UniGe | MaLGa

[Deep] Neural Networks

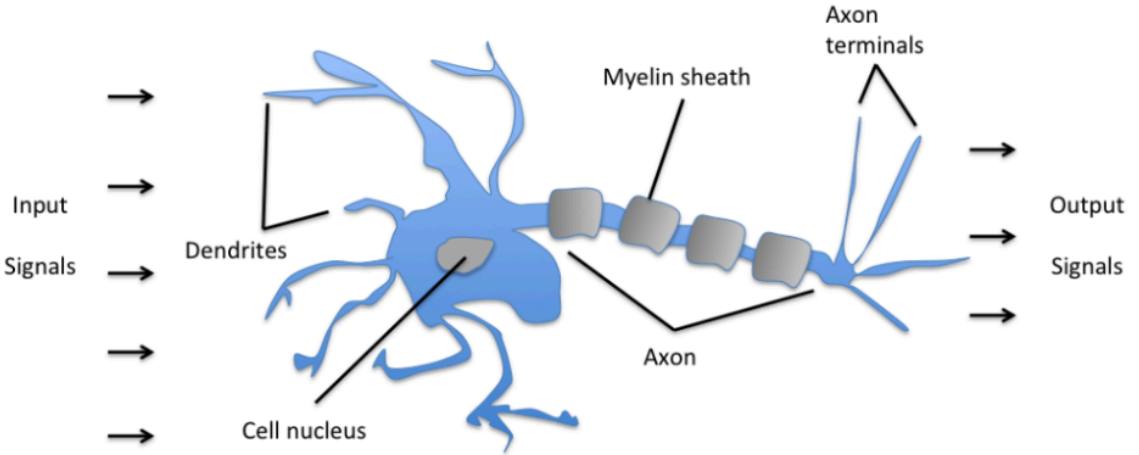
Deep Learning: A hands-on introduction

A course offered in the PhD program in Computer Science and Systems Engineering

Nicoletta Noceti

Machine Learning Genoa Center – University of Genoa

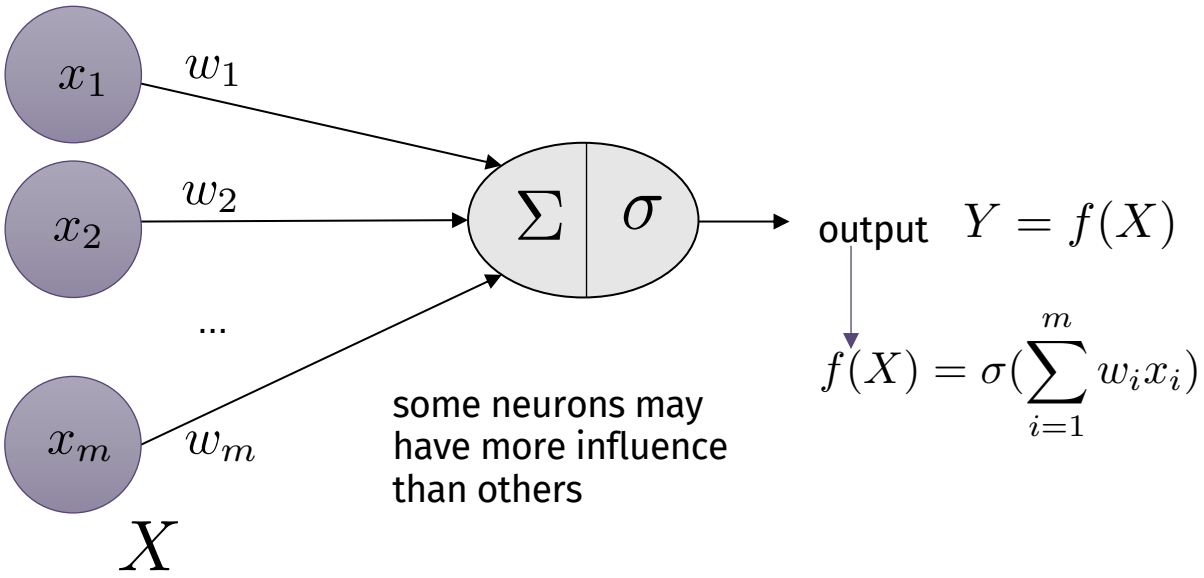
Biological neuron



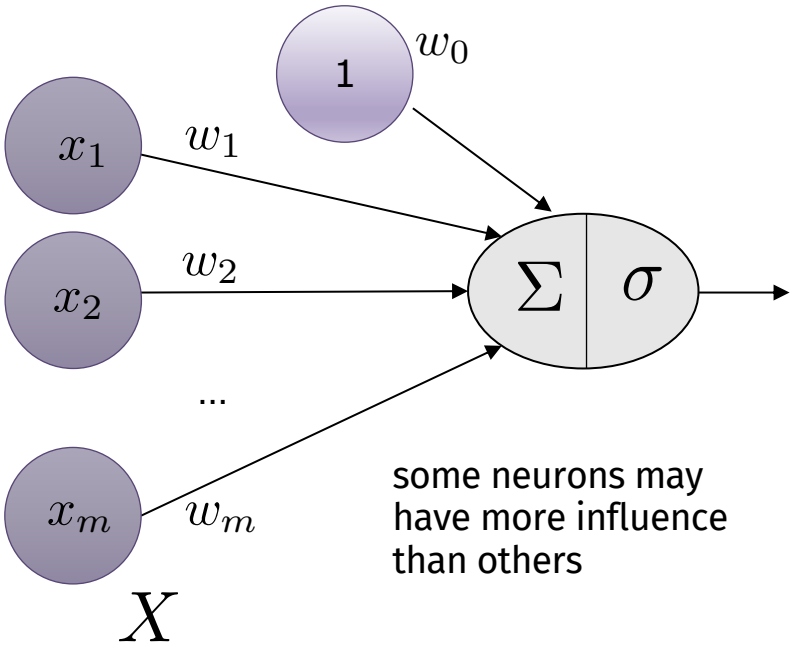
UniGe | MaLGa

Single Layer Perceptron

Single layer perceptron



Single layer perceptron



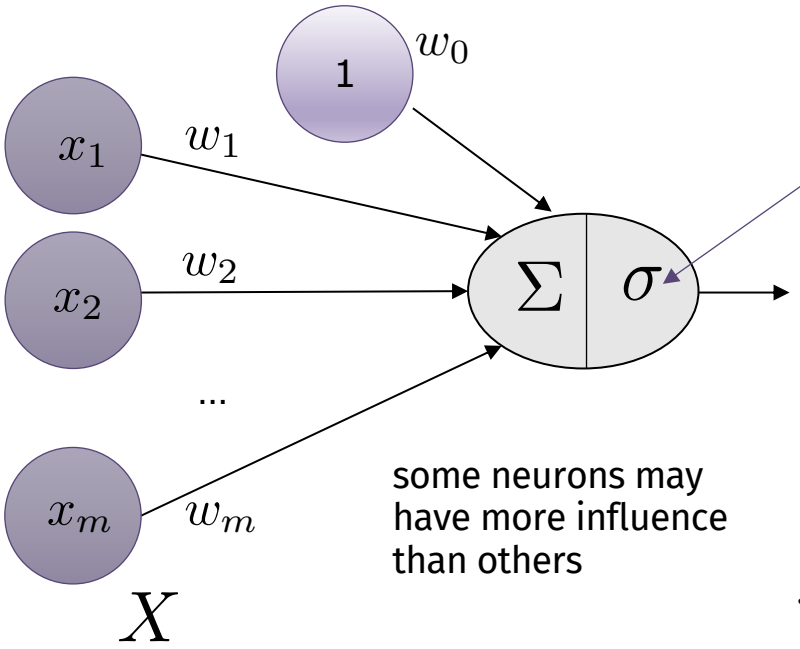
output $Y = f(X)$

some neurons may have more influence than others

$$f(X) = \sigma(w_0 + \sum_{i=1}^{m} w_i x_i)$$

$$f(X) = \sigma(w_0 + X^T W)$$

Single layer perceptron



Activation functions allow to account for non-linearity in our model

output $Y = f(X)$

some neurons may have more influence than others

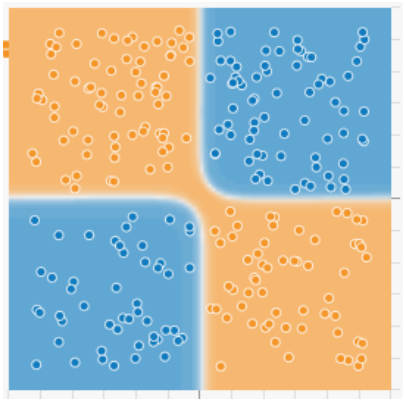
