UniGe | MaLGa

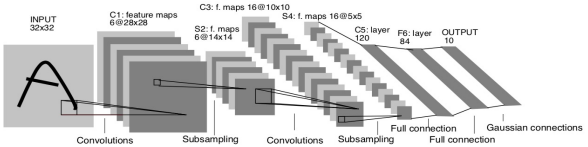
Neural Networks

Machine Learning

Computer Science Master Degree

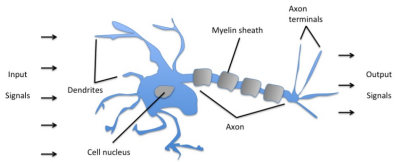
Nicoletta Noceti
Nicoletta.noceti@unige.it

Some definitions



- **Deep learning** is a family of machine learning methods based on artificial neural networks (ANNs) that use multiple layers to **progressively extract higher level features from raw input**

- **ANNs** are computing systems inspired by the **biological neural networks**, based on a collection of units/nodes, the artificial neurons, loosely modelling the neurons. a biological brain

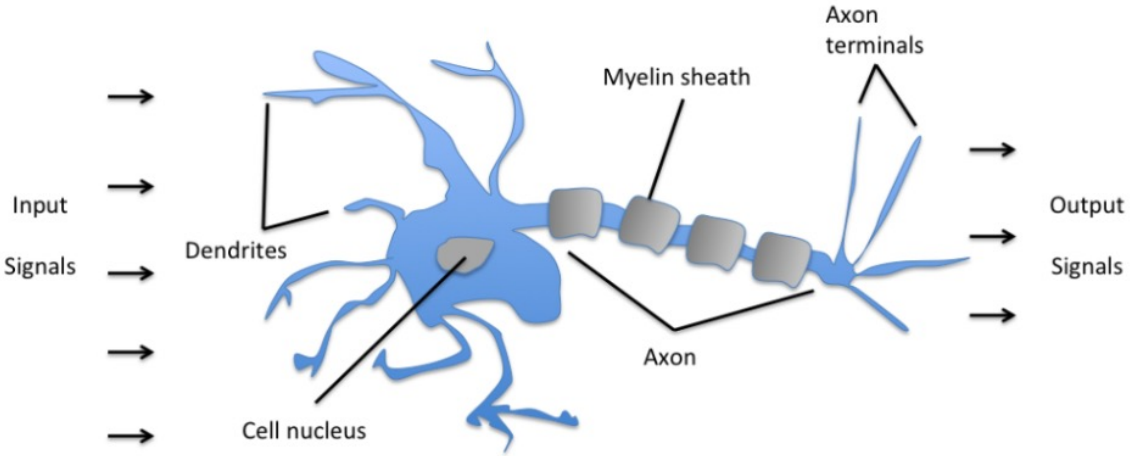


- <u>But also</u>: Deep learning methods are representation-learning methods with multiple levels of representation

UniGe | MaLGa

Single Layer Perceptron

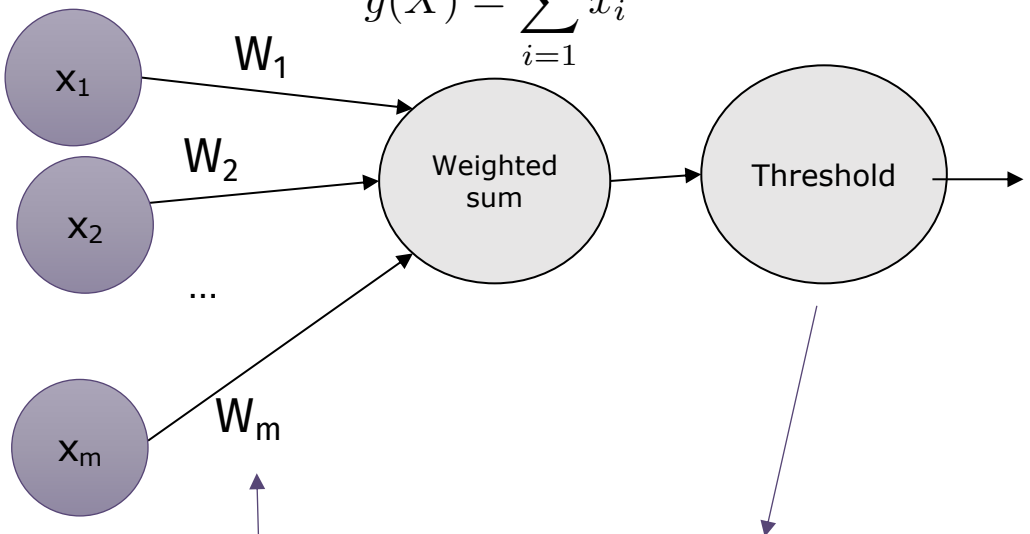
Biological neuron



McCulloch & Pitts Neuron Model (1943)

Described by Theta. If the sum of the inputs is larger than it the neuron fires

$$g(X) = \sum_{i=1}^{m} x_i$$



Y=1 indicates the neuron fires, Y=0 indicates it is at rest

$$f(X) = \sigma(g(X)) = \begin{cases} 1 & \text{if } g(X) > \Theta \\ 0 & \text{otherwise} \end{cases}$$

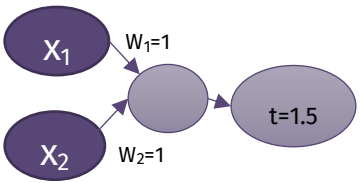
All the weights are set to unity

The activation threshold

McCulloch & Pitts Neuron Model (1943)

