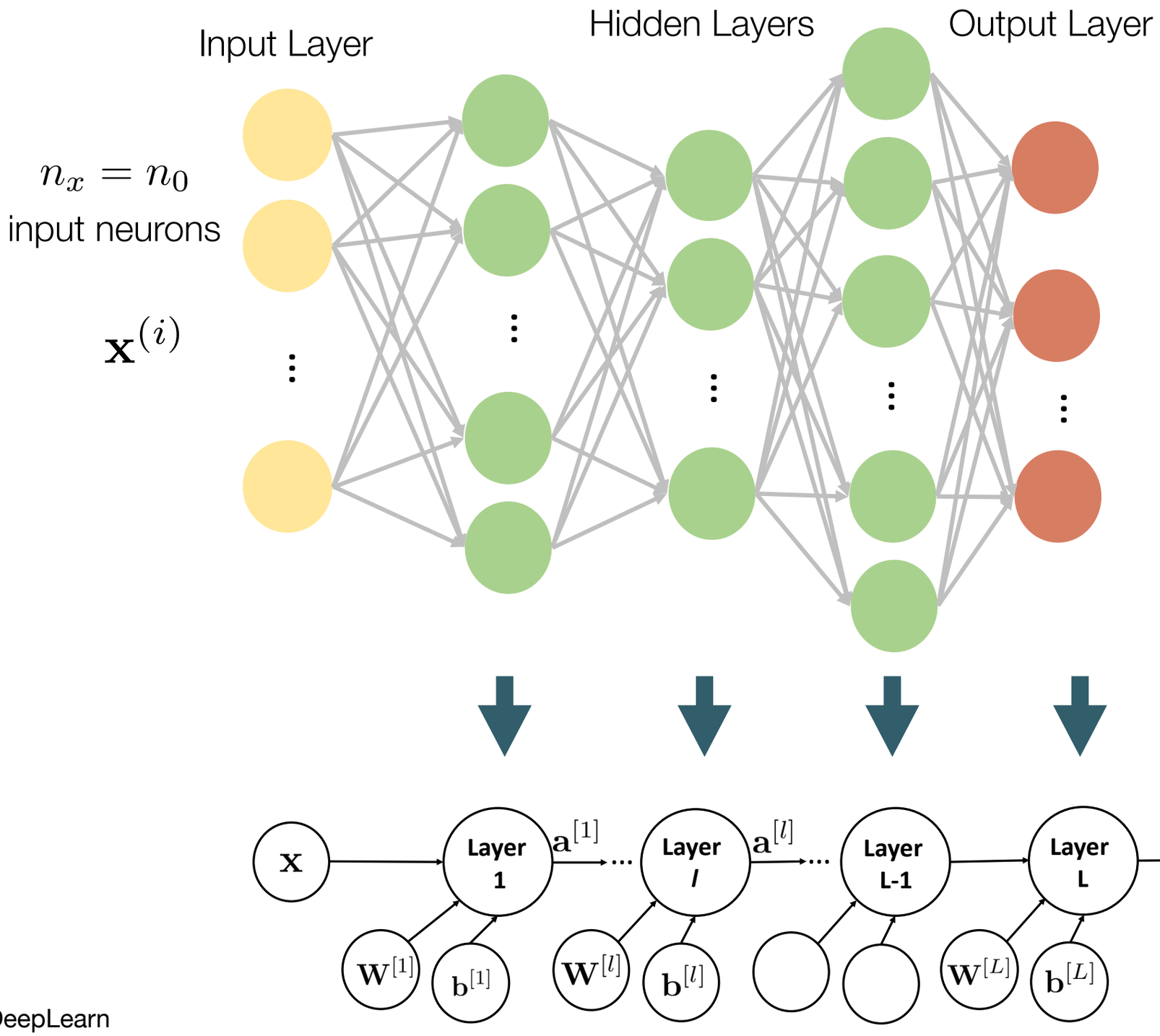
# Multi-Layer Perceptron (MLP)

Represents function that computes from a sample *(i)* of $n_x = n_0$ input features represented by an $n_x$-vector $\mathbf{x}^{(i)}$ an output vector $\hat{\mathbf{y}}^{(i)}$.




Here, the nodes represent neural units which operate on output originating from many other units and produce scalar output.

# Unit-/Layer-Wise Representation







Inputs to or outputs from all units of a layer represented by vectors.

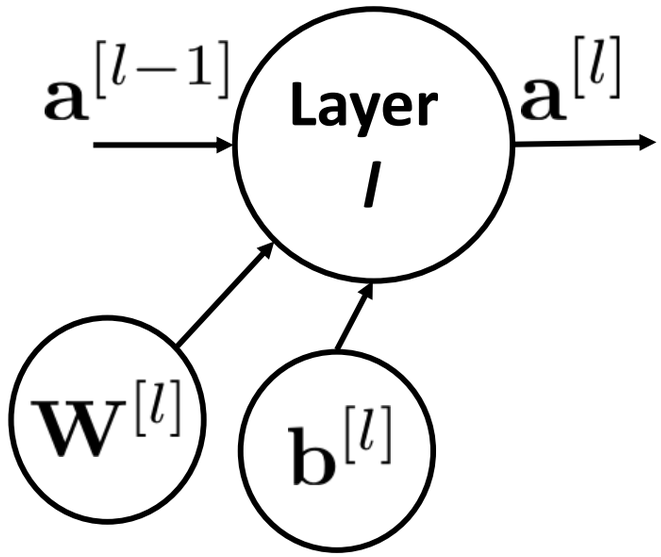
All units of one layer represented as a single node of a computational graph.

# Notations for Layer in an MLP

See Notations Sheet

**activations** in layer *l -1* (column vector with $n_{l-1}$ elements)

$$\mathbf{a}^{[l-1]} = \begin{pmatrix} a_1^{[l-1]} \\ a_2^{[l-1]} \\ \vdots \\ a_{n_{l-1}}^{[l-1]} \end{pmatrix}$$




**activations** in layer *l* (column vector with $n_l$ elements)

$$\mathbf{a}^{[l]} = \begin{pmatrix} a_1^{[l]} \\ a_2^{[l]} \\ \vdots \\ a_{n_l}^{[l]} \end{pmatrix}$$

$$\mathbf{W}^{[l]} = \begin{pmatrix} w_{11} & \dots w_{1n_{l-1}} \\ w_{21} & \dots w_{2n_{l-1}} \\ \vdots & \\ w_{n_l 1} & \dots w_{n_l n_{l-1}} \end{pmatrix}$$

$n_l \times n_{l-1}$-matrix

$$\mathbf{b}^{[l]} = \begin{pmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n_l} \end{pmatrix}$$

$n_l$ vector

$$\begin{aligned} \mathbf{z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{a}^{[l]} &= g^{[l]}(\mathbf{z}^{[l]}) \end{aligned}$$

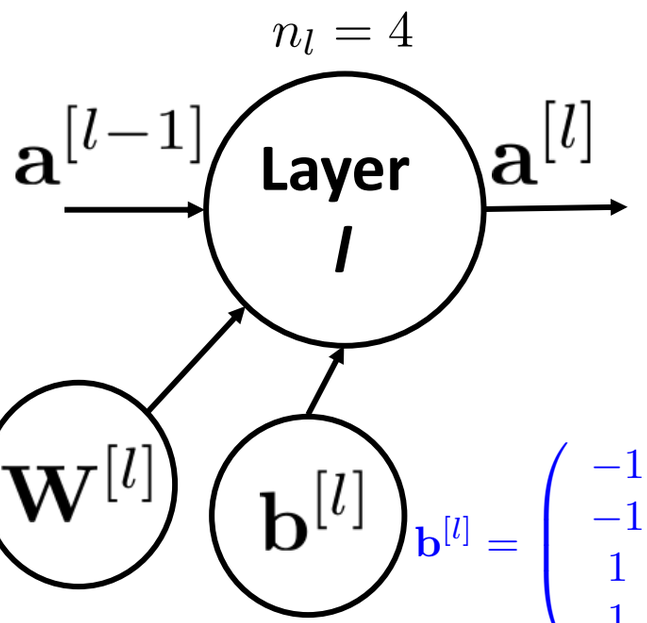
$g^{[l]}(\cdot)$: activation function for layer *l* (applied element-wise).

# Layer in an MLP: Example

$n_{l-1} = 3$

$$\mathbf{a}^{[l-1]} = \begin{pmatrix} 1 \\ -1 \\ -0.5 \end{pmatrix}$$

$3 \times 1$




$$\mathbf{a}^{[l]} = \begin{pmatrix} 0.78 \\ 0.50 \\ 0.56 \\ 0.18 \end{pmatrix}$$

$4 \times 1$

$$\mathbf{W}^{[l]} = \begin{pmatrix} 1 & -1 & -0.5 \\ 1 & 0.5 & -1 \\ -0.2 & 0.3 & 0.5 \\ -1 & 1 & 1 \end{pmatrix}$$

$4 \times 3$

$$\mathbf{b}^{[l]} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

$4 \times 1$

$$\begin{aligned} \mathbf{z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{a}^{[l]} &= g^{[l]}(\mathbf{z}^{[l]}) \end{aligned}$$

**sigmoid activation function**

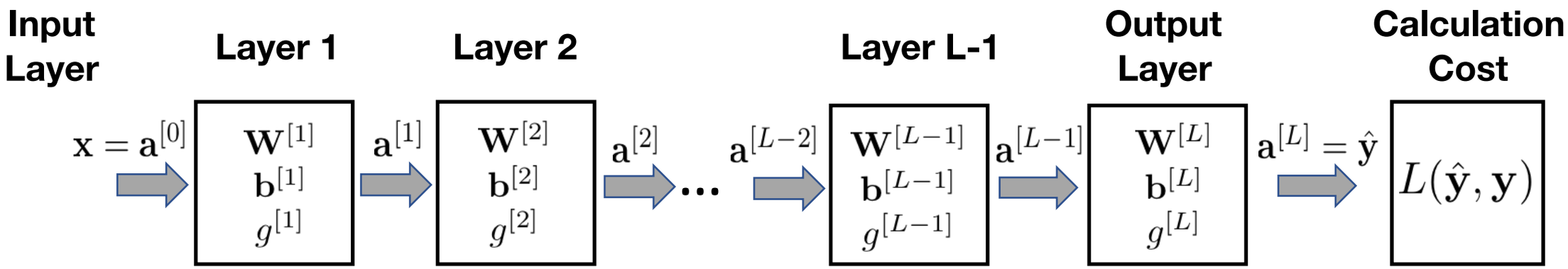
$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} = \begin{pmatrix} 1.25 \\ 0. \\ 0.25 \\ -1.5 \end{pmatrix}$$

$$\mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]}) = \begin{pmatrix} 0.78 \\ 0.50 \\ 0.56 \\ 0.18 \end{pmatrix}$$

