Introduction to Neural Networks and Deep Learning

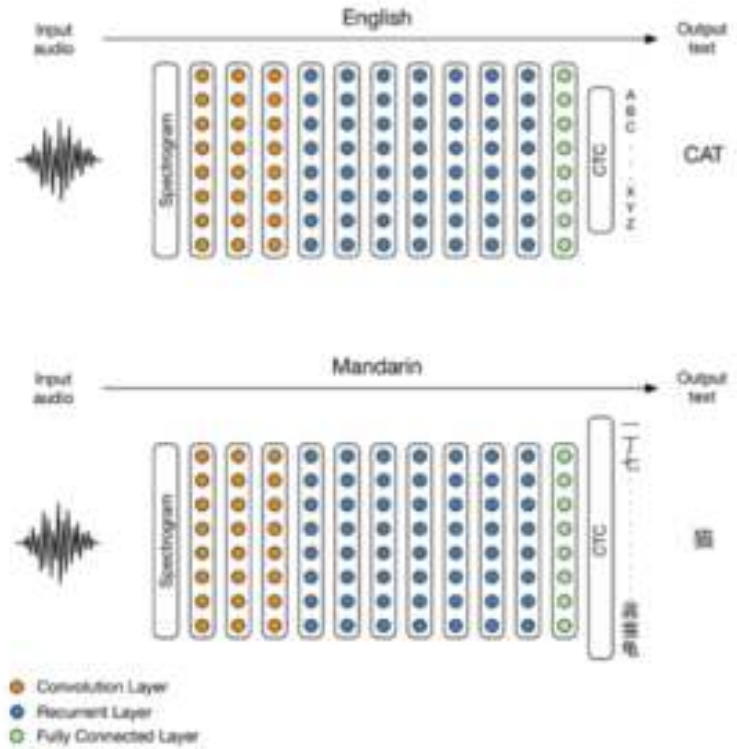
B. Michel

Ecole Centrale de Nantes

Statistical Learning

Outline

Deep Learning and Speech Recognition



[Baidu 2014]

(Source: Krizhevsky)

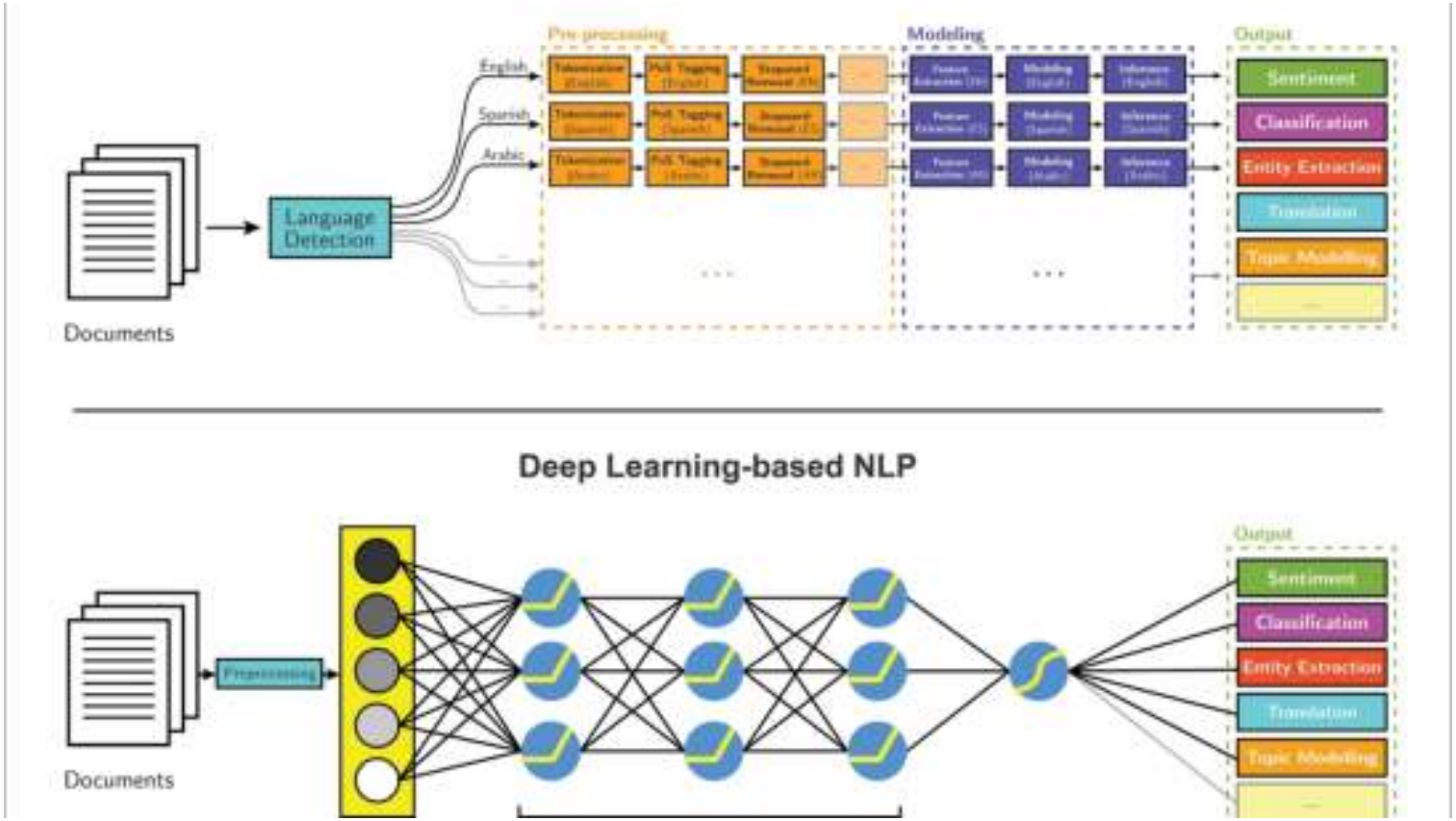
Deep Learning and Computer Vision

- Classification, segmentation, object detection, localization ...

[Source: Krizhevsky]

- Application to Healthcare, but also to person re-identification

Deep Learning and Natural Language Processing



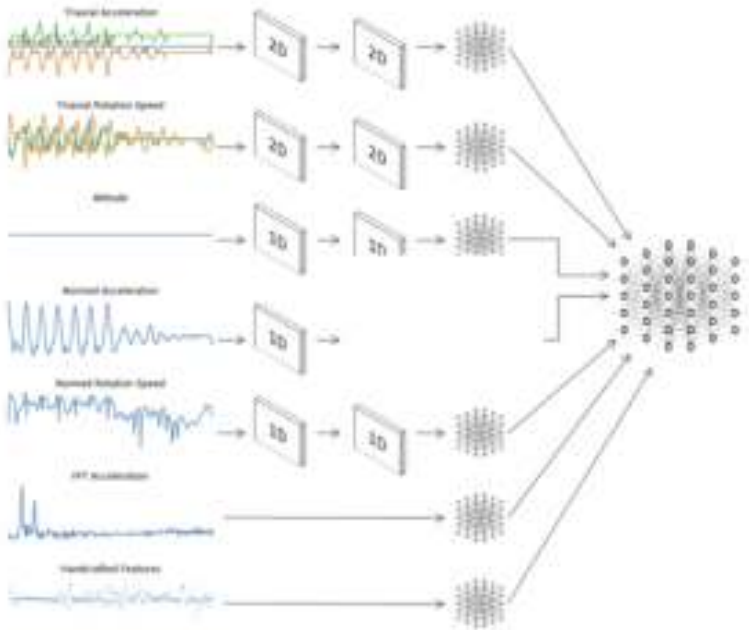
[Source: S. Pulman]

References

- Deep learning book. Ian Goodfellow, Yoshua Bengio and Aaron Courville
- https://github.com/m2dsupsdlclass/lectures-labs, Charles Ollion - Olivier Grisel.
- Hands-on machine learning with scikit-learn and tensorflow. Aurélien Géron, OŔeilly Media, 2017.
- Wikistat lectures and lab.

Deep Learning: Representation Learning and non linear transformation

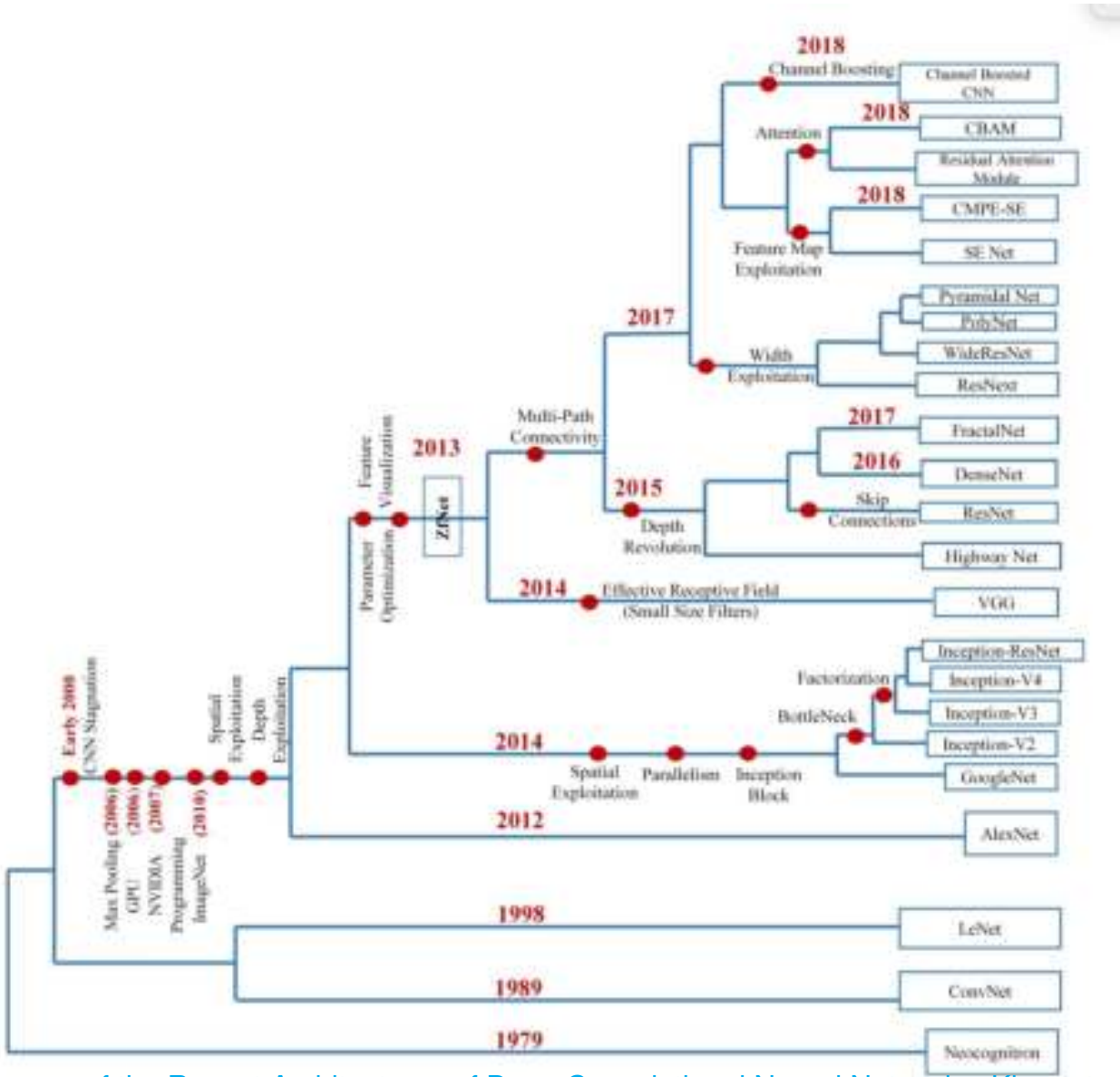
- Deep learning is a set of learning methods attempting to model data with complex architectures combining different transformations.
- **Compositions** of functions.
- **Non-linear** transformations.
- **Representation learning** instead of feature engineering.



Deep Learning are Neural Networks

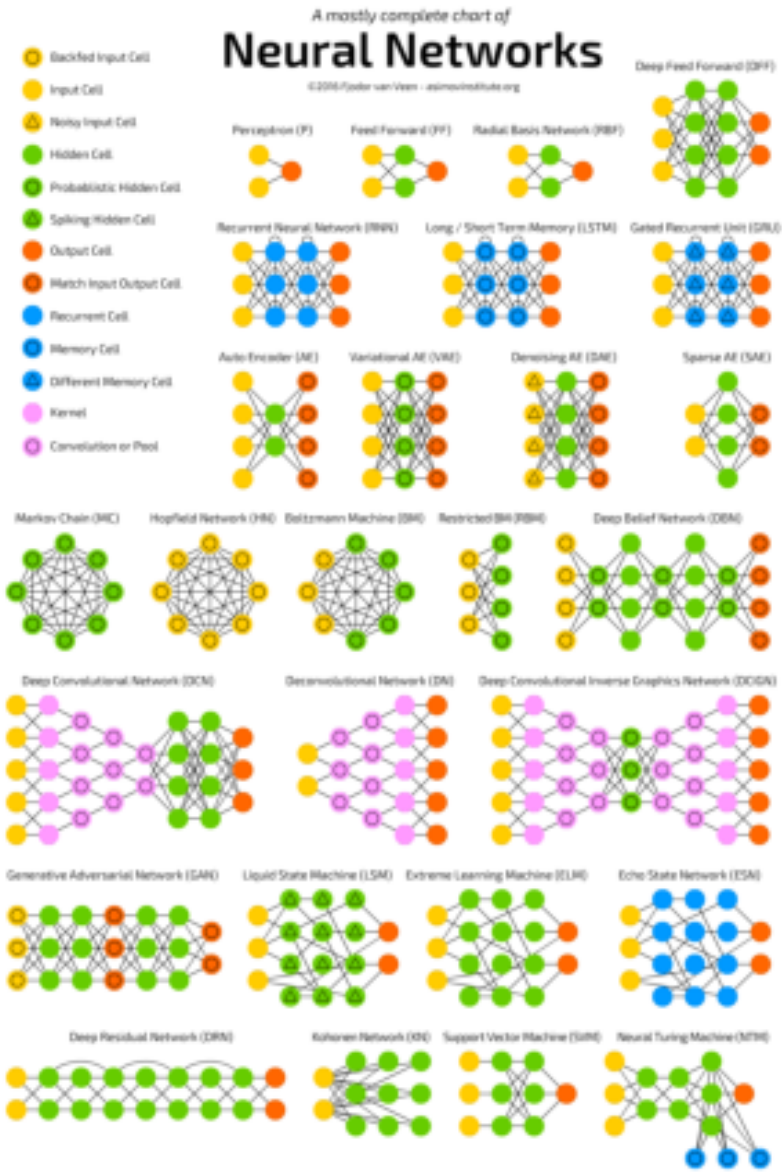
- The elementary bricks of deep learning are the Neural Networks (NN), that are combined to form the deep neural networks.
- Started with the perceptron (50s, Frank Rosenblatt)
- The Thinking Machine (Artificial Intelligence in the 1960s) : video link.
- Deep Learning Networks (DLN) are good old Neural Networks but with more data, more layers, more computational powers (GPU clusters).
- Their length (number of layers) and width (dimension of the hidden units) have increased over the years.

Evolutionary history of deep CNNs



[source : A Survey of the Recent Architectures of Deep Convolutional Neural Networks, Khan et al. 2019]

Neural-network-zoo

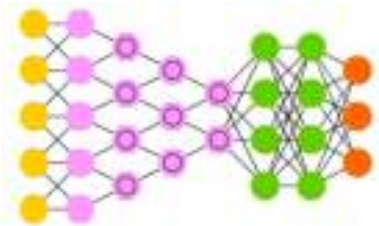


[source : asimovinstitute.org]

Three popular examples of NN

- Feed-forward neural networks (FFNN)

- Convolutional neural networks (CNN)



- Recurrent neural networks (RNN)

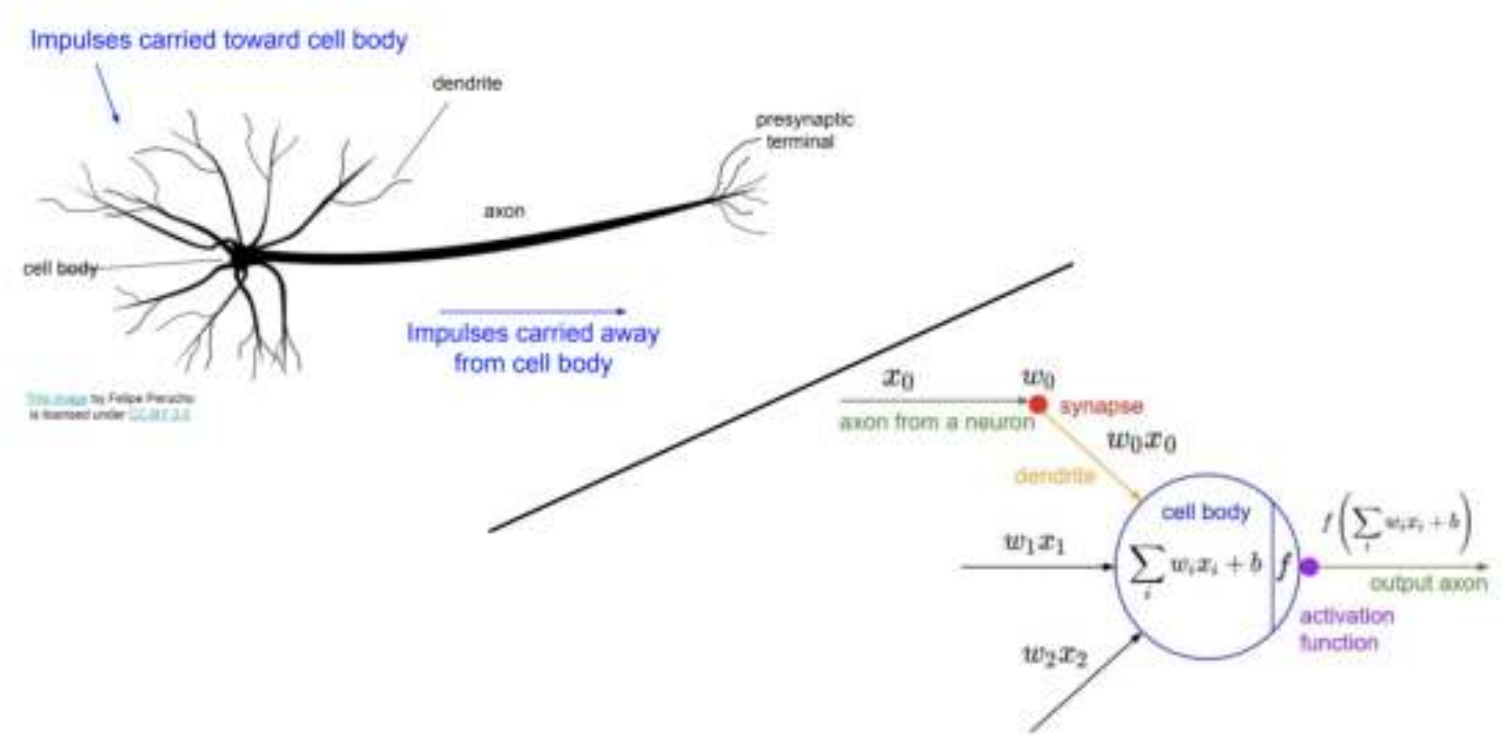
[source : asimovinstitute.org]

Deep Learning for Everyone and (Almost) Free

- The field has grown to the extent where sophisticated software packages are available in the public domain, many produced by high-profile technology companies.
- Chip manufacturers are also customizing their graphics processing units (GPUs) for kernels at the heart of deep learning.
- Open source and free ... but for Amazon, Microsoft, and Google it is all about the competition for customers of their cloud services.

Outline

Biological inspiration



[Source: Stanford's CS231n]

- But analogy between computational and biological neurons is a very rough analogy.

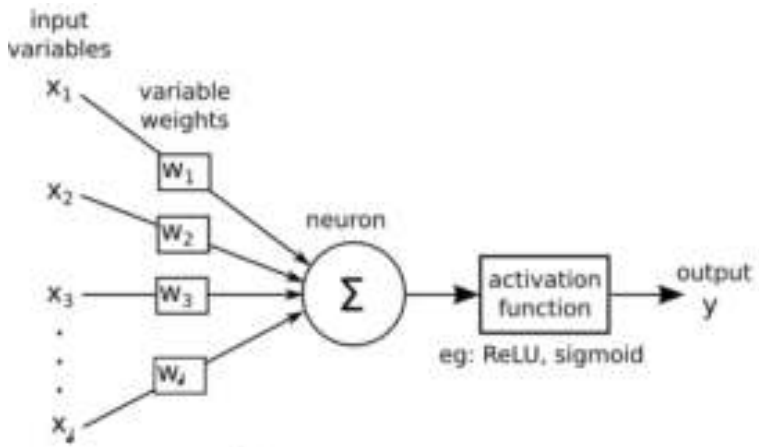
Artificial Neuron

- An artificial neuron is a function f such that

$$y = f(x) = g(\langle w, x \rangle + b)$$

  - input $x = (x_1, \ldots, x_d)$,
  - vector of connection weights $w = (w_1, \ldots, w_d)^T$,
  - neuron bias b,
  - associated to an activation function g.

- Schematic representation of an artificial neuron where $\Sigma = \langle w, x \rangle + b$:



[Source: andrewjames turner.co.uk]

- Activation function : is a **non linear** transformation.