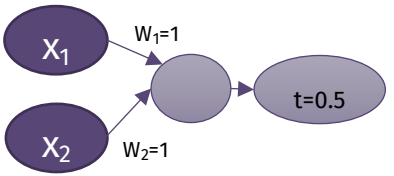
AND

X_1	X_2	Out
0	0	0
0	1	0
1	0	0
1	1	1



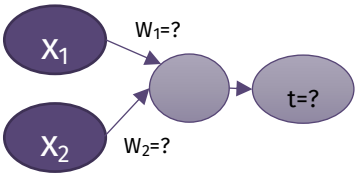
OR

X_1	X_2	Out
0	0	0
0	1	1
1	0	1
1	1	1



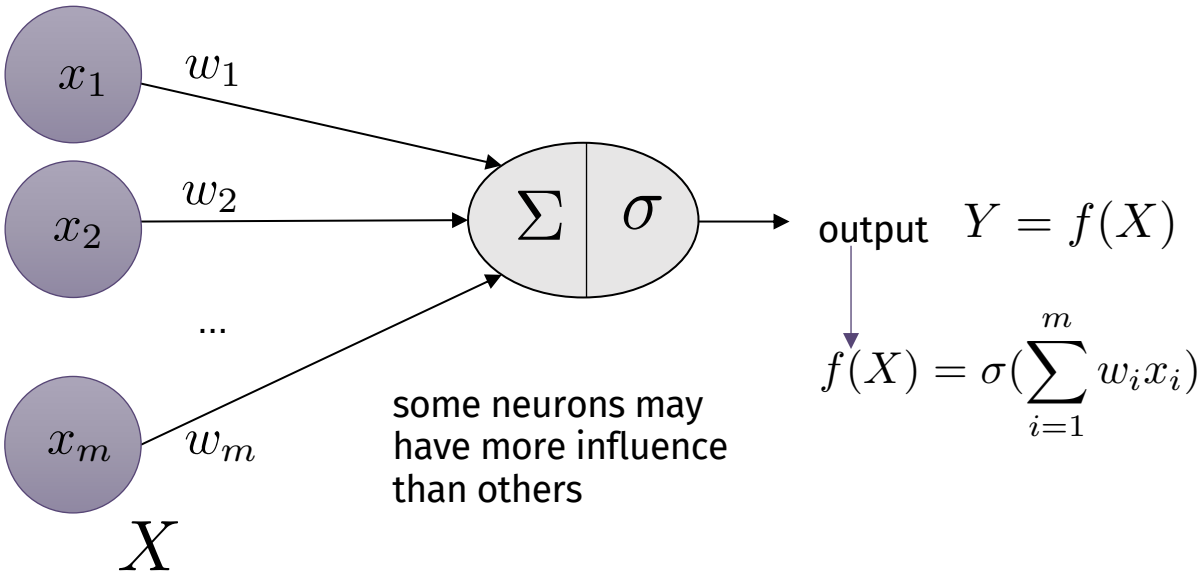
XOR

X_1	X_2	Out
0	0	0
0	1	1
1	0	1
1	1	0

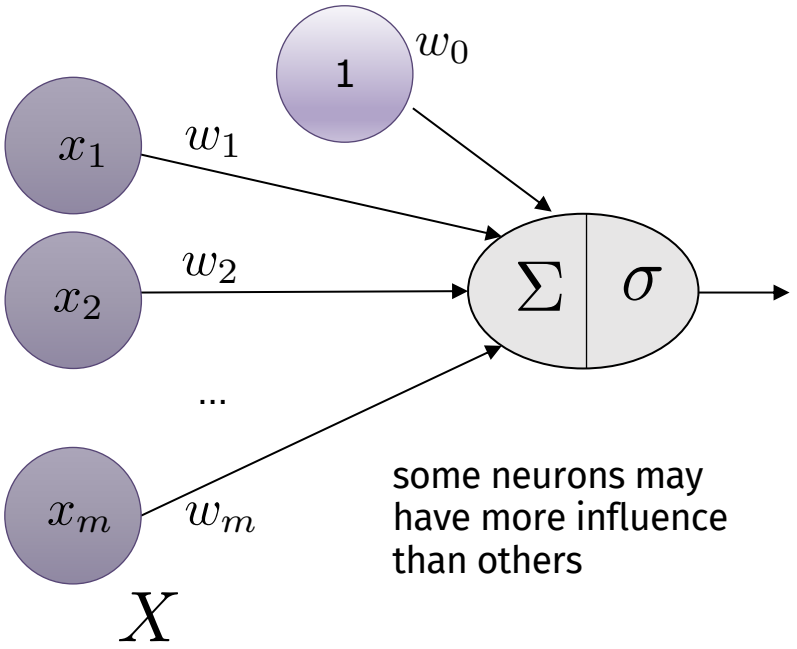


It doesn't work...the neuron cannot handle nonlinearities

Single layer perceptron



Single layer perceptron



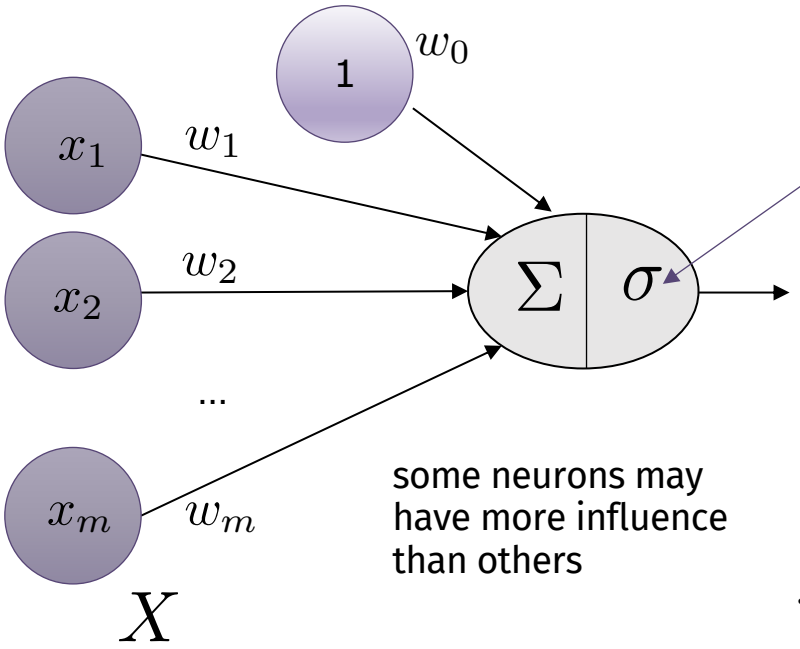
output $Y = f(X)$

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 $Y = f(X)$

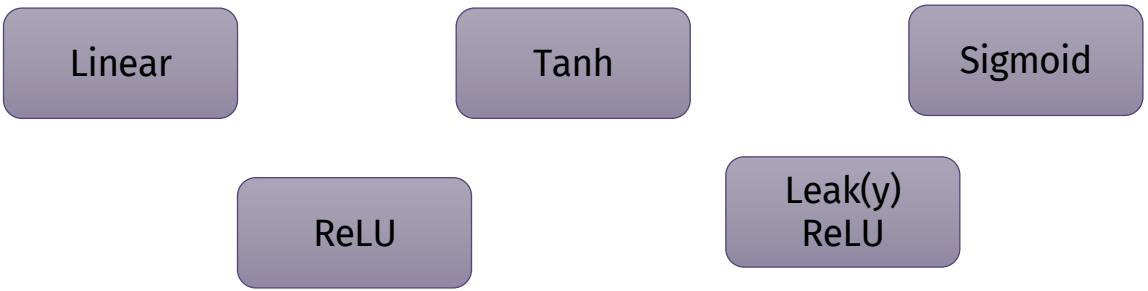
some neurons may have more influence than others

