Introduction to Neural Networks and Deep Learning

Roberto Basili, Danilo Croce

Machine Learning, Web Mining & Retrieval 2021/2022

Outline

- 2020-21 NN Course Structure
- Background: from the Statistical Learning Theory to Deep Learning
- Representation and Architectures for Neural Learning
  - Convolutional NNs
    - Their use in Image Processing and Sentiment Analysis
  - Recurrent NeuralNetworks: *Long Short Term Memories*
  - Autoencoders and Language Modeling: embeddings
  - Transformers: Architectures & Applications
- Laboratory of NN design & Programming

Syllabus

- (11/5) From Statistical Learning to Deep learning
- (12/5) Training Deep Networks
- (16/5) Laboratory: Programming neural learners
- (18/5) Convolutional Networks for Image Processing
- (19/5) Laboratory: Designing Convolutional Networks in Python
- (23/5) Recurrent Neural Networks
- (25/5) Neural Language Modeling
- (26/5) Designing Recurrent Networks for streaming analytics
- (28/5) Encoder-Decoder Networks: Transformers

Introduction to DL: Outline

- An AI perspective on DL: from Statistical Learning Theory to Deep Learning
- Representation Learning in Deep Learning Architectures
  - MLP and *non linearity*
- History and types of NNs:
  - Multilayer Perceptrons
  - Autoencoders
  - Convolutional NNs
  - Recurrent Neural Networks: *Long Short Term Memories*
  - Attentive networks
- Training a Neural Network
  - Stochastic Gradient Descent
  - The Backpropagation algorithm

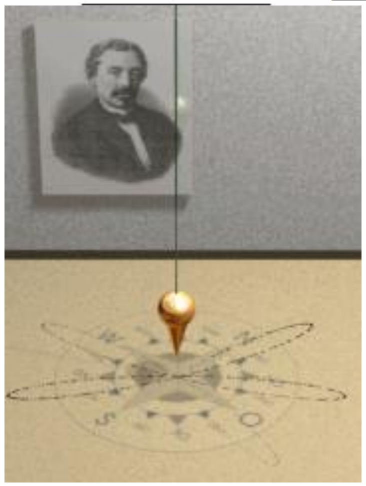
Artificial Intelligence: *the pendulum*

- "A physical symbol system has the necessary and sufficient means for general intelligent action.

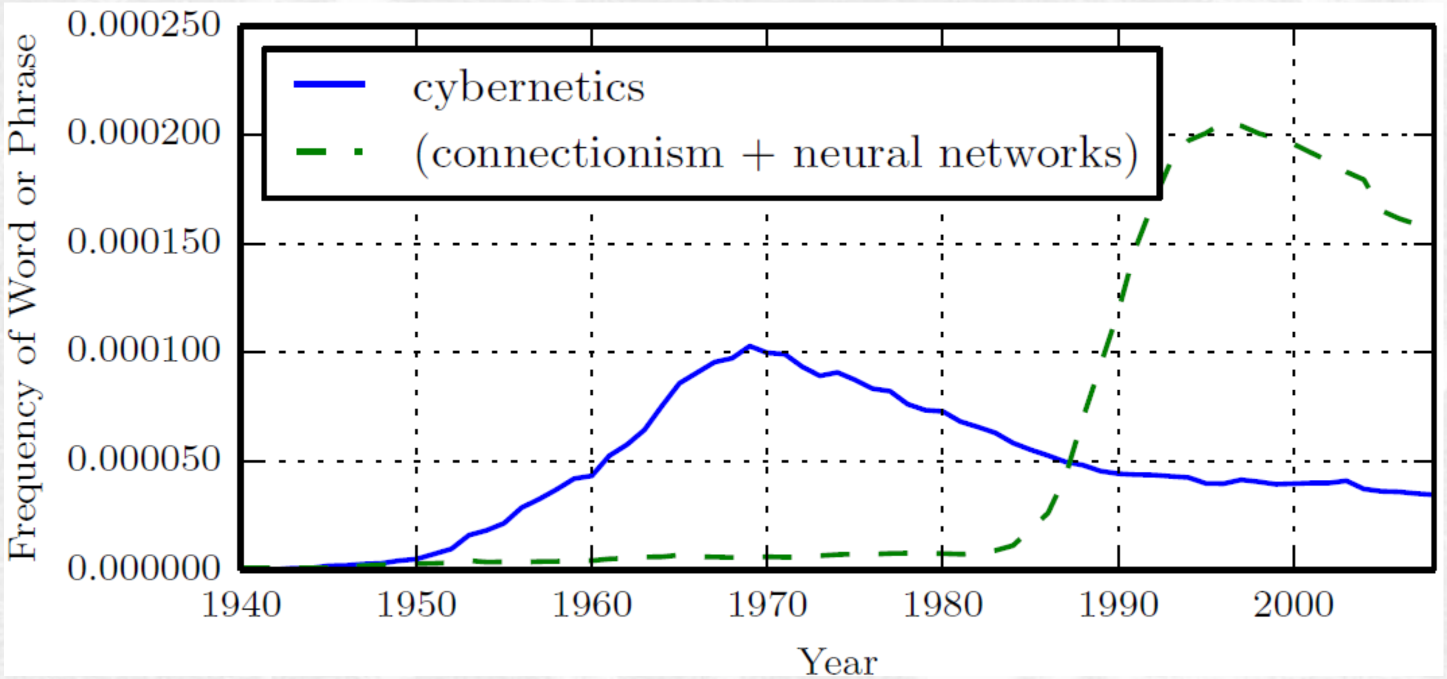
  --Allen Newell & Herbert Simon

- Symbols are Luminiferous Aether of AI

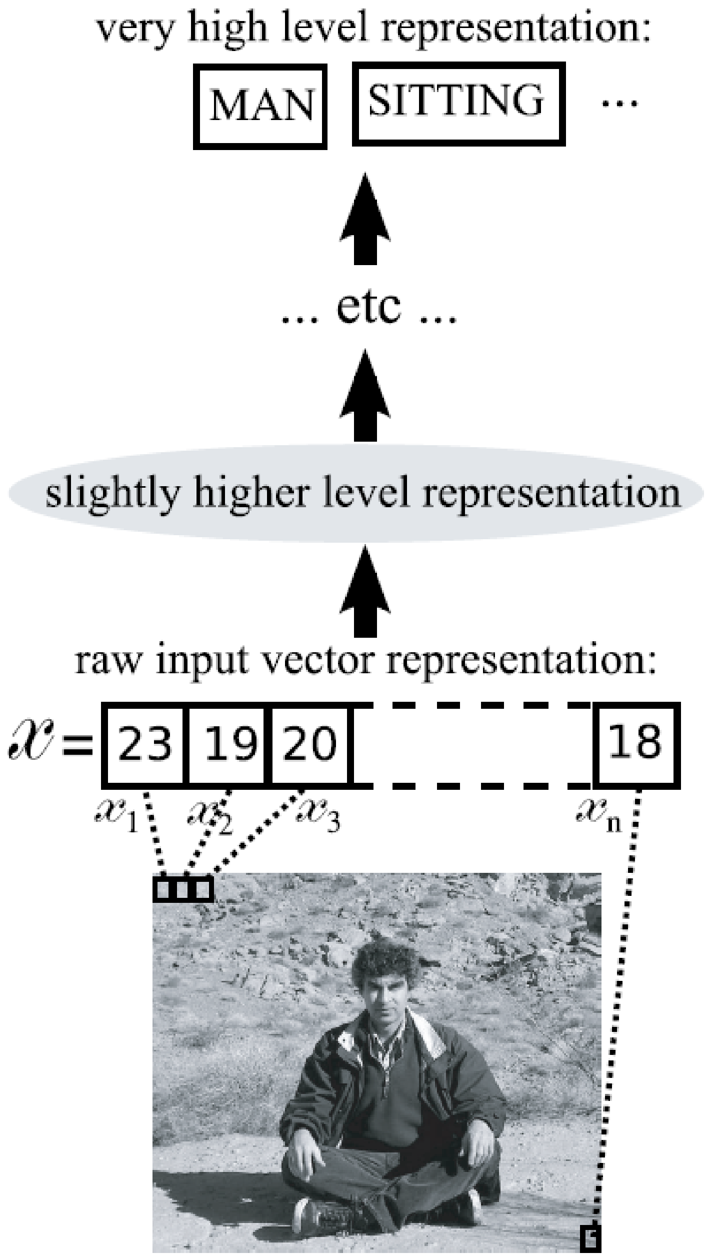
  —Geoff Hinton



Neural Networks, Connectionism and Deep Learning



from Goodfellow et al., DL MIT book



very high level representation:
MAN
SITTING
...
... etc ...
slightly higher level representation
raw input vector representation:
x = 23 19 20 18
x_1
x_2
x_3
x_n

Show & Tell in italiano

Current work at UniTV (Croce, Masotti & Basili,

(a) im2txt+translation: *Un giocatore di baseball che oscilla una mazza ad una sfera*, Italian model: *Un giocatore di baseball che tiene una mazza da baseball su un campo.*

(b) im2txt+translation: *Una grande torre dell'orologio che sovrasta una città*, Italian model: *Un grande edificio con un orologio sulla parte superiore.*

(c) *di*
dei
che

(d) im2txt+translation: *Una persona che salta una tavola skate in aria*, Italian model: *Un uomo che cavalca uno skateboard su una strada.*

A bit of history …

- McCollouch & Pitts 1943 - The logic of the MCP (≈*Perceptron*), through early electronics
- Hebb 1942 - Associative Memories: adaptive storage
- Rosenblatt, 1958 – Perceptron & on-line learning algorithm
- Minsky & Papert, 1969 – mathematical limits of the perceptron
- Rumelhart et al., 1986, McClelland et al., 1995 Backpropagation, Distributed representations
- LSTSMs –Hochreiter & Schmidhuber 1997
- Le Cun et al., 1998 - Convolutional Nets
- Hinton et al., 2006 – Deep Belief nets (autoencoders)
- Bengio et al., 2007 – Depth vs. Breadth in NNs
- Nair & Hinton, 2010 – further training support (e.g. RLU)
- Hinton, 2012 - Dropout

- from (Wang&Raj, 2017):

Wang, Haohan; Raj, Bhiksha,
On the Origin of Deep Learning,

https://arxiv.org/abs/1702.07800 ,
Feb2017

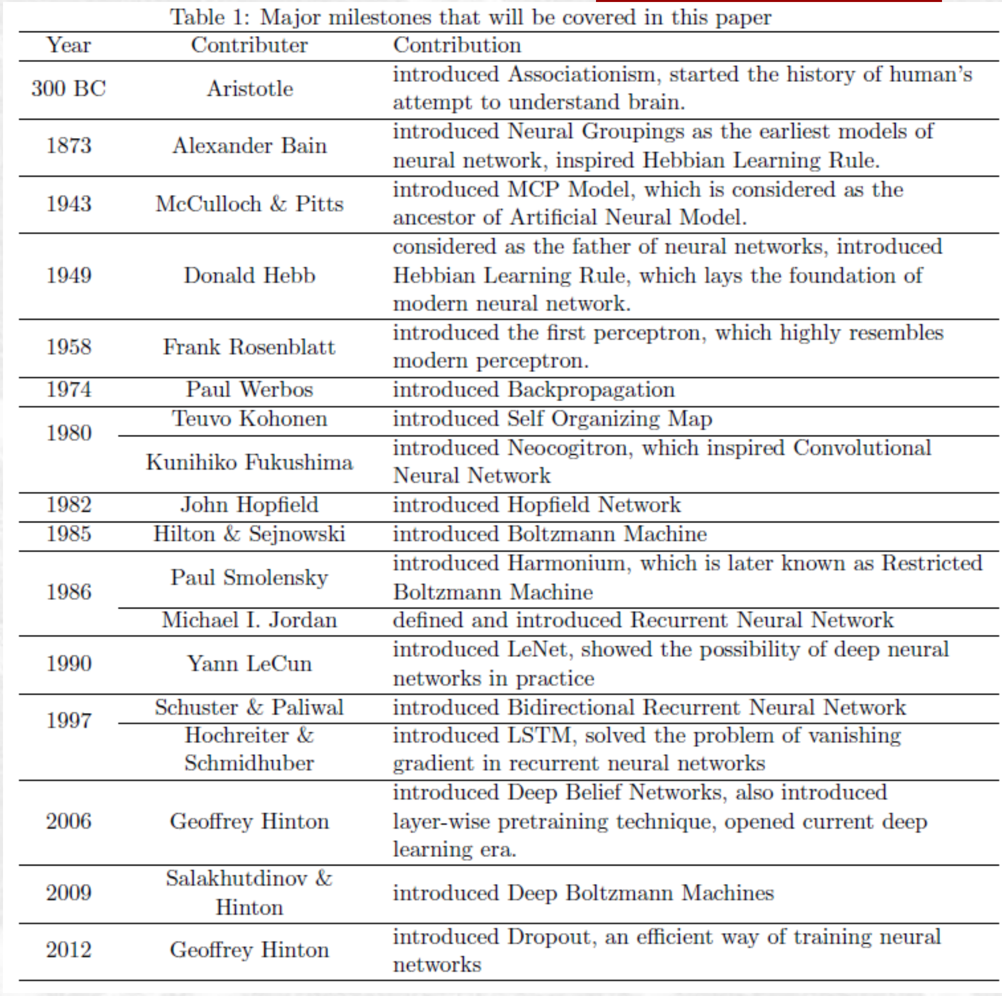
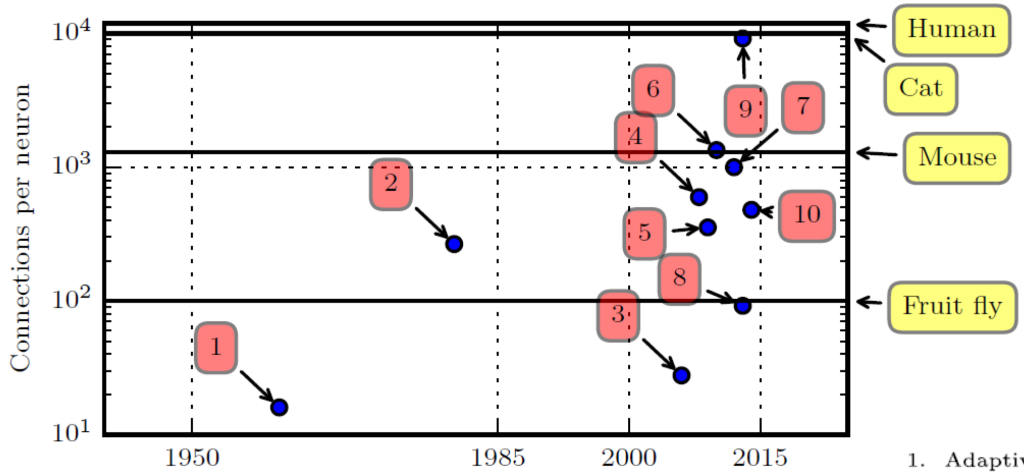


Table 1: Major milestones that will be covered in this paper

Year	Contributer	Contribution
300 BC	Aristotle	introduced Associationism, started the history of human's attempt to understand brain.
1873	Alexander Bain	introduced Neural Groupings as the earliest models of neural network, inspired Hebbian Learning Rule.
1943	McCulloch & Pitts	introduced MCP Model, which is considered as the ancestor of Artificial Neural Model.
1949	Donald Hebb	considered as the father of neural networks, introduced Hebbian Learning Rule, which lays the foundation of modern neural network.
1958	Frank Rosenblatt	introduced the first perceptron, which highly resembles modern perceptron.
1974	Paul Werbos	introduced Backpropagation
1980	Teuvo Kohonen	introduced Self Organizing Map
	Kunihiko Fukushima	introduced Neocogitron, which inspired Convolutional Neural Network
1982	John Hopfield	introduced Hopfield Network
1985	Hilton & Sejnowski	introduced Boltzmann Machine
1986	Paul Smolensky	introduced Harmonium, which is later known as Restricted Boltzmann Machine
	Michael I. Jordan	defined and introduced Recurrent Neural Network
1990	Yann LeCun	introduced LeNet, showed the possibility of deep neural networks in practice
1997	Schuster & Paliwal	introduced Bidirectional Recurrent Neural Network
	Hochreiter & Schmidhuber	introduced LSTM, solved the problem of vanishing gradient in recurrent neural networks
2006	Geoffrey Hinton	introduced Deep Belief Networks, also introduced layer-wise pretraining technique, opened current deep learning era.
2009	Salakhutdinov & Hinton	introduced Deep Boltzmann Machines
2012	Geoffrey Hinton	introduced Dropout, an efficient way of training neural networks

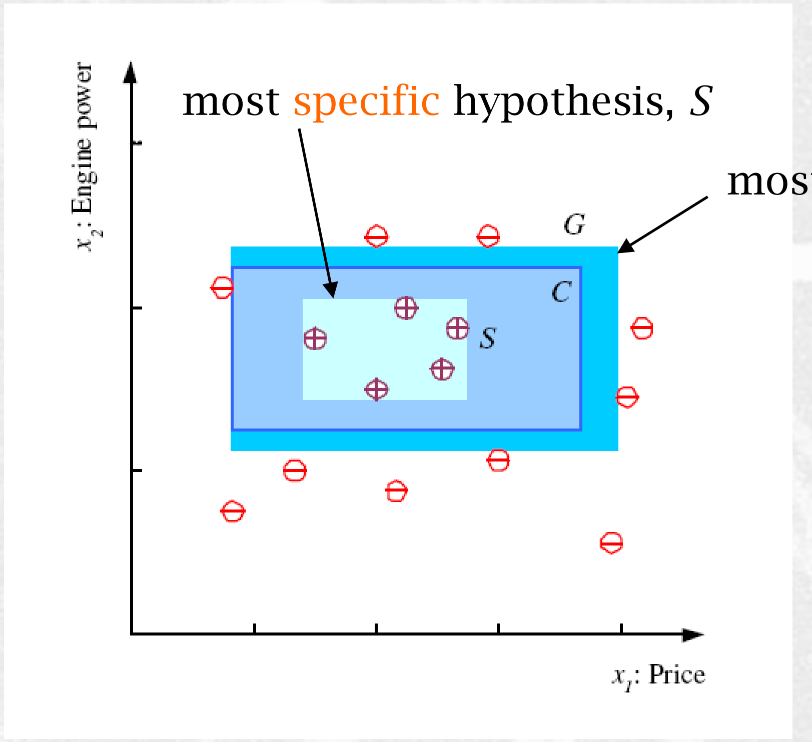
Connections per Neuron



1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett *et al.*, 2009)
6. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
7. Distributed autoencoder (Le *et al.*, 2012)
8. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
9. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

