Artificial Neural Networks
Backpropagation
&
Deep Neural Networks

Jan Drchal

`drchajan@fel.cvut.cz`

Computational Intelligence Group
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague

Outline

- Learning MLPs: Backpropagation.
- Deep Neural Networks.

This presentation is partially inspired and uses several images and citations from Geoffrey Hinton's *Neural Networks for Machine Learning* course at Coursera.
Go through the course, it is great!

Backpropagation (BP)

- Paul Werbos,
- 1974, Harvard, PhD thesis.
- Still popular method,
- many modifications.
- **BP is a learning method for MLP:**
  - **continuous, differentiable activation functions!**

BP Overview (Online Version)

random weight initialization
repeat
  repeat *// epoch*
    choose an instance from the training set
    apply it to the network
    evaluate network outputs
    compare outputs to desired values
    modify the weights
  until all instances selected from the training set
until global error < criterion

Jan Drchal, drchajan@fel.cvut.cz, http://cig.felk.cvut.cz

ANN Energy

Backpropagation is based on a minimalization of ANN *energy* (= error). Energy is a measure describing how the network is trained on given data. For BP we define the energy function:

$$E_{TOTAL} = \sum_p E_p$$

The total sum computed over all patterns of the training set.

where

$$E_p = \frac{1}{2} \sum_{i=1}^{N_o} (d_i^o - y_i^o)^2$$

we will omit "p" in following slides

Note, ½ – only for convenience – we will see later...

ANN Energy II

The energy/error is a function of:

$$E = f\left(\vec{X}, \vec{W}\right)$$

$\vec{W}$ weights (thresholds) → **variable**,

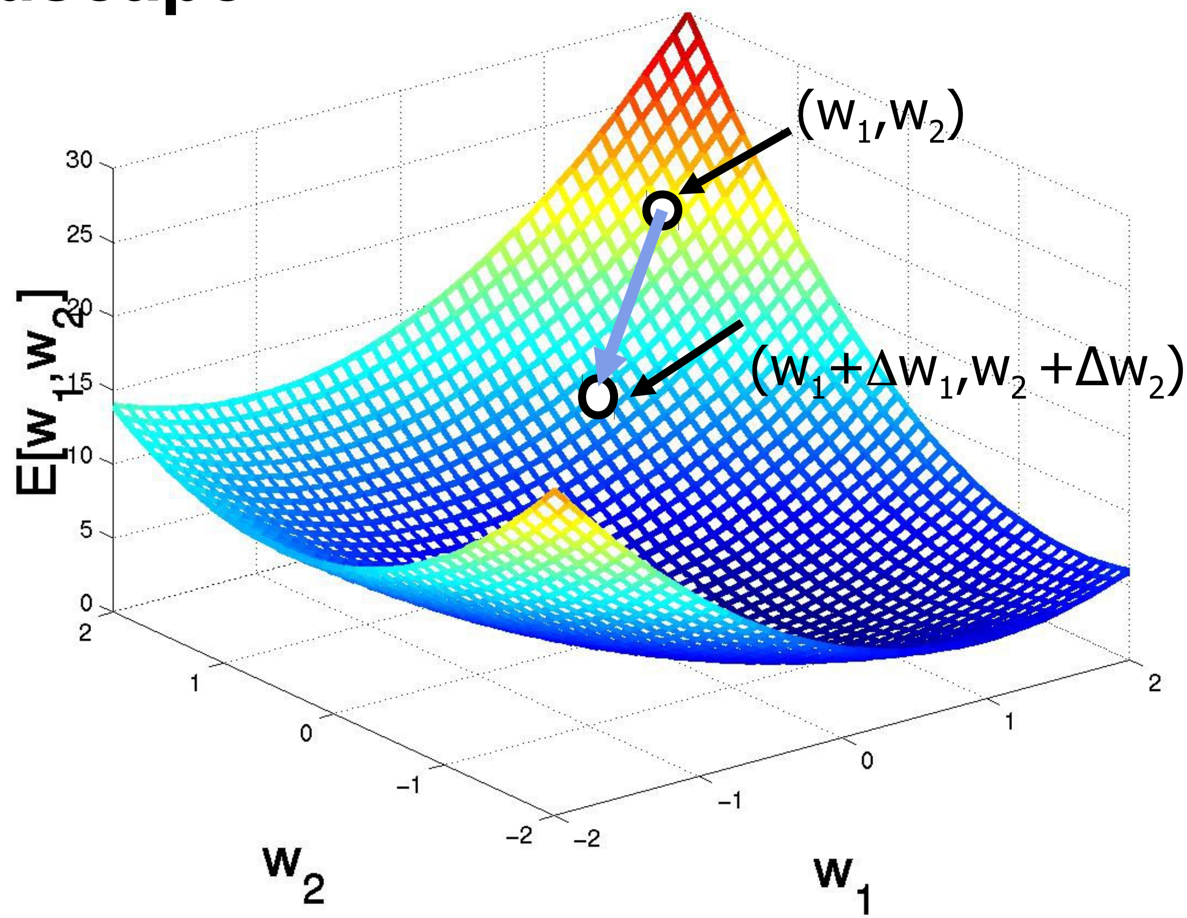
$\vec{X}$ inputs → **fixed (for given pattern)**.

Backpropagation Keynote

- For given values at network inputs we obtain an energy value.
- Our task is to minimize this value.
- The minimization is done via modification of weights and thresholds.
- We have to identify how the energy changes when a certain weight is changed by Δw.
- This corresponds to partial derivatives $\frac{\partial E}{\partial w}$.
- We employ a **gradient method**.

Gradient Descent in Energy Landscape

Energy/Error Landscape



Weight Update

- We want to update weights in opposite direction to the gradient:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}}$$

weight "delta"

learning rate

Note: gradient of energy function is a vector which contains partial derivatives for all weights (thresholds)

Notation

w_{jk} weight of connection from neuron *j* to neuron *k*

w_{0k}^{m} threshold of neuron *k* in layer *m*

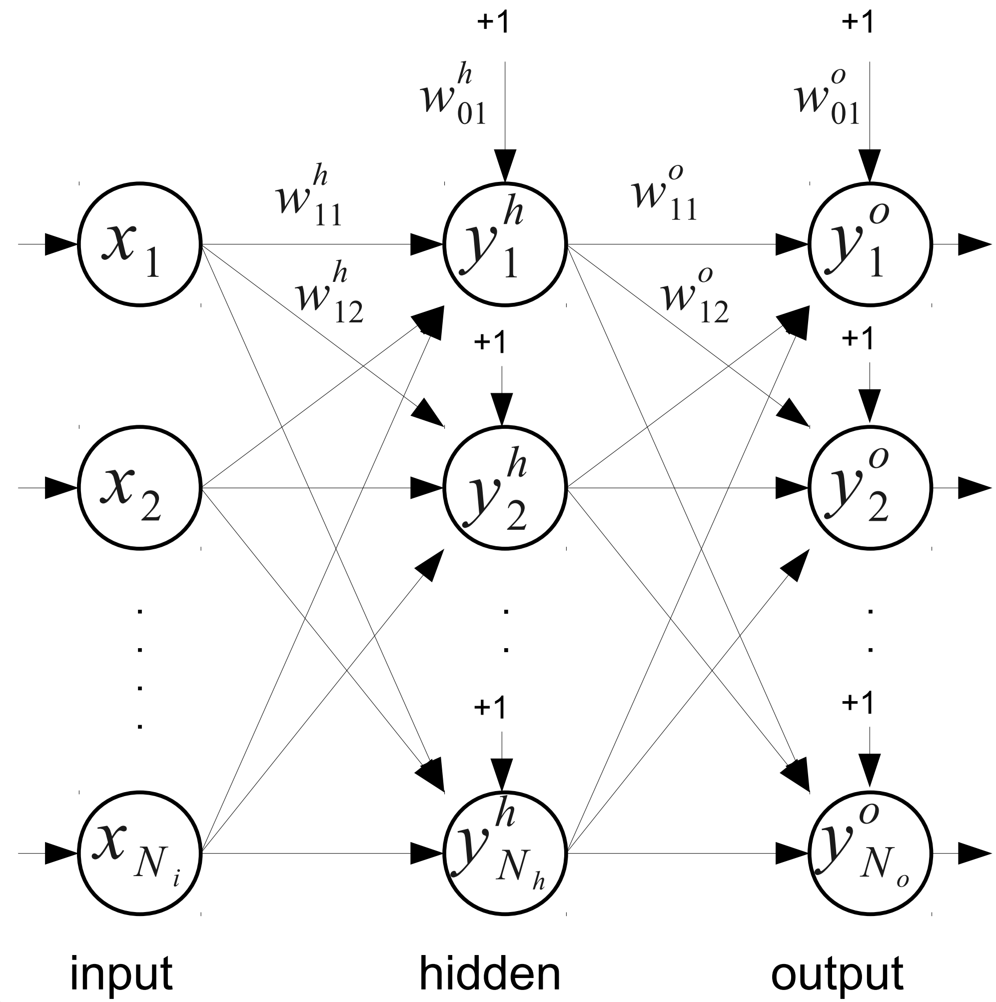
w_{jk}^{m} weight of connection from layer *m-1* to *m*