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

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

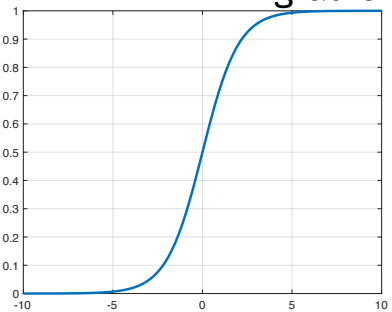
Activation functions

- **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

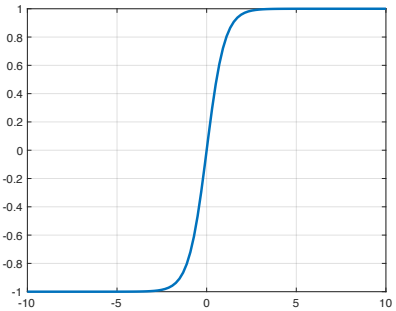


Near-0 gradient

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

- 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

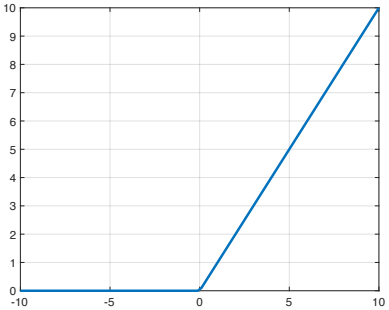


$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- 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*:

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

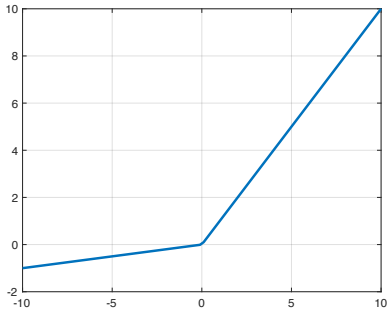
ReLU (Rectified Linear Unit)



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

- 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

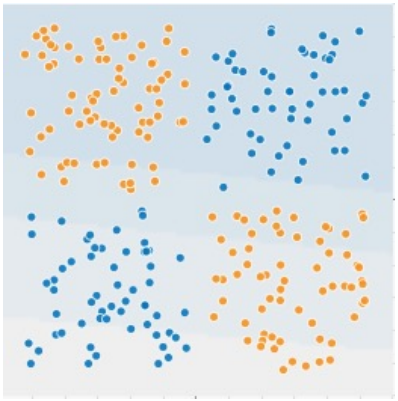
$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

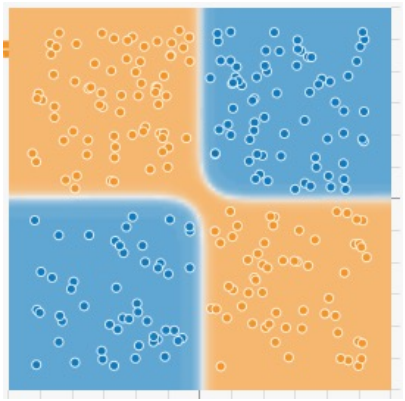
$$\alpha = 0.1$$

Why activation functions are so important

They introduce non-linearities into the network

https://playground.tensorflow.org





UniGe | MaLGa

Multi-Layer Perceptron

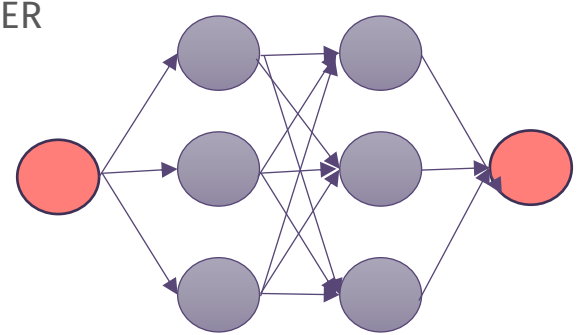
MultiLayer perceptrons

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

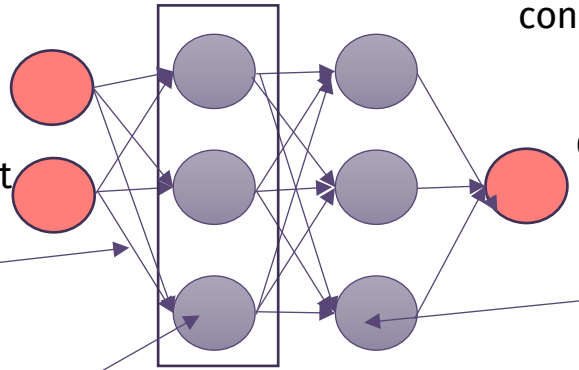
- The **deep feedforward networks** (aka **multilayer perceptrons MLPs**) are represented by composing together many different functions

