Classification and Representation Learning

Course 6&7&8 : (Deep) Neural Networks

Hoel Le Capitaine

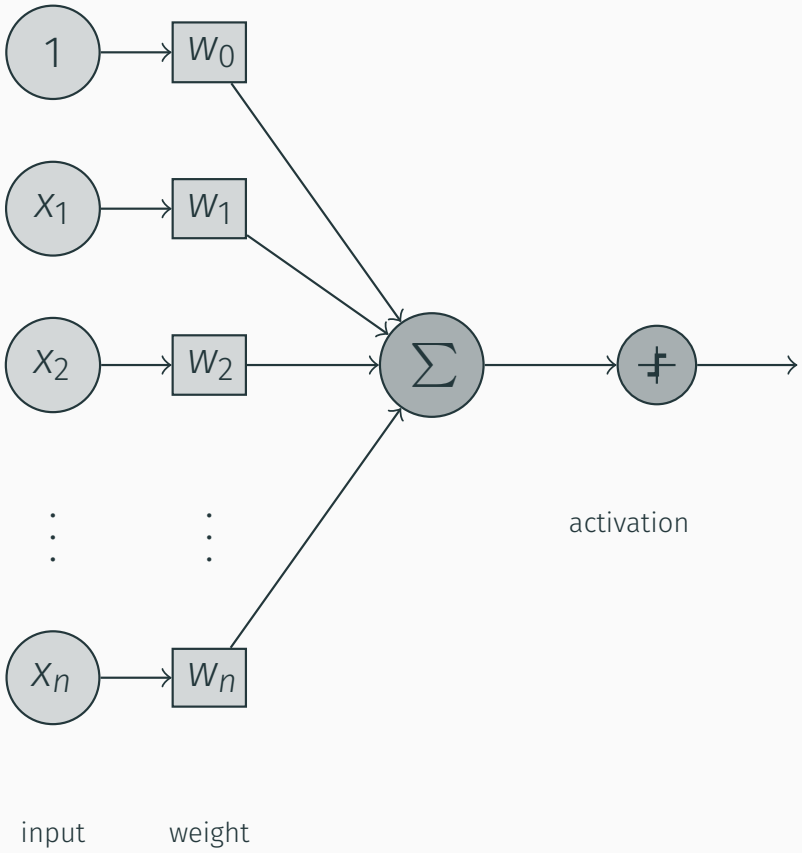
Academic year 2017-2018

Artificial neural networks

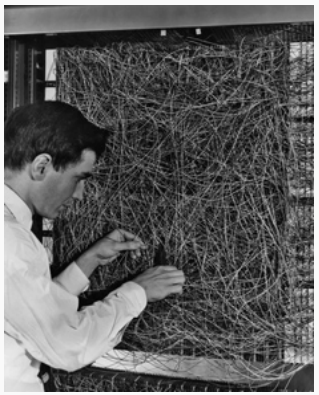
Artificial neural networks

The brain

- brain activity, dendrits, axon, synapses.
- McCulloch and Pitts (1943)



Neuron



Rosenblatt (60's)

Neural network structure

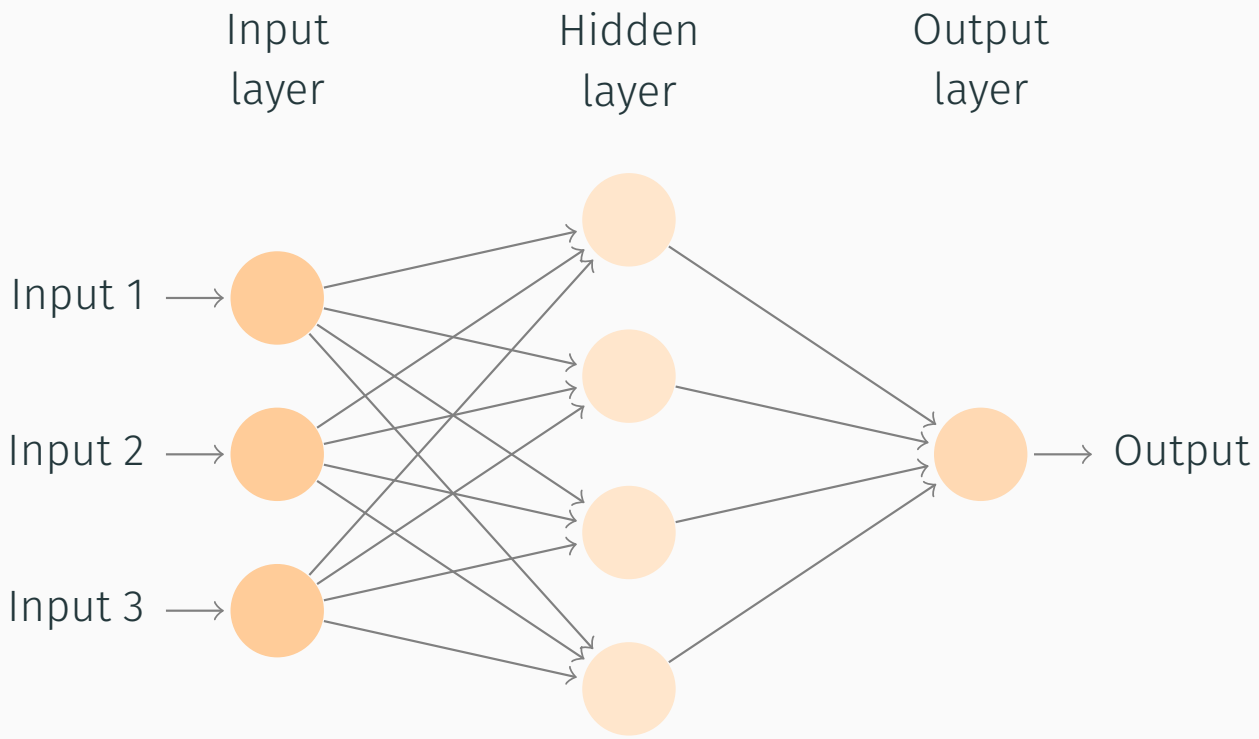
Perceptrons

- neural networks are made of nodes connected with each other through edges (oriented). Nodes are also called perceptrons
- a link from node i to node j propagates the activation a_i from i to j
- each link (edge) is given a weight describing the strenght of the connection
- each unit j is a weighted sum of its entries, $in_j = \sum_{i=0} w_{i,j} a_i$
- and applies an activation function on it: $g(in_j)$
- the activation function is often the threshold (perceptron) or the logistic (sigmoid perceptron)
- the non-linearity of g allows to represent (learn) a non linear function

Perceptron structure

How perceptrons are linked?

- Feedforward network : the network is as directed acyclic graph (DAG)
- Recurrent network : outputs are linked to the input, creating a dynamical system
- networks are organised into different layers, so that each node receives data from the previous layer.

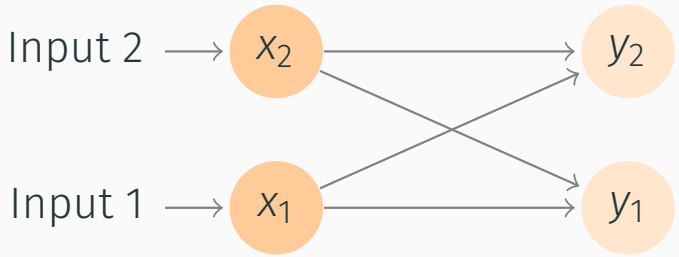


Modelisation

- if all entries are connected to all outputs, this is a perceptron network
- let consider the following data

x_1	x_2	y_1 (carry)	y_2 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

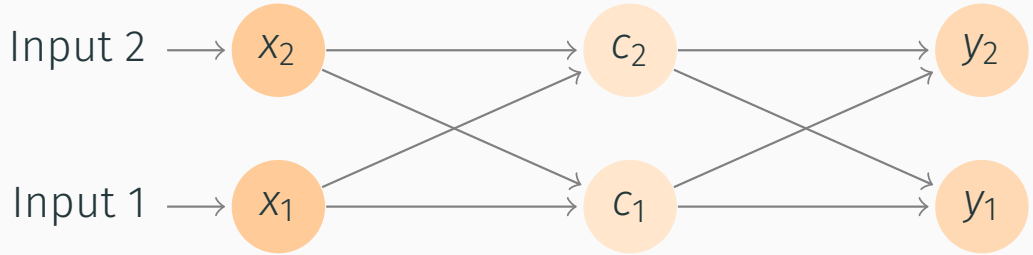
- and the following network (2 independent sub-networks)



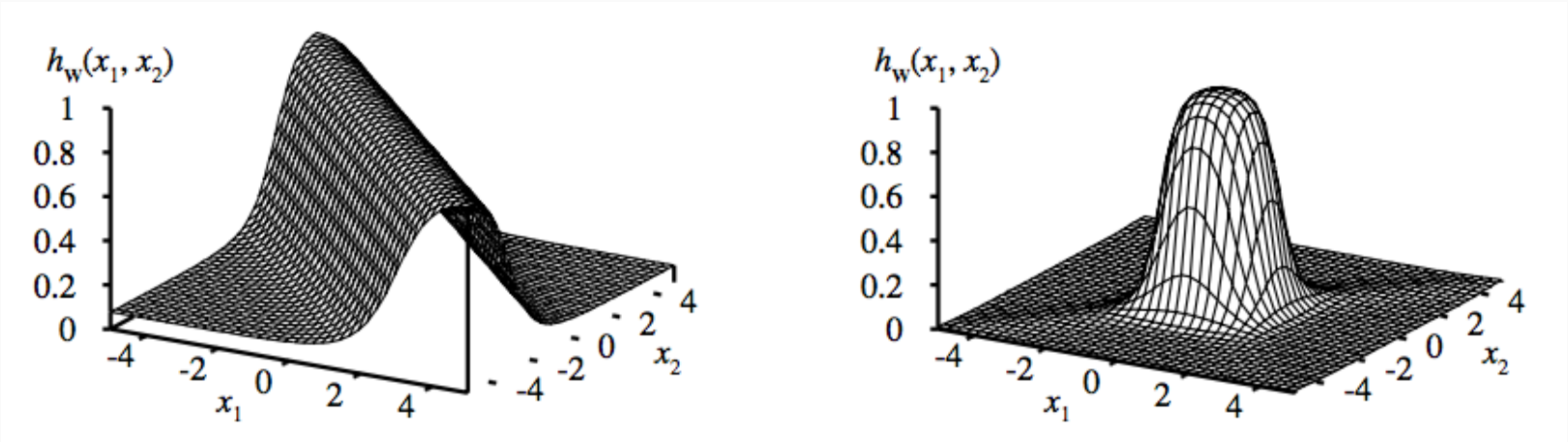
- depending the activation function, perceptron rule, or logistic rule
- y_1 ok, not y_2

Non linearity

- right from the early definition, several layers were considered, but they did not know how to train them
- appears to be simple if we consider a network as a function $h_{\mathbf{w}}(\mathbf{x})$



- $y_2 = g(w_{0,y_2} + w_{c_2,y_2} c_2 + w_{c_1,y_2} c_1)$
  $y_2 = g(w_{0,y_2} + w_{c_2,y_2} g(w_{0,c_2} + w_{x_2,c_2} x_2 + w_{x_1,c_2} x_1) + w_{c_1,y_2} g(w_{0,c_1} + w_{x_2,c_1} x_2 + w_{x_1,c_1} x_1))$
- composition of non-linear functions



Propagation

- a perceptron network decompose the problem, not the mlp !
- if the loss function is additive, we may have random weights

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \sum_k \frac{\partial}{\partial w} (y_k - a_k)^2$$

- main problem for the hidden layer: we do not know the desired output on it
- backpropagation of the output error by chain derivation
- we can write a function $f(x)$ from an intermediary result $g(x)$:

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

- with many intermediary results $g_i(x)$:

$$\frac{\partial f(x)}{\partial x} = \sum_i \frac{\partial f(x)}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$$