$4 \times 1$

```
import numpy as np
W = np.array([[1, -1, -0.5],[1, 0.5, -1], [-0.2, 0.3, 0.5],[-1, 1, 1]]).reshape(4,3)
b = np.array([-1, -1, 1, 1]).reshape(4,1)
aprev = np.array([1, -1, -0.5]).reshape(3,1)
z = np.matmul(W,aprev)+b
a = 1./(1.0+np.exp(-z))
```

# Forward Propagation — Chaining Calculation of Activations in MLP Layers




$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]} \cdot \mathbf{a}^{[0]} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]})$$

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]} \cdot \mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$
$$\mathbf{a}^{[2]} = g^{[2]}(\mathbf{z}^{[2]})$$

...

$$\mathbf{z}^{[L-1]} = \mathbf{W}^{[L-1]} \cdot \mathbf{a}^{[L-2]} + \mathbf{b}^{[L-1]}$$
$$\mathbf{a}^{[L-1]} = g^{[L-1]}(\mathbf{z}^{[L-1]})$$

$$\mathbf{z}^{[L]} = \mathbf{W}^{[L]} \cdot \mathbf{a}^{[L-1]} + \mathbf{b}^{[L]}$$
$$\mathbf{a}^{[L]} = g^{[L]}(\mathbf{z}^{[L]})$$

$$L(\hat{\mathbf{y}}, \mathbf{y})$$

(single sample)

# Computing the Output of MLP — Vectorised

So far we have vectorised over the units in a single layer. But we can also vectorise over the samples in a (mini-)batch. This will allow for efficient parallelisation on GPU. The i-th input sample $\mathbf{x}^{(i)}$ is column vector with $n_x = n_0$ elements. We can put all these vectors as columns in a $n_x \times m$ - matrix:

$$\mathbf{X} = \mathbf{A}^{[0]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \dots & \mathbf{x}^{(m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

Similarly, for the activations and the logits in layer $l = 1, \dots, L$ ( $n_l \times m$ -matrices ):

$$\mathbf{A}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{a}^{[l](1)} & \mathbf{a}^{[l](2)} & \dots & \mathbf{a}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad \mathbf{Z}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)} & \mathbf{z}^{[l](2)} & \dots & \mathbf{z}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

We can now very compactly write the equations for layer $l = 1, \dots, L$ as:

**See Notations Sheet**

$$\begin{aligned} \mathbf{Z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{A}^{[l]} &= g^{[l]}(\mathbf{Z}^{[l]}) \end{aligned}$$

applied element-wise

$$\begin{aligned} \mathbf{A}^{[l-1]} &: n_{l-1} \times m \\ \mathbf{W}^{[l]} &: n_l \times n_{l-1} \\ \mathbf{A}^{[l]} &: n_l \times m \end{aligned}$$

**HERE, BROADCASTING IS INVOLVED!**

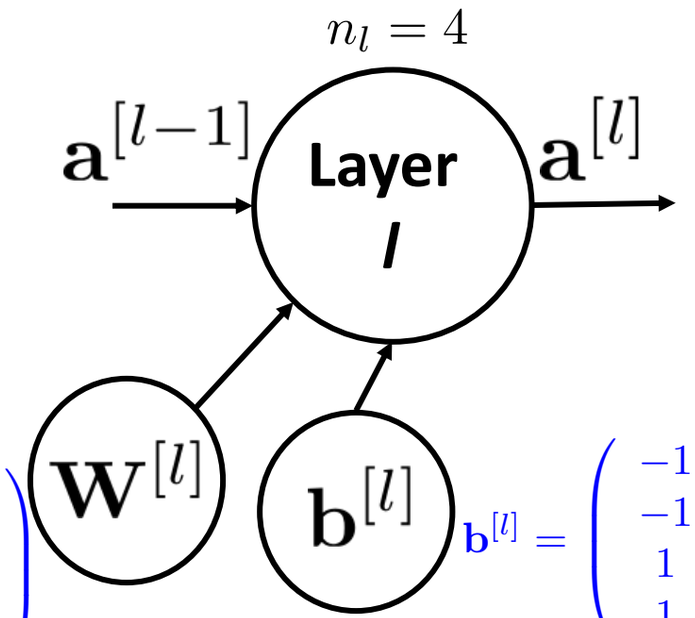
https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html

# Layer in an MLP: Example

$$n_{l-1} = 3 \qquad \mathbf{A}^{[l-1]} = \begin{pmatrix} 1 & -1 \\ -1 & -0.5 \\ -0.5 & 1 \end{pmatrix} \qquad 3 \times 2$$

$m = 2$

m=2, two samples which is the batch size.




$$\mathbf{A}^{[l]} = \begin{pmatrix} 0.78 & 0.12 \\ 0.50 & 0.04 \\ 0.56 & 0.83 \\ 0.18 & 0.92 \end{pmatrix} \qquad 4 \times 2$$

$$\mathbf{W}^{[l]} = \begin{pmatrix} 1 & -1 & -0.5 \\ 1 & 0.5 & -1 \\ -0.2 & 0.3 & 0.5 \\ -1 & 1 & 1 \end{pmatrix} \qquad 4 \times 3$$

$$\mathbf{b}^{[l]} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix} \qquad 4 \times 1$$

$$\begin{aligned} \mathbf{Z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{A}^{[l]} &= g^{[l]}(\mathbf{Z}^{[l]}) \end{aligned}$$

**sigmoid activation function**

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} = \begin{pmatrix} 1.25 & -2.0 \\ 0.0 & -3.25 \\ 0.25 & 1.55 \\ -1.5 & 2.5 \end{pmatrix} \qquad 4 \times 2$$

$$\mathbf{A}^{[l]} = \sigma(\mathbf{Z}^{[l]}) = \begin{pmatrix} 0.78 & 0.12 \\ 0.50 & 0.04 \\ 0.56 & 0.83 \\ 0.18 & 0.92 \end{pmatrix}$$

```
import numpy as np
W = np.array([[1, -1, -0.5],[1, 0.5, -1], [-0.2, 0.3, 0.5],[-1, 1, 1]]).reshape(4,3)
b = np.array([-1, -1, 1, 1]).reshape(4,1)
aprev = np.array([[1, -1, -0.5],[-1, -0.5, 1]]).T.reshape(3,2)
z = np.matmul(W,aprev)+b
a = 1./(1.0+np.exp(-z))
```

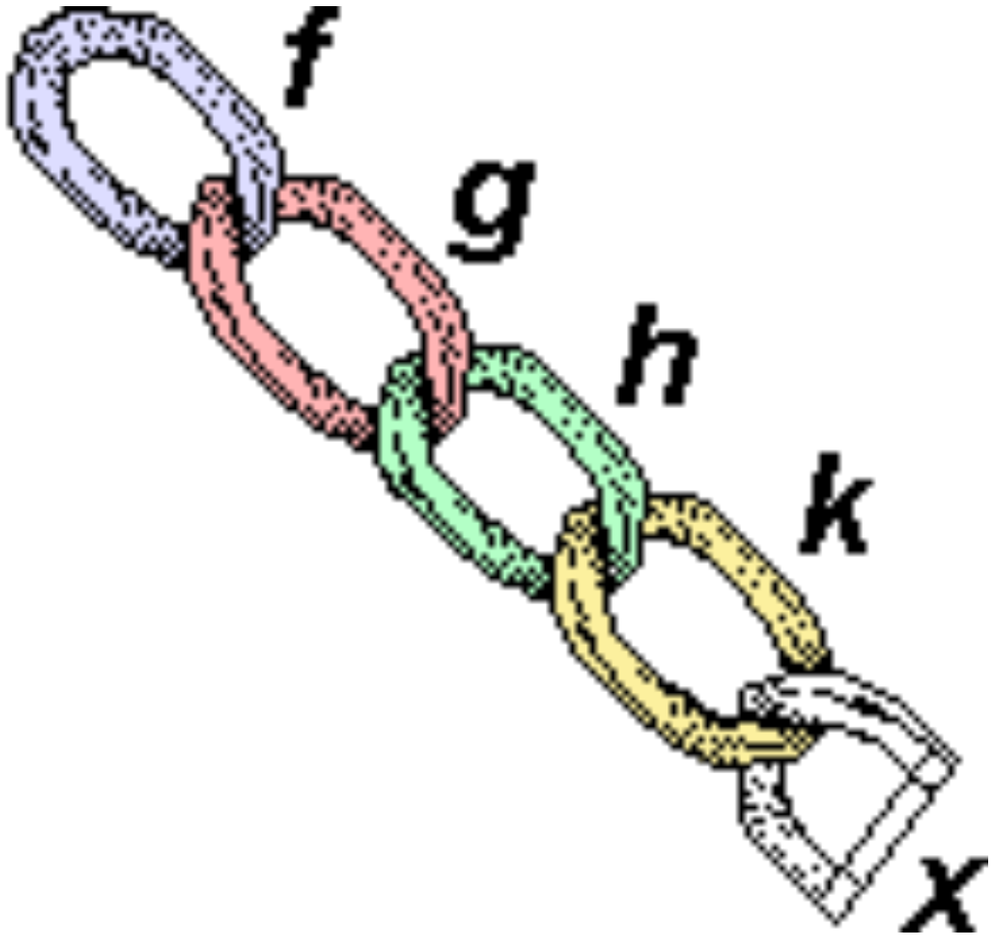
# Implementation in `numpy`

**Some helpful commands:**

| | |
|---|---|
| Create Array with zeros, with shape (m,n) | `np.zeros(shape=(m,n), dtype='float)` |
| Create Array with standard normal, shape (m,n) | `np.random.randn(shape=(m,n))` |
| Reshape array A (e.g. (mn,1)) to (m,n) | `A.reshape(m,n)` |
| Transpose matrix A | `A.T` |
| Matrix-multiplication of A (m,n) and B (n,k) | `np.dot(A,B)` |
| Element-wise multiply of A, B | `np.multiply(A,B)` |
| Sum elements of A along axis 0 (1) | `np.sum(A, axis=1, keepdims=True)` |

Use `numpy` arrays and avoid explicit looping over its elements as far as you can!