Activation functions

Most popular activation functions:

- Sigmoid function (logistic σ) :

$$g(x) = \frac{1}{1 + \exp(-x)}$$

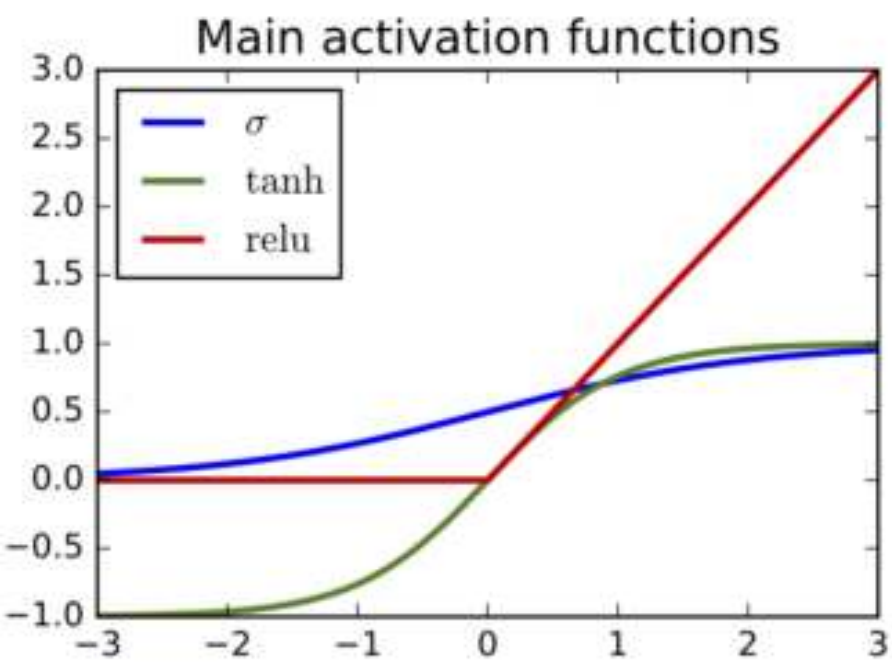
- Hyperbolic tangent function (tanh):

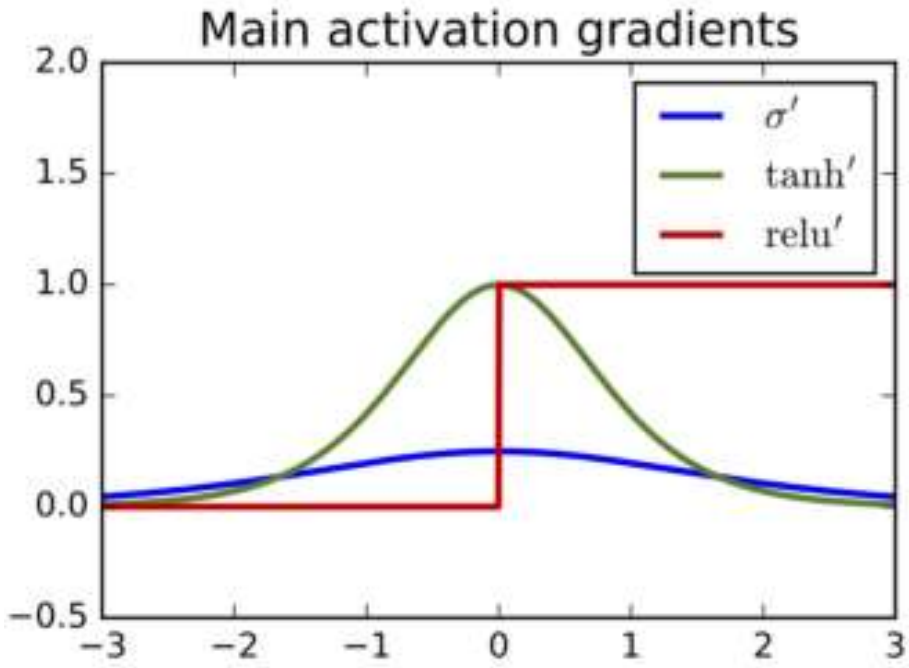
$$g(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = \frac{\exp(2x) - 1}{\exp(2x) + 1}.$$

- Rectified Linear Unit (ReLU) activation function:

$$g(x) = \max(0, x).$$

Activation functions



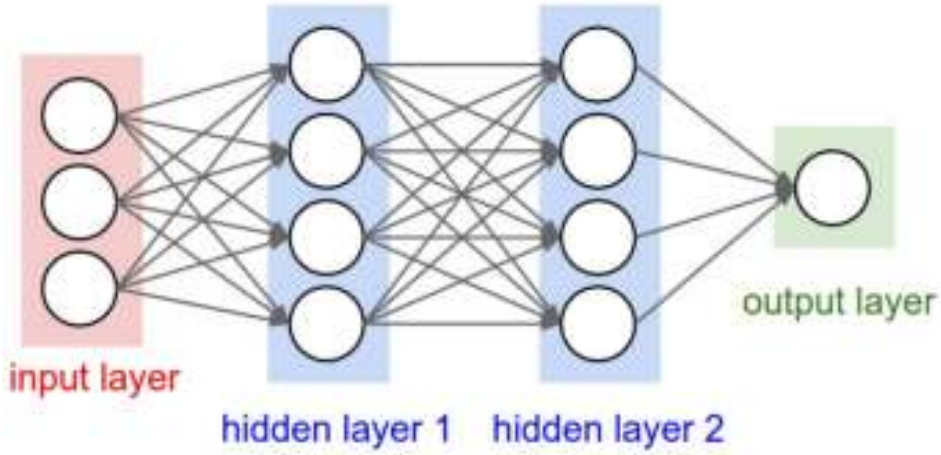


[Source: Gaiffas]

Activation functions

- Sigmoid :
  - Differentiable and keeps values in $[0, 1]$,
  - Historically was the mostly used activation function,
  - But its gradient is very close to 0 when $|x|$ is not close to 0
    $\rightarrow$ causes troubles for the backpropagation algorithm (see further).
- RELU is now the most common activation function
  - Non linear
  - Not differentiable in 0 (not a problem since during training, it's unlikely that many inputs equal 0)
  - Sparsification effect: ReLU and ReLU' are equal to 0 for negative values
  - Easier to optimize since behavior is closer to linear

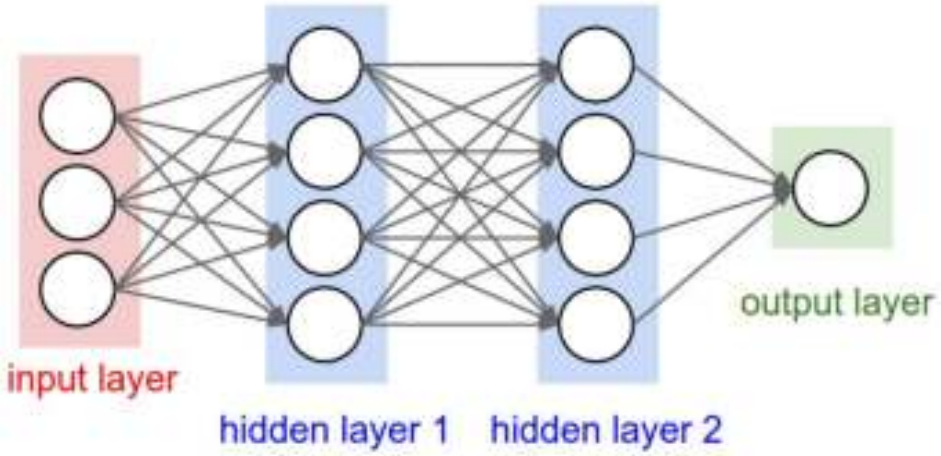
Feedforward neural network



[Source: Standford cs231n]

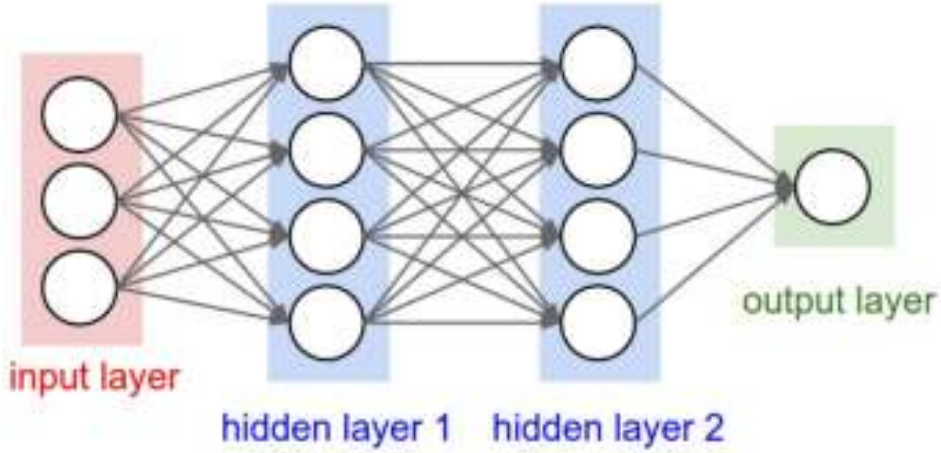
- A **feedforward** neural network is an artificial neural network wherein connections between the units do not form a cycle (different from **recurrent** neural networks).
- The feedforward neural network was the first and simplest type of artificial neural network devised.
- Example : multilayer perceptron (neural network).

Multilayer perceptron



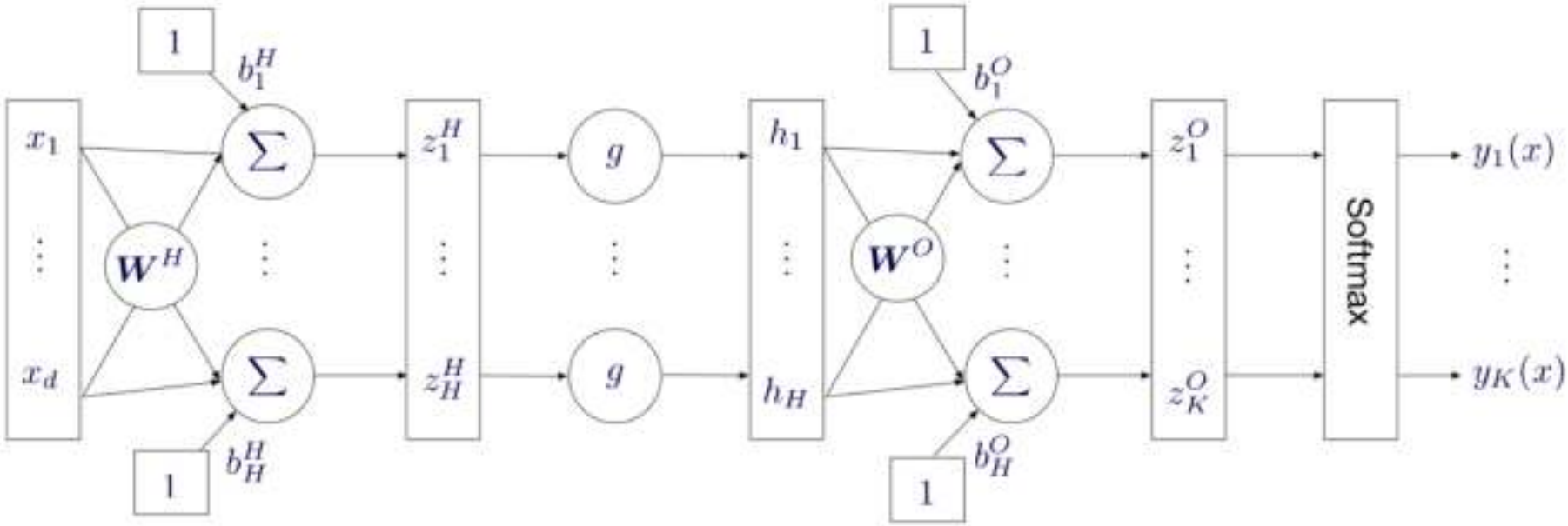
- A **multilayer perceptron** is a structure composed by several hidden layers of neurons where the output of a neuron of a layer becomes the input of a neuron of the next layer.
- Layers correspond to functions that are **composed** in some order: $f(x) = f_3(f_2(f_1(x)))$ has 3 layers.

Multilayer perceptron



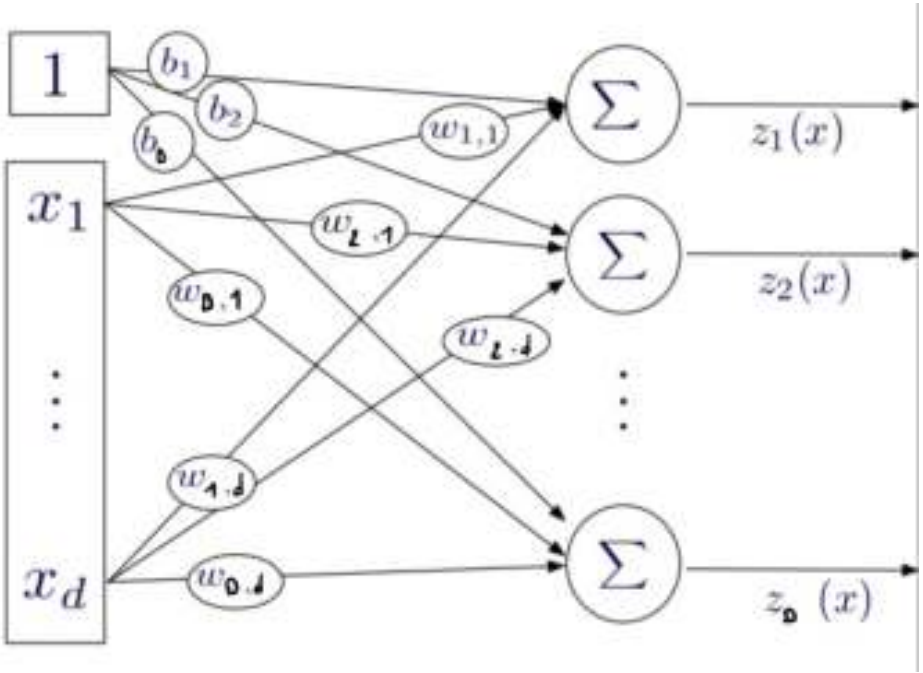
- Each **layer** has several artificial neurons (2^j neurons) and an activation function.
- Typically **deceasing** number of neurons per layers along the network.
- Activation functions (for hidden layers and for the output layer) have to be chosen by the user.

Let's look at a one-hidden layer network



[Source: Gaiffas]

Pre-activations for the hidden layer

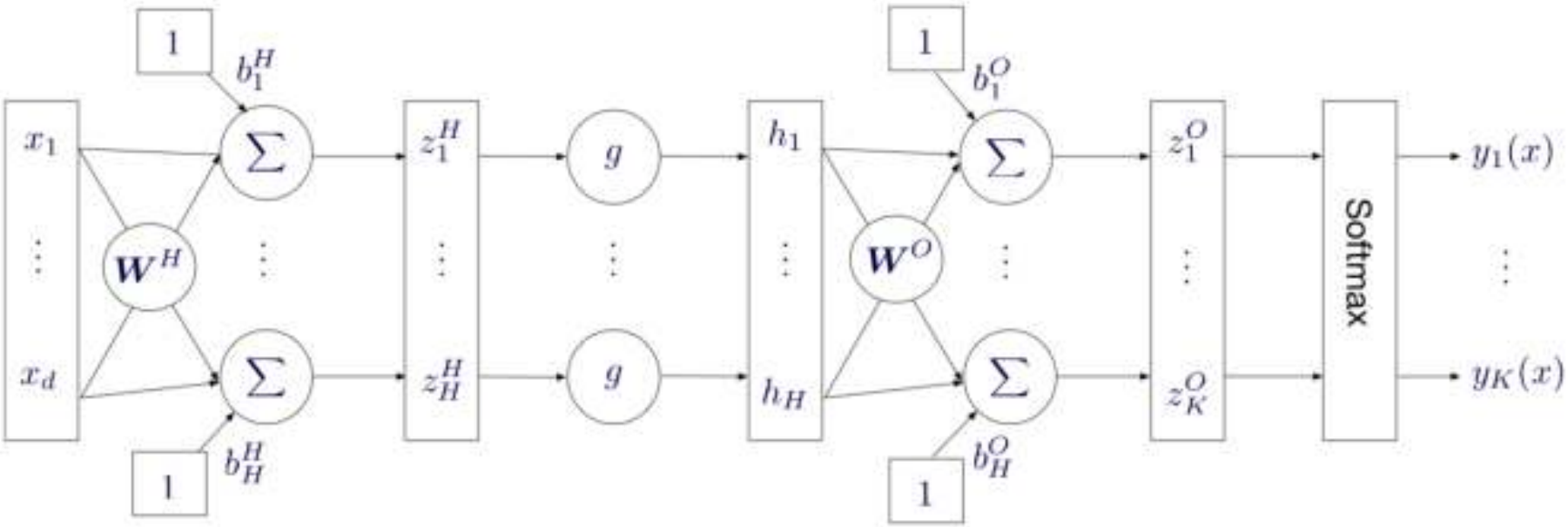


[Source: Gaiffas]

- x input
- One vector of weighs w_i for each neuron $i = 1 \dots D$.
- Stack the weights in a $D \times d$ matrix W with $W_{i,\bullet} = w_i^T$.
- Pre-activations (or logits): for $i = 1 \dots D$: $z_i(x) = \langle W_{\bullet,i}, x \rangle + b_i$ and the matrix expression of the layer is

$$z(x) = Wx + b$$

Activation for the hidden layer



- Pre-activation : $z(x) = Wx + b$.
- Activation : $h(x) = g(z(x))$.
- E.g sigmoïd, Relu, tanh ... (non linear).
- If $g = Id$, then the hidden layer reduces to a linear transformation.

Activation functions for the output layer

- For regression:
  - The output layer has width 1
  - No activation function, i.e. $g(z) = z$.
- For binary classification:
  - The output layer has width 1
  - Sigmoid activation function to give a prediction of $P(Y = 1|X = x)$
- For multi-class classification:
  - The output layer contains one neuron per class k giving a prediction of $P(Y = k|X = x)$.
  - The multidimensional function $\mathrm{Softmax}$ is generally used :

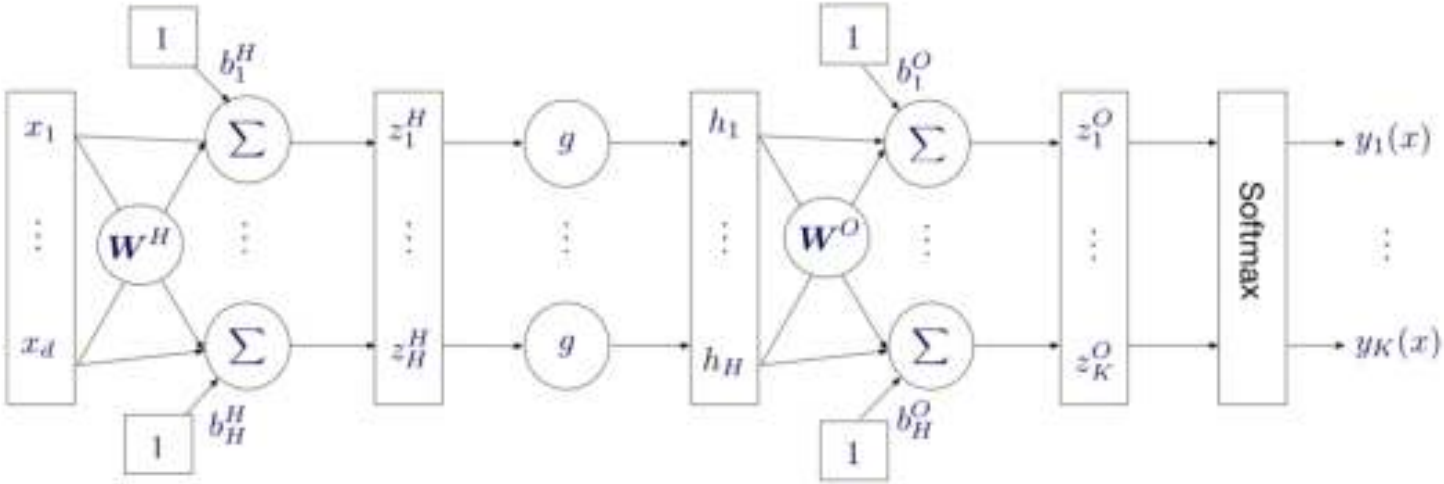
$$\mathrm{Softmax}(z)_k = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)}.$$

Softmax

$$\text{Softmax}(z)_k = \frac{1}{\sum_{j=1}^{K} \exp(z_j)} \begin{bmatrix} \exp(z_1) \\ \vdots \\ \exp(z_K) \end{bmatrix}$$

- Softmax is not an element-wise activation : it maps $z \in \mathbb{R}^K$ to the space of vector in $[0,1]^K$ **with entries that sum to 1**.
- The inputs of the softmax is called the “logits" in deep learning.

One-hidden layer network (Softmax output layer)

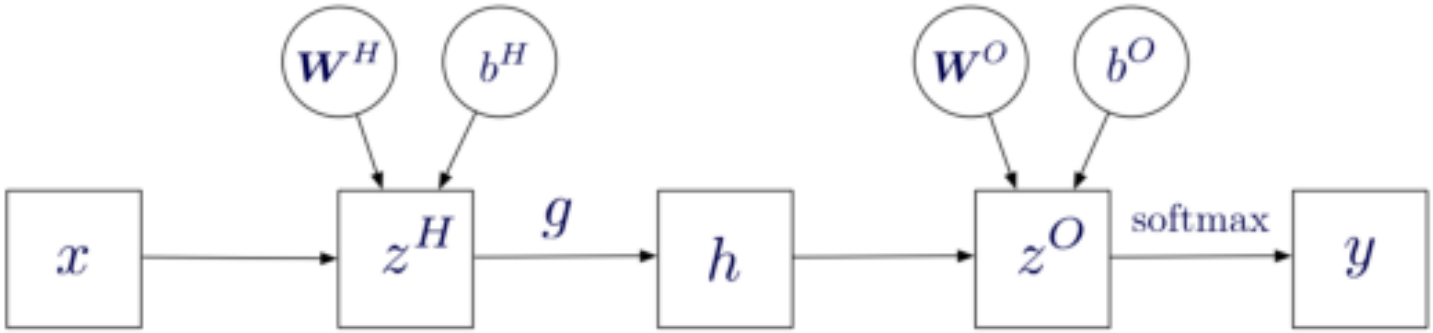


Mathematical formulation of a one-hidden layer (width H) with softmax output:

$$\begin{aligned} y(x) &= \text{Softmax}(z^{(O)}) = \text{Softmax}(W^{(O)}h + b^{(O)}) \\ &= \text{Softmax}(W^{(O)}g(z^{(H)}) + b^{(O)}) \\ &= \text{Softmax}\left(W^{(O)}g(W^{(H)}x + b^{(H)}) + b^{(O)}\right) \end{aligned}$$

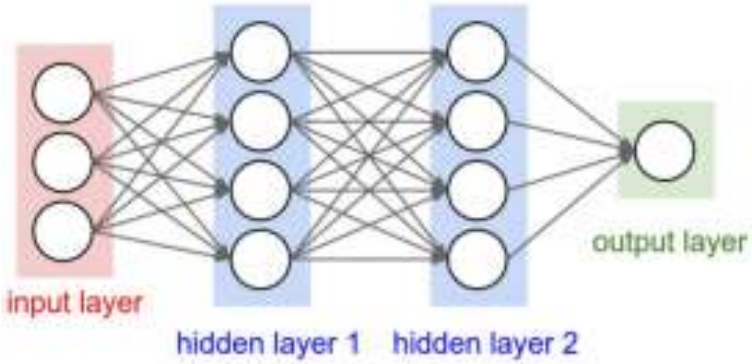
One-hidden layer network (Softmax output layer)

Alternative representation of this one-hidden layer network :



[Source: Gaiffas]

Multilayer perceptron : math. formulation



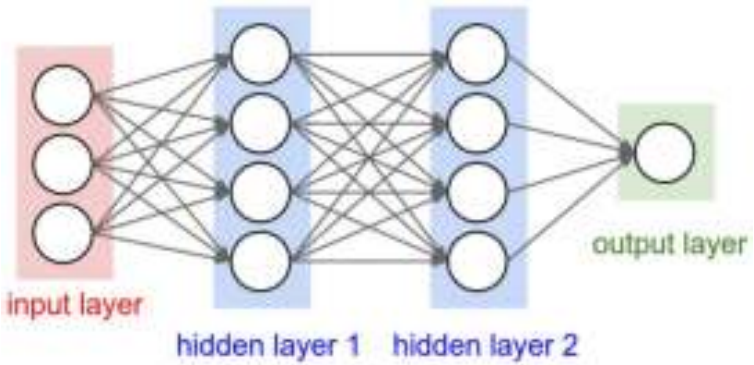
Multilayer perceptron with L hidden layers.