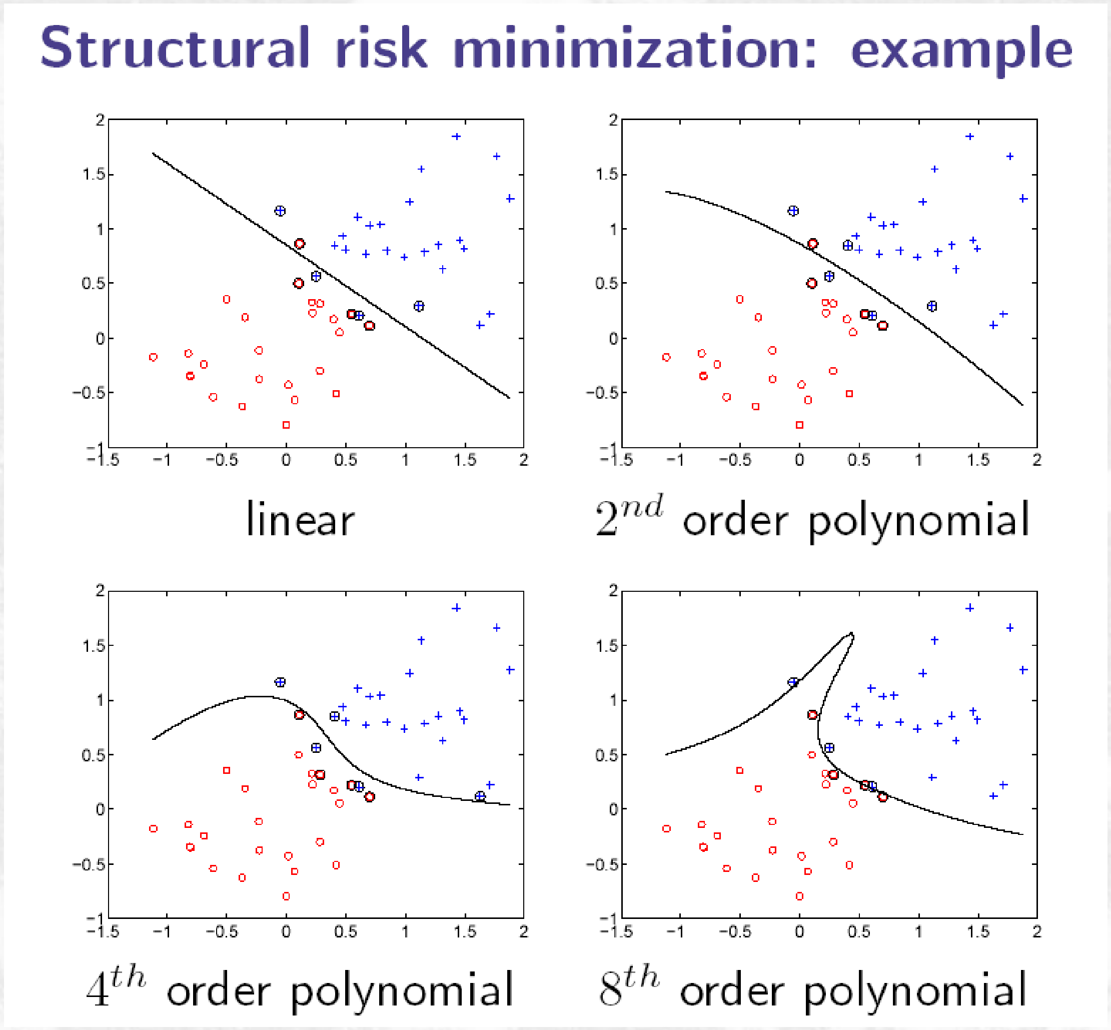
from Goodfellow et al., DL MIT book

(Vector) Spaces, Functions and Learning



The $h \in \mathcal{H}$ floats between S and G to be consistent
It makes up the version space

(Mitchell, 1997)



Structural risk minimization: example

linear

2^{nd} order polynomial

4^{th} order polynomial

8^{th} order polynomial

$$y = f^*(\vec{x})$$

$$f^*(\vec{x}) \approx h(\vec{x}) = g(\vec{x};\vec{\theta})$$

$$\text{such that } \forall \vec{x}_l \in \mathcal{L} \quad h(\vec{x}_l) \approx y_l$$

Machine Learning: in search of good functions

- Model and Learning

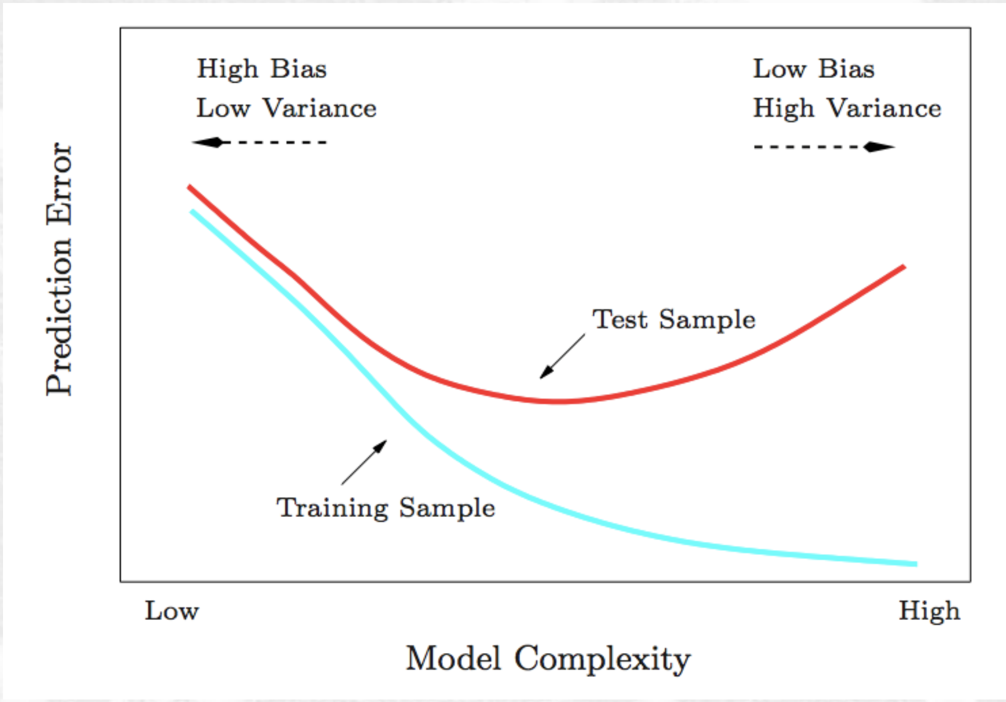
$$y = f^*(\vec{x})$$

$$f^*(\vec{x}) \approx h(\vec{x}) = g(\vec{x};\vec{\theta})$$

$$such\,that\ \forall \vec{x}_l \in \mathrm{L} \quad h(\vec{x}_l) \approx y_l$$

- Linear models

$$h(\vec{x}) = g(\sum_n \theta_n x_n + b)$$

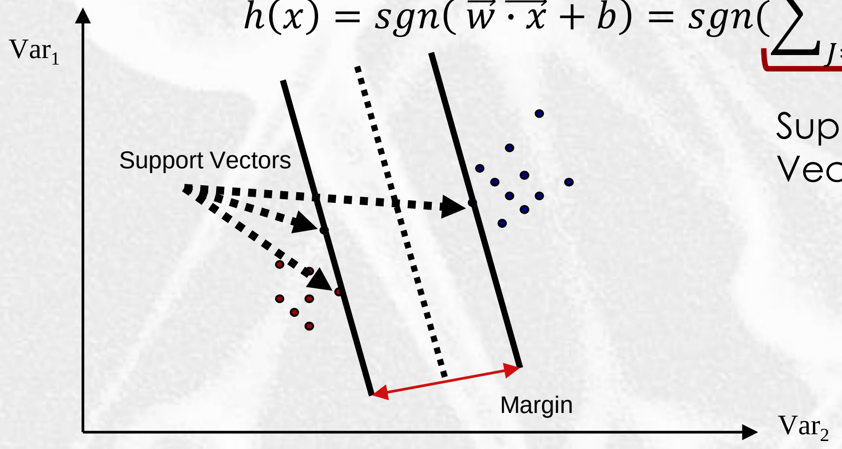


Support Vector Machines

- Support Vector Machines (SVMs) are a machine learning paradigm based on the statistical learning theory [Vapnik, 1995]
  - No need to remember everything, just the discriminating instances (i.e. the support vectors, SV)
  - The classifier corresponds to the linear combination of SVs

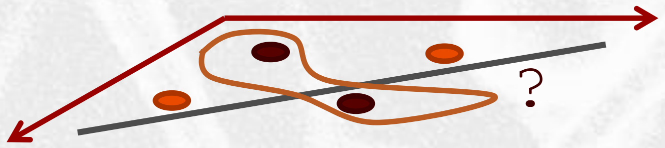
$$h(x) = sgn(\vec{w} \cdot \vec{x} + b) = sgn(\underbrace{\sum_{J=1}^{l} \alpha_j y_j \boxed{\vec{x_j} \cdot \vec{x}}}_{\text{Support Vectors}} + b)$$

Only the dot product is required



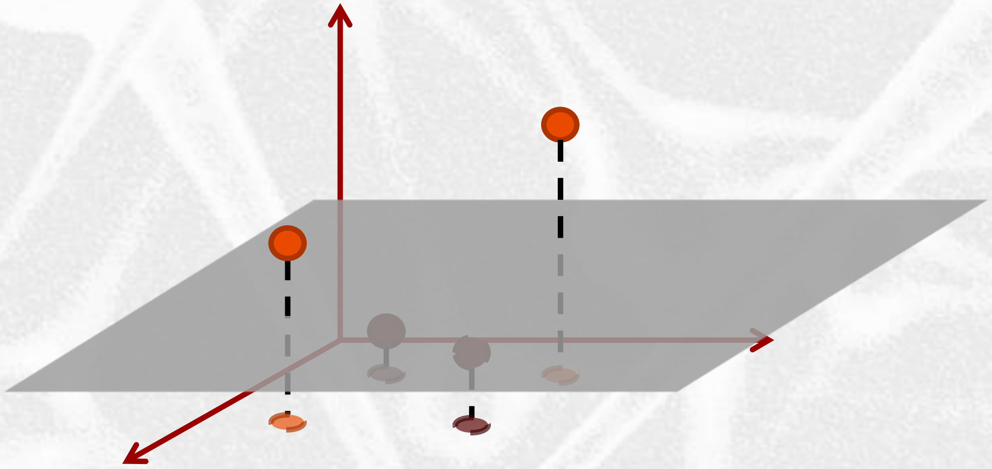
Linear classifiers and separability

- In a R2 space, 3 point can always be separable by a linear classifier
  - but 4 points cannot always be shattered [Vapnik and Chervonenkis(1971)]
- One solution could be a more complex classifier
  - Risk of over-fitting



Linear classifiers and separability (2)

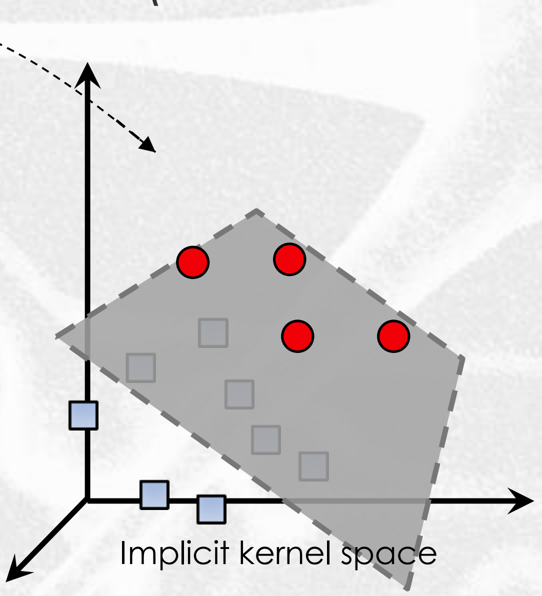
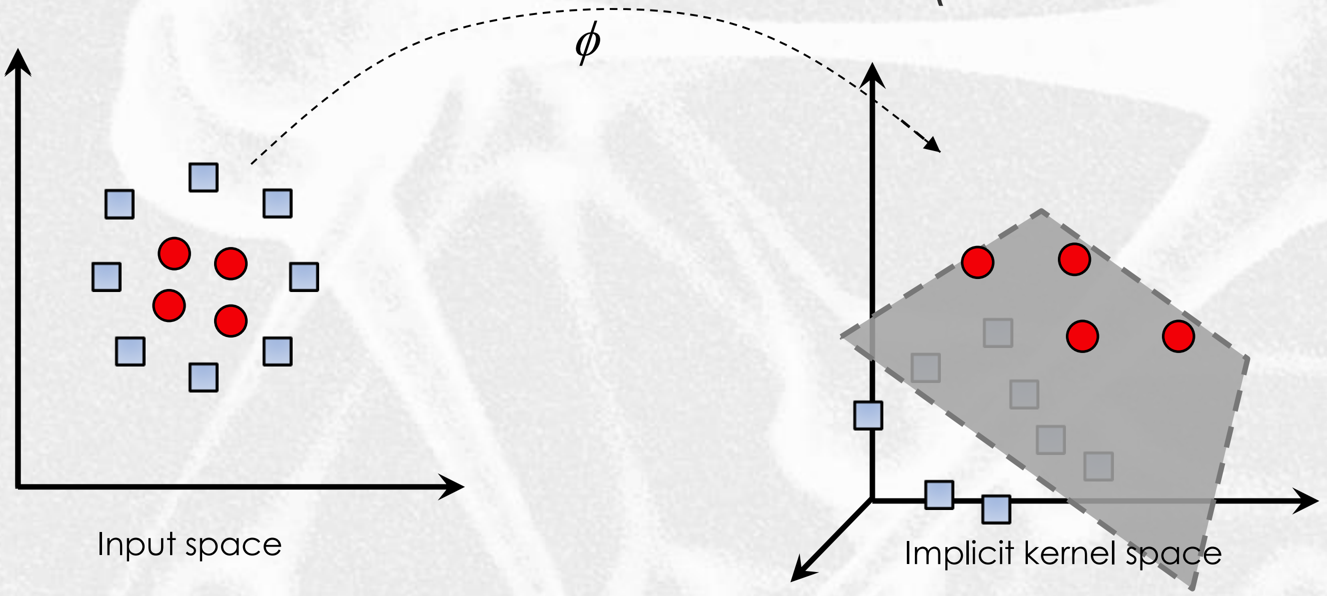
- … but things change when projecting instances in a higher dimension feature space through a function ϕ
- IDEA: It is better to have a more complex feature space instead of a more complex function



SVM First Advantage: making examples linearly separable

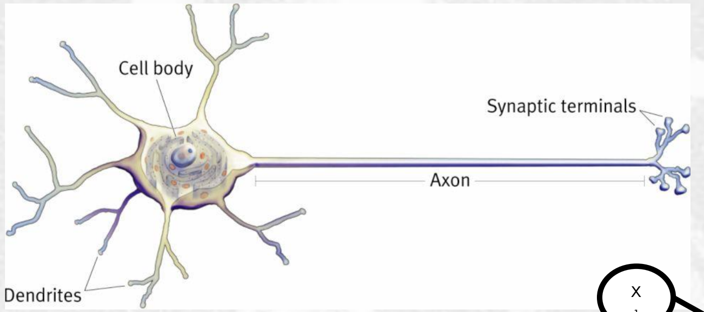
- Mapping data in a (richer) feature space where linear separability holds $\vec{x} \rightarrow \Phi(\vec{x})$

(attributes $\rightarrow$ features)

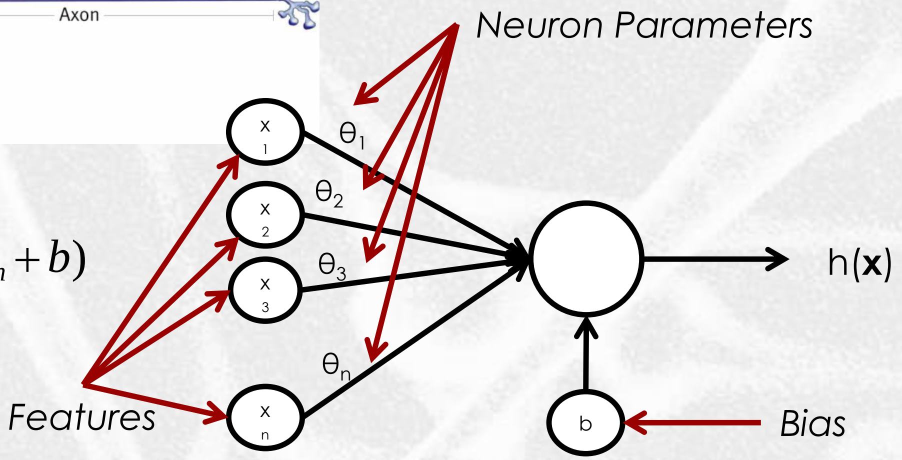



Perceptron (Rosenblatt, 1958)

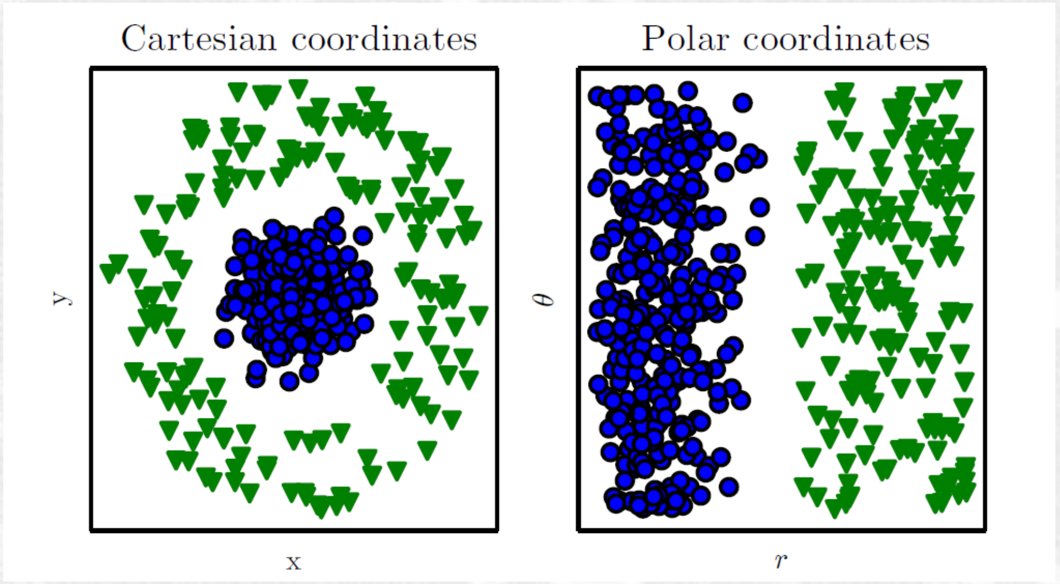
- Linear Classifier mimicking a neuron

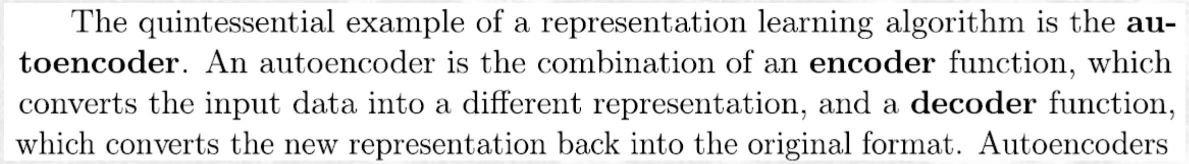


$$h(\vec{x}) = g(\sum_{n} \theta_n x_n + b)$$



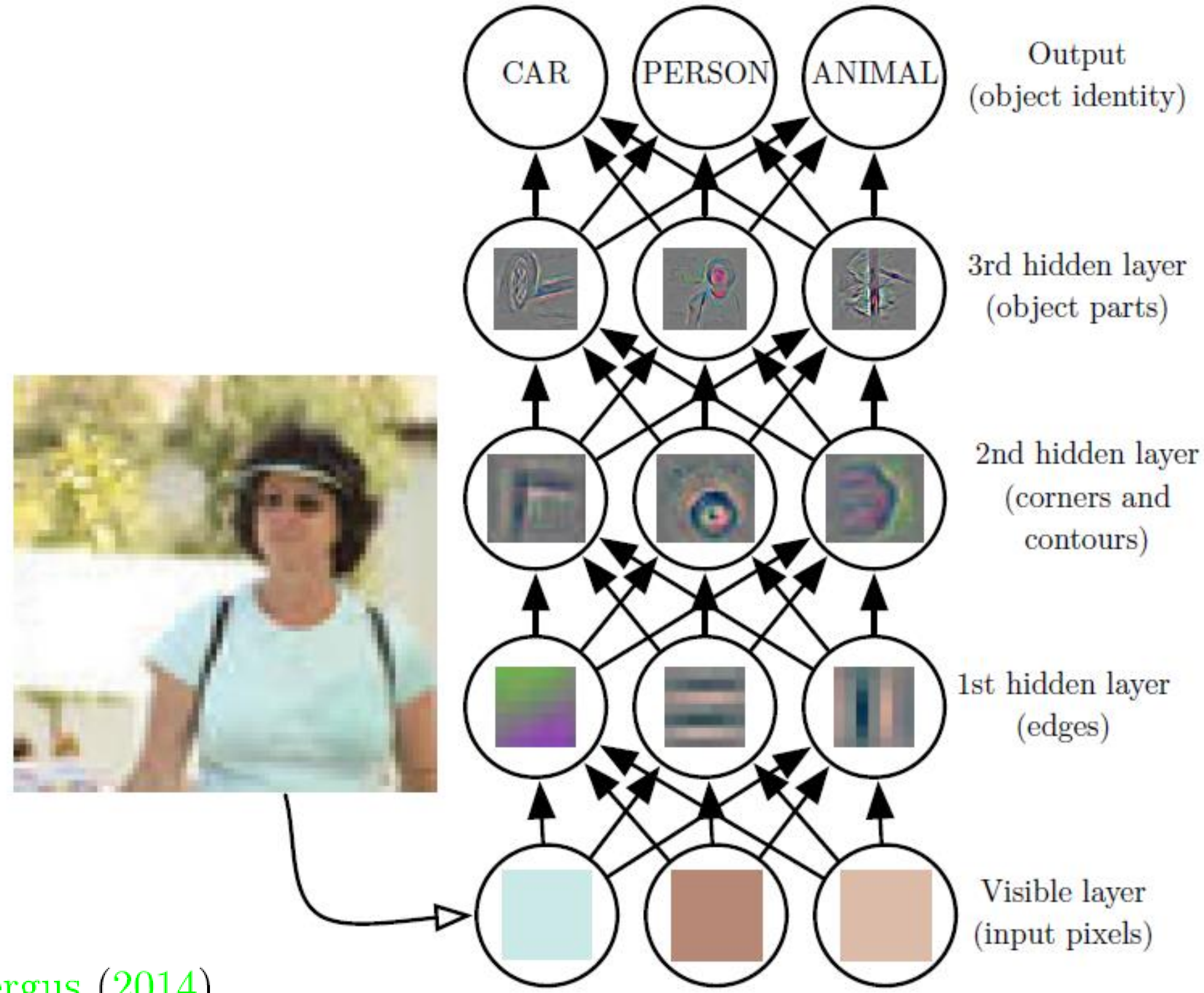
The role of Representation





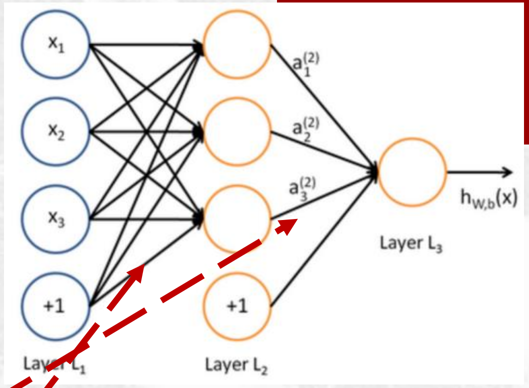
The quintessential example of a representation learning algorithm is the **autoencoder**. An autoencoder is the combination of an **encoder** function, which converts the input data into a different representation, and a **decoder** function, which converts the new representation back into the original format. Autoencoders