# Chain Rule



# Chain Rule in a Few Words

- Compute the amount of change of a function by changing one of its variables.
- If function depends on the variable through a nested structure of inner functions, the change is given by the product of the changes in the nested components.

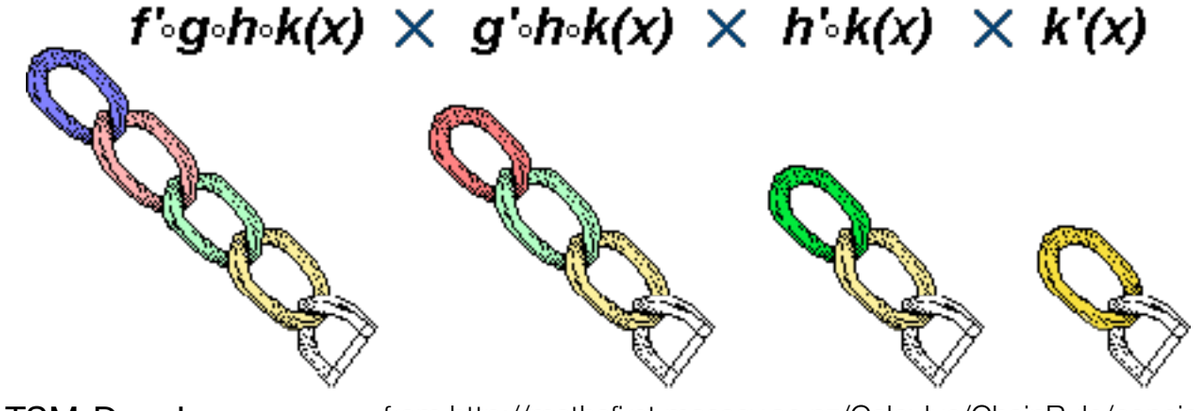
$$L = L(g(h(x)) \qquad dL = \frac{\partial L}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x} \cdot dx$$

Nested structure:
L depends on x only
through g and h

Change of function value

Cascade of changes in its nested components

Change in the variable




$$\begin{aligned} dL &\rightarrow \frac{\partial L}{\partial g} dg \\ &\rightarrow \frac{\partial L}{\partial g}\frac{\partial g}{\partial h} dh \\ &\rightarrow \frac{\partial L}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x} dx \end{aligned}$$



from http://mathsfirst.massey.ac.nz/Calculus/ChainRule/concise.html

# Chain Rule and Computational Graph

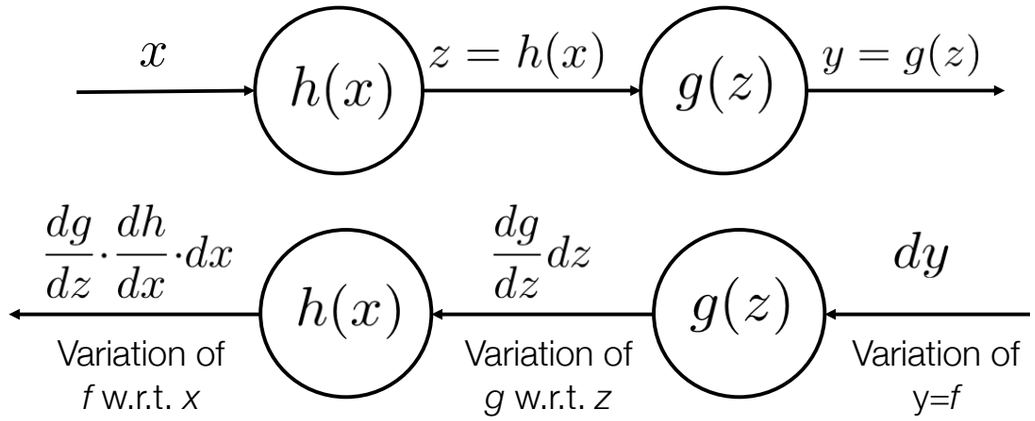
## Computation of derivatives of nested functions

$$f(x) = g(h(x))$$

$$\frac{df}{dx}(x) = \frac{dg}{dz}(z)\Big|_{z=h(x)} \cdot \frac{dh}{dx}(x)$$

!

Variation of $f$ w.r.t. $x$ | Variation of $g$ w.r.t. $z$ | Variation of $h$ w.r.t. $x$

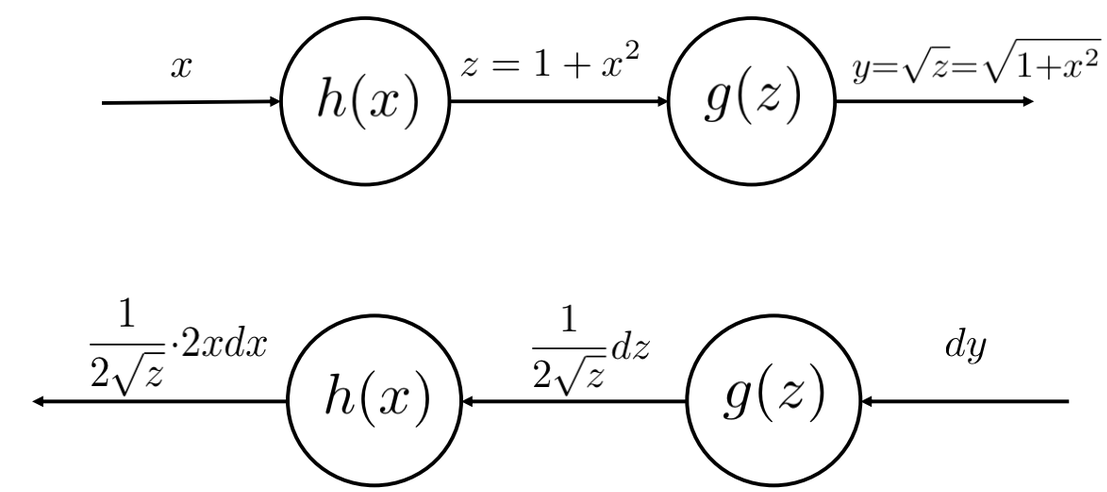



## Example

$$f(x) = \sqrt{1+x^2}, \quad g(z) = \sqrt{z}, \quad h(x) = 1 + x^2$$

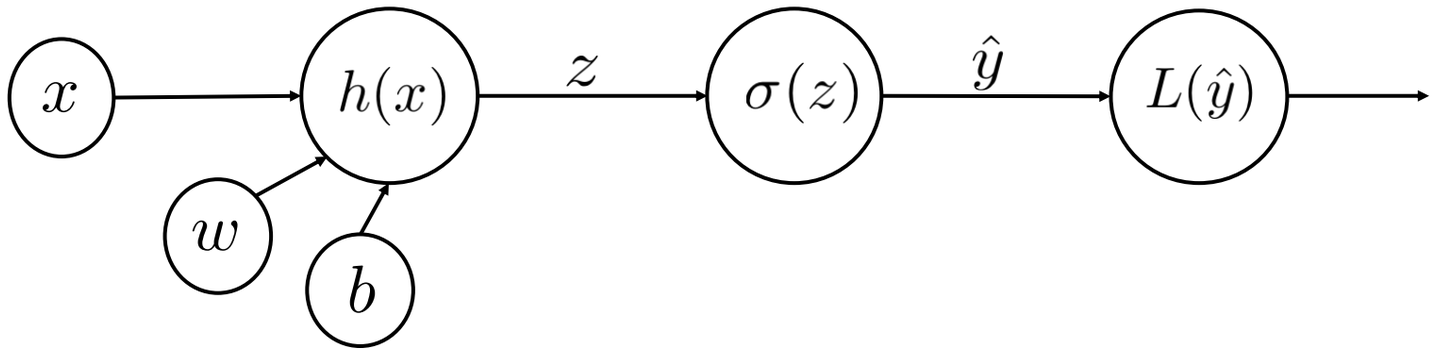
$$\frac{dg}{dz} = \frac{1}{2\sqrt{z}}, \frac{dh}{dx} = 2x$$

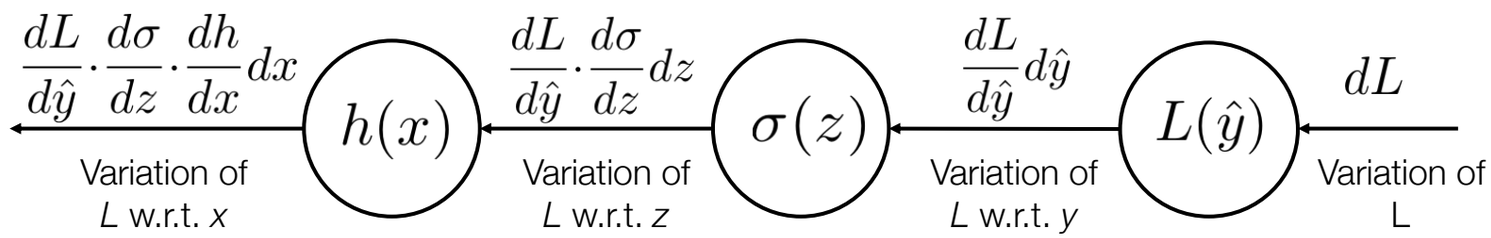
$$\frac{df}{dx}(x) = \frac{dg}{dz}(z) \cdot \frac{dh}{dx} = \frac{1}{2\sqrt{z}} \cdot 2x = \frac{x}{\sqrt{1+x^2}}$$



# Example

$$x \to z = h(x) = \omega \cdot x + b \to \hat{y} = \sigma(z) \to L(\hat{y})$$
$$z{=}h(x){=}\omega{\cdot}x{+}b \qquad \hat{y}{=}\sigma(z) \qquad L(\hat{y}) = g(x;\omega,b)$$