$$f(X) = \sigma(w_0 + \sum_{i=1}^{m} w_i x_i)$$

$$f(X) = \sigma(w_0 + X^T W)$$

Why activation functions are so important

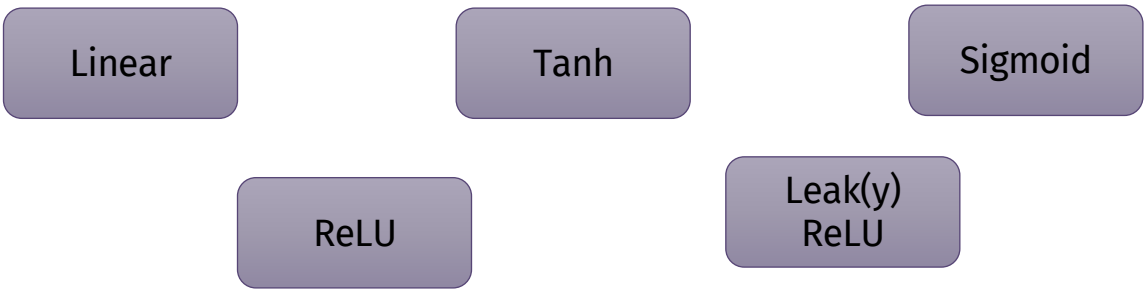
They introduce non-linearities into the network

https://playground.tensorflow.org



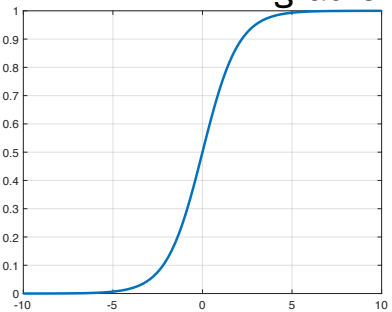
Activation functions

- **Nonlinear activation** is key to achieving good function approximation
- It takes a single number and maps it to a different numerical value
- Popular functions



Sigmoid

Near-0 gradient

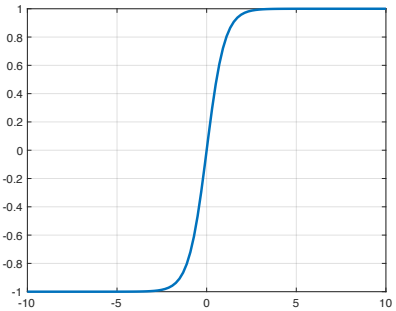


Near-0 gradient

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

- Historically the most used for binary classification
- It corresponds in fact to the well-known logistic regression
- It suffers from the vanishing gradient problem
- Non-zero centered output that may cause zig-zagging