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**

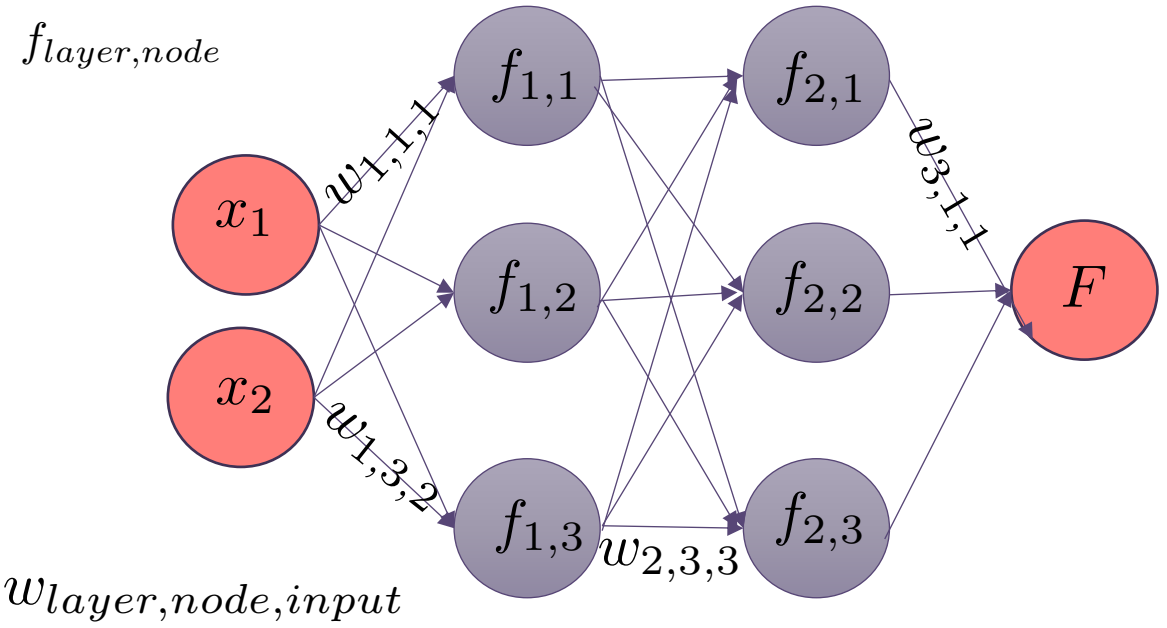
Each element of the layer plays the role of a **neuron**

Each hidden unit is associated with an activation function

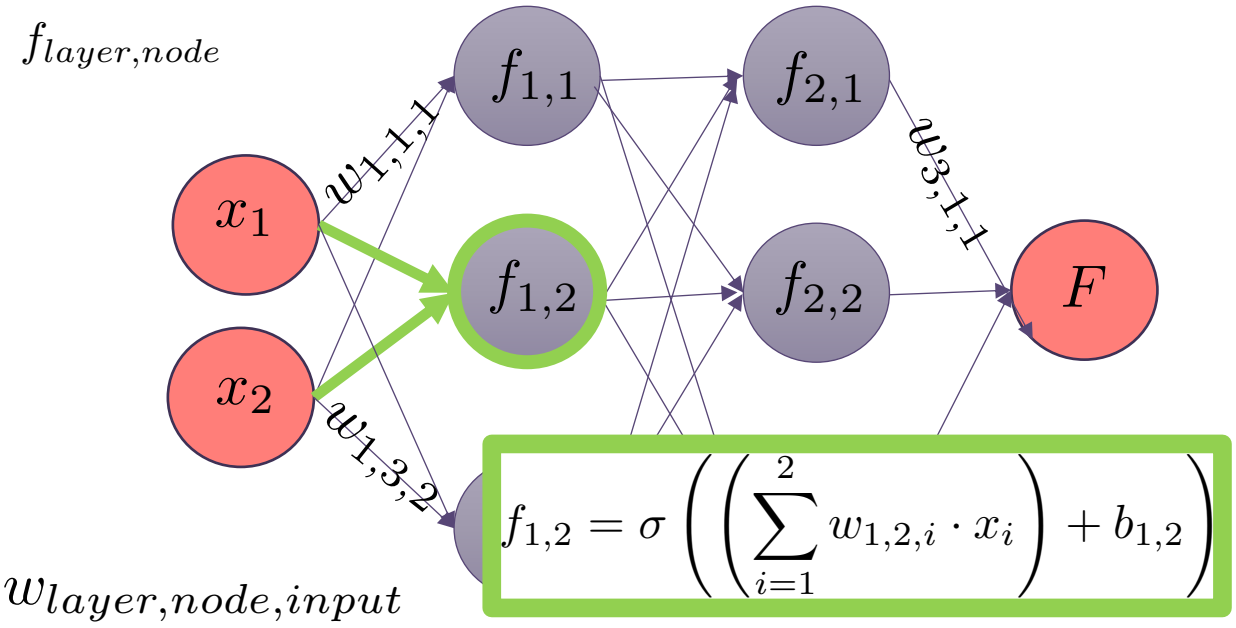
Depth (or number of layers)

