Introduction to Machine Learning and Deep Learning

CNRS Formation Entreprise

Stéphane Gaïffas – Karine Tribouley

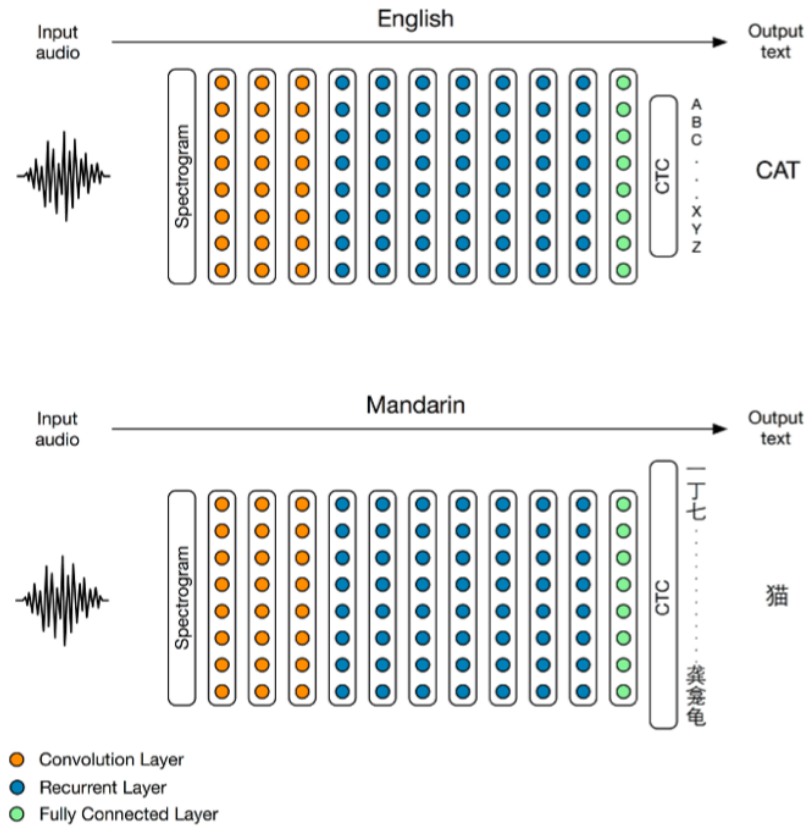
Deep learning. What it is?

First, it sounds **cool** !

What is it used for ?

Here are some examples, among many many (many) others...

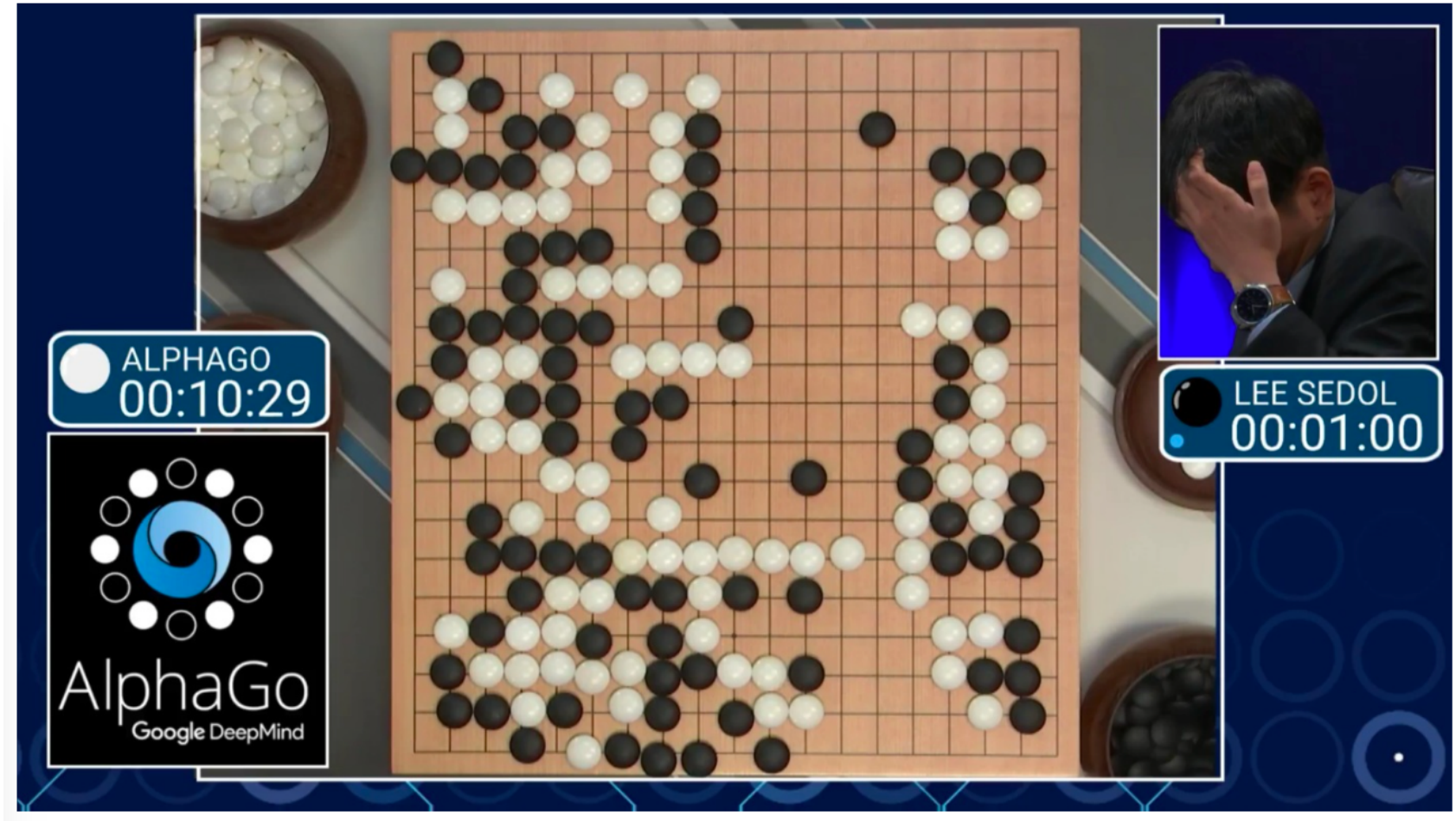
Where is deep learning today



[Baidu 2014]

Speech recognition

Where is deep learning today



Beat humans at Go (deep learning and reinforcement learning)

Where is deep learning today

Stanford | News

Search Stanford news...

Home Find Stories For Journalists Contact

JANUARY 25, 2017

Deep learning algorithm does as well as dermatologists in identifying skin cancer

In hopes of creating better access to medical care, Stanford researchers have trained an algorithm to diagnose skin cancer.

BY TAYLOR KUBOTA

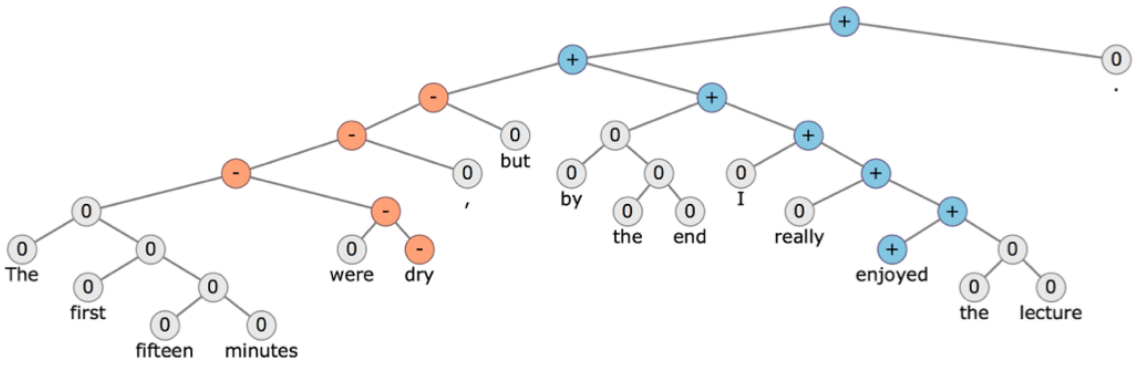
It's scary enough making a doctor's appointment to see if a strange mole could be cancerous. Imagine, then, that you were in that situation while also living far away from the nearest doctor, unable to take time off work and unsure you had the money to cover the cost of the visit. In a scenario like this, an option to receive a diagnosis through your smartphone could be lifesaving.

Universal access to health care was on the minds of computer scientists at Stanford when they set out to create an artificially intelligent diagnosis algorithm for skin cancer. They made a database of nearly 130,000 skin disease images and trained their algorithm to visually diagnose potential cancer. From the very first test, it performed with inspiring accuracy.

Analysis of medical pictures

Where is deep learning today

[Google Translate System - 2016]



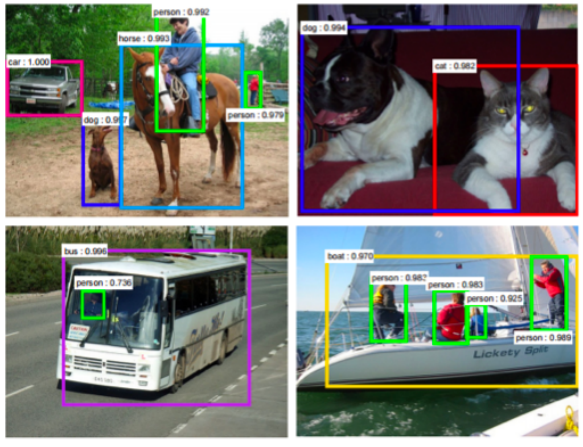
[Socher 2015]

Automatic translation

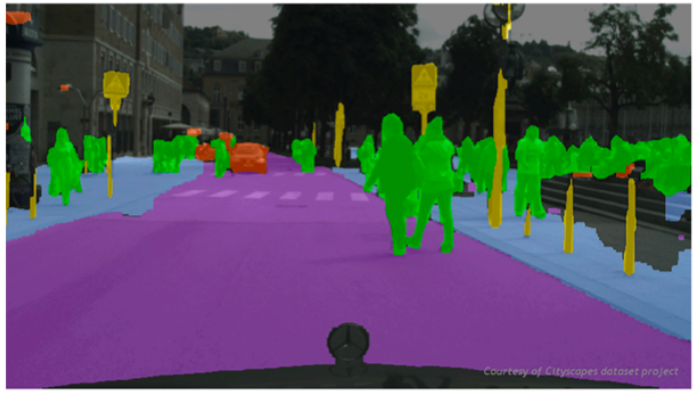
Where is deep learning today

[Krizhevsky 2012]

[Ciresan et al. 2013]



[Faster R-CNN - Ren 2015]



[NVIDIA dev blog]

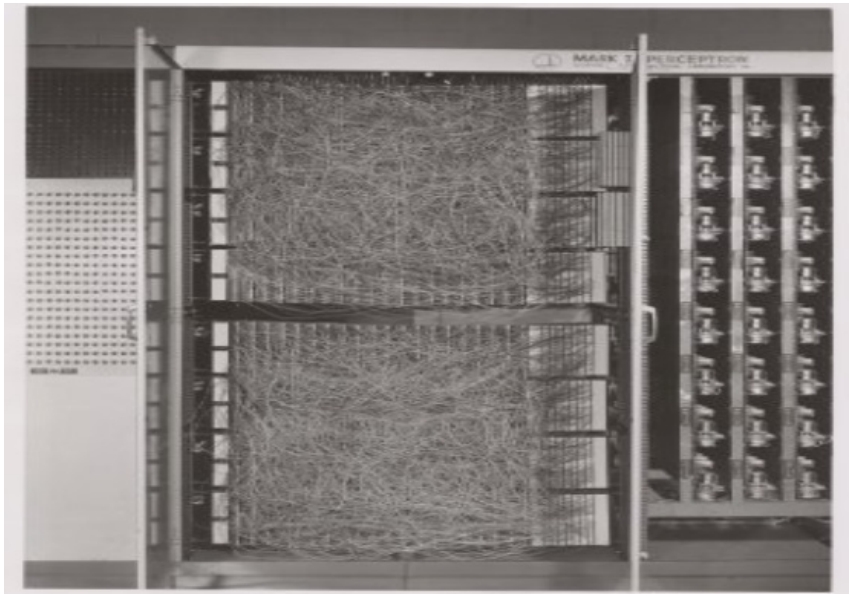
Image classification, entity naming, automatic cars

Where is deep learning today

Even scary pictures!