For Hidden layer l:

- Input of layer l is the output of layer $l-1$ namely $h^{(l-1)}(x)$. Set $h^{(0)}(x) = x$ for the input layer ($l = 0$).
- Pre-activations for layer l:
  - Weight matrix $W^{(l)}$ has dimension: (width of layer $l-1$) $\times$(width of layer l)
  - Bias vector $b^{(l)}$ has dimension the width of layer l.
  - Pre-activation is the vector $z^{(l)}(x) = b^{(l)} + W^{(l)} h^{(l-1)}(x)$.
- Activation for layer l:

$$h^{(l)}(x) = g^{(l)}\left(b^{(l)} + W^{(l)} h^{(l-1)}(x)\right)$$

Multilayer perceptron : math. formulation



Multilayer perceptron with L hidden layers.

For Output layer $l = L + 1$:

$$\begin{aligned} h^{(L+1)}(x) &= g^{(L+1)}\left(z^{(L+1)}(x)\right) \\ &= g^{(L+1)}\left(W^{(L+1)}h^{L}(x) + b^{(L+1)}\right) \end{aligned}$$

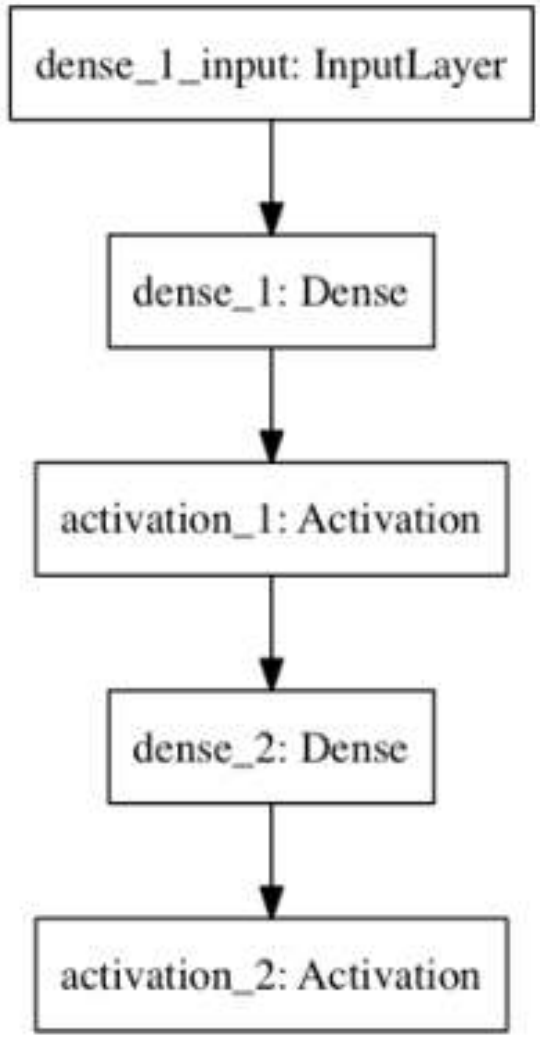
Layers with Keras :

Algorithm 1 Simple code for defining a one hidden layer network with Keras.

```
from keras.models import Sequential

model = Sequential()

model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
model.add(Dense(10, input_dim=32))
model.add(Activation('softmax'))
```



Universal Approximation Theorem

Theorem (Hornik 1991)

Let g be a bounded, continuous and non decreasing (activation) function. Let K_d be some compact set in $\mathbb{R}^d$ and $\mathcal{C}(K^d)$ the set of continuous functions on K_d . Let $f \in \mathcal{C}(K_d)$. Then for all $\varepsilon > 0$, there exists $N \in \mathbb{N}$, real numbers v_i , b_i and $\mathbb{R}^d$-vectors w_i such that the function

$$F(x) = \sum_{j=1}^{N} v_j g(\langle w_j, x \rangle + b)$$

satisfies

$$\forall x \in K_d, \ |F(x) - f(x)| \leq \varepsilon.$$

That's good news, but :

- It is believed that several layers with a smaller width lead to better generalization (more abstract layers)
- This result does not say how to choose the parameters : in practice we have to train the network parameters.

Outline

Main tools

- Main tools to train FFNN and Deep NN :
  - **Forward-propagation** to compute the loss and to make predictions,
  - **Back-propagation** to compute gradients (which also requires forward-propagation)
  - **Stochastic optimization algorithms**.
- Main (open source) libraries : Pytroch , tensorflow,keras...
- GPU computing

Outline

Maximum likelihood approach

- Maximum likelihood and least squares approaches for estimating the weights of the network (bias terms, weigh terms).
- Vector of parameters for one given network : $\theta \in \mathcal{T}$.
- Training set (X_i, Y_i), $i = 1, \ldots, n$.
- We define loss, risk and empirical risk for each framework.

Loss for Binary Classification $Y \in \{0, 1\}$

- The functions $\{f(\cdot, \theta)\, \theta \in \mathcal{T}\}$ can be computed by the network.
- We want to choose θ such that $f(\cdot, \theta)$ is as close as possible to the (true) conditional probability $P_\theta(Y = 1|X = x)$.
- How to "compare" an observation Y with a given conditional probability $f(\cdot, \theta)$?
- Cross entropy between (discrete) distributions p and q:

$$\ell(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

- In our case : $p = \delta_Y$ and $q = (q_0, q_1) = (1 - f(X, \theta), f(X, \theta))$ two distributions on $\{0, 1\}$:

$$\begin{aligned} \ell(Y, q) := \ell(\delta_Y, q) &= -\left[\mathbb{1}_{Y=1} \log q_1 + \mathbb{1}_{Y=0} \log(1 - q_1)\right] \\ &= -\left[Y \log q_1 + (Y - 1) \log(1 - q_1)\right] \end{aligned}$$

  - the loss is close to zero when $y = 1$ with q_1 close to one.
  - the loss is large when $y = 1$ with q_1 close to zero.

Loss for Binary Classification $Y \in \{0, 1\}$

- The cross entropy risk of $f(\cdot, \theta)$ for $\theta \in \mathcal{T}$ is

$$\mathcal{R}(\theta) \quad := \quad \mathbb{E}_{X,Y}\left[\ell(Y, f(X, \theta))\right]$$

- Empirical risk minimization with cross entropy:

$$\hat{\theta} \in \operatorname*{argmin}_{\theta \in \mathcal{T}} \hat{\mathcal{R}}_n(\theta) := \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f(X_i, \theta))$$

and

$$\hat{\mathcal{R}}_n(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \left[Y_i \log f(\theta, X_i) + (1 - Y_i) \log(1 - f(\theta, X_i))\right]$$

- Remember that $f(\theta, x)$ represents or approximates the probabilities $P(Y = 1 | X = x)$.
- We see that minimizing the empirical risk is maximizing the log-likelihood of the conditional distribution $(Y|X)$

Loss for Multi-class Classification

- Multi-class: $Y \in \{1, \ldots, K\}$.
- Let $f(\cdot, \theta) \in (0,1)^K$ for $\theta \in \mathcal{T}$ be the output of the network.
- The aim is to approximate the conditional probabilities $(P_\theta(Y = 1|X = x), \ldots, P_\theta(Y = K|X = x))^T$ by $f(\cdot, \theta)$.
- Cross entropy for comparing the observation Y with a given probability q on $\{1, \ldots, K\}$:

$$\ell(\delta_Y, q) = -\sum_{k=1}^{K} \mathbb{1}_{Y=k} \log q_k$$

- Cross entropy risk:

$$\mathcal{R}(\theta) \quad = \quad \mathbb{E}_{X,Y}\left[\ell(Y, f(X, \theta))\right]$$

Loss for Multi-class Classification

- Empirical cross entropy risk :

$$\begin{aligned}\hat{\mathcal{R}}_n(\theta) &= \frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f(X_i,\theta)) \\ &= -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\mathbb{1}_{Y_i=k}\log(f(X_i,\theta)_k)\end{aligned}$$

- Remember that the K-vector $f(\theta, x)$ approximates $(P(Y=1|X=x),\ldots,P(Y=K|X=x))$.
- Minimizing the empirical risk is maximizing the log-likelihood of the conditional distribution $(Y|X)$.

Loss for regression

- Regression: $Y \in \mathbb{R}$.
- The output of the network gives a prediction of Y (no output activation).
- Quadratic loss, associated risk and empirical risk.

Penalized empirical risk

- The parameter θ is high dimensional, in particular for Deep Neural Networks.
- We add a regularization term to the empirical risk :

$$\hat{\theta} \in \underset{\theta \in \mathcal{T}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f(X_i, \theta)) + \lambda \Omega(\theta)$$

- L^2 regularization :

$$\Omega(\theta) = \sum_{l} \sum_{i} \sum_{j} |W_{i,j}^{(l)}|_2^2 = \sum_{l} \|W_{i,j}^{(l)}\|_F^2$$

where $W^{(l)}$ is the weight matrix for layer l and $\|W\|_F$ is the Frobenius norm of W.

- L^1 regularization (parcimonious):

$$\Omega(\theta) = \sum_{l} \sum_{i} \sum_{j} |W_{i,j}^{(l)}|.$$

Outline

Training

- For minimizing

$$F_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(Y_i, f(X_i, \theta)) + \lambda\Omega(\theta),$$

  we apply a gradient descent algorithm.

- It requires :
  - **Forward propagation** to compute $f(x, \theta)$
  - **Backward propagation** to compute the gradients of the empirical risk.

Forward Propagation to compute predictions

Given :

- a mini-batch (x_i, y_i) for $i \in B$ (or the full dataset).
- model weights θ : $W^{(l)}$ and $b^{(l)}$ for $l = 1, \ldots, L+1$.
- X : $|B| \times d$-matrix with rows x_i. Set $H^{(0)} = X$.
- Y : $|B| \times K$-matrix with rows $(y_i = 1, \ldots, y_i = K)$.

Algorithm 2 Forward Propagation algorithm.

---

for $l = 1$ to L **do**
  Compute $Z^{(l)} = b^{(l)} + W^{(l)} H^{(l-1)}$
  Compute $H^{(l)} = g^{(l)}(Z^{(l)})$
end for
Compute $Z^{(L+1)} = b^{(L+1)} + W^{(L+1)} H^{(L)}$
Compute $\hat{Y} = \text{Softmax}(H^{(L+1)})$ (for classification)
Compute $F_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, \hat{Y}_i) + \lambda \Omega(\theta)$
return $F_n(\theta)$, $Z^{(l)}$, $H^{(l)}$, $l = 1 \ldots L$.

- The activations and pre-activations values of the hidden layers will be also used for computing the gradients of the loss.

We need to compute the gradients...

- Let's look at a multilayer perceptron with Softmax output.
- $\ell(f(x,\theta),y) = -\sum_{k=1}^{K} \mathbb{1}_{y=k} \log f(x,\theta)_k$ and the output layer that yields f is softmax.
- For applying a gradient descent algorithm we need to compute $\nabla_\theta \ell(f(x,\theta),y)$, i.e. all the derivatives $\frac{\partial \ell(f(x,\theta),y)}{\partial W_{i,j}^{(l)}}$ and $\frac{\partial \ell(f(x,\theta),y)}{\partial b_j^{(l)}}$ for all the layers (output and hidden layers).
- Gradient of $\mathsf{Softmax}$ for $k, k' \in \{1,\dots,K\}$ is

$$\frac{\partial \,\mathsf{Softmax}(z)_k}{\partial z_{k'}} = \begin{cases} \mathsf{Softmax}(z)_k \times (1 - \mathsf{Softmax}(z)_k) & \text{if } k = k' \\ -\mathsf{Softmax}(z)_k \times \mathsf{Softmax}(z)_{k'} & \text{if } k \neq k' \end{cases}$$

- For computing all the derivatives: chaining rule principle and back propagation algorithm.

Chaining rule

- Functions are composed along the NN :

$$\begin{aligned} z^{(l)}(x) &= b^{(l)} + W^{(l)} h^{(l-1)}(x) \\ h^{(l)}(x) &= g^{(l)}\left(z^{(l)}(x)\right) \end{aligned}$$

- Simple idea : $f_2(f_1(x))' = f_2'(f_1(x)) \times f_1(x)'$.
- Chaining rule : if $\gamma = f_2(f_1(\alpha))$ and $\beta = f_1(\alpha)$ then

$$\frac{\partial \gamma}{\partial \alpha} = \frac{\partial \gamma}{\partial \beta} \frac{\partial \beta}{\partial \alpha}.$$

- Also valid for vectors. For instance :

$$\frac{\partial h^{(l)}}{\partial x_i} = \sum_{j=1}^{J} \frac{\partial h^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial x_i} = \langle \nabla g^{(l)}, \frac{\partial \mathbf{z^{(l)}}}{\partial x_i} \rangle$$

where $\mathbf{z^{(l)}} = (z_1^{(l)}, \ldots, z_J^{(l)})^T$.

Gradients: back-propagation

- Back-propagation performs such Jacobian-gradient products for each operation in the computational graph, respecting the order of the operations.
- First, compute the gradients for the parameters of the output layer and proceed with the hidden layers (from last to the first layer).
- Chaining rule (and back propagation) is not only valid for vectors x, but also matrix and even for 3D, 4D objects (tensors).
- Speed-up the computation with GPU.

Back propagation for softmax multilayer perceptron

Given the current weights θ (weight and bias terms): computing the gradients requires

- a forward propagation pass to compute the pre-activations values, the activations values along and the predicted values $f(X_i, \theta)$.
- a back propagation pass.

Back propagation for softmax multilayer perceptron

Algorithm 3 Back propagation - see WikiStat.

Compute the output gradient $\nabla_{z^{(L+1)}(x)} \ell(f(x,\theta),y) = f(x,\theta) - e(y)$.
for $l = L+1$ to 1 **do**
  • Compute derivatives w.r. to **parameters** of layer l:
    $\nabla_{W^{(l)}} \ell(f(x,\theta),y) = \nabla_{z^{(l)}(x)} \ell(f(x,\theta),y)\,(h^{(l-1)}(x))'$
    $\nabla_{b^{(l)}} \ell(f(x,\theta),y) = \nabla_{z^{(l)}(x)} \ell(f(x,\theta),y)$
  • Compute derivatives w.r. to **pre-activ. and activ.** of layer $l-1$:
    $\nabla_{h^{(l-1)}(x)} \ell(f(x,\theta),y) = (W^{(l)})^T \nabla_{z^{(l)}(x)} \ell(f(x,\theta),y)$
    $\nabla_{z^{(l-1)}(x)} \ell(f(x,\theta),y) =$
      $\nabla_{h^{(l-1)}(x)} \ell(f(x,\theta),y) \odot (\ldots,(g^{(l-1)})'(z^{(l-1)}(x)_j),\ldots)^T$
end for
return $\nabla_\theta \ell(f(x,\theta),y)$

where :

- $e(y) = (\mathbb{1}_{y=1},\ldots,\mathbb{1}_{y=K})$
- $\odot$ is the Hadamard product.

Stochastic gradient descent algorithm (SGD)

- For minimizing $\frac{1}{n}\sum_{i=1}^{n} \ell(Y_i, f(X_i, \theta)) + \lambda\Omega(\theta)$ we apply a gradient descent algorithm.
- For big data sets (and for deep learning), computing the gradient for the complete sample has a prohibitive computational cost.
- **SGD**: at each iteration, the gradient is computed only on a random batch of size *m*.

Algorithm 4 SGD algorithm.

Initialization : choose of θ (all the weights an bias terms)
for $ep = 1$ to ept **do**
  for $t = 1$ to n/mt **do**
    Pick a subsample B of size m with no replacement
    Compute $\theta = \theta - \varepsilon \left[\frac{1}{m}\sum_{i\in B} \nabla_\theta \ell(f(X_i, \theta), Y_i) + \lambda \nabla_\theta \Omega(\theta)\right]$
  end for
end for
return θ

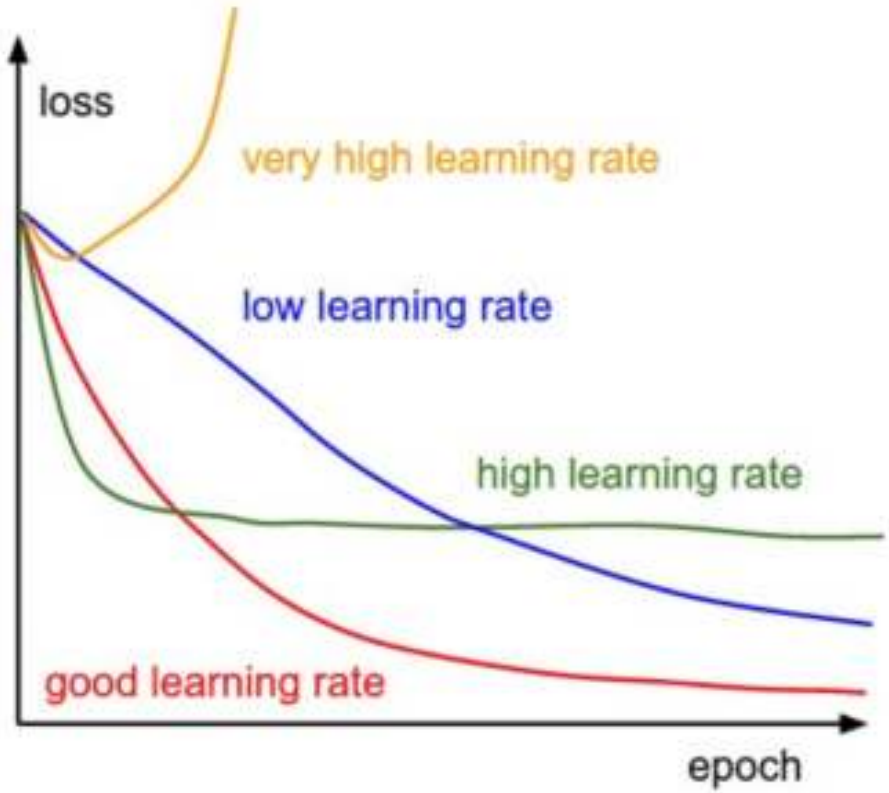
SGD: learning rate

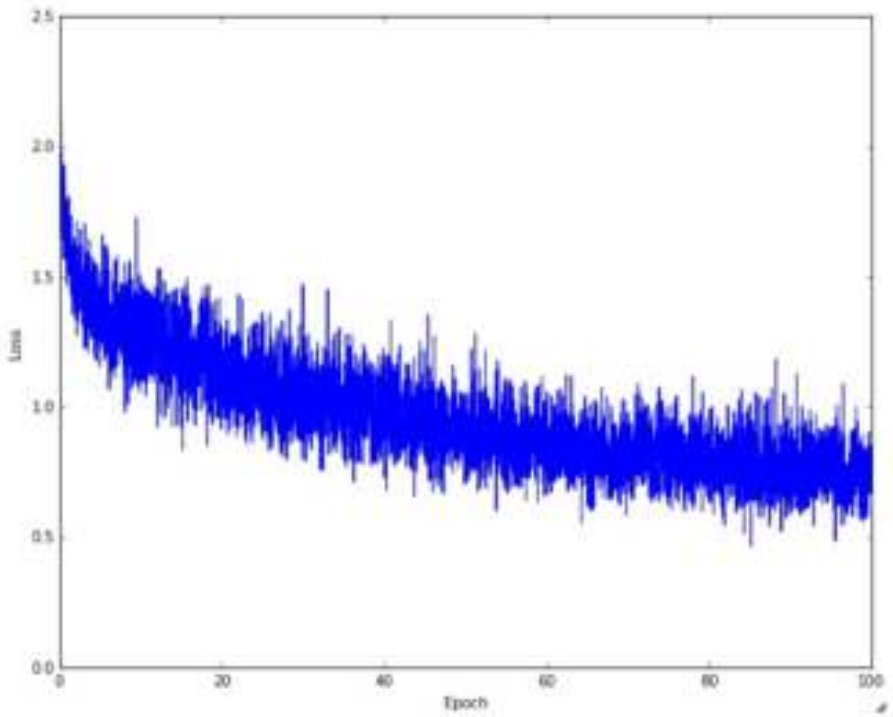
$$\theta = \theta - \varepsilon \left[\frac{1}{m} \sum_{i \in B} \nabla_\theta \ell(f(X_i, \theta), Y_i) + \lambda \nabla_\theta \Omega(\theta) \right]$$

Learning rate ε:

- Very important parameter for the algorithm.
- If it is too small: the convergence is very slow (can be trapped on a local minimum)
- If it is too large: oscillation around an optimum without stabilizing and converging.
- Adapt the learning rate during the training : take it large at the beginning and then reduce its value during the successive iterations.
- No general rule, need to control the evolution of the loss function.

SGD: learning rate





Left: A cartoon depicting the effects of different learning rates. With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape. **Right:** An example of a typical loss function over time, while training a small network on CIFAR-10 dataset. This loss function looks reasonable (it might indicate a slightly too small learning rate based on its speed of decay, but it's hard to say), and also indicates that the batch size might be a little too low (since the cost is a little too noisy).

[Source: Standford cs231n]

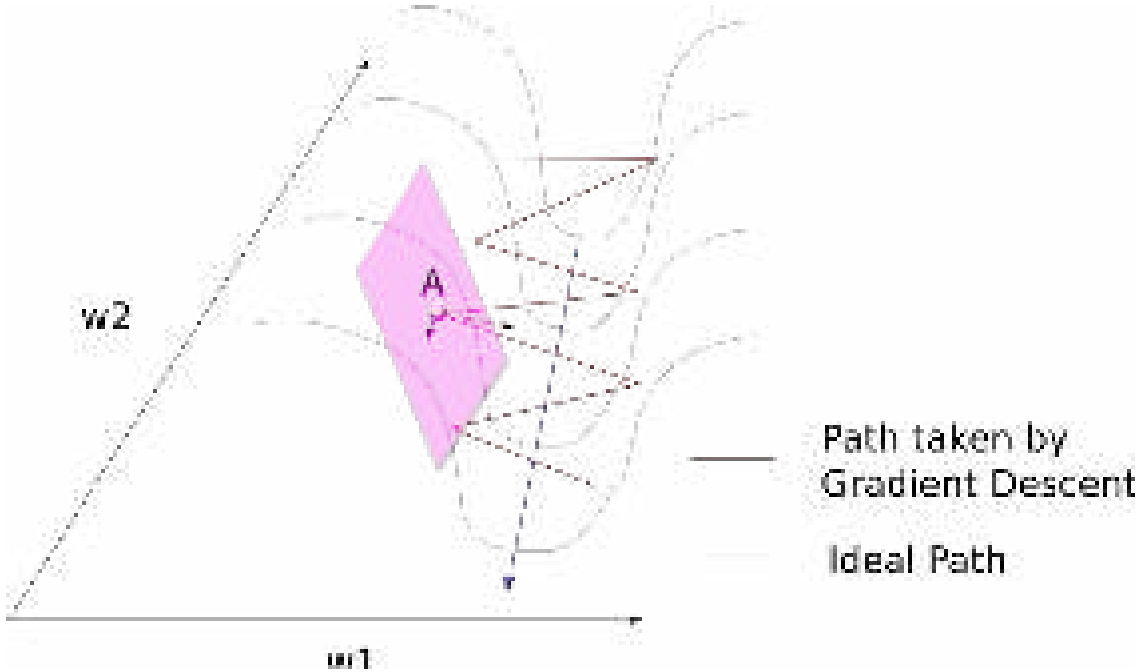
SGD: epochs

$$\theta = \theta - \varepsilon \left[\frac{1}{m} \sum_{i \in B} \nabla_\theta \ell(f(X_i, \theta), Y_i) + \lambda \nabla_\theta \Omega(\theta) \right]$$

- Only m training examples are randomly chosen to update the parameters at each iteration
- An epoch corresponds to a pass through the whole learning dataset.
- Max number of iterations is given in terms of epochs.
- Batch learning is used for computational reasons: the back-propagation algorithm needs to store all the intermediate values computed at the forward step.
- m is a parameter of the algorithm to calibrate (take $m = 2^j$).