Representation and Learning: the role of depth


CAR
PERSON
ANIMAL
Output (object identity)
3rd hidden layer (object parts)
2nd hidden layer (corners and contours)
1st hidden layer (edges)
Visible layer (input pixels)

Zeiler and Fergus (2014)

Adding Layers …



- From simple linear laws …

$$h(\vec{x}) = g(\vec{x};\vec{\theta},b) = g(\sum_n \theta_n x_n + b)$$

- to feedforward structures. It can be made dependent on a sequence of functions g(1) and g(2), …, g(k) that give rise to a structured hypothesis:

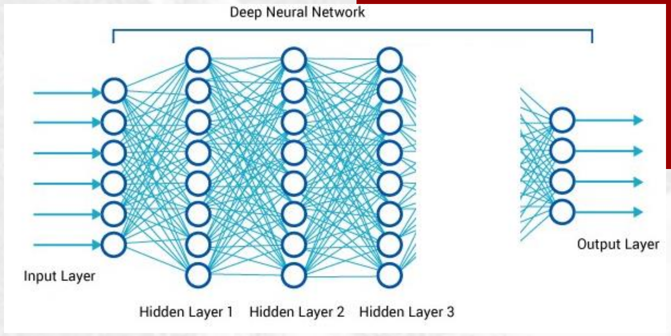
$$h(\vec{x}) = g^{(2)}\left(g^{(1)}\left(\vec{x};\,\vec{\theta}^{(1)}, b^{(1)}\right); \vec{\theta}^{(2)}, b^{(2)}\right) =$$
$$= g^{(2)}\left(W^{(2)} g^{(1)}\left(W^{(1)}\vec{x} + b^{(1)}\right) + b^{(2)}\right)$$

- Hidden layers

$$h^{(1)}(\vec{x}) = g^{(1)}\left(W^{(1)}\vec{x} + b^{(1)}\right)$$

In our example:
$W^{(1)}\ is\ a\ 3 \times 3\ matrix$
$W^{(2)}\ is\ a\ 3 \times 1\ matrix$

Adding Layers …



- From simple linear laws …

$$h(\vec{x}) = g(\vec{x};\vec{\theta},b) = g(\sum_n \theta_n x_n + b)$$

- to feedforward structures. They depend on a sequence of functions g$^{(1)}$, g$^{(2)}$, …, g$^{(k)}$ that give rise to structured hypothesis

$$h(\vec{x}) = g^{(k)}\,(g^{(k-1)}(\ldots \boxed{g^{(1)}(\vec{x};\vec{\theta}^{(1)},b^{(1)});} \ldots);\vec{\theta}^{(k-1)},b^{(k-1)});\vec{\theta}^{(k)},b^{(k)}\,)=$$

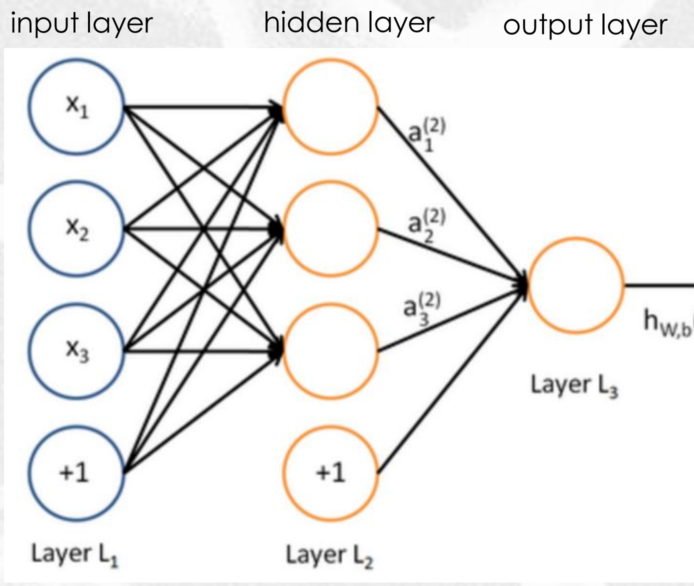
$$= \boxed{g^{(k)}\big(W^{(k)}g^{(k-1)}(W^{(k-1)}\ldots \boxed{g^{(1)}(W^{(1)}\vec{x} + \ b^{(1)})} \ldots + b^{(k-1)}) + b^{(k)}\big)}$$

- Hidden layers

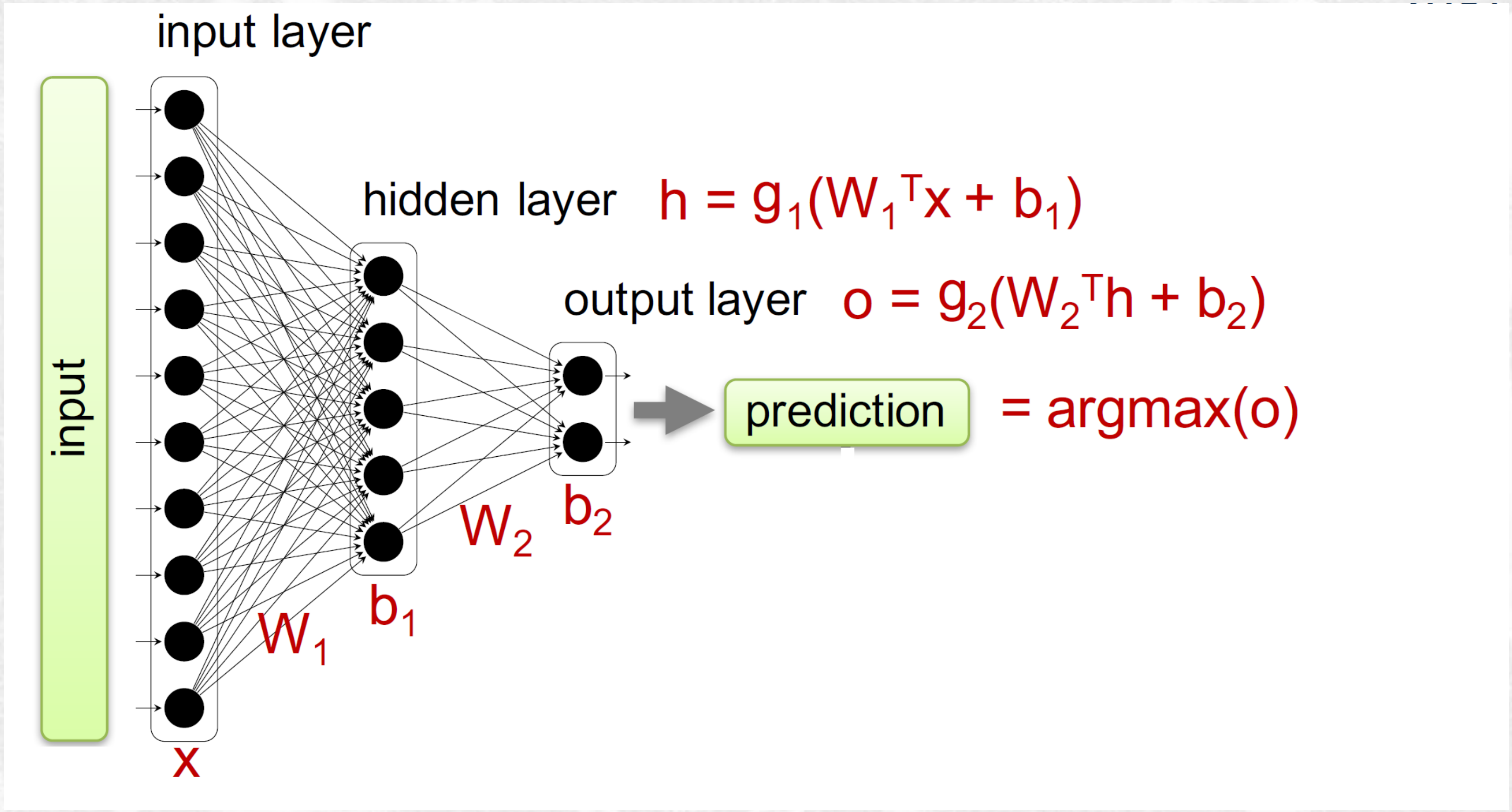
$$h^{(j)}(\vec{x}) = g^{(j)}\left(W^{(j)}g^{(j-1)}(\vec{x};\ \vec{\theta}^{(j-1)},\ \ b^{(j-1)}) + b^{(j)}\right) \qquad j = 1,\ldots,k-1$$

Neural Networks

- Each circle represent a **neuron** (or unit)
  - 3 **input**, 3 **hidden** and 1 **output**
- n_l= 3 is the number of layers
- s_l denotes the number of units in layer l
- Layers:
  - The first layer, i.e. the layer l, is denoted as L_l
  - Layer l and l+1 are connected by a matrix $W^{(l)}$ parameters
    - $W^{(l)}_{i,j}$ connects the *j*-th neuron in layer l with the *i*-th neuron in layer $l+1$
- $b^{(l)}_i$ is the *bias* associated to neuron I in layer l+1



Forward Step: classification

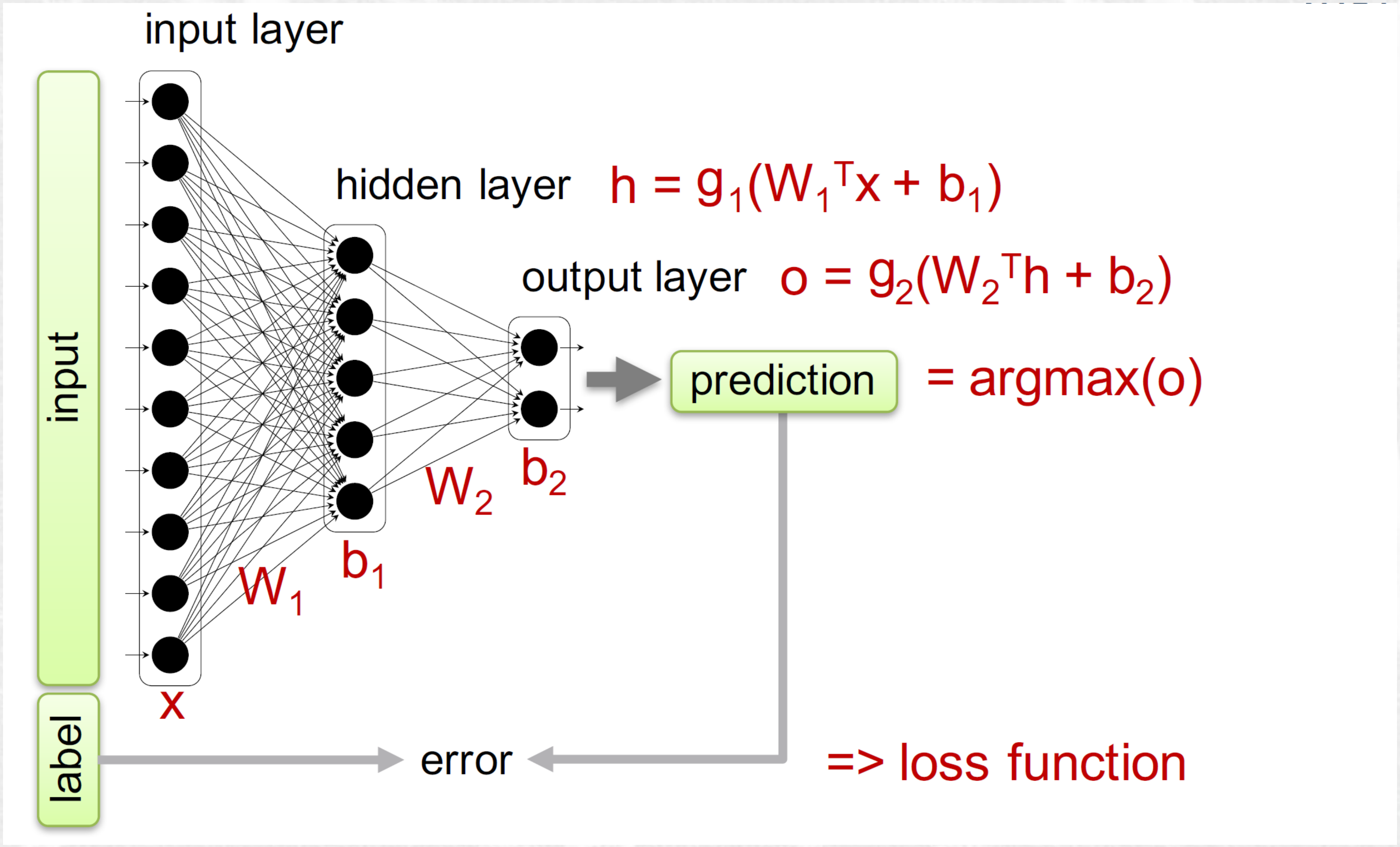


$h = g_1(W_1^T x + b_1)$

$o = g_2(W_2^T h + b_2)$

$= \text{argmax}(o)$

Forward Step: training



$h = g_1(W_1^T x + b_1)$

$o = g_2(W_2^T h + b_2)$

prediction $= \text{argmax}(o)$

=> loss function

Demystifying neural networks

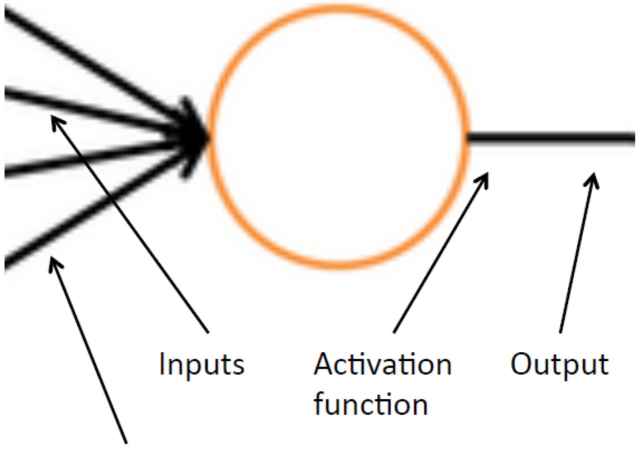
Neural networks come with their own terminological baggage

… just like SVMs

But if you understand how logistic regression or maxent models work

Then **you already understand** the operation of a basic neural network neuron!

A single neuron
A computational unit with *n* (3) inputs and 1 output and parameters *W*, *b*

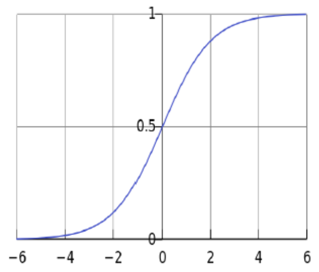


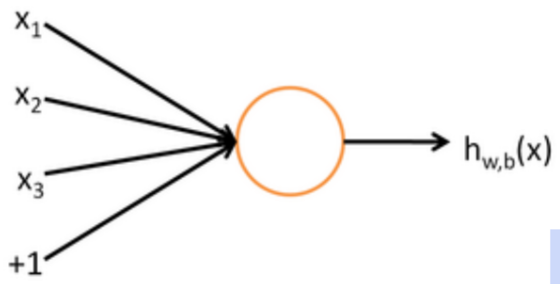
Bias unit corresponds to intercept term

$$h_{w,b}(x) = f(w^{\mathsf{T}}x + b)$$

b: We can have an "always on" feature, which gives a class prior, or separate it out, as a bias term

$$f(z) = \frac{1}{1+e^{-z}}$$

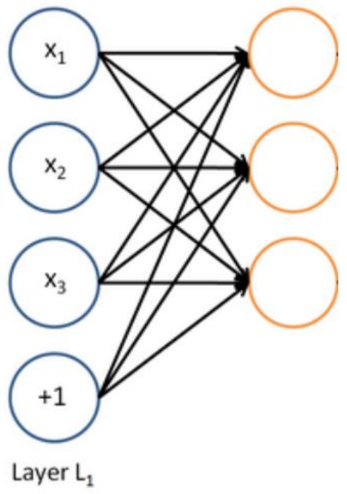




w, *b* are the parameters of this neuron i.e., this logistic regression model

A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

What is Deep Learning

- It is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layer

- *Learning representations* of data
  - feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features