s_k^m inner potential of neuron *k* in layer *m*

y_k^m output of neuron *k* in layer *m*

x_k *k*-th input

N_i, N_h, N_o number of neurons in input, hidden and output layers



Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

$$s_k = \sum_j w_{jk} y_j$$

use

$$\frac{\partial s_k}{\partial w_{jk}} = y_j$$

Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

$$s_k = \sum_j w_{jk} y_j$$

use

denote

$$\frac{\partial s_k}{\partial w_{jk}} = y_j$$

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

Energy as a Function Composition

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{jk}}$$

$$s_k = \sum_j w_{jk} y_j$$

use

denote

$$\frac{\partial s_k}{\partial w_{jk}} = y_j$$

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

$$\Delta w_{jk} = \eta \delta_k y_j$$

Remember the delta rule?

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \xrightarrow{\text{output layer}} \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \xrightarrow{\text{output layer}} \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o} \frac{\partial y_k^o}{\partial s_k^o}$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

output layer →

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o} \frac{\partial y_k^o}{\partial s_k^o}$$

derivative of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

output layer →

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

derivative of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

output layer →

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency of energy on a network output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

derivate of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

output layer →

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency of energy on a network output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

That is why we used the ½ in energy definition.

derivate of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Jan Drchal, drchajan@fel.cvut.cz, http://cig.felk.cvut.cz

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

output layer

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency of energy on a network output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

That is why we used 1/2.

Again, remember the delta rule?

derivate of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

Output Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \quad \xrightarrow{\text{output layer}} \quad \delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

$$E = \frac{1}{2}\sum_{i=1}^{N_o}(d_i^o - y_i^o)^2$$

$$\frac{\partial E}{\partial s_k^o} = \frac{\partial E}{\partial y_k^o}\frac{\partial y_k^o}{\partial s_k^o}$$

use

dependency of energy on a network output

$$\frac{\partial E}{\partial y_k^o} = -(d_k^o - y_k^o)$$

That is why we used 1/2.

Again, remember the delta rule?

derivate of activation function

$$\frac{\partial y_k^o}{\partial s_k^o} = S'(s_k^o)$$

$$\Delta w_{jk}^o = \eta\,\delta_k^o\,y_j^h = \eta(d_k - y_k^o)S'(s_k^o)\,y_j^h$$

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \xrightarrow{\text{hidden layer}} \delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k} \quad \xrightarrow{\text{hidden layer}} \quad \delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

hidden layer →

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

$$\frac{\partial y_k^h}{\partial s_k^h} = S'(s_k^h)$$

Same as output layer.

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

hidden layer →

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

Note, this is output of a **hidden** neuron.

$$\frac{\partial y_k^h}{\partial s_k^h} = S'(s_k^h)$$

Same as output layer.

Hidden Layer

$$\delta_k = -\frac{\partial E}{\partial s_k}$$

hidden layer →

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h}$$

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h}$$

Note, this is output of a **hidden** neuron.

? let's look at this partial derivation

$$\frac{\partial y_k^h}{\partial s_k^h} = S'(s_k^h)$$

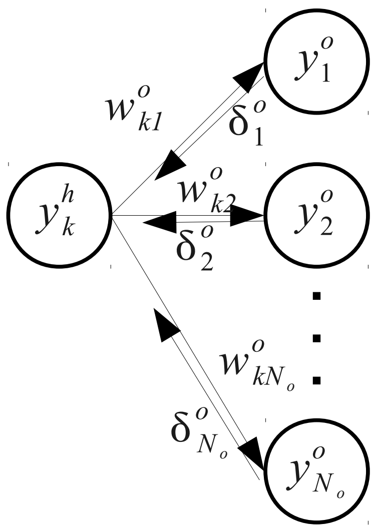
Same as output layer.

Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

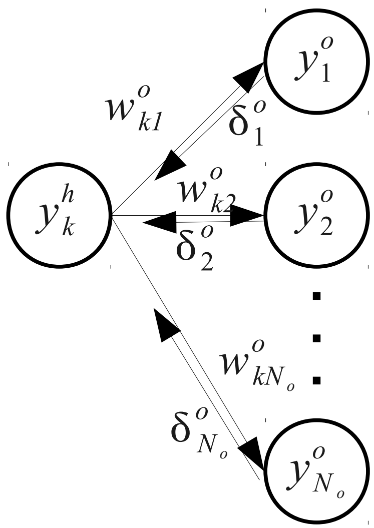
$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = -\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$



Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).



$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = - \sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$

But we know this already.

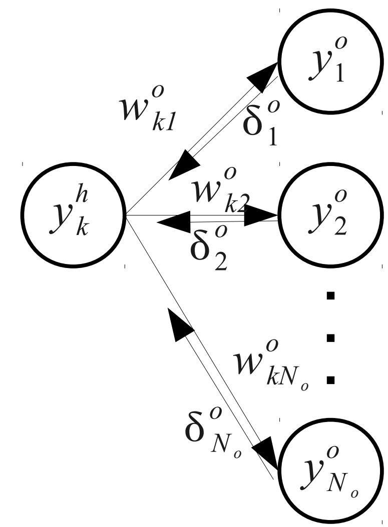
$$\delta_k^o = - \frac{\partial E}{\partial s_k^o}$$

Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = - \sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$



But we know this already.

$$\delta_k^o = - \frac{\partial E}{\partial s_k^o}$$

Take the error of the output neuron

and multiply it by the input weight.

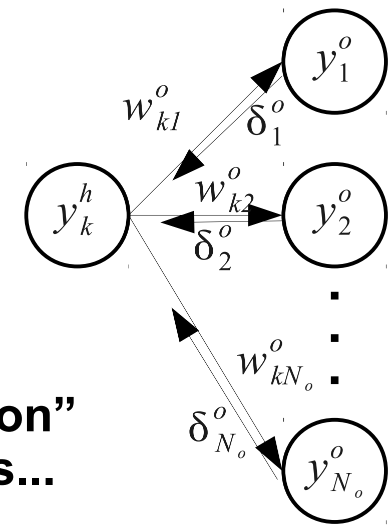
Hidden Layer II

$$\frac{\partial E}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial s_l^o}{\partial y_k^h} = \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} \frac{\partial}{\partial y_k^h} \left(\sum_{i=1}^{N_h} w_{il}^o y_i^h \right) =$$

Apply the chain rule
(http://en.wikipedia.org/wiki/Chain_rule).

$$= \sum_{l=1}^{N_o} \frac{\partial E}{\partial s_l^o} w_{kl}^o = -\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o$$

Here the "back-propagation" actually happens...