Hidden: the intermediate output is not explicitly shown

Forward propagation with an example



Forward propagation with an example



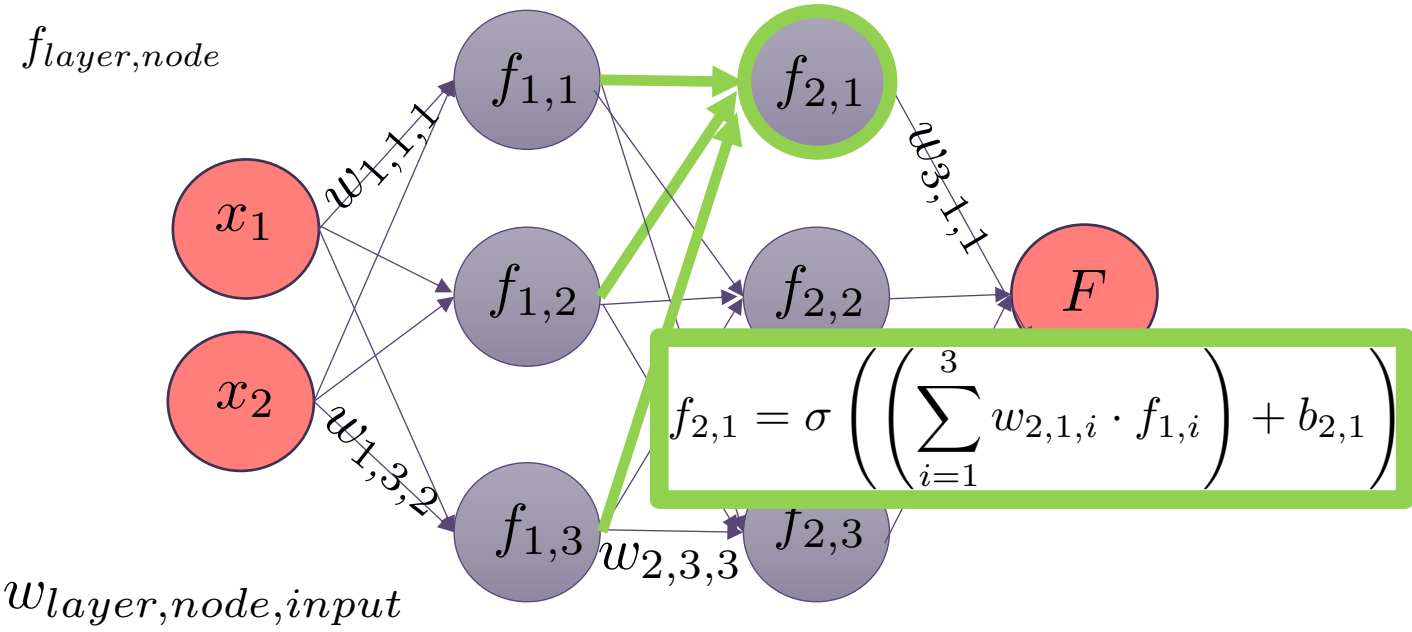
Forward propagation with an example

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

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

$$n = 1 \ldots 3$$

Forward propagation with an example



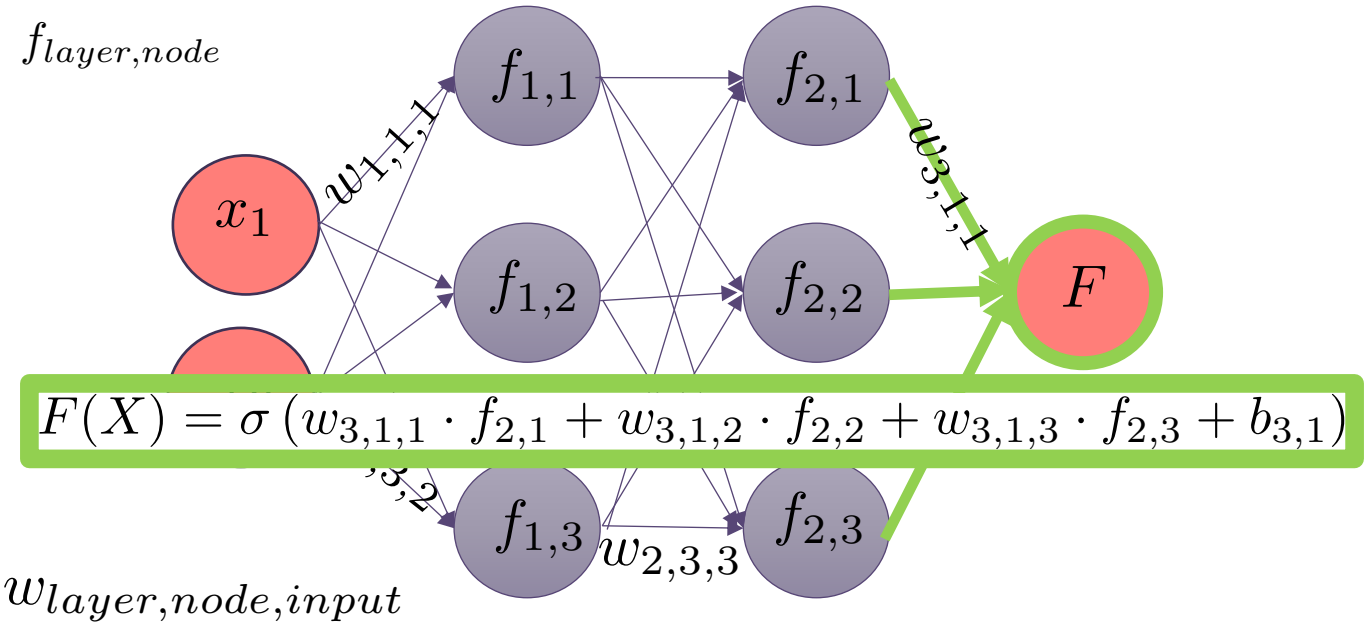
Forward propagation with an example

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

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

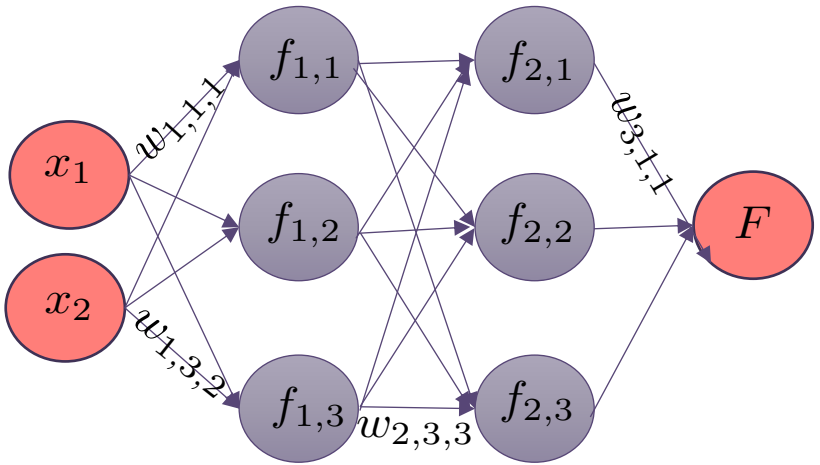
$$n = 1 \ldots 3$$

Forward propagation with an example



How many parameters?

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

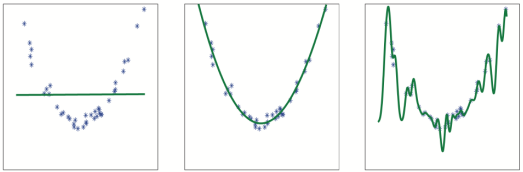


6 + 9 + 3 weights
7 bias terms

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



Training a DNN

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

- An essential element of training is (as usual) 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 would need to compute the gradient of a highly complex cost function

Training a DNN

We make use of gradient descent algoritms

$$w_0 = 0$$

$$w_t = w_{t-1} + \gamma \nabla J(w_{t-1})$$

It says how to update the weights of the network to move towards the minimum

We need to compute the gradient of a possibly very complex function!

Training a DNN

Back-propagation

- Backpropagation (1960s) aims to minimize the cost function by adjusting weights and biases of the networks
- The level of adjustment is determined by the gradients of the cost function with respect to those parameters.
- The goal of backpropagation is to compute the partial derivatives of the cost function with respect to any weight w or bias b in the network.