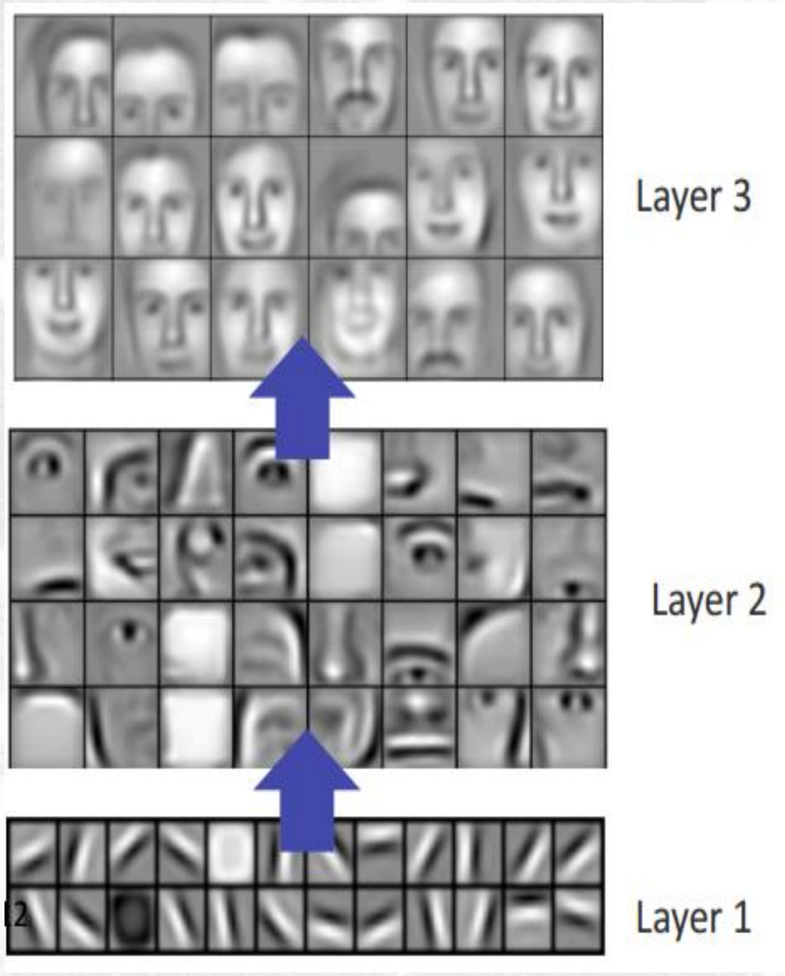
From Machine Learning…

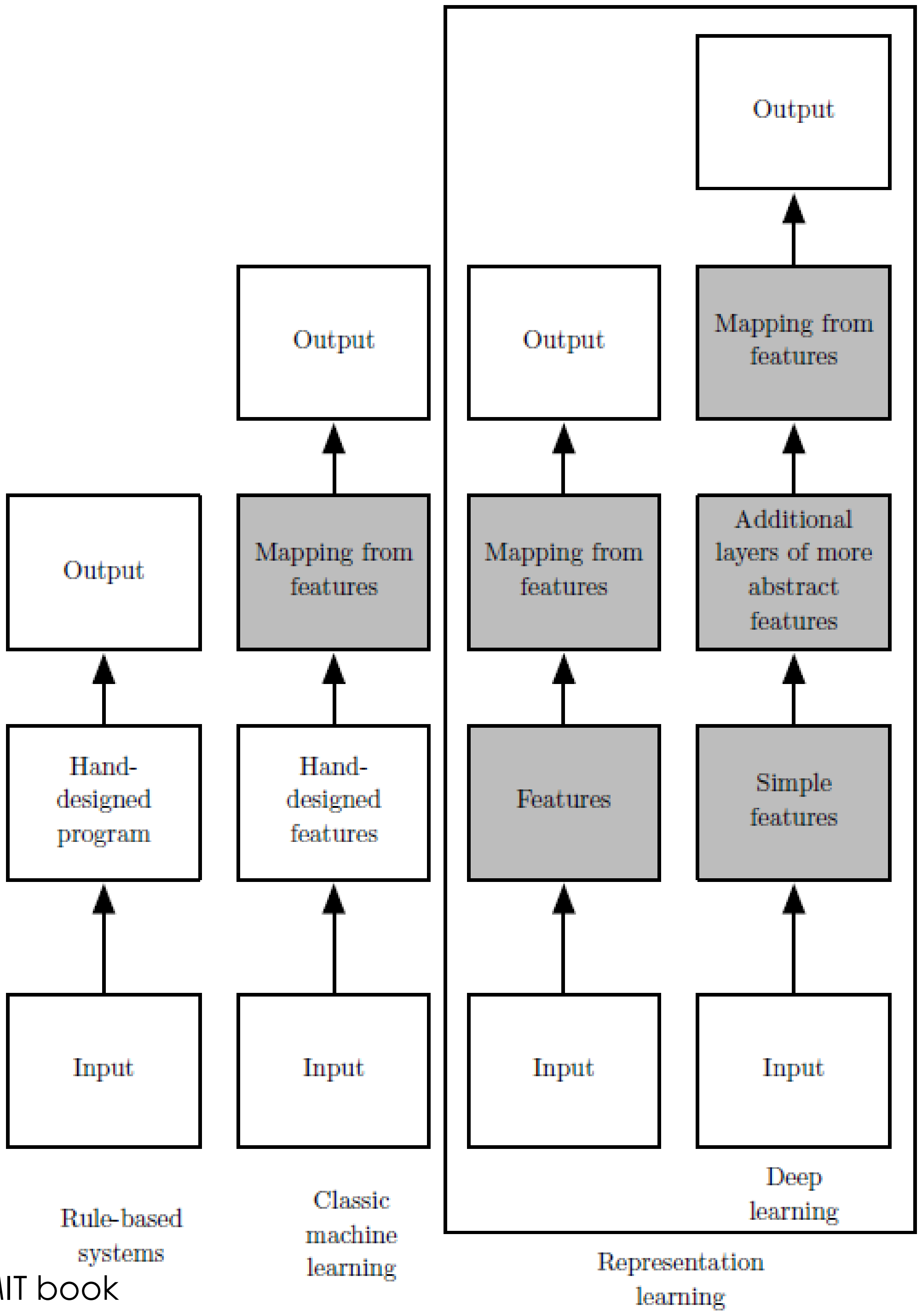
- Machine Learning in general works well because of human-designed features
  - E.g. the so-called "Bag-of-Word" vector

- In this sense, machine learning is optimizing a set of parameters to obtain best performances
  - a costly operation
  - to be repeated for each new task

… to Deep Learning

- Representation Learning attempts at automatically learning the features (as well as the parameters)
- Deep Learning attempts at learning multiple levels (a hierarchy) of features of increasing complexity
- For example, in Face Detection
  - A face can be composed by eyes, nose, mouth
  - Each of them is composed from simpler shapes
- How to automatically learn these "features"?





from Goodfellow et al., DL MIT book

AI desiderata

- **Ability to learn complex, highly-varying functions**, i.e., with a number of variations much greater than the number of training examples.
- **Ability to learn with little human input** the low-level, intermediate, and high-level abstractions that would be useful to represent the kind of complex functions needed for AI tasks.
- **Ability to learn from a very large set of examples**: computation time for training should scale well with the number of examples, i.e., close to linearly.
- **Ability to learn from mostly unlabeled data**, i.e., to work in the semi-supervised setting, where not all the examples come with complete and correct semantic labels.
- **Ability to exploit the synergies present across a large number of tasks**, i.e., multi-task learning. These synergies exist because all the AI tasks provide different views on the same underlying reality.
- **Strong unsupervised learning** (i.e., capturing most of the statistical structure in the observed data), which seems essential in the limit of a large number of tasks and when future tasks are not known ahead of time.

Basic Notation & Formalisms

- Basic *jargon*:
  - Vectors spaces, inner products and Topology: Vectors, Matrices and Tensors
  - Training vs. Classification
  - Forward step, backpropagation,
  - Cost Function, Loss & Regularization
  - Input representation
    - Dense vs. Discrete
    - Embeddings
  - Output format
  - Tasks: classification aka labeling, autoencoding, encoding-decoding, stacking, multiple task learning

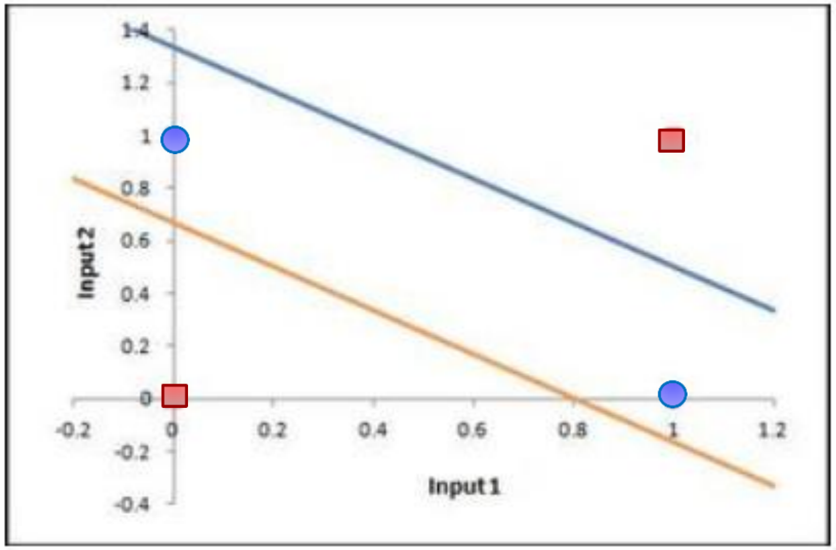
Non linearity: the MLP

- In order to capture complex non linear functions with can apply a still linear model not to $\underline{x}$ itself but rather to one of its transformed form, e.g. $\Phi(\underline{x})$
- Which mapping Φ :
  - Exploit generic mathematical, domain-independent mappings (e.g. polynomial kernels or RBFs)
  - Manually engineering Φ
  - Learn the proper Φ with respect to the task
- The result is a new form of the learning problem

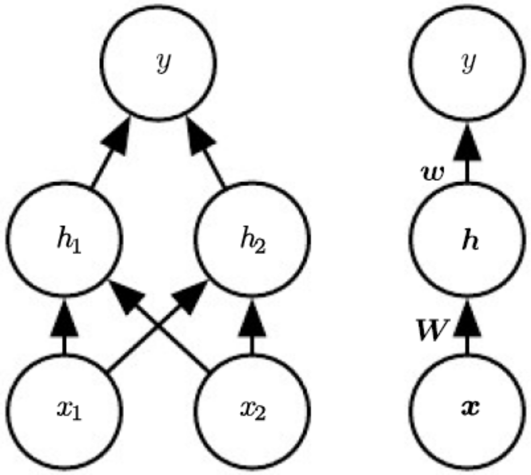
$$y = f(\underline{x};\theta,W) = W \cdot \Phi(\underline{x}) + b$$

A simple MLP: the XOR function

Input1	Input2	Output	
0	0	0	🟥
1	0	1	🔵
0	1	1	🔵
1	1	0	🟥



A MLP for the XOR problem

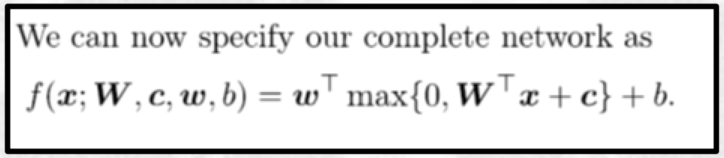


We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

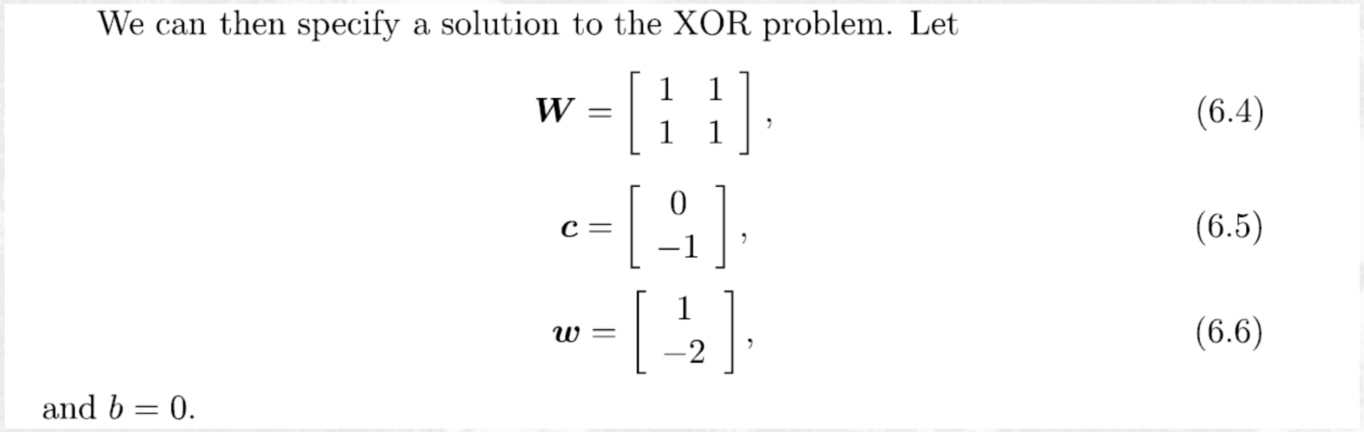
Figure 6.2: An example of a feedforward network, drawn in two different styles. Specifically, this is the feedforward network we use to solve the XOR example. It has a single hidden layer containing two units. *(Left)* In this style, we draw every unit as a node in the graph. This style is explicit and unambiguous, but for networks larger than this example, it can consume too much space. *(Right)* In this style, we draw a node in the graph for each entire vector representing a layer's activations. This style is much more compact. Sometimes we annotate the edges in this graph with the name of the parameters that describe the relationship between two layers. Here, we indicate that a matrix $\boldsymbol{W}$ describes the mapping from $\boldsymbol{x}$ to $\boldsymbol{h}$, and a vector $\boldsymbol{w}$ describes the mapping from $\boldsymbol{h}$ to y. We typically omit the intercept parameters associated with each layer when labeling this kind of drawing.

The solution



We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$



We can then specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{6.4}$$

Rotazione

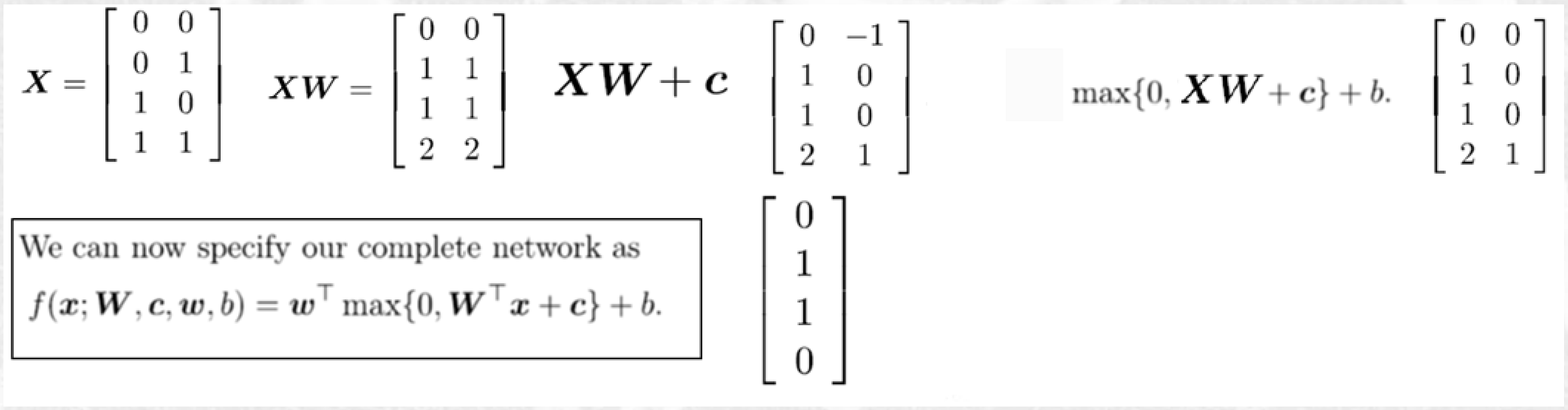
$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{6.5}$$

Traslazione

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \tag{6.6}$$

Scaling

and $b = 0$.



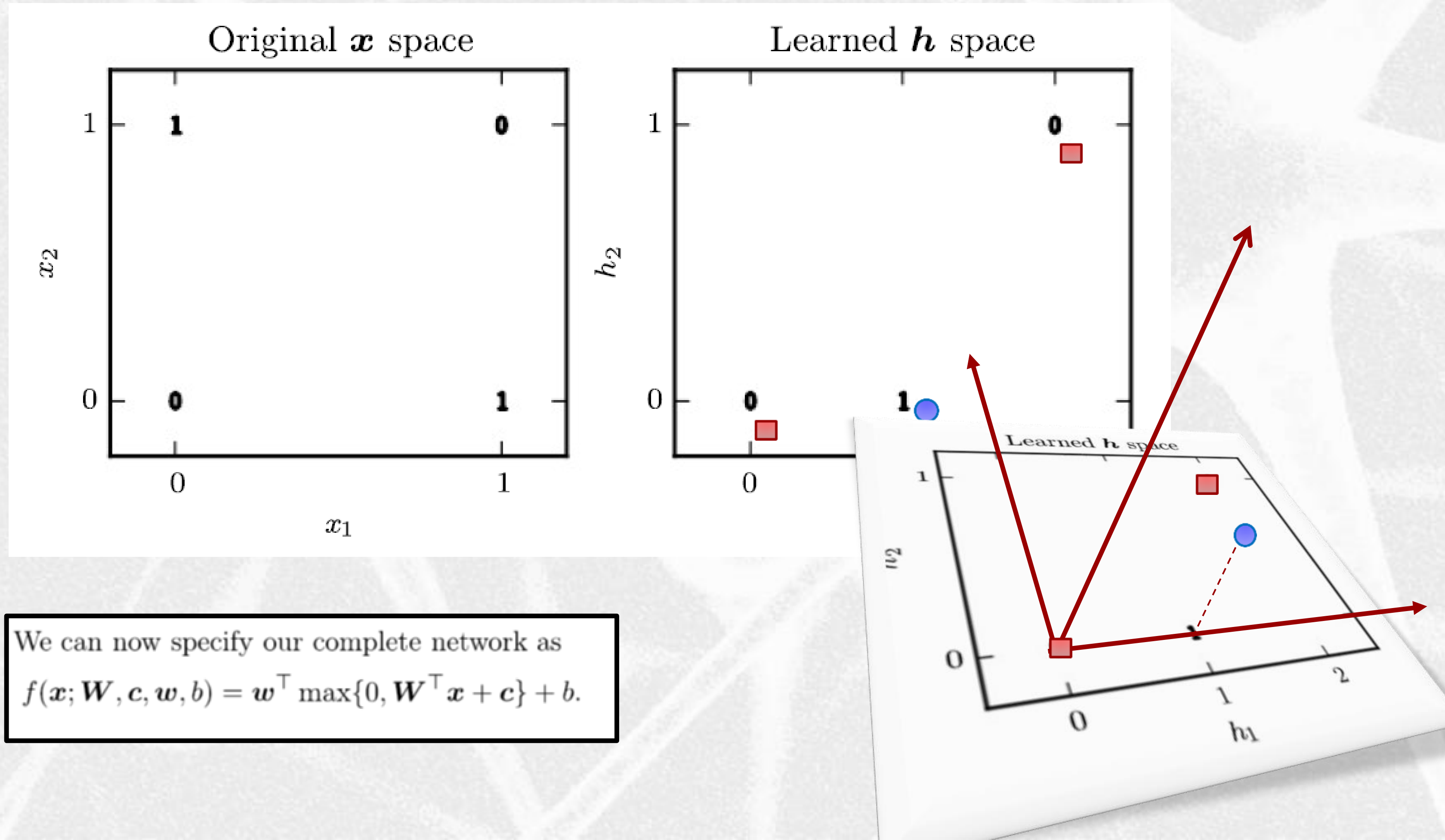
$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \quad \boldsymbol{XW} + \boldsymbol{c} \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \quad \max\{0, \boldsymbol{XW} + \boldsymbol{c}\} + b. \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

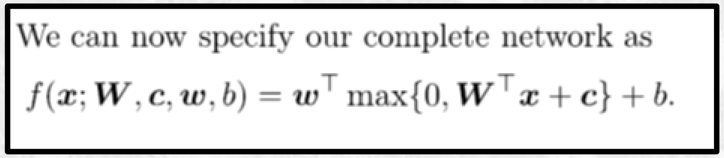
We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The new representation space