Deep learning

- Actually based on old idea from the 80s (and even earlier...)
- Started with the perceptron (50s, Frank Rosenblatt)



An old perceptron

Look at the youtube video (**BUT NOT NOW** !!!)

`https://youtu.be/aygSMgK3BEM`

Deep learning

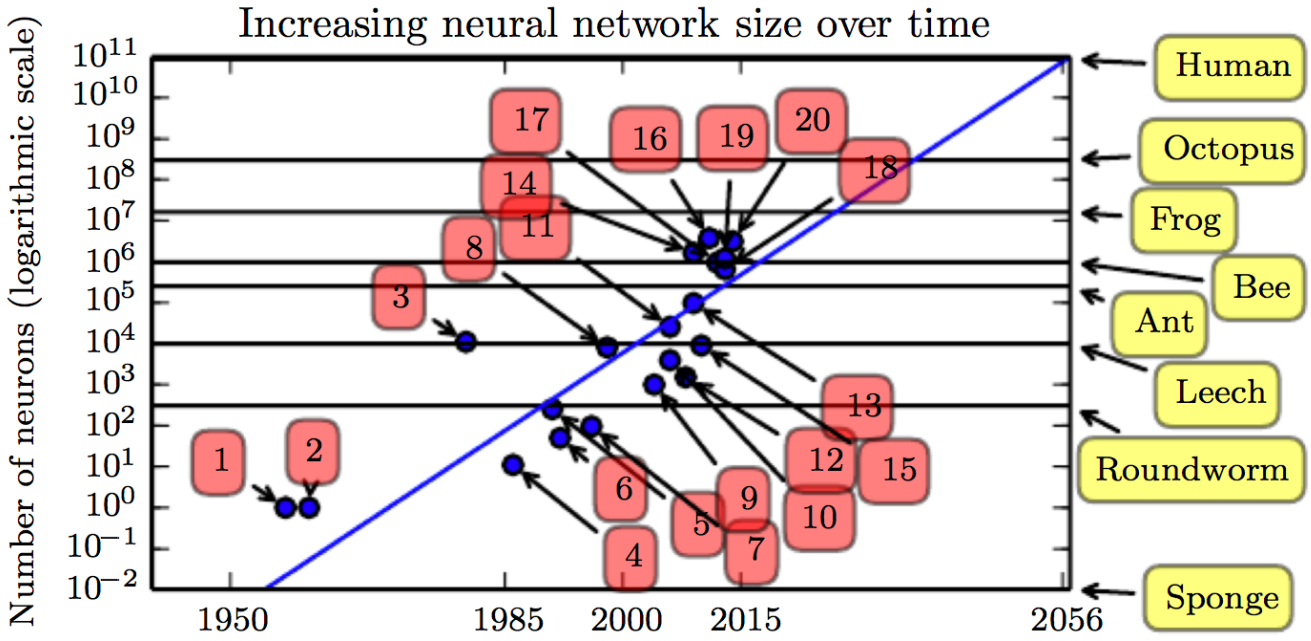
- Deep learning is good old neural networks...

But with:

- more data
- more computational powers (GPU clusters)
- more layers
- more everything...

Their **length** (number of layers) and **width** (dimension of the hidden units) has increased over the years

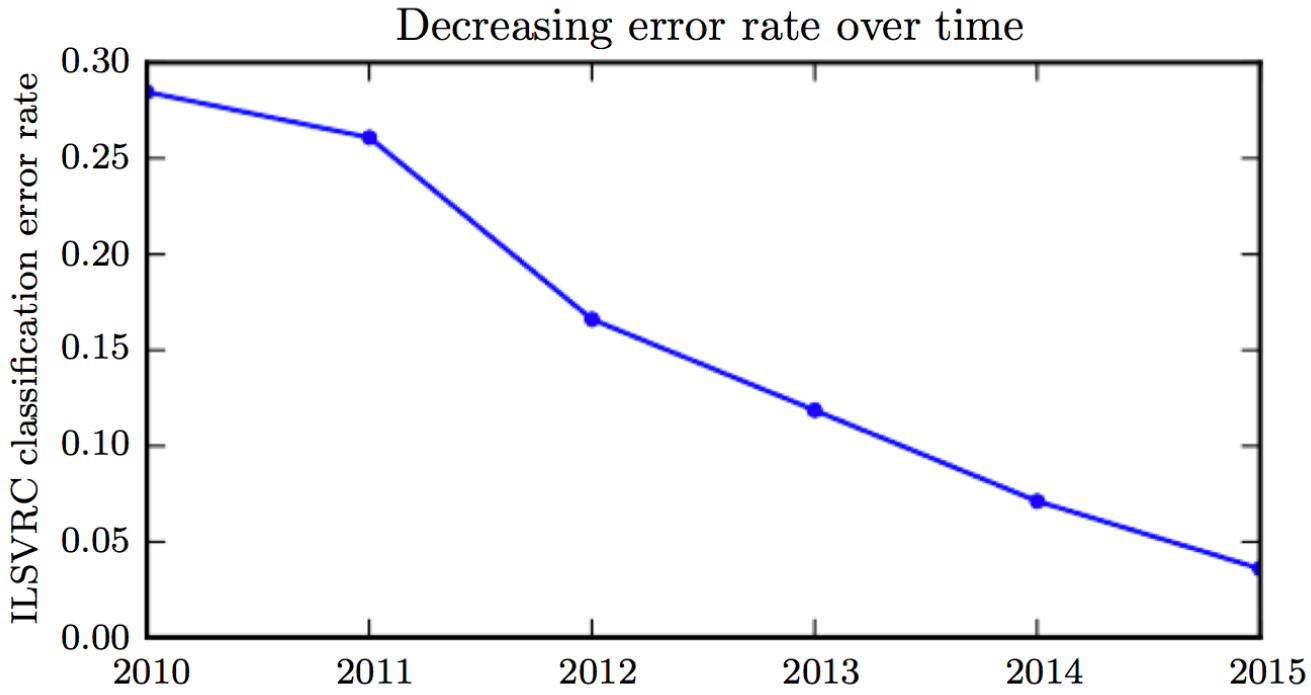
Increasing size of neural networks



[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Neural networks have doubled in size roughly every 2.4 years (in research paper, red)
- Comparison with connections in animal brains (yellow)
- Warning: analogy between computational and biological neurons is a **very rough** analogy

Strong improvement of the error rates



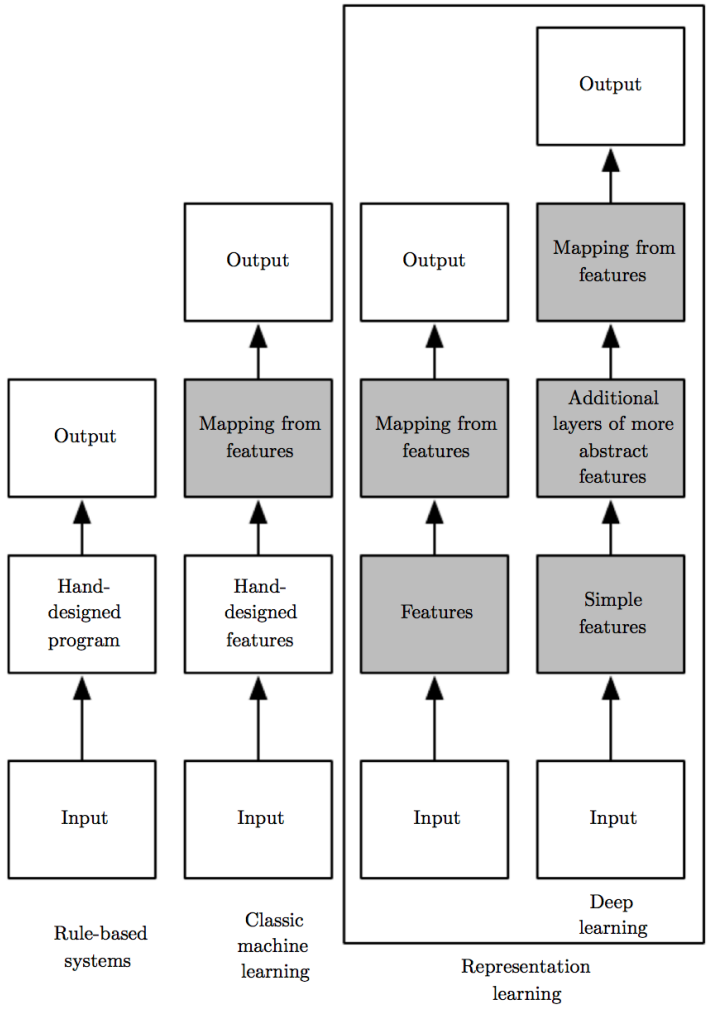
[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Deep networks reached the goal of ImageNet Large Scale Visual Recognition Challenge
- Consistently won the competition and lower the error rate each year

Representation learning

- Deep learning is representation learning
- Computes hierarchical, abstract representations of the data
- **Learn features**
- Actually learns **several levels of abstractions** of the features