Backpropagation of the gradient

Learning rule derivation

- by gradient, we obtain

$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial w_{i,j}} Loss(w)$$

- and chain derivation gives

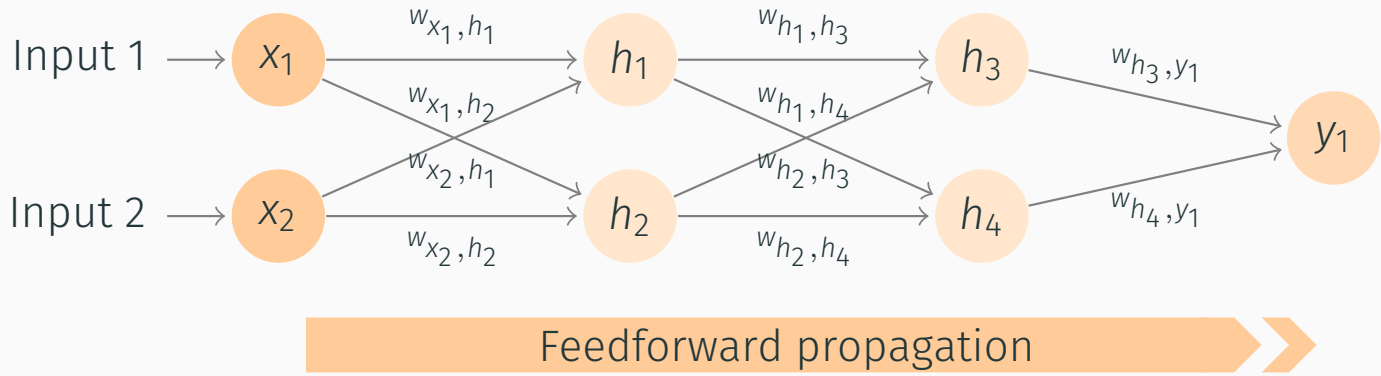
$$w_{i,j} \leftarrow w_{i,j} - \alpha \frac{\partial}{\partial in_j} Loss(w) \frac{\partial}{\partial w_{i,j}} in_j$$

- naive computation, not useful. The derivative on layer ℓ can be computed with backpropagation from the derivative of the layer $\ell + 1$:

$$\begin{aligned}
\frac{\partial Loss}{\partial in_j} &= \frac{\partial Loss}{\partial a_j} \frac{\partial a_j}{\partial in_j} \\
&= \left(\sum_k \frac{\partial Loss}{\partial in_k} \frac{\partial in_k}{\partial a_j} \right) \frac{\partial g(in_j)}{\partial in_j} \\
&= \left(\sum_k \frac{\partial Loss}{\partial in_k} w_{j,k} \right) g(in_j)(1 - g(in_j))
\end{aligned}$$

Visualizing the backpropagation

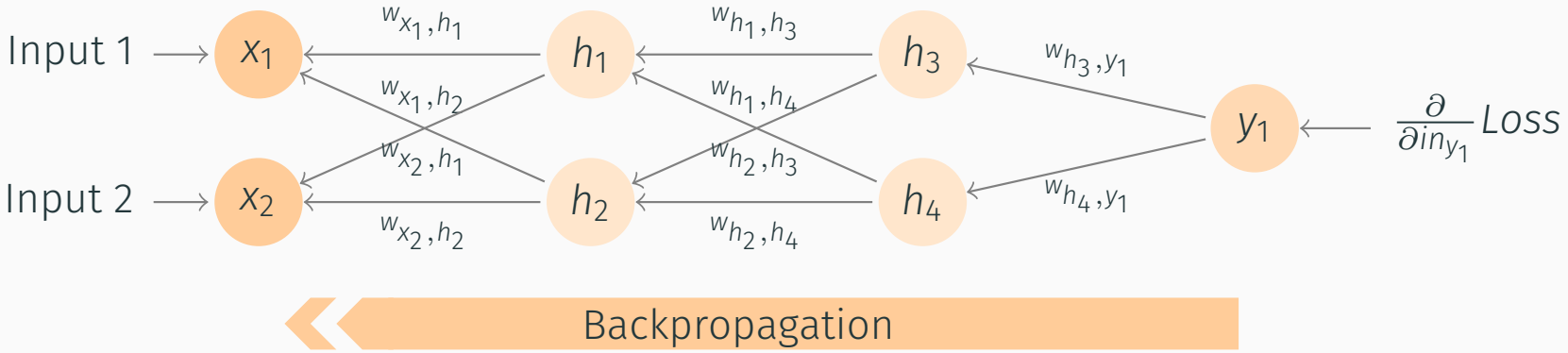
The algorithm starts with feedforward propagation



$$h_1 = g\left(w_{0,h_1} + w_{x_1,h_1}x_1 + w_{x_2,h_1}x_2\right)$$

Visualizing the backpropagation

Output gradient is obtained, and backpropagated



$$\frac{\partial}{\partial in_j} Loss = g(in_j)(1 - g(in_j)) \sum_k w_{j,k} \frac{\partial}{\partial in_k} Loss = -\Delta_j$$

- update rule

$$\begin{aligned} w_{i,j} &= w_{i,j} - \alpha \frac{\partial}{\partial in_j} Loss \frac{\partial}{\partial w_{i,j}} in_j \\ &= w_{i,j} + \alpha a_i \Delta_j \end{aligned}$$

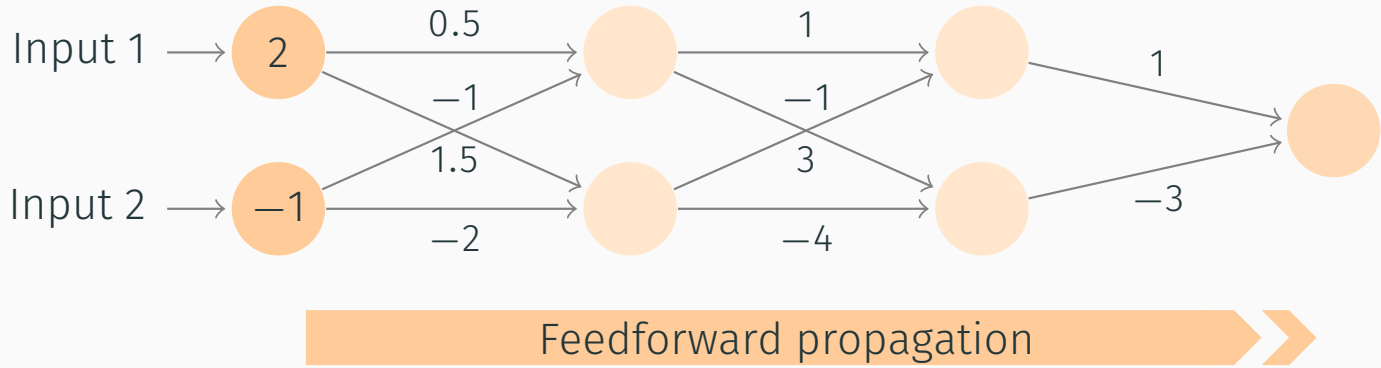
Gradient backpropagation algorithm

Input : examples (x, y), mlp with L layers, weights $w_{i,j}$, activation function g
Output : neural network
foreach *weight* $w_{i,j}$ **do**
  $w_{i,j} \leftarrow$ random low value
end
repeat
  foreach *example* (x, y) **do**
    foreach *node i of input layer* **do**
      $a_i \leftarrow x_i$
    end
    for $\ell \leftarrow 2$ **a** L **do**
      foreach *node j of layer* ℓ **do**
        $in_j \leftarrow \sum_i w_{i,j} a_i$
        $a_j \leftarrow g(in_j)$
      end
    end
    foreach *node j of output layer* **do**
      $\Delta_j \leftarrow (y_j - a_j)\ (= -\partial Loss / \partial in_j)$
    end
    for $\ell \leftarrow L - 1$ **a** 1 **do**
      foreach *node i of layer* ℓ **do**
        $\Delta_i \leftarrow g'(in_i) \sum_j w_{i,j} \Delta_j$
      end
    end
    foreach $w_{i,j}$ **do**
      $w_{i,j} \leftarrow w_{i,j} + \alpha a_i \Delta_j$
    end
  end
until *stopping criterion satisfied*;

Example

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



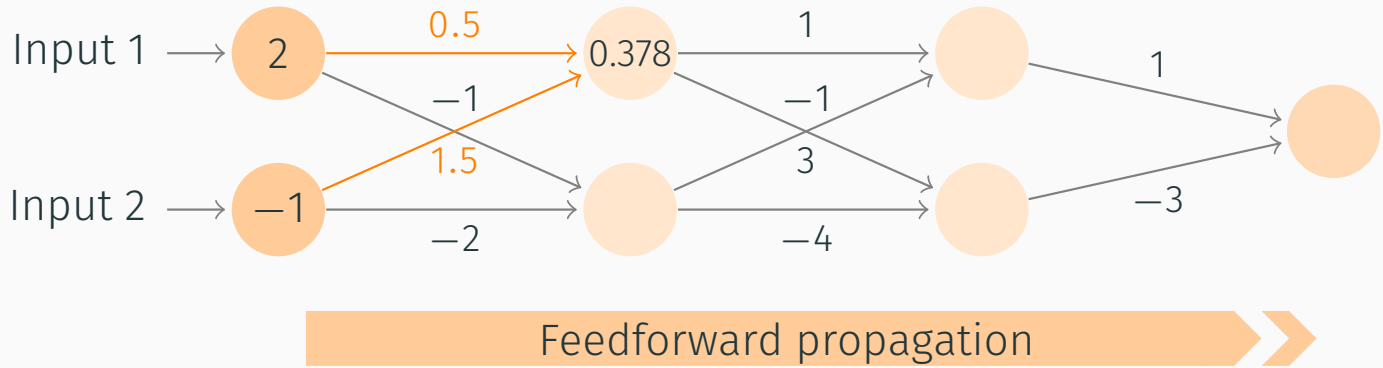
Feedforward propagation

$$a_k = g\left(\sum_j w_{j,k} a_j\right)$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



Feedforward propagation

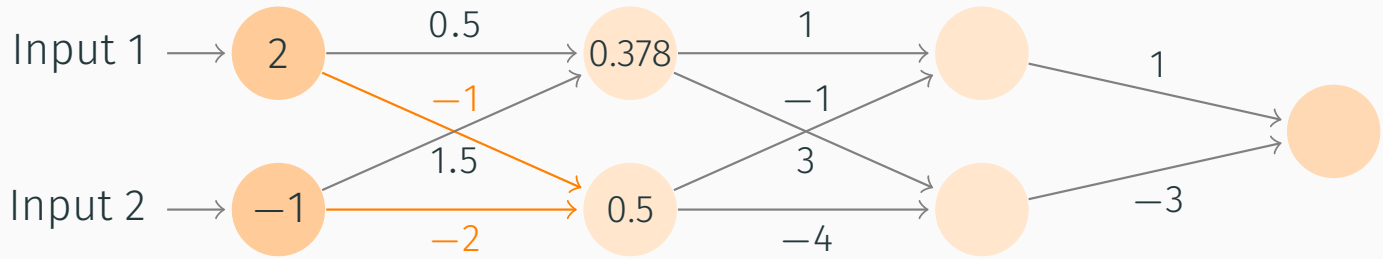
$$a_k = g\left(\sum_j w_{j,k} a_j\right)$$

$$g(0.5 \times 2 + 1.5 \times -1) = g(-0.5) = 0.378$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



Feedforward propagation

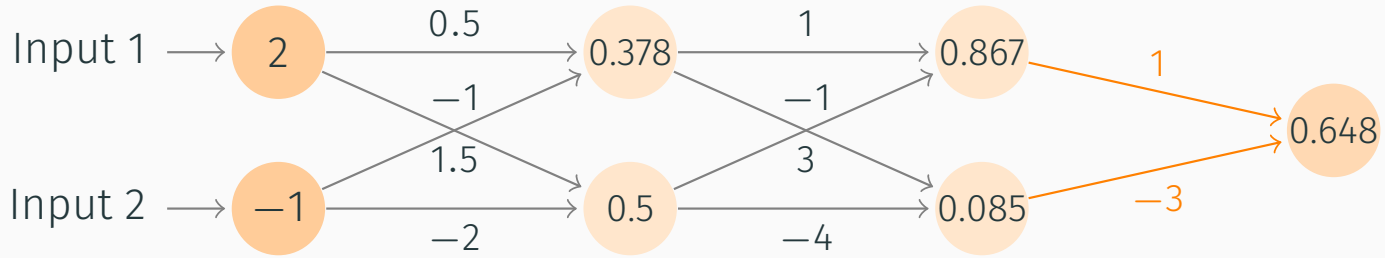
$$a_k = g\left(\sum_j w_{j,k} a_j\right)$$

$$g(-1 \times 2 - 2 \times -1) = g(0) = 0.5$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