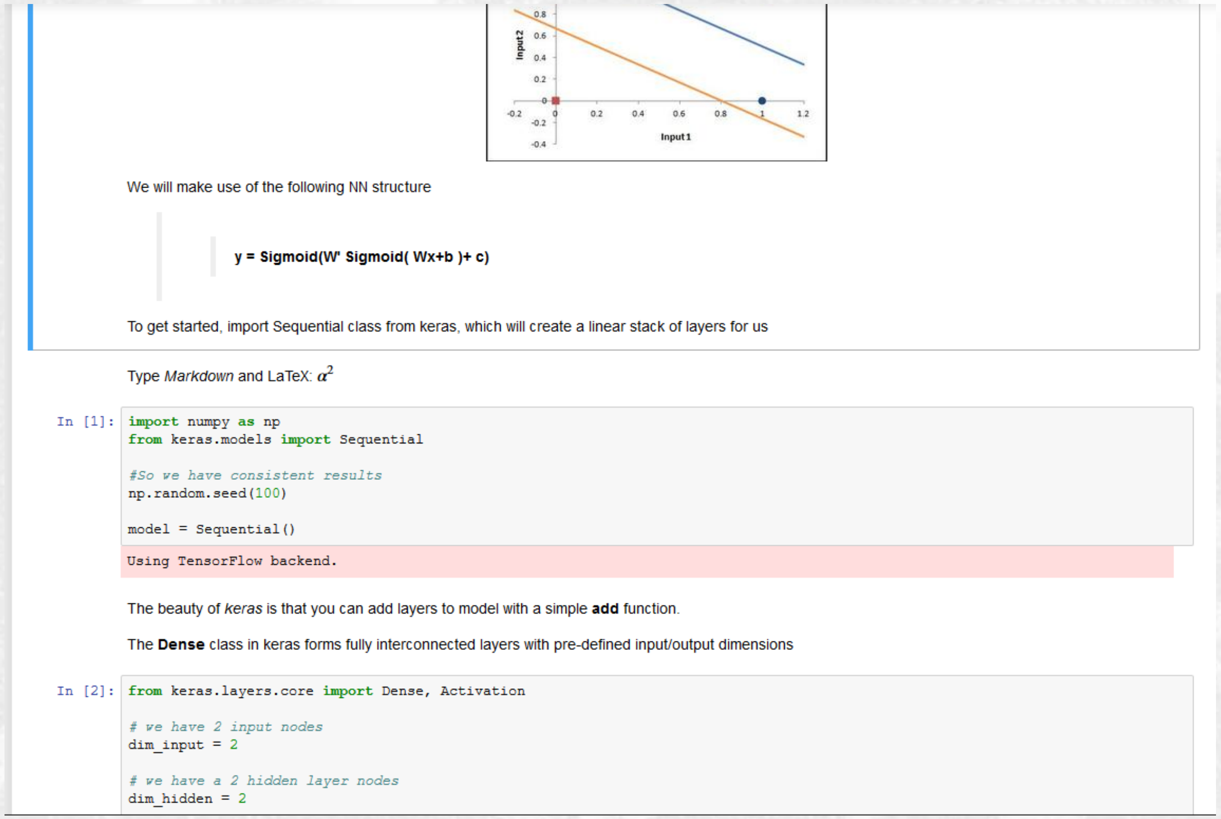


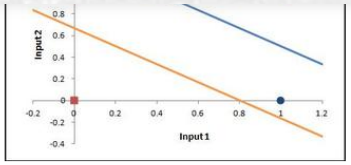
We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

An example in Keras

- See the XOR Keras example in the Jupiter Notebook made available on MS Teams





We will make use of the following NN structure

> > **y = Sigmoid(W' Sigmoid(Wx+b)+ c)**

To get started, import Sequential class from keras, which will create a linear stack of layers for us

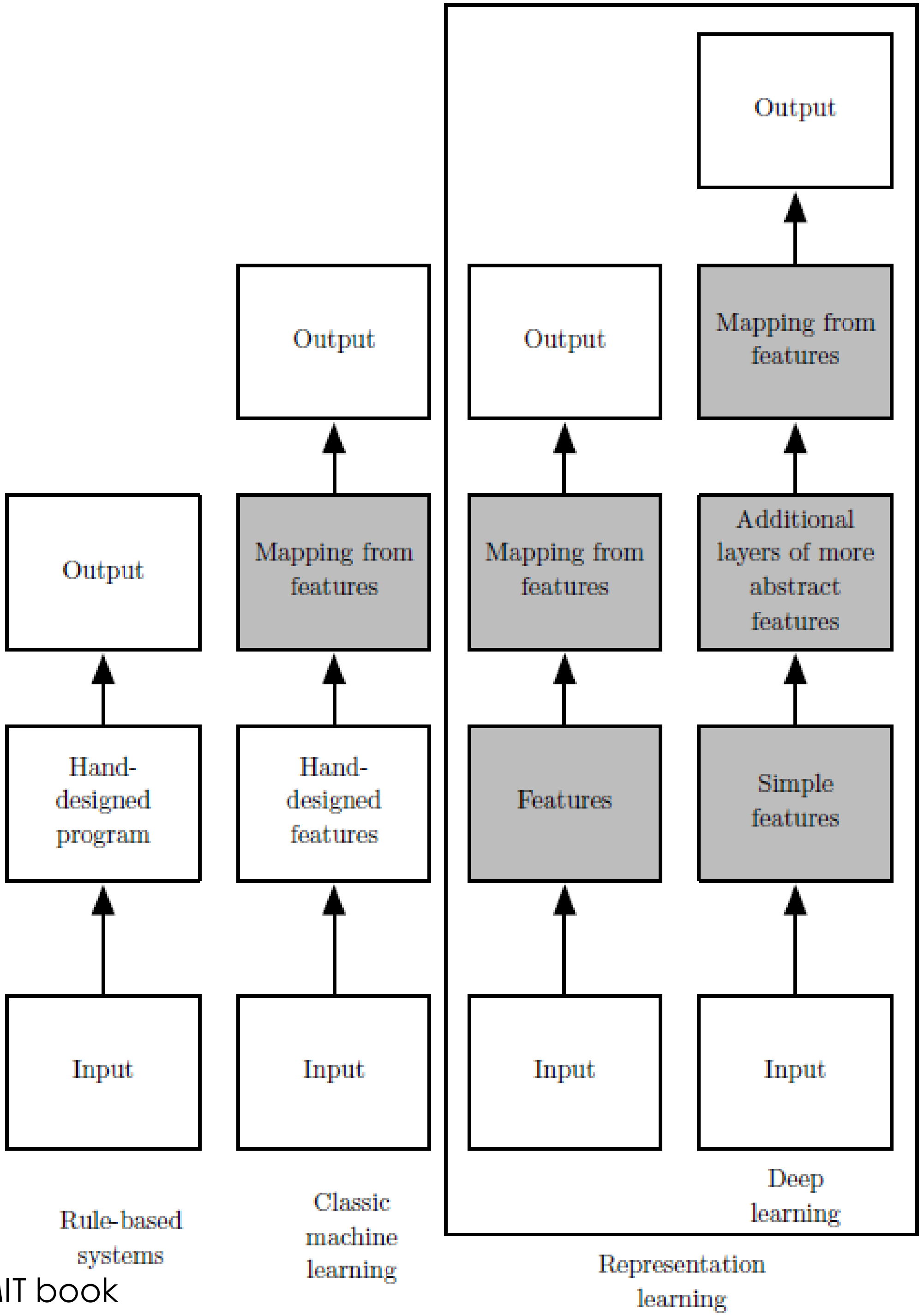
Type *Markdown* and LaTeX: α^2

```
In [1]: import numpy as np
        from keras.models import Sequential

        #So we have consistent results
        np.random.seed(100)

        model = Sequential()

        Using TensorFlow backend.
```

The beauty of *keras* is that you can add layers to model with a simple **add** function.

The **Dense** class in keras forms fully interconnected layers with pre-defined input/output dimensions

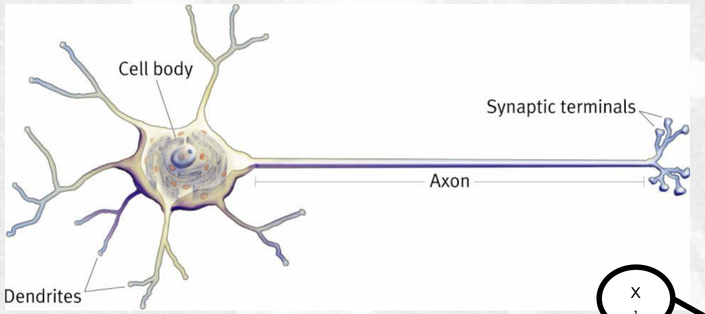
```
In [2]: from keras.layers.core import Dense, Activation

        # we have 2 input nodes
        dim_input = 2

        # we have a 2 hidden layer nodes
        dim_hidden = 2
```



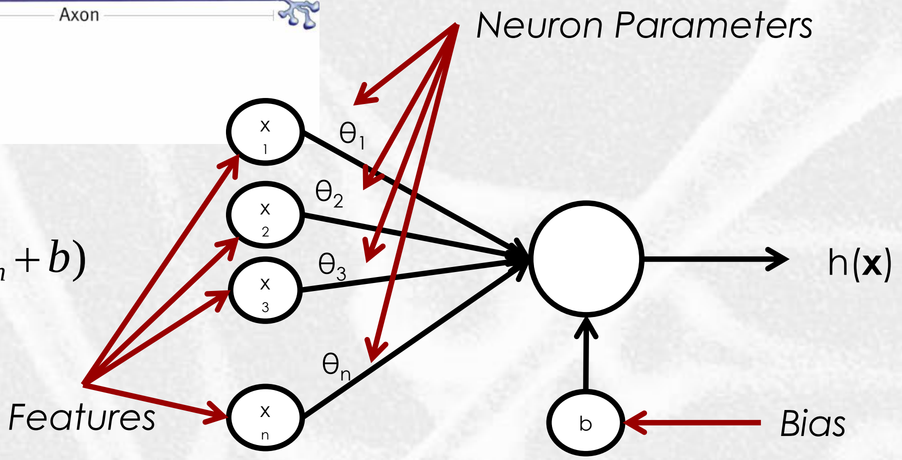
from Goodfellow et al., DL MIT book

Perceptron (Rosenblatt, 1958)

- Linear Classifier mimicking a neuron



$$h(\vec{x}) = g(\sum_n \theta_n x_n + b)$$

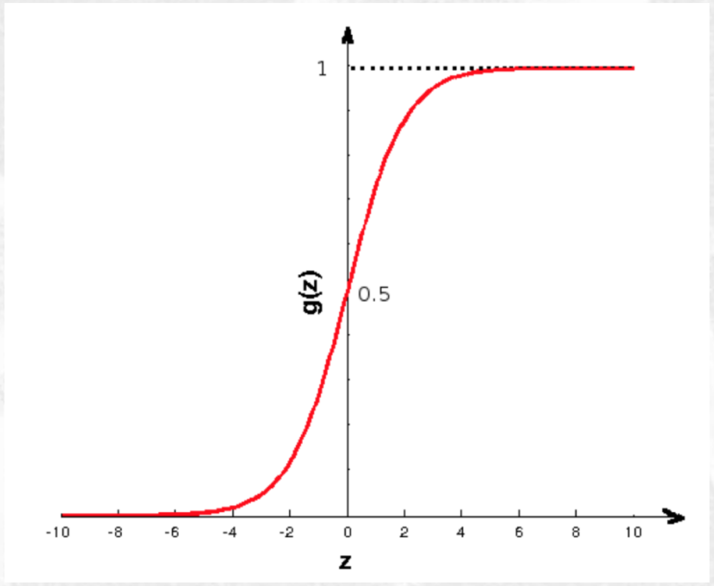


Perceptron and non-linear activation functions

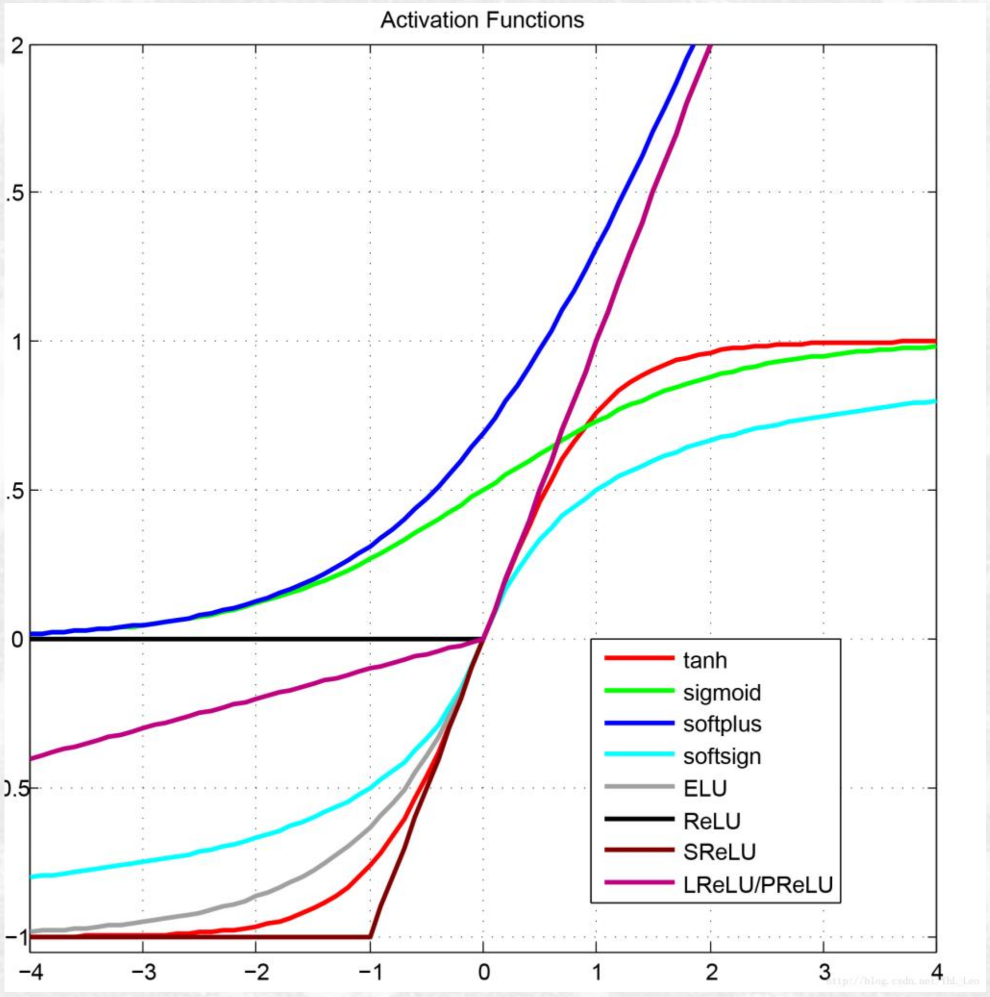
- We can adopt the *sigmoid* function instead of the *sgn*()
  - to bound the final values between 0 and 1
  - can be interpreted as probabilities of belonging to a class
  - belonging threshold is ">0.5"
- It remains a linear classifier

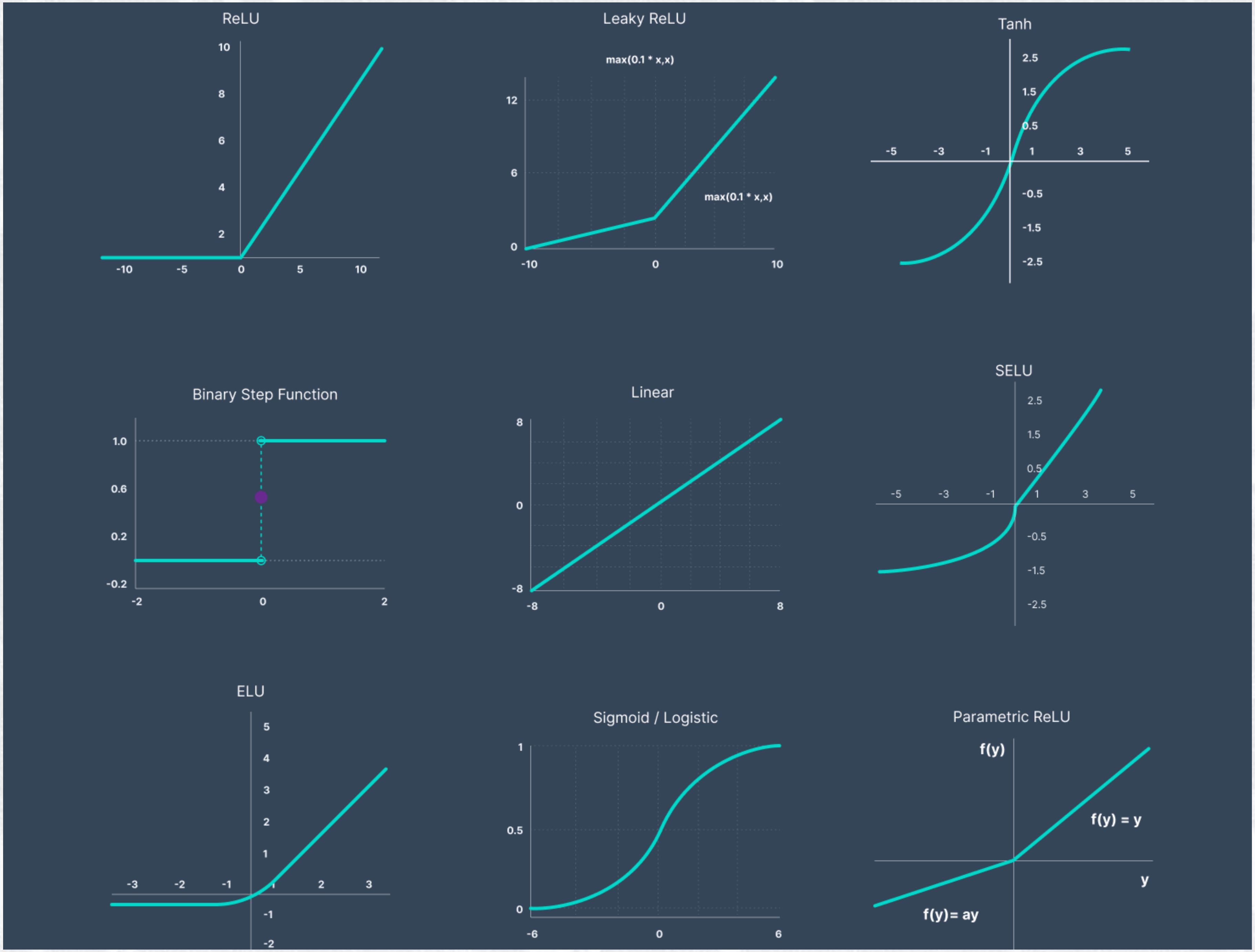
$$g(z) = \frac{1}{1+e^{-z}}$$

$$h(\vec{x}) = g(\sum_{n} \theta_n x_n + b)$$



Perceptron and non-linear activation functions




ReLU
Leaky ReLU
max(0.1 * x,x)
max(0.1 * x,x)
Tanh
Binary Step Function
Linear
SELU
ELU
Sigmoid / Logistic
Parametric ReLU
f(y)
f(y) = y
f(y)= ay
y

How to induce h from examples

- Learn the parameters θ and b
- To find these we look at the past data (i.e. training data) optimizing an objective function
- **Objective function**: the error we make on the training data
  - the sum of differences between the decision function h and the label y
  - also called **Loss Function** or **Cost Function**

$$J(\theta, b) = \sum_{i=1}^{m} (h(x^{(i)}; \theta, b) - y^{(i)})^2$$

A general training procedure: Stochastic Gradient Descent

- Optimizing J means **minimizing** it
  - it measures the errors we make on the training data.
- We can iterate over examples and update the parameters in the direction of smaller costs
  - we aim at finding the minimum of that function
- Concretely,

$$\theta_1 = \theta_1 - \alpha \Delta \theta_1$$
$$\theta_2 = \theta_2 - \alpha \Delta \theta_2$$
$$b = b - \alpha \Delta b$$

- α is a meta-parameter, the learning rate
- Δ are the partial derivatives of the cost function *wrt* each parameter