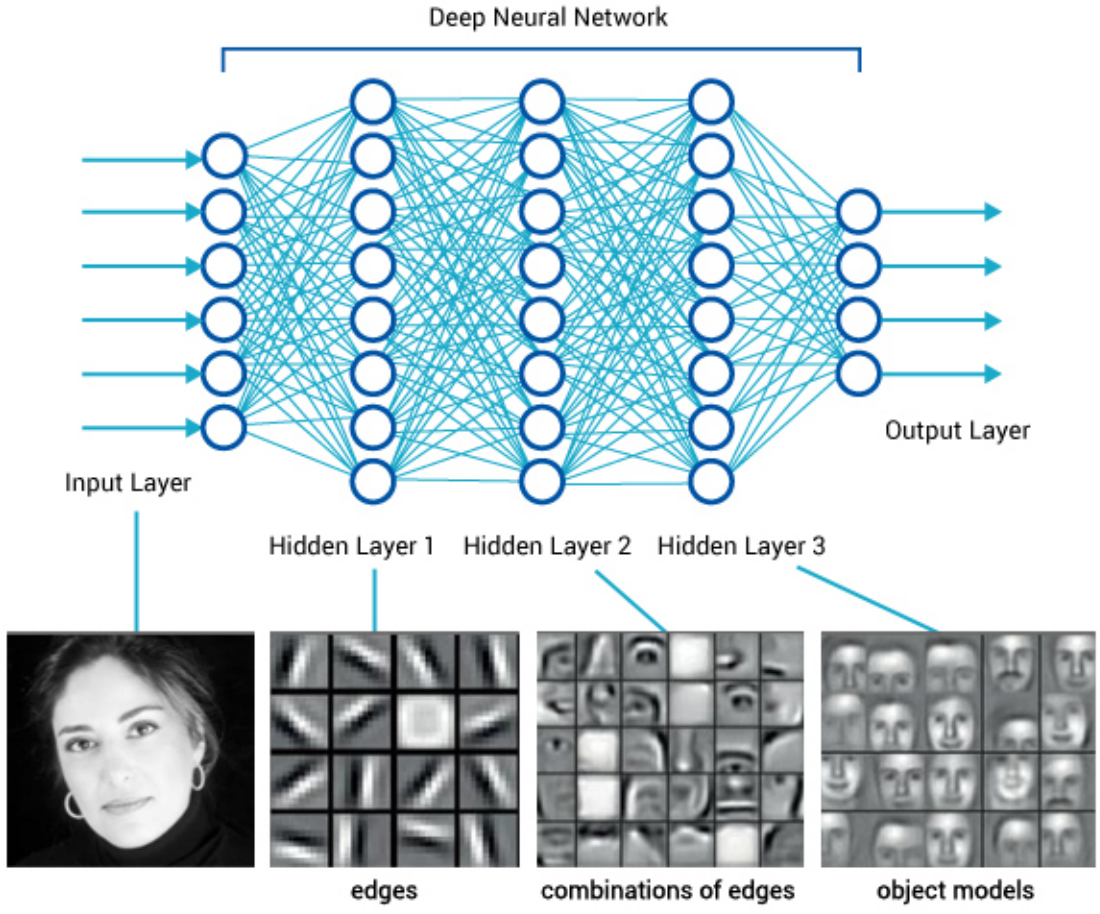
Representation learning



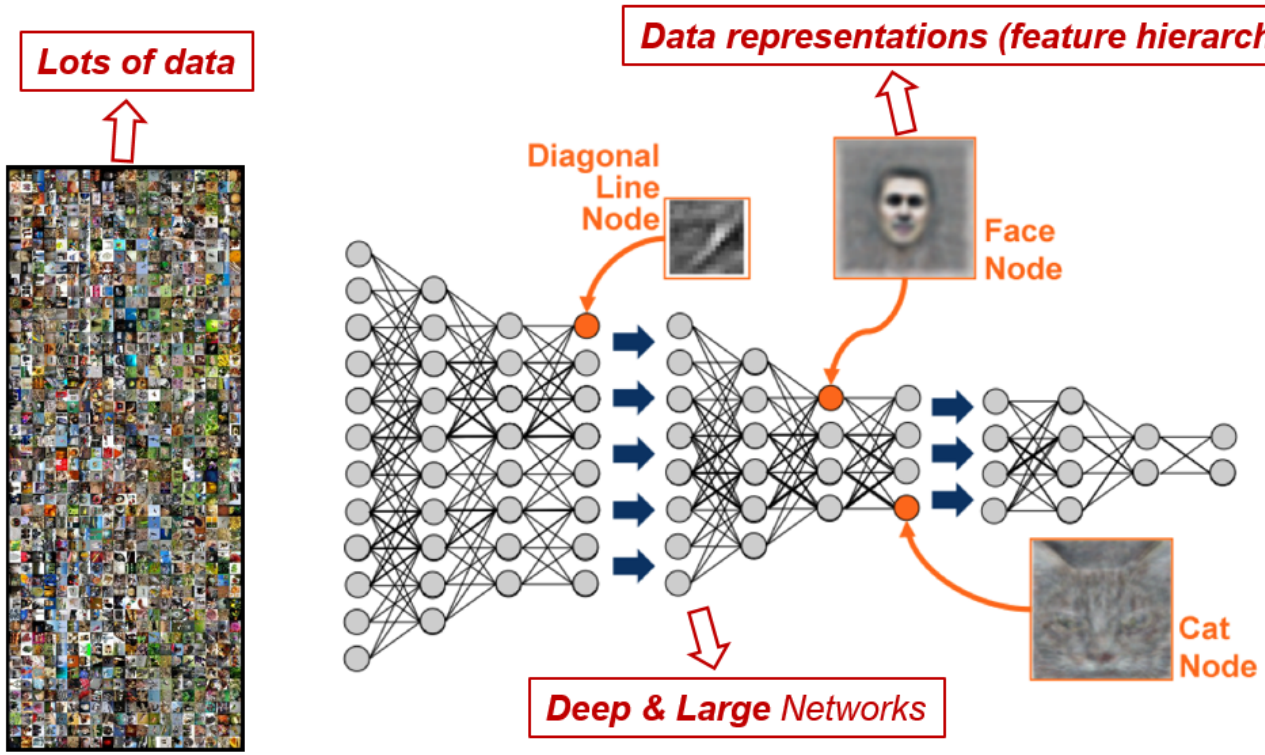
Grey rectangles are machine-learned

[from *Deep Learning*, Goodfellow, Bengio and Courville]

Representation learning



Representation learning



Main examples

- Feed-forward neural networks (FFNN)
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)

Let's learn about the main tools

- Forward-propagation and back-propagation for gradient computation
- Stochastic gradient, adaptive learning rates
- Regularization techniques / dropout
- Weights sharing: CNN

Let's start with...

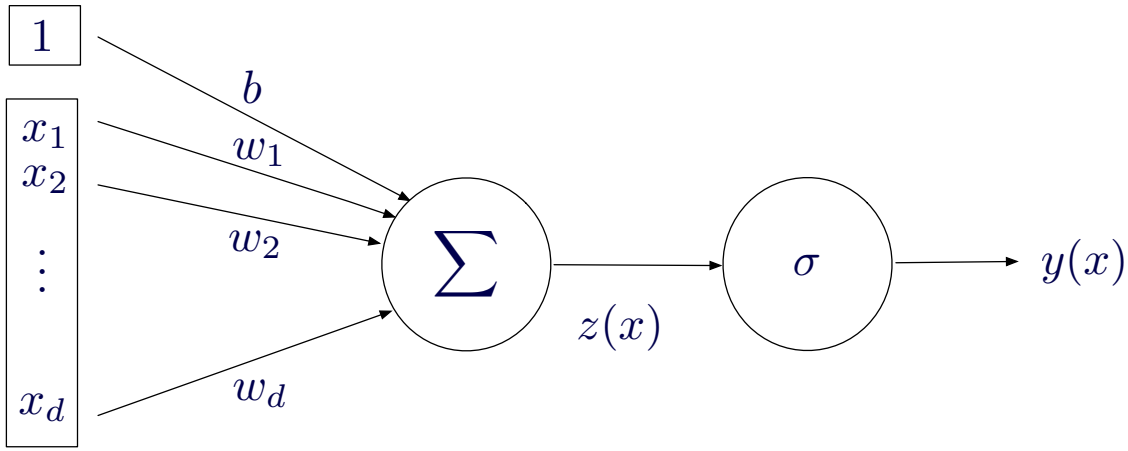
Logistic regression

- Dataset $(x_1, y_1), \ldots, (x_n, y_n)$ with $y_i \in \{0, 1\}$
- Given an input features vector $x \in \mathbb{R}^d$, consider

$$\mathbb{P}(Y = 1 | X = x) = \sigma(\langle w, x \rangle + b) = \frac{1}{1 + e^{-\langle w, x \rangle - b}}$$

- Model **weights** $w \in \mathbb{R}^d$ and intercept (or **bias**) $b \in \mathbb{R}$

Graphically, a logistic regression is as follows



In deep learning vocabulary:

- x is the input (a single feature vector)
- $z(x) = \langle w, x \rangle + b$ is called **pre-activation**
- $y(x) = \sigma(z(x))$ is the output (in $[0, 1]$ in this case)
- w is **weights** and b is **bias**
- σ is an **activation** function

It is called a **unit** or an **artificial neuron**

Softmax regression or **Multinomial logistic regression**

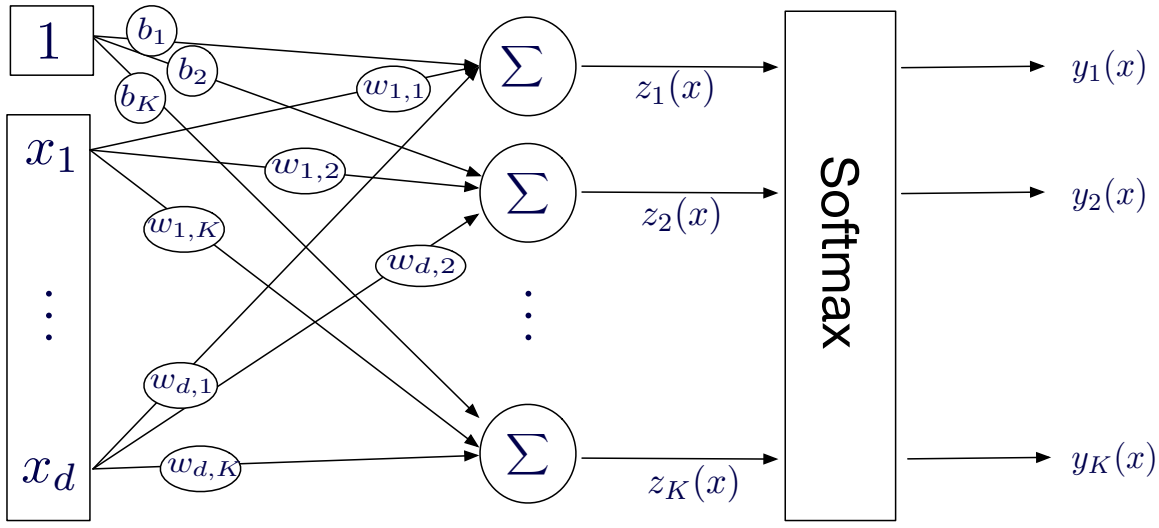
- Dataset $(x_1, y_1), \ldots, (x_n, y_n)$ with $y_i \in \{1, \ldots, K\}$
- Softmax regression is a generalization of logistic regression for more than two classes
- Given an input features vector $x \in \mathbb{R}^d$, consider

$$\mathbb{P}(Y = k | X = x) = \frac{e^{\langle w_k, x \rangle + b_k}}{\sum_{k'=1}^{K} e^{\langle w_{k'}, x \rangle + b_{k'}}}$$

  for $k \in \{1, \ldots, K\}$
- One vector of weights $w_k \in \mathbb{R}^d$ for each class k. Stack the model weights in a $d \times K$ matrix $\boldsymbol{W}$ with $\boldsymbol{W}_{\bullet,k} = w_k$
- Negative log-likelihood given by

$$-\ell(\boldsymbol{W}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} \mathbf{1}_{y_i = k} \log \left(\frac{e^{\langle w_k, x \rangle + b_k}}{\sum_{k'=1}^{K} e^{\langle w_{k'}, x \rangle + b_{k'}}} \right)$$

A softmax regression is as follows



- x input and $y_1(x), \ldots, y_K(x)$ output
- $z_k(x) = \langle \boldsymbol{W}_{\bullet,k}, x \rangle + b_k$ pre-activations or "logits"
- $y_k(x) = \frac{e^{z_k(x)}}{\sum_{k'=1}^{K} e^{z_{k'}(x)}}$ coming out of the softmax activation

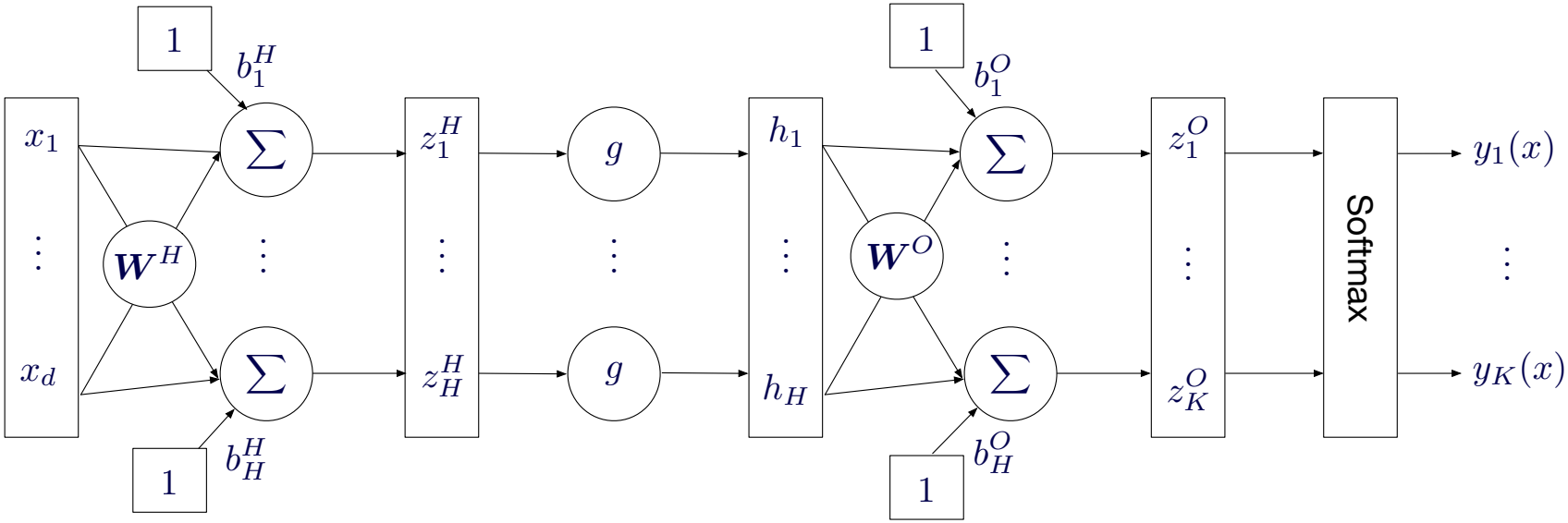
Written matricially as

$$y(x) = \text{softmax}(z(x)) = \text{softmax}(\boldsymbol{W}^\top x + b)$$

Called a **layer** (softmax is often the **output** layer)