Feedforward propagation

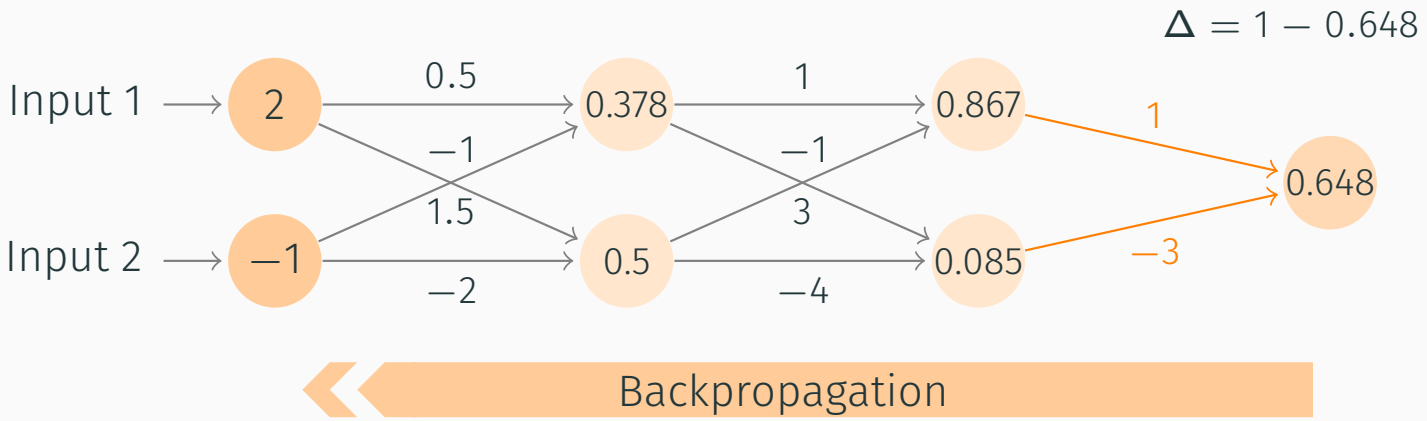
$$a_k = g\left(\sum_j w_{j,k} a_j\right)$$

$$g(1 \times 0.867 - 3 \times 0.085) = g(0.612) = 0.648$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)

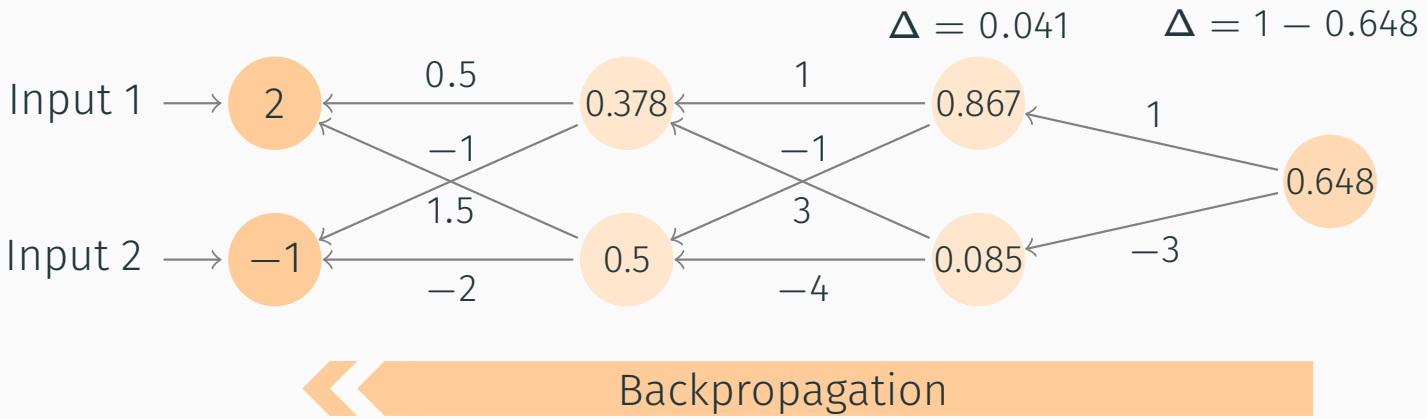


$$\Delta = y - a$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



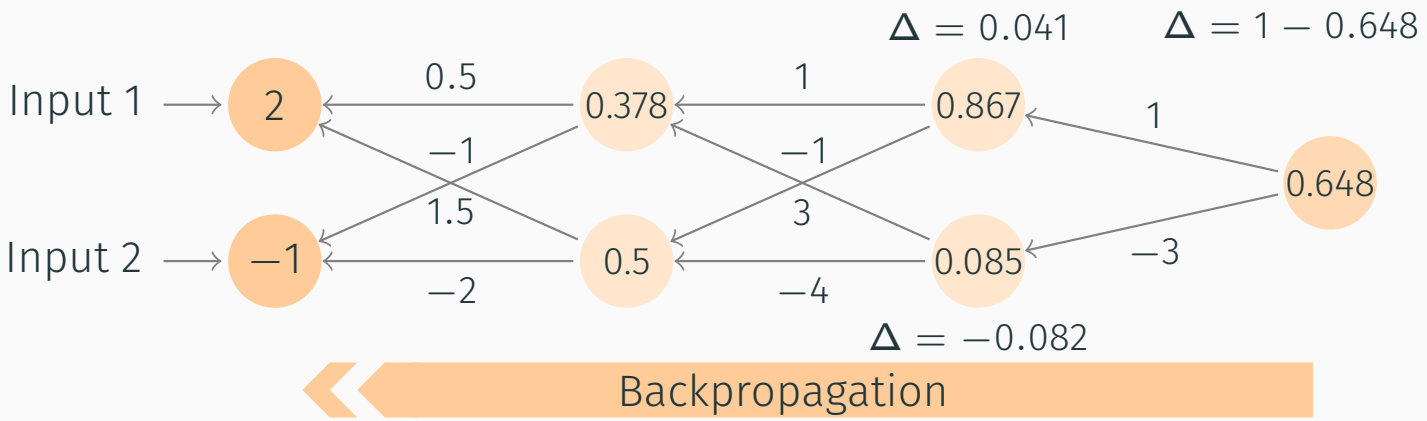
$$\Delta_i = g(in_i) * (1 - g(in_i)) \sum_j w_{i,j} \Delta_j$$

$$\Delta = 0.867 \times (1 - 0.867) \times 1 \times 0.352 = 0.041$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



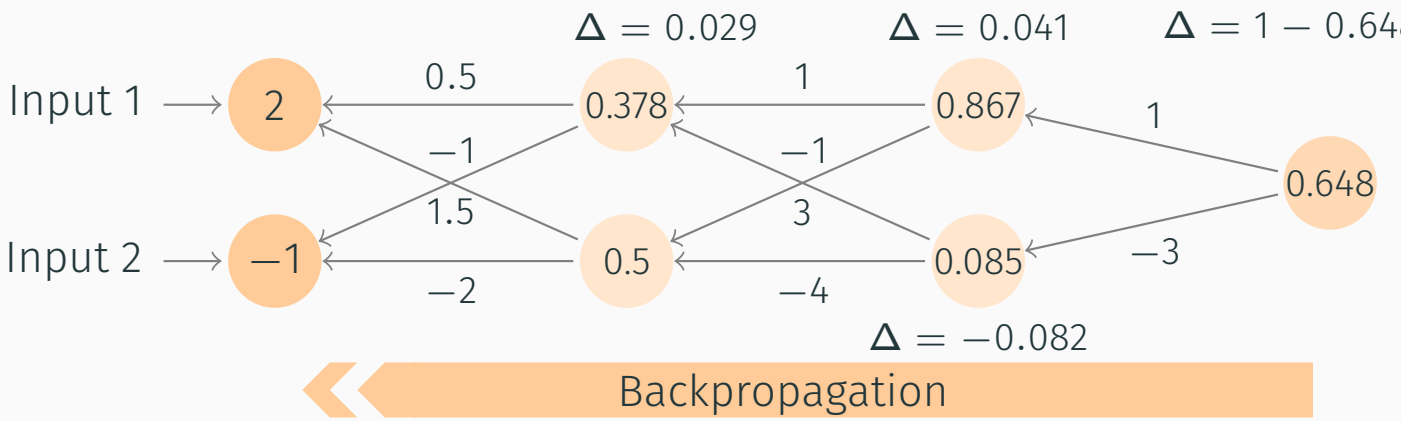
$$\Delta_i = g(in_i) * (1 - g(in_i)) \sum_j w_{i,j} \Delta_j$$

$$\Delta = 0.085 \times (1 - 0.085) \times -3 \times 0.352 = -0.082$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



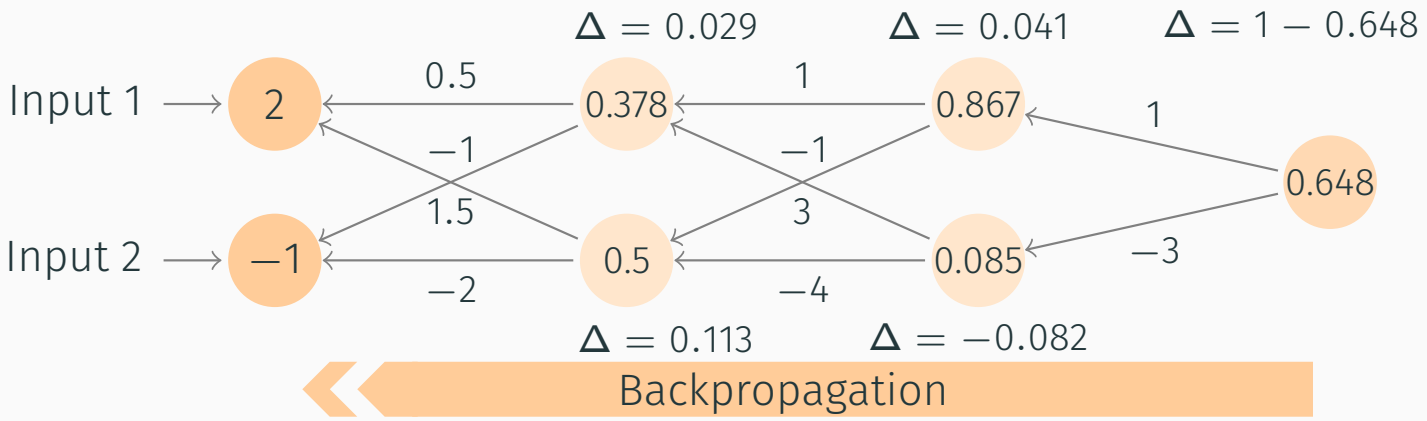
$$\Delta_i = g(in_i) * (1 - g(in_i)) \sum_j w_{i,j} \Delta_j$$

$$\Delta = 0.378 \times (1 - 0.378) \times (1 \times 0.041 + -1 \times -0.082) = 0.029$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



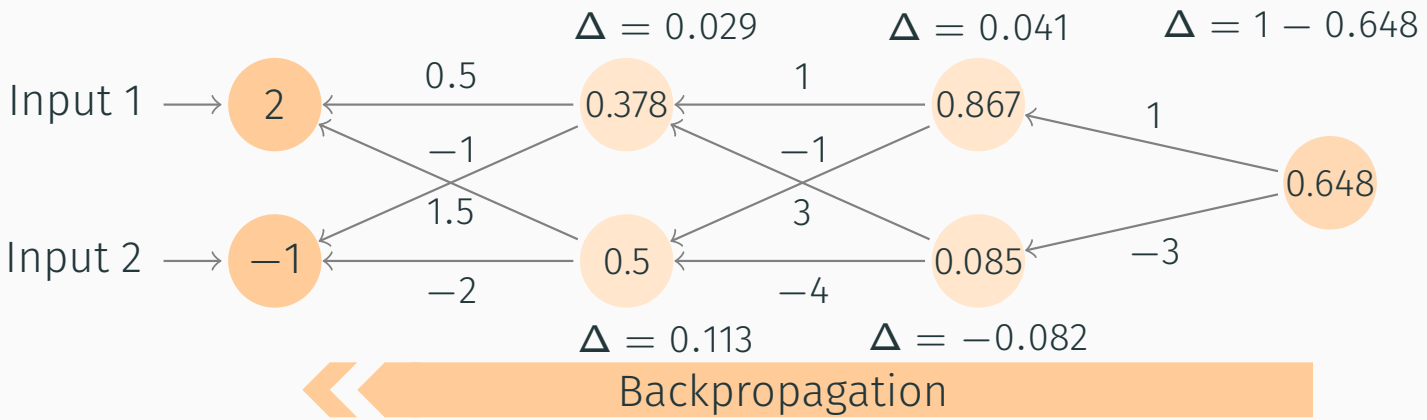
$$\Delta_i = g(in_i) * (1 - g(in_i)) \sum_j w_{i,j} \Delta_j$$

$$\Delta = 0.5 \times (1 - 0.5) \times (3 \times 0.041 + -4 \times -0.082) = 0.113$$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



$$\Delta_i = g(in_i) * (1 - g(in_i)) \sum_j w_{i,j} \Delta_j$$

Weight update $w_{i,j} = w_{i,j} + \alpha a_i \Delta_j$, (we choose $\alpha = 0.2$) :