Extensions and variants of SGD

- Non-convexity :
  - Neural networks have multiple (and a lot of) local minima (because of the non identifiability and because of RELU)
  - Challenging problem, a lot of research on this topic.
- Gradient descent is bouncing along the ridges of the ravine, and moving a lot slower towards the minima :



[blog.paperspace.com]

Extensions and variants of SGD

- SGD :

$$\begin{aligned}\eta^{(r)} &= \varepsilon\left[\frac{1}{m}\sum_{i\in B}\nabla_\theta \ell(f(X_i,\theta^{(r-1)}),Y_i)+\lambda\nabla_\theta\Omega(\theta)\right]\\ \theta^{(r)} &= \theta^{(r-1)}-\eta^{(r)}\end{aligned}$$

- Momentum : determines the next update as a linear combination of the gradient and the previous update

$$\begin{aligned}\eta^{(r)} &= \varepsilon\left[\frac{1}{m}\sum_{i\in B}\nabla_\theta \ell(f(X_i,\theta^{(r-1)}),Y_i)+\lambda\nabla_\theta\Omega(\theta)\right]+\gamma\eta^{(r-1)}\\ \theta^{(r)} &= \theta^{(r-1)}-\eta^{(r)}\end{aligned}$$

- AdaGrad (for adaptive gradient algorithm): with per-parameter learning rate, using a cumulative sums of squared past values of the gradients.
- RMSProp (for Root Mean Square Propagation) modifies AdaGrad by replacing cumulative sums by exponentially weighted averages.
- Adam : combines the ideas of Momentum optimization and RMSProp.

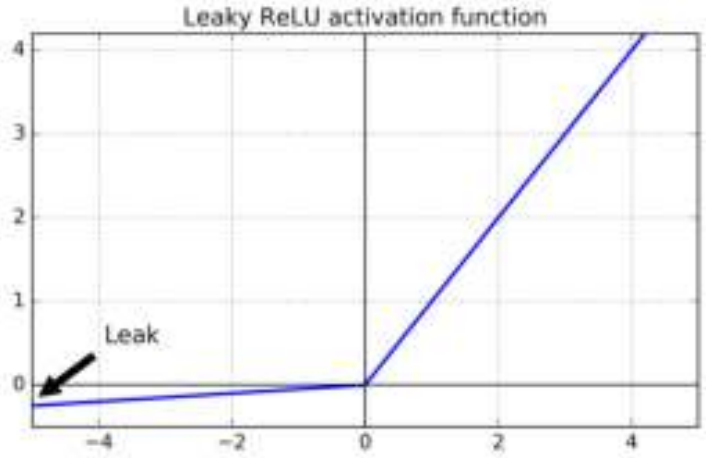
Outline

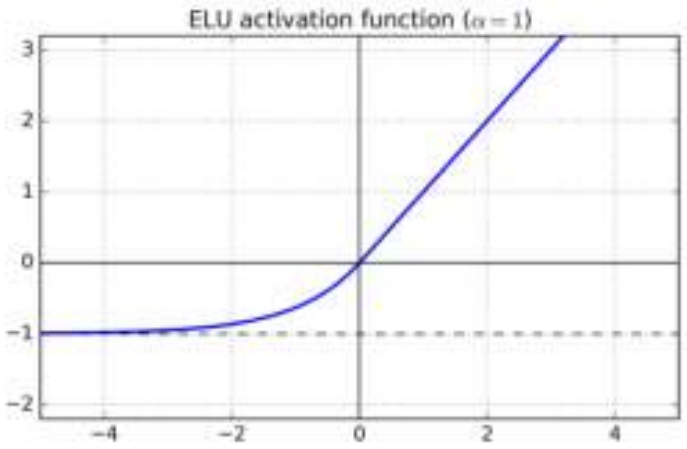
Gradient instability in DeepNN

- In many situations, gradients often get smaller and smaller as the algorithm progresses down to the lower layers : vanishing gradients problem.
- In other cases (Recurrent NN) : the gradients can grow bigger and bigger which makes the algorithm diverges: exploding gradients problem.
- Deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.
- Training may never converges to a good solution.
- Glorot-Bengio (2010): the combination of logistic sigmoid activation function + random weight initialization using standard normal distribution are mostly responsible for this gradient instability.

Gradient instability : use RELU and its variants

- The Logistic activation function saturates at 0 or 1 for large inputs, with a derivative extremely close to 0 : no gradient to propagate back (and back-propagation fails).
- ReLU activation function does not saturate for positive values.
- During training, some neurons effectively die (dying ReLUs) and such neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.
- Alternative : $LeakyReLU_\alpha(z) = \max(\alpha z, z)$ and exponential linear unit (ELU) activations:





[Hands on ML]

Gradient instability : use Glorot (Xavier) Initialization

- The input data have to be normalized to have approximately the same range.
- The biases can be initialized to 0.
- The weights :
  - cannot be initialized to 0 because gradients are likely to be 0
  - Recipe (Glorot): Initialize the weights at random : the values $W_{i,j}^{(k)}$ are i.i.d. Uniform on $[-c, c]$ with possibly $c = \frac{\sqrt{6}}{N_k + N_{k-1}}$ where N_k is the size of the hidden layer k.
- This initialization tends to control the variance of activations.

Gradient instability : others improvements

- Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

  Batch Normalization : center and normalize the inputs of a layers by estimating mean and variance using the batch sample.

- Exploding gradients can occur when the gradient becomes too large and error gradients accumulate, resulting in an unstable network.

  Gradient clipping : clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for recurrent neural networks).

Outline

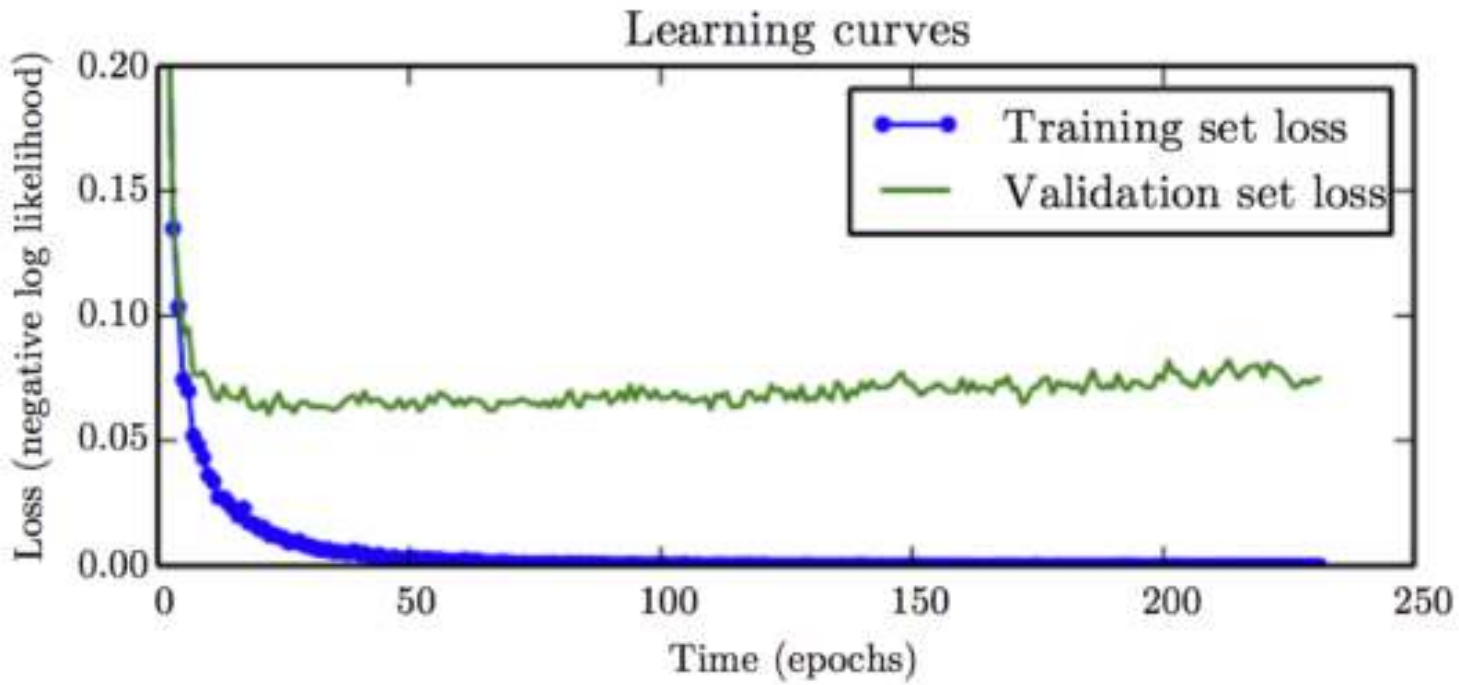
Number of Hidden Layers

- For many problems, one or two hidden layers will be sufficient to reach high accuracy.
- For more complex problems : gradually increase the number of hidden layers, until you start overfitting the training set.
- Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and also need a huge amount of training data.
- For such complex problem, we rarely have to train such networks from scratch: reuse parts of a pre-trained state-of-the-art network that performs a similar task. Training will be a lot faster and require much less data.
- If no pre-trained model available: find an efficient architecture in the literature

Number of Neurons

- Number of neurons in the input is determined by input size vector.
- Number of neurons in the output is determined by the learning task.
- Number of neurons in the hidden layers : size them to form a tunnel, with fewer and fewer neurons at each layer : “low-level features can coalesce into far fewer high-level features".

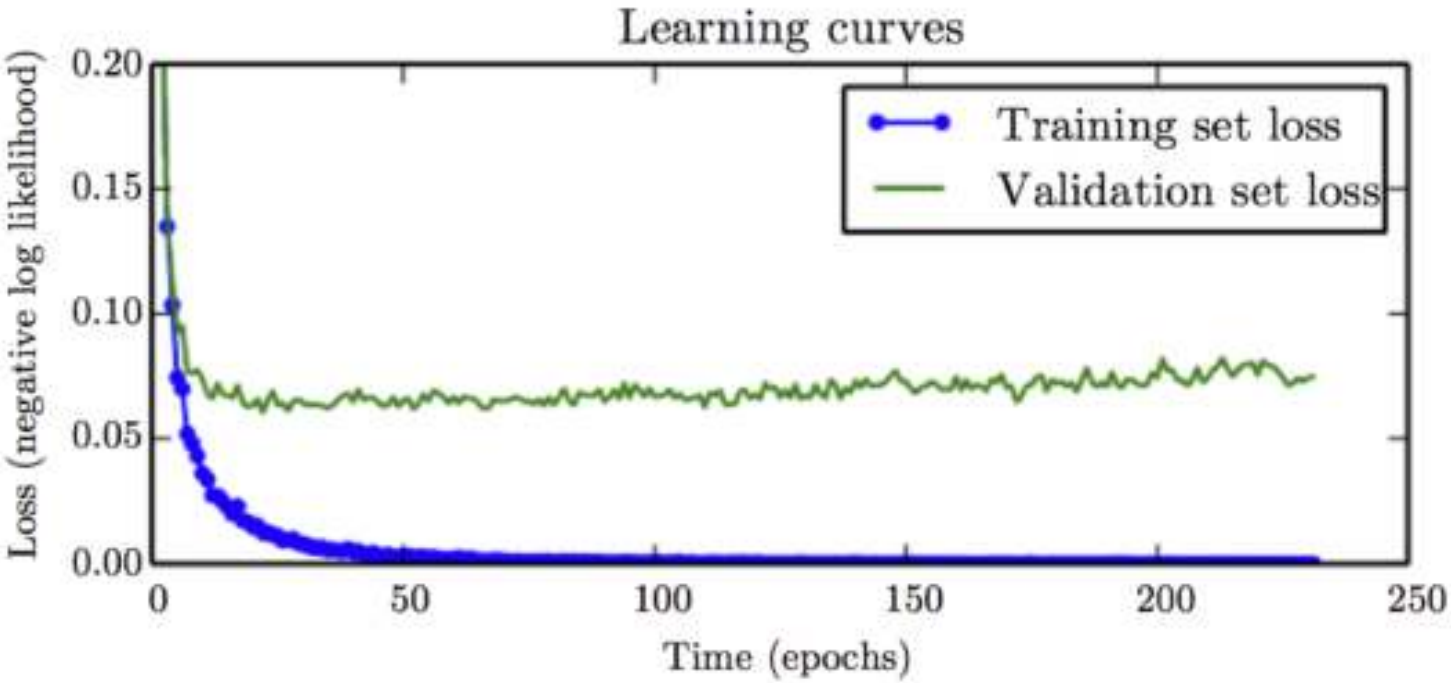
Regularization: early stopping



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

- When fitting parameters with gradient descent, up until a certain number of iterations, new iterations improve the model.
- After that point, however, the model's ability to generalize can weaken as it begins to overfit the training data.

Regularization: early stopping

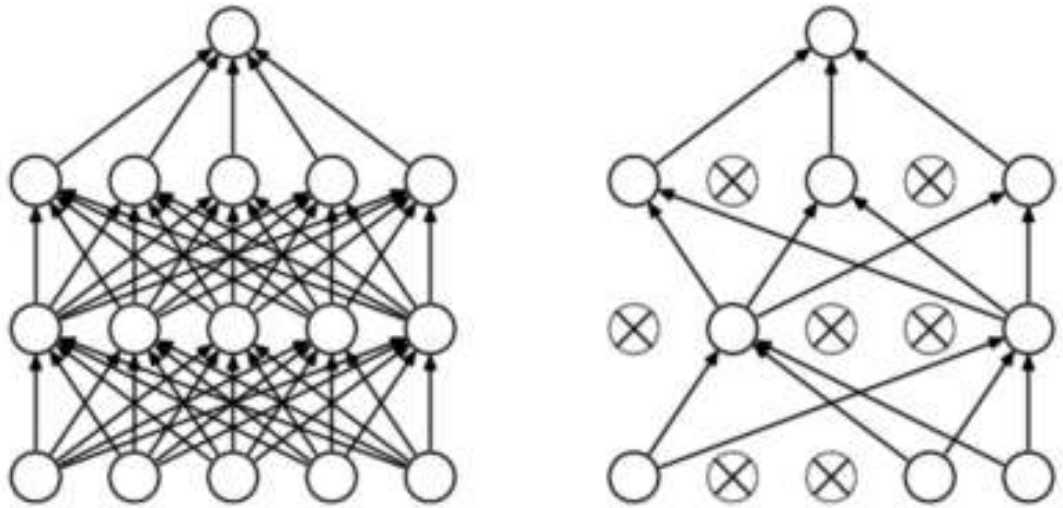


[from *Deep Learning*, Goodfellow, Bengio and Courville]

- Easy to implement:
  - Splitting : (Train - Validation) - Test.
  - If validation has not improved for some time, stop and return θ.
- For least-squares regression and gradient descent, early stopping can be seen to be equivalent to ridge penalization on the weights.

Regularization: dropout

- Train an ensemble of sub-networks formed by removing non-output units from the base network.
- At each iteration of the gradient descent, with probability p, and independently of the others, each unit of the network is set to 0.



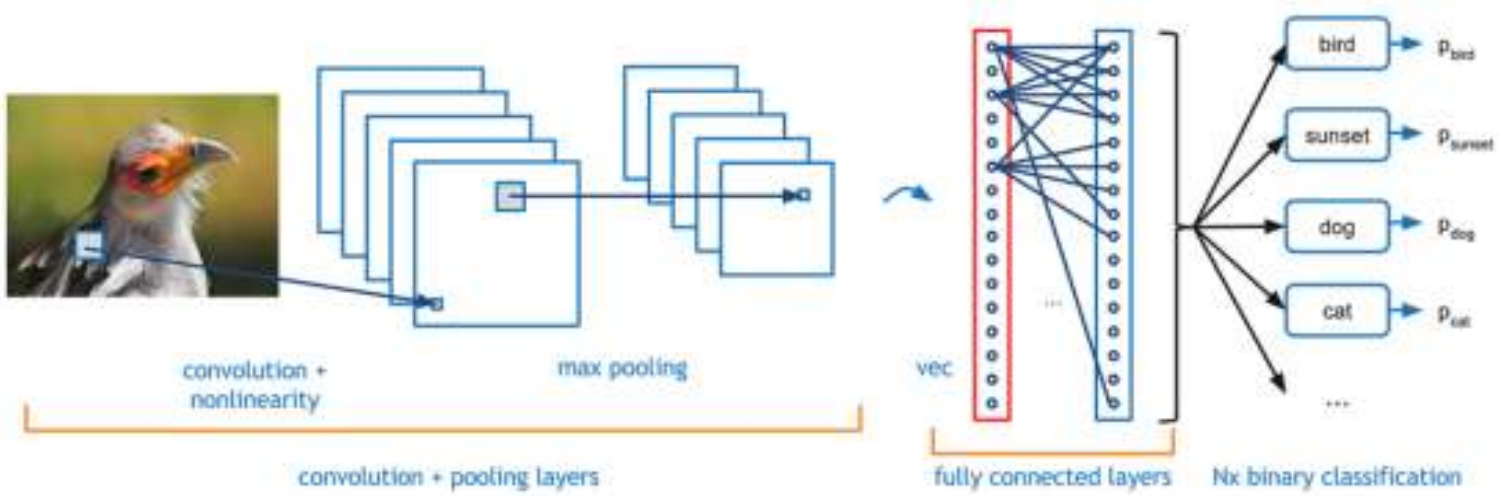
(a) Standard Neural Net (b) After applying dropout.

[Source: Srivastava]

- The probability p is a hyper-parameter (popular choice is 0.5 for units in the hidden layers and 0.2 for the entry layer).
- Low computational cost.
- Improves the generalization properties of deep neural networks

Outline

Convolution Networks (CNNs)



[Source: A. Deshpande]

- CNNs, are a specialized kind of neural network for processing time-series data and image data.
- CNNs use **convolution** in place of general matrix multiplication in at least one of their layers.
- Specific tools for CNNs :
  - Convolution,
  - Pooling.

Outline

What computer see

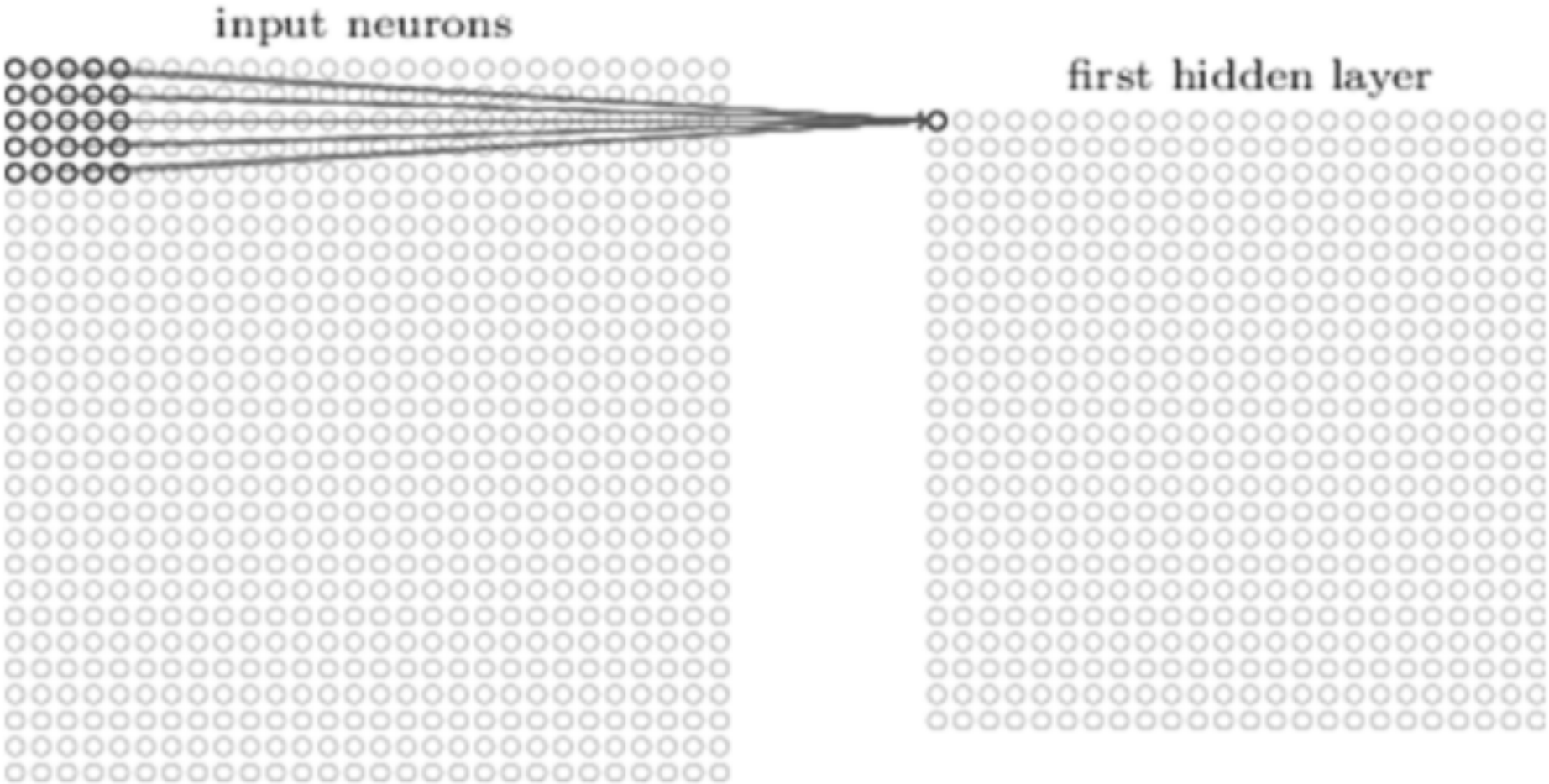
What We See

What Computers See

[Source: A. Deshpande]

Objective: identify low level features such as edges and curves, and then building up to more abstract concepts through a series of convolutional layers.

Convolutional Layer



[Source: Neural Networks and Deep Learning by Michael Nielsen]

Convolutional Layer

Convolution function with 1 channel:

$$Z(i,j) = (I \star K)(i,j) = \sum_{|u| \leq n_1} \sum_{|v| \leq n_2} I(i+u, j+v) K(u,v)$$

where

- I is the input and K is the kernel or filter
- Z is a feature map.
- The values of the kernel correspond to neuron's weights : $K(u,v) = w_{u,v}$.
- Filter **size** is defined by (n_1, n_2).
- The values $K(u,v)$ of the kernel are learned by the CNN.

Filters

- Filters can be thought of as feature identifiers or feature maps.
- E.g straight edges, simple colors, and curves.