Tanh

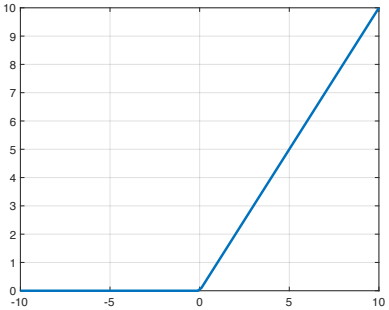


$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- It suffers from the vanishing gradient problem
- Output is zero centered, thus it has better gradient properties than sigmoid
- It is a scaled version of *Sigmoid*:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

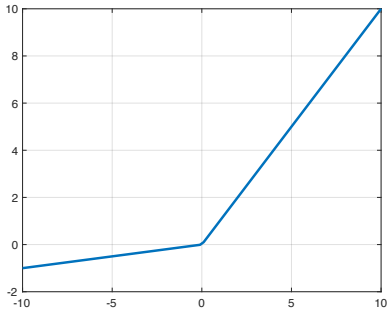
ReLU (Rectified Linear Unit)



$$f(x) = max(0, x)$$

- Very popular and simple: it thresholds values below 0
- It allows for fast convergence of the optimization function
- The weight may irreversibly die

Leaky ReLU



- It is aimed to fix the dying ReLU problem
- In a variant (called parametric ReLU) the slope for negative values can be learnt

$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$\alpha = 0.1$$

UniGe | MaLGa

Multi-Layer Perceptron

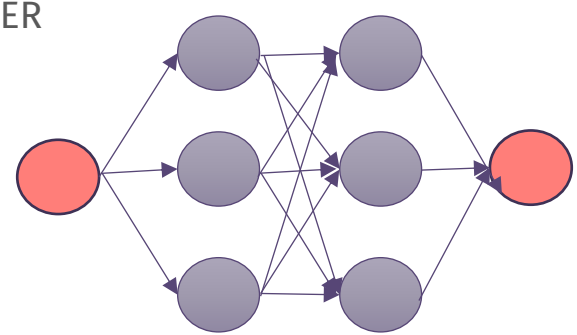
MultiLayer perceptrons

- Mid 1980: the need for architecture to improve the model has been recognized

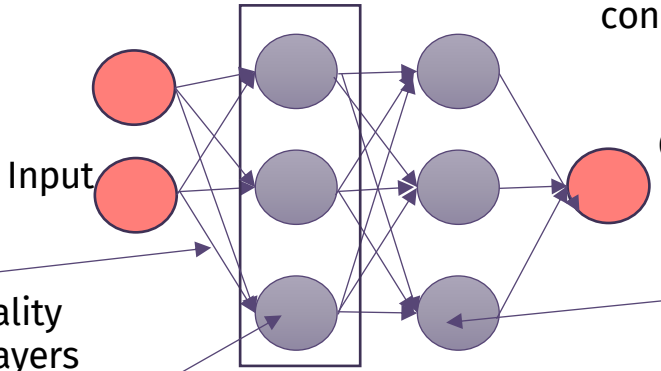
- The **deep feedforward networks** (aka **multilayer perceptrons MLPs**) are represented by composing together many different functions

MultiLayer perceptrons

- The are composed as chains of layers that may be of three different types:
  - INPUT LAYER
  - (MULTIPLE) HIDDEN LAYER(S)
  - OUTPUT LAYER



MultiLayer perceptrons



The dimensionality of the hidden layers determine the **width**

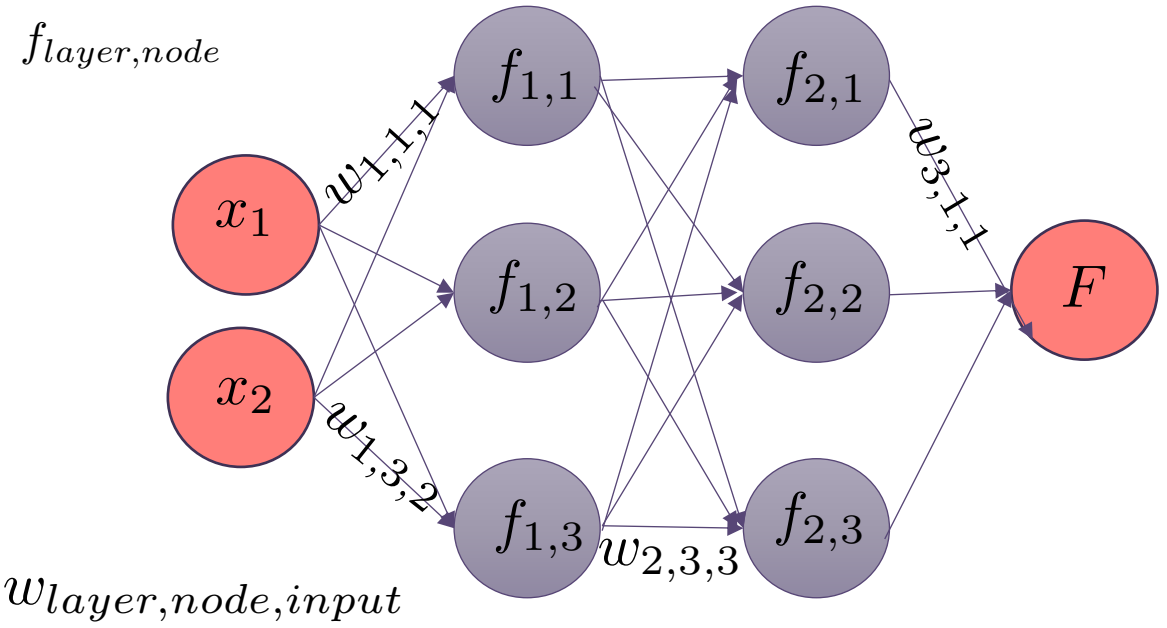
Each element of the layer plays the role of a **neuron**

