DEEP LEARNING
FOR ARTIFICIAL INTELLIGENCE

Master Course UPC ETSETB TelecomBCN Barcelona. Autumn 2017.

Instructors

Xavier Giró-i-Nieto
Marta R. Costa-jussà
Jordi Torres
Elisa Sayrol
Santiago Pascual
Verónica Vilaplana
Ramon Morros
Javier Ruiz

Organizers

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH
UPC

telecom BCN

Supporters

aws educate
Barcelona Supercomputing Center
GitHub Education

+ info: http://dlai.deeplearning.barcelona

Day 2 Lecture 1

Multilayer Perceptron

Elisa Sayrol

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
UPC
Department of Signal Theory
and Communications
Image Processing Group

Acknowledgements

Antonio Bonafonte

Kevin McGuinness
kevin.mcguinness@dcu.ie
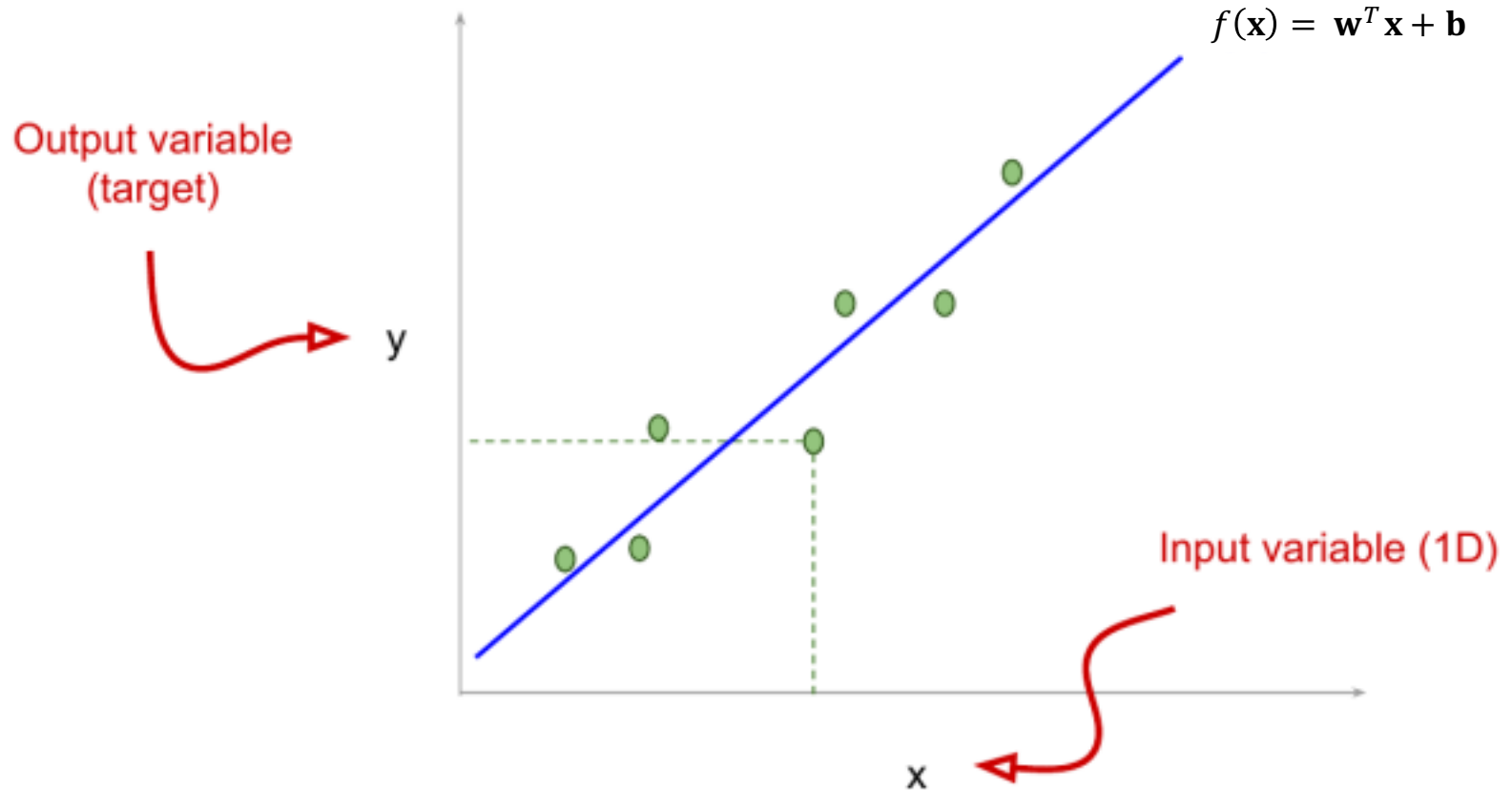
Research Fellow
Insight Centre for Data Analytics
Dublin City University