A one-hidden layer neural network is as follows

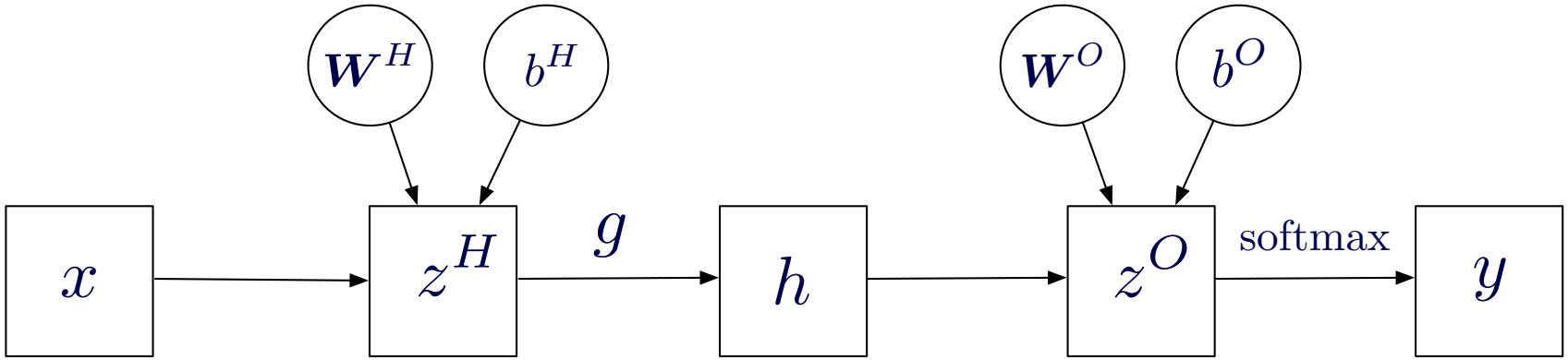
- Also called 1 hidden layer feed-forward neural network



$$y(x) = \text{softmax}(z^{(O)}) = \text{softmax}(\boldsymbol{W}^{(O)\top} h + b^{(O)})$$
$$= \text{softmax}(\boldsymbol{W}^{(O)\top} g(z^{(H)}) + b^{(O)})$$
$$= \text{softmax}(\boldsymbol{W}^{(O)\top} g(\boldsymbol{W}^{(H)\top} x + b^{(H)}) + b^{(O)})$$

- g is an **activation function** applied entrywise on $z_1^H, \ldots, z_H^H$
- This neural network has a width-H hidden layer

Another way (among others) to represent the same network is as follows



$$
\begin{aligned}
y(x) &= \text{softmax}(z^{(O)}) = \text{softmax}(\boldsymbol{W}^{(O)\top} h + b^{(O)}) \\
&= \text{softmax}(\boldsymbol{W}^{(O)\top} g(z^{(H)}) + b^{(O)}) \\
&= \text{softmax}(\boldsymbol{W}^{(O)\top} g(\boldsymbol{W}^{(H)\top} x + b^{(H)}) + b^{(O)})
\end{aligned}
$$

- g is an **activation function** applied entrywise on $z_1^{(H)}, \ldots, z_H^{(H)}$
- This neural network has a width-H hidden layer

Activation functions

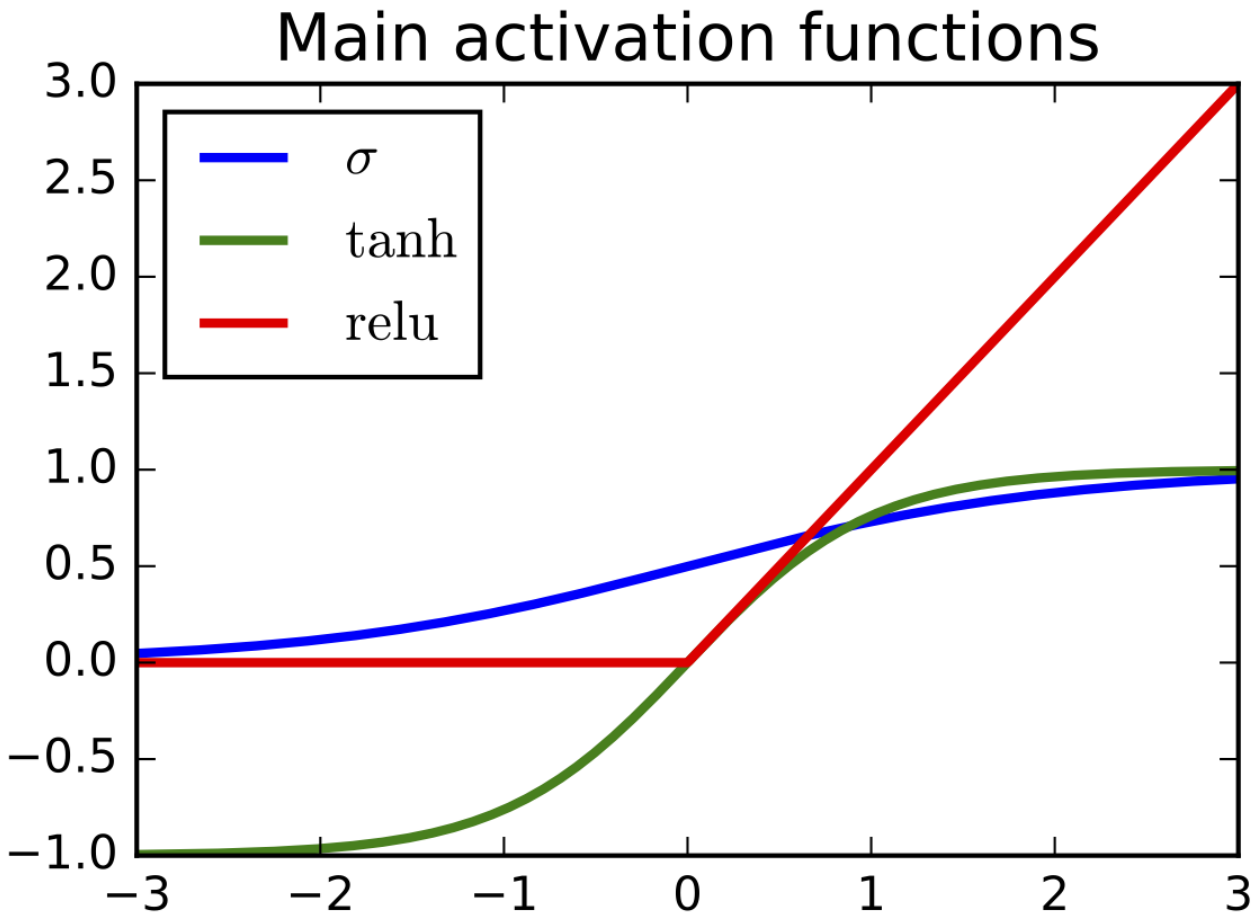
- Applied entrywise on the inputs
- Main examples are the **sigmoid**, **tanh** and **relu** (rectified linear unit)

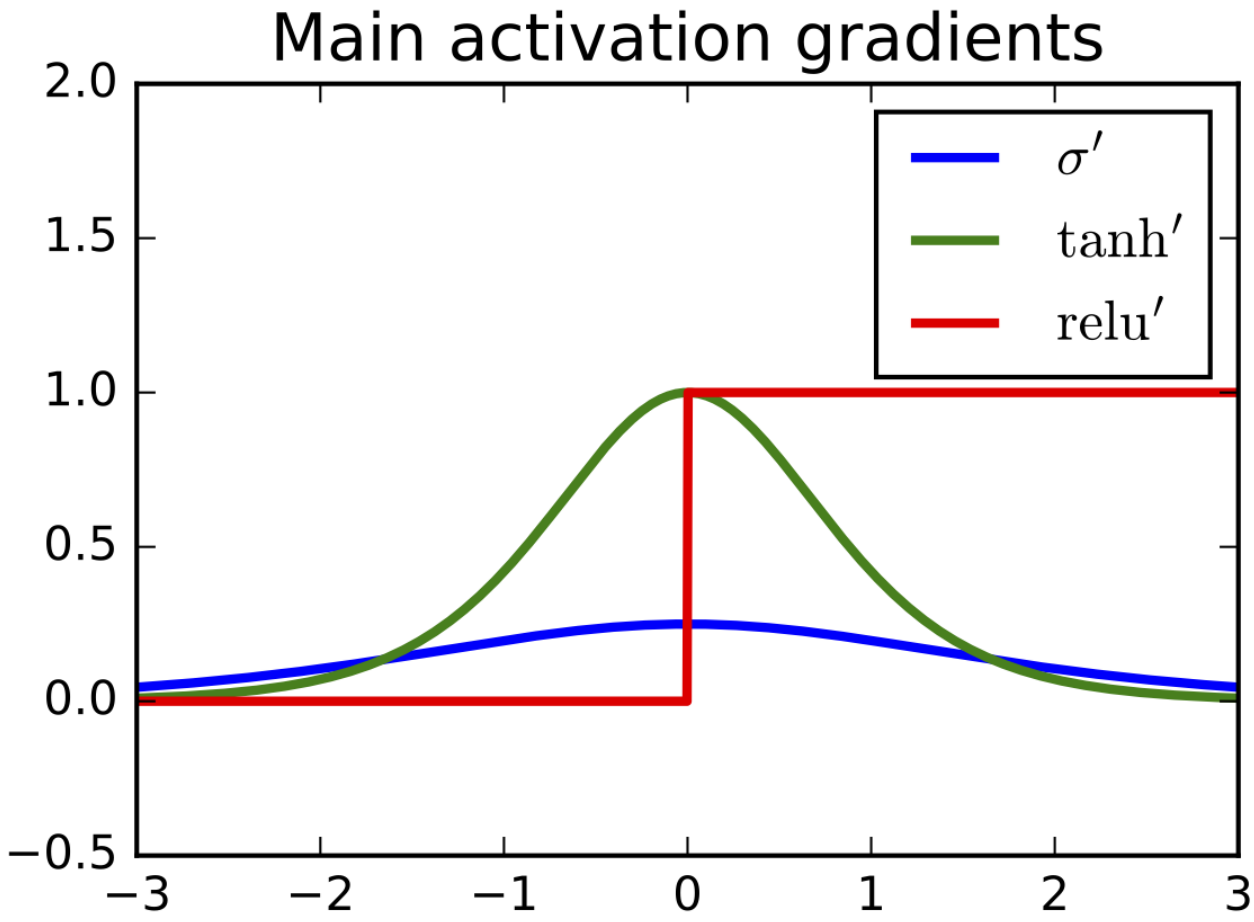
$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}, \quad \mathrm{relu}(z) = \max(0, z)$$