But we know this already.

$$\delta_k^o = -\frac{\partial E}{\partial s_k^o}$$

Take the error of the output neuron

and multiply it by the input weight.

Hidden Layer III

Now, let's put it all together!

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h} = -\left(\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o \right) S'\left(s_k^h \right)$$

Hidden Layer III

Now, let's put it all together!

$$\frac{\partial E}{\partial s_k^h} = \frac{\partial E}{\partial y_k^h} \frac{\partial y_k^h}{\partial s_k^h} = -\left(\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o \right) S'(s_k^h)$$

$$\delta_k^h = -\frac{\partial E}{\partial s_k^h} = \left(\sum_{l=1}^{N_o} \delta_l^o w_{kl}^o \right) S'(s_k^h)$$

Hidden Layer III

Now, let's put it all together!

$$\frac{\partial E}{\partial s_k^h}=\frac{\partial E}{\partial y_k^h}\frac{\partial y_k^h}{\partial s_k^h}=-\left(\sum_{l=1}^{N_o}\delta_l^o w_{kl}^o\right)S'(s_k^h)$$

$$\delta_k^h=-\frac{\partial E}{\partial s_k^h}=\left(\sum_{l=1}^{N_o}\delta_l^o w_{kl}^o\right)S'(s_k^h)$$

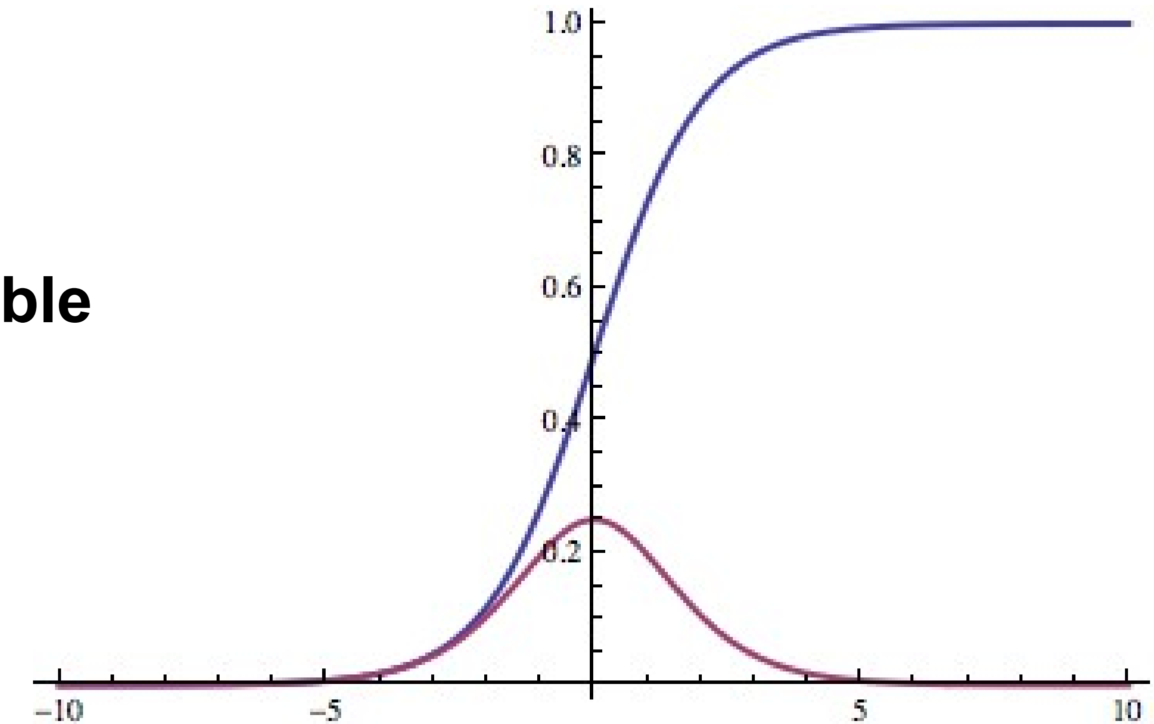
The derivation of the activation function is the last thing to deal with!

$$\Delta w_{jk}^h=\eta\delta_k^h x_j=\eta\left(\sum_{l=1}^{N_o}\delta_l^o w_{kl}^o\right)S'(s_k^h)x_j$$

Sigmoid Derivation

$$S'(s_k) = \left(\frac{1}{1+e^{-\gamma s_k}} \right)' = \frac{\gamma}{1+e^{-\gamma s_k}} \frac{e^{-\gamma s_k}}{1+e^{-\gamma s_k}} = \gamma\, y_k (1 - y_k)$$

That is why we needed continuous & differentiable activation functions!



BP Put All Together

Output layer:

$$\Delta w_{jk}^{o} = \eta \gamma y_k^o (1 - y_k^o)(d_k - y_k^o) y_j^h$$

This is equal to x_j when we get to inputs.

Hidden layer *m* (note that *h+1 = o*):

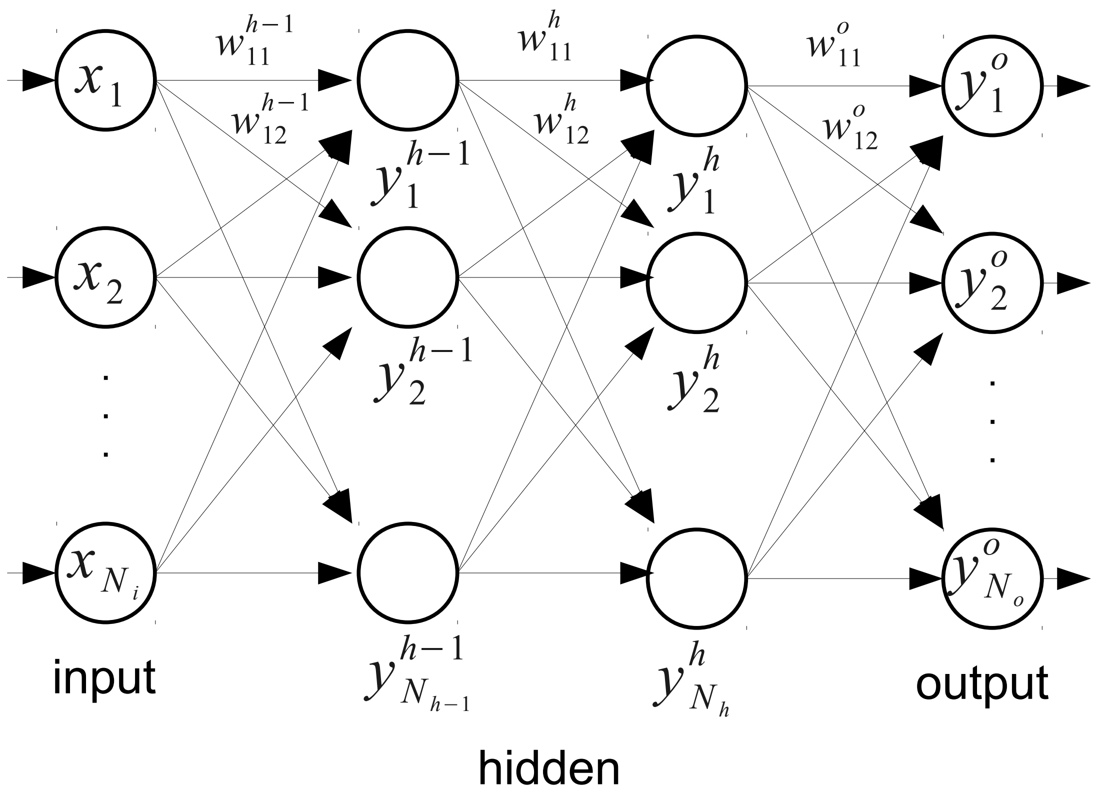
$$\Delta w_{jk}^{m} = \eta \delta_k^m y_j^{m-1} = \eta \gamma y_k^m (1 - y_k^m) \left(\sum_{l=1}^{N_{m+1}} \delta_l^{m+1} w_{kl}^{m+1} \right) y_j^{m-1}$$

Weight (threshold) updates:

$$w_{jk}(t+1) = w_{jk}(t) + \Delta w_{jk}(t)$$

How About General Case?