…in our last lecture

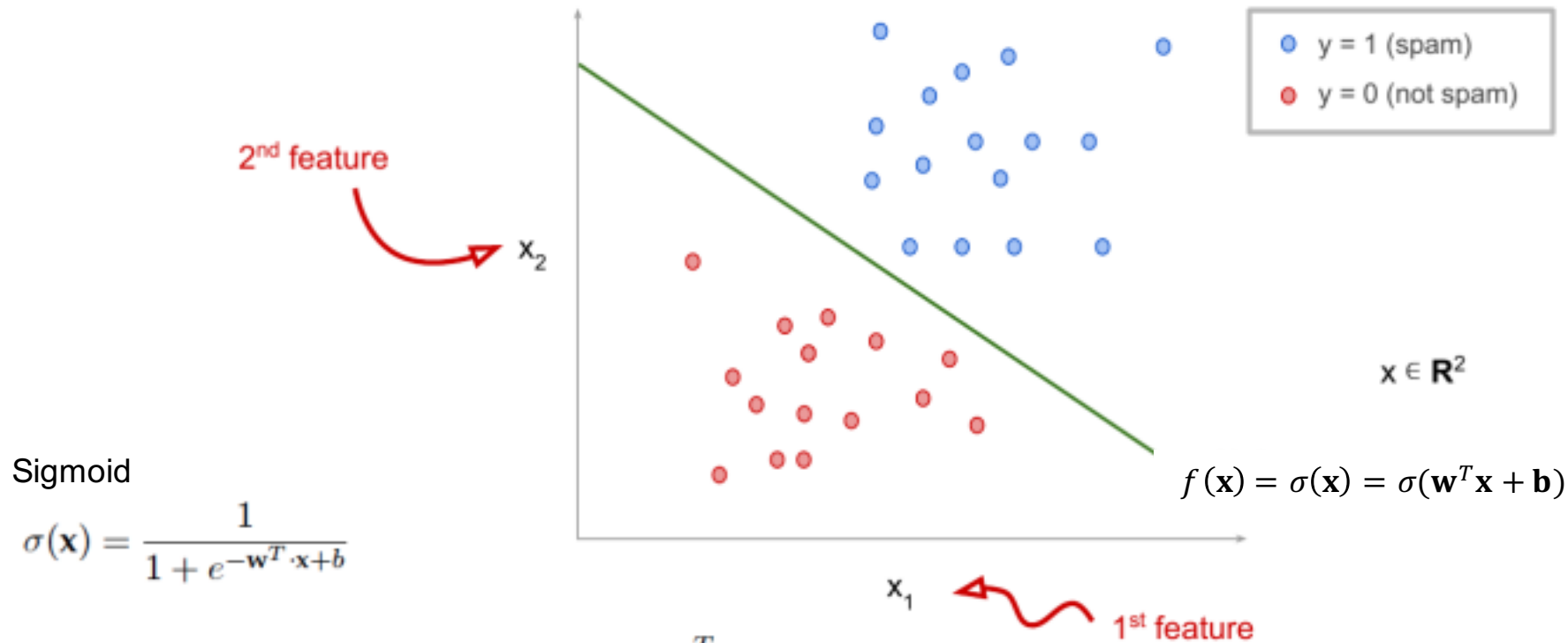# Linear Regression (eg. 1D input - 1D ouput)

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + \mathbf{b}$$

Output variable
(target)

y

Input variable (1D)

x

Binary Classification (eg. 2D input, 1D ouput)

Sigmoid

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \cdot \mathbf{x} + b}}$$

$f(\mathbf{x}) = \sigma(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + \mathbf{b})$

$\mathbf{x} \in \mathbf{R}^2$

MultiClass: Softmax

$$P(y = k | \mathbf{x}) = \frac{\exp \mathbf{x}^T \mathbf{w}_k}{\sum_{n=1}^{N} \exp \mathbf{x}^T \mathbf{w}_n}$$

Non-linear decision boundaries
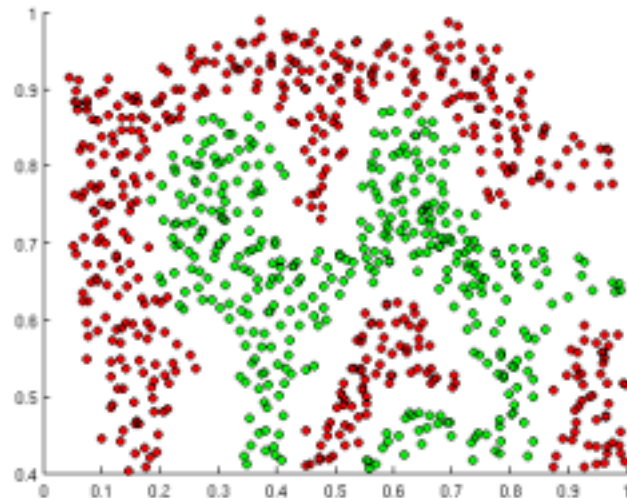
Linear models can only produce linear
decision boundaries

Real world data often needs a non-linear
decision boundary

    Images

    Audio

   Text

Non-linear decision boundaries

What can we do?

1. Use a non-linear classifier
    - Decision trees (and forests)
    - K nearest neighbors

2. Engineer a suitable representation
    - One in which features are more linearly separable
    - Then use a linear model