Each hidden unit is associated with an activation function

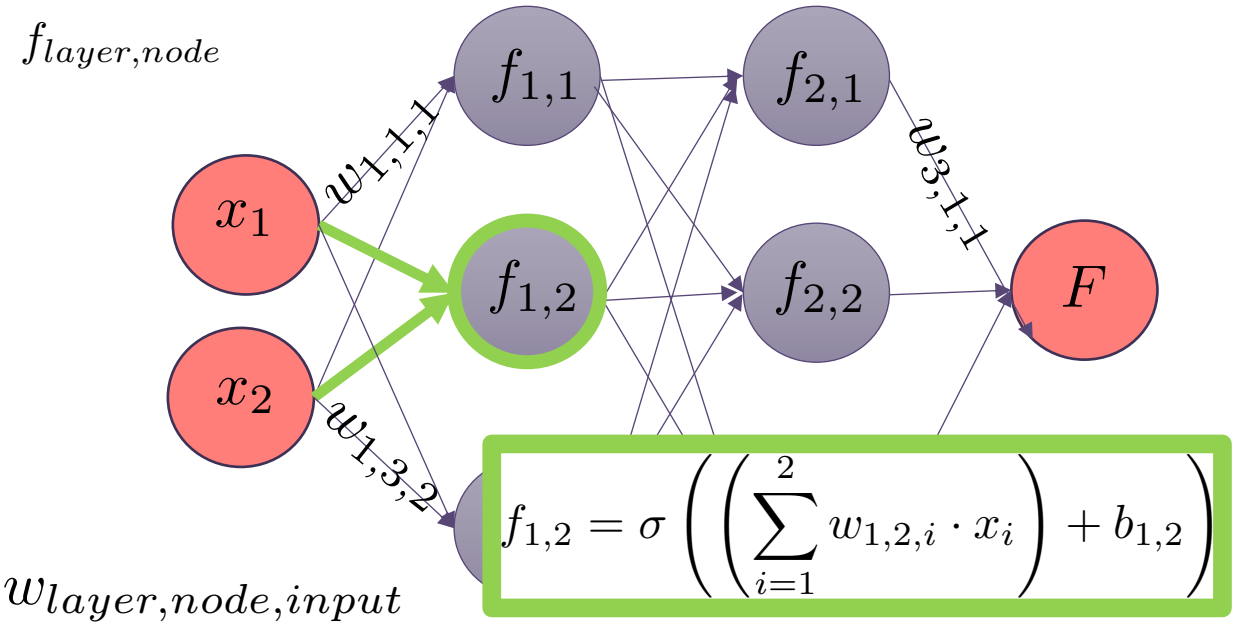
Depth (or number of layers)