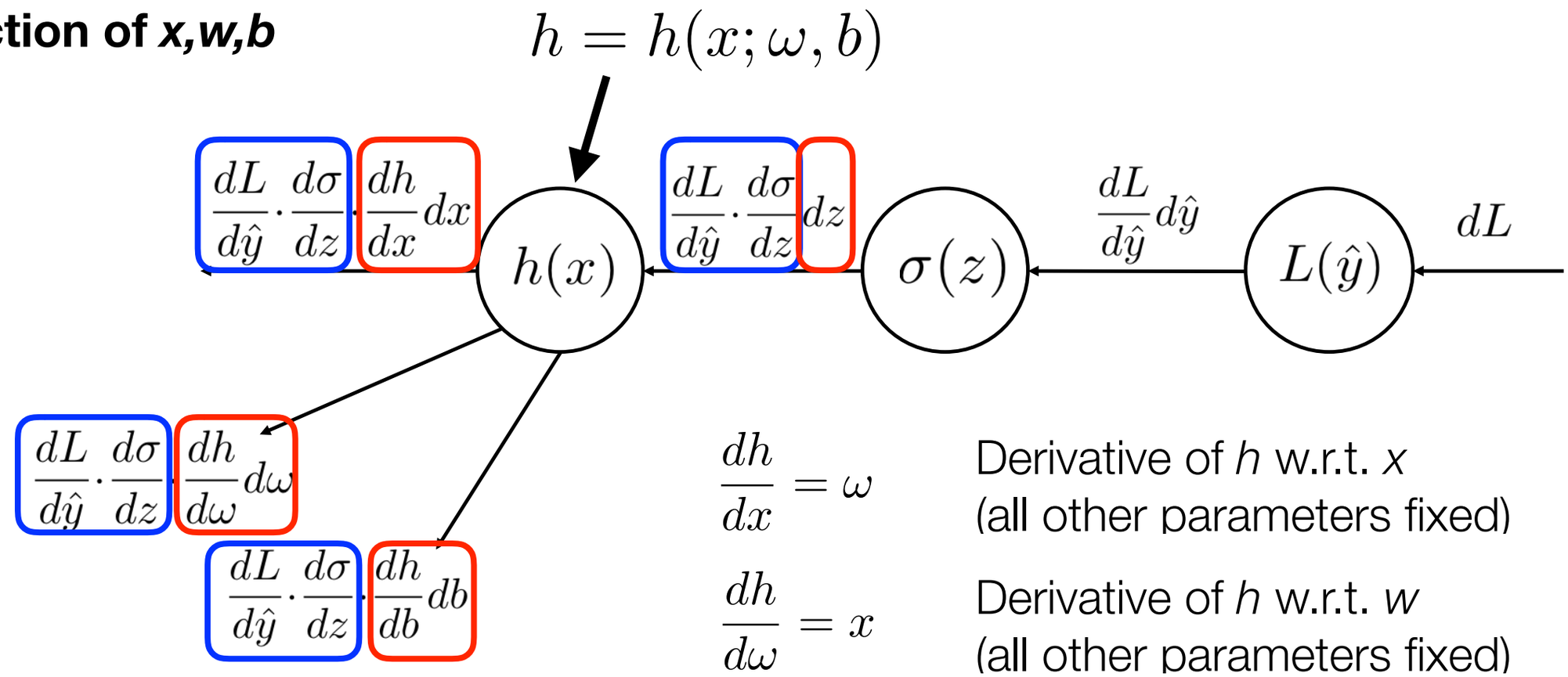




$$\frac{dL}{d\hat{y}} \cdot \hat{y}(1-\hat{y}) \cdot \omega dx \leftarrow \frac{dL}{d\hat{y}} \cdot \hat{y}(1-\hat{y}) dz \leftarrow \frac{dL}{d\hat{y}} d\hat{y} \leftarrow dL$$

$$\frac{dh}{dx} = \omega \qquad \frac{d\sigma}{dz} = \sigma(z)(1-\sigma(z))$$

# Multiple Variables

***h* a function of *x,w,b***




$$\frac{dh}{dx} = \omega$$ Derivative of *h* w.r.t. *x* (all other parameters fixed)

$$\frac{dh}{d\omega} = x$$ Derivative of *h* w.r.t. *w* (all other parameters fixed)

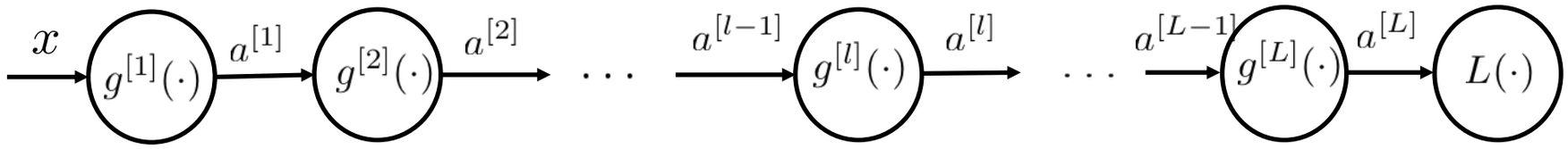
$$\frac{dh}{db} = 1$$ Derivative of *h* w.r.t. *b* (all other parameters fixed)

The expressions in the blue boxes are common to the derivatives w.r.t. all the variables *x,w,b*

$$\frac{dL}{dz} = \frac{dL}{d\hat{y}} \cdot \frac{d\sigma}{dz}$$

**actually, partial derivatives**
**—> multi-variate calculus**
**(more on that later …)**

# Deep Nested Structures



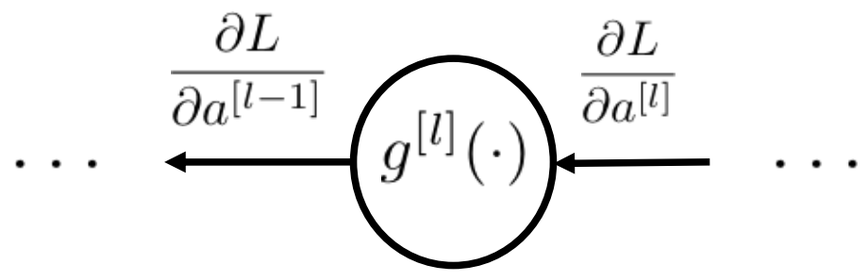

$$L = L(g^{[L]}(g^{[L-1]}(\dots(g^{[l]}(\dots(g^{[1]}(x))\dots))\dots)\dots))$$

for suitable functions $g^{[l]}(\cdot)$

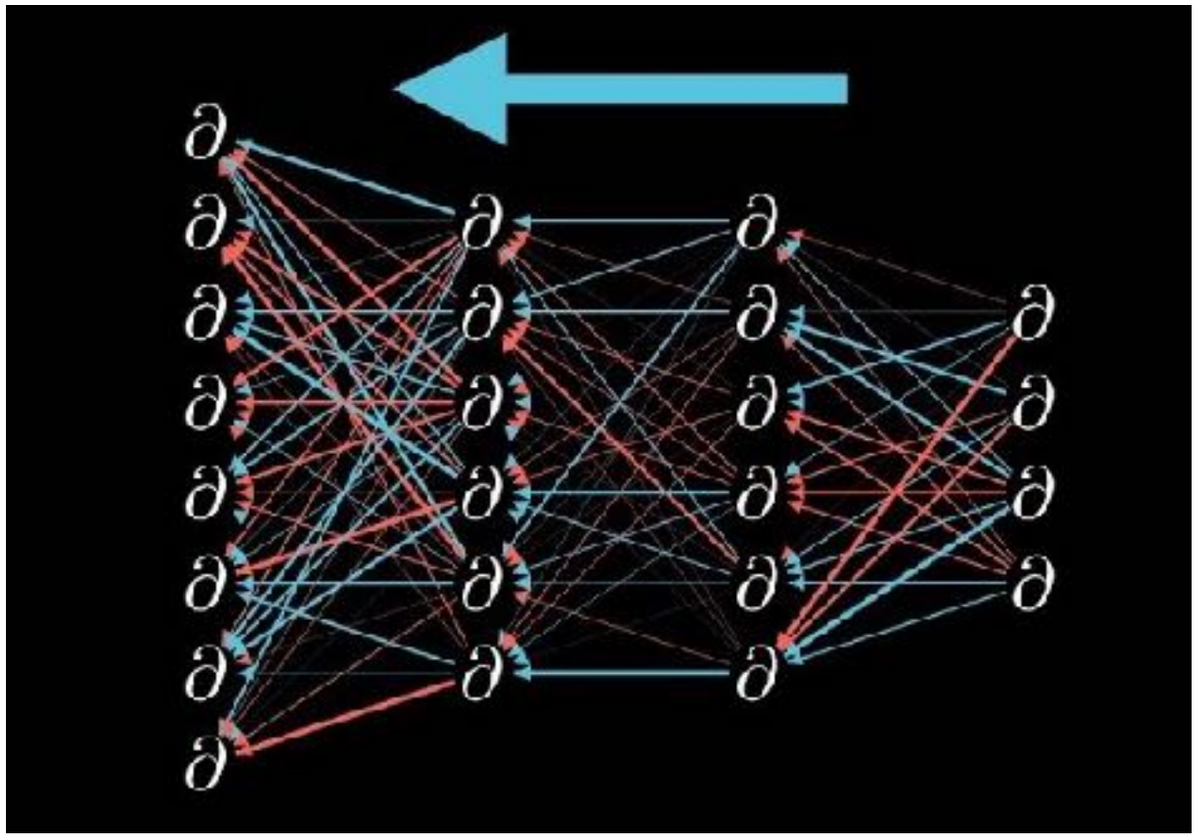
By defining $a^{[l]} = g^{[l]}(a^{[l-1]})$ we compute the derivative of L w.r.t. to $a^{[l-1]}$ as

$$\frac{\partial L}{\partial a^{[l-1]}} = \frac{\partial L}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial a^{[l-1]}} = \frac{\partial L}{\partial a^{[l]}} \cdot \frac{\partial g^{[l]}(a^{[l-1]})}{\partial a^{[l-1]}}$$

which provides the recipe to compute $\frac{\partial L}{\partial a^{[l-1]}}$ from $\frac{\partial L}{\partial a^{[l]}}$ which works backwards.



# Back-Propagation



# Learning

- Learning consists in **minimising the cost** (quadratic for regression problems, cross-entropy for classification)

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(h_\theta(\mathbf{x}^{(i)}), y^{(i)})$$

with respect to the model parameters

$$\theta = (\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \ldots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]})$$

- **Gradient descent** is an iterative scheme to accomplish that.
Update rule with learning rate $\alpha$ :

$$\mathbf{W}^{[l]} \leftarrow \mathbf{W}^{[l]} - \alpha \frac{\partial J}{\partial \mathbf{W}^{[l]}}$$

$$\mathbf{b}^{[l]} \leftarrow \mathbf{b}^{[l]} - \alpha \frac{\partial J}{\partial \mathbf{b}^{[l]}}$$

Hence we need to compute the partial derivatives of the cost function w.r.t. the parameters.