3. Engineer a kernel
    - Design a kernel $K(x1, x2)$
    - Use kernel methods (e.g. SVM)

4. Learn a suitable representation space from the data
    - Deep learning, deep neural networks
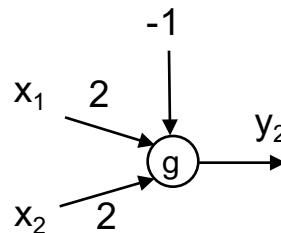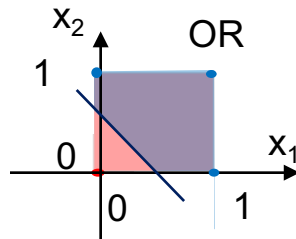   - Boosted cascade classifiers like Viola Jones also take this approach
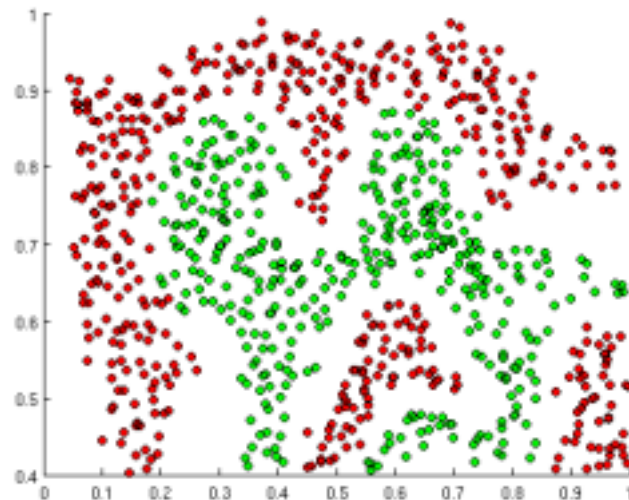
Example: X-OR.
AND and OR can be generated with a single perceptron
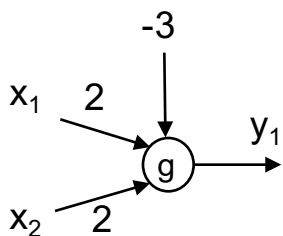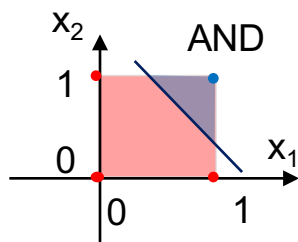
Input vector (x1,x2)	Class AND
(0,0)	0
(0,1)	0
(1,0)	0
(1,1)	1

Input vector (x1,x2)	Class OR
(0,0)	0
(0,1)	1
(1,0)	1
(1,1)	1

$$y_1 = g(\mathbf{w}^T\mathbf{x} + b) = u\left((2 \quad 2)\cdot\begin{pmatrix}x_1\\x_2\end{pmatrix} - 3\right)$$

$$y_2 = g(\mathbf{w}^T\mathbf{x} + b) = u\left((2 \quad 2)\cdot\begin{pmatrix}x_1\\x_2\end{pmatrix} - 1\right)$$

Example: X-OR
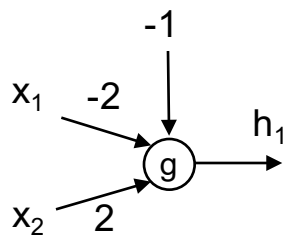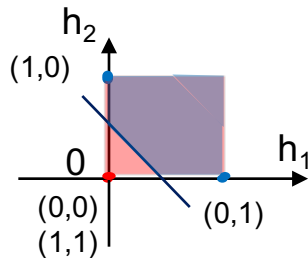# X-OR a Non-linear separable problem can not be generated with a single perceptron
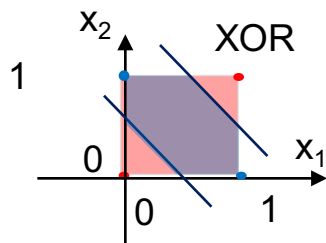
Input vector (x1,x2)	Class XOR
(0,0)	0
(0,1)	1
(1,0)	1
(1,1)	0

Example: X-OR. However…..
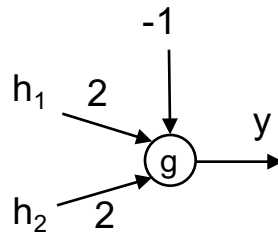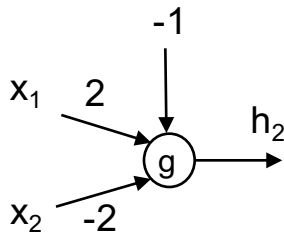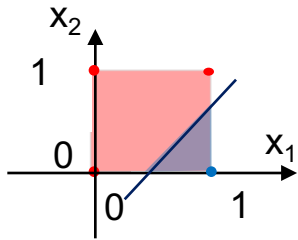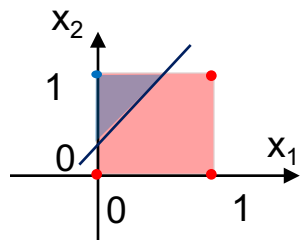
$$h_1 = g\left(\mathbf{w}_{11}^T \, \mathbf{x} + b_{11}\right) = u\left(\begin{pmatrix} -2 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 1\right)$$

$$h_2 = g\left(\mathbf{w}_{12}^T \, \mathbf{x} + b_{12}\right) = u\left(\begin{pmatrix} 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + 1\right)$$

$$y = g\left(\mathbf{w}_2^T \, \mathbf{h} + b_2\right) = u\left(\begin{pmatrix} 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} + 1\right)$$

Example: X-OR. Finally

Three layer Network:
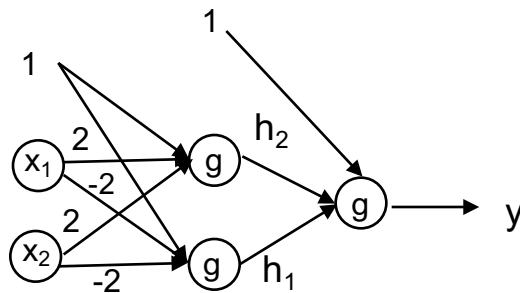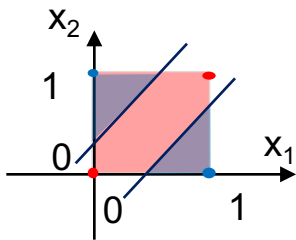
-Input Layer
-Hidden Layer
-Output Layer