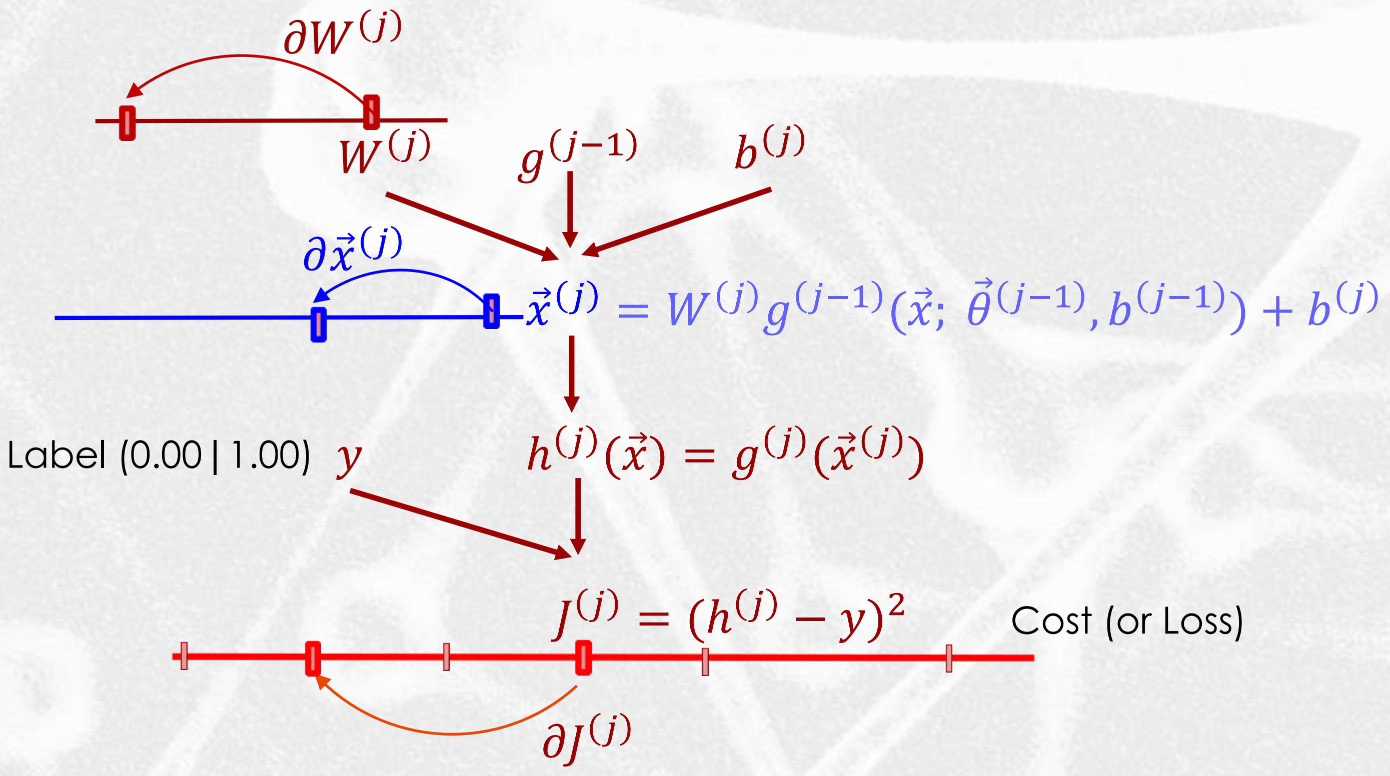
Optimizing *J*

- From the network

$$h(\vec{x}) = g^{(k)}\,(g^{(k-1)}(\ldots g^{(1)}(\vec{x}; \vec{\theta}^{(1)}, b^{(1)}); \ldots); \vec{\theta}^{(k-1)}, b^{(k-1)}); \vec{\theta}^{(k)}, b^{(k)}\,)=$$
$$= \; g^{(k)}\big(W^{(k)} g^{(k-1)}(W^{(k-1)} \ldots . g^{(1)}(W^{(1)}\vec{x} + \; b^{(1)}\big) \ldots + b^{(k-1)}) + b^{(k)}\big)$$

- and *j*-th layers equation:

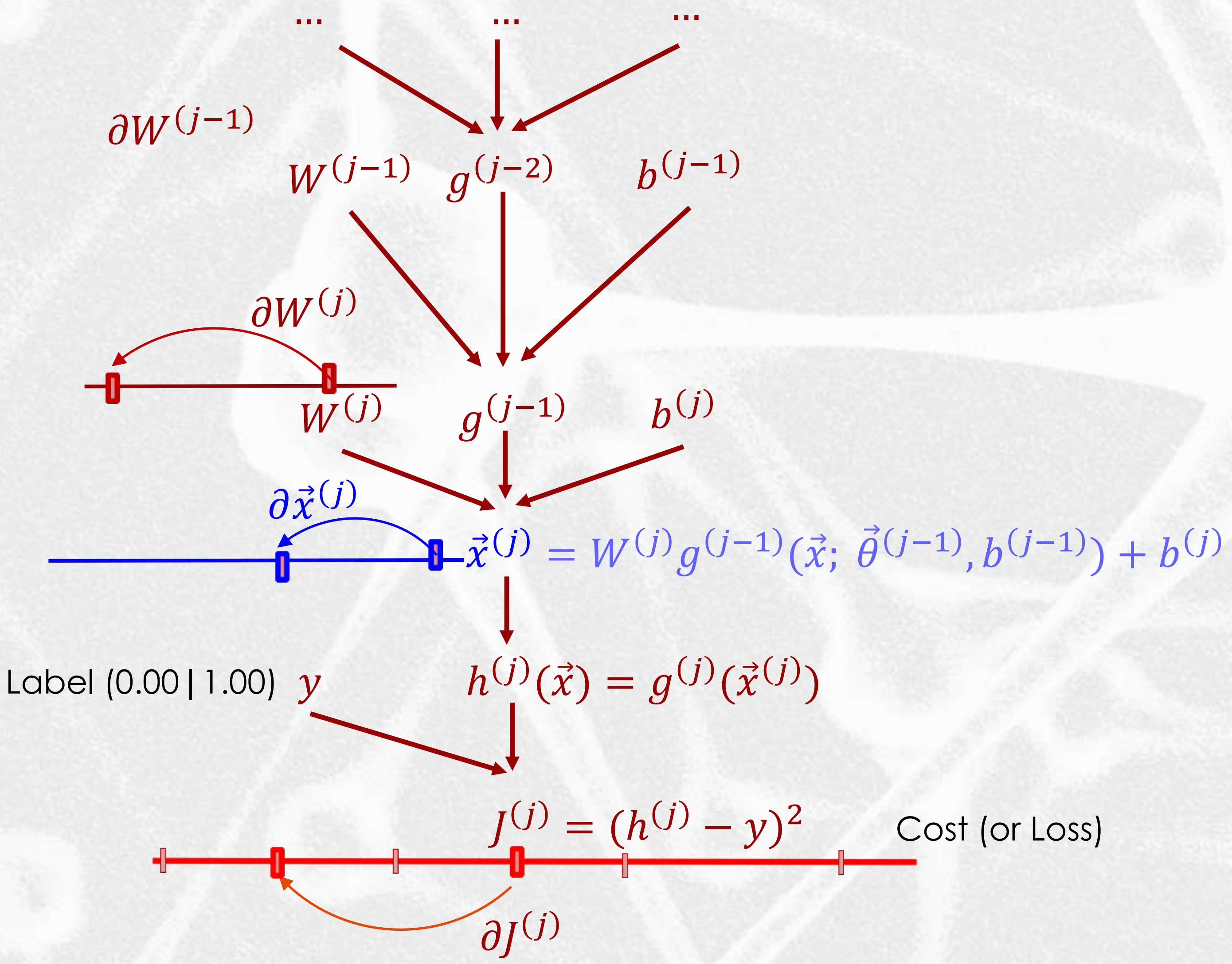
$$h^{(j)}(\vec{x}) = g^{(j)}\left(W^{(j)} g^{(j-1)}(\vec{x};\; \vec{\theta}^{(j-1)},\;\; b^{(j-1)}) + b^{(j)}\right) \qquad j = 2, \ldots ., k-1$$



$\partial W^{(j)}$
$W^{(j)}$
$g^{(j-1)}$
$b^{(j)}$
$\partial \vec{x}^{(j)}$
$\vec{x}^{(j)} = W^{(j)} g^{(j-1)}(\vec{x};\; \vec{\theta}^{(j-1)}, b^{(j-1)}) + b^{(j)}$
Label (0.00 | 1.00) y
$h^{(j)}(\vec{x}) = g^{(j)}(\vec{x}^{(j)})$
$J^{(j)} = (h^{(j)} - y)^2$
Cost (or Loss)
$\partial J^{(j)}$

$$\frac{\partial J^{(j)}}{\partial W^{(j)}}$$

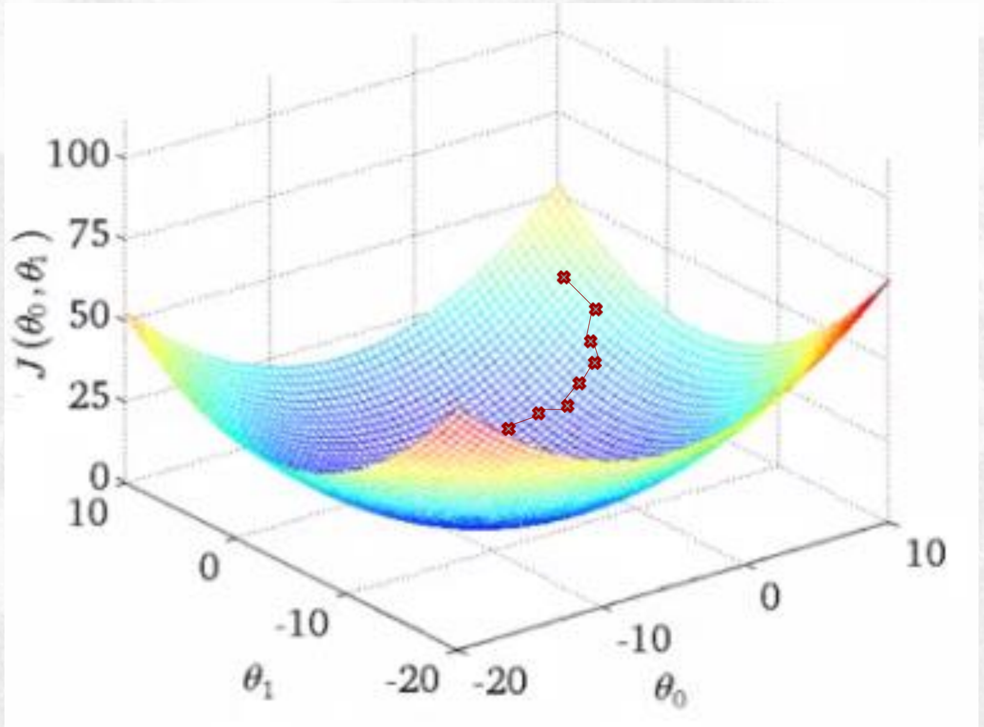
Optimizing J … backwards



$$\frac{\partial J^{(j)}}{\partial W^{(j)}}$$

Why SGD?

- Weights are updated using the partial derivatives
- Derivative pushes down the cost following the steepest descent path on the error curve



SGD procedure

- Choose an initial random values for θ and b
- Choose a learning rate
- Repeat until stop criterion is met:
  - Pick a random training example $x^{(i)}$
  - Update the parameters with

$$\theta_1 = \theta_1 - \alpha\Delta\theta_1$$
$$\theta_2 = \theta_2 - \alpha\Delta\theta_2$$
$$b = b - \alpha\Delta b$$

- We can stop **WHEN**
  - when the **parameters do not change** or,
  - the **number of iteration exceeds a certain upper bound**

Cost Function Derivative

- In order to update the parameters in SGD, we need to compute the **partial derivatives** *wrt* the learnable parameters.

- Remember the chain rule:
  - if J is a function of a given $z(x)$, then the derivative of J *wrt* x is:

$$\frac{\vartheta J}{\vartheta x} = \frac{\vartheta J}{\vartheta z}\frac{\vartheta z}{\vartheta x}$$

- Thus (in R^2), we need to compute
  - for the *i*-th example $x^{(i)}$

$$\Delta\vartheta_1 = \frac{\vartheta}{\vartheta\theta_1}(h(x^{(i)};\theta,b) - y^{(i)})^2$$

$$\Delta\vartheta_2 = \frac{\vartheta}{\vartheta\theta_2}(h(x^{(i)};\theta,b) - y^{(i)})^2$$

$$\Delta b = \frac{\vartheta}{\vartheta b}(h(x^{(i)};\theta,b) - y^{(i)})^2$$

Cost Function Derivatives

$$\Delta\theta_1 = \frac{\vartheta}{\vartheta\theta_1}(h(x^{(i)};\theta,b) - y^{(i)})^2 =$$

$$g(z) = \frac{1}{1+e^{-z}}$$

$$= 2((h(x^{(i)};\theta,b) - y^{(i)})\frac{\vartheta}{\vartheta\theta_1}(h(x^{(i)};\theta,b))$$

$$\frac{\vartheta g}{\vartheta z} = (1-g(z))g(z)$$

$$= 2(g(\theta^T x^{(i)} + b) - y^{(i)})\frac{\vartheta}{\vartheta\theta_1}(g(\theta^T x^{(i)} + b))$$

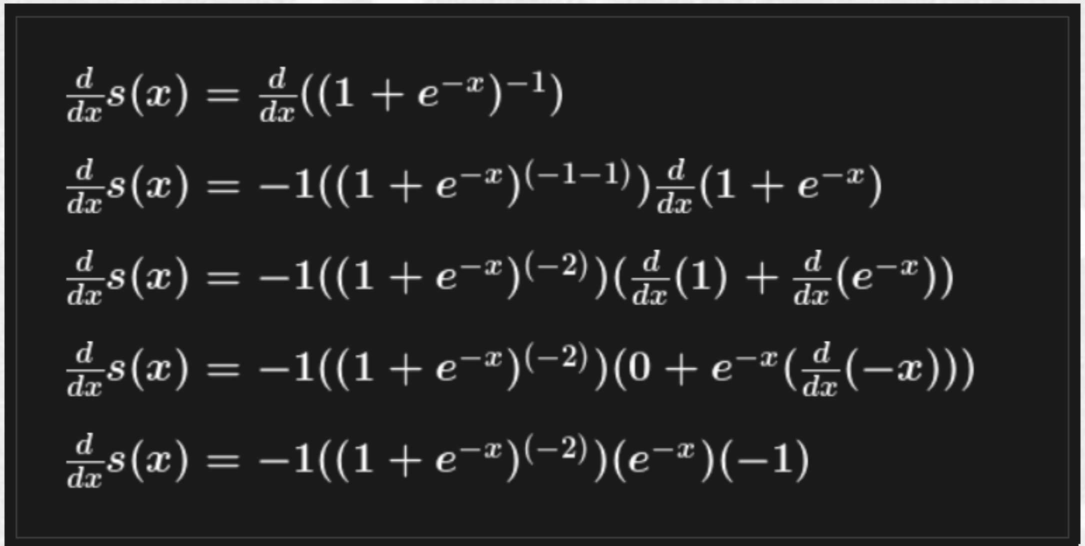
We have that:

$$\frac{\vartheta}{\vartheta\theta_1}(g(\theta^T x + b)) = \frac{\vartheta g(\theta^T x + b)}{\vartheta(\theta^T x + b)}\frac{\vartheta(\theta^T x + b)}{\vartheta\theta_1}$$

$$(1-g(\theta^T x + b))g(\theta^T x + b)\frac{\vartheta(\theta_1 x_1 + \theta_2 x_2 + b)}{\vartheta\theta_1}$$

$$= (1-g(\theta^T x + b))g(\theta^T x + b)x_1$$

$$s(x) = \frac{1}{1+e^{-x}} \quad \textit{then} \quad \frac{\vartheta s}{\vartheta x} = (1-s(x))s(x)$$



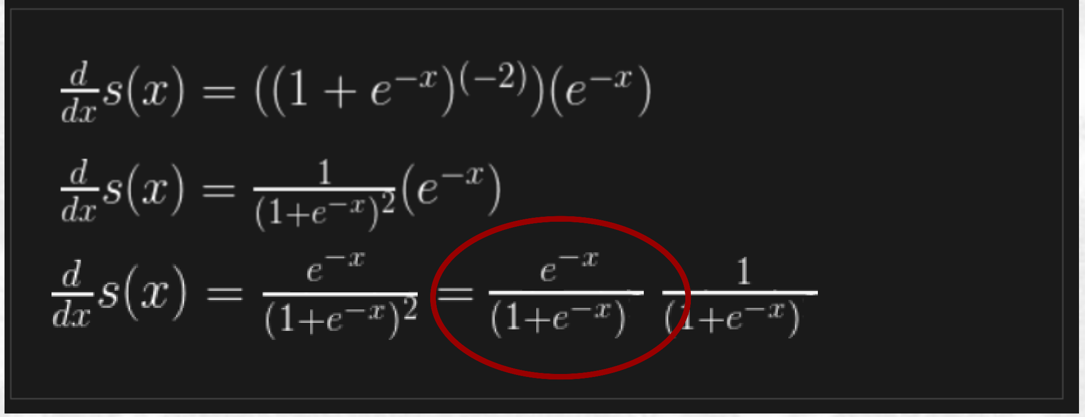
$$\frac{d}{dx}s(x) = \frac{d}{dx}((1+e^{-x})^{-1})$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-1-1)})\frac{d}{dx}(1+e^{-x})$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-2)})(\frac{d}{dx}(1) + \frac{d}{dx}(e^{-x}))$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-2)})(0 + e^{-x}(\frac{d}{dx}(-x)))$$

$$\frac{d}{dx}s(x) = -1((1+e^{-x})^{(-2)})(e^{-x})(-1)$$



$$\frac{d}{dx}s(x) = ((1+e^{-x})^{(-2)})(e^{-x})$$

$$\frac{d}{dx}s(x) = \frac{1}{(1+e^{-x})^2}(e^{-x})$$

$$\frac{d}{dx}s(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})} \frac{1}{(1+e^{-x})}$$

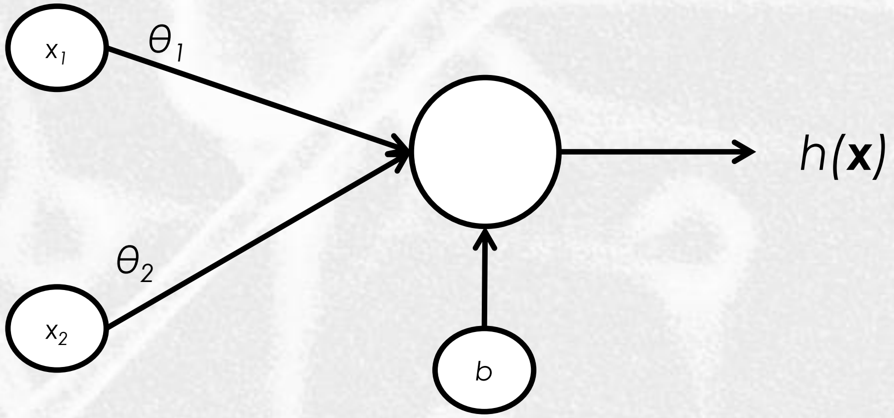
Cost Function Derivatives

Then,

$$\Delta\theta_1 = 2[(g(\theta^T x^{(i)} + b) - y^{(i)})][(1 - g(\theta^T x^{(i)} + b))g(\theta^T x^{(i)} + b)x^{(i)}{}_1]$$

and we can do the same for θ_2

$$\Delta\theta_2 = 2[(g(\theta^T x^{(i)} + b) - y^{(i)})][(1 - g(\theta^T x^{(i)} + b))g(\theta^T x^{(i)} + b)x^{(i)}{}_2]$$



Cost Function Derivatives for *b*

- For the b parameter, the same steps apply:

$$\Delta b = \frac{\vartheta}{\vartheta b}(h(x^{(i)};\theta,b) - y^{(i)})^2 =$$

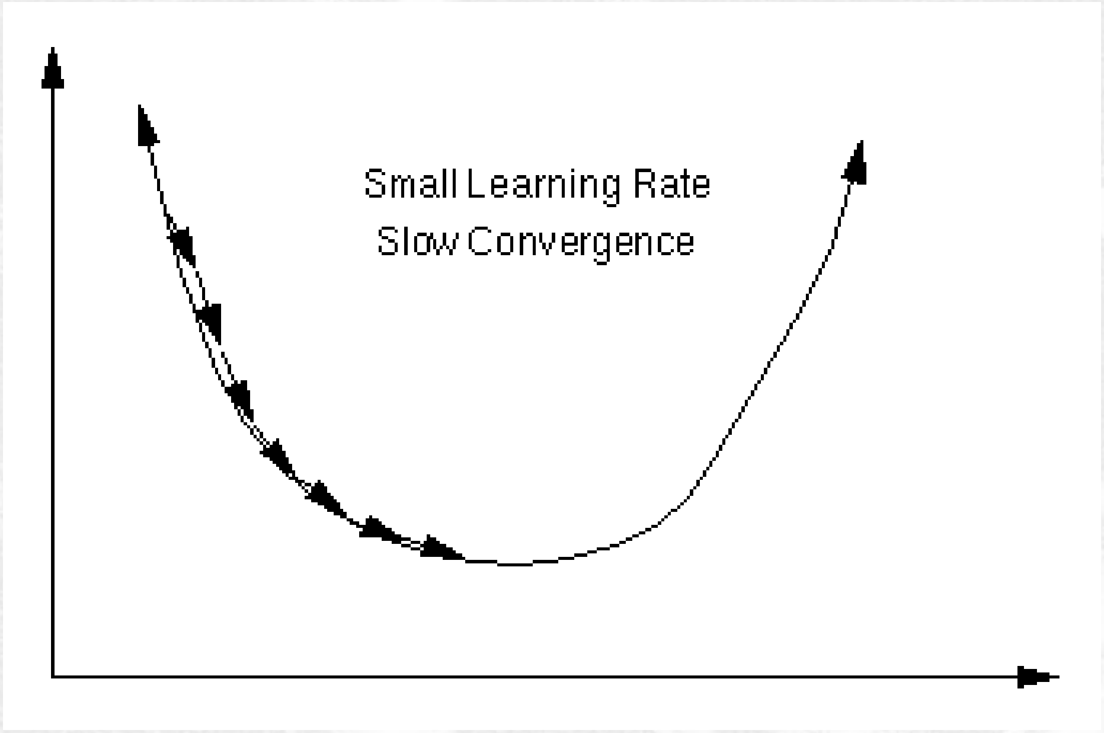
$$= 2((h(x^{(i)};\theta,b) - y^{(i)})\frac{\vartheta}{\vartheta b}(h(x^{(i)};\theta,b))$$

$$= 2(g(\theta^T x^{(i)} + b) - y^{(i)})\frac{\vartheta}{\vartheta b}(g(\theta^T x^{(i)} + b))$$

$$\frac{\vartheta}{\vartheta b}(g(\theta^T x + b)) = \frac{\vartheta g(\theta^T x + b)}{\vartheta(\theta^T x + b)}\frac{\vartheta(\theta^T x + b)}{\vartheta b} = (1 - g(\theta^T x + b))g(\theta^T x + b)$$

$$\Delta b = 2[(g(\theta^T x^{(i)} + b) - y^{(i)})][(1 - g(\theta^T x^{(i)} + b))g(\theta^T x^{(i)} + b)]$$