# Back Propagation Algorithm

- While the computation of the output of an MLP consists in a forward propagation of the inputs through the NN — the computation of the partial derivatives of the cost function starts at the end (with the changes in the cost function).

- Back-Propagation Algorithm is an efficient scheme to compute the gradient. In the form suited for applications with neural networks it has been introduced by Rumelhart.(1) In 1974, Werbos (2) mentioned the possibility of applying this principle to artificial neural networks.

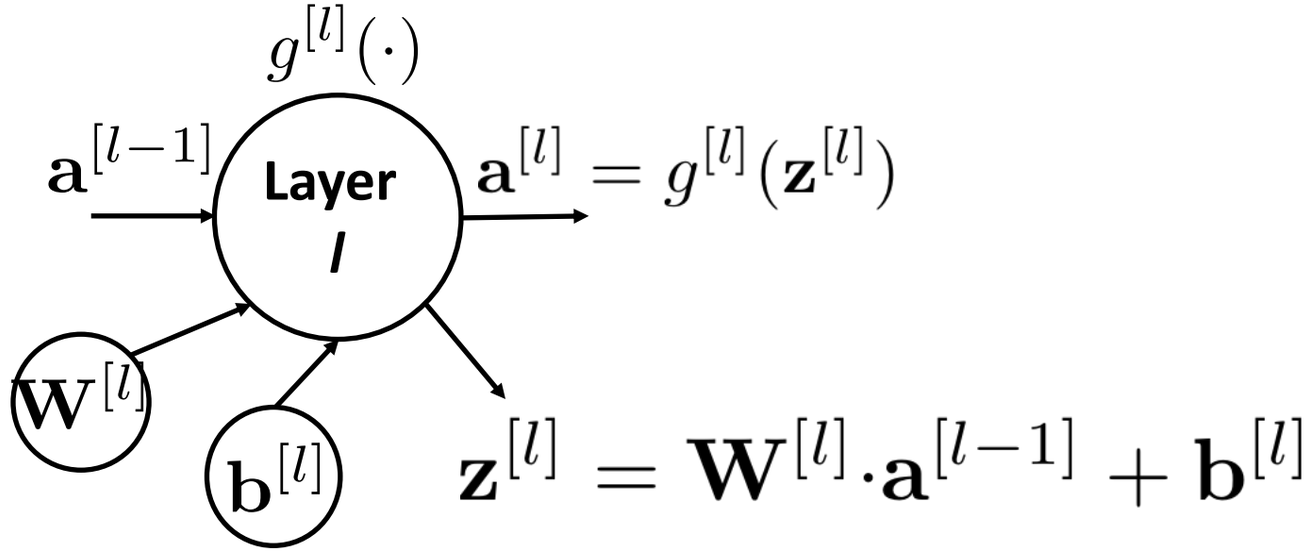
- It is heavily based on the **chain rule** of calculus.

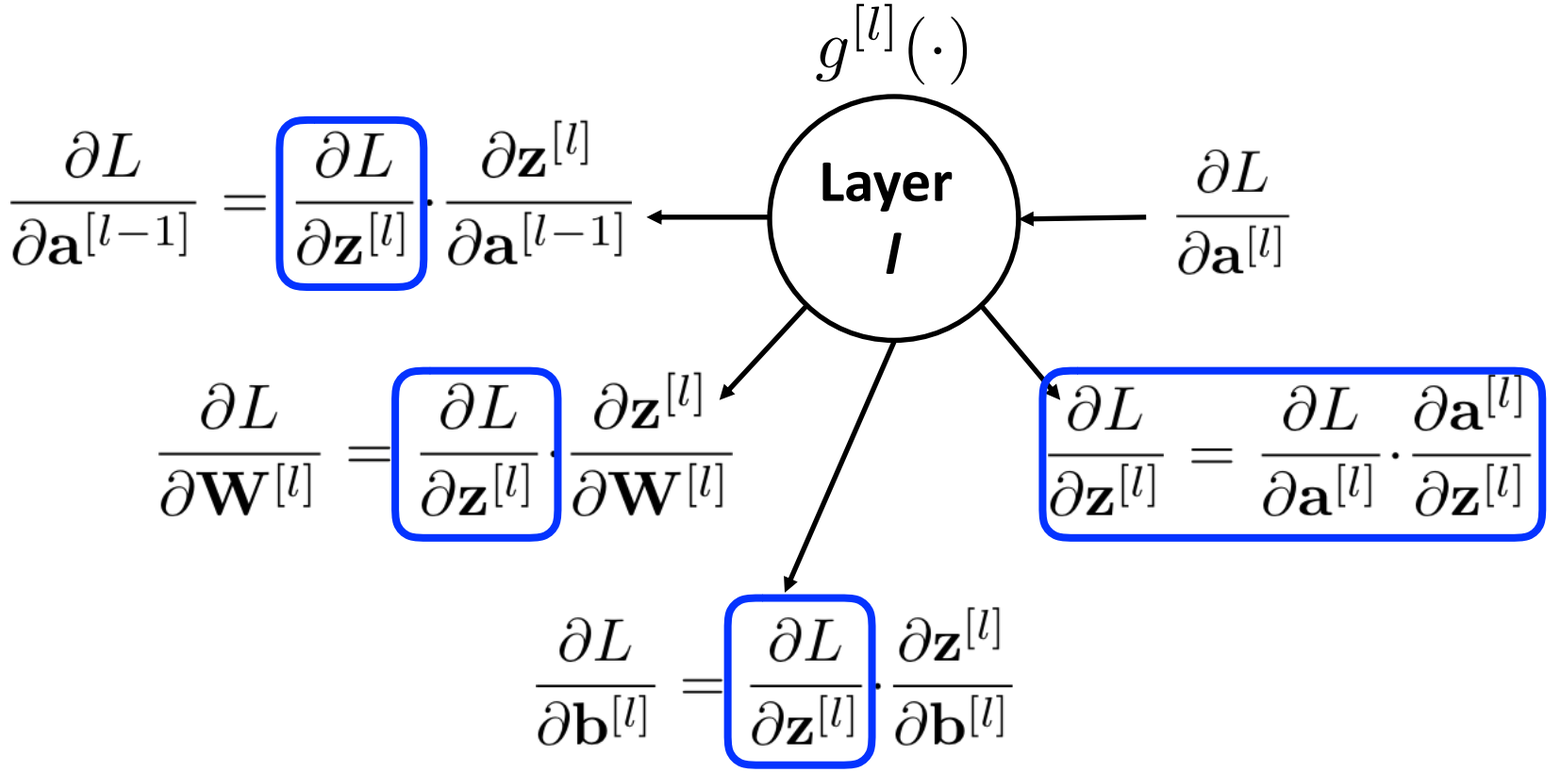
(1) "Learning Internal Representations by Error Propagation", D. Rumelhart, G. Hinton, R. Williams (1986), http://adsabs.harvard.edu/abs/1986Natur.323..533R

(2) "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences", P. Werbos (1975)

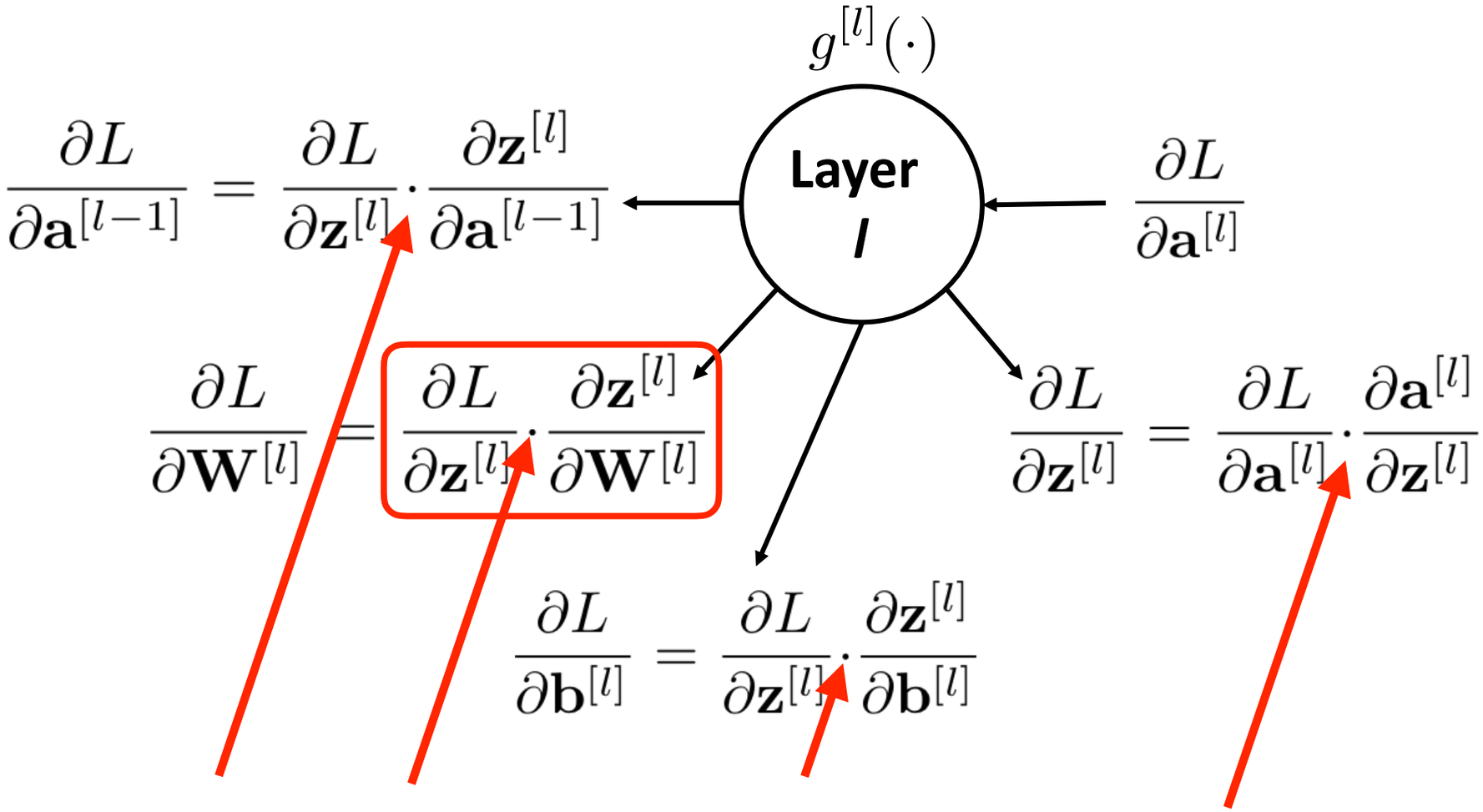
# Backprop through a Single Layer

Forward propagation:



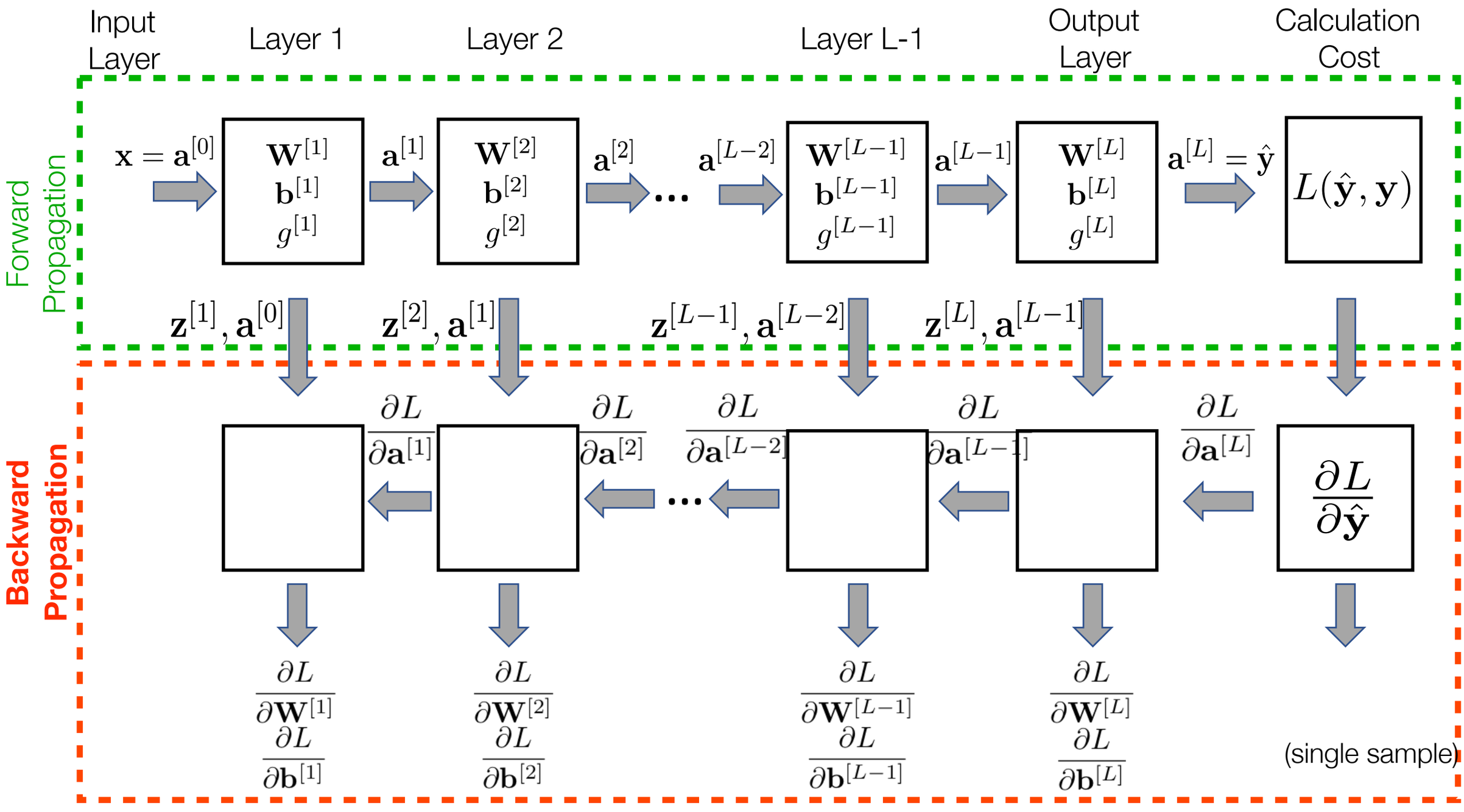
# Backprop through a Single Layer




Sum over contributions of the different components of the vector $\mathbf{z}^{[l]}$ or $\mathbf{a}^{[l]}$, e.g.

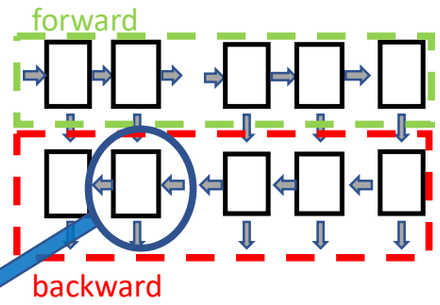
$$\frac{\partial L}{\partial W_{ki}^{[l]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial W_{ki}^{[l]}}$$

# Overview Back-Propagation Scheme




Notation in Andrew Ng's Coursera): $d\mathbf{W}^{[l]} = \frac{\partial L}{\partial \mathbf{W}^{[l]}}$ , …etc.

# Basic Building Block of Backprop




$$\mathbf{a}^{[l-1]}, \mathbf{z}^{[l]}$$

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{a}^{[l-1]}} &= W^{[l]T} \cdot \left(\frac{\partial L}{\partial \mathbf{a}^{[l]}} * g^{[l]\prime}(\mathbf{z}^{[l]})\right) \\
\frac{\partial L}{\partial \mathbf{W}^{[l]}} &= \left(\frac{\partial L}{\partial \mathbf{a}^{[l]}} * g^{[l]\prime}(\mathbf{z}^{[l]})\right) \mathbf{a}^{[l-1]T} \\
\frac{\partial L}{\mathbf{b}^{[l]}} &= \left(\frac{\partial L}{\partial \mathbf{a}^{[l]}} * g^{[l]\prime}(\mathbf{z}^{[l]})\right)
\end{aligned}$$

$\frac{\partial L}{\partial \mathbf{a}^{[l-1]}}$ ← $\qquad$ → $\frac{\partial L}{\partial \mathbf{a}^{[l]}}$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}}, \frac{\partial L}{\partial \mathbf{b}^{[l]}}$$

Notation in Andrew Ng's Coursera: $d\mathbf{a}^{[l]} = \frac{\partial L}{\partial \mathbf{a}^{[l]}}$, $d\mathbf{W}^{[l]} = \frac{\partial L}{\partial \mathbf{W}^{[l]}}$, $d\mathbf{b}^{[l]} = \frac{\partial L}{\partial \mathbf{b}^{[l]}}$

# Derivatives with Output Layer and Loss

## Classification Task with Softmax Layer and Cross Entropy Loss

$$L(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{n_L} y_k \cdot \log(\hat{y}_k) = -\mathbf{y} \cdot \log(\hat{\mathbf{y}})$$

$$\hat{y}_k = \frac{\exp(a_k^{[L]})}{\sum_{j=1}^{n_L} \exp(a_j^{[L]})}$$

Components: $\frac{\partial L}{\partial a_k^{[L]}} = \hat{y}_k - y_k$

Vectorised: $\frac{\partial L}{\partial \mathbf{a}^{[L]}} = \hat{\mathbf{y}} - \mathbf{y}$

## Regression Task with Square Distance Loss

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2} \sum_{k=1}^{n_l} (y_k - \hat{y}_k)^2 = \frac{1}{2} \left\| y - \hat{y} \right\|^2$$

$$\hat{\mathbf{y}} = \mathbf{a}^{[L]}$$

Vectorised: $\frac{\partial L}{\partial \mathbf{a}^{[L]}} = \mathbf{a}^{[L]} - \mathbf{y}$

**Different loss functions have different forms of partial derivatives so that** $\frac{\partial L}{\partial \mathbf{a}^{[L]}}$ **will be different!**

# Vectorised Formulation

Use matrices of the form

$$\mathbf{A}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{a}^{[l](1)} & \mathbf{a}^{[l](2)} & \dots & \mathbf{a}^{[l](N)} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad \mathbf{Z}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)} & \mathbf{z}^{[l](2)} & \dots & \mathbf{z}^{[l](N)} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad (n_l \times N\text{-matrices})$$

with the logits, activations per layer associated with the according input data samples.

As seen before, the equations for the forward propagation are given by

$$\begin{aligned} \mathbf{Z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{A}^{[l]} &= g^{[l]}(\mathbf{Z}^{[l]}) \end{aligned}$$

The backdrop equations can be expressed similarly be using the similar matrices

$$\frac{\partial L}{\partial \mathbf{A}^{[l]}} = \begin{pmatrix} \vdots & & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{a}^{[l](1)}} & \dots & \frac{\partial L}{\partial \mathbf{a}^{[l](2)}} & \dots & \frac{\partial L}{\partial \mathbf{a}^{[l](m)}} \\ \vdots & & \vdots & & \vdots \end{pmatrix}$$

partial derivatives of *L* w.r.t. to the activations in the *l*-th layer, evaluated for the *i*-th sample

# Implementation in `numpy`

**See PW for week 5**

```python
class Layer(object):
    def __init__(self, _):
        self.weights = None  # shape (nunits,nunits_prev)
        self.bias = None  # shape (nunits,1)
        self.activations = None  # shape(nunits,m)
        self.logits = None  # shape(nunits,m)
        self.grad_logits = None  # shape(nunits,m)

    def propagate(self, activations_prev):
        self.logits = self.weights.dot(activations_prev) + self.bias
        self.activations = self.activ_func.compute_value(self.logits)
        return self.activations

    def backpropagate(self, grad_activations):
        self.grad_logits = _
        grad_activations_prev = _
        return grad_activations_prev

    def gradient_weights(self, activations_prev):
        _
        return grad_weights

    def gradient_bias(self):
        _
        return grad_bias
```

cache logits (z)

cache gradient w.r.t. logits (z)

compute gradient w.r.t. to weights and bias by using gradient w.r.t. logits (z)

softmax layer needs to be handled specifically

```python
class MLP(object):
    def __init__(self, units_per_layer, ...):
        self.layers = []
        # instantiate layers of class Layer with given units

    def propagate(self, x):
        a = x
        for layer in self.layers:
            a = layer.propagate(a)
        return a

    def backpropagate(self, grady):
        grad = grady
        for layer in reversed(self.layers):
            grad = layer.backpropagate(grad)
        return grad

    def update_params(self, learning_rate):
        a = self.x
        for layer in self.layers:
            layer.weights -= learning_rate * layer.gradient_weights(a)
            layer.bias -= learning_rate * layer.gradient_bias()
            a = layer.activations
```

mini-batches of images

output of the network - typically, probabilities per class

gradient of cost w.r.t. input to cost function

parameter update according to gradient descent

main loop of gradient descent optimisation of MLP

```python
for iepoch in range(nepochs):
    for ibatch in range(nbatches):
        xbatch,ybatch = dataset.next_batch(batchsize)
        pred_probs = mlp.propagate(xbatch)
        gradJ = cost.compute_derivative(ybatch,pred_probs)
        grad0 = mlp.backpropagate(gradJ)
        mlp.update_params(learning_rate(iepoch))
```