$w_{x_1,h_1} = 0.5 + 0.2 \times 2 \times 0.029 = 0.512$
$w_{x_1,h_2} = -1 + 0.2 \times 2 \times 0.113 = -0.954$
$w_{x_2,h_1} = 1.5 + 0.2 \times -1 \times 0.029 = 1.494$
$w_{x_2,h_2} = -2 + 0.2 \times -1 \times 0.113 = -2.022$

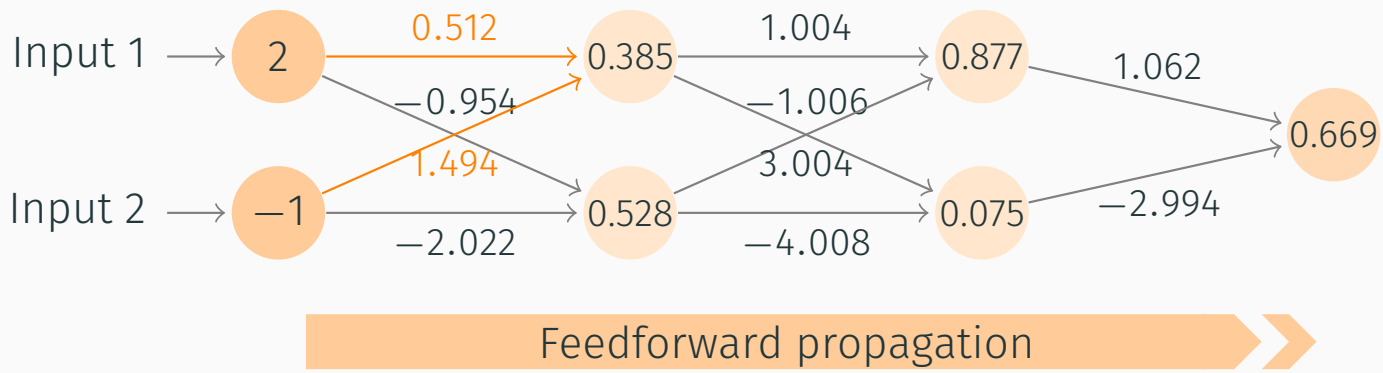
$w_{h_1,h_3} = 1 + 0.2 \times 0.378 \times 0.041 = 1.004$
$w_{h_1,h_4} = -1 + 0.2 \times 0.378 \times -0.082 = -1.006$
$w_{h_2,h_3} = 3 + 0.2 \times 0.5 \times 0.041 = 3.004$
$w_{h_2,h_4} = -4 + 0.2 \times 0.5 \times -0.082 = -4.008$

$w_{h_3,y_1} = 1 + 0.2 \times 0.867 \times 0.352 = 1.062$
$w_{h_4,y_1} = -3 + 0.2 \times 0.085 \times 0.352 = -2.994$

Exemple

For one observation : forward/backward

Consider an input $\mathbf{x} = [2, -1]$, with corresponding output $y = 1$, and the following network (already initialized)



$$a_k = g\left(\sum_j w_{j,k} a_j\right)$$

$$g(0.512 \times 2 + 1.494 \times -1) = g(-0.47) = 0.385$$

Example : **`http://playground.tensorflow.org/`**

Exercise

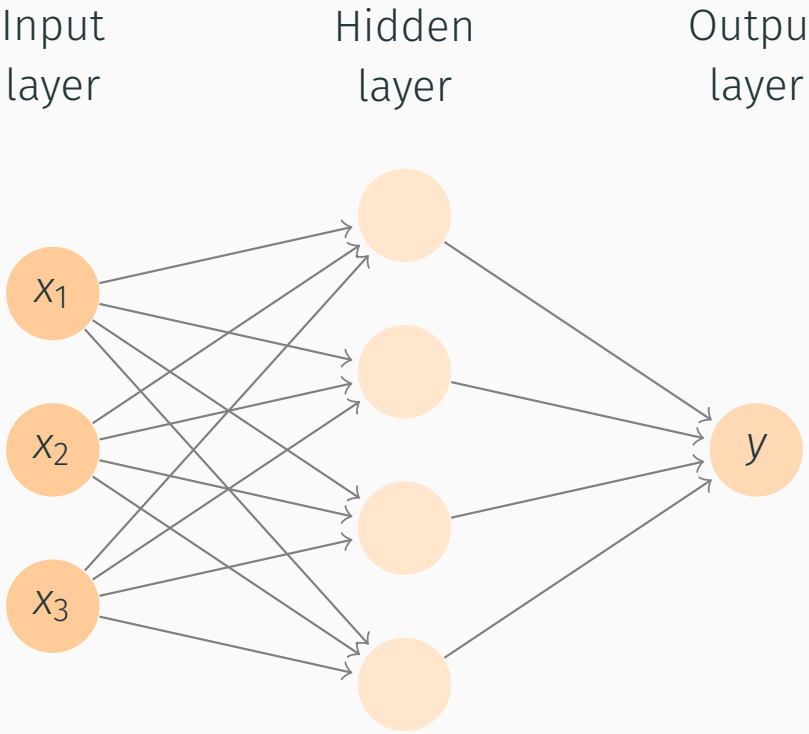
Exercise 1: Backpropagation learning

Given the previous network and considering a second input $\mathbf{z} = [1, 2]$, with output $y = 0$,

- what is the class prediction of $\mathbf{z}$ by the network ?
- draw the network after weight update according to this new observation $\mathbf{z}$

Auto-Encoders

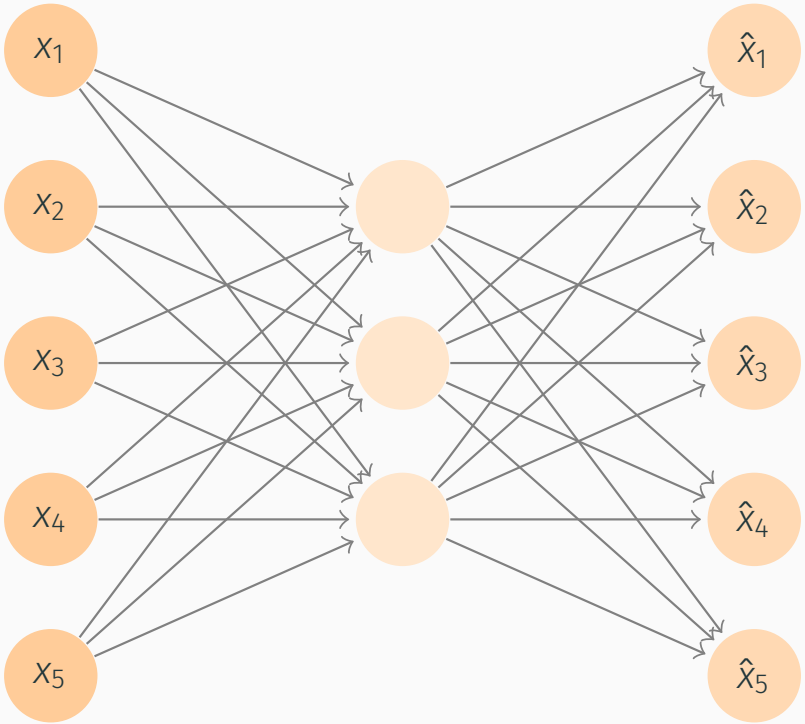
Multi-layer perceptron



Summary

- in supervised learning, a MLP has one input layer, one or more hidden layers, and one output layer
- the prediction of the output $y = h(\mathbf{x})$ for an input $\mathbf{x}$ is given by feedforward propagation
- weights are learned with backpropagation algorithm

Auto-encoders



Principle

- an auto-encoder is a MLP learning the identity function $f(\mathbf{x}) = \mathbf{x}$, using a different number of neurons in the hidden layer than in the input layer
- an auto-encoder does unsupervised learning, output data used for learning is the input data
- enforcing projection onto the hidden layer, with less dimensions, means that relevant features are extracted in this layer (dimensionality reduction)

Auto-encoders and sparseness

- There can also be more neurons in the hidden layer than in the input layer, but we have to make sure that hidden neurons are not often active.
- Sparseness can be enforced by penalizing the hidden neurons which are too often active for the training data.
- Optimally, we would like that the mean activity of a hidden neuron $y_j$for all input images is smaller than a threshold ρ (e.g. 0.05):

$$\rho_j = \frac{1}{N} \sum_{i=1}^{N} y_j(x_i) < \rho$$

Algorithm modification

$$\rho_j = \frac{1}{N}\sum_{i=1}^{N} y_j(x_i) < \rho \quad \forall j \in [1, K]$$

The backpropagation algorithm can be modified to take into account this new constraint. The objective function to minimize is now:

$$J(\mathbf{w}, b) = E(\mathbf{w}, b) + \beta \sum_{i=1}^{K} KL(\rho || \rho_j)$$

where

- $E(\mathbf{w}, b)$ is the quadratic error $\sum_{i=1}^{N}(h(\mathbf{x}_i) - y_i)^2$
- $\sum_{i=1}^{K} KL(\rho || \rho_j)$ is the sum of the Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean ρ_j.

This is a regularization process trying to minimize the training error while maximizing the sparseness, controlled by the parameter β.

Auto-encoders and sparseness

KL-divergence

$$KL(\rho \| \rho_j) = \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}$$

- The KL-divergence is minimal when $\rho = \rho_j$ (sparse encoding), and increases for different values
- KL-divergence is a standard function for measuring how different two distributions are

Modification of the algorithm

$$J(\mathbf{w}, b) = E(\mathbf{w}, b) + \beta \sum_{i=1}^{K} KL(\rho || \rho_j)$$

Backpropagated error

Instead of

$$\Delta_i = g'(in_i) \sum_j w_{i,j} \Delta_j$$

the backpropagated error becomes

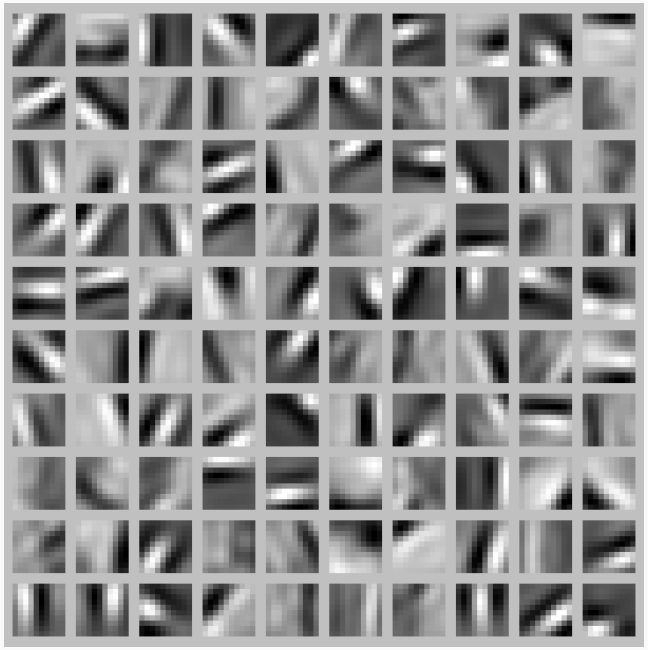
$$\Delta_i = g'(in_i) \sum_j w_{i,j} \Delta_j + \beta \left(-\frac{\rho}{\rho_j} + \frac{1-\rho}{\rho_j} \right)$$

Need to compute the feedforward pass twice: once to get the sparseness ρ_j for all examples, and once to apply the learning rule

Visualizing autoencoders

What has been learnt?

- consider the case of training on 10 x 10 images, so that $d = 100$
- each hidden unit i computes a function of the input:
- we will visualize the function computed by hidden unit i, which depends on the parameters W_{ij} (ignoring the bias term for now) using a 2D image
- with 100 hidden units :



Each square in the figure above shows the (norm bounded) input image x that maximally actives one of 100 hidden units.