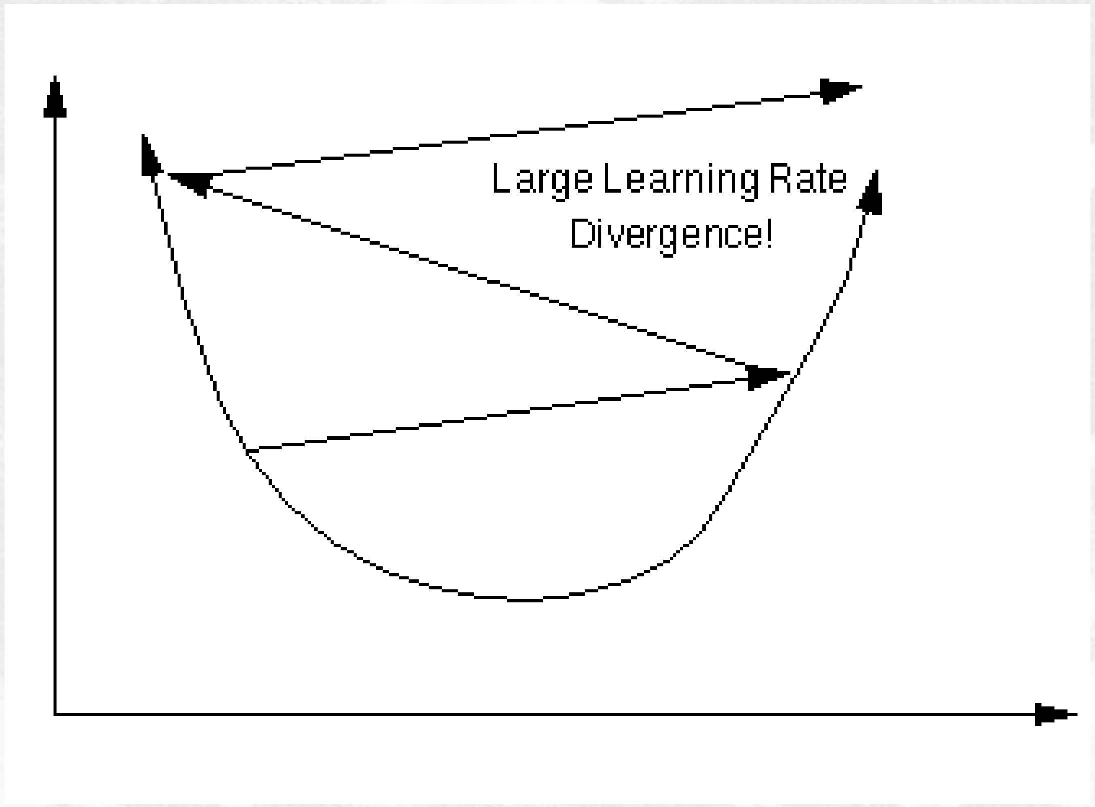
Learning rate: low values



- make the algorithm converge slowly
- it is a conservative and safer choice
- However, it implies very long training

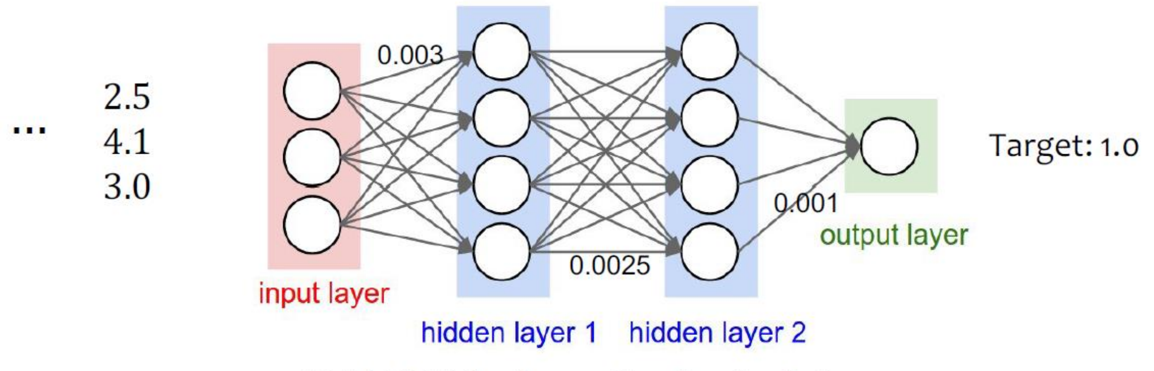
$$\theta_1 = \theta_1 - \alpha \Delta \theta_1$$
$$\theta_2 = \theta_2 - \alpha \Delta \theta_2$$
$$b = b - \alpha \Delta b$$

Learning rate: high values

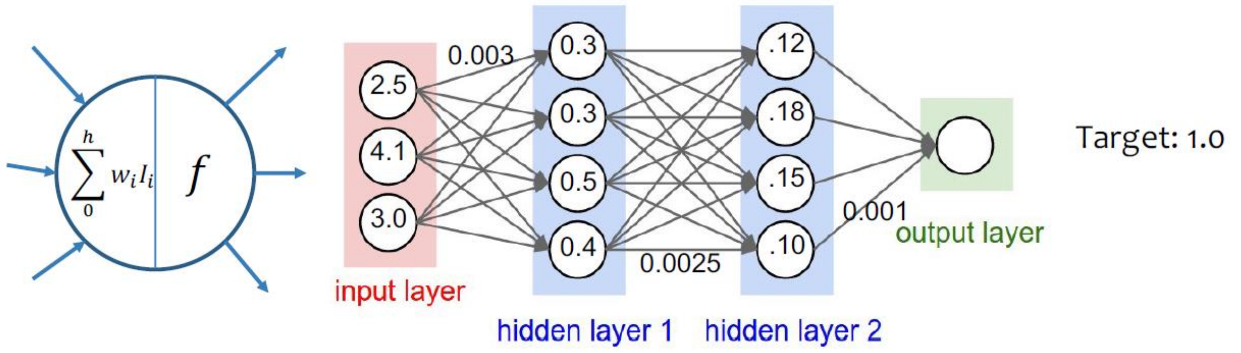


- make the algorithm converge quickly
- Training time is reduced
- it is a a less safer choice
  - risk of divergence

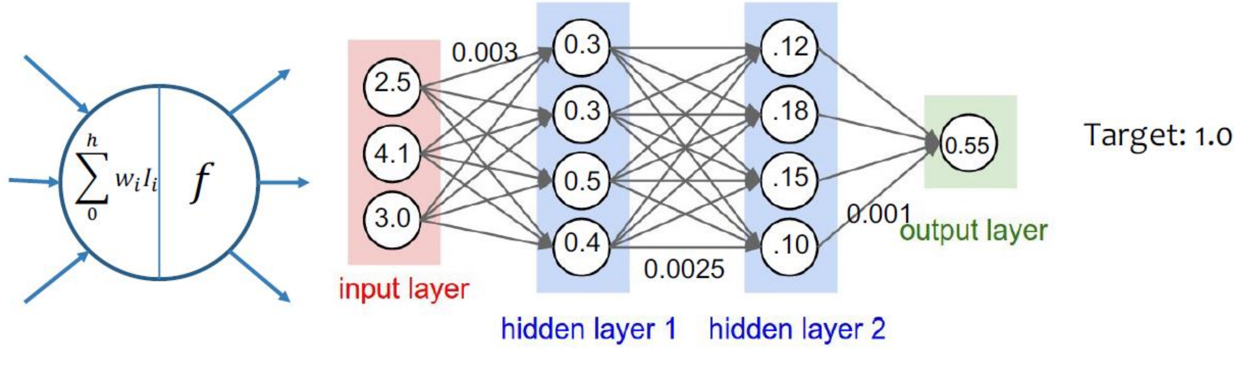
$$\theta_1 = \theta_1 - \alpha\Delta\theta_1$$
$$\theta_2 = \theta_2 - \alpha\Delta\theta_2$$
$$b = b - \alpha\Delta b$$



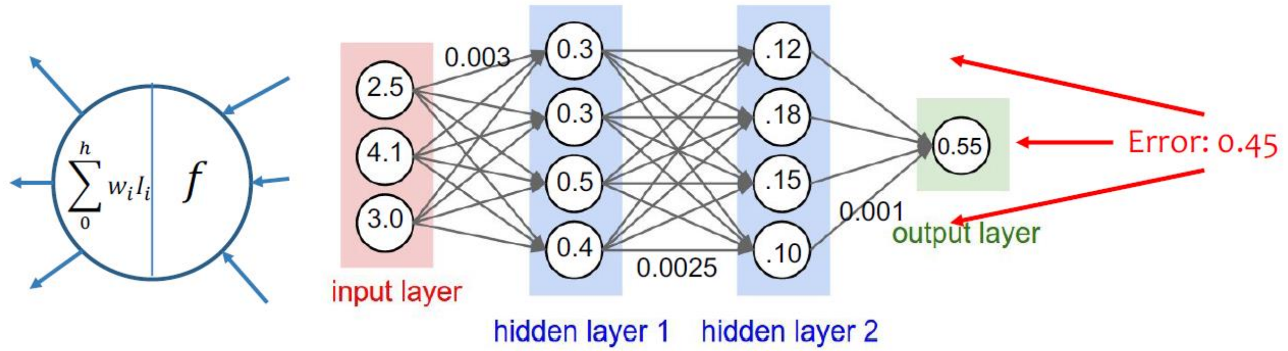
Weight Initialization and Loading the Data



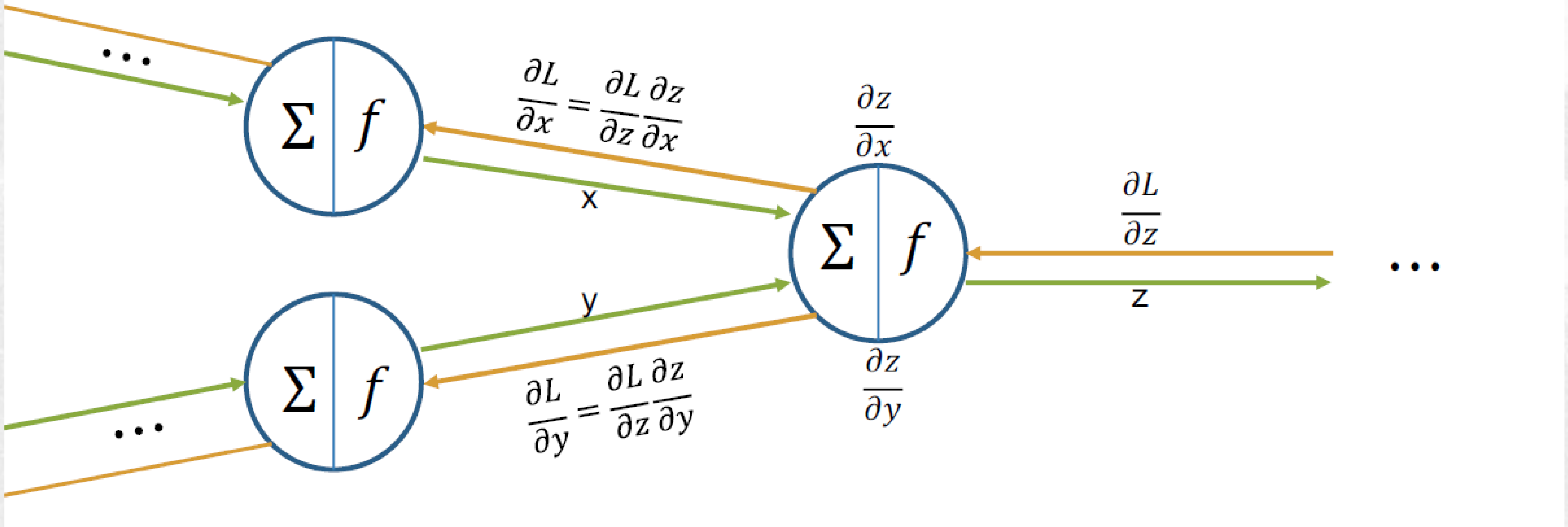
Going forward (Regularization, Dropout, Batch Normalization, …)



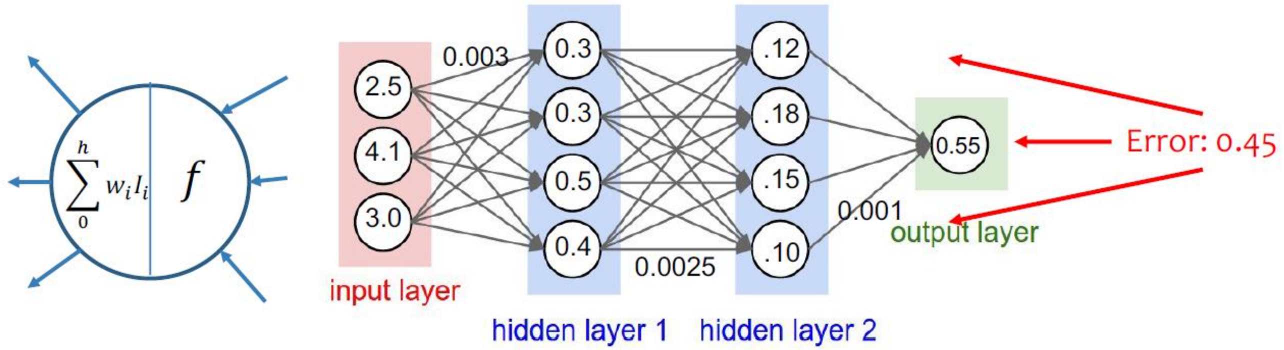
$Loss\ Function: (T - O) \Rightarrow Loss: 0.45$



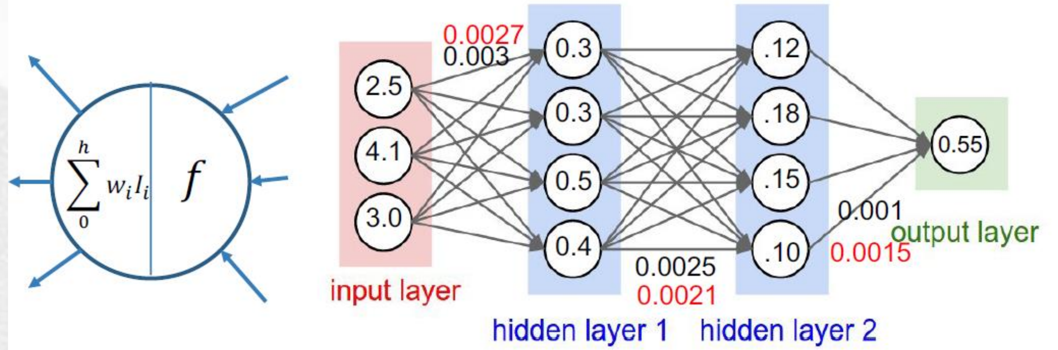
Updating the weights using Backpropagation $\frac{\partial E}{\partial w_j^i}$


$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial z}$
x
y
z
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
Σ
f
...

Backpropagation



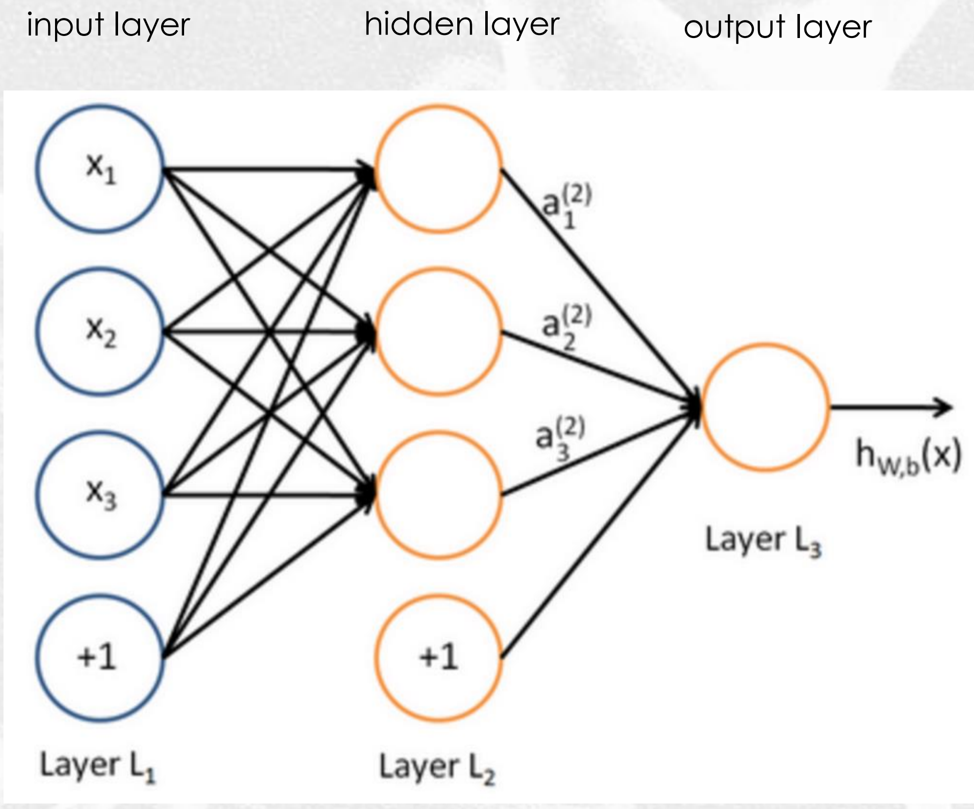
Updating the weights using Backpropagation $\frac{\partial E}{\partial w_j^i}$



By Choosing the Optimizer, new weights will be computed $w_{new} = w - \eta \frac{\partial E}{\partial w}$

Multilayer Networks

- Each circle represent a **neuron** (or unit)
  - 3 inputs, 3 hiddens and 1 output
- $n_l=3$ is the number of layers
- s_l denotes the number of units in layer l
- Layers:
  - Layer l is denoted as L_l
  - Layer l and $l+1$ are connected by a matrix of parameters $W^{(l)}$
    - $W^{(l)}_{i,j}$ connects neuron j in layer l with neuron *i* in layer $l+1$
- $b^{(l)}_i$ is the bias associated to neuron *i* in layer $l+1$



Multilayer Networks cont.