# Gradient Checking

For debugging, it is helpful to check numerically whether the calculation of the gradient of the loss function w.r.t. the weights and the biases is correct.

$$L = L(\dots, W_{kl}^{[l]}, \dots)$$

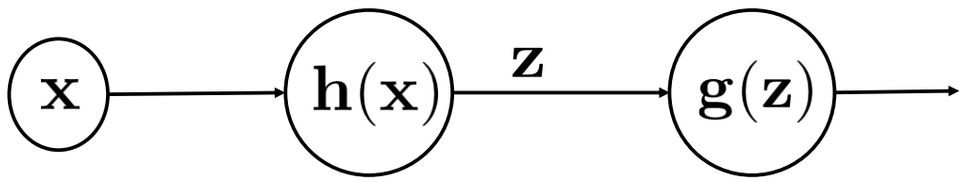
all other variables equal

$$\begin{aligned} \frac{\partial L}{\partial W_{kl}^{[l]}} &= \lim_{\epsilon \to 0} \frac{L(\dots, W_{kl}^{[l]} + \epsilon, \dots) - L(\dots, W_{kl}^{[l]}, \dots)}{\epsilon} \\ &\approx \frac{L(\dots, W_{kl}^{[l]} + \epsilon_0, \dots) - L(\dots, W_{kl}^{[l]}, \dots)}{\epsilon_0} \end{aligned}$$

# Backup

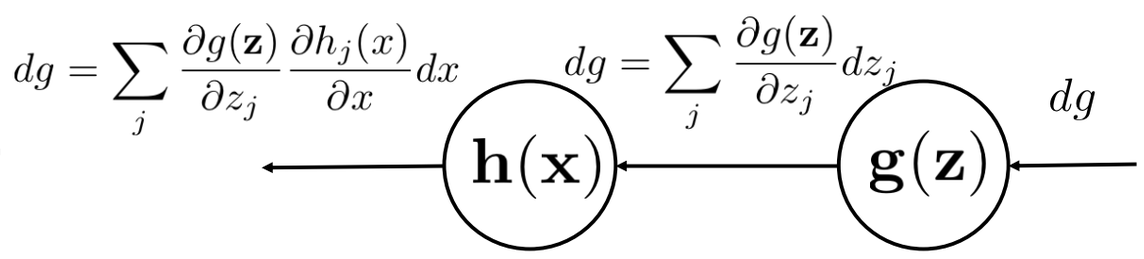
# Multiple Variables

$$x \to \mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} h_1(x) \\ h_2(x) \end{pmatrix} \to g(\mathbf{z}) = g(\mathbf{h}(x)) = g(h_1(x), h_2(x))$$




$$\begin{aligned} dg &= \frac{\partial g(\mathbf{z})}{\partial z_1} dz_1 + \frac{\partial g}{\partial z_2} dz_2 \Big|_{\mathbf{z}=\mathbf{h}(x)} \\ &= \frac{\partial g(\mathbf{z})}{\partial z_1} \frac{\partial h_1(x)}{\partial x} dx + \frac{\partial g(\mathbf{z})}{\partial z_2} \frac{\partial h_2(x)}{\partial x} dx \Big|_{\mathbf{z}=\mathbf{h}(x)} \end{aligned}$$

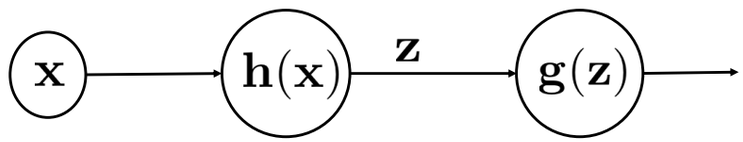
Sum over contributions from different variables.



# Chain Rule for Multi-Variate Functions

Composed multi-variate function

$$\mathbf{h} : \mathbb{R}^n \rightarrow \mathbb{R}^m \ , \ g : \mathbb{R}^m \rightarrow \mathbb{R} \ , \ f(\mathbf{x}) = g \circ h(\mathbf{x})$$



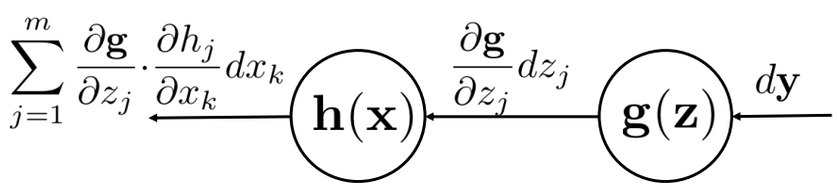

Gradient is computed according to

$$\frac{\partial f(\mathbf{x})}{\partial x_k} = \sum_j \frac{\partial g(\mathbf{z})}{\partial z_j} \cdot \frac{\partial h_j(\mathbf{x})}{\partial x_k} \Bigg|_{\mathbf{z}=\mathbf{h}(\mathbf{x})}$$

Variation of *f* w.r.t. *x*

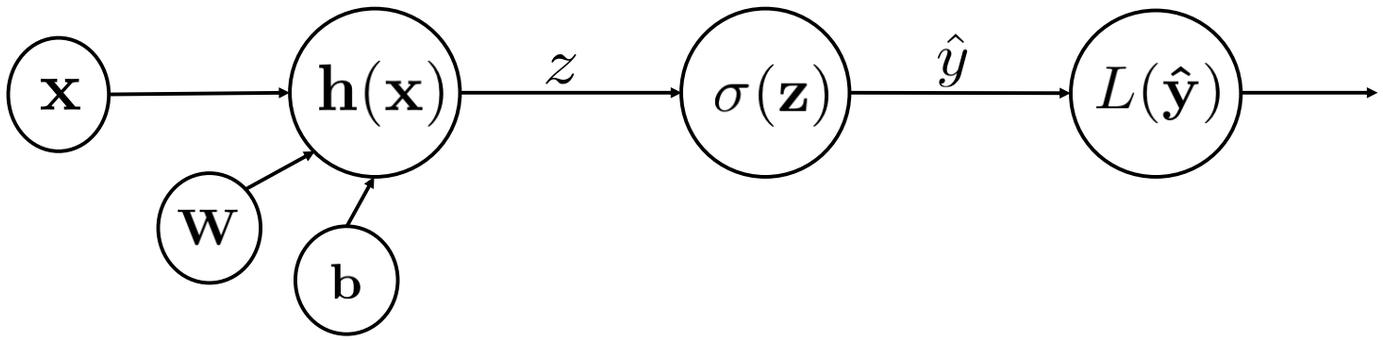
Variation of *g* w.r.t. *z*

Variation of *h* w.r.t. *x*



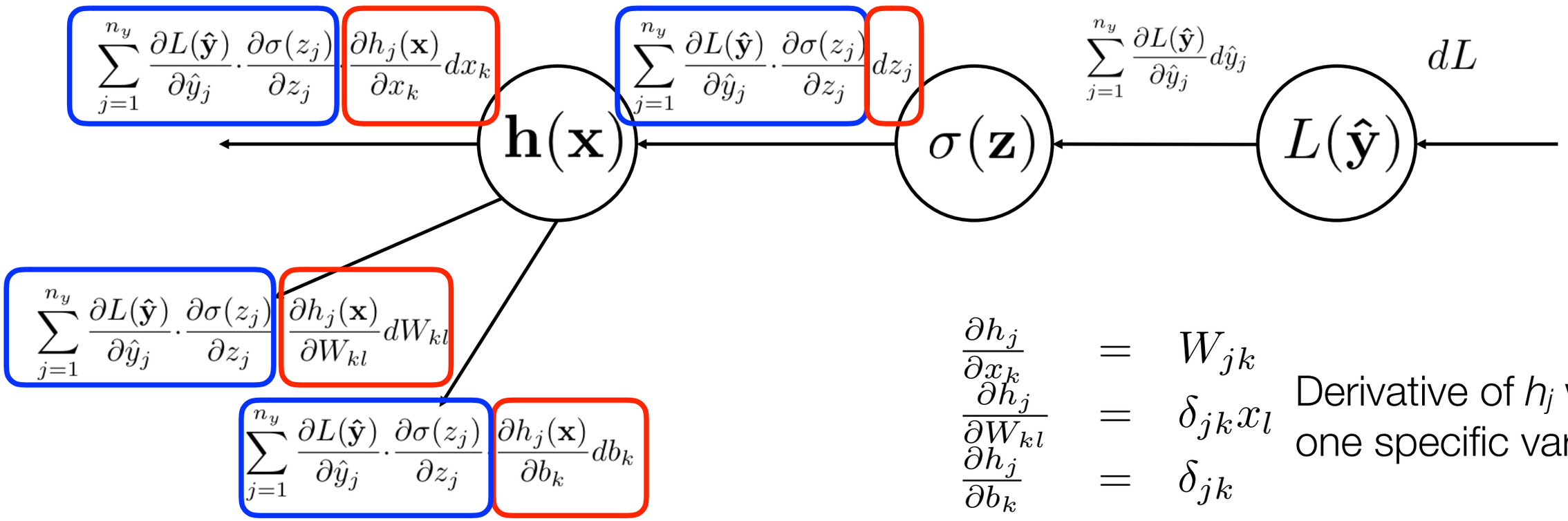
# Chain Rule - Vector-Valued Functions

$$\mathbf{x} \rightarrow \mathbf{z} = \mathbf{h}(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x} + \mathbf{b} \rightarrow \hat{\mathbf{y}} = \sigma(\mathbf{z}) \rightarrow L(\hat{\mathbf{y}})$$