- Arbitrary number of hidden layers?



- It's the same: for layer *h-1* use δ_k^h.

Potential Problems

- High dimension of weight (threshold) space.
- Complexity of energy function:
  - multimodality,
  - large plateaus & narrow peaks.
- Many layers: back-propagated error signal vanishes, see later...

Weight Updates

- When to apply delta weights?
- **Online learning/Stochastic Gradient Descent (SGD)**: after each training pattern.
- **(Full) Batch learning**: apply average/sum of delta weights after sweeping through the whole training set.
- **Mini-batch learning**: after a small sample of training patterns.

Momentum

- Simple, but greatly helps when avoiding local minima:

$$\Delta w_{ij}(t) = \eta \delta_j(t) y_i(t) + \alpha \Delta w_{ij}(t-1)$$

momentum constant: $\alpha \in [0,1)$

- Analogy: a ball (parameter vector) rolling down a hill (error landscape).

Resilient Propagation (RPROP)

- Motivation: magnitude of gradient differ a lot for different weights in practise.
- RPROP does not use gradient value – the step size for each weight is adapted using its sign, only.
- Method:
    - increase the step size for a weight if the signs of the last two partial derivatives agree (e.g., multiply by 1.2),
    - decrease (e.g., multiply by 0.5) otherwise,
    - limit step size (e.g., $[10^{-6} - 50.0]$).

Resilient Propagation (RPROP) II.

- Read: *Igel, Hüsken: Improving the Rprop Learning Algorithm, 2000.*
- Good news:
  - typically faster by an order of magnitude than plain BPROP,
  - robust to parameter settings,
  - no learning rate parameter.
- Bad news: works for full batch learning only!

Resilient Propagation (RPROP) III.

- Why not mini-batches?
  - weight gets 9 times a gradient of +0.1,
  - and once a gradient of -0.9 (the tenth mini-batch),
  - we expect it to stay roughly where it was at the beginning,
  - but it will grow a lot (assuming adaptation of the step size is small)!
  - This example is due to Geoffrey Hinton (Neural Networks for Machine Learning, Coursera)

Other Methods

- Quick Propagation (QUICKPROP)
  - based on Newton's method
  - second-order approach.
- Levenberg–Marquardt
  - combines Gauss-Newton algorithm and Gradient Descent.

Other Approaches Based on Numerical Optimization

- Compute partial derivatives over the total energy:

$$\frac{\partial E_{TOTAL}}{\partial w_{jk}}$$

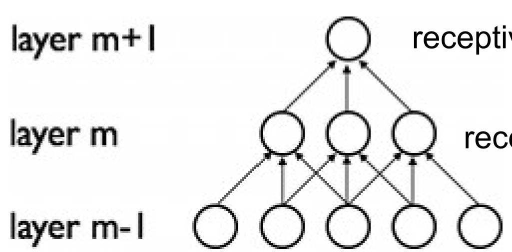
  and use **any** numerical optimization method, i.e.:
    - Conjugated gradients,
    - Quasi-Newton methods,
    - ...

Deep Neural Networks (DNNs)

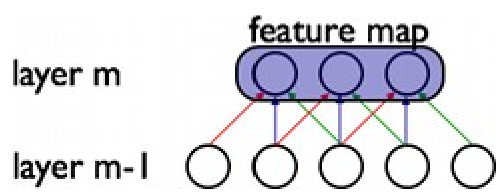
- ANNs having *many* layers: complex nonlinear transformations.
- DNNs are hard to train:
  - many parameters,
  - vanishing (exploding) gradient problem: back-propagated signal gets quickly reduced.
- Solutions:
  - reduce connectivity/share weights,
  - use large training sets (to prevent overfitting),
  - unsupervised pretraining.

Convolutional Neural Networks (CNNs)

- Feed-forward architecture using convolutional, pooling and other layers.
- Based on visual cortex research: receptive field.
- Fukushima: NeoCognitron (1980)
- Yann LeCun: LeNet-5 (1998)



sparse connectivity



shared weights

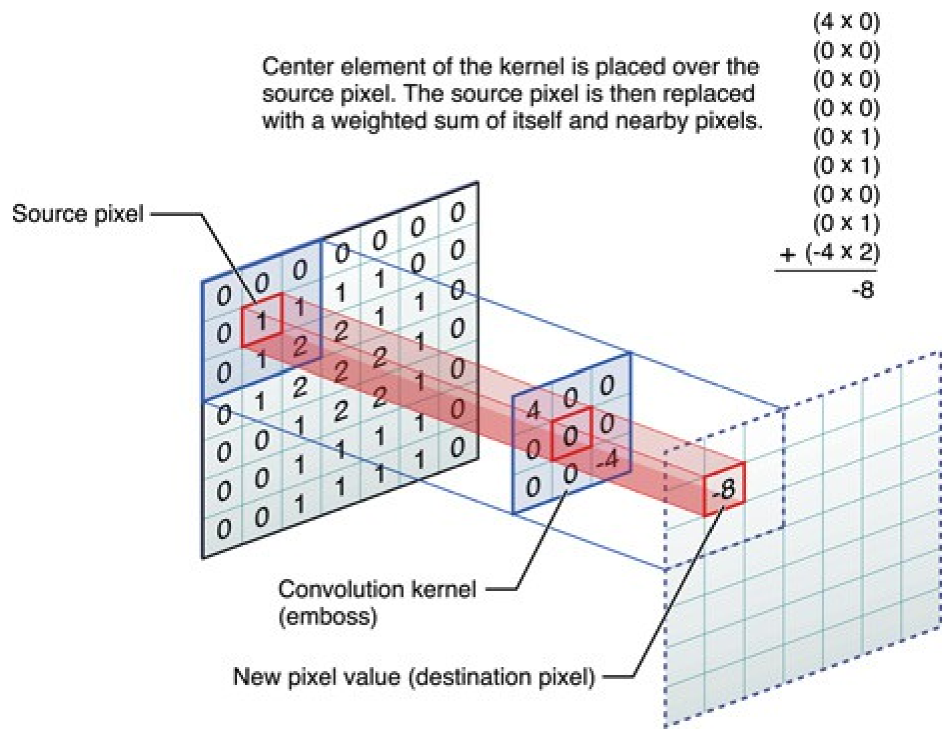
Detect features regardless of their position in the visual field.

Images from http://deeplearning.net/tutorial/lenet.html

BACKPROP for Shared Weights

- For two weights $w_1 = w_2$
- we need $\Delta w_1 = \Delta w_2$
- Compute $\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}$
- Use $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$ or average.

Convolution Kernel



Pooling

- Reducing dimensionality.
- Max-pooling is the method of choice.
- Problem: After several levels of pooling, we lose information about the precise positions.