- Their derivatives are given by

$$\begin{aligned} \sigma'(z) &= \sigma(z)(1-\sigma(z)) \\ \tanh(z) &= 1 - \tanh(z)^2, \\ \mathrm{relu}'(z) &= \mathbf{1}_{z>0} \end{aligned}$$

- Note that $\tanh(z) = 2\sigma(2z) - 1$


Main activation functions
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
σ
tanh
relu


Main activation gradients
2.0
1.5
1.0
0.5
0.0
−0.5
−3
−2
−1
0
1
2
3
σ'
$\tanh'$
relu'

About the **relu** activation

- Recent research shows that it works better in practice for deep neural networks
- Easier to optimize, since their behaviour is closer to linear
- "Sigmoid" activations σ and tanh saturate for large positive or large negative values, while relu don't. This is why sigmoid activations in hidden units are now not recommended
- Its gradient is not defined at $z = 0$: not a problem since during training, it's unlikely that many inputs equal 0

Feed-forward neural network (FFNN)

- Aims at building an approximation f of some function f^* that maps an input x to an output y: $y = f^*(x)$
- The name comes from the fact that information flows through the layers that define f
- Layers correspond to functions that are composed in some order: $f(x) = f_3(f_2(f_1(x)))$ has 3 layers
- The network describes how the functions are composed together
- f_1 is the first layer, while f_3 is the last one

Feed-forward neural networks (FFNN)

$$
\begin{aligned}
h^{(1)} &= g^{(1)}(\boldsymbol{W}^{(1)\top} x + b^{(1)}) \\
h^{(2)} &= g^{(2)}(\boldsymbol{W}^{(2)\top} h^{(1)} + b^{(2)}) \\
&\vdots \\
h^{(L)} &= g^{(L)}(\boldsymbol{W}^{(L)\top} h^{(L-1)} + b^{(L)}) \\
y &= \text{softmax}(\boldsymbol{W}^{(O)\top} h^{(L)} + b^{(O)})
\end{aligned}
$$

First layer, second layer, L-th layer and so on

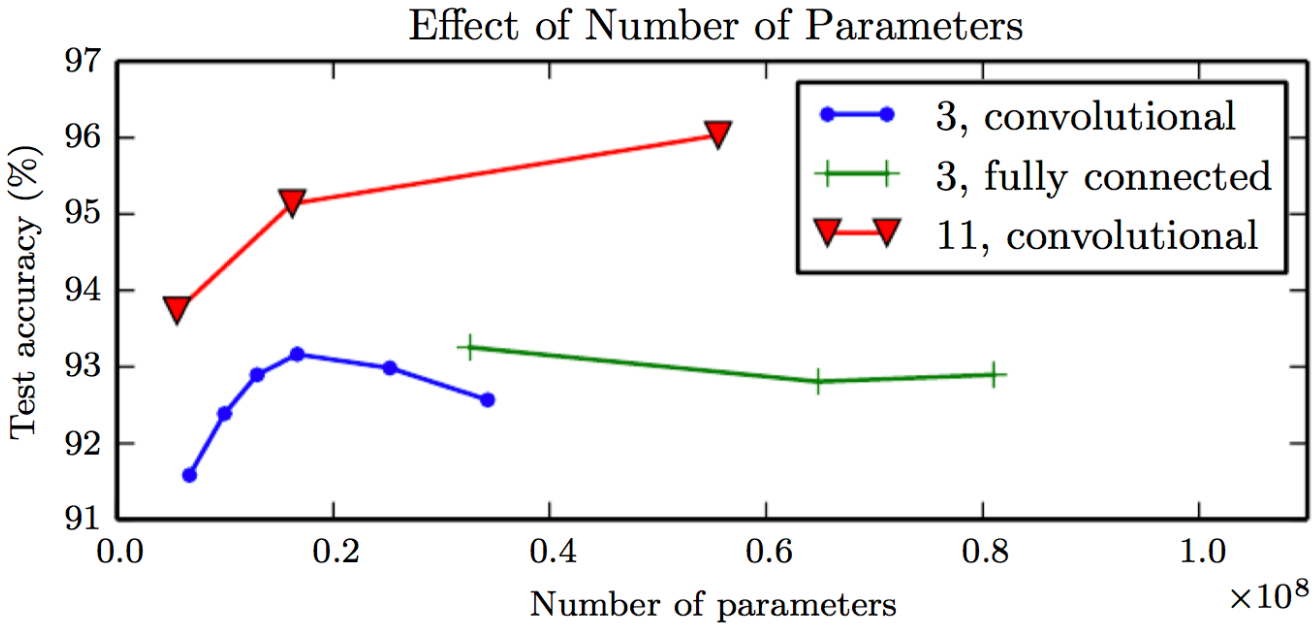
- A chain based structure
- Need to choose the width of each layer
- And the depth of the network
- They have some universal approximation properties

A question

- One hidden layer with large width?
- ... or several layers with smaller width ?

A recipe

- It is believed that several layers with a smaller width lead to better generalization (more abstract layers)



[from *Deep Learning*, Goodfellow, Bengio and Courville]

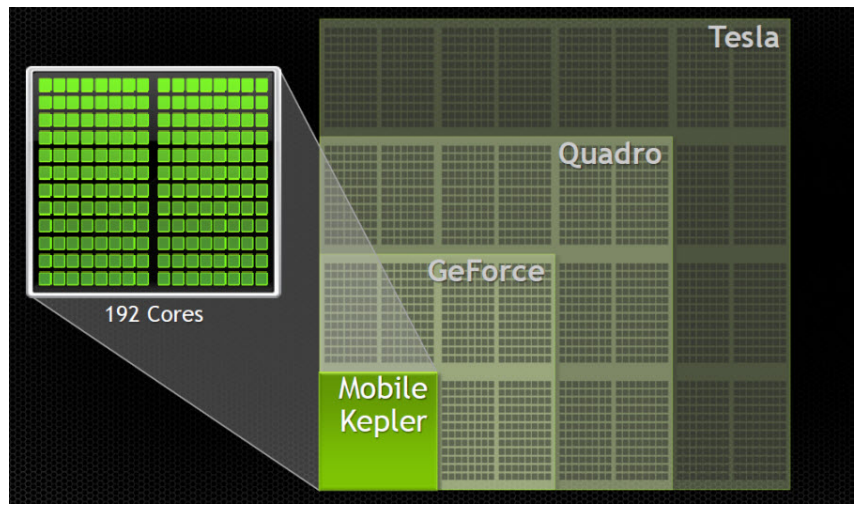
- Deeper models seem to perform better

Main tools to train and use a FFNN (and other DNN)

- To compute the loss, and for prediction: **forward-propagation**
- To compute gradients: **back-propagation** (uses forward-propagation at the beginning)
- A **stochastic optimization algorithm**

And mostly

- A nice open source library (`tensorflow`, `caffe`, `pytorch`, `keras`)
- A GPU... since training a DNN benefits from massive parallel computations



Training of the network. Recall the one-hidden layer FFNN from before

$$y = \text{softmax}(\boldsymbol{W}^{(O)\top} g(\boldsymbol{W}^{(H)\top} x + b^{(H)}) + b^{(O)})$$

Goodness-of-fit of parameters $\boldsymbol{\theta} = (\boldsymbol{W}^H, b^H, \boldsymbol{W}^O, b^O)$ is given by negative log-likelihood (also called cross-entropy)

$$-\text{LogLik}(\boldsymbol{\theta}) =$$

$$-\sum_{i=1}^{n}\sum_{k=1}^{K} \mathbf{1}_{y_i=k} \log\left(\frac{\exp\left(\left\langle \boldsymbol{W}^{(O)}_{\bullet,k}, g(\boldsymbol{W}^{(H)\top} x_i + b^{(H)})\right\rangle + b^{(O)}_k\right)}{\sum_{k'=1}^{K}\exp\left(\left\langle \boldsymbol{W}^{(O)}_{\bullet,k'}, g(\boldsymbol{W}^{(H)\top} x_i + b^{(H)})\right\rangle + b^{(O)}_{k'}\right)}\right)$$

Add some regularization to avoid overfitting, such as ridge

$$\text{pen}(\boldsymbol{\theta}) = \text{pen}(\boldsymbol{W}^{(O)}, \boldsymbol{W}^{(H)}) = \lambda(\|\boldsymbol{W}^{(O)}\|_F^2 + \|\boldsymbol{W}^H\|_F^2)$$

Remark. biases are not penalized

Training of the network

We need to minimize

$$F(\boldsymbol{\theta}) = -\text{LogLik}(\boldsymbol{\theta}) + \text{pen}(\boldsymbol{\theta})$$

In general, note that

$$-\text{LogLik}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(x_i; \boldsymbol{\theta}))$$

where

- ℓ is a loss function (often softmax)
- $f(\cdot; \boldsymbol{\theta})$ is the output of the network obtained by **forward-propagation** for parameters $\boldsymbol{\theta}$
- Gradients $\nabla F(\boldsymbol{\theta})$ (on mini-batches) are obtained using **back-propagation**

Forward-propagation

- Example on the 1-hidden layer FFNN
- Given a mini-batch (x_i, y_i) for $i \in B$ (or the full dataset)
- Given model weights $\boldsymbol{\theta} = (\boldsymbol{W}^H, b^H, \boldsymbol{W}^O, b^O)$
- Let $\boldsymbol{X}$ be the $|B| \times d$ matrix with rows x_i Let $\boldsymbol{Y}$ be the $B \times K$ matrix with rows y_i

Do the following

- $z^H \leftarrow \boldsymbol{W}^{(H)\top} \boldsymbol{X} + b^{(H)}$
- $h \leftarrow g(z^H)$
- $z^O \leftarrow \boldsymbol{W}^{(O)\top} h + b^{(O)}$
- $\hat{\boldsymbol{Y}} \leftarrow \text{softmax}(z^O)$
- Compute $-\ell(\theta)$ as cross_entropy$(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$
- Compute $-\ell(\theta) + \text{pen}(\boldsymbol{\theta})$

Back-propagation

- An algorithm to compute gradients
- Not specific not neural networks, can be used to compute the gradient of an arbitrary function $\nabla_x f(x, y)$, where x is a set of variables whose derivatives are required, while we don't require those w.r.t. y
- For NN, we require gradients of the NN objective with respect to the model weights $\boldsymbol{\theta}$
- Uses the computational graph associated to the NN
- Uses the chain rule to compute the derivative of composed functions

Back-propagation

- Very simple idea: $f(g(x))' = f'(g(x)) \times g'(x)$
- Forward-propagation: compute and save $y = g(x)$ (value of intermediate layers) at the **current parameters**
- Back-propagation: compute $f'(y) \times g'(x)$, namely

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

- Beyond the scalar case $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$ and $g : \mathbb{R}^m \to \mathbb{R}^n$ then if $y = g(x)$ and $z = f(y)$ we have

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Back-propagation

- in vector notations

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z,$$

  where $\frac{\partial y}{\partial x}$ is the $n \times m$ Jacobian matrix of g
- Gradient of a variable z with respect to another variable x can be obtained my multiplying $\frac{\partial y}{\partial x}$ by $\nabla_y z$
- Back-prop performs such Jacobian-gradient products for each operation in the computational graph, respecting the order of the operations
- Not only for vectors x, but also tensors $\boldsymbol{X}$ of any shape (2D=matrix, but 3D, 4D is also very useful)
- Exact same rules of calculus apply

- Once again, $x = f(w)$, $y = f(x)$, $z = f(y)$ then the chain rule gives

$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\ &= f'(y)f'(x)f'(w) \quad (1) \\ &= f'(f(f(w)))f'(f(w))f'(w) \quad (2)\end{aligned}$$

- Back-prop computes and saves $x \leftarrow f(w)$ and $y \leftarrow f(x)$ (forward propagation) and then uses Eq. (1) to compute $\frac{\partial z}{\partial w}$
- Eq. (2) can be used as well: uses less memory, but does more computations (actually a lot more for deep compositions)

Nearly all of deep learning is powered by a very important algorithm: **stochastic gradient descent** (SGD)

- The same algorithm as the one we saw last time
- With stochastic gradients based on mini-batches

In DL **mini-batch** gradients are used. Mini-batches of size m.