Hidden: the intermediate output is not explicitly shown

Forward propagation with an example



Forward propagation with an example



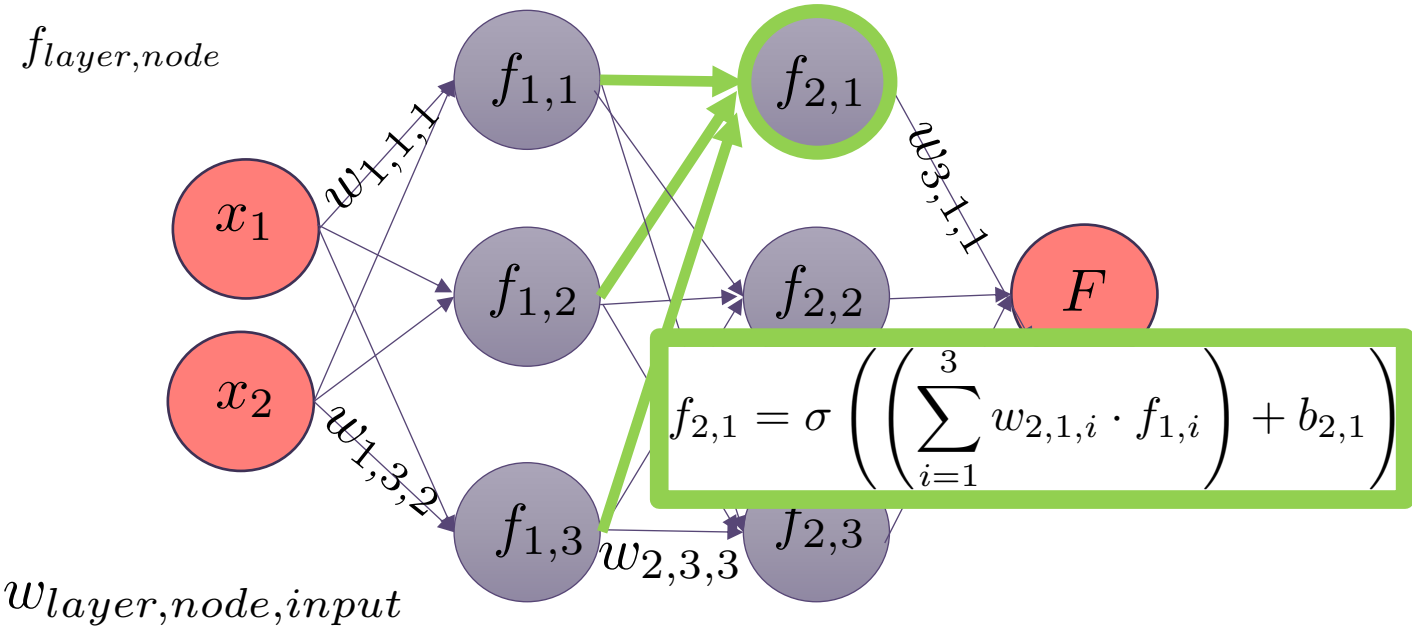
Forward propagation with an example

$$f_{1,2} = \sigma\left(\left(\sum_{i=1}^{2} w_{1,2,i} \cdot x_i\right) + b_{1,2}\right)$$

$$f_{1,n} = \sigma\left(\left(\sum_{i=1}^{2} w_{1,n,i} \cdot x_i\right) + b_{1,n}\right)$$

$$n = 1 \ldots 3$$

Forward propagation with an example



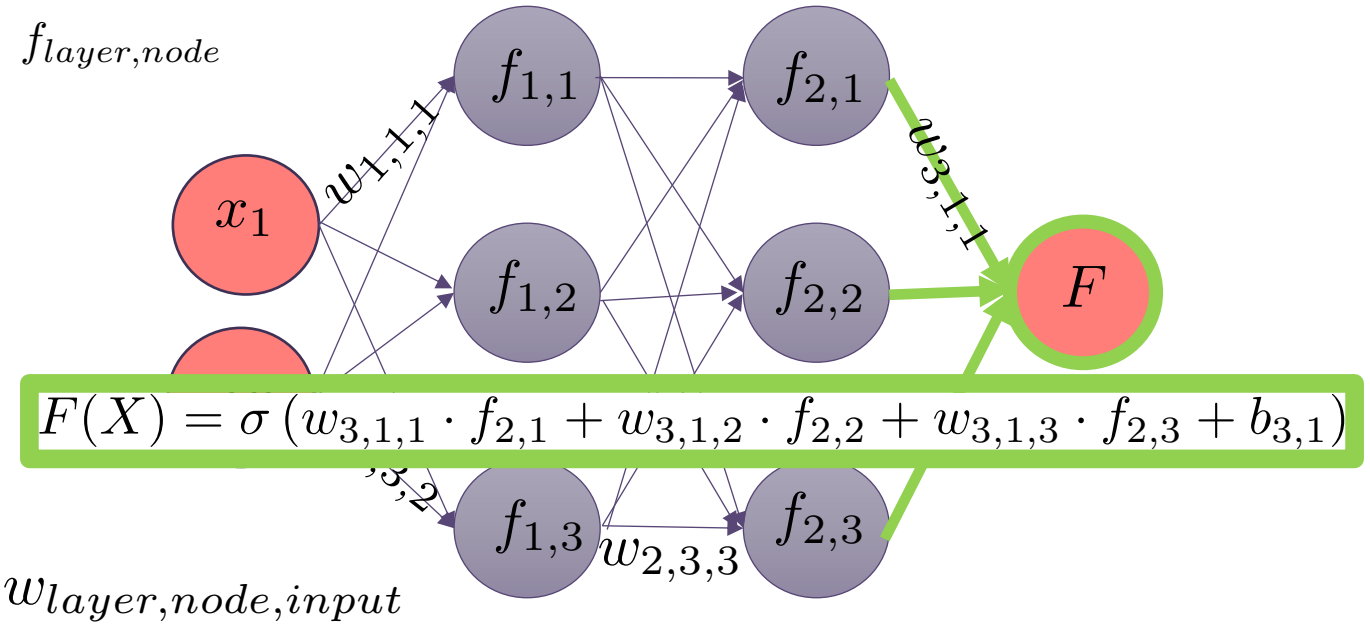
Forward propagation with an example

$$f_{2,1} = \sigma\left(\left(\sum_{i=1}^{3} w_{2,1,i} \cdot f_{1,i}\right) + b_{2,1}\right)$$

$$f_{2,n} = \sigma\left(\left(\sum_{i=1}^{3} w_{2,n,i} \cdot f_{1,i}\right) + b_{2,n}\right)$$

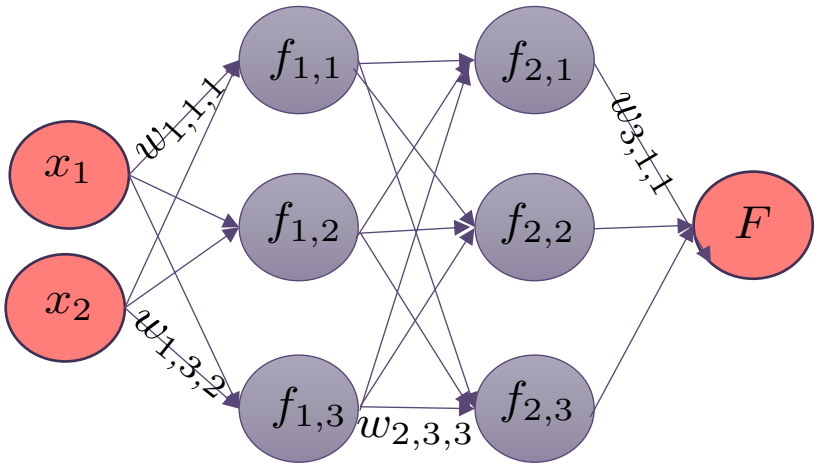
$$n = 1 \ldots 3$$

Forward propagation with an example



How many parameters?