Deep networks

Using an autoencoder to classify

How to do this?

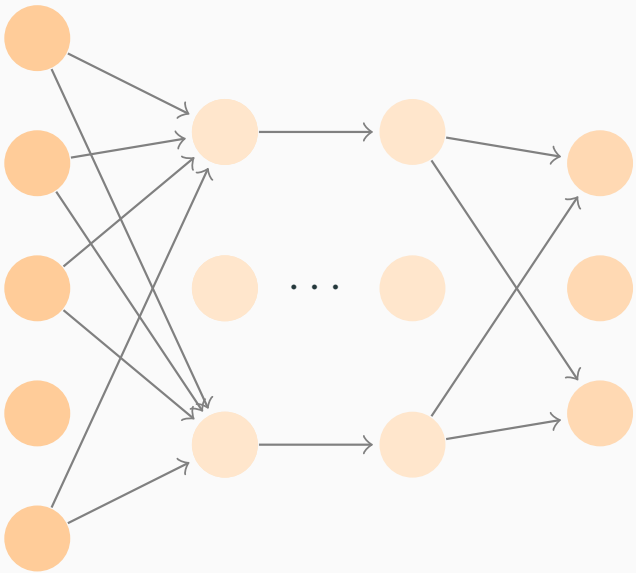
- prior to training using a lot of unlabeled data, one can extract a feature vector from any image using the weights from the input to the hidden layer of a sparse autoencoder.
- during training classification is then learned on a restricted set a labeled data, using whatever algorithm (logistic classifier, MLP, SVM...).
- post training: one could also fine-tune the features by allowing backpropagation to the feature space during the training phase, but this is only useful when there is a lot of labeled data.

Shallow and deep networks

From shallow to deep

- NN with 1 or 2 hidden layers are shallow networks.
- If the function to learn is too complex, a shallow network may need too many hidden neurons.
- An autoencoder may only extract firstorder features (edges in the case of natural images), what may not be enough to recognize complex objects.
- A hierarchy of autoencoders (deep network) may progressively extract edges, contours, forms and objects.

Deep network

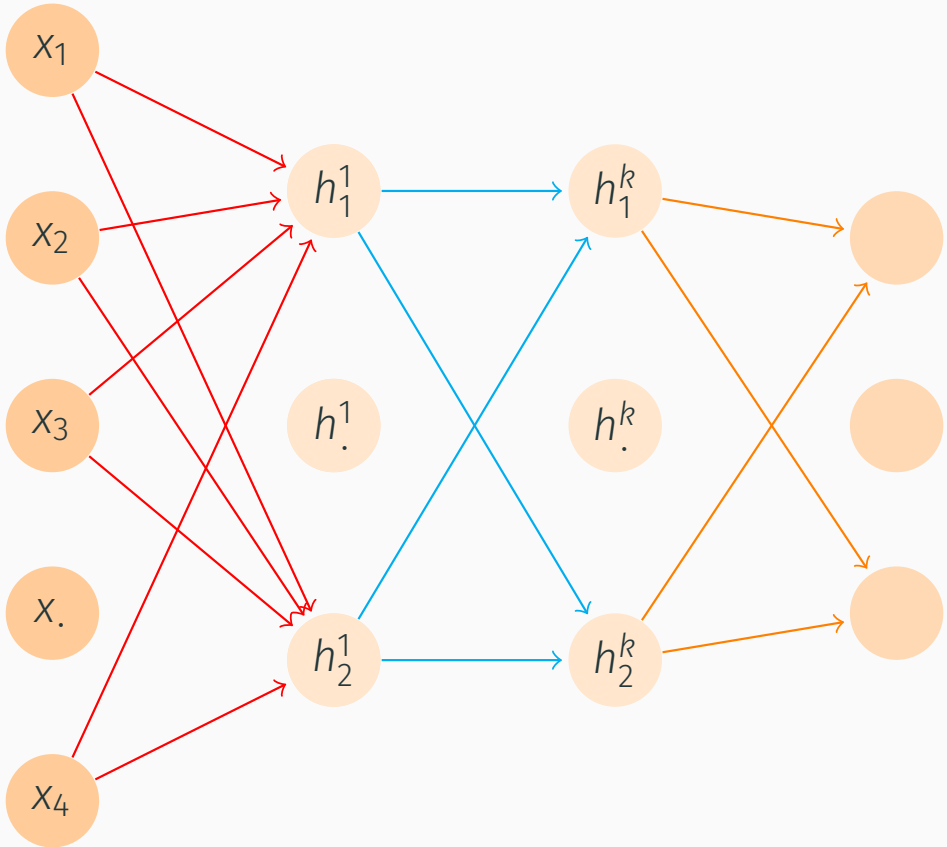


- A deep network is a MLP with several hidden layers learning a classification task.
- The hope is that the different layers extract progressively more and more complex features.
- Backpropagation does not work well:
  - The VC dimension increases, so you need a huge amount of labeled data to avoid overfitting.
  - The error function is nonconvex: local minima.
  - Gradient vanishing problem.
- Regularization (L2/L1, dropout) and a correct choice of transfer function (ReLU) may help, but a lot of labeled data is needed.

Greedy layer-wise learning

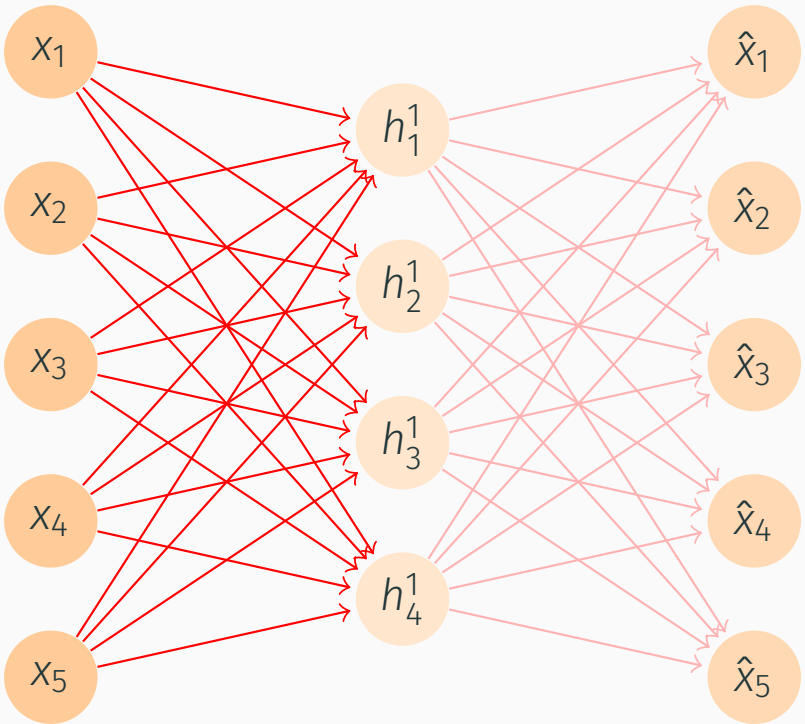
Stacked auto-encoder

Learn a stacked autoencoder by learning progressively each feature vector.



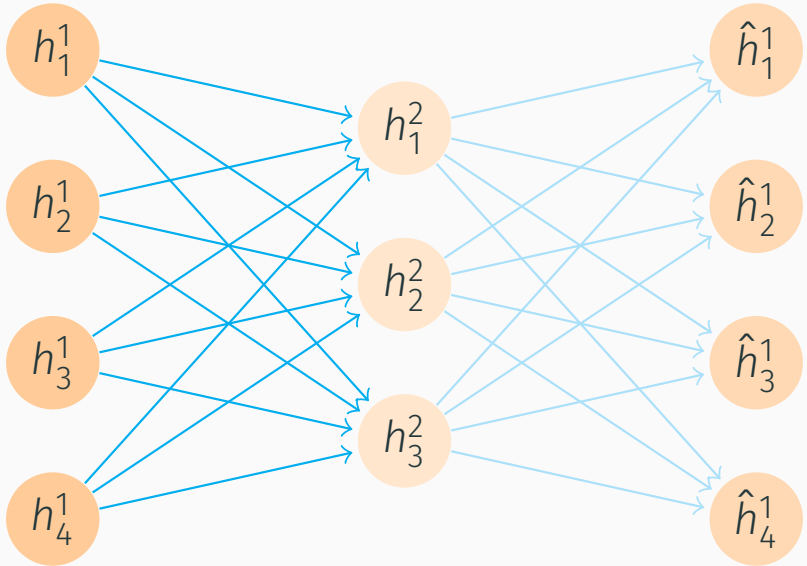
Stacked auto-encoders

Using unlabeled data, learn an auto-encoder to extract first order features.



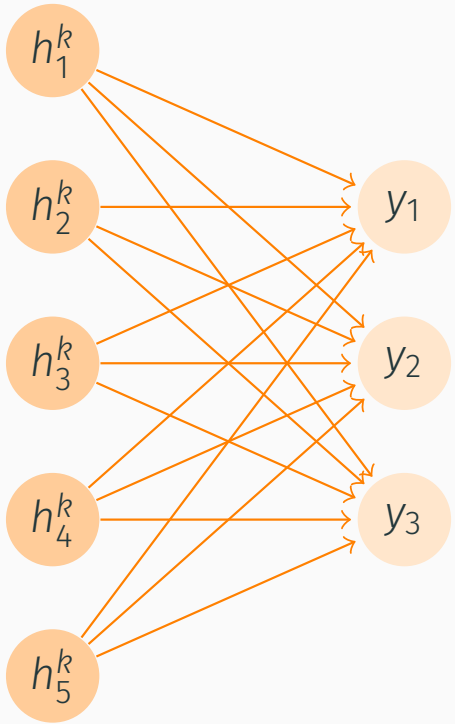
Stacked auto-encoders

Once it converged, learn another autoencoder on the same unlabeled data, but using the previous feature space as input/output layer, not the input data.

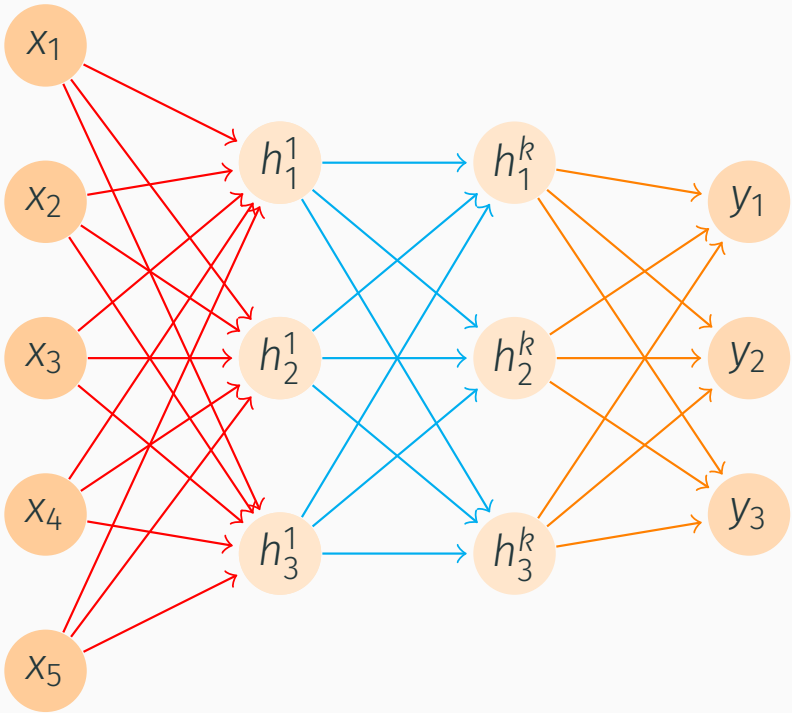


Stacked auto-encoders

Repeat the operation as often as needed, and finish with a simple classifier using the labeled data.