**Shapes**

| | | |
|---|---|---|
| $\mathbf{x}$ | : | $(n_x, 1), (n_x, m)$ |
| $\mathbf{W}$ | : | $(n_y, n_x)$ |
| $\mathbf{b}$ | : | $(n_y, 1)$ |
| $\mathbf{z}$ | : | $(n_y, 1), (n_y, m)$ |
| $\hat{\mathbf{y}}$ | : | $(n_y, 1), (n_y, m)$ |




$$\frac{\partial h_j}{\partial x_k} = W_{jk}$$
$$\frac{\partial h_j}{\partial W_{kl}} = \delta_{jk} x_l$$
$$\frac{\partial h_j}{\partial b_k} = \delta_{jk}$$

Derivative of $h_j$ w.r.t. one specific variable

# Ingredients for Backprop Equations (1)

**Goal:** Compute partial derivatives of the loss function w.r.t. weights and biases

Forward equations (single sample): $\mathbf{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]})\ ,\ \mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$

Observe that $\mathbf{W}^{[l]}, \mathbf{b}^{[l]}$ enter the loss functions only through $\mathbf{a}^{[l]}, \mathbf{z}^{[l]}, \mathbf{a}^{[l+1]}, \mathbf{z}^{[l+1]}, \ldots$

Start with computing the derivatives w.r.t. $\mathbf{a}^{[l-1]}$ by using the chain rule:

$$\begin{aligned}
\frac{\partial L}{\partial a_i^{[l-1]}} &= \sum_{k=1}^{n_l} \frac{\partial L}{\partial a_k^{[l]}} \cdot \frac{\partial a_k^{[l]}}{\partial a_i^{[l-1]}} \\
&= \sum_{k,j=1}^{n_l} \frac{\partial L}{\partial a_k^{[l]}} \cdot \frac{\partial a_k^{[l]}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial a_i^{[l-1]}} \\
&= \sum_{k=1}^{n_l} \frac{\partial L}{\partial a_k^{[l]}} \cdot g^{[l]\prime}(\mathbf{z}^{[l]})_k \cdot W_{ki}^{[l]}
\end{aligned}$$

In vector notation:

$$\frac{\partial L}{\partial \mathbf{a}^{[l-1]}} = {W^{[l]}}^T \cdot \left(\frac{\partial L}{\partial \mathbf{a}^{[l]}} * g^{[l]\prime}(\mathbf{z}^{[l]})\right)$$

(matrix multiplication: $\cdot$, element-wise multiplication: $*$)

**These equations tell us how changes in the cost due to variations in the activation relate across subsequent layers.**

# Ingredients for Backprop Equations (2)

In component notation:

$$\frac{\partial L}{\partial W_{ij}^{[l]}} = \sum_{k,m=1}^{n_l} \frac{\partial L}{\partial a_k^{[l]}} \cdot \frac{\partial a_k^{[l]}}{\partial z_m^{[l]}} \cdot \frac{\partial z_m^{[l]}}{\partial W_{ij}^{[l]}} = \frac{\partial L}{\partial a_i^{[l]}} \cdot g^{[l]\prime}(\mathbf{z}^{[l]})_i \cdot a_j^{[l-1]}$$

$$\frac{\partial L}{\partial b_i^{[l]}} = \sum_{k,m=1}^{n_l} \frac{\partial L}{\partial a_k^{[l]}} \cdot \frac{\partial a_k^{[l]}}{\partial z_m^{[l]}} \cdot \frac{\partial z_m^{[l]}}{\partial b_i^{[l]}} = \frac{\partial L}{\partial a_i^{[l]}} \cdot g^{[l]\prime}(\mathbf{z}^{[l]})_i$$

In vector notation:

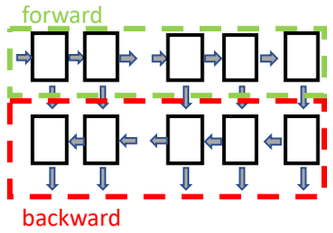
$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \left(\frac{\partial L}{\partial \mathbf{a}^{[l]}} * g^{[l]\prime}(\mathbf{z}^{[l]})\right) \mathbf{a}^{[l-1]^T}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \left(\frac{\partial L}{\partial \mathbf{a}^{[l]}} * g^{[l]\prime}(\mathbf{z}^{[l]})\right)$$

**These equations tell us how changes in the cost due to variations in the weights and biases can be related to changes in the activations of the according layer.**

(single sample)

# Summary Forward / Backprop




**Forward Propagation**

$$
\begin{aligned}
\mathbf{a}^{[0]} &= \mathbf{x} \\
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]} \cdot \mathbf{a}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= g^{[1]}(\mathbf{z}^{[1]}) \\
&\vdots \\
\mathbf{z}^{[L]} &= \mathbf{W}^{[L]} \cdot \mathbf{a}^{[L-1]} + \mathbf{b}^{[L]} \\
\mathbf{a}^{[L]} &= g^{[L]}(\mathbf{z}^{[L]}) \\
L &= L(\hat{\mathbf{y}} = \mathbf{a}^{[L]}, \mathbf{y})
\end{aligned}
$$

**Backward Propagation**

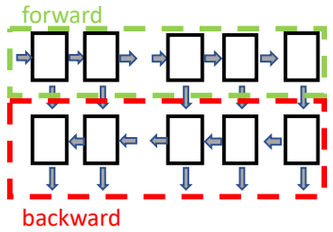
$$
\begin{aligned}
d\mathbf{a}^{[L]} &= \begin{cases} \mathbf{y} - \mathbf{a}^{[L]} & \text{(cross-entropy loss, softmax)} \\ \mathbf{a}^{[L]} - \mathbf{y} & \text{(square distance loss)} \end{cases} \\
d\mathbf{W}^{[L]} &= \left(d\mathbf{a}^{[L]} * g^{[L]\prime}(\mathbf{z}^{[L]})\right) \mathbf{a}^{[L-1]^T} \\
d\mathbf{b}^{[L]} &= \left(d\mathbf{a}^{[L]} * g^{[L]\prime}(\mathbf{z}^{[L]})\right) \\
d\mathbf{a}^{[L-1]} &= W^{[L]T} \cdot \left(d\mathbf{a}^{[L]} * g^{[L]\prime}(\mathbf{z}^{[L]})\right) \\
&\vdots \\
d\mathbf{a}^{[1]} &= W^{[2]T} \cdot \left(d\mathbf{a}^{[2]} * g^{[2]\prime}(\mathbf{z}^{[2]})\right) \\
d\mathbf{W}^{[1]} &= \left(d\mathbf{a}^{[1]} * g^{[1]\prime}(\mathbf{z}^{[1]})\right) \mathbf{a}^{[0]^T} \\
d\mathbf{b}^{[1]} &= \left(d\mathbf{a}^{[1]} * g^{[1]\prime}(\mathbf{z}^{[1]})\right)
\end{aligned}
$$

(per sample)

**Note that $d\mathbf{a}^{[L]}$ differs for different form of the loss function.**

Here, using the same notation as in Andrew Ng's Coursera lecture, i.e. $d\mathbf{a}^{[l]} = \frac{\partial L}{\partial \mathbf{a}^{[l]}}$, $d\mathbf{W}^{[l]} = \frac{\partial L}{\partial \mathbf{W}^{[l]}}$, $d\mathbf{b}^{[l]} = \frac{\partial L}{\partial \mathbf{b}^{[l]}}$

# Vectorised Formulation of Forward and Backprop




## Forward Propagation

$$
\begin{aligned}
\mathbf{A}^{[0]} &= \mathbf{X} = \begin{pmatrix} x_1^{(1)} & \dots & x_1^{(N)} \\ \dots & \dots & \dots \\ x_D^{(1)} & \dots & x_D^{(N)} \end{pmatrix} \\
\\
\mathbf{Z}^{[1]} &= \mathbf{W}^{[1]} \cdot \mathbf{A}^{[0]} + \mathbf{b}^{[1]} \\
\mathbf{A}^{[1]} &= g^{[1]\prime}(\mathbf{Z}^{[1]}) \\
&\vdots \\
\mathbf{Z}^{[L]} &= \mathbf{W}^{[L]} \cdot \mathbf{A}^{[L-1]} + \mathbf{b}^{[L]} \\
\mathbf{A}^{[L]} &= g^{[L]\prime}(\mathbf{Z}^{[L]}) \\
\\
J &= \frac{1}{N} \sum_{i=1}^{N} L(\hat{\mathbf{y}} = \mathbf{a}^{[L](i)}, \mathbf{y}^{(i)})
\end{aligned}
$$

## Backward Propagation

$$
\begin{aligned}
d\mathbf{A}^{[L]} &= \begin{cases} \mathbf{Y} - \mathbf{A}^{[L]} & \text{(cross-entropy loss, softmax)} \\ \mathbf{A}^{[L]} - \mathbf{Y} & \text{(square distance loss)} \end{cases} \\
d\mathbf{W}^{[L]} &= \frac{1}{N} \left( d\mathbf{A}^{[L]} * g^{[L]\prime}(\mathbf{Z}^{[L]}) \right) \cdot \mathbf{A}^{[L-1]^T} \\
d\mathbf{b}^{[L]} &= \frac{1}{N} \text{sum} \left( \left( d\mathbf{A}^{[L]} * g^{[L]\prime}(\mathbf{Z}^{[L]}) \right), \text{axis} = 1 \right) \\
d\mathbf{A}^{[L-1]} &= W^{[L]T} \cdot \left( d\mathbf{A}^{[L]} * g^{[L]\prime}(\mathbf{Z}^{[L]}) \right) \\
&\vdots \\
d\mathbf{A}^{[1]} &= W^{[2]T} \cdot \left( d\mathbf{A}^{[2]} * g^{[2]\prime}(\mathbf{Z}^{[2]}) \right) \\
d\mathbf{W}^{[1]} &= \frac{1}{N} \left( d\mathbf{A}^{[1]} * g^{[1]\prime}(\mathbf{Z}^{[1]}) \right) \cdot \mathbf{A}^{[0]^T} \\
d\mathbf{b}^{[1]} &= \frac{1}{N} \text{sum} \left( d\mathbf{A}^{[1]} * g^{[1]\prime}(\mathbf{Z}^{[1]}), \text{axis} = 1 \right)
\end{aligned}
$$