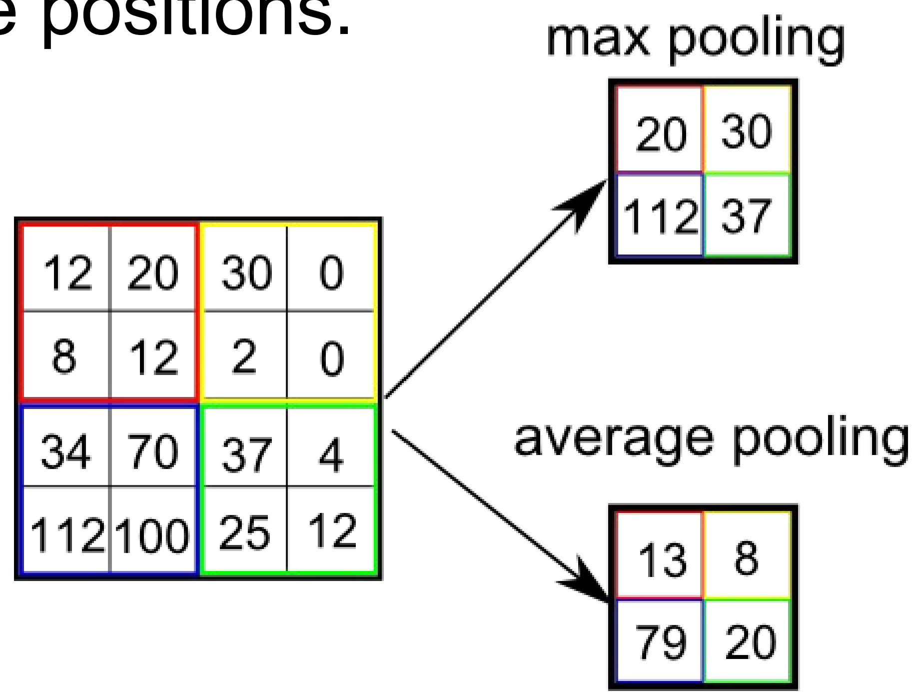
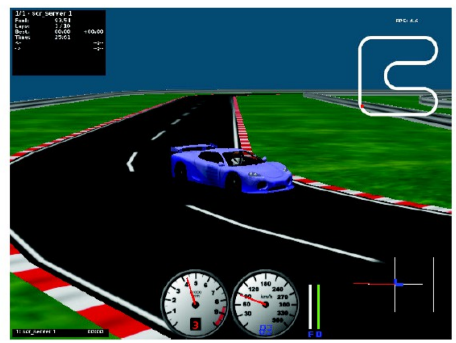
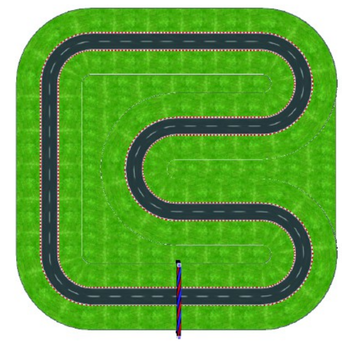


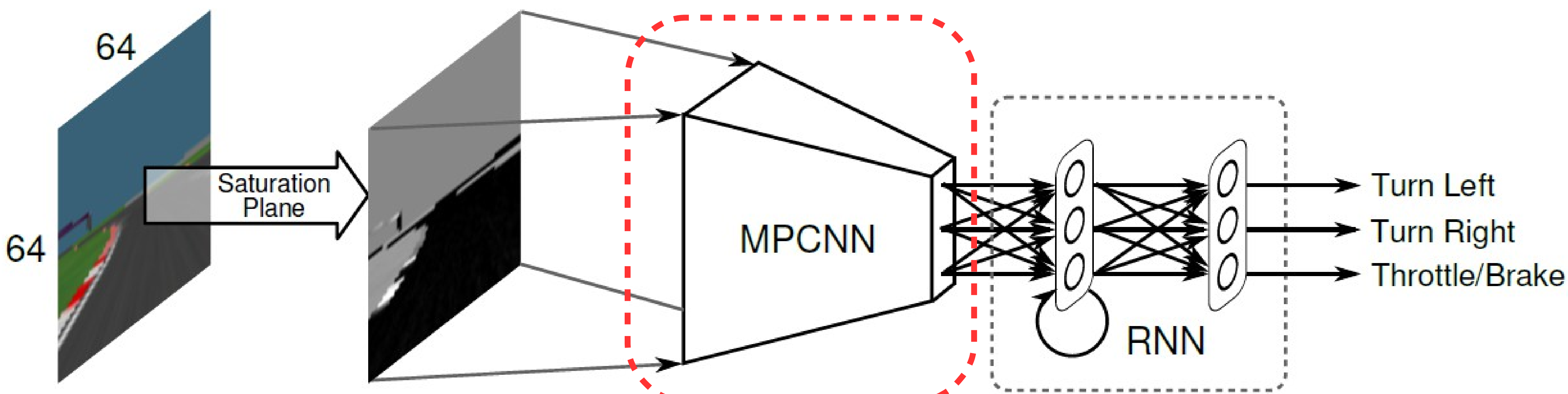
Image from http://vaaaaaanquish.hatenablog.com

CNN Example

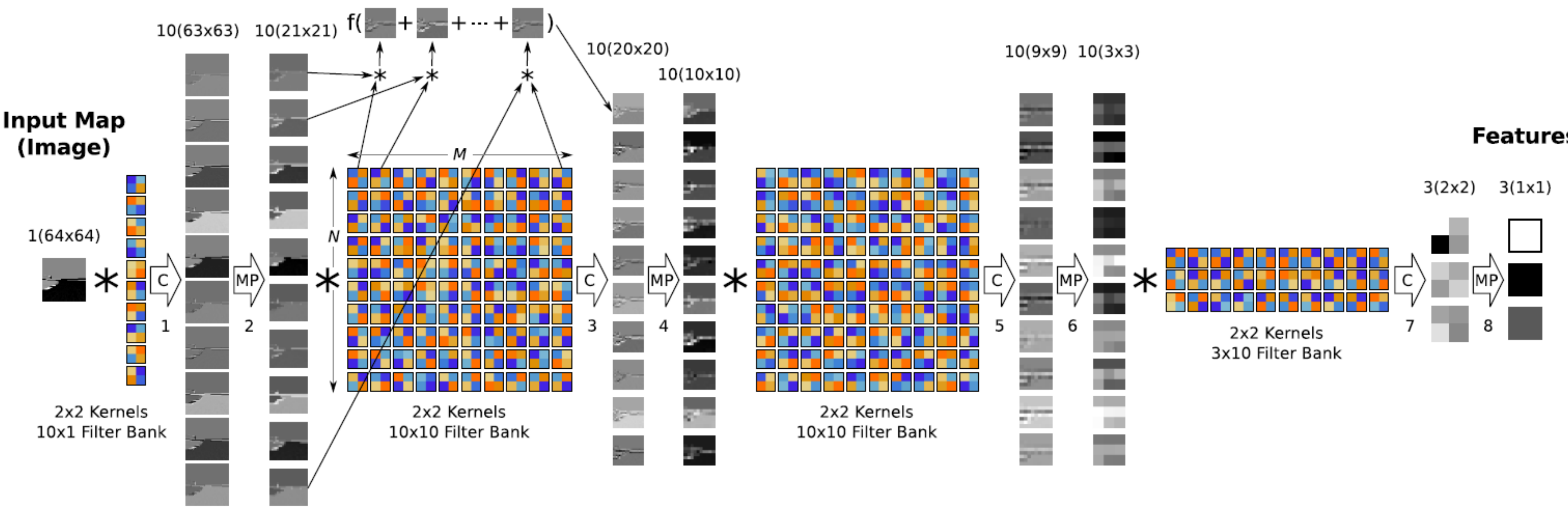
- TORCS: Koutnik, Gomez, Schmidhuber: Evolving deep unsupervised convolutional networks for vision-based RL, 2014.







CNN Example II.



Images from Koutnik, Gomez, Schmidhuber: Evolving deep unsupervised convolutional networks for vision-based RL, 2014.

CNN LeNet5: Architecture for MNIST

- MNIST: written character recognition dataset.
- See http://yann.lecun.com/exdb/mnist/
- training set 60,000, testing set 10,000 examples.