- Draw uniformly at random from the training dataset $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ a subset $\{(x_i, y_i) : i \in B\}$ with $|B| = m$
- m is between 1 and a few hundreds. Often chosen as a power of two, typically $m \in \{32, 64, \ldots, 256\}$
- The mini-batch gradient is given by

$$\frac{1}{m} \sum_{i \in B} \nabla_{\boldsymbol{\theta}} \ell(y_i, f_{\boldsymbol{\theta}}(x_i)) + \nabla_{\boldsymbol{\theta}} \operatorname{pen}(\boldsymbol{\theta})$$

Recipe

- If the network is deep, use m small (≈ 16) and waaaaaiiit !

Stochastic gradient descent (SGD)

Input: learning rates η_t, initial parameter $\boldsymbol{\theta}$

While *stopping criterion not met* **do**

- Sample a mini-batch of m examples for the training dataset $\{(x_i, y_i) : i \in B\}$ with $|B| = m$
- Compute the gradient estimate

$$\boldsymbol{G} \leftarrow \frac{1}{m} \sum_{i \in B} \nabla_{\boldsymbol{\theta}} \ell(y_i, f(x_i; \boldsymbol{\theta})) + \nabla_{\boldsymbol{\theta}} \operatorname{pen}(\boldsymbol{\theta})$$

- Apply the update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_t \boldsymbol{G}$$

About the **learning rate** η_t

- More an art than a science...
- Must be decreasing along iterations
- Common practice is to decay it until iteration τ:

$$\eta_t = (1 - \alpha)\eta_0 + \alpha\eta_\tau$$

  with $\alpha = t/\tau$ and η_t kept constant after τ iterations
- Common practice is $\tau \approx$ #iterations for a few 100 passes on the data and α such that $\eta_\tau = 10^{-2}\eta_0$
- Choice of η_0: not too small, not too large. Try out several of them, take the one that gives best result in early iterations
- While training a neural network, training and validation errors should be monitored
- Stop iterations when validation increases: **early stopping** (more on that later)

Several algorithms are commonly used to train DNNs. Mostly algorithms with **adaptive learning rates** such as

- AdaGrad: scales training rates using a cumulative sums of squared past values of the gradients
- RMSProp: modifies AdaGrad by replacing cumulative sums by exponentially weighted averages
- Adam, Adadelta: some extra tricks...

Remarks

- Initialization, regularization (more later) and design of the network have a strong interplay, everything is important!
- If you can't train successfuly a DNN, the problem is probably related to the data or the architecture of the DNN itself, or you need to spend more time tuning it!

Parameters initialization

- Solution might differ depending on initialization

Biases.

- Initialize at 0 or some small positive constant $\approx 1e-1$

Weights.

- Can't initialize at 0: we'd back-propagate zeros...
- Can't initialize all the weights to the same value: all neurons in a layer would behave the same
- Need to break "symmetry"

Recipe.

- Sample around 0 but break symmetry

A typical initialization is

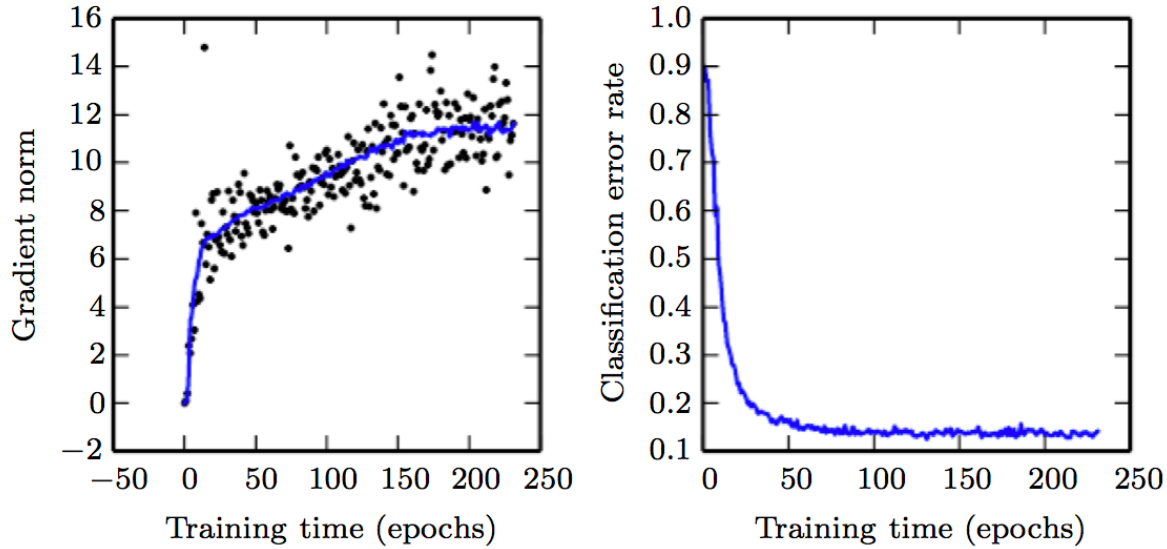
$$W_{i,j}^k \sim \text{Uniform}([-a, a]) \quad \text{with} \quad a = \sqrt{\frac{6}{H_k + H_{k-1}}}$$

with H_k = width of the output and H_{k-1} = width of the input of layer k

- Other values of a can work as well
- Once again: a lot of recipes, not an exact science !

Non-convexity

- **Number of local minima is extremely large** in a neural network objective function



- SGD often **does not converge at a critical point** (neither local minimum or saddle point)
- Gradient norm **increases (!!)** along iterations while training seems successful
- A strong departure from convex optimization

Open questions

- Are there **local minima of high cost** for neural networks of practical interest?
- Do optimization algorithms are **likely to meet them**?

For many years, we were **frightened by local minima**

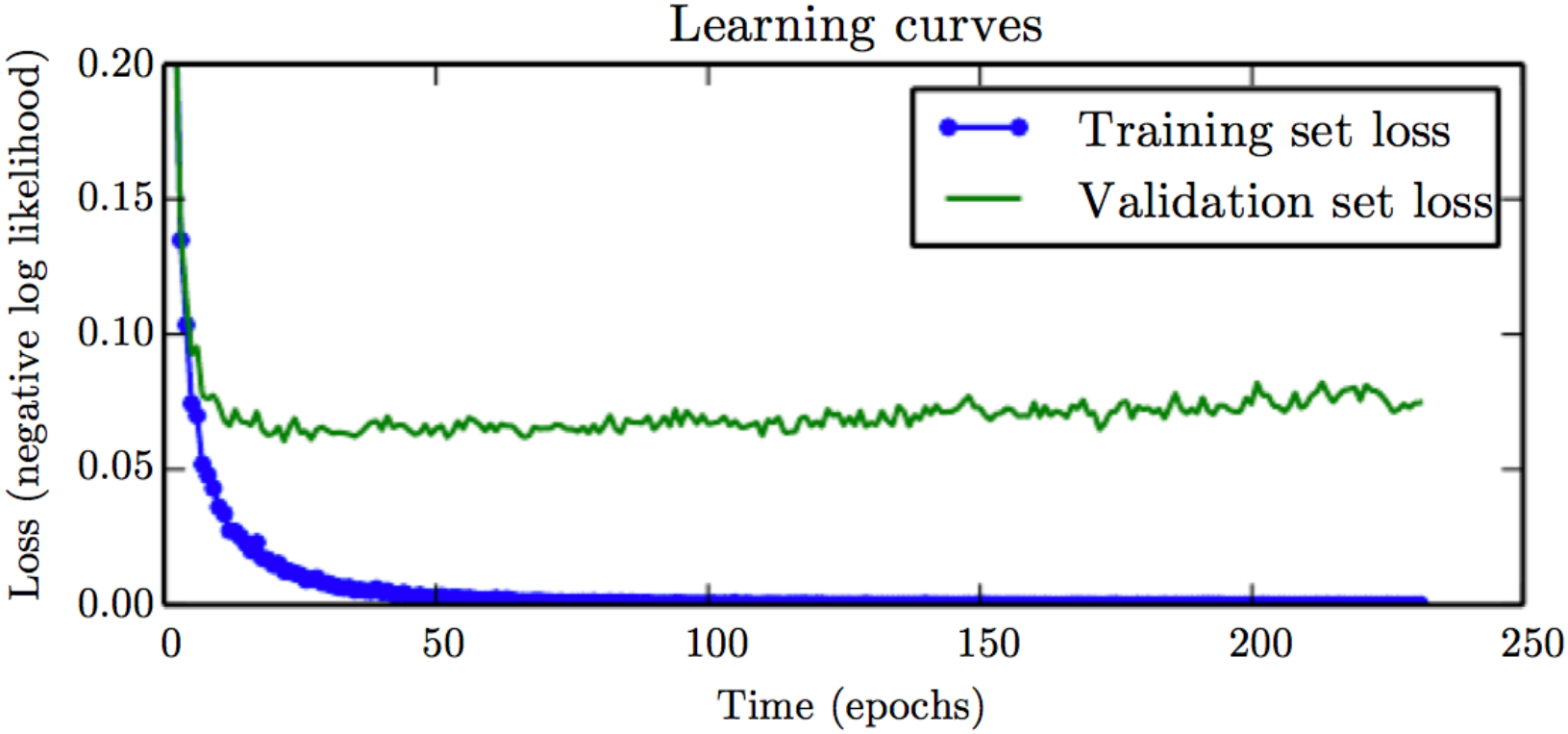
We thought that local minima were plaguing neural networks.

It is **not the case anymore**

- Common belief now is that **for sufficiently large networks, local minima have a lost cost value**
- It is **not important to find a global minimum**, but a **good local minimum**

Regularization for deep learning: early stopping

- Training large models with powerful representational power (called model capacity in machine learning), might leads to overfitting the training dataset
- Training error goes down while validation error rises
- Often occurs after several passes over the data



[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Training of a neural network on MNIST dataset
- Objective over training decreases, while validation begins to increase again at round 20 epochs

Early stopping

- Allow to obtain a better model with better validation error (so hopefully better test error)
- Use the parameters $\boldsymbol{\theta}$ many iterations ago, where the validation error started to increase
- Simple to implement: save parameters along iterations, and compute validation error every τ steps
- Each time validation improves, store a copy of the parameters
- If validation hasn't approved for some time, stop and return the last saved $\boldsymbol{\theta}$

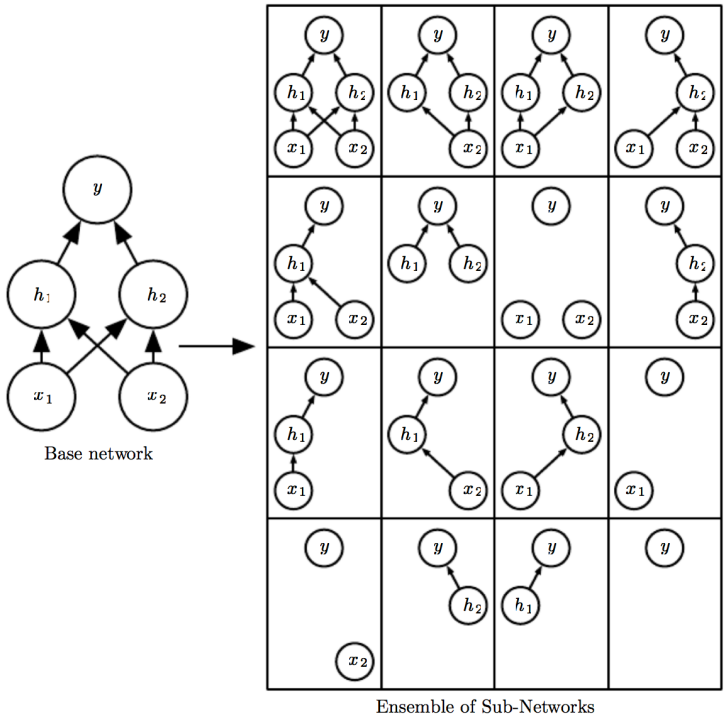
Early stopping

- Probably the most widely used form of regularization in DL: very simple and efficient
- Number of iterations becomes a meta-parameter of the algorithm: can be saved an re-used later, as the level of penalization can be
- For least-squares regression and gradient descent, early stopping can be seen to be equivalent to ridge penalization on the weights