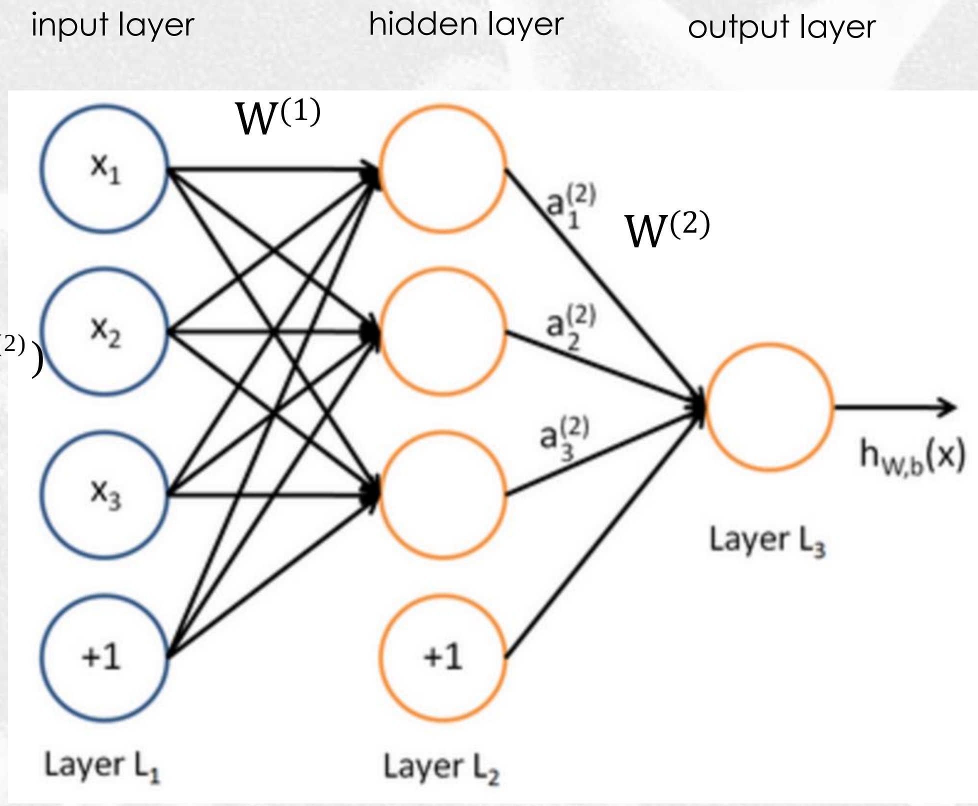
- $a^{(l)}_I$ is the activation of unit i in layer l
  - for l=1 $a^{(1)}_i = x_i$

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

- We call $z^{(l)}_i$ the weighted sum of inputs to unit i in layer l, i.e.

$$z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}$$
$$a_i^{(l)} = f(z_i^{(l)})$$

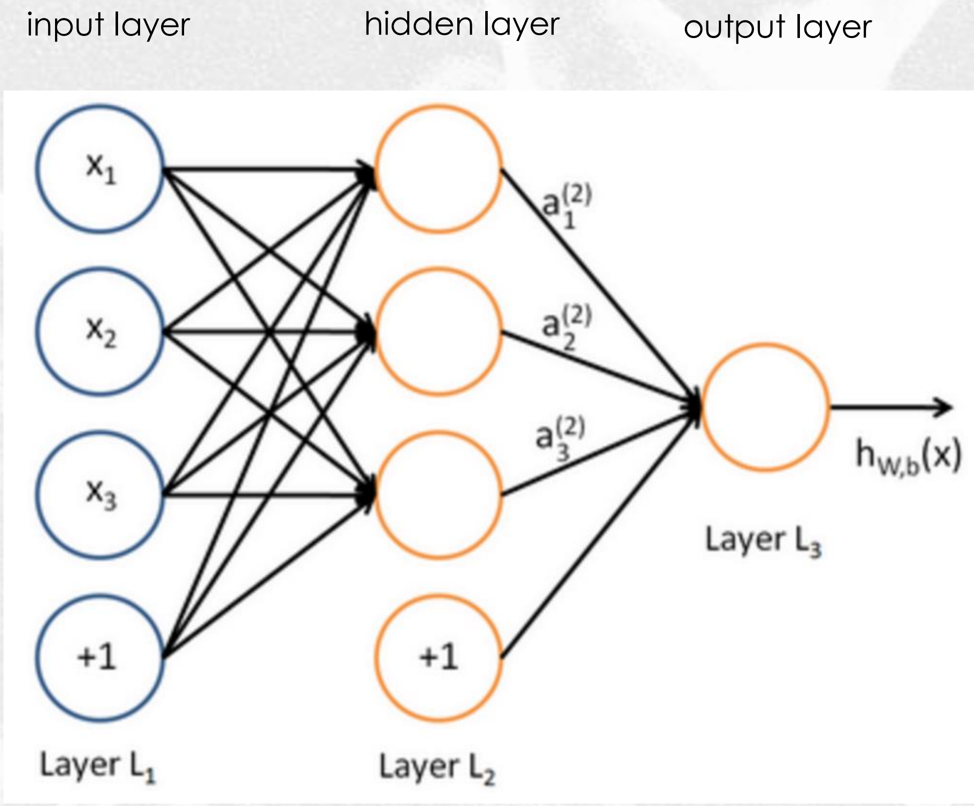
- f is a non-linearity function
  - e.g. the sigmoid



Multilayer Network Classification

- The classification corresponds in getting the value(s) in the output layer
- Propagating the input towards the network given W,b
- This process is called **forward propagation**

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = g(z^{(l+1)})$$



How to Train a NN?

- We can **re-use the gradient descent algorithm**
  - define a cost function
  - compute the partial derivatives *wrt* to all the parameters
- As the NN models function composition
  - we are going to exploit the chain rule (again)
- Setup:
  - we have a training set of m examples
  - $\{(x^{(1)},y^{(1)}), \ldots, (x^{(m)},y^{(m)})\}$
  - x are the inputs and y are the labels

$h(z(x))$

$$\frac{\vartheta h}{\vartheta x} = \frac{\vartheta h}{\vartheta z}\frac{\vartheta z}{\vartheta x}$$

Cost Function of a NN

- Given a single training example *(x,y)* the cost is

$$J(W,b;x,y)=\frac{1}{2}|h_{W,b}(x)-y|^2$$

- For the whole training set *J* is the mean of the errors plus a regularization term (**weight decay**)

$$J(W,b)=\frac{1}{m}\sum_{i=1}^{m}J(W,b;x^{(i)},y^{(i)})+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

$$=\frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}|h_{W,b}(x^{(i)})-y^{(i)}|^2)+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

- λ controls the importance of the two terms (it has a similar role to the C parameter in SVM)

… *digression:* On regularization

- *"any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."*
- In practical deep learning scenarios: the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately
- Many regularization approaches are based on *limiting the capacity of models*, such as neural networks, linear regression, or logistic regression, *by adding a parameter norm penalty $\Omega(\theta)$* to the objective function *J*
- **Regularization methods**:
  - Weight decay (*ridge regression*)
  - … Constrained optimization
  - Data Augmentation
  - Early stopping

A GD step

- A GD step update the parameters according to

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\vartheta}{\vartheta W_{ij}^{(l)}} J(W, b)$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \alpha \frac{\vartheta}{\vartheta b_{i}^{(l)}} J(W, b)$$

- where α is the learning rate.
- The partial derivatives are computed with the **Backpropagation** algorithm

The backpropagation algorithm

- First, we compute for each example $\frac{\vartheta}{\vartheta W_{ij}^{(l)}} J(W,b,x^{(i)},y^{(i)})$
- Backpropagation works as follow:
  1. do a forward pass for an example $x^{(i)},y^{(i)}$
  2. for each node i in layer l, compute an error term δ^{l}_{i}
     1. it measures how unit i is responsible for the error on the current example
  3. The error of an output node is the difference between the true output value and the predicted one
  4. For the intermediate layer l, a node receives a portion of the error based on the units it is linked to of the layer $l+1$
- Partial derivatives will be computed given the error terms

The backpropagation algorithm cont.

1. Perform a forward propagation for an example
2. For each unit i in the output layer (n_l)

$$\delta_i^{(n_l)} = \frac{\vartheta}{\vartheta z_i^{(n_l)}} |y - h_{W,b}(x)|^2 = -(y_i - a_i^{(n_l)}) \cdot g'(z_i^{(n_l)})$$

3. For $l=n_l-1,\ldots,2$
   1. for each node i in layer l $\quad \delta_i^{(l)} = (\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}) g'(z_i^{(l)})$
4. Compute the partial derivatives as:

$$\frac{\vartheta}{\vartheta W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\vartheta}{\vartheta b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}$$

The full backpropagation algorithm

1. Set $\Delta W^{(l)}=0$, $\Delta b^{(l)}=0$ for all l

2. For each examples (x,y), for each layer l
   1. Compute $\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^T, \nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}$
   2. Set $\Delta W^{(l)} = \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y)$

      $\Delta b^{(l)} = \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y)$
   3. Update the parameters with:

$$W^{(l)} = W^{(l)} - \alpha[(\frac{1}{m}\Delta W^{(l)}) + \lambda W^{(l)}]$$

$$b^{(l)} = b^{(l)} - \alpha[(\frac{1}{m}\Delta b^{(l)})]$$

Some considerations

- Randomly initialize the parameters of the network
  - for symmetry breaking
- Remember that the function *g* is a non-linear activation function
  - if g is the sigmoid

$$g(z) = \frac{1}{1+e^{-z}}$$

$$g'(z) = (1-g(z))g(z)$$

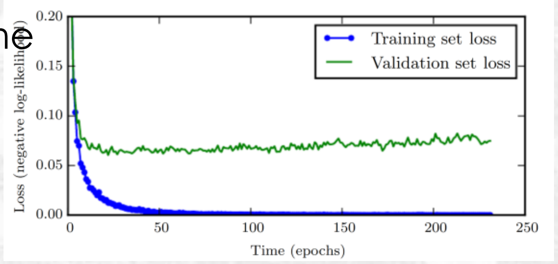
- Activations values can be cached from the forward propagation step!

$$g'(z_i^{(l)}) = (1-g(z_i^{(l)}))g(z_i^{(l)}) = (1-a_i^{(l)})a_i^{(l)}$$

- If you must perform multi-classification
  - there will be an output unit for each of the labels

Some considerations

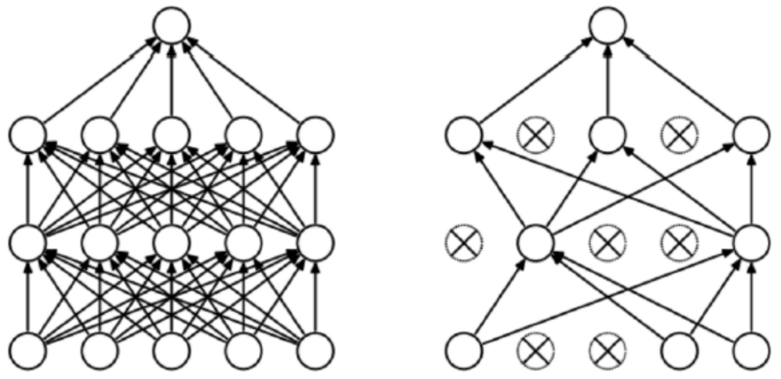
- How to stop and select the best model
  - Waiting the iteration in which the cost function doesn't change significantly
    - Risk of overfitting
- **Early stopping**
  - Provide hints as to how many iterations can be run before overfitting
  - Split the original training set into a new training set and a validation set
  - Train only on the training set and evaluate the error on the validation set
  - Stop training as soon as the error is higher than it was the last time
  - Use the weights the network had in that previous step
- **Dropout**
  - another form of regularization to avoid overfitting data
  - during training (only) randomly "turn off" some of the neurons of a layer
  - it prevents co-adaptation of units between layers



Dropout (Svrivastava et al., 2014)

- Dropout can be interpreted as a way of regularizing a neural network by adding noise to its hidden units.
- It speeds-up the learning algorithm through model averaging
- It helps in reducing the risk of greedily promote simplistic solutions

Randomly setting a fraction rate of input units to 0 at each update during training time.



(a) Standard Neural Net (b) After applying dropout.

Dropout: effects

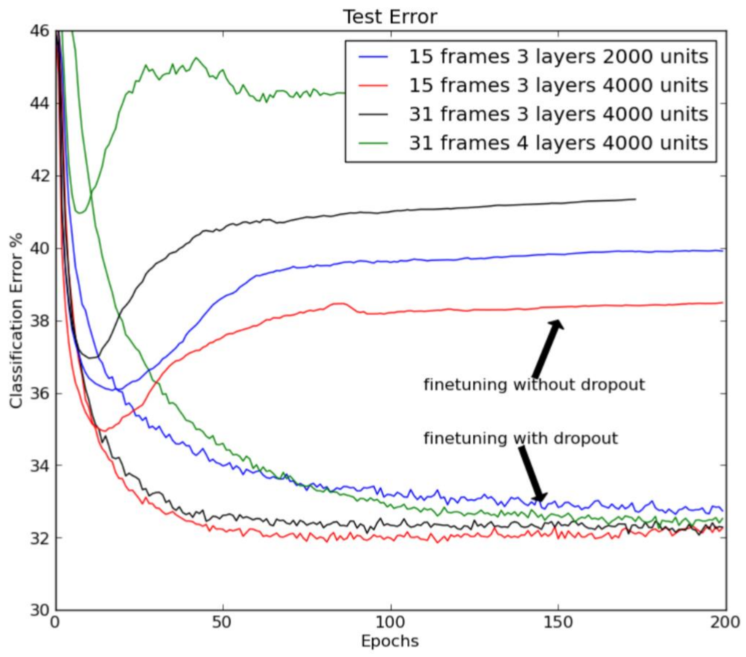


Fig. 2: The frame *classification* error rate on the core test set of the TIMIT benchmark. Comparison of standard and dropout finetuning for different network architectures. Dropout of 50% of the hidden units and 20% of the input units improves classification.

Dropout: effects

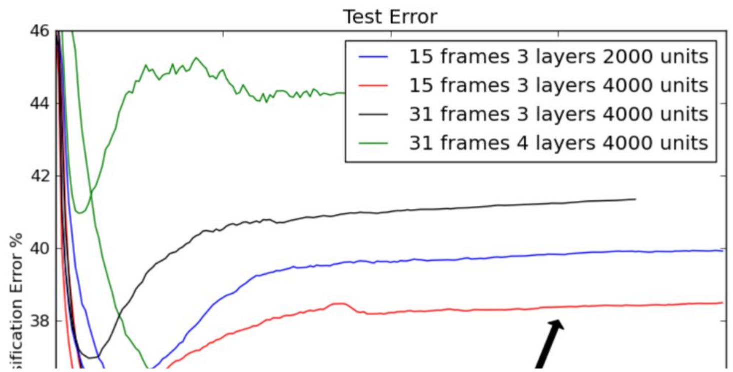


Fig. 2:
pariso
of the

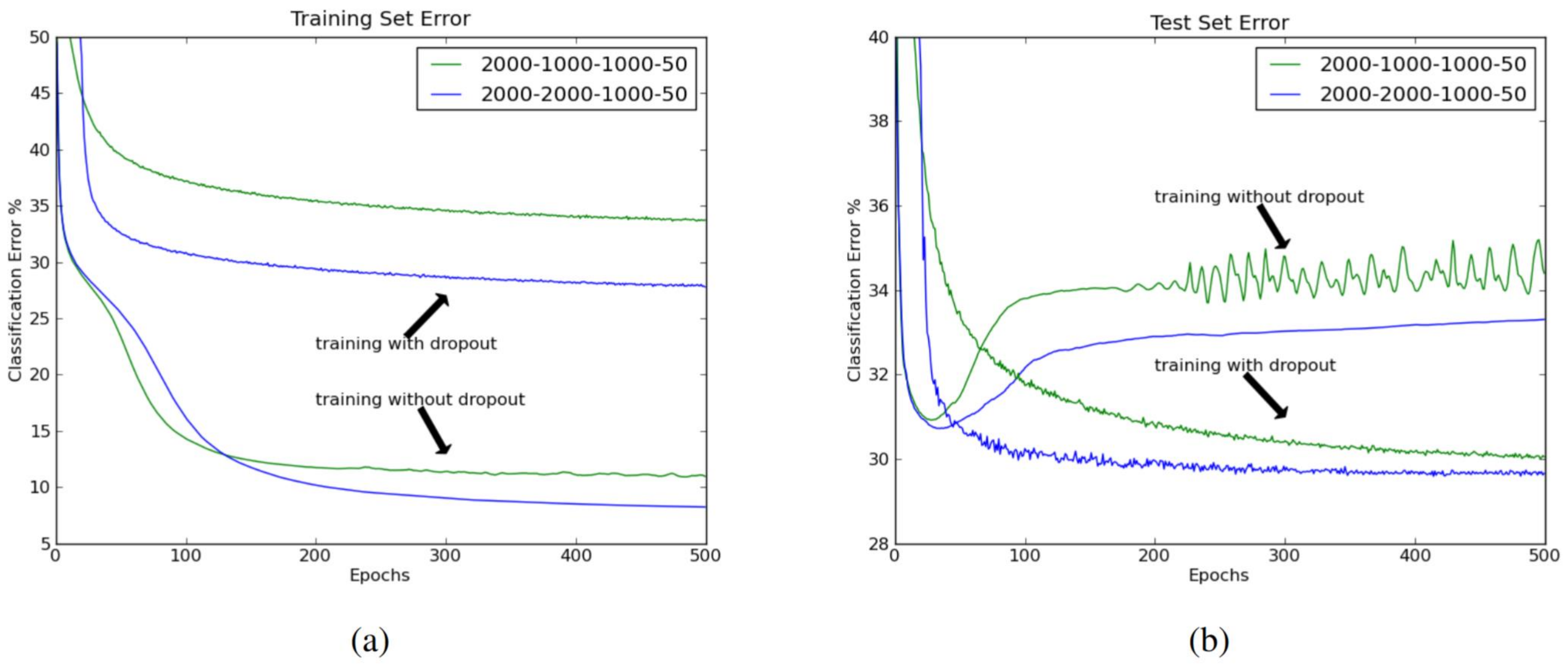


Fig. 7: Classification error rate on the (a) training and (b) validation sets of the Reuters dataset as learning progresses. The training error is computed using the stochastic nets.

Next steps …
complex NN architectures

- Convolutional Neural Networks (Neocogitron, Fukushima (1980))
- Recurrent Neural Networks (Jordan, 1986), (Elman, 1990)
    - Bidirectional RNNs (Schuster and Paliwal, 1997)
    - BP Through-Time (Robinson & Fallside, 1987)
    - Long Short Time Memories LSTMS, (Hochreiter & Schmidhuber, 1997)
    - Attention mechanisms (firstly discussed by (Larochelle & Hinton, 2010; Denil et al., 2012)).
- Autoencoders (Bengio et al., 2007), Encoder-Decoders (Cho et al., 2015)

Bibliografia: an historical overview

- Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115{133, 1943.
- Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6):386, 1958.
- Donald Olding Hebb. The organization of behavior: A neuropsychological theory. Psychology Press, 1949.
- John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences, 79(8):2554-2558, 1982.
- David E Rumelhart, Georey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.
- Rumelhart, D. E., McClelland, J. L., and the PDP Research Group (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition. MIT Press, Cambridge.
- Teuvo Kohonen. The self-organizing map. Proceedings of the IEEE, 78(9):1464{1480, 1990.
- David H Ackley, Georey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. Cognitive science, 9(1):147-169, 1985.
- Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaected by shift in position. Biological cybernetics, 36(4): 193-202, 1980.
- Le Cun B. Boser, John S. Denker, D. Henderson, Richard E. Howard, W. Hubbard and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In Advances in neural information processing systems. Citeseer, 1990.

Bibliografia: an historical overview (2)

- Michael I Jordan. Serial order: A parallel distributed processing approach. Advances in psychology, 121:471-495, 1986.
- Jerey L Elman. Finding structure in time. Cognitive science, 14(2):179-211, 1990.
- AJ Robinson and Frank Fallside. The utility driven dynamic error propagation network. University of Cambridge Department of Engineering, 1987.
- Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11):2673-2681, 1997.
- Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735-1780, 1997.
- Hugo Larochelle and Georey E Hinton. Learning to combine foveal glimpses with a third-order boltzmann machine. In Advances in neural information processing systems, pages 1243-1251, 2010
- Denil, M., Bazzani, L., Larochelle, H., and de Freitas, N. (2012). Learning where to attend with deep architectures for image tracking. Neural Computation, 24 (8), 2151–2184
- Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. Advances in neural information processing systems, 19:153, 2007.
- Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078, 2014.