Stacked auto-encoders

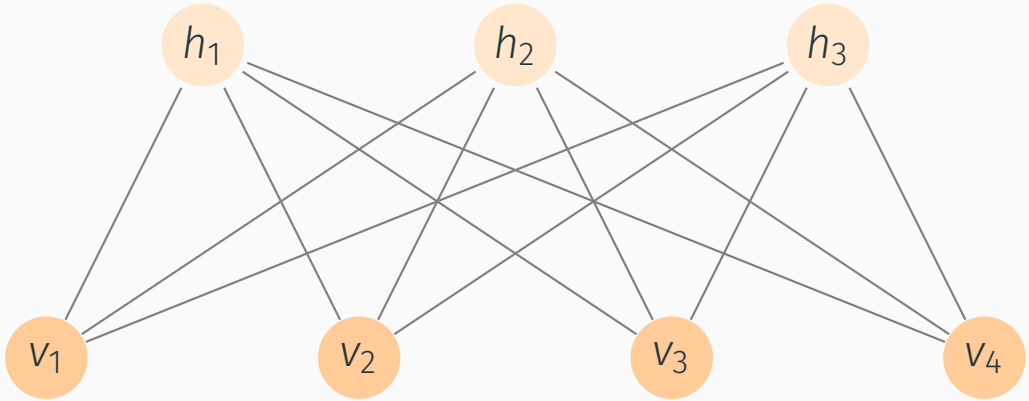


- This defines a stacked auto-encoder.
- The procedure of learning is called Greedy layer wise learning.
- Each layer progressively learns more and more complex features of the input data (edges - contours - forms - objects).
- This method can also be used to learn a deep MLP directly on labeled data: pretraining ensures that the gradient will not diffuse, as the weights are already closer to the optimal solution.

Restricted Boltzmann machines

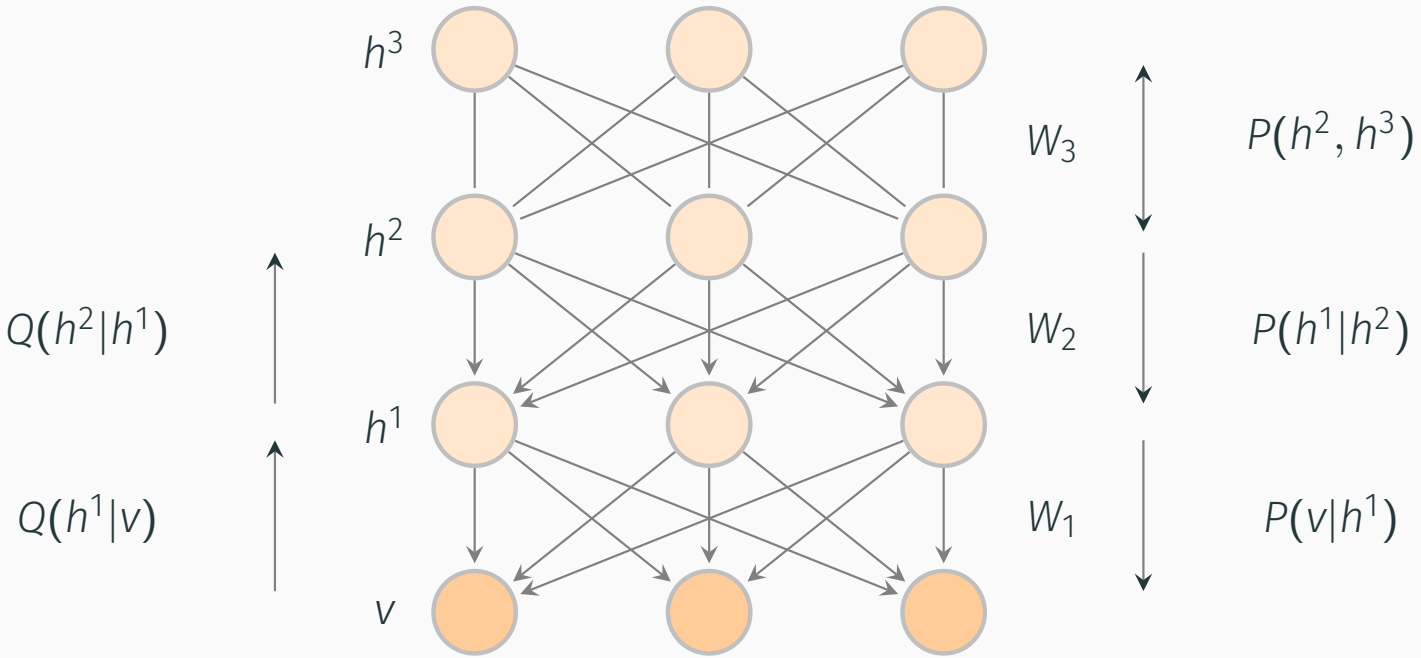
Restricted Boltzmann Machine

- Autoencoders are not the only feature extractors that can be stacked.
- Restricted Boltzmann Machines (RBM) are generative stochastic artificial neural networks that can learn a probability distribution of their inputs.
- Their neurons must form a bipartite graph with two groups of reciprocally connected units: the ”visible” and ”hidden” units.
- The goal of learning is to find the weights allowing the network to explain best the input data.
- Learning (usually with the contrastive divergence algorithm) is much more complex than for feedforward NN.
- Visible units are conditionally independent on hidden units and vice versa



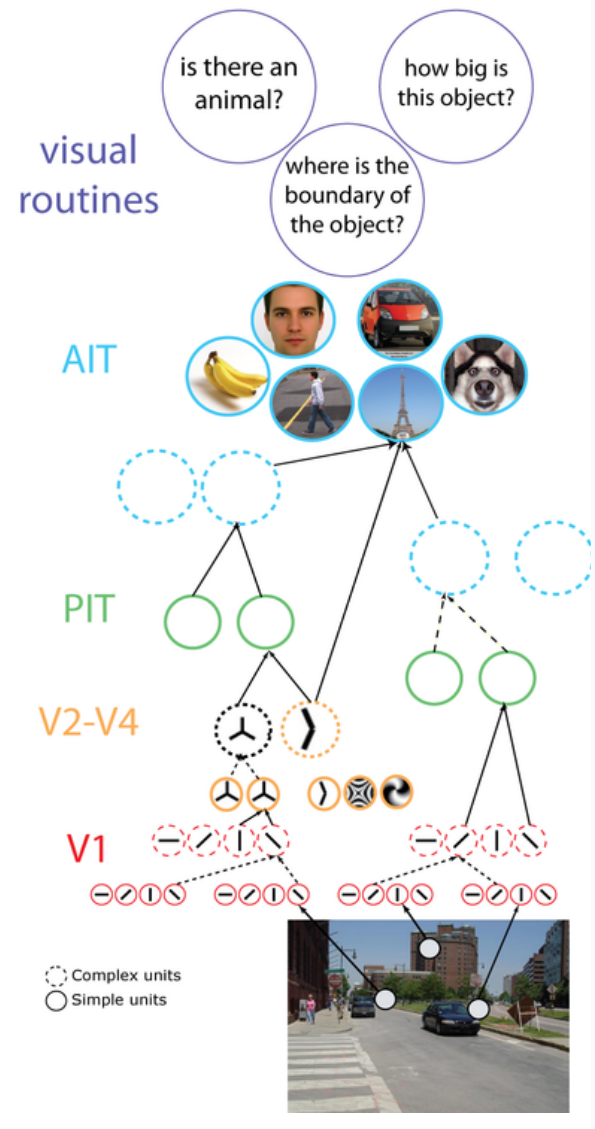
Deep Belief Network

- Top most layer is RBM
- Others are directed belief networks



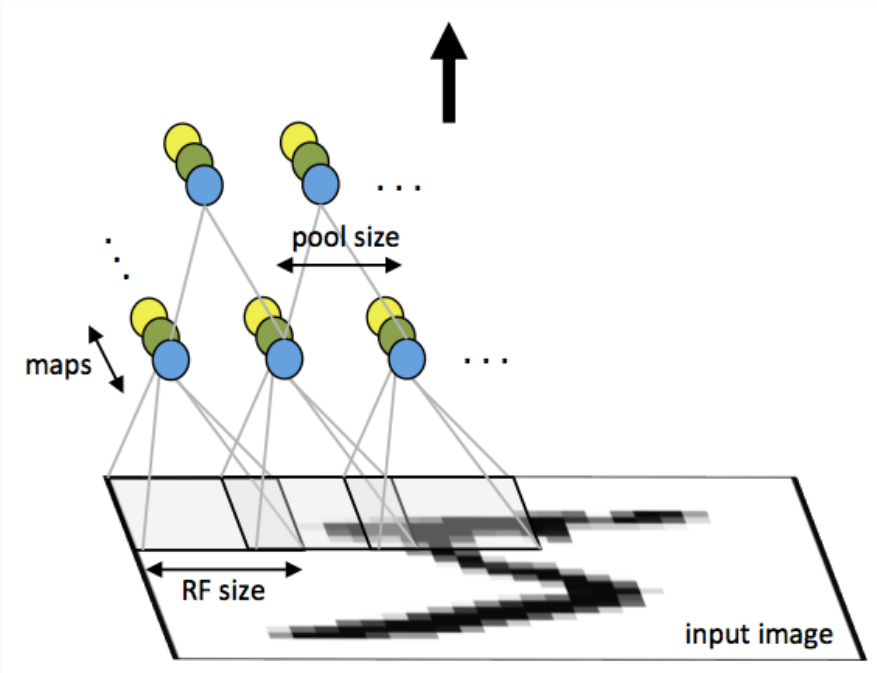
Convolutional networks

Problem with fully connected networks



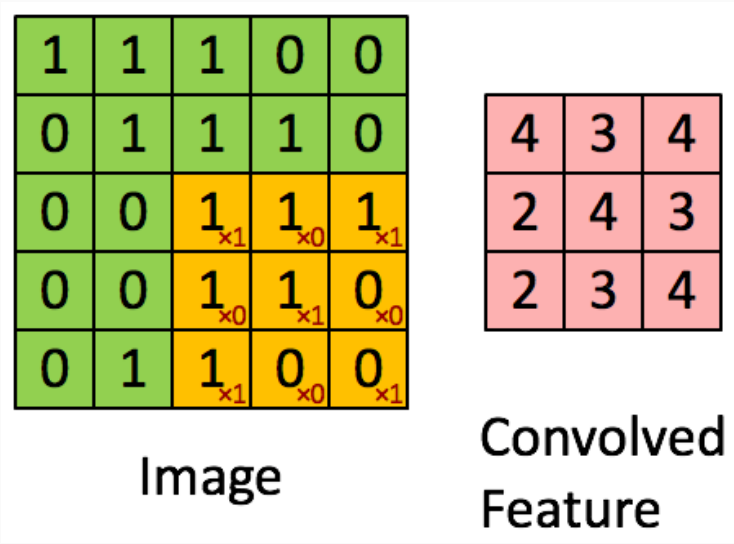
- Using full images instead of small patches may lead to an explosion of the number of weights to be learned.
- Early features (edges) are usually local, no need to learn weights from the whole image.
- Natural images are stationary: the statistics of the pixel in a 14*14 patch is the same, regardless the position on the image.
- One only needs to extract features on small patches and then convolve the resulting filter on the whole image.

Conv Nets



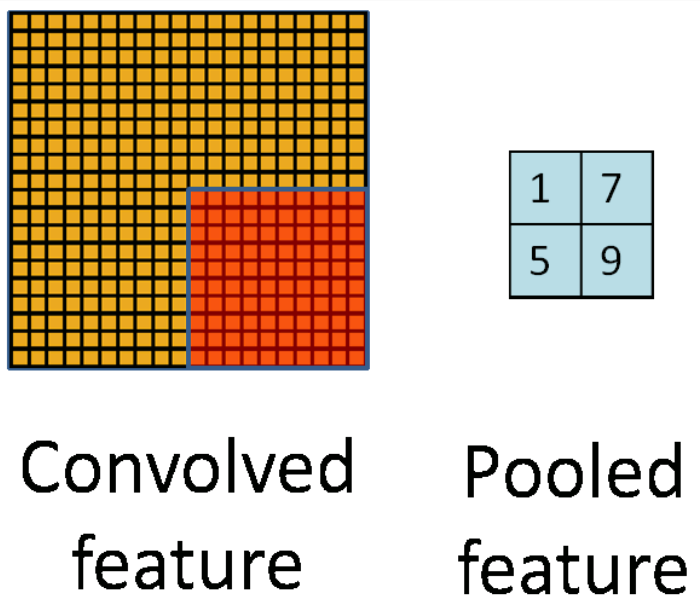
- The first features are extracted using small patches of the bigger input image.
- For example 96 * 96 images decomposed into 8 * 8 patches are composed of 89 * 89 = 7821 overlapping patches.
- If we extract 400 features per patch, that gives 3,168,400 features per image.
- The next layer must average over these overlapping features, what is called pooling.