Training a DNN

Why is it called backpropagation?

There are two main phases during deep networks training

- Forward propagation: the input flows into the network and produces a cost
- Backward propagation: allows the info to flow back into the net to compute the gradient

A key ingredient: the chain rule of derivation

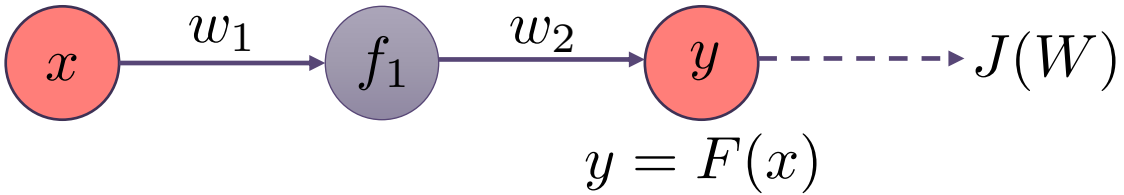
- For arbitrarily long composite functions

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

- 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

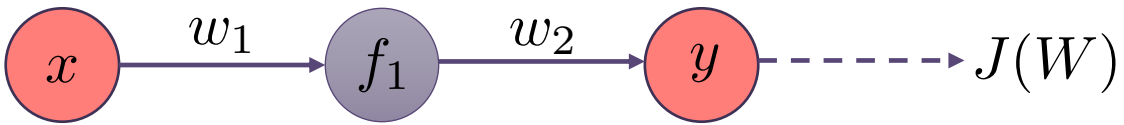
Backpropagation with an example



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



$$y = F(x)$$

$$\frac{\partial J(W)}{\partial w_2}$$

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

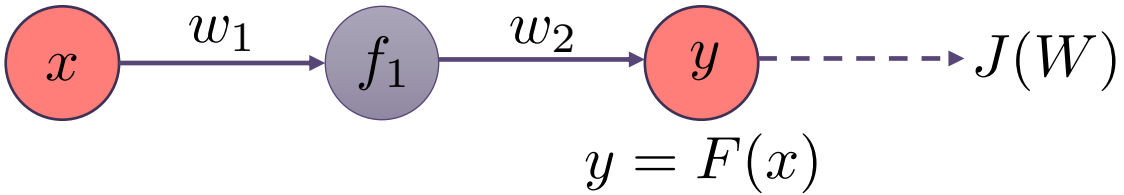
Backpropagation with an example

$$y = F(x)$$

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

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

$$y = F(x)$$

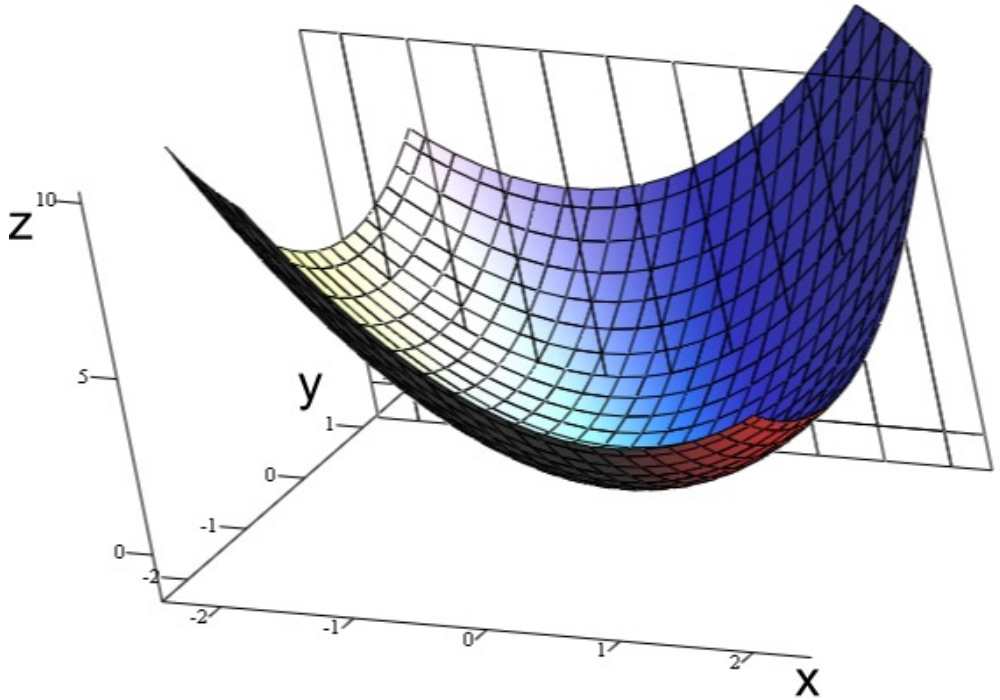
$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial y} \cdot \frac{\partial y}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1}$$

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

UniGe | MaLGa

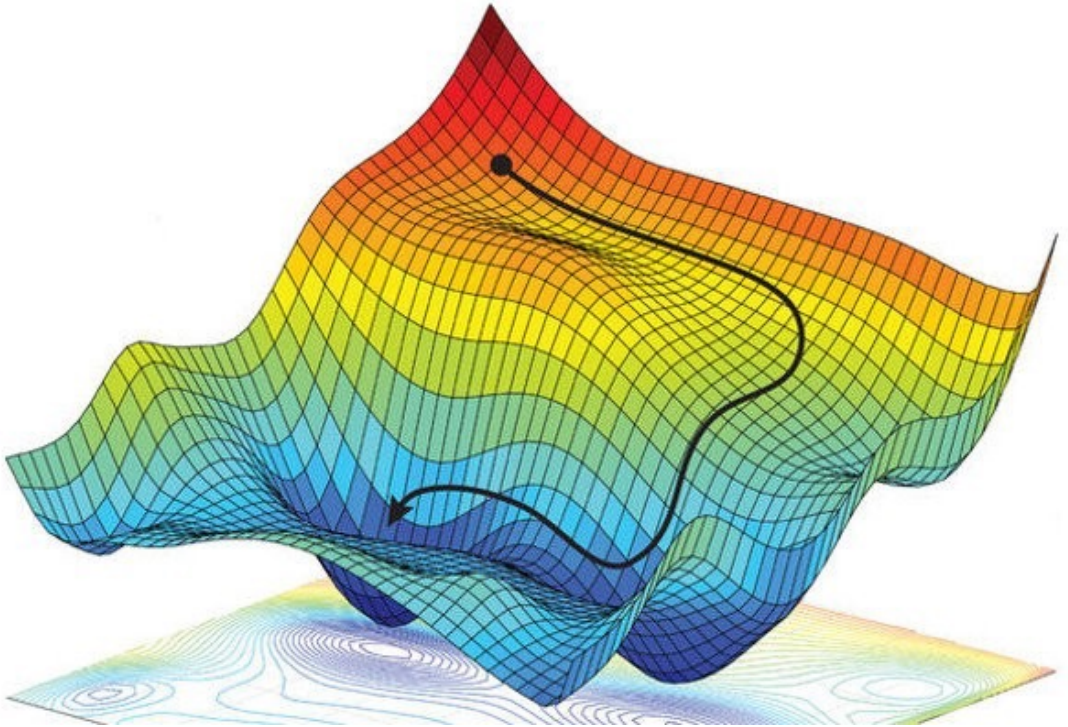
More details about deep networks training