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Visualization of a curve detector filter

[Source: Neural Networks and Deep Learning by Michael Nielsen]

Filters

Original image

Visualization of the filter on the image

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

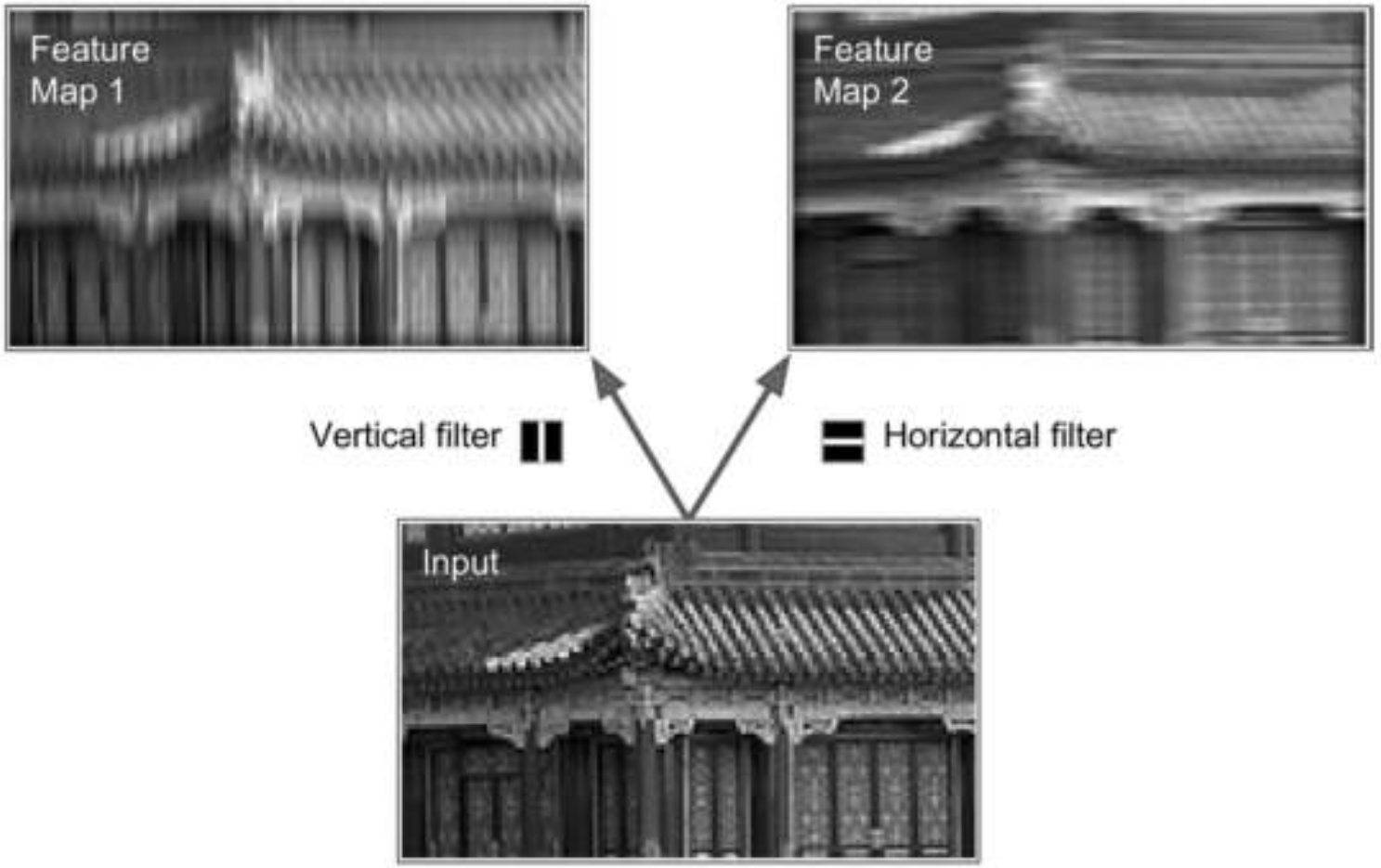
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

[Source: Neural Networks and Deep Learning by Michael Nielsen]

Filters



[Source: Hands on ML]

Parameter sharing

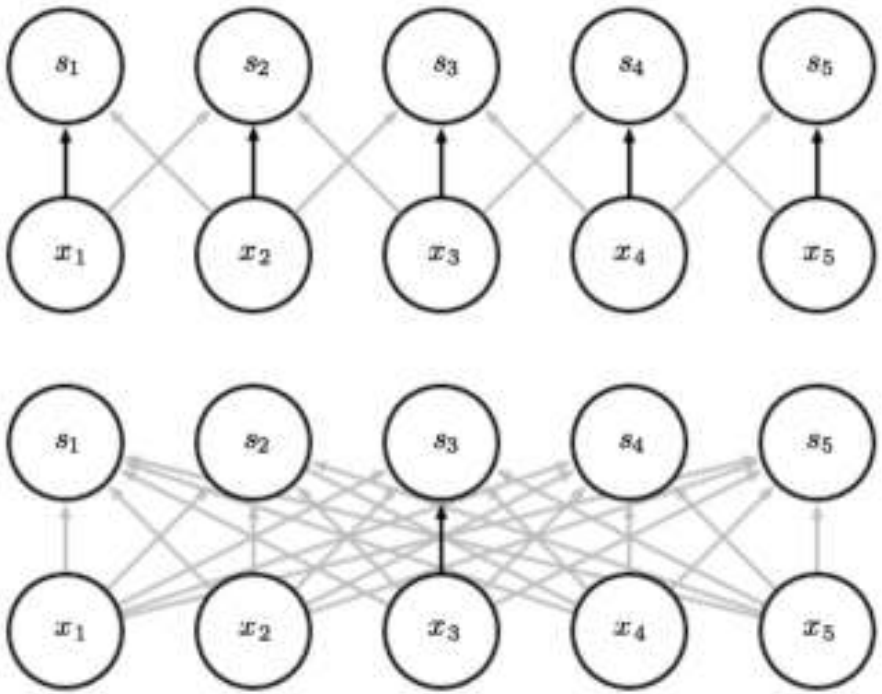
- **Parameter sharing** : Rather than learning a separate set of weights for every location, we learn only one set :

$$\begin{aligned} S(i,j) &= (I \star K)(i,j) = \sum_{l,j} I(i+k, j+l) K(k,l) \\ &= \sum_{l \leq n_1, j \leq n_2} I(i+k, j+l) w_{k,l} \end{aligned}$$

- This means that we learn the same kernel (or the same set of kernels) for every location: $S(i,j) = (I \star K)(i,j)$.

Parameter sharing

- **Much more efficient** than dense matrix multiplication in terms of the memory requirements $O(N_{inputs})$ and statistical efficiency.

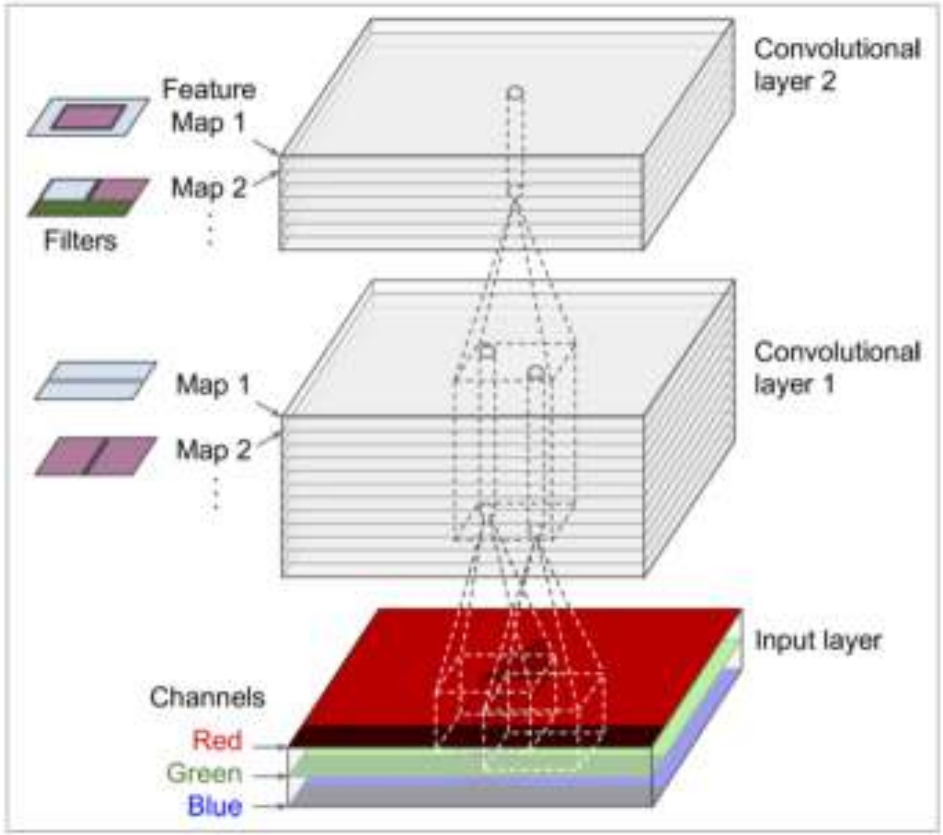


[from Deep Learning, Goodfellow, Bengio and Courville]

Outline

Stacking sublayers

- Convolutional layers : not one but k filters (take $k = 2^p$) that create k feature maps.
- Neuron's receptive field extends across all the previous layers's feature maps : **multichannel convolution**.



[Source: Hands on ML]

Multichannel convolution

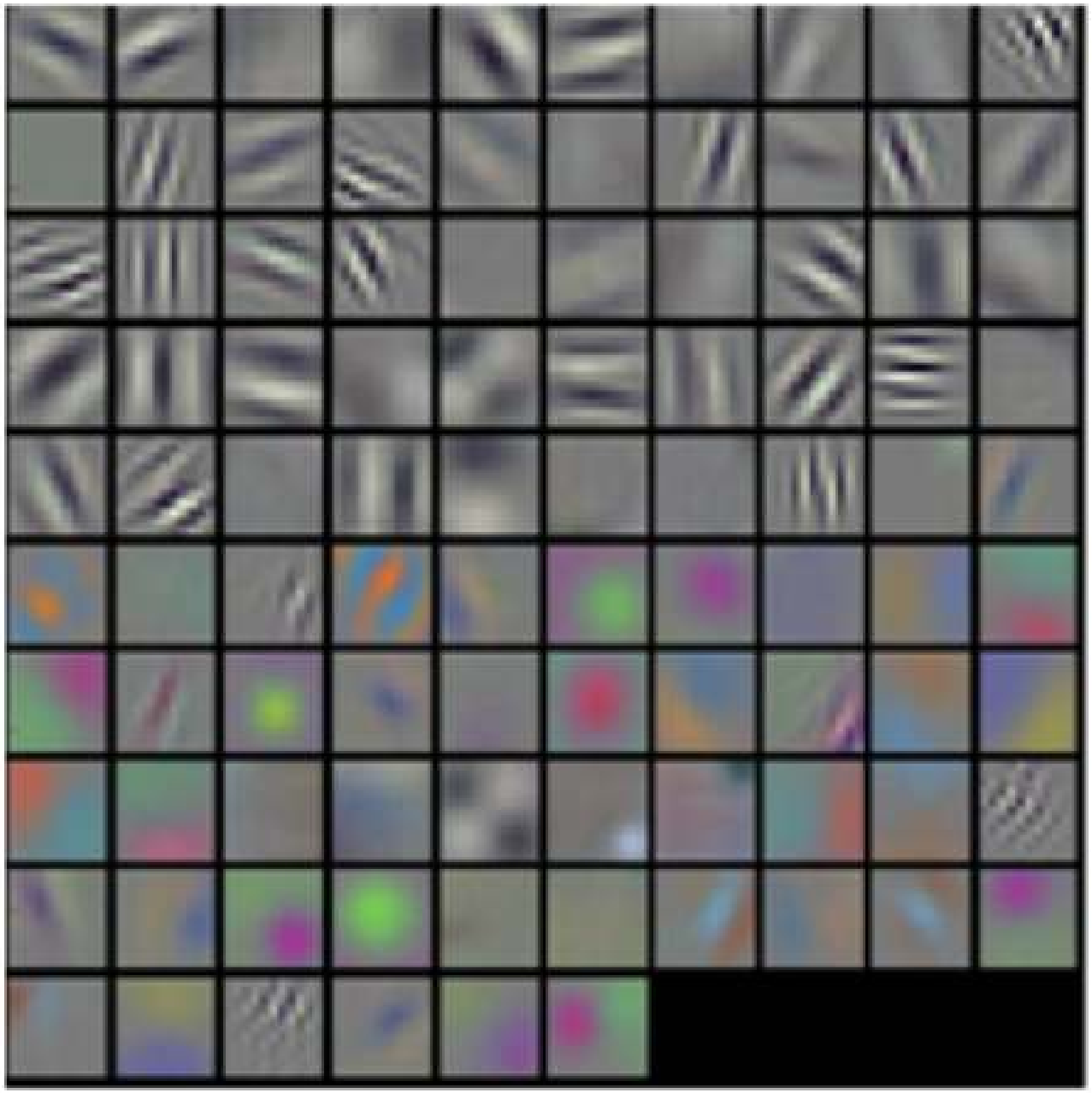
- Convolutional networks usually use multichannel convolution:

$$Z(i,j,l) = (V \star K)(l,i,j) = \sum_{u,v,w} V(i+u, j+v, w) K(u,v,l,w)$$

  where
  - V and Z have the same dimensions (multichannel).
  - Z is a feature map.
  - $K(u,v,l,w)$ gives the connection strength between a unit in channel l of the output and a unit in channel w of the input, with an offset of u rows and v columns between the output unit and the input unit.

- Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch.

First (trained) convolution layer



Convolution in CNNS

Convolution leverages a least two important ideas:

- Sparse interactions ;
- Parameter sharing ;

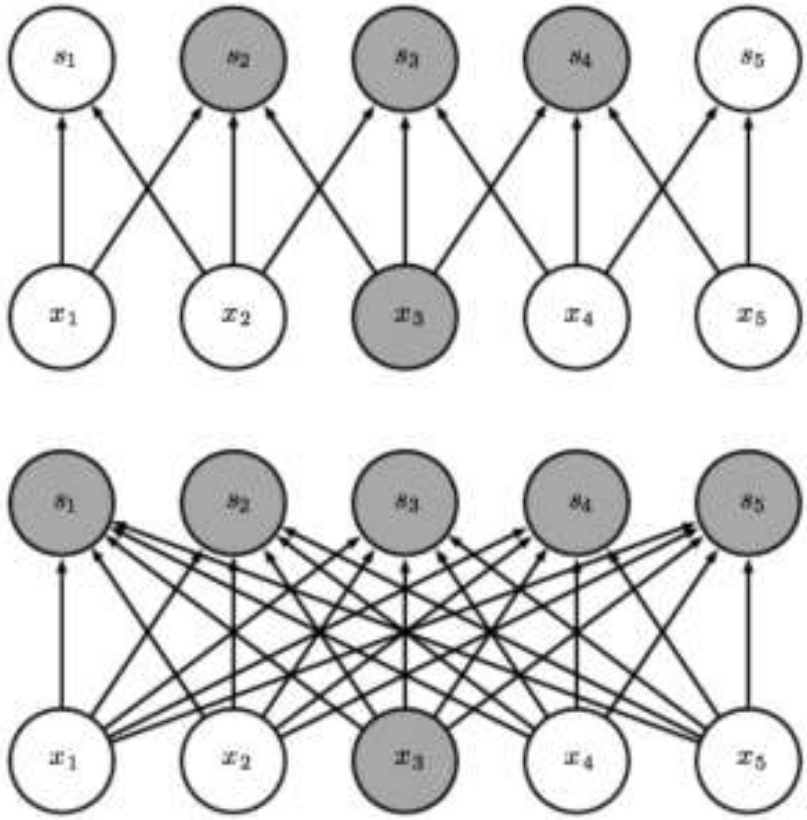
Sparse interactions

- For traditional NN, every output unit interacts with every input unit (full connected layers)
- When processing an image of thousands or millions of pixels, meaningful features such as edges occupy only hundreds of pixels.
- Need to focus and store fewer parameters.

Sparse interactions

- CNNs : **sparse interactions** by making the kernel smaller than the input.

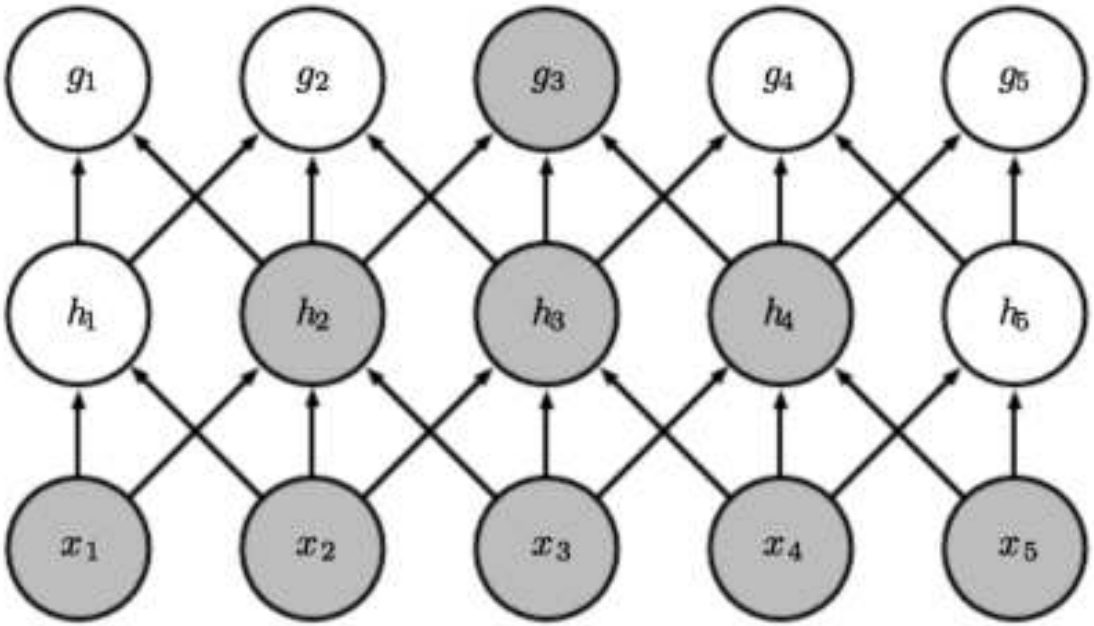
$$S(i,j) = (I \star K)(i,j) = \sum_{l \leq n_1, j \leq n_2} I(i+k, j+l) K(k,l)$$



[from Deep Learning, Goodfellow, Bengio and Courville]

Receptive field of the units in the deeper layer

- Even though direct connections in a convolutional net are very sparse, units in the deeper layers can be indirectly connected to all or most of the input image:

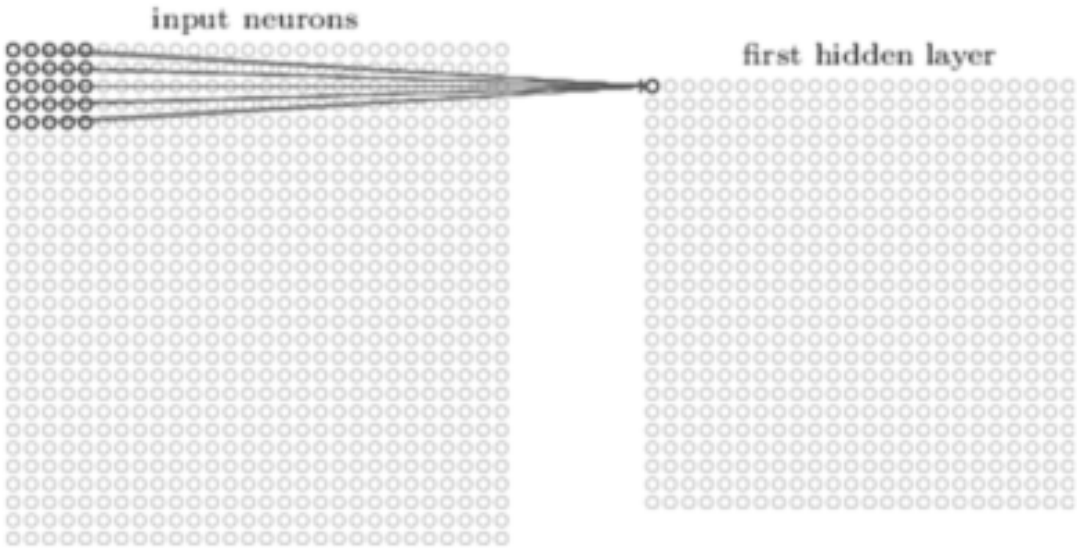


[from Deep Learning, Goodfellow, Bengio and Courville]

Outline

Padding

Valid Padding: output volume is reduced.



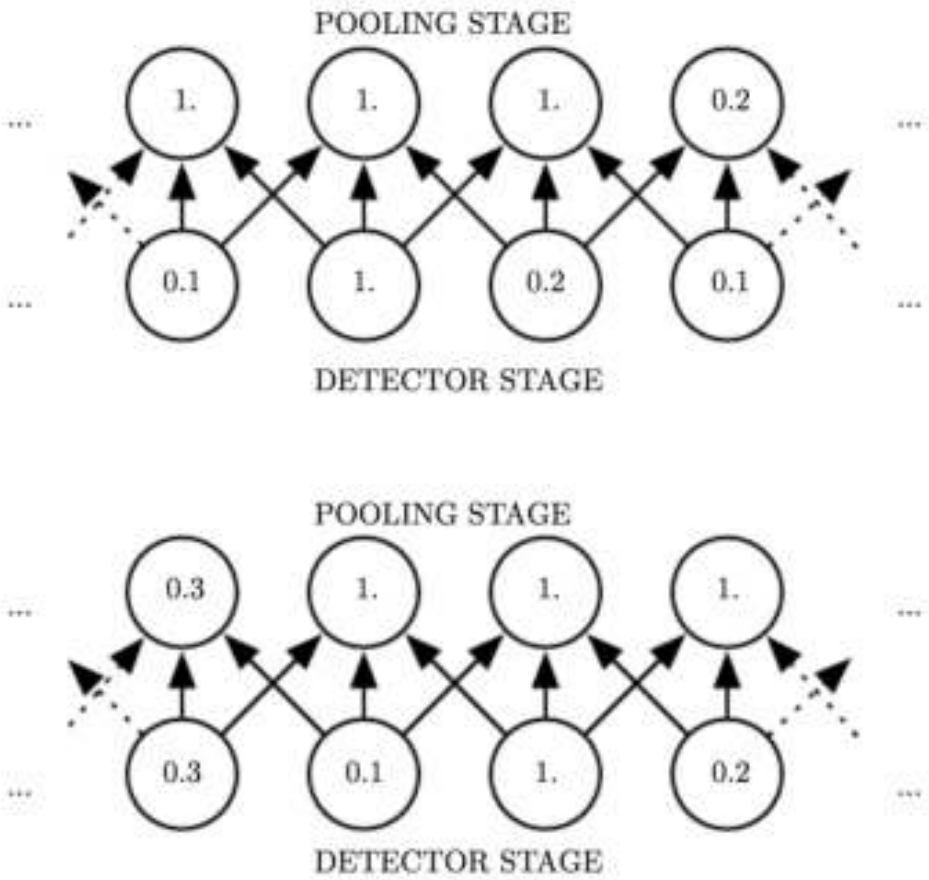
Zero padding: output volume is unchanged.

Pooling

- A **pooling function** replaces the output of the net at a certain location with a summary statistic of the nearby outputs.
- Max pooling: reports the maximum output within a rectangular neighborhood.
- Other popular pooling functions: average of a rectangular neighborhood, L^2 norm, weighted average based on the distance from the central pixel...