$$F(X) = \sigma\left(w_{3,1,1} \cdot f_{2,1} + w_{3,1,2} \cdot f_{2,2} + w_{3,1,3} \cdot f_{2,3} + b_{3,1}\right)$$



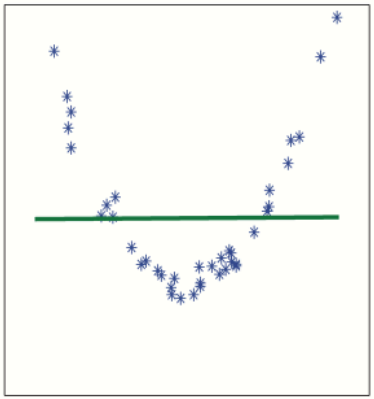
6 + 9 + 3 weights
7 bias terms

25 unknowns

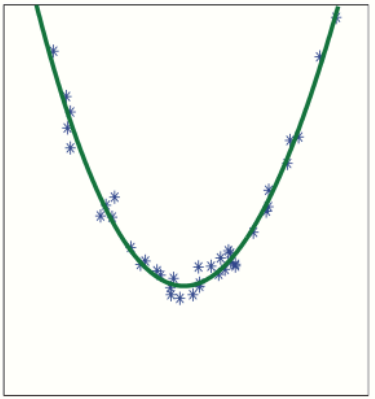
Towards DNN: model capacity

- The universal approximation theorem (Hornik 1991) tells that a infinitely wide single hidden layer can represent any function
- **Infeasible in practice, and the resulting model may not generalize**
- Model capacity is related to **bias-variance**
  - Low capacity: high bias, low variance
  - High capacity: low bias, high variance
- High capacity models may tend to overfit

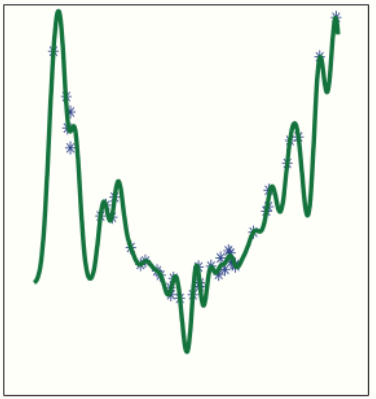
A parenthesis on under/overfitting and bias-variance



The algorithm underfits: it's stable but disregards the data → High bias



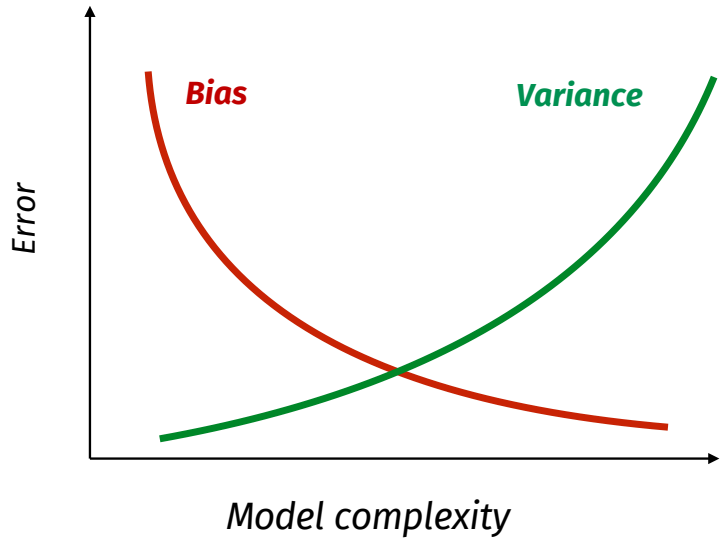
Ideal fitting



The algorithm overfits: it fits the data but is unstable → High variance

From http://lcsl.mit.edu/courses/mlcc/mlcc2019/slides/MLCC_SLT.pdf

A parenthesis on under/overfitting and bias-variance



Training a DNN

- Training a DNN means learning the values for the model parameters (weights, bias terms) from a training set

- An essential element of training is the **loss function**, that estimates how much we loose predicting F(X) if place of the real output y

$$\mathcal{L} : Y \times Y \rightarrow [0, \inf)$$

where Y is the space of the output of F

$$F : X \rightarrow Y$$

Training a DNN

$$W^* = \arg\min_W \frac{1}{n} \sum_{i=1}^{N} \mathcal{L}(F(X^{(i)}; W), y^{(i)})$$

$$W^* = \arg\min_W J(W)$$

We need to compute the gradient of the function **J**

Optimization: Gradient descent

An iterative procedure for estimating the solution of a functional, in our case the minimization of J(W)

- Initialize the weights W^0
- Until convergence $||W^{t+1} - W^t|| < toll$
  - Compute the gradient $\nabla_W J(W^t)$
  - Update the weights $W^{t+1} = W^t - \alpha \nabla_W J(W^t)$

Gradient descent

- We need efficient and reliable algorithms for gradient descent since in NNs we usually deal with millions of parameters

- More information on gradient descent optimization algorithms can be found here: http://ruder.io/optimizing-gradient-descent/

Gradient descent: variation on the theme

- **Batch GD**: it requires the computation of the gradient after the evaluation of the loss function on the entire training set

$$W^{t+1} = W^t - \alpha \nabla_W J_X(W^t)$$

Stable but computationally unfeasible most of the times

- **Stichastic GD**: a step is made after the evaluation of each sample

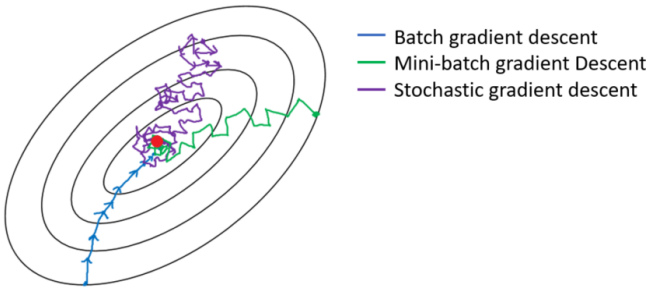
$$W^{t+1} = W^t - \alpha \nabla_W J_{X^{(i)}}(W^t)$$

It converges well in practice, providing a better exploration of the loss function space, but it's very noisy (stochastic...)