Training a deep NN is not...



https://towardsdatascience.com/understand-convexity-in-optimization-db87653bf920?gi=b93b0698a062

… rather…



https://medium.com/swlh/non-convex-optimization-in-deep-learning-26fa30a2b2b3

Loss functions

- For regression problems you may use the square loss

$$\ell(y_i, \hat{f}(x_i)) = (y_i - \hat{f}(x_i))^2$$

- For classifiers?
  - Binary classification: cross-entropy log

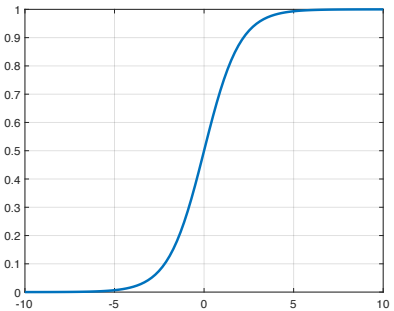
$$\ell(y_i, \hat{f}(x_i)) = -y_i \cdot log(\hat{f}(x_i)) - (1 - y_i) \cdot log(1 - \hat{f}(x_i))$$

	$\hat{f}(x_i) = 0$	$\hat{f}(x_i) = 1$
$y_i = 0$	0	inf
$y_i = 1$	inf	0

Output units

For binary classification

A good choice is the sigmoid output → it «enforces» the binary values



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

For multi-class classification

It is covenient to use the Softmax operator, that allows us to «represent» the probability distribution over the M different classes

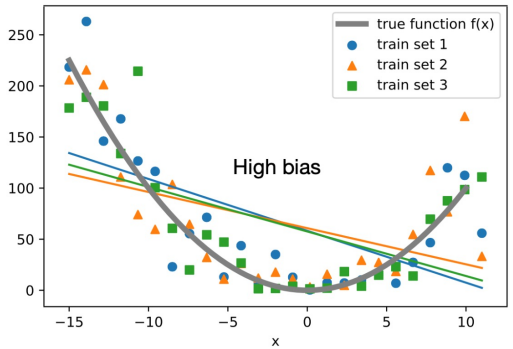
$$\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

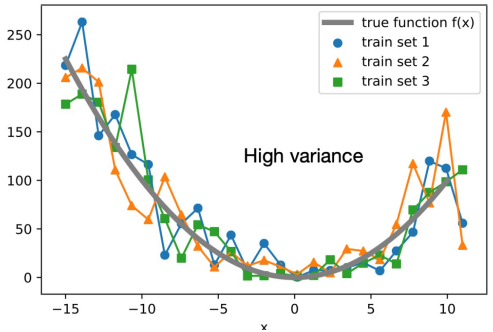
$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{M} e^{z_k}}$$

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





Regularization

- High capacity models fit training data well but exhibit high variance and do not generalize well

- What we do not want: Error(training) << Error(test)

- To prevent over-fitting, **regularization** can be used, to stabilize the learning of weights, but we need to pay attention:
  - Strong regularization: adds significant bias
  - Weak regularization: leads to high variance

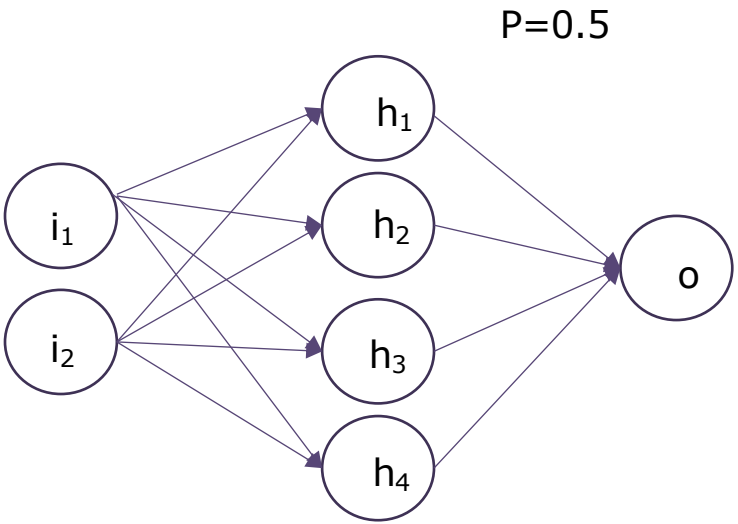
Bagging and Dropout

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

- With limited training data weights become adapted to them, and the model doesn't generalize