Pooling introduces invariance

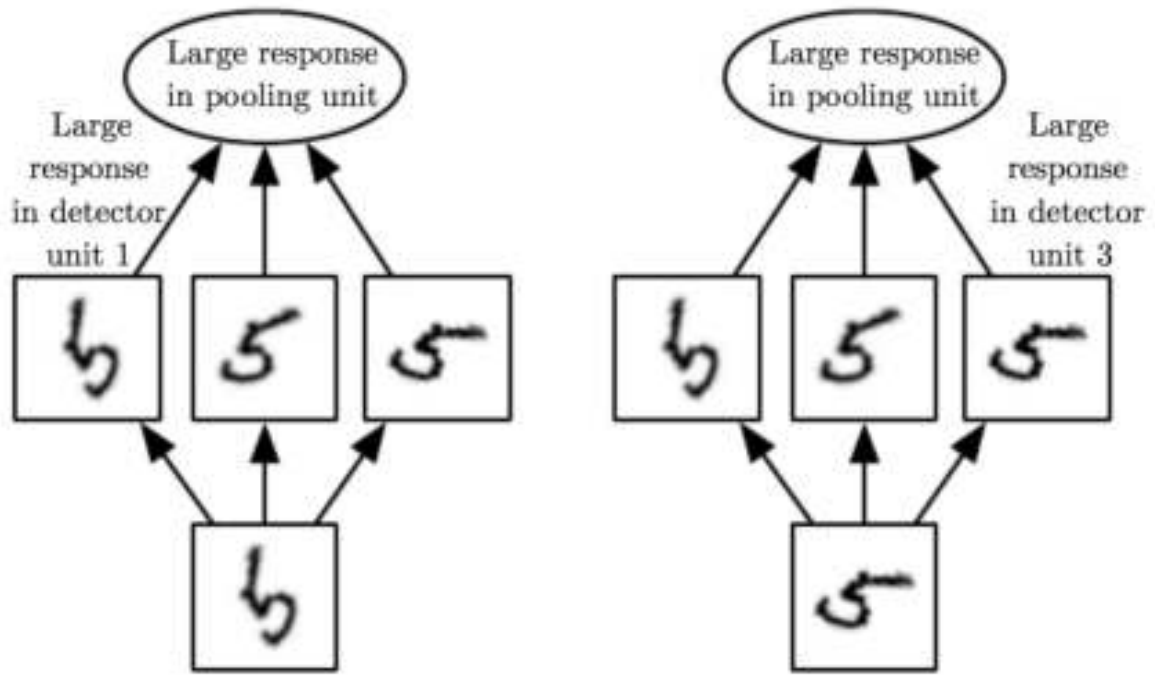
- A stride of one pixel is applied in the input Output of max-pooling is almost invariant:



[from Deep Learning, Goodfellow, Bengio and Courville]

Pooling introduces invariance

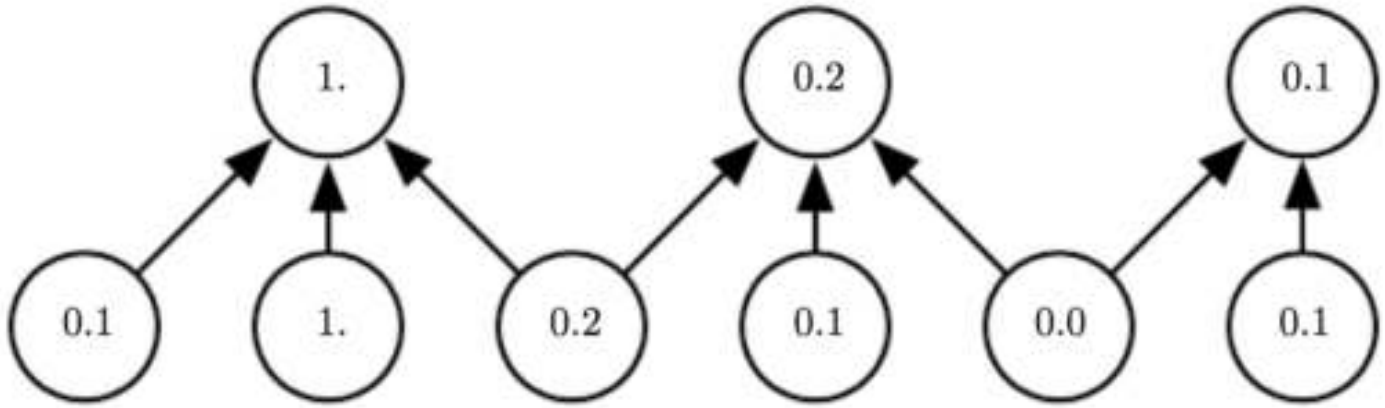
- By pooling over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant
- A set of three learned flters and a max pooling unit can learn to become invariant to rotation:



[from Deep Learning, Goodfellow, Bengio and Courville]

Pooling allows to handle inputs with different sizes

- Pictures can have different sizes, but the output classification layer must be of fixed size.
- Pooling also allows to handle inputs with different sizes.

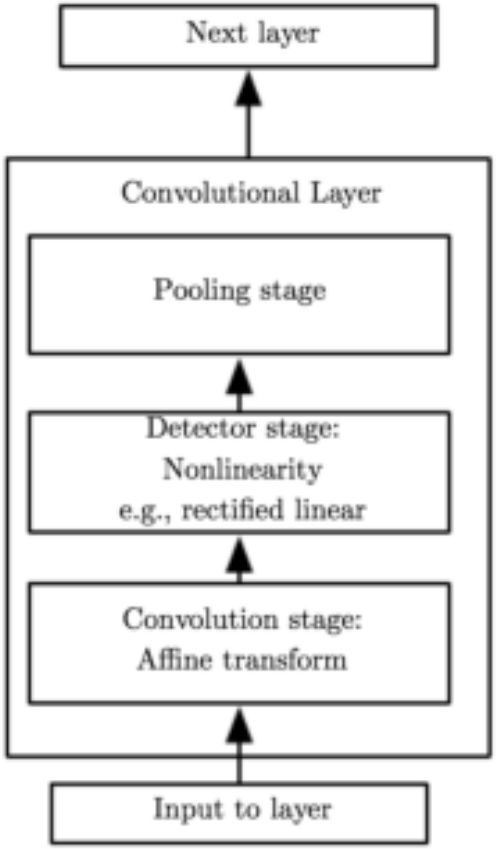


[from Deep Learning, Goodfellow, Bengio and Courville]

CNN architectures

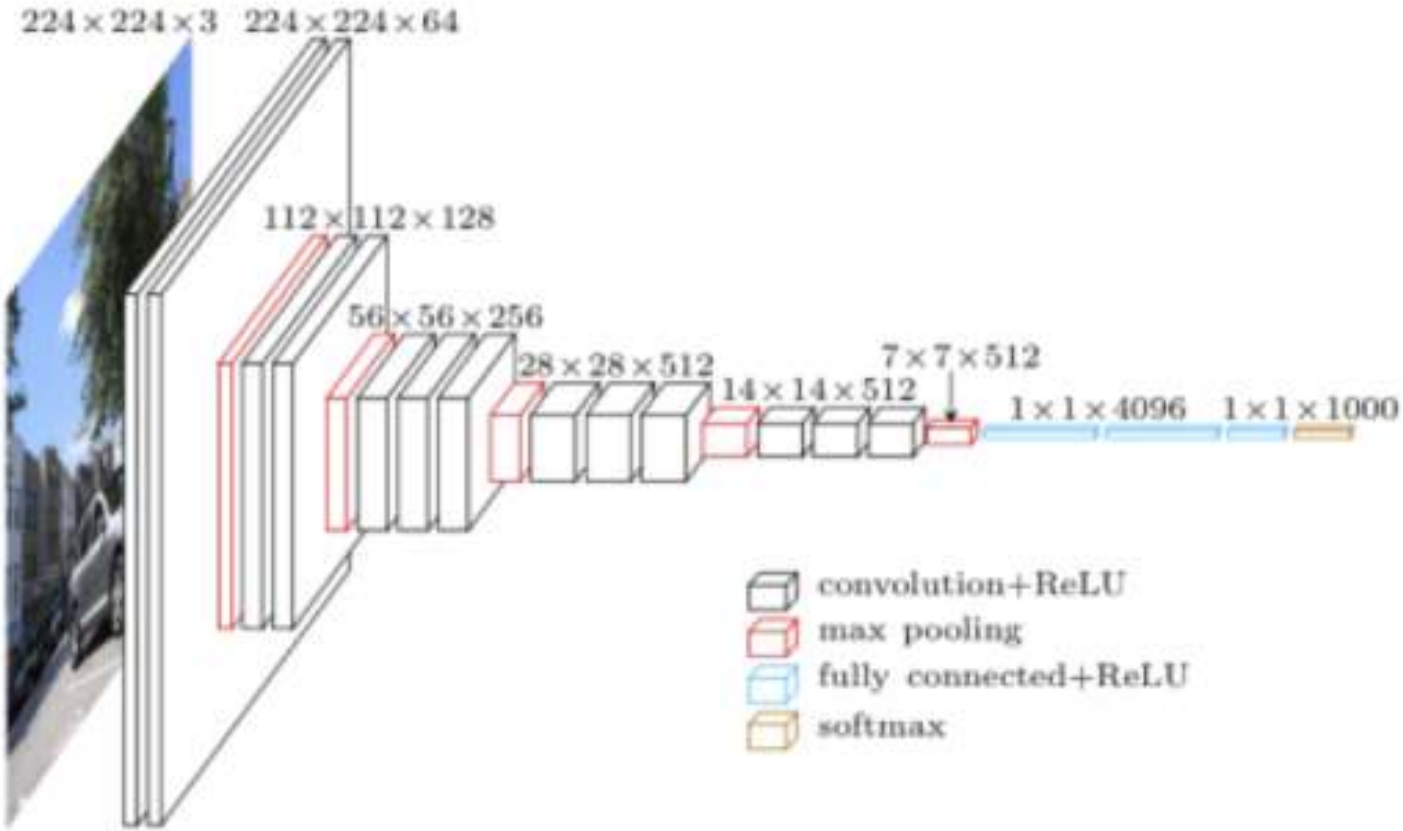
A typical layer of a convolutional network consists of three stages:

- **Convolution layer**: the layer performs several convolutions in parallel to produce a set of linear activations.
- **Detector layer**: each linear activation is run through a nonlinear activation function.
- **Pooling layer**: pooling function to modify the output of the layer further.



[from Deep Learning, Goodfellow, Bengio and Courville]

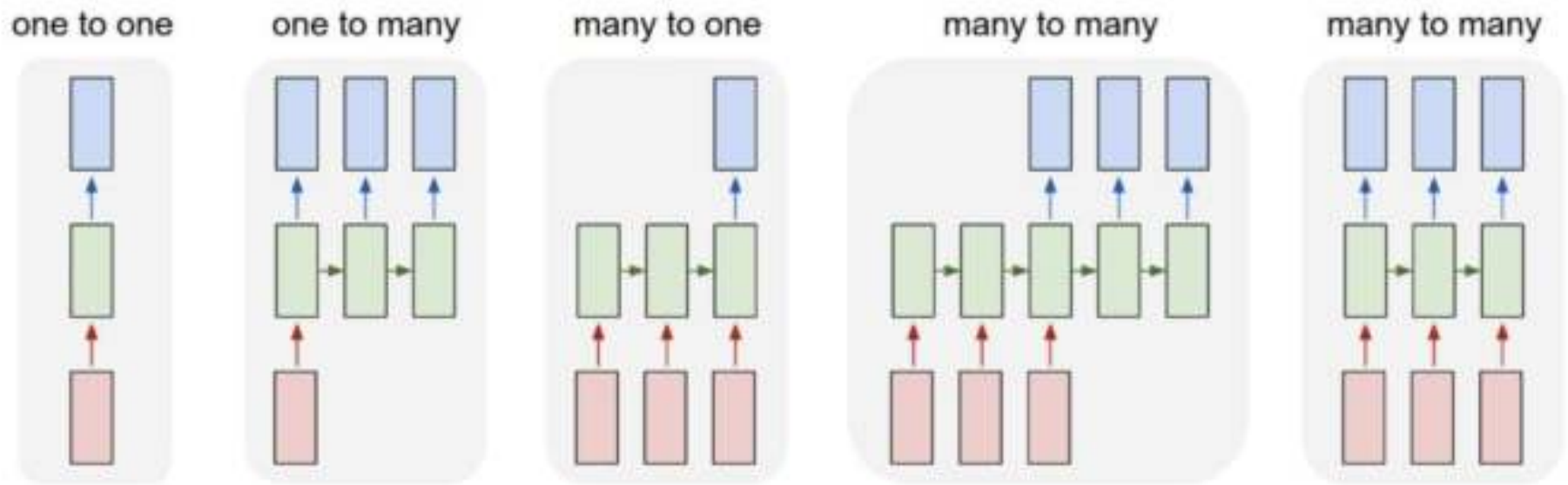
Deep CNN for large-scale image recognition



[from Simonyan, K. and Zisserman 2014.]

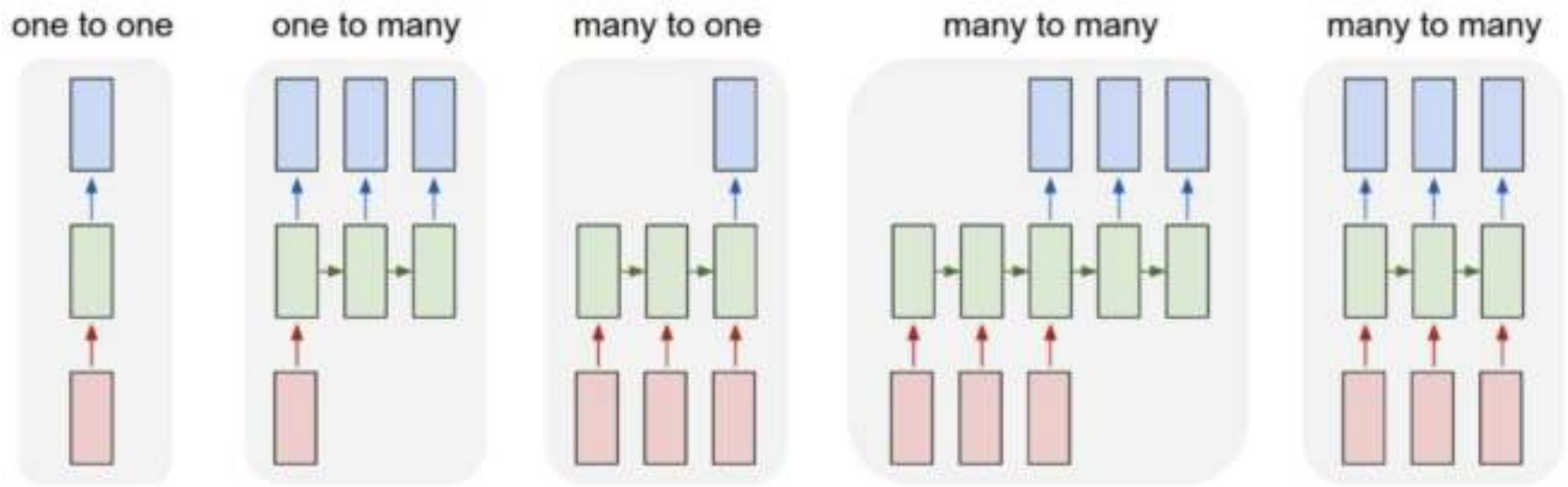
Outline

Various prediction problems



- **One to one**: Vanilla Neural Networks
- **One to many**: Image Captioning: image/sequence of words
- **Many to one**: Sentiment classification: sequence of words/sentiment
- **Many to many**: Translation: sequence of words/sequence of words
- **Many to many**: Video classification on frame level: sequence of image/sequence of label

Various prediction problems



Limitations of (feed-forward) neural networks:

- They can't deal with sequential or "temporal" data
- They lack memory
- They have a fixed architecture: fixed input size and fixed output size

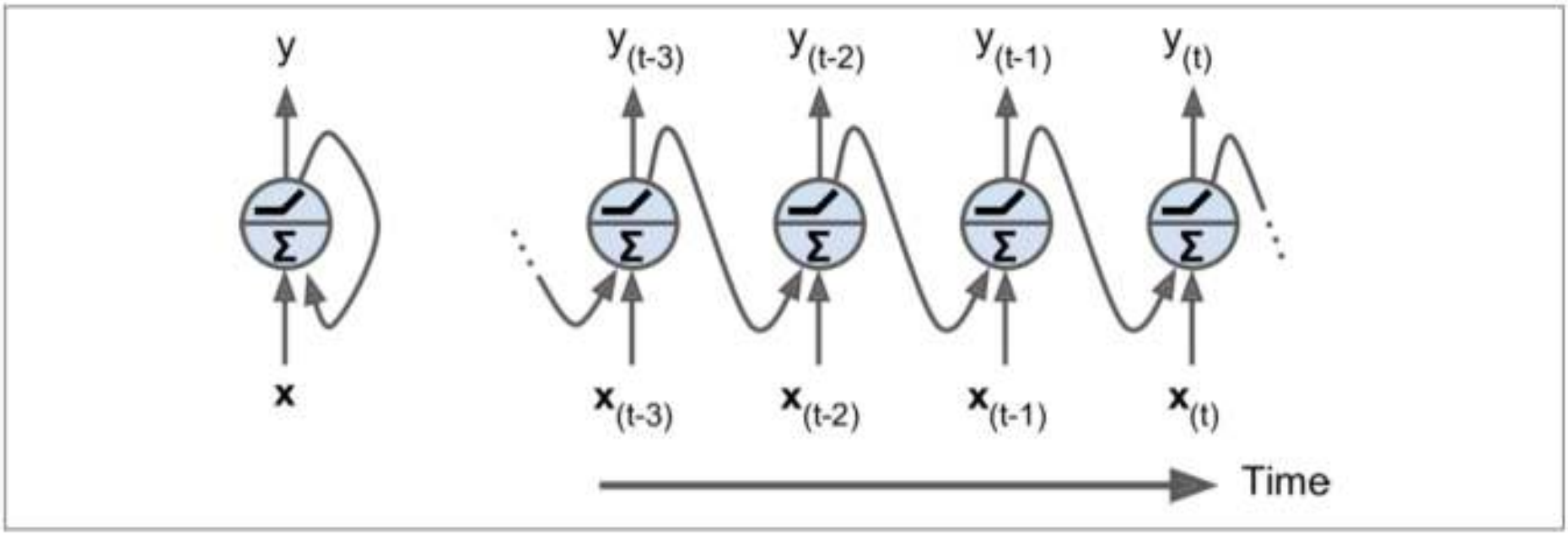
Recurrent neural networks (RNNs) are introduced to overcome successfully these limitations

Parameter sharing for sequential data

- Recurrent Neural Networks : family of neural networks for processing sequential data (time series) : $x(t), t = 1 \ldots \tau$.
- Applications :
  - predict stock prices
  - predict trajectories
  - natural language processing (NLP)
- RNNs use **Parameter sharing**
  - 1D-CNNs is also based on parameter sharing: the output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input.
  - Parameter sharing is different with RNNs: each member of the output is a function of the previous members of the output.

Recurrent neuron

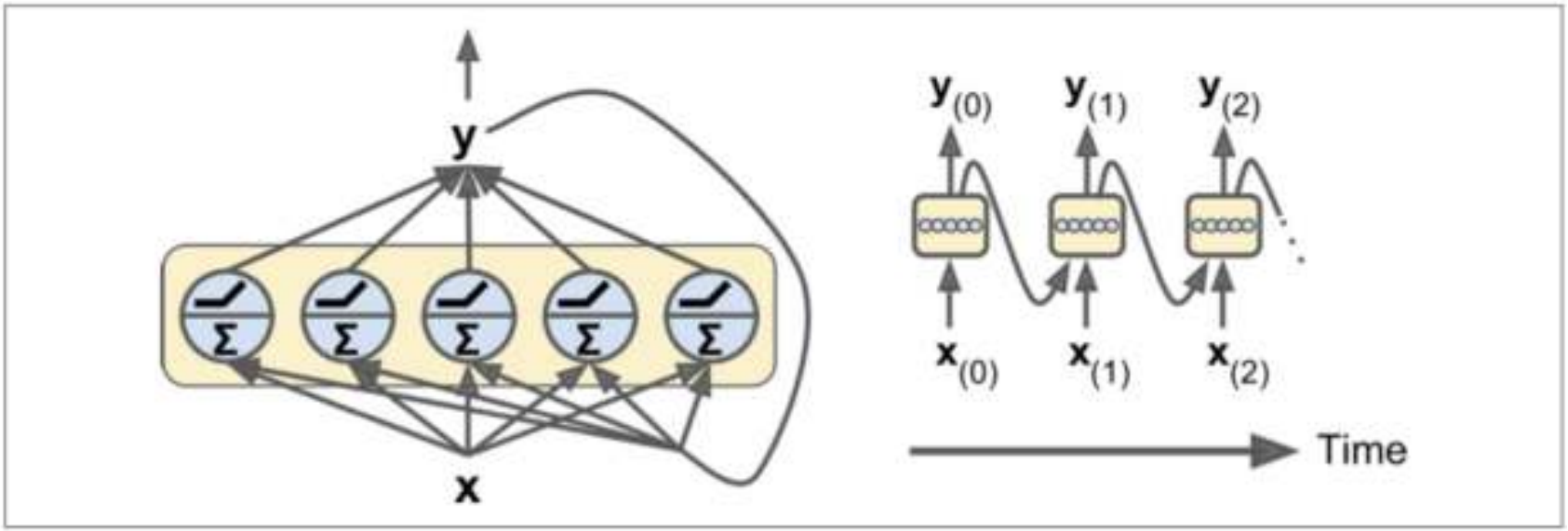
- A recurrent neuron has connections pointing backward.
- At each time step t, a recurrent neuron receives the inputs scalar $x(t)$ as well as its own scalar output from the previous time step $y(t-1)$.
- In the unfolded computational graph each node is associated with one particular time instance.



[source : Hands on ML]

Recurrent layer

- At each time step t, every neuron in a recurrent layer receives both the input vector $x(t)$ and the output vector from the previous time step $y(t-1)$.



[source : Hands on ML]

Output of recurrent neurons

- For some activation function g, the scalar output at time t of one single recurrent neuron is given by

$$y(t) = g\left(x(t)w_x + y(t-1)w_y + b\right)$$

where w_x is the weight for the input $x(t)$ and w_y for the output of the previous time step $y(t-1)$.