A compromise: using mini-batches

- Select a portion of the training set of size s (the mini-batch size) and update the weights after evaluating the loss function on it

$$W^{t+1} = W^t - \alpha \nabla_W J_{X(i \ldots i+s)}(W^t)$$



Picture from https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Gradient Descent algorithms

- Training a Deep network may be very complex!
- It's useful in general to play with the learning rate and adapt it to the problem at hand
- This gives rise to a number or GD algorithms: SGD, ADAM, ADADELTA, ADAGRAD, RMSProp... all available in Keras (and you'll play with them in the hands-on activity)

Gradient computation: backpropagation

- The key for gradient descent is the computation of the gradient
- To this purpose the backpropagation algorithm (1960s) is employed, in essence
  - with forward prop the input flows into the network and produces a scalar cost
  - backprop allows the info to flow back into the net to compute the gradient
- Backpropagation **makes efficient and effective use of the chain rule of calculus** (and is commonly done by deep learning platforms)

The chain rule

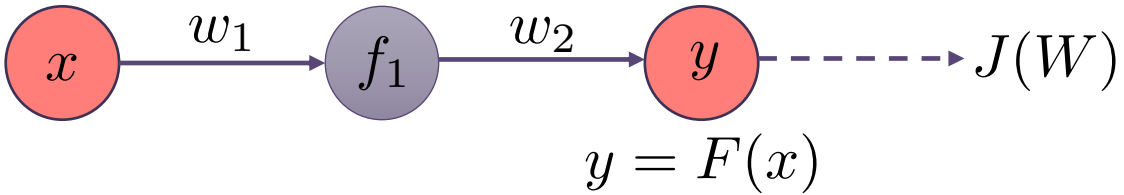
- For arbitrarily long composite functions

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{du}{dx}$$

- Also called reverse model, since we start from the outer function (from right to left)

- The chain rule is the essence of DNN training, as it allows to estimate the gradient for high dimensional spaces from the partial derivatives with respect to each weight

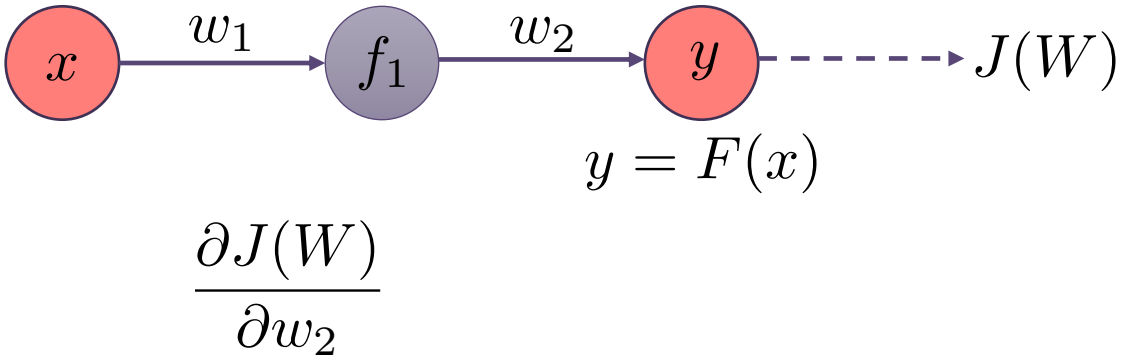
Backpropagation with an example



How does a small change in a weight affect the loss J(W)?

From http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf

Backpropagation with an example



From http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf

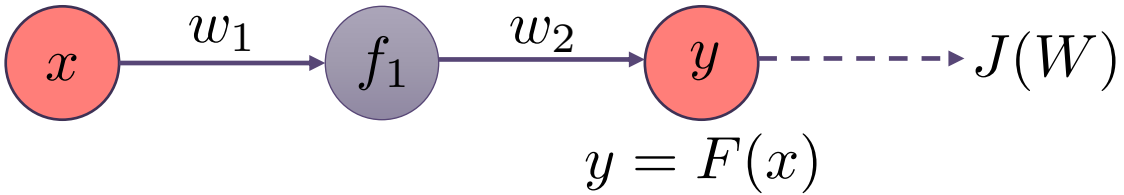
Backpropagation with an example

$$y = F(x)$$

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial y} \cdot \frac{\partial y}{\partial w_2}$$

From http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf

Backpropagation with an example



$$\frac{\partial J(W)}{\partial w_1}$$

From http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf

Backpropagation with an example

$$y = F(x)$$

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial y} \cdot \frac{\partial y}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1}$$