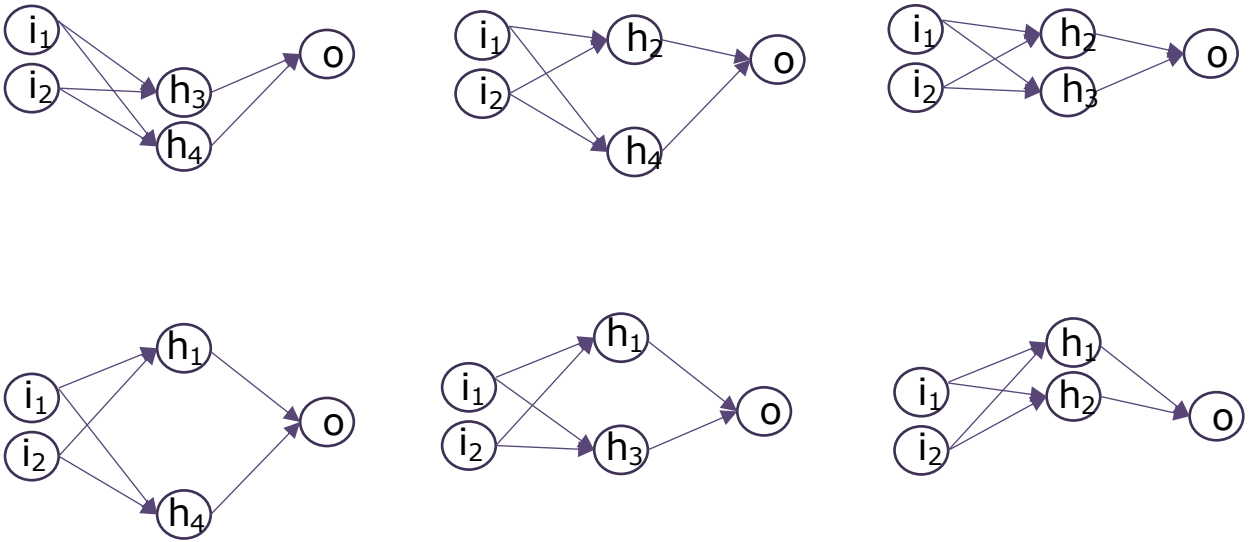
- Bagging and Dropout are ways to break this co-adaptation

Bagging (intuition)



Bagging (intuition)

P=0.5



From bagging to dropout

- Each sub-network is trained and evaluated on each test sample
- The final prediction is given by the votes of all models
- This is computationally very expensive, dropout is a way to approximate the same behavior
- At each step of the gradient descent some fraction of weights are dropped-out of each layer

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 would need to compute the gradient of a highly complex function

Gradient descent

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

Batch gradient descent

- **Simple idea**: iterate the weights estimation until a stopping criteria or error tolerance is reached

- **Cons**: requires the computation of the gradient for all weights at one time as a batch at each step... so it's computationally heavy

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t, X)$$

$$||W_{t+1} - W_t|| < toll$$

Stochastic Gradient Descent

- The weights update is done after the evaluating the loss function for each sample

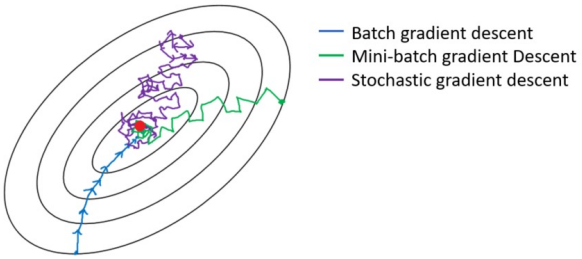
$$W_{t+1} = W_t + \alpha \nabla_W J(W_t, x_i)$$

- SGD is known to converge well in practice: empirically, it provides a better exploration of the loss function space

A compromise: using mini-batches

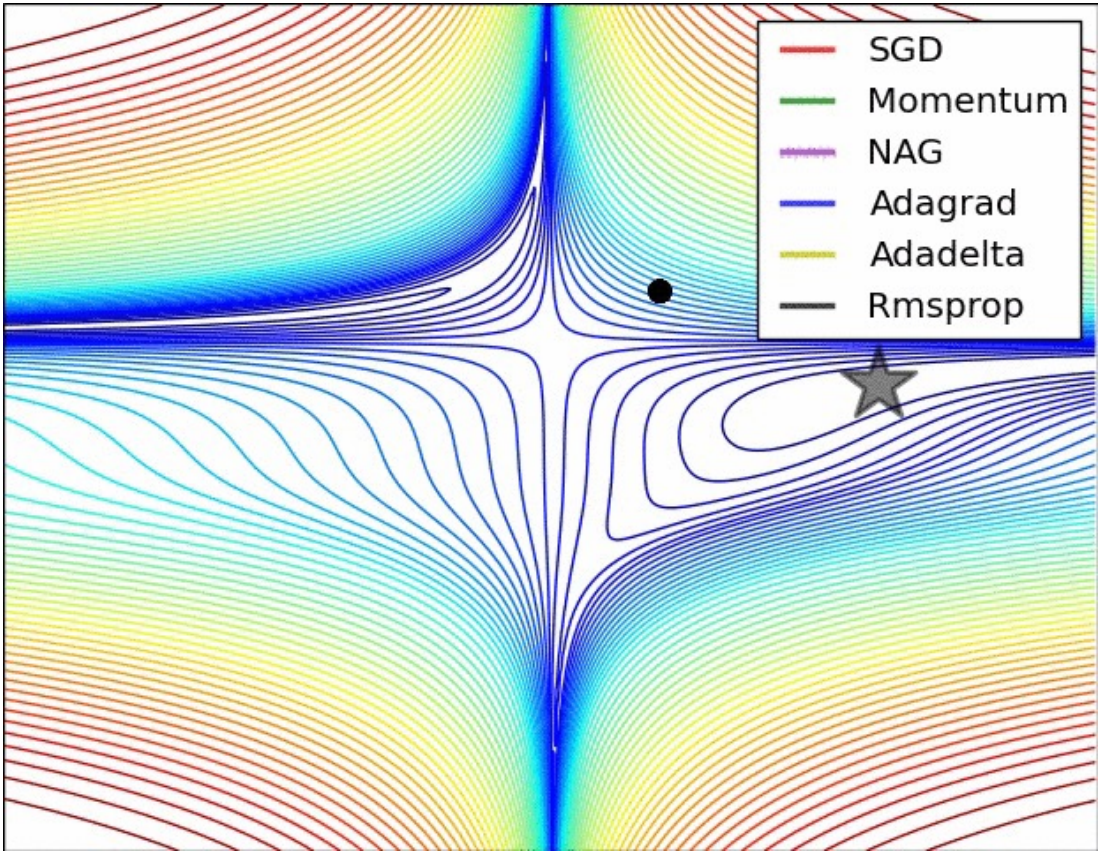
Select a portion of the training set of size s (the mini-batch size) and update the weights after evaluating the loss function on it

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t, x_{i:i+s})$$



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

Not only GD



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