- The matrix output at time t of a layer of recurrent neurons for a mini batch sample (of size m) is given by

$$Y(t) = g\left(\begin{bmatrix} X(t) & Y(t-1) \end{bmatrix} \begin{bmatrix} W_x \\ W_y \end{bmatrix} + B\right)$$

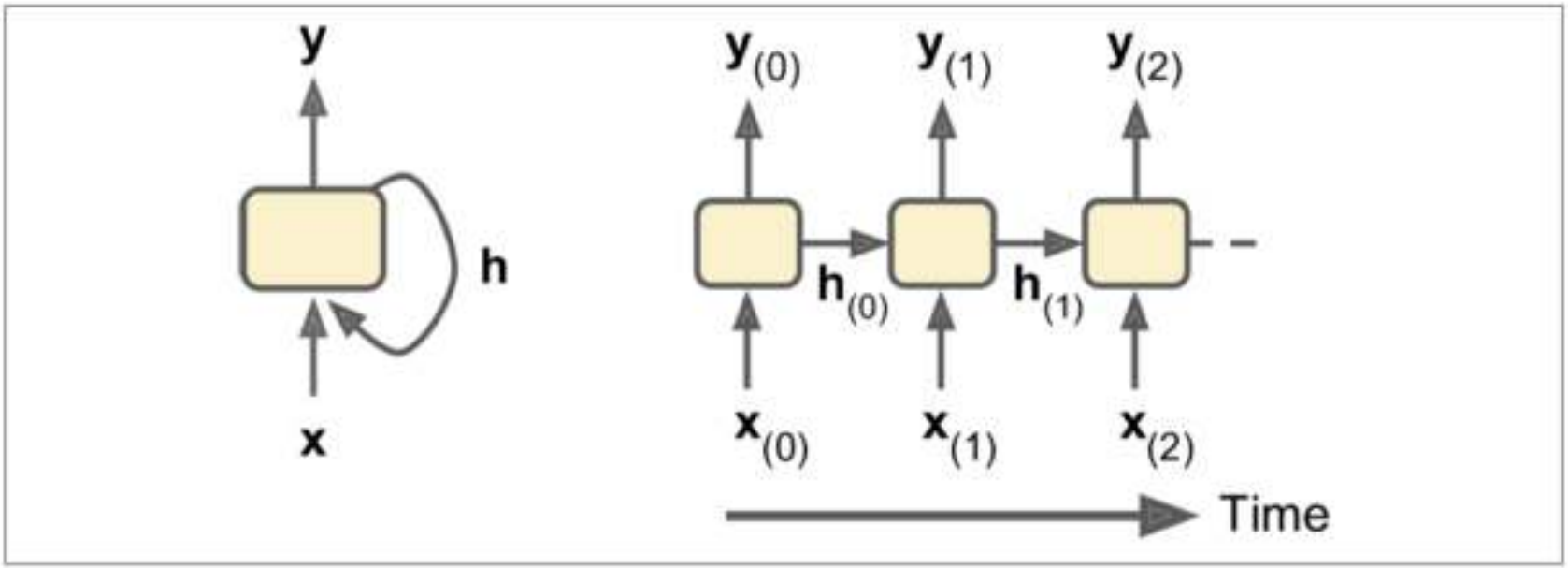
where $X(t)$ is a $n_{input} \times m$ matrix and $Y(t)$ is a $n_{neur} \times m$ matrix.

- $Y(t)$ is a function of all the inputs since time $t = 0$.

- This is parameter sharing over time !

Memory cell

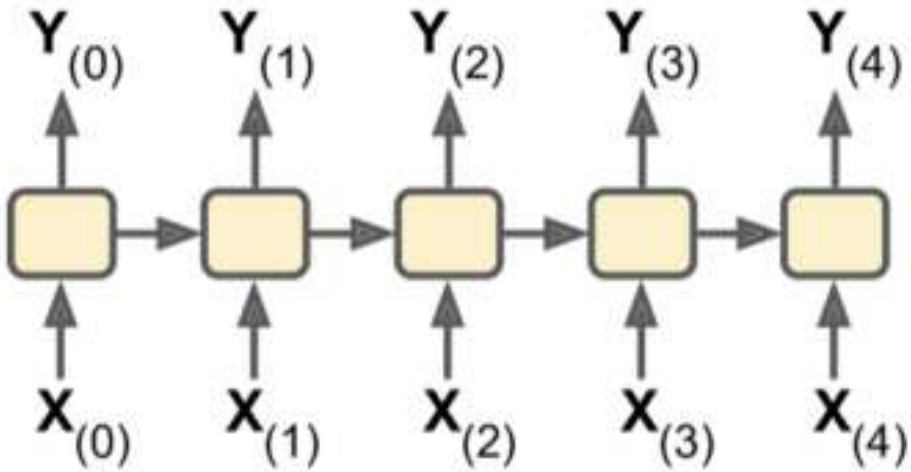
- A part of a neural network that preserves some state across time steps is called a memory cell (e.g. single recurrent neuron or a layer of recurrent neurons).



[source : Hands on ML]

Input and Output Sequences

- A sequence-to-sequence network takes a sequence of inputs and produces a sequence of outputs:

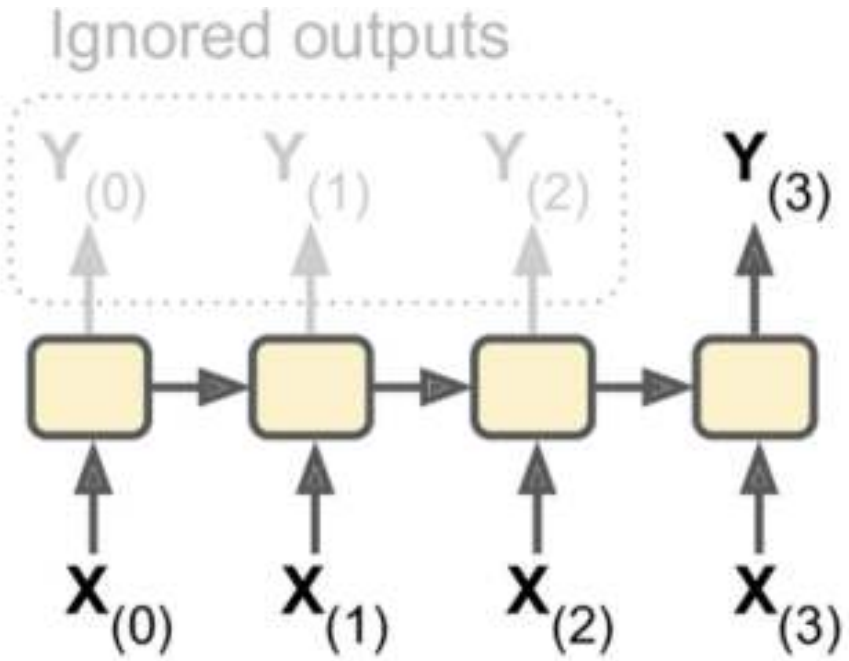


[source : Hands on ML]

- Application : time series regression.

Input and Output Sequences

- Sequence- to-vector network: feed the network a sequence of inputs, and ignore all outputs except for the last one:

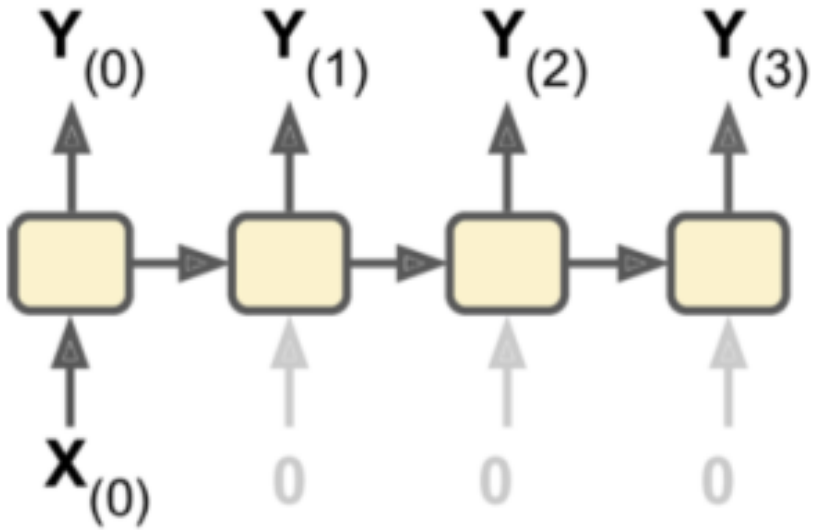


[source : Hands on ML]

- Application: text classification, sentiment analysis (TP).

Input and Output Sequences

- Vector-to-sequence network:

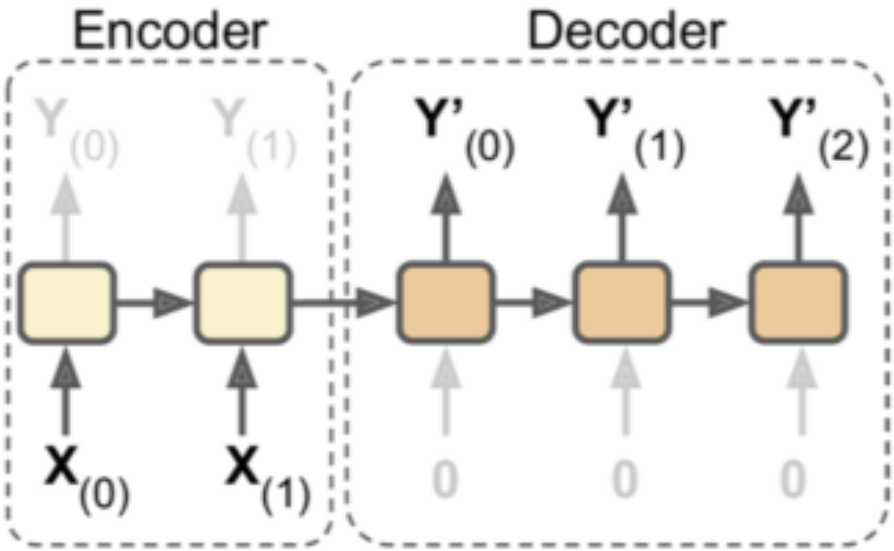


[source: Hands on ML]

- For example, the input could be an image, and the output could be a caption for that image.

Input and Output Sequences

- Encoder - Decoder network :

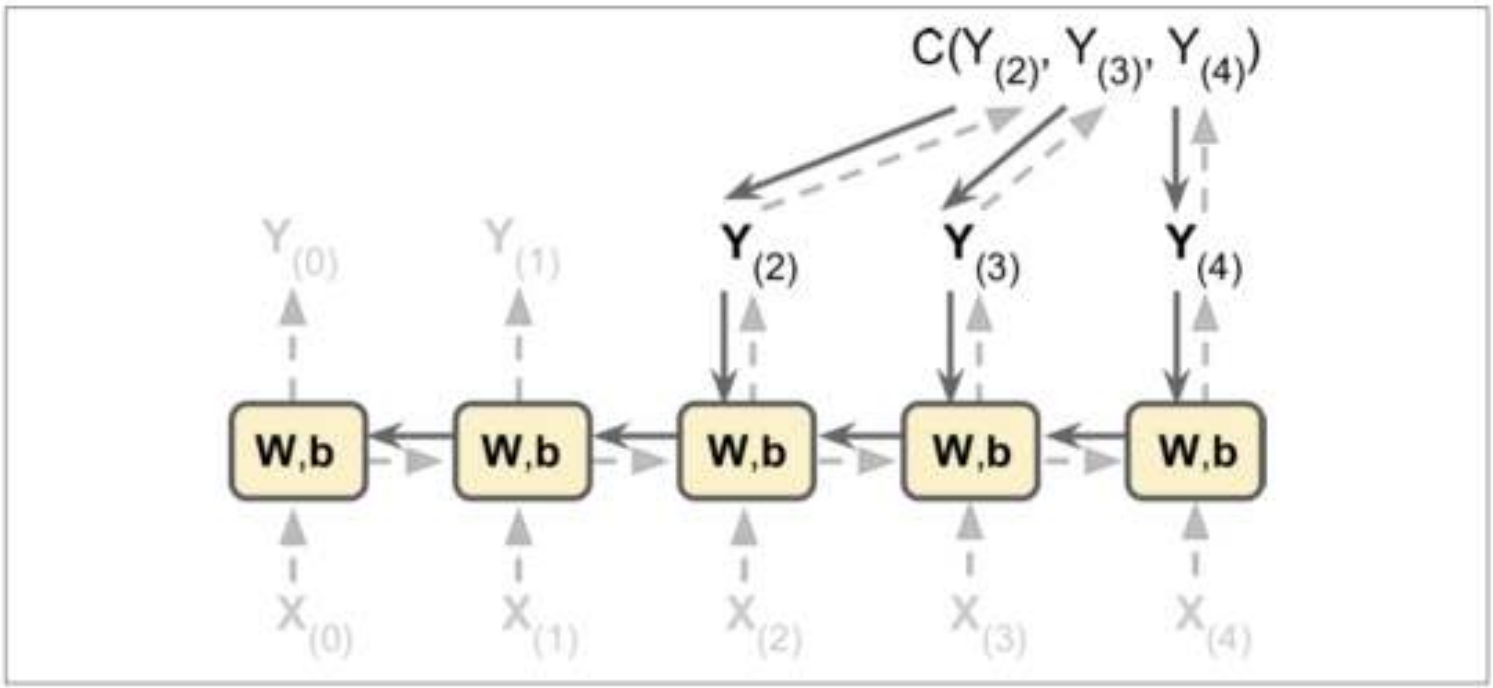


[source : Hands on ML]

- Application : translation of a sentence from one language to another (with a more complex architecture) : Neural Machine Translation (NMT)
  - feed the network a sentence in one language
  - the encoder converts this sentence into a single vector representation
  - the decoder decodes this vector into a sentence in another language.

Training RNNs

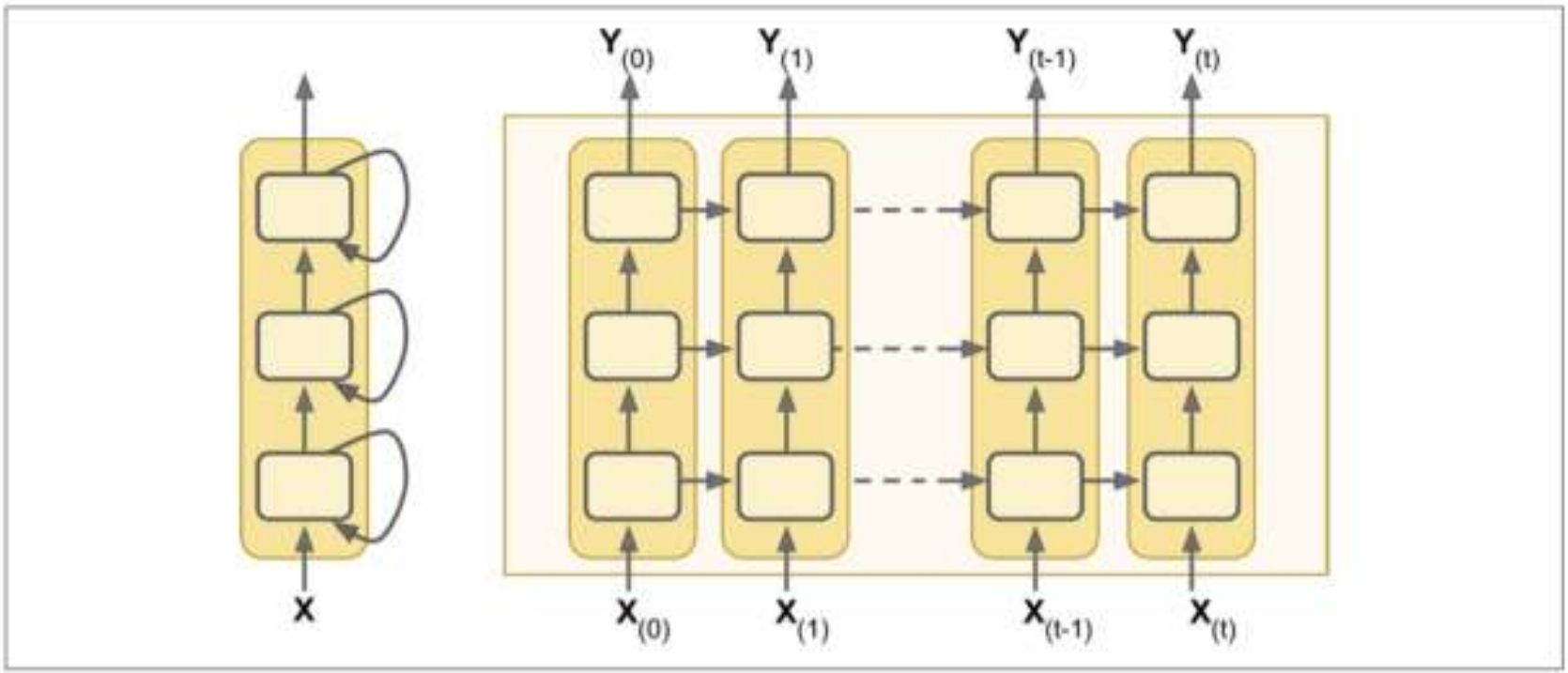
- Forward propagation for computing the loss.
- Back-propagation algorithm called back-propagation through time: unroll the RNN through time and then simply use regular back-propagation.



[source : Hands on ML]

Deep RNNs

- Stacking multiple layers of cells gives you a deep RNN (left), (right: unrolled through time) :

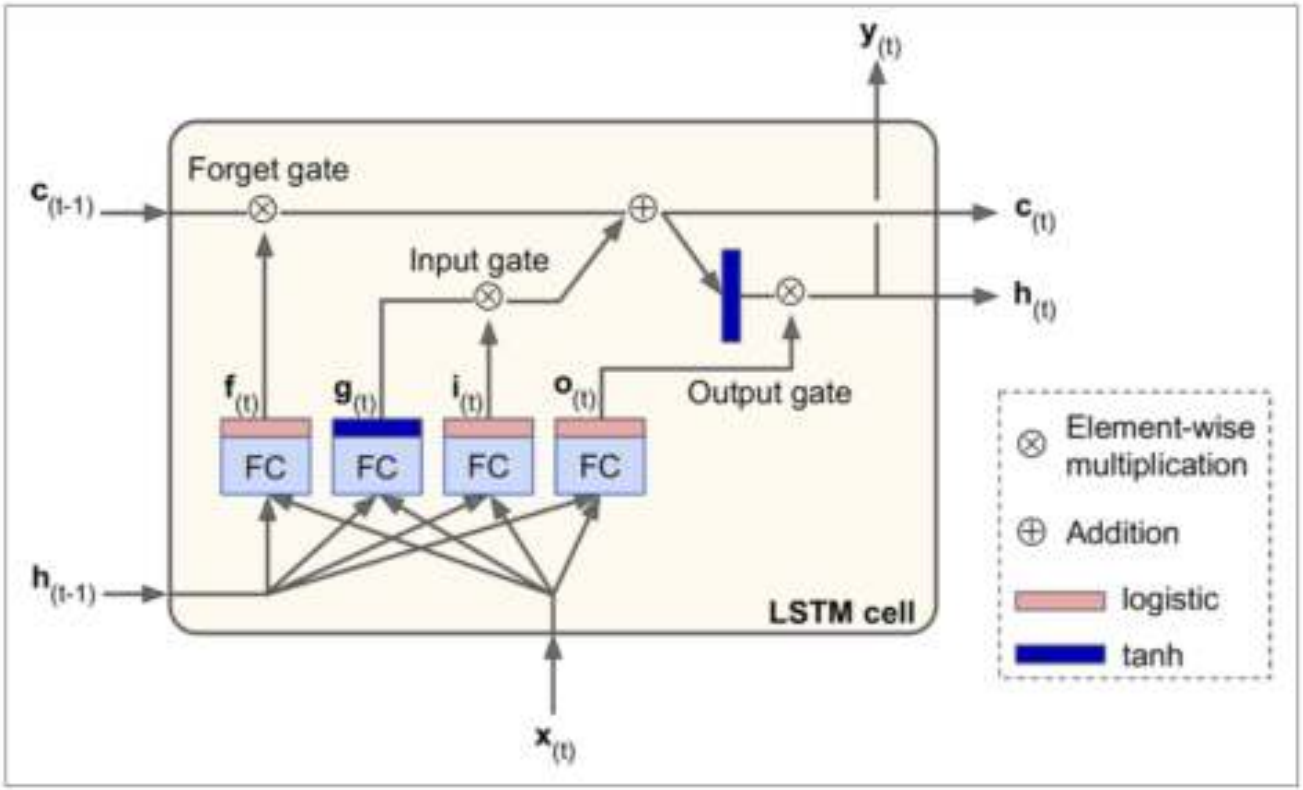


[source : Hands on ML]

The Challenge of Long-Term Dependencies

- Gradients propagated over many stages tend to either vanish most of the time or explode.
- Whenever the model is able to represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction.
- A Long Short-Term Memory (LSTM) cell answers this problem by creating paths through time that have derivatives that neither vanish nor explode.
- LSTM are widely used in natural language processing.

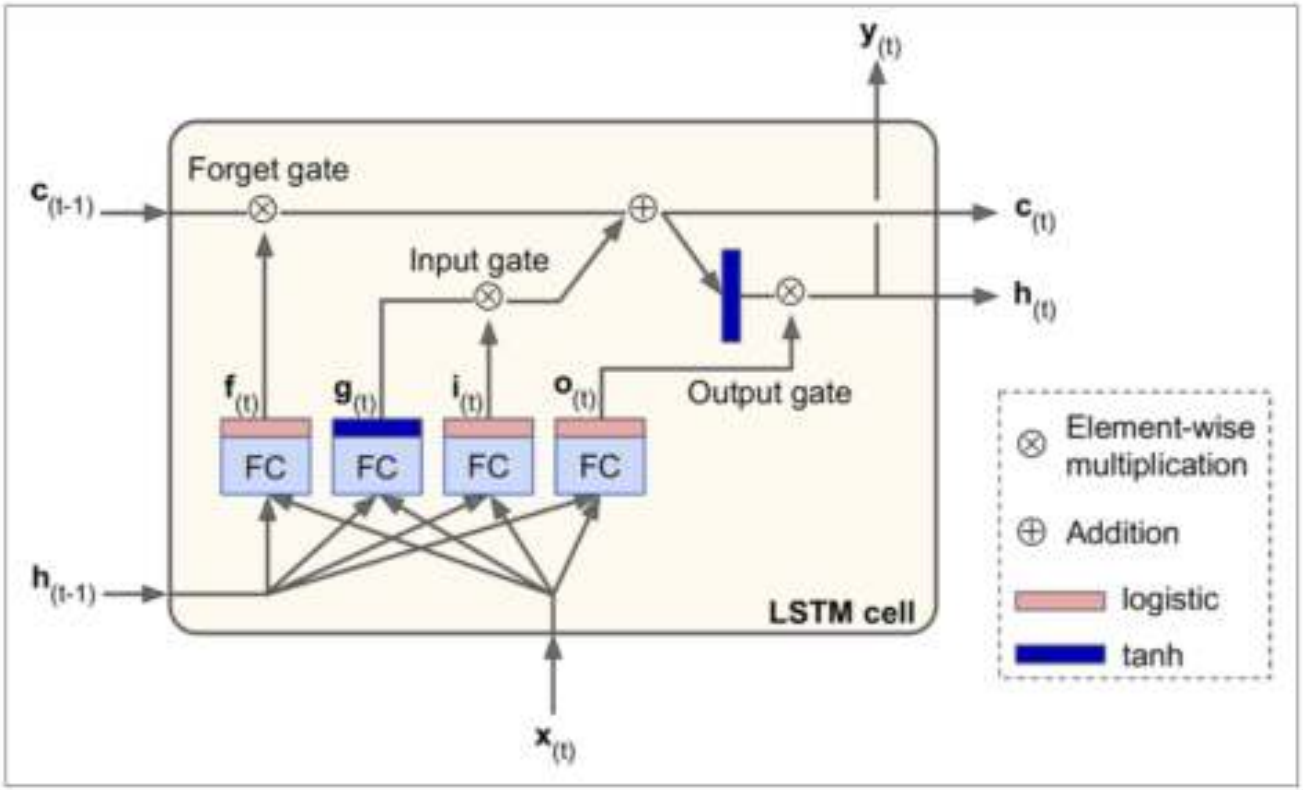
LSTM



The “state" of a LSTM cell is represented by two vectors:

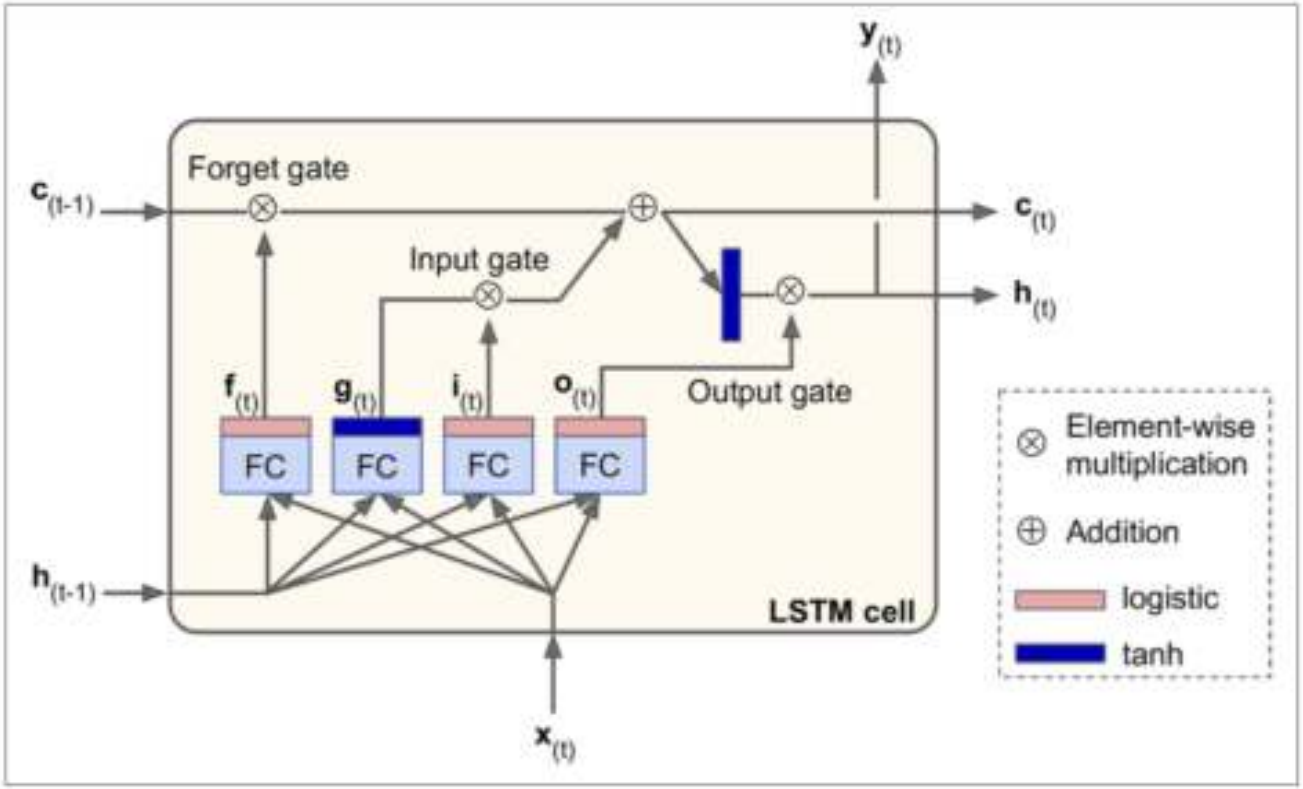
- a short-term state $h(t)$,
- a long-term state $c(t)$.