Dropout

- Another very popular form of regularization is dropout
- Idea is to train an ensemble of sub-networks formed by removing non-output units from the base network
- Dropout simply removes a hidden unit by multiplying its output by 0 with probability p (often $p = 1/2$)
- Some similarities with bagging (model averaging)
- Patented by Google! (but we don't care)

Dropout



[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Sixteen possible subsets
- Only keep the ones with connections between the input and output

Playground time !!!

http://playground.tensorflow.org

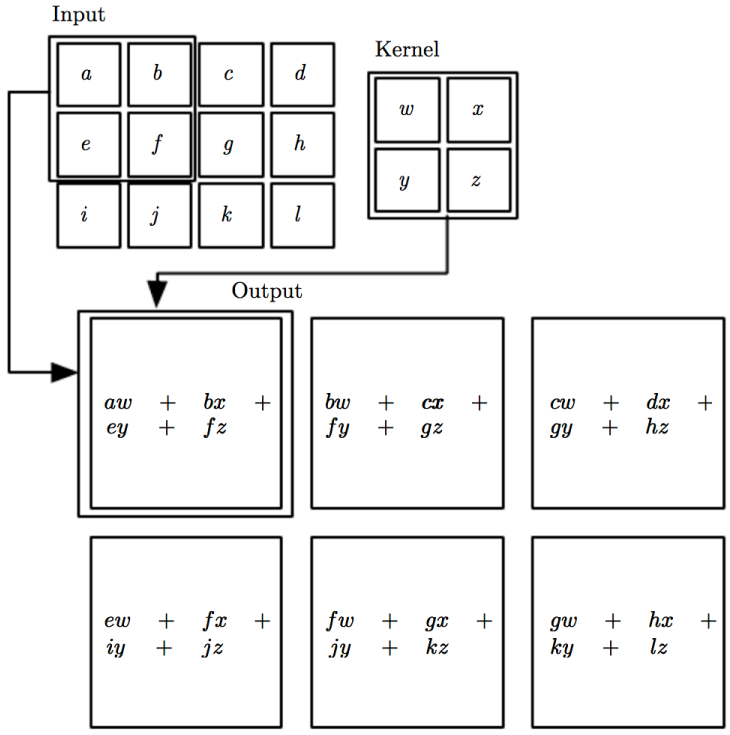
Convolutional neural networks (CNNs)

- Neural networks that use **convolution instead of matrix product** in one of the layers
- A CNN layer typically includes 3 operations: **convolution**, **activation** and **pooling**
- Using the more general idea of **parameters sharing**, instead of **full connection** (convolution instead of matrix product)

Convolution operator in neural networks is as follows

$$S(i,j) = (I \star K)(i,j) = \sum_k \sum_l I(i+k, j+l) K(k,l)$$

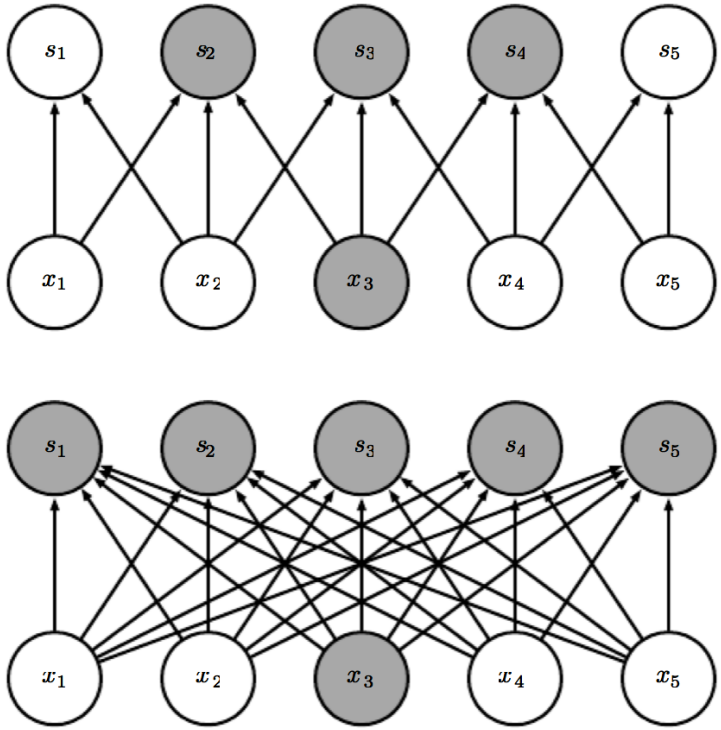
- I is the input and K is called the kernels
- The kernel K will be **learned** (replaces the weights $\boldsymbol{W}$ in a fully connected layer)



[from *Deep Learning*, Goodfellow, Bengio and Courville]

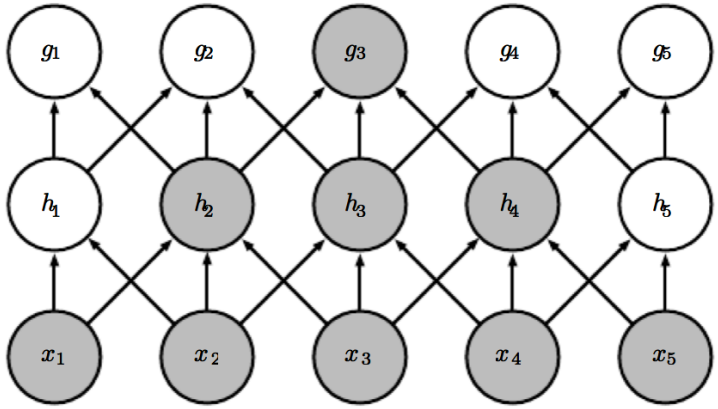
- Example of 2D convolution
- Positions are restricted to positions where the kernels lies within the image
- This is called "valid" convolution

- When using a matrix product, all input and output units are connected
- Replacing the matrix product by a convolution restricts the connections, when the size of the kernel is smaller than the input
- Input image contains millions of pixel values, but we want to detect small meaningful features such as edges with kernels that use only few hundred of pixels
- A lot less parameters, improves memory and statistical efficiency, and faster computations



[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Top: in a convolution with a kernel of width 3, only three outputs are affected by the input x. We say that the **connectivity is sparse**
- Bottom: when using matrix multiplication, all outputs are connected to an input. We say that **connectivity is dense**



[from *Deep Learning*, Goodfellow, Bengio and Courville]

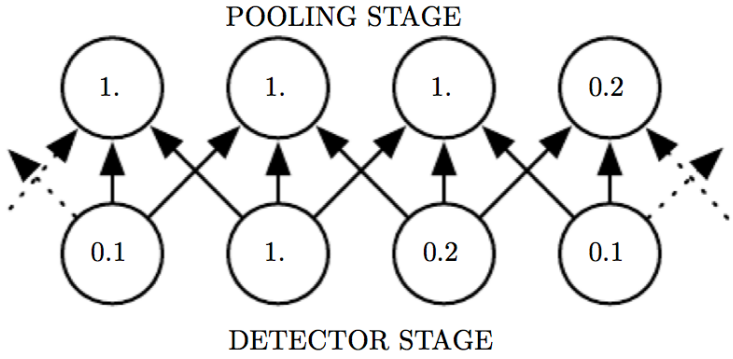
- Deeper layers are indirectly connected to the whole input image

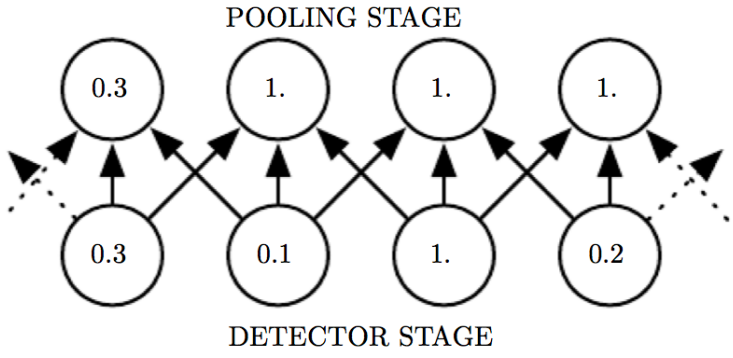
Convolutional layers

- When using a convolution, the particular form of parameter sharing causes the layer to have a property of **equivariance to translation**
- If the input is changed by a translation, then the output changes in the same way
- If we move an object inside an image, the output will change accordingly
- Useful when we know that applying the same function of a small number of neighboring pixels at different places of the image might lead to nice features
- Sometimes, it is not: if images are cropped and centered on individuals' faces for instance

Pooling

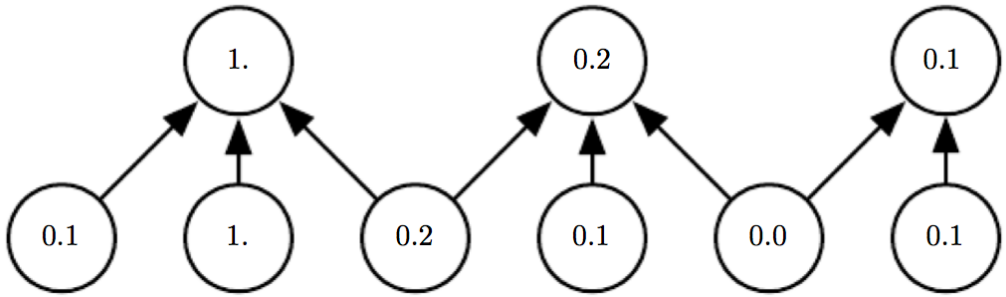
- Pooling replaces the output at a certain location by a summary statistic of neighboring outputs
- The most widely used is the max aggregation, called max-pooling
- Pooling helps the representation to become approximately invariant to small translations of the input
- If a small translation is applied, output of the layer is almost unchanged
- Very useful is we care more about the presence of some feature than its position in the image: for face detection (presence of eyes is more important than where they are)
- Pooling also allows to handle inputs with different sizes: pictures can have different sizes, but the output classification layer must be of fixed size





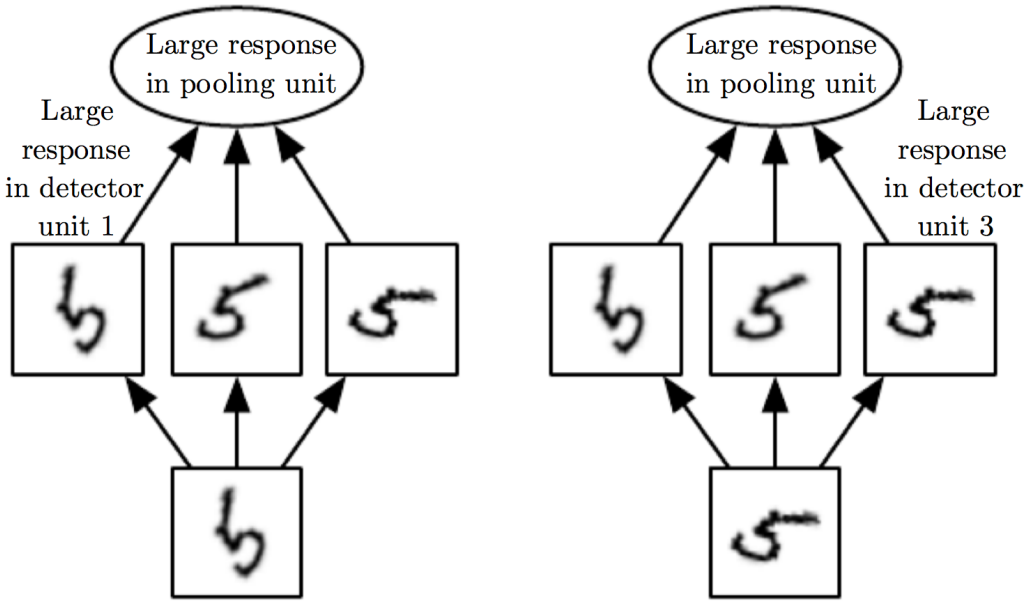
[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Max pooling introduces invariance
- In this example: a stride of one pixel is applied in the input
- Output of max-pooling is almost invariant