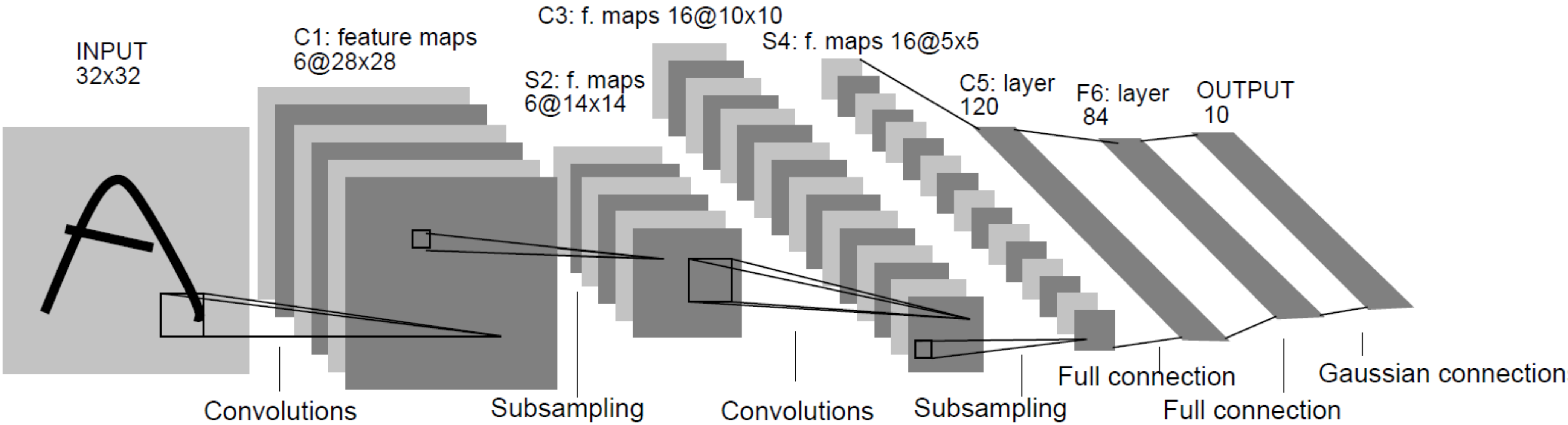


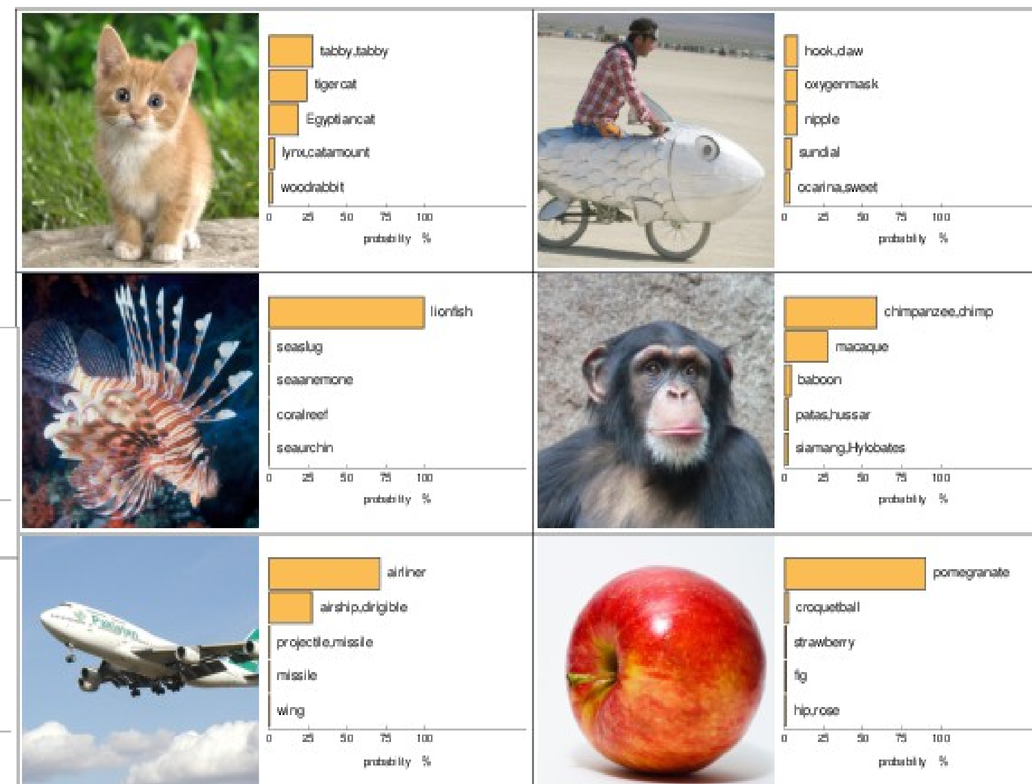
Image by LeCun et al.: Gradient-based learning applied to document recognition, 1998.

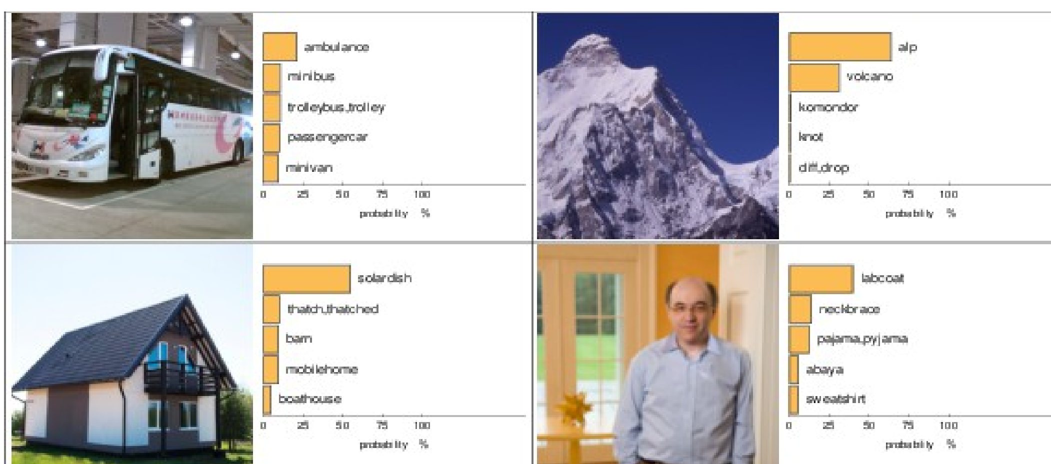
Errors by LeNet5

- 82 errors (can be reduced to about 30).
- Human error rate would be about 20 to 30.

ImageNet

- Dataset of high-resolution color images.
- Based on Large Scale Visual Recognition Challenge 2012 (ILSVRC2012).
- 1,200,000 training examples, 1000 classes.
- 23 layers!