Extracting features on patches



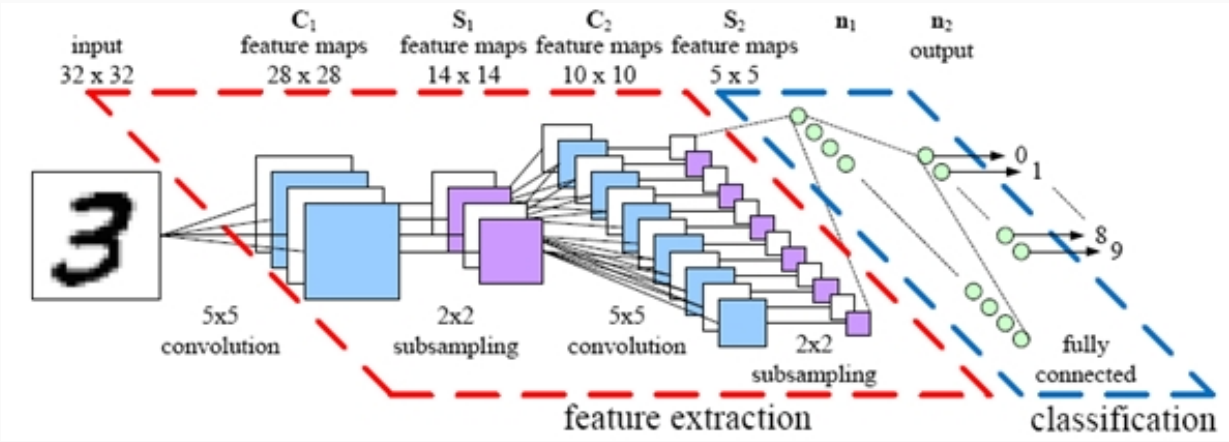
- The first features are extracted using small patches of the bigger input image.
- For example 96 * 96 images decomposed into 8 * 8 patches are composed of 89 * 89 = 7821 overlapping patches.
- If we extract 400 features per patch, that gives 3,168,400 features per image.
- The next layer must average over these overlapping features, what is called pooling.

Pooling



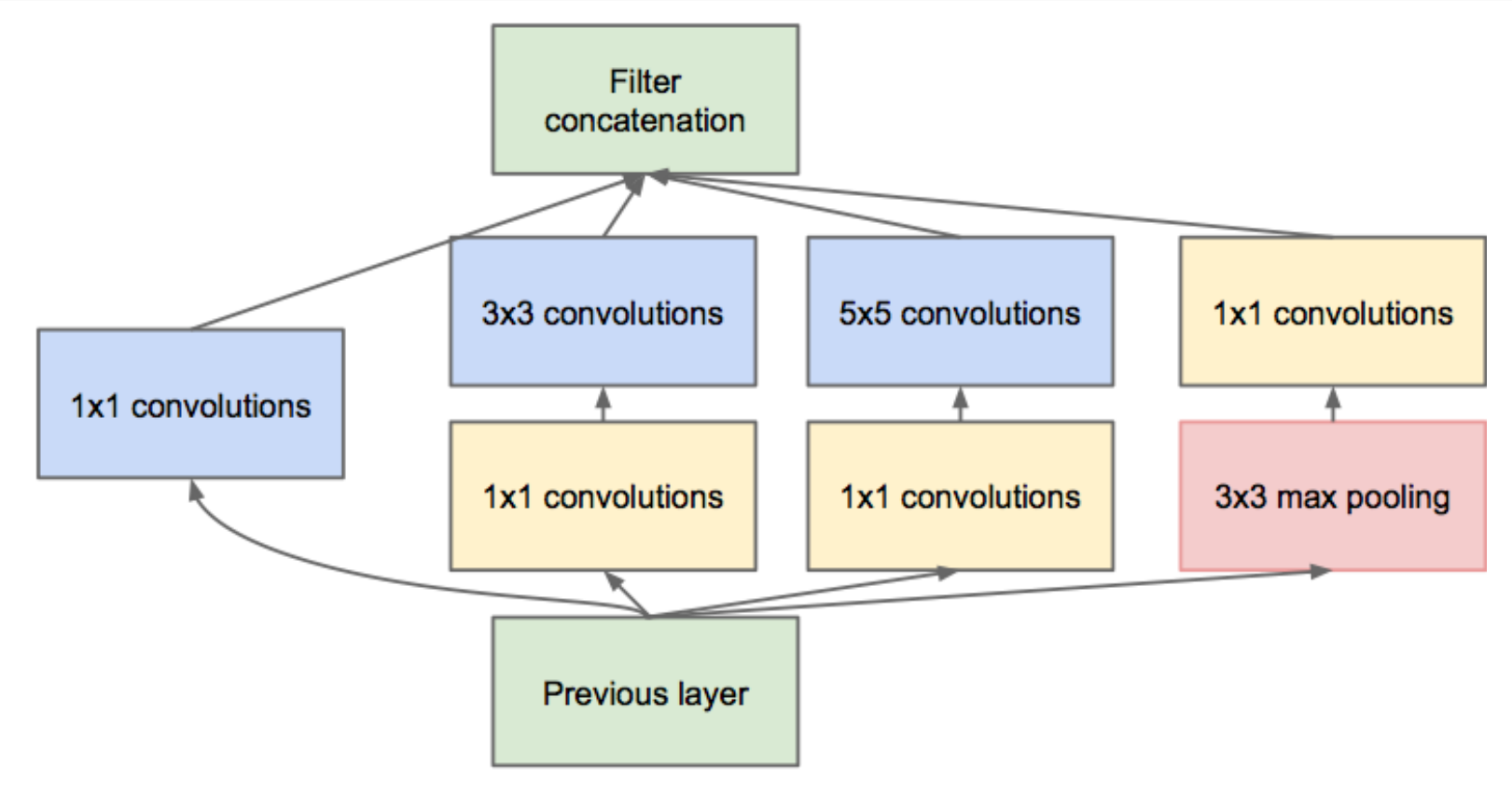
- For each feature, the pooling phase takes the maximum (or mean) for each patch in a subregion of the image.
- If a particular oriented edge is present in this subregion, the corresponding pooling feature will be active.
- Pooling allows translation invariance: the same input pattern will be detected whatever its position in the input image.

Conv Nets

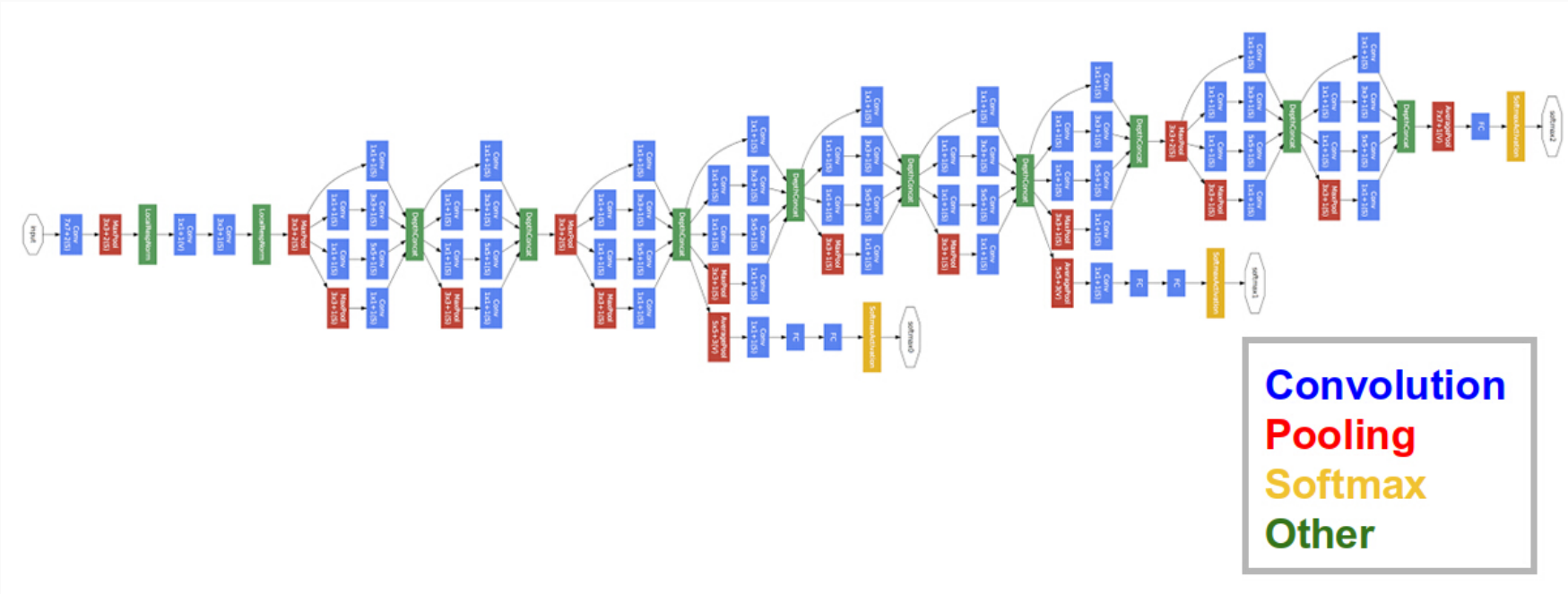


- A convolutional network is a cascade of convolution and pooling operations, extracting step by step more and more complex features.
- Convoluted networks obtain the best performance on image recognition (used for digit recognition at the US post).
- They are very easy to compute in parallel (CPU, GPU).

Inception module



GoogleNet



- 27 layers, 10M parameters
- Won the ILSVRC 2014 Classification Challenge (1.2M images, 1000 classes).
- Szegedy et al. Going Deeper with Convolutions, CVPR'14.

Applications and libraries

Applications

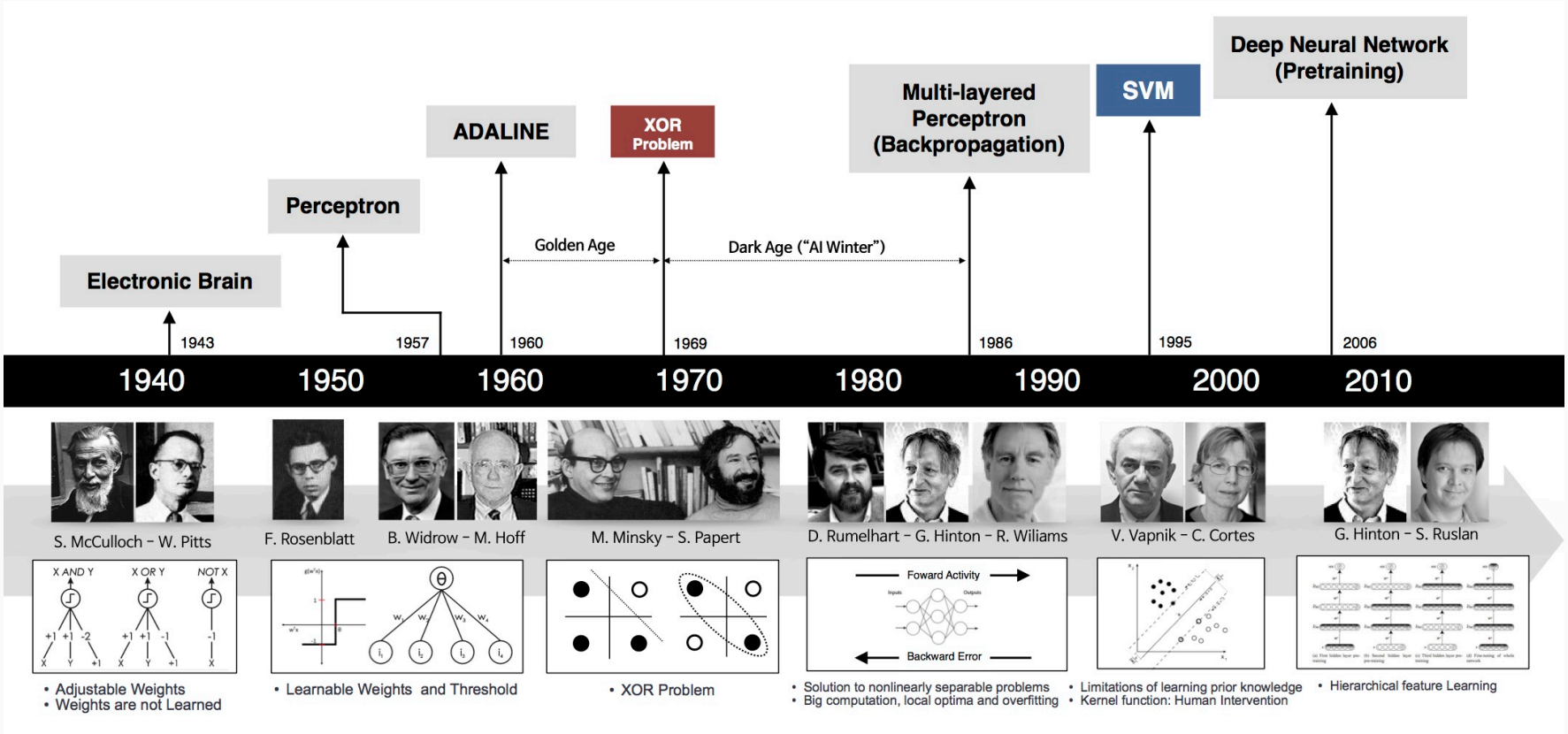
Competitions

Deep networks have won lately most ML competitions on pattern recognition

- IJCNN 2011 Traffic Sign Recognition Competition
- ICPR 2012 Mitosis Detection in Breast Cancer Histological Images
- ImageNet 2012 Large Scale Visual Recognition Challenge
- ILSVRC 2014 Classification Challenge.