2-2-1 Fully connected topology
(all neurons in a layer connected
Connected to all neurons in the
following layer)

$$h_1 = g\left(\mathbf{w}_{11}^T \mathbf{x} + b_{11}\right) = u\left(\begin{pmatrix} -2 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 1\right)$$

$$h_2 = g\left(\mathbf{w}_{12}^T \mathbf{x} + b_{12}\right) = u\left(\begin{pmatrix} 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + 1\right)$$

$$y = g\left(\mathbf{w}_2^T \mathbf{h} + b_2\right) = u\left(\begin{pmatrix} 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} + 1\right)$$

Another Example: Star Region (Univ. Texas)
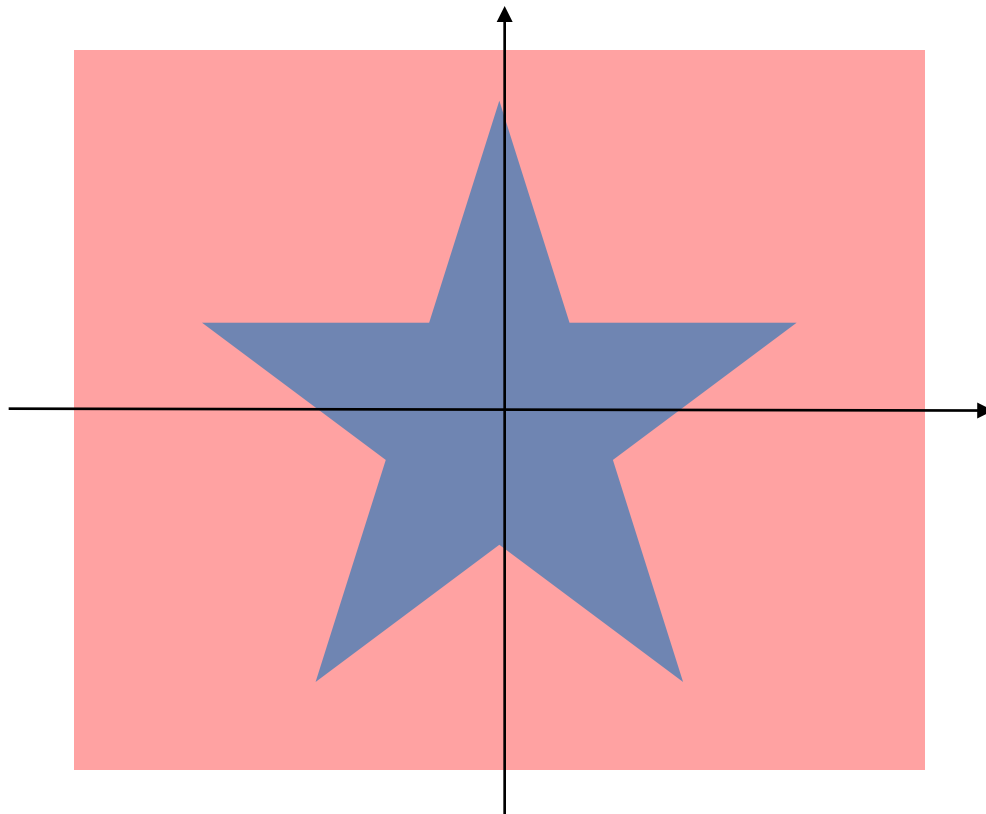
Neural networks

A neural network is simply a **composition** of simple neurons into several layers

Each neuron simply computes a **linear combination** of its inputs, adds a bias, and passes the result through an **activation function** g(x)

The network can contain one or more **hidden layers**. The outputs of these hidden layers can be thought of as a new **representation** of the data (new features).

The final output is the **target** variable (y = f(x))
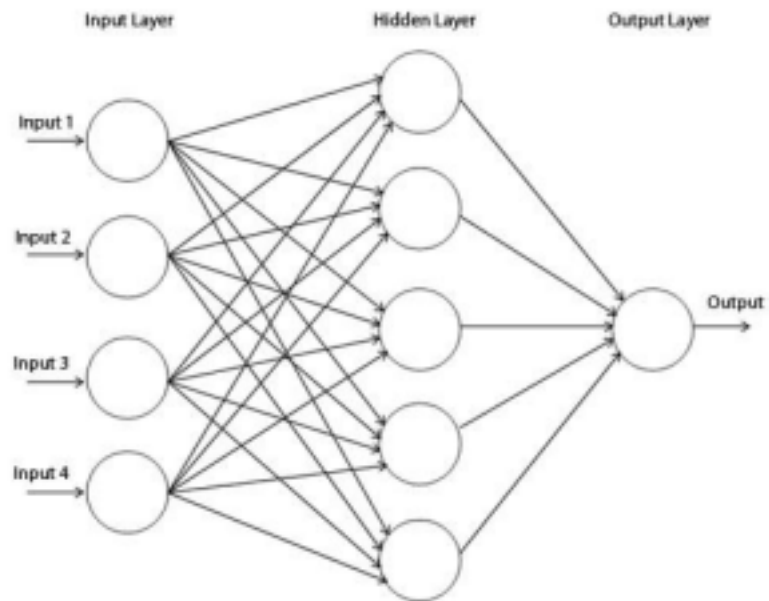
Multilayer perceptrons

When each node in each layer is a linear combination of **all inputs from the previous layer** then the network is called a multilayer perceptron (MLP)

Weights can be organized into matrices.