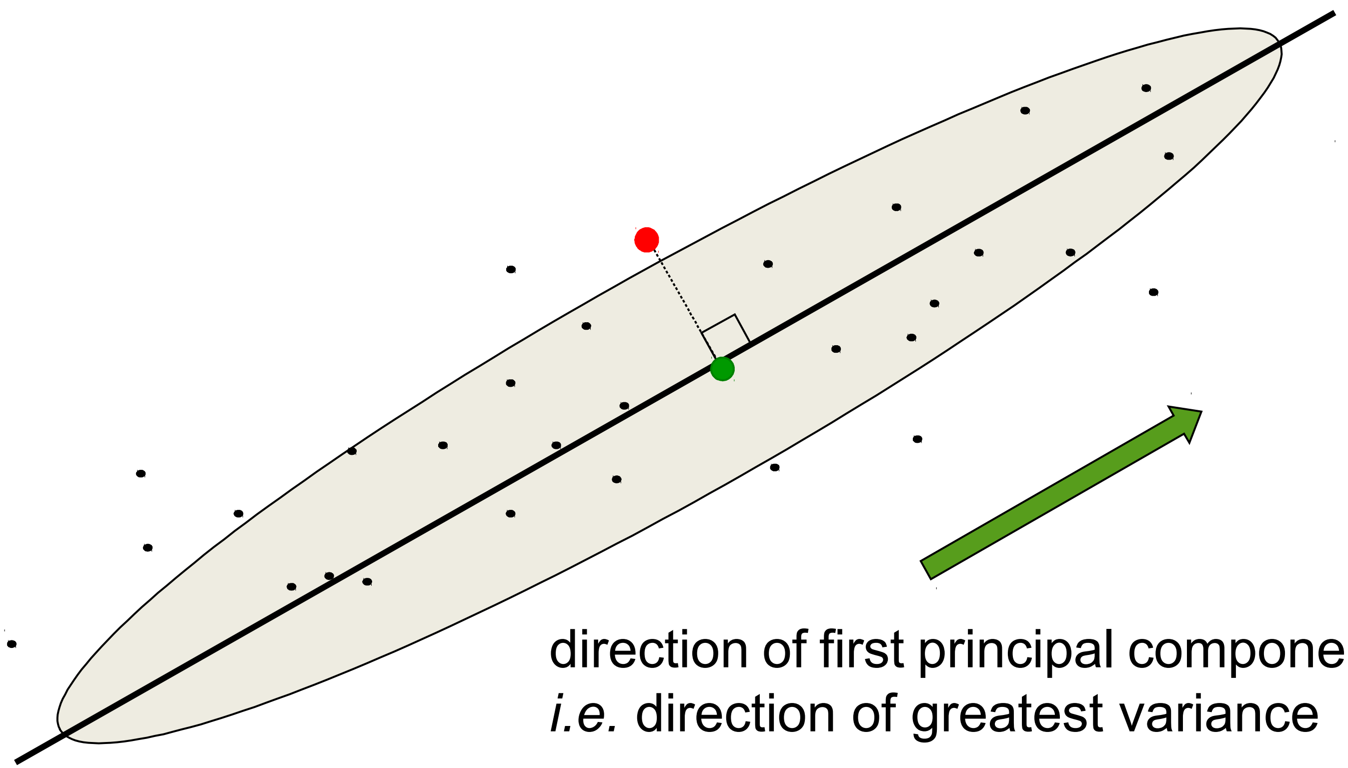




Principal Components Analysis (PCA)

- Take *N*-dimensional data,
- Find *M* orthogonal directions in which the data have the most variance.
- *M* principal directions: a lower-dimensional subspace.
- Linear projection with dimensionality reduction at the end.

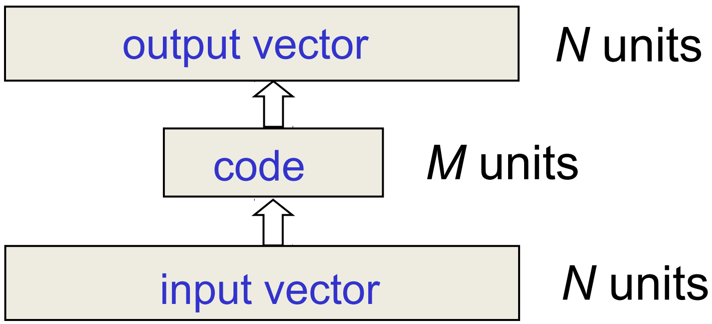
PCA with *N*=2 and *M*=1



direction of first principal component
i.e. direction of greatest variance

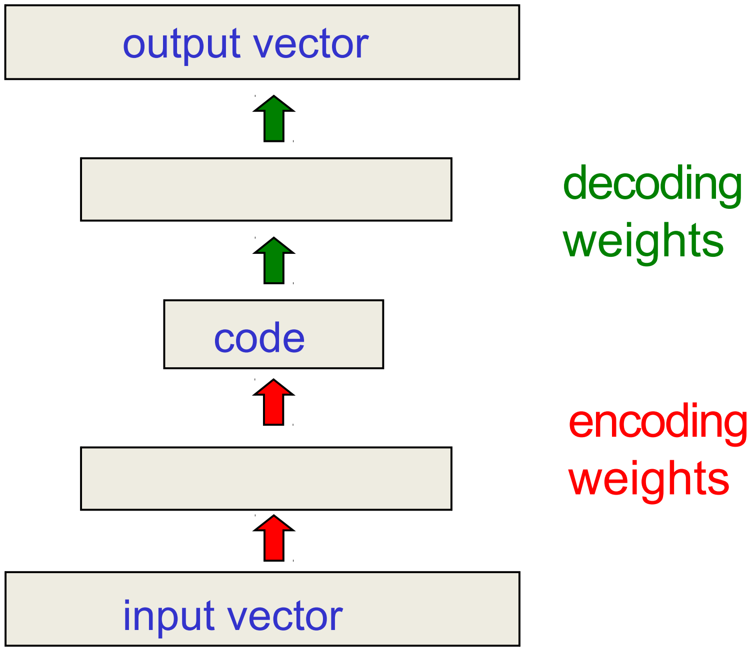
PCA by MLP with BACKPROP (inefficiently)

- Linear hidden & output layers.
- The *M* hidden units will span the same space as the first *M* components found by PCA



Generalize PCA: Autoencoder

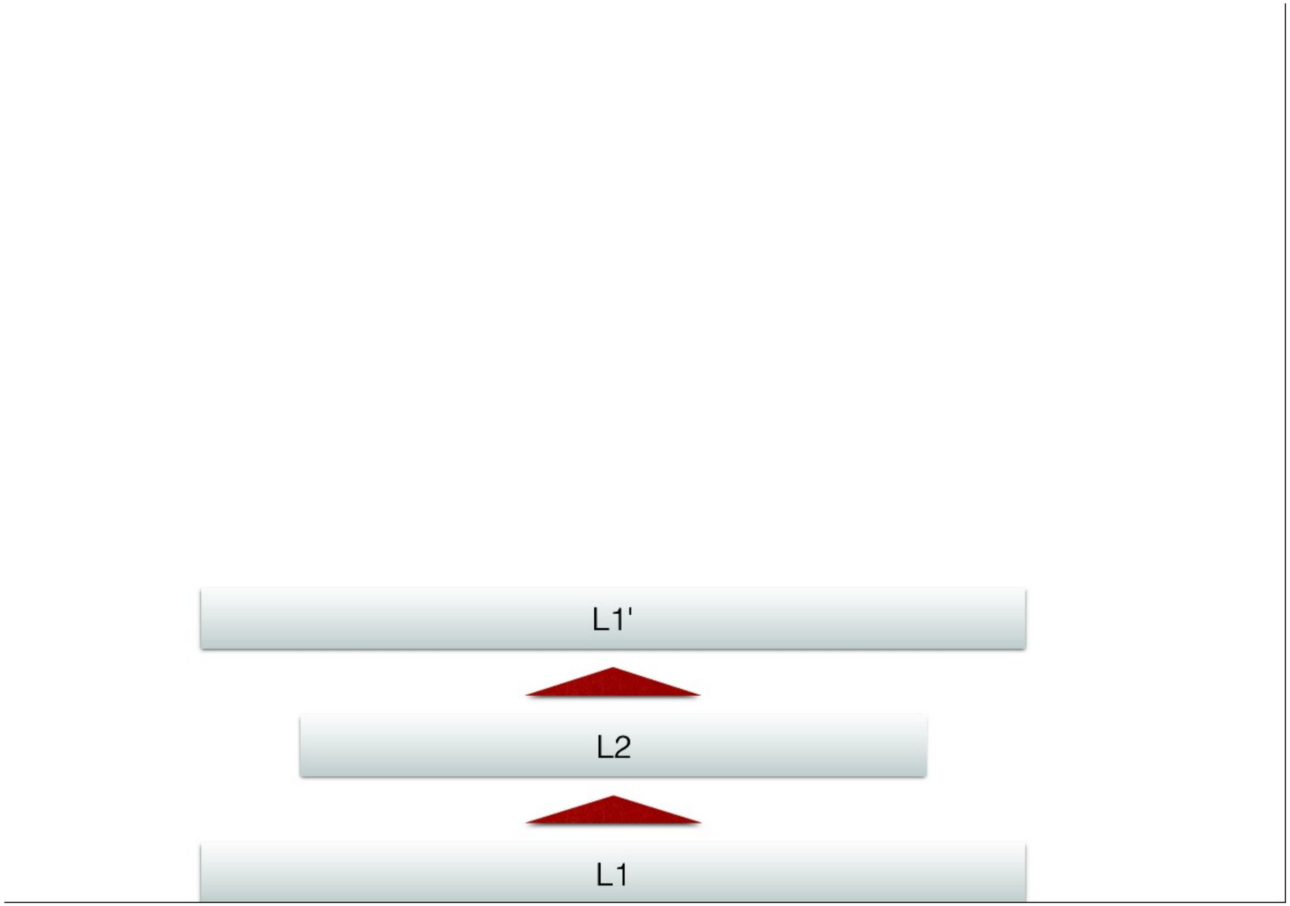
- What about non-linear units?