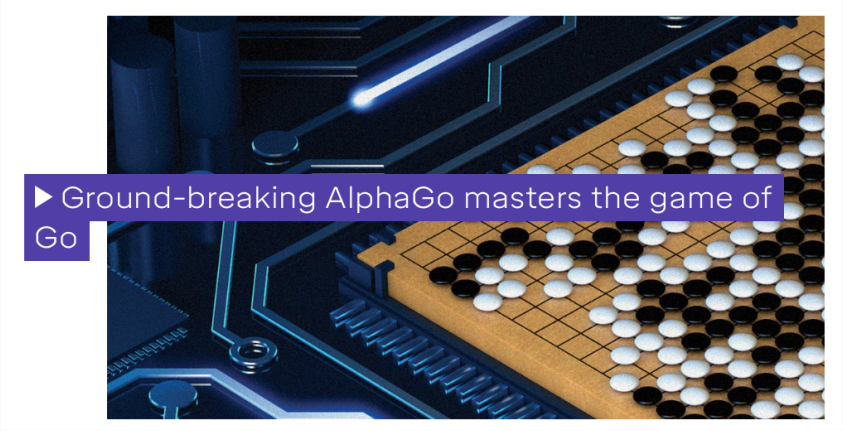
Not all deep networks use greedy layerwise learning, some are purely backpropagation based.

A little bit of history ...

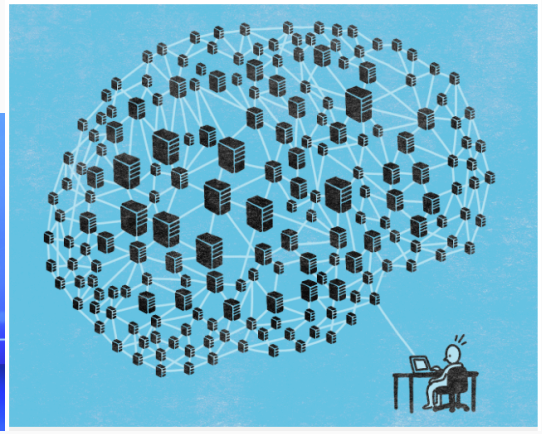


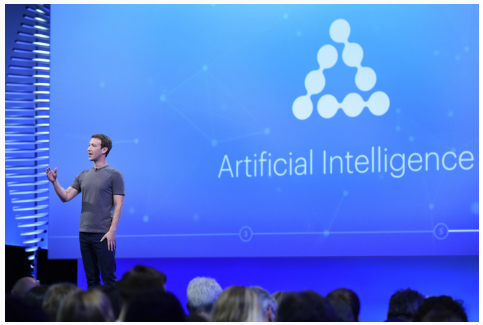
In industry

- Google DeepMind
- Facebook (FAIR)
- Microsoft
- Elon Musk (OpenAI - see `http://gym.openai.com`)
- Toyota
- many, **many** others ...



Microsoft releases CNTK, its open source deep learning toolkit, on GitHub



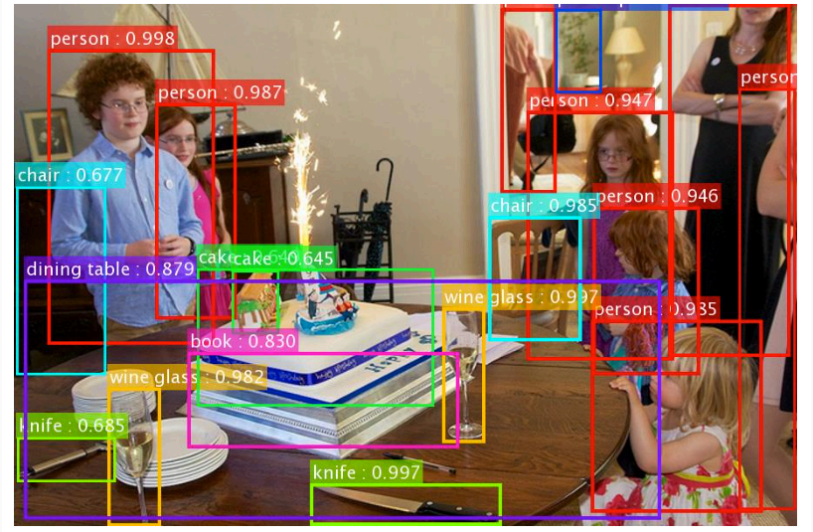


OpenAI

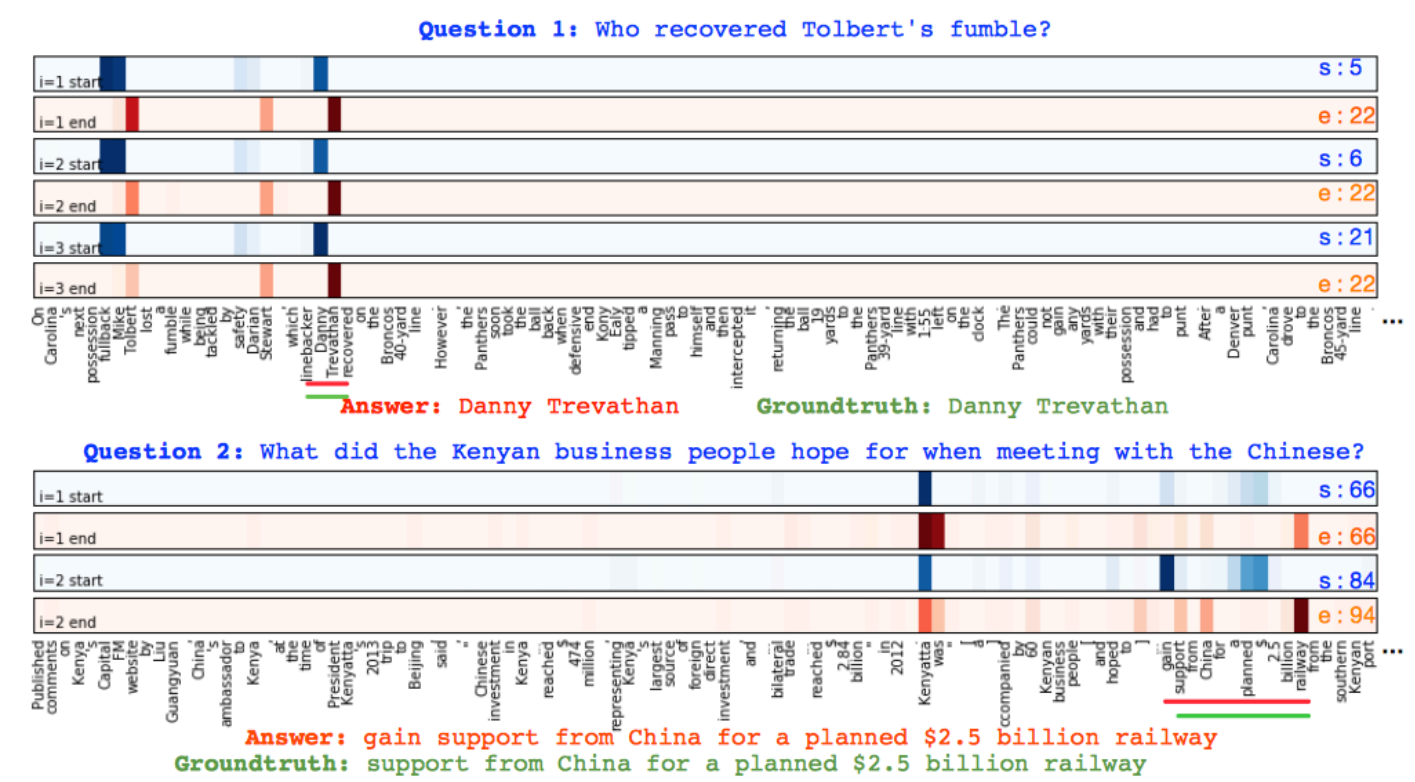
Computer vision

Captioning

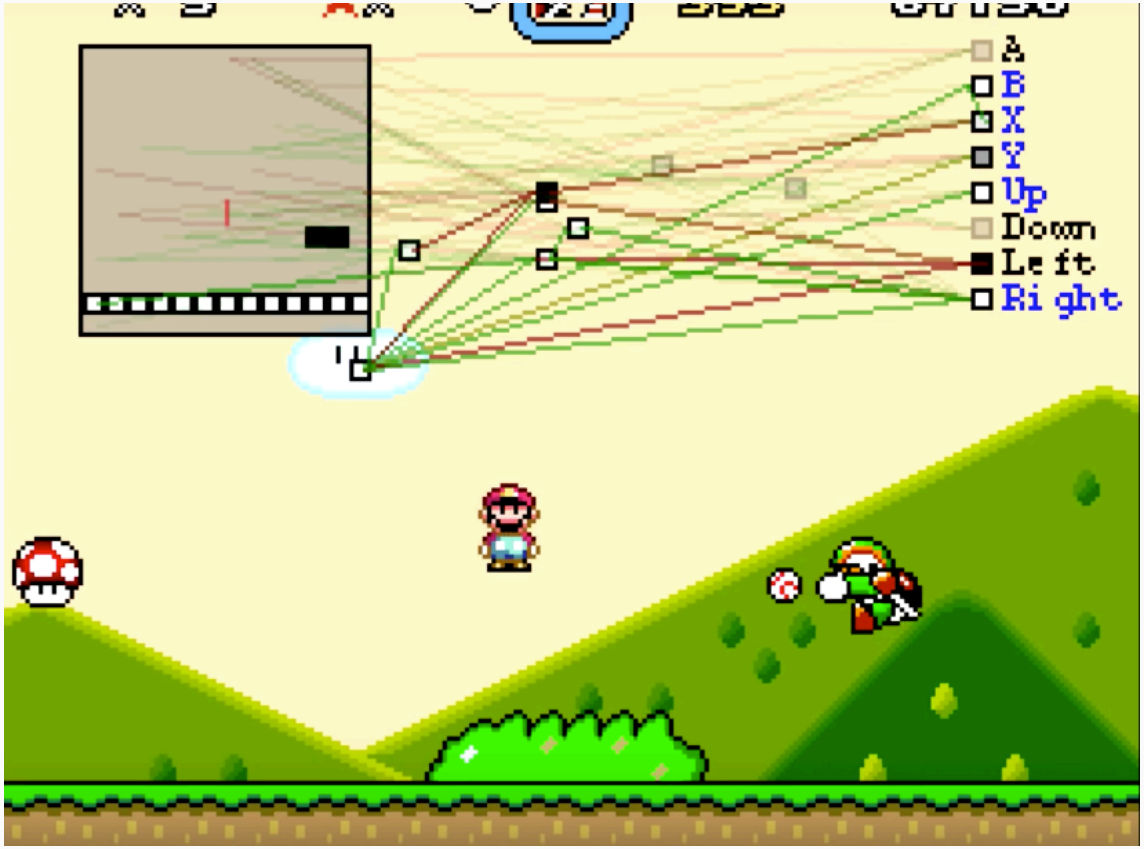
Object localization



Questions/Answers



Games



Libraries

Plateforms

- **TensorFlow** - generic library, Python
- Marvin - neural networks GPU (C++, Cuda)
- Caffe - neural networks GPU (C++, Cuda) - Computer Vision
- Theano – compiler CPU/GPU of symbolic expressions, Python
- Torch – Matlab like environement (Lua language)
- see `http://deeplearning.net/software_links/`