[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Max pooling with a pool of width 3 and a stride between pools of 2
- Allows to reduce the representation by a factor of 2, hence reducing the computational and statistical dimension in the next layer

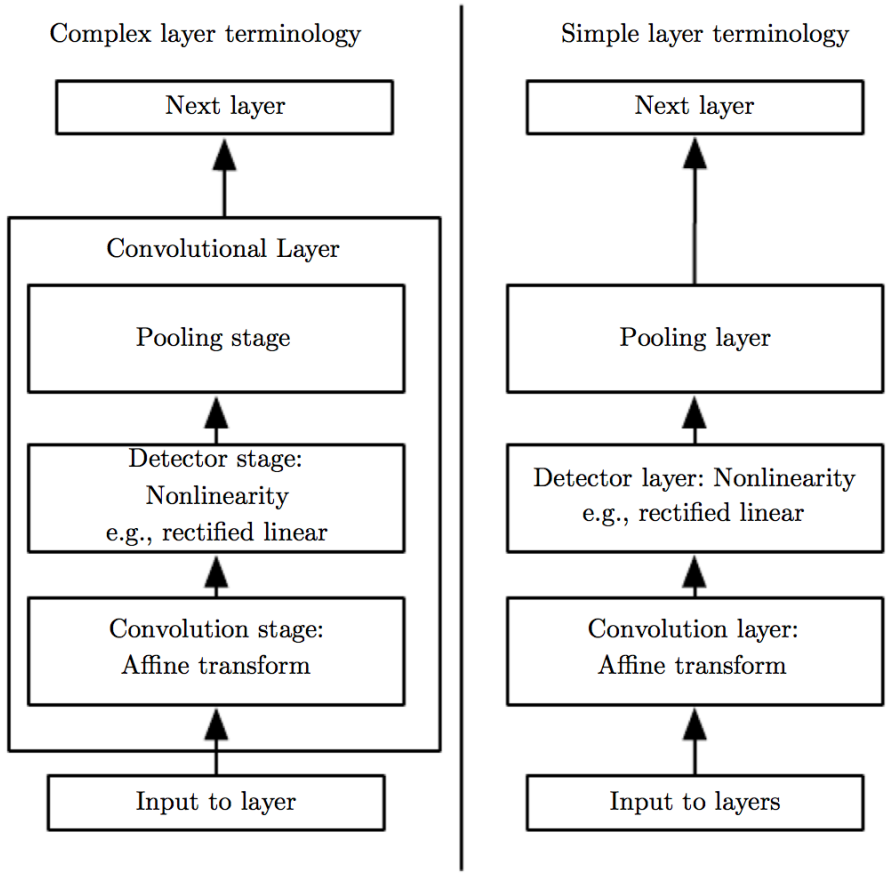


[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Examples of learned invariances
- A pooling that pools over multiple features with separate parameters can learn to be invariant to transformations of the input

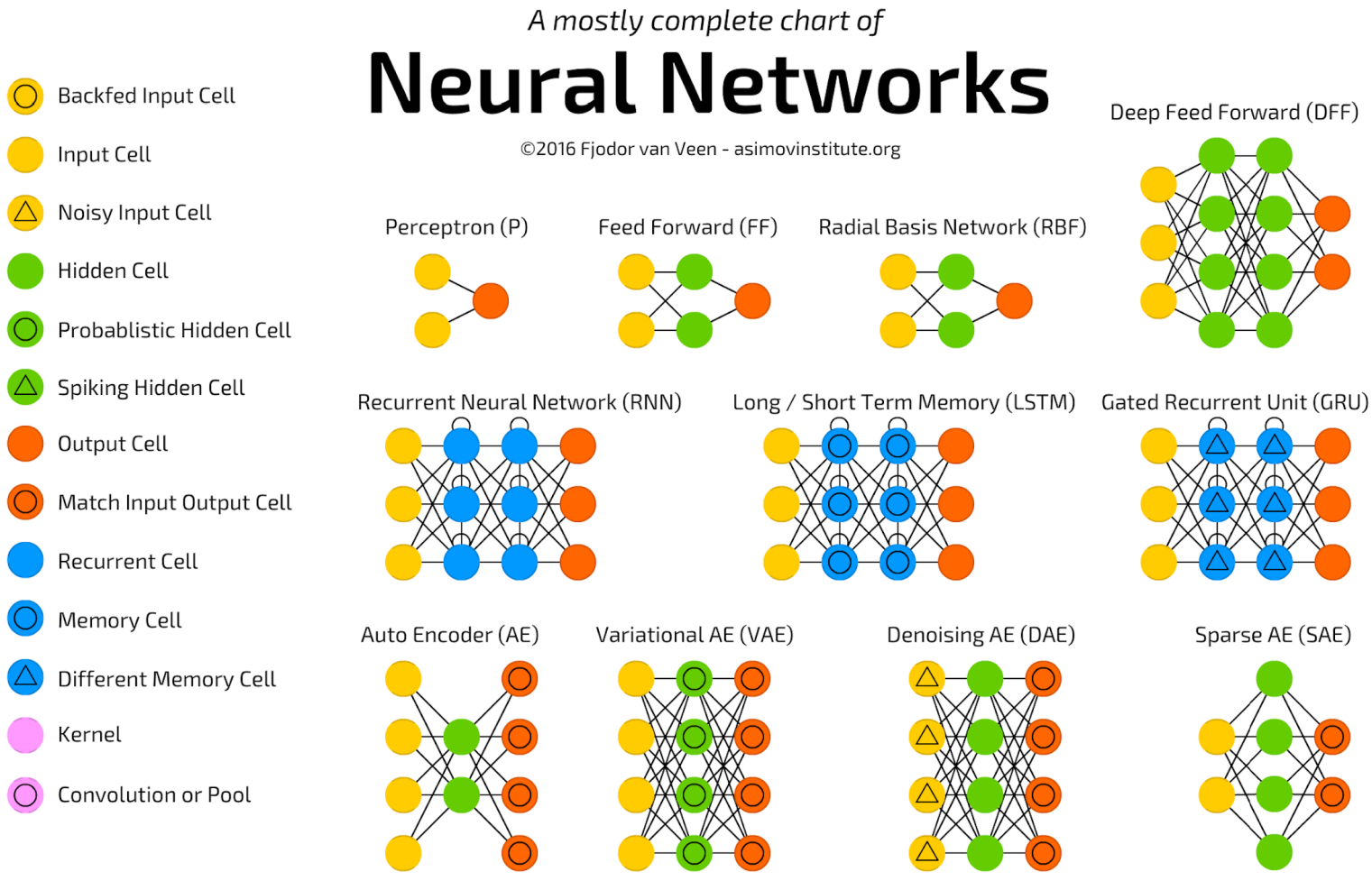
Convolutional neural network

- A typical layer of a convolutional network contains **three stages**
- **Convolution**: many convolutions done in parallel to produce sets of linear activations
- **Detection**: applying a non-linear activation such as the relu
- **Pooling**: modify the output layer even further, by doing local aggregations of inputs

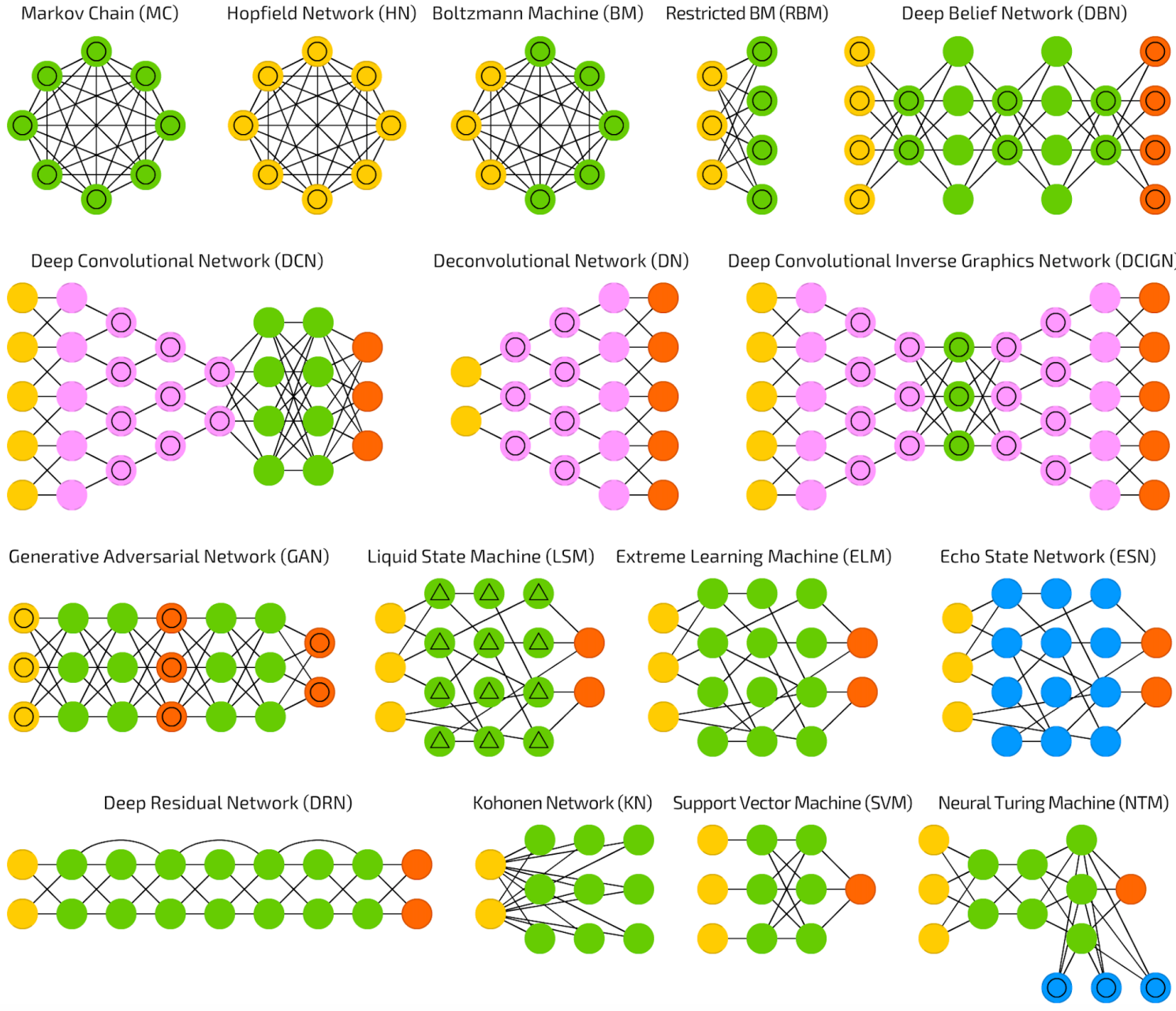


[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Layer of a CNN is made of three stages



Many many other neural networks...



Many many other neural networks...

Thank you!