From http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf

Regularization

- DNN models have a huge number of parameters, hard to be trained even if on very large datasets
- A large number of parameters leads to high chance of overfitting with models that do not generalize and have poor response to input noise
- To prevent over-fitting, **regularization** can be used, to stabilize the learning of weights
- A common choice: adding a regularization term to the functional

$$J(W) + \lambda\Omega(W)$$

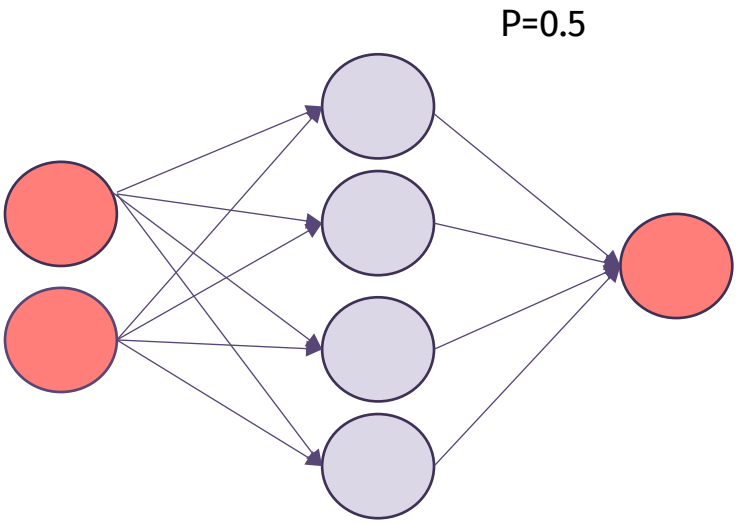
Dropout

- Overfitted DNN models tend to suffer from a problem of co-adaptation: models weights are adjusted co-linearly to learn the model training data too well... so the model doesn't generalize

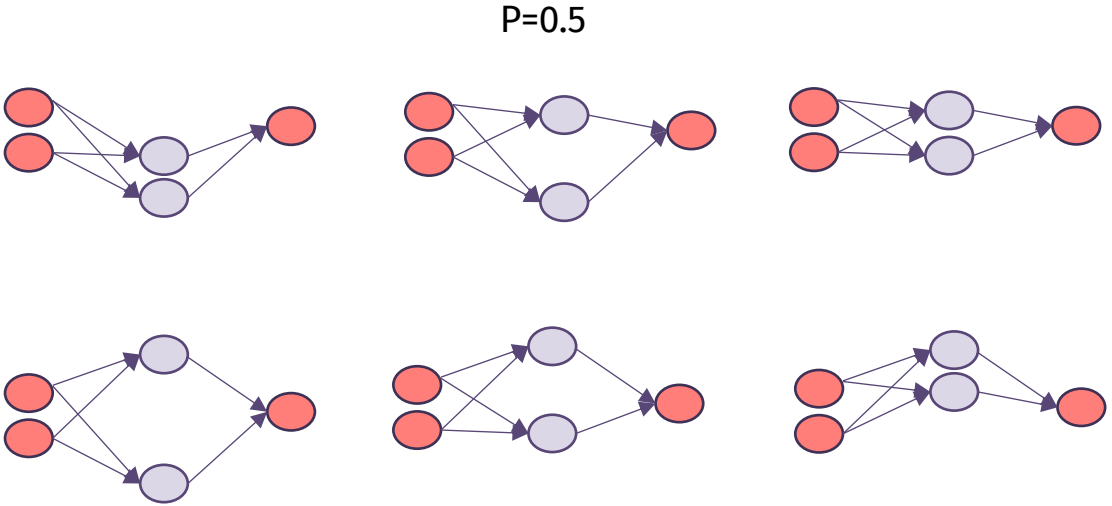
- Especially true with limited training data

- Dropout is a way to break this co-adaptation: at each training step some fraction of weights are dropped-out of each layer

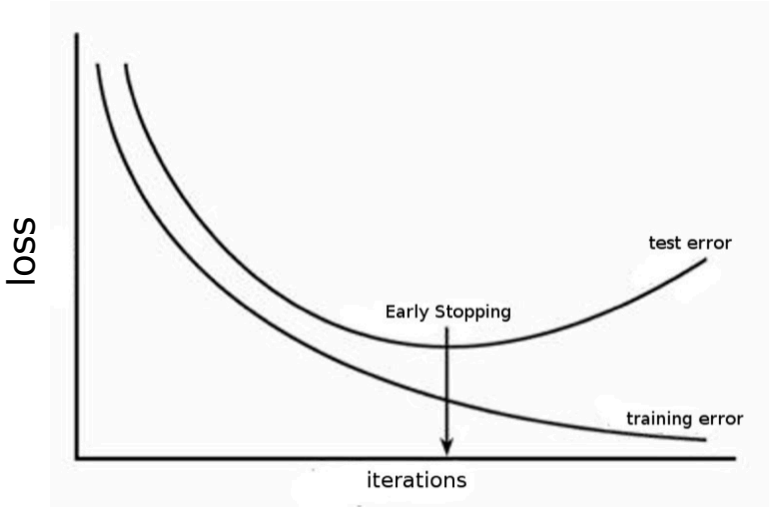
Dropout (intuition)



Dropout (intuition)



Early stopping



– From http://lcsl.mit.edu/courses/regml/regml2020/slides/lect3.pdf

Epochs

- As the number of epochs increases, the learnt curve goes from underfitting to optimal, to overfitting

- No easy to find an optimal number of epochs, but it is related to the complexity of the problem

Some terminology

- One **epoch** is when the entire dataset is passed forward and backward through the neural network only once (multiple times are usually needed)
- The **batch** size is the number of training examples in a mini-batch
- An **iteration** is the number of batches needed to complete one epoch
- Ex. For a dataset of 10000 sample with mini-batch size 1000, 10 iterations will complete 1 epoch

UniGe