Forward pass computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$

$$\mathbf{h}^{(1)} = g(W^{(1)}\mathbf{h}^{(1)} + \mathbf{b}^{(1)})$$

$\mathbf{h}^{(1)}$ $\mathbf{h}^{(2)}$

$\mathbf{x}$

$f(\mathbf{x})$

Inputs

Outputs

$W^{(1)}$ $W^{(2)}$ $W^{(3)}$

Input Layer   Hidden Layers   Output Layer

Width

Depth
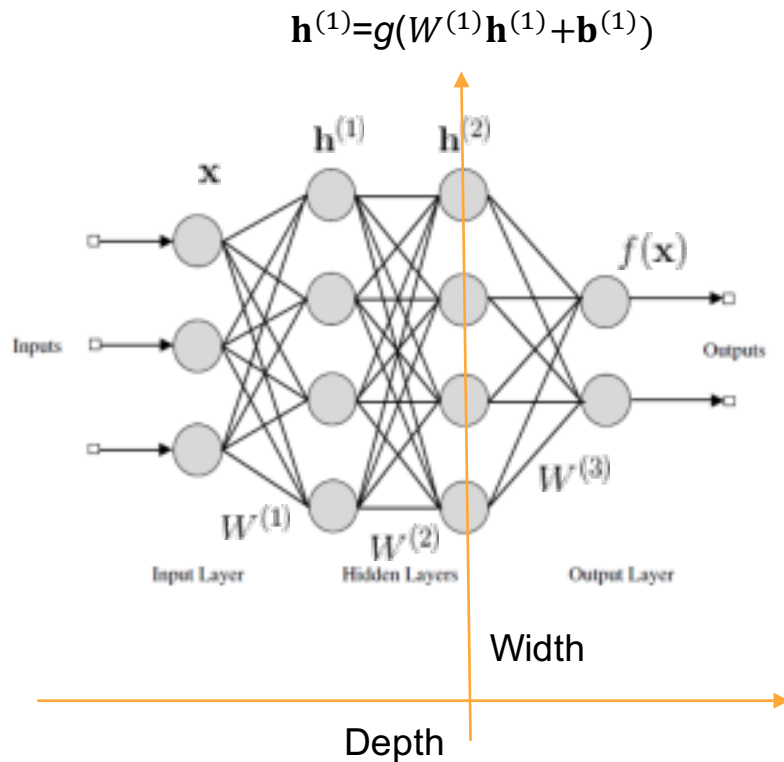
Activation functions

(AKA. transfer functions, nonlinearities, units)

Question:
   Why do we need these nonlinearities at all? Why not
      just make everything linear?

…..composition of linear transformations would be
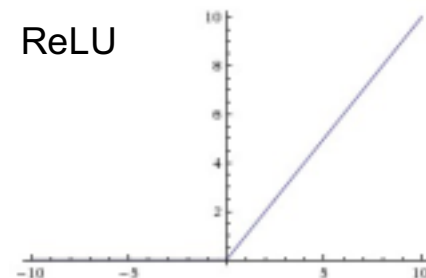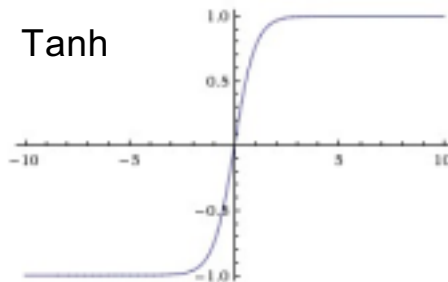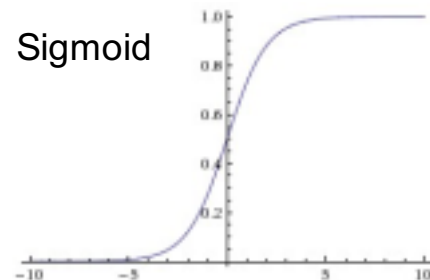equivalent to one linear transformation

Desirable properties
   Mostly smooth, continuous, differentiable

   Fairly linear

Common nonlinearities
   Sigmoid   Tanh   ReLU = max(0, x)   LeakyReLU

Sigmoid

Tanh

ReLU

Universal approximation theorem

[Universal approximation theorem](#) states that "the standard multilayer feed-forward network with **a single hidden layer**, which contains **finite number of hidden neurons**, is a **universal approximator** among continuous functions on compact subsets of R^n, under mild assumptions on the activation function."

If a 2 layer NN is a universal approximator, then why do we need deep nets??

The universal approximation theorem:

Says nothing about the how easy/difficult it is to fit such approximators

Needs a "finite number of hidden neurons": finite may be extremely large

In practice, deep nets can usually represent more complex functions with less total neurons (and therefore, less parameters)

…Learning
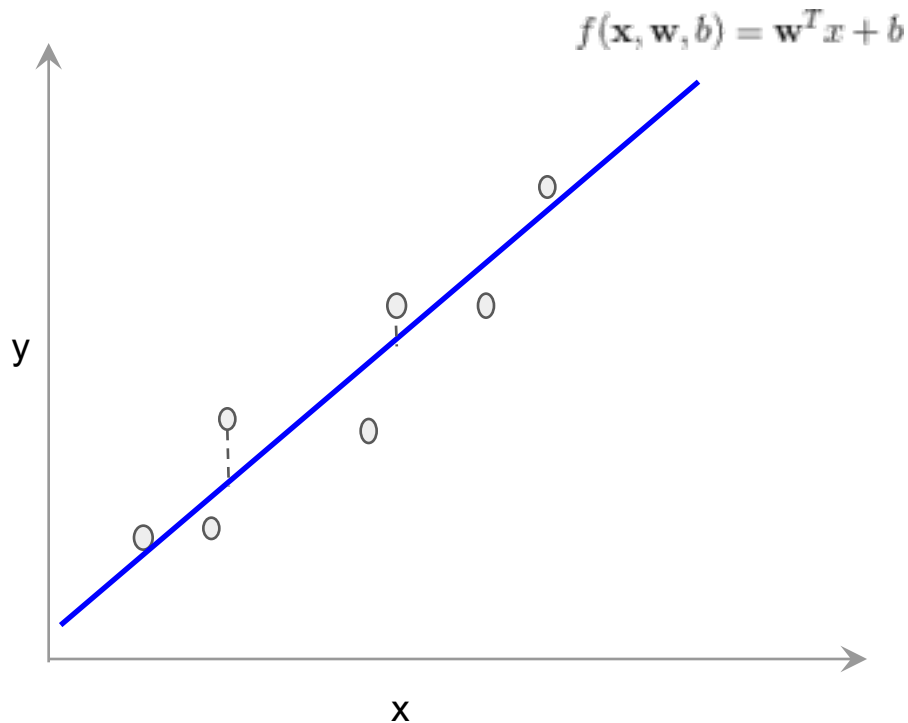
Linear regression – **Loss Function**