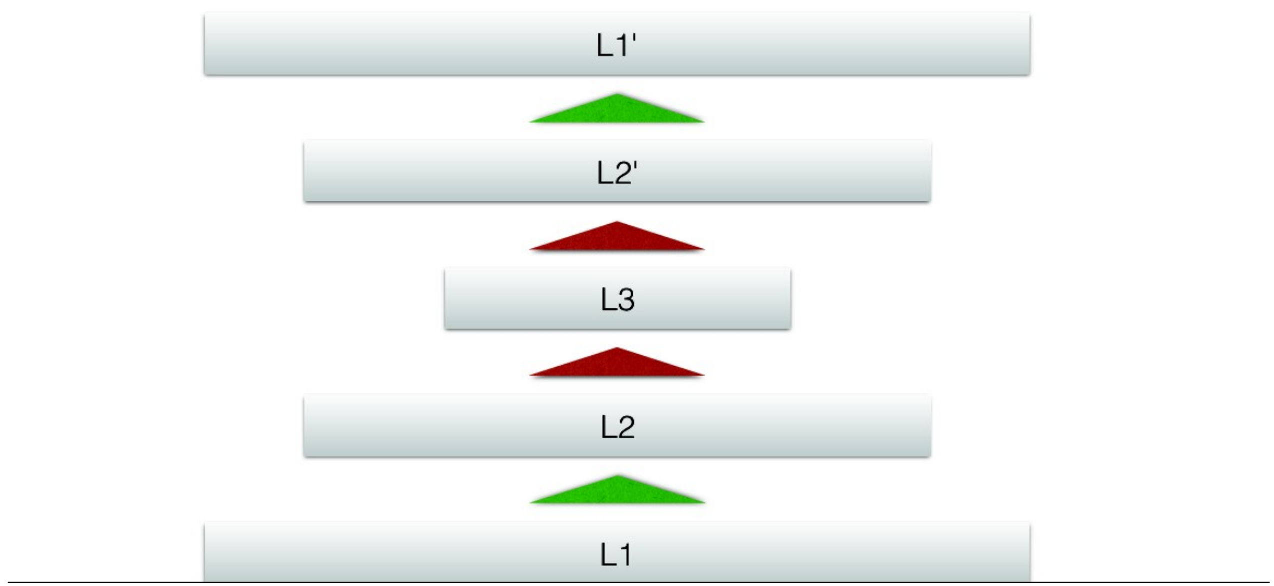
Stacked Autoencoders: Unsupervised Pretraining

- We know, it is hard to train DNNs.
- We can use the following weight initialization method:
  1. Train the first layer as a shallow autoencoder.
  2. Use the hidden units' outputs as an input to another shallow autoencoder.
  3. Repeat (2) to until desired number of layers is reached.
  4. Fine-tune using supervised learning.
- Steps 1 & 2 are unsupervised (no labels needed).

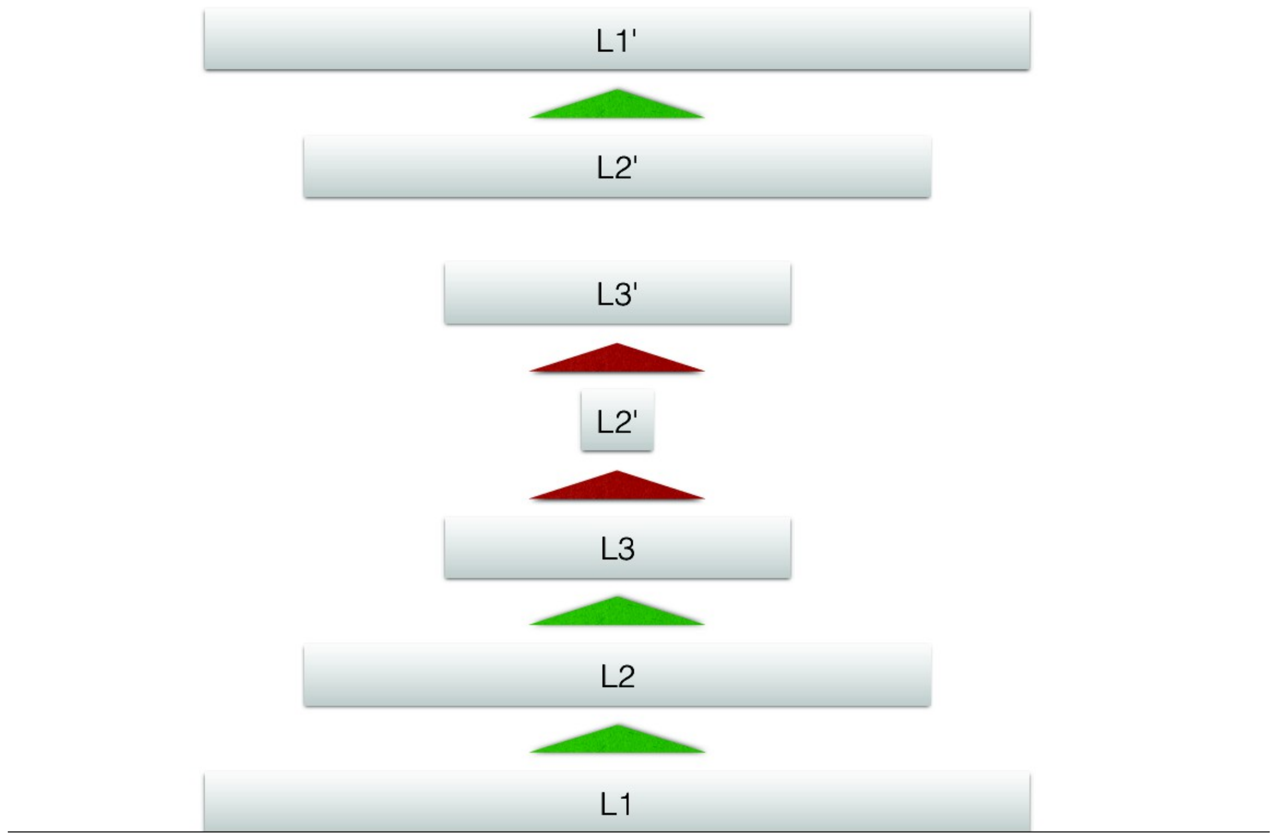
Stacked Autoencoders II.



Stacked Autoencoders III.



Stacked Autoencoders IV.



Other Deep Learning Approaches

- Deep Neural Networks are not the only implementation of Deep Learning.
- Graphical Model approaches.
- Key words:
  - Restricted Boltzmann Machine (RBM),
  - Stacked RBM,
  - Deep Belief Network.

Tools for Deep Learning

- GPU acceleration.
  - cuda-convnet2 (C++),
  - Caffe (C++, Python, Matlab, Mathematica),
  - Theano (Python),
  - DL4J (Java),
  - and many others.

Next Lecture

- Recurrent ANNs = RNNs