$$X = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times D} \qquad \mathbf{y} \in \mathbb{R}^N$$
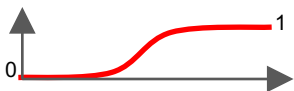
Loss function is square (Euclidean) loss

$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{w}, b))^2$$

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

y

x

Logistic regression

Activation function is the sigmoid

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

Loss function is cross entropy

$$L = -\frac{1}{N} \sum_{i=1}^{N} y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i))$$

$g(\mathbf{w}^T\mathbf{x} + b) > \frac{1}{2}$

$\mathbf{w}$

x_2

$g(\mathbf{w}^T\mathbf{x} + b) = \frac{1}{2}$
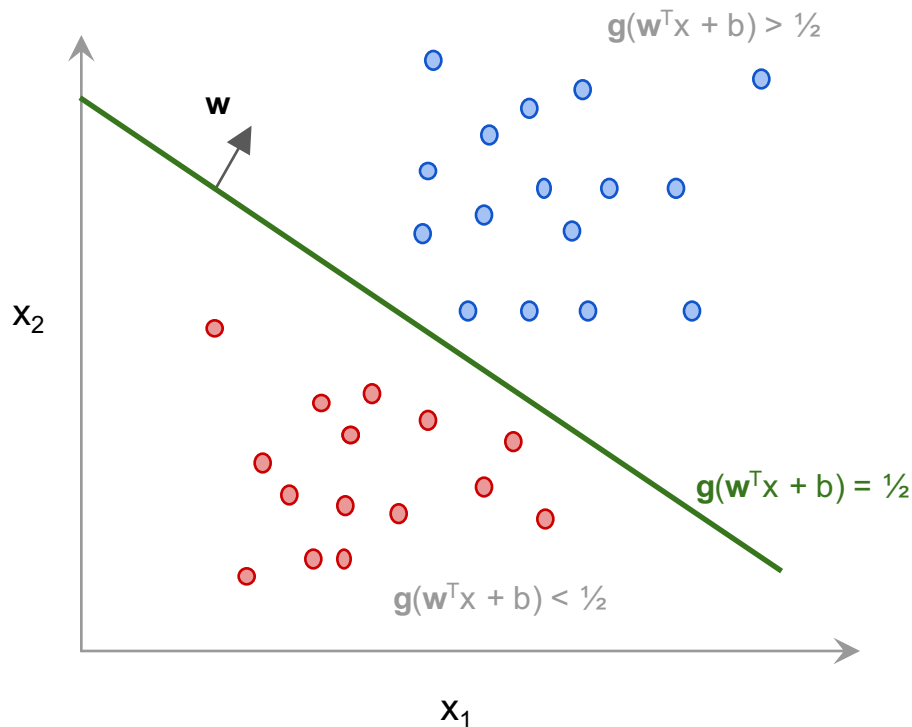
$g(\mathbf{w}^T\mathbf{x} + b) < \frac{1}{2}$

x_1

Fitting linear models

E.g. linear regression

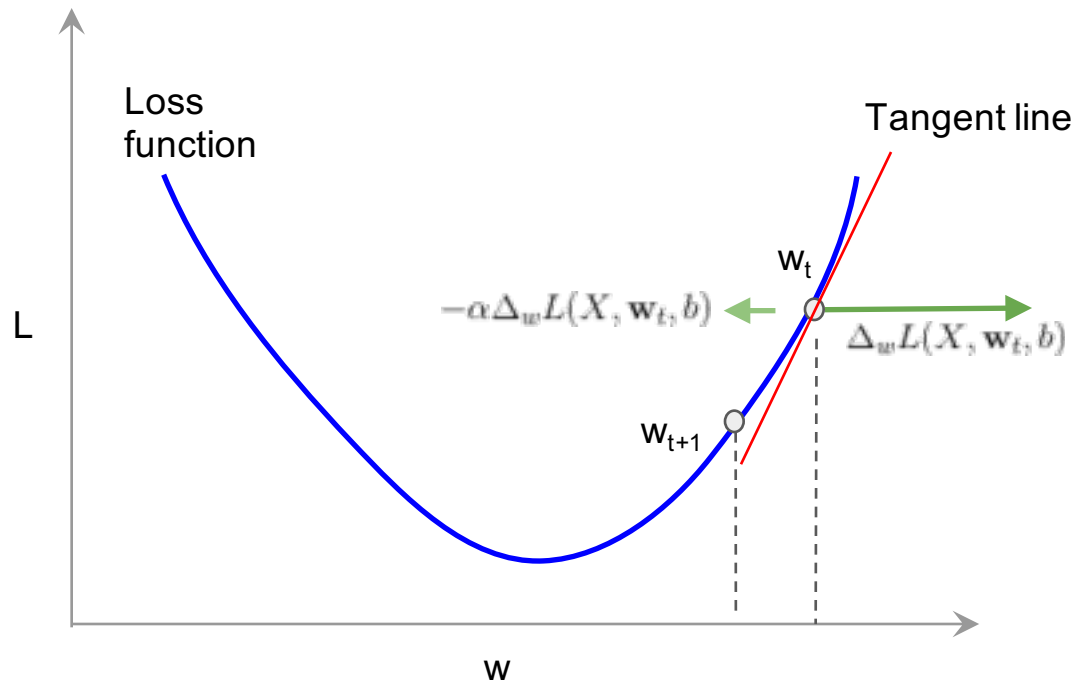$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{w}, b))^2$$

Need to **optimize** L

Gradient descent

$$\Delta_w L = \frac{1}{N} \sum_{i=1}^{N} (f(x_i) - y_i) x_i$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \Delta_w L(X, \mathbf{w}_t, b)$$

Loss function

Tangent line

L

$-\alpha\Delta_w L(X, \mathbf{w}_t, b)$

w_t

$\Delta_w L(X, \mathbf{w}_t, b)$

w_{t+1}

w

Choosing the learning rate

For **first order** optimization methods, we need to choose a learning rate (aka **step size**)

    Too large: overshoots local minimum, loss increases
    Too small: makes very slow progress, can get stuck
    Good learning rate: makes steady progress toward local minimum
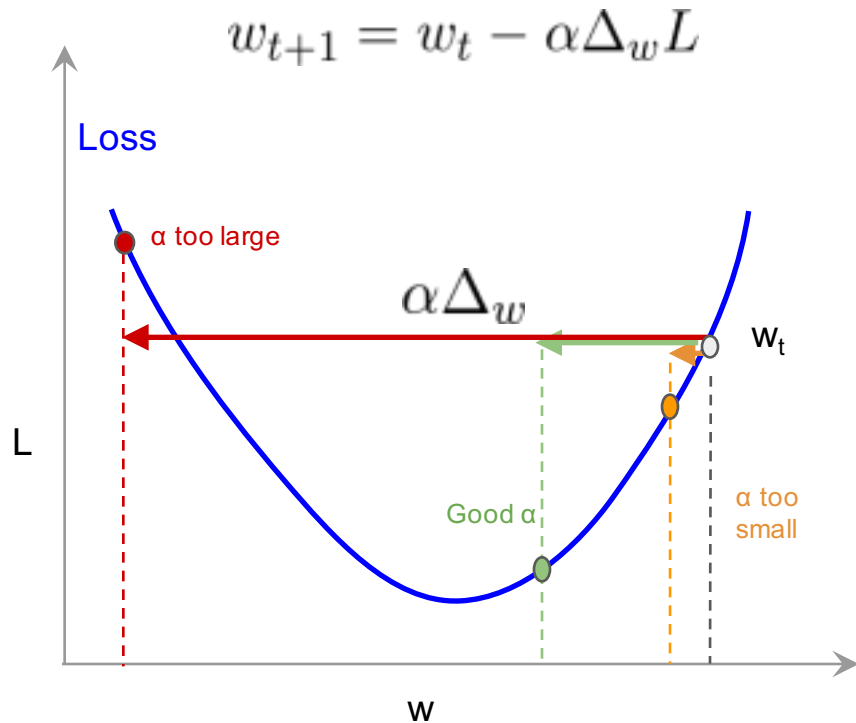
Usually want a higher learning rate at the start and a lower one later on.

Common strategy in practice:

    Start off with a high LR (like 0.1 - 0.001),
    Run for several **epochs** (1 – 10)
    Decrease LR by multiplying a constant factor (0.1 - 0.5)

$$w_{t+1} = w_t - \alpha \Delta_w L$$

Loss

α too large

$$\alpha \Delta_w$$

w_t

L

Good α

α too small

w

Training

Estimate parameters θ(W$^{(k)}$, b$^{(k)}$) from training examples

Given a Loss Function $W^* = argmin_\theta \mathcal{L}(f_\theta(x), y)$

In general no close form solutions:

• Iteratively adapt each parameter, numerical approximation

  Basic idea: **gradient descent.**

• Dependencies are very complex.

    Global minimum: challenging. Local minima: can be good enough.

• Initialization influences in the solutions.

Training

Gradient Descent: Move the parameter θ_j in small steps in the direction opposite sign of the derivative of the loss with respect j.

$$\theta^{(n)} = \theta^{(n-1)} - \alpha^{(n-1)} \cdot \nabla_\theta \mathcal{L}(y, f(x))$$

• Stochastic gradient descent (SGD): estimate the gradient with one sample, or better, with a **minibatch** of examples.

• **Momentum:** the movement direction of parameters averages the gradient estimation with previous ones.

Several strategies have been proposed to update the weights: Adam, RMSProp, Adamax, etc. known as: **optimizers**

Training and monitoring progress

1. Split data into train, validation, and test sets
   Keep 10-30% of data for validation

2. Fit model parameters on train set using SGD

3. After each epoch:
   Test model on validation set and compute loss
   Also compute whatever **other metrics** you are interested in
   Save a snapshot of the model

4. Plot **learning curves** as training progresses

5. Stop when validation loss starts to increase

6. Use model with minimum validation loss
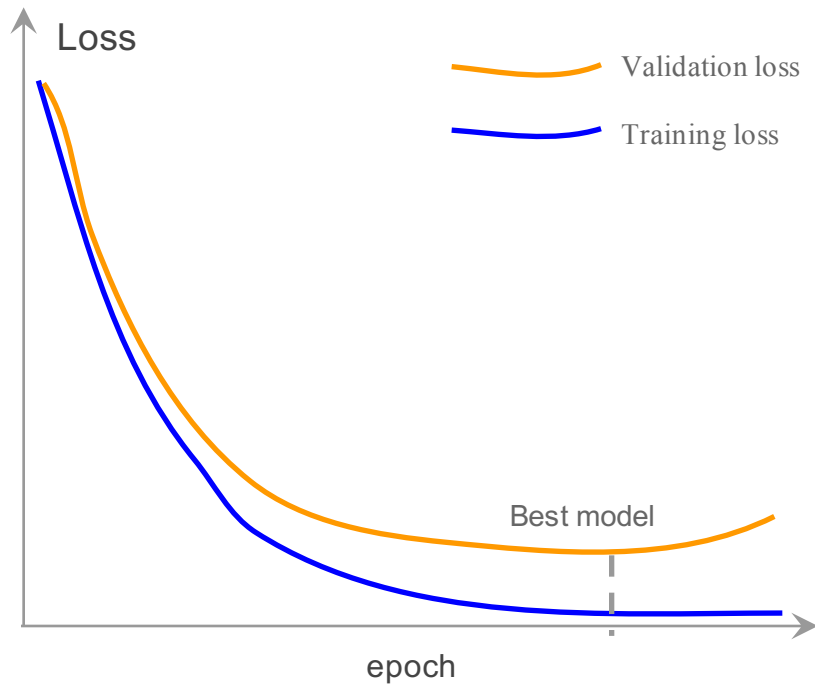
Gradient descent examples

Linear regression

http://nbviewer.jupyter.org/github/kevinmcguinness/ml-examples/blob/master/notebooks/GD_Regression.ipynb

https://github.com/kevinmcguinness/ml-examples/blob/master/notebooks/GD_Regression.ipynb

Logistic regression

http://nbviewer.jupyter.org/github/kevinmcguinness/ml-examples/blob/master/notebooks/GD_Classification.ipynb

https://github.com/kevinmcguinness/ml-examples/blob/master/notebooks/GD_Classification.ipynb

MNIST Example

Handwritten digits

- 60.000 examples

- 10.000 test examples

- 10 classes (digits 0-9)

- 28x28 grayscale images(784 pixels)
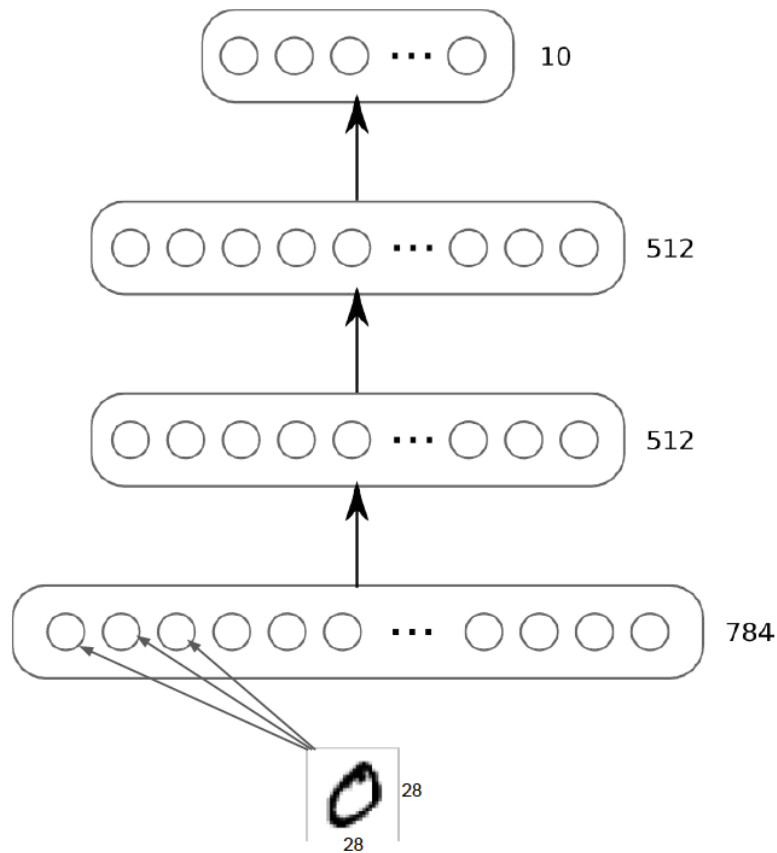
- http://yann.lecun.com/exdb/mnist/

The objective is to learn a function that predicts the digit from the image

MNIST Example

Model

- 3 layer neural-network (2 hidden layers)

- Tanh units (activation function)

- 512-512-10

- Softmax on top layer

- Cross entropy Loss

Layer	#Weights	#Biases	Total
1	784 x 512	512	401,920
2	512 x 512	512	262,656
3	512 x 10	10	5,130
			669,706

MNIST Example

Training

- 40 epochs using min-batch SGD
- Batch Size: 128
- Leaning Rate: 0.1 (fixed)
- Takes 5 minutes to train on GPU

Metrics

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

there are other metrics….

Accuracy Results

- 98.12% (188 errors in 10.000 test examples)

there are ways to improve accuracy…

Training MLPs

With Multiple layers we need to find the gradient of the loss function with respect to all the parameters of the model ($W^{(k)}$, $b^{(k)}$)

These can be found using the **chain rule** of differentiation.

The calculations reveal that the gradient wrt. the parameters in layer k only depends on the error from the above layer and the output from the layer below.

This means that the gradients for each layer can be computed iteratively, starting at the last layer and propagating the error back through the network. This is known as the **backpropagation** algorithm.