LSTM



- As the long-term state $c(t-1)$goes through the network from left to right, it first goes through a forget gate, dropping some memories, and then it adds some new memories via the addition operation.
- After the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the output gate. This produces the short-term state $h(t)$ and $y(t)$.

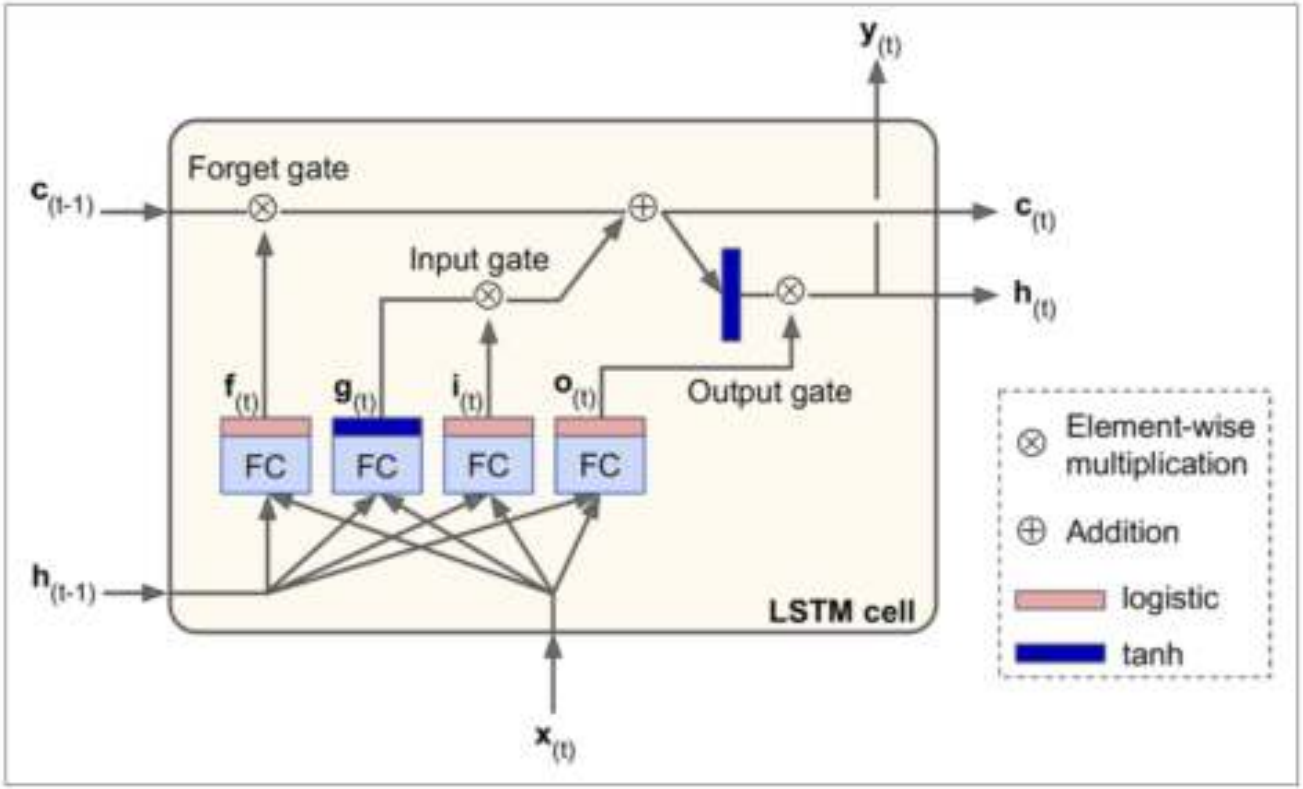
LSTM



The current input vector $x(t)$ and the previous short-term state $h(t-1)$ are fed to four different fully connected layers (FC):

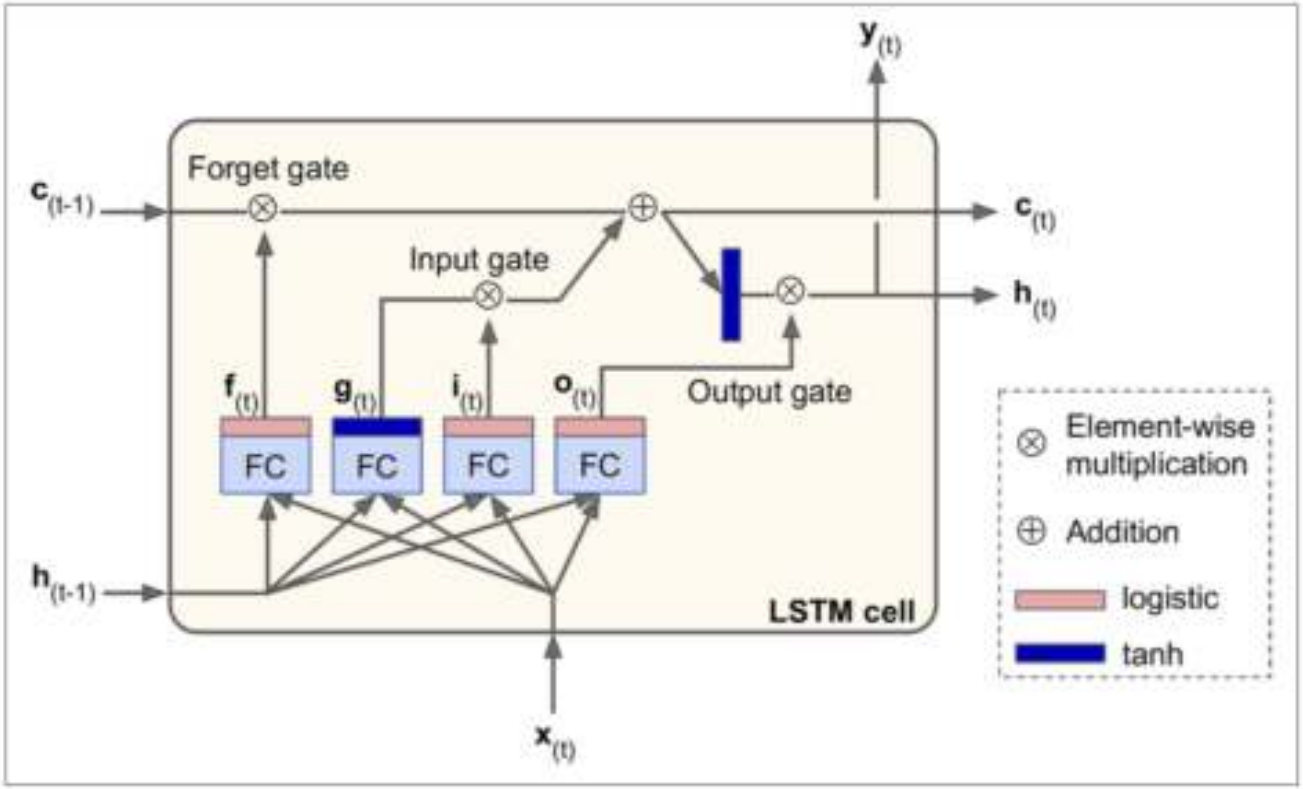
- The main layer is the one that outputs $g(t)$. It analyzes the current inputs $x(t)$ and the previous (short-term) state $h(t-1)$.
- The three other layers are gate controllers. They use the logistic activation function (outputs in [0,1]) and together with the multiplication operations they can open or close the gates.

LSTM



- The forget gate (controlled by $f(t)$) controls which parts of the long-term state should be erased,
- The input gate (controlled by $i(t)$) controls which parts of g(t) should be added to the long-term state,
- The output gate (controlled by $o(t)$) controls which parts of the long-term state should be read and output at this final step.

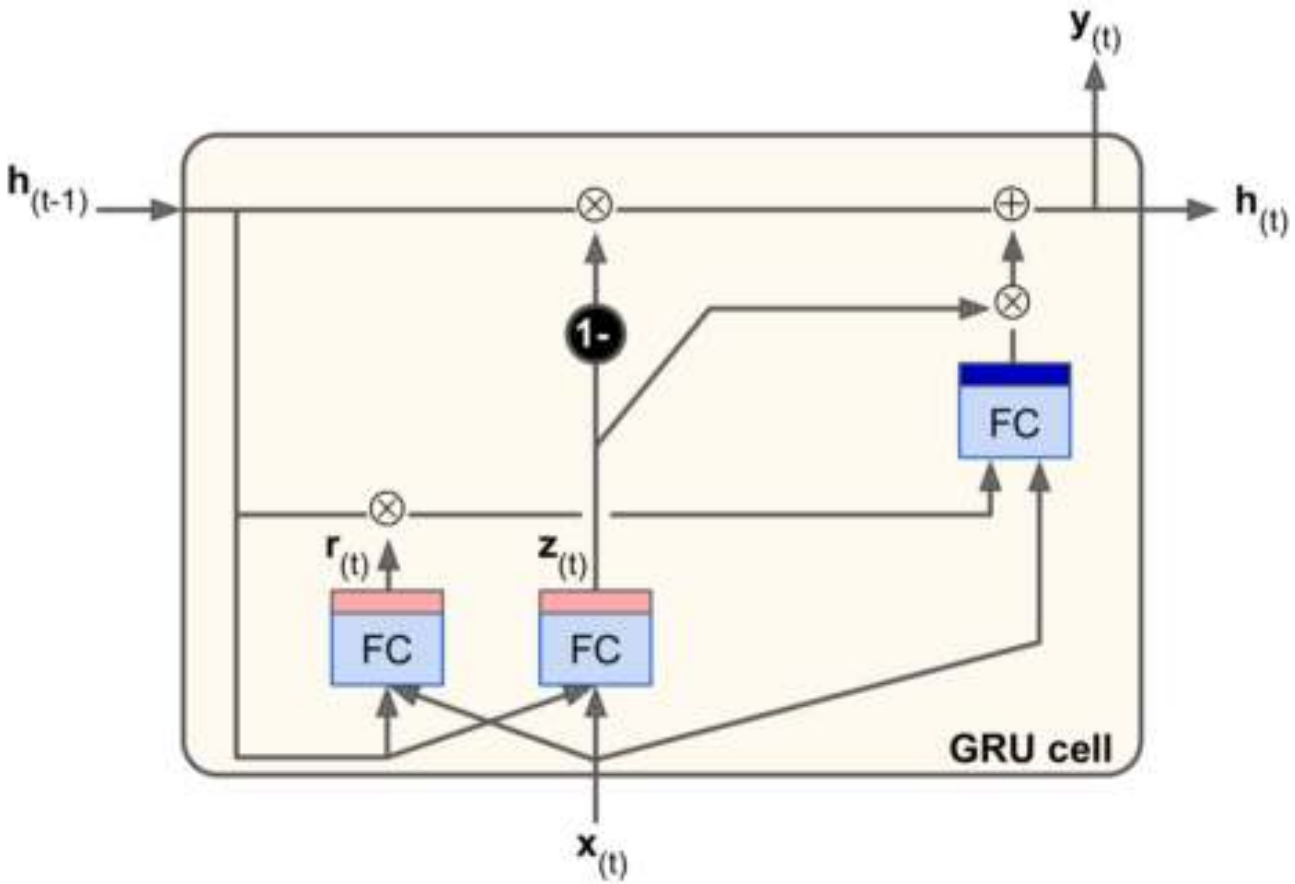
LSTM



In short:

- An LSTM cell can learn to recognize an important input, store it in the long-term state, learn to preserve it for as long as it is needed, and learn to extract it whenever it is needed.
- Efficient for capturing long-term patterns in time series, long texts, audio recordings, and more.

Gated Recurrent Unit (GRU): a popular variant of LSTM



[source : Hands on ML]

- GRU cell is a simplified version of the LSTM
- Both state vectors are merged into a single vector h(t).

Outline

Natural Language programming (NLP)

- Sentence and Document level Classification (topic, sentiment)
- Translation
- Chatbots
- dialogue systems / assistants
- ...

Recommended reading: A Primer on Neural Network Models for Natural Language Processing by Yoav Goldberg.

Word representation: Word2vec

- Words are represented as integers (for features) and 1-hot vectors (for labels when the task is to predic words).
- Large Vocabulary of possible words |V|.
- Word embedding: words or phrases from the vocabulary are mapped to vectors of real numbers (of dimension 50, 100, ...).
- Embeddings are (always) used as inputs in all Deep NLP tasks.

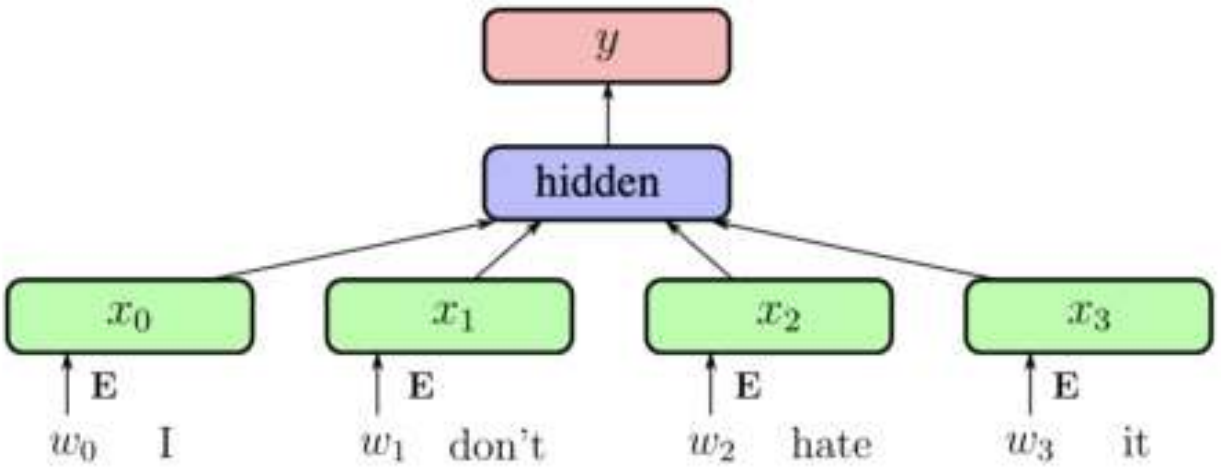
[source : mathworks.com]

Embedding of words

There are two ways to obtain word embeddings:

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine-learning task than the one you are trying to solve. These are called pretrained word embeddings.

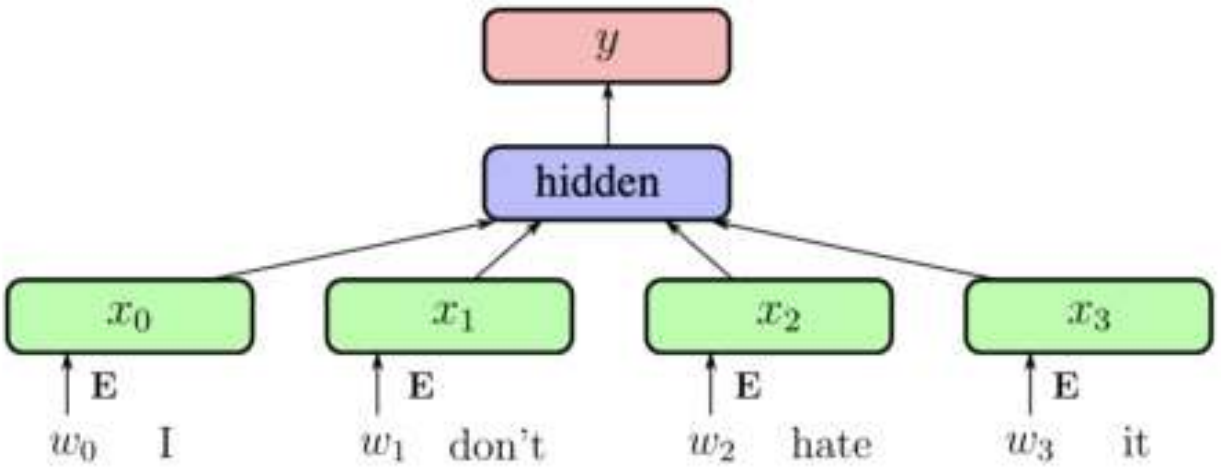
Embedding for text classification



[source : C. Ollion - O. Grisel / S. Gaiffas]

- First layer: embedding **E** of size $|V| \times E$
- Embeddings of the sequence of words are averaged : a single vector $h \in \mathbb{R}^E$ describes the whole sentence.
- h is fed to a dense layer that predicts the label, with softmax activation and cross-entropy loss.

Embedding for text classification



[source : C. Ollion - O. Grisel / S. Gaiffas]

- Very efficient (speed and accuracy) on large datasets,
- Little gains from depth,
- State-of-the-art (or close to) on several classification, when adding bigrams/trigrams.

Ref : Joulin et al. "Bag of tricks for efficient text classification." FAIR 2016.

n-grams

An n-gram is a contiguous sequence of n items from a given sample of text (or speech).

- unigram (1-gram):

a	swimmer	likes	swimming	thus	he	swims

- bigram (2-gram):

a swimmer	swimmer likes	likes swimming	swimming thus	...

- trigram (3-gram):

a swimmer likes	swimmer likes swimming	likes swimming thus	...

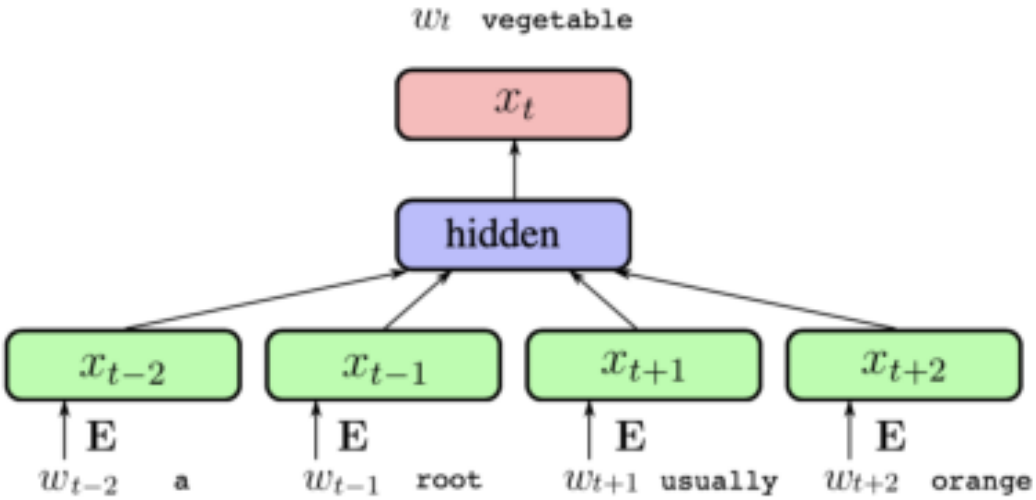
[source : https://sebastianraschka.com]

Learn word embeddings by predicting word contexts

- Aim: find representations that are generic enough to transfer from one task to another.
- Self-supervised learning of word representations: no need for class labels. Self-supervision comes from context.
- Require to learn the embedding on many texts, but unlabelled text data is almost infinite: Wikipedia dumps, Project Gutenberg, Social Networks, etc.
- Two approaches: CBoW and Skip Gram.

Learn embeddings from context: CBoW

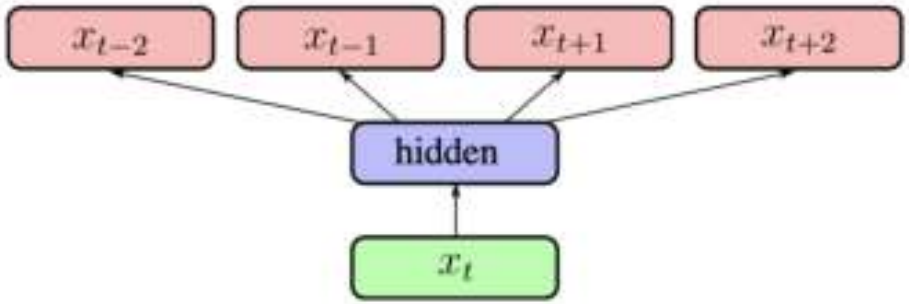
the carrot is a root vegetable, usually orange



[source : C. Ollion - O. Grisel]

- CBoW: representing the context as Continuous Bag-of-Word.
- Similar to text classification with $|V|$ classes.
- One hot encoding of the labels. A “negative" word is one for which we want the network to output 0.
- Huge number of classes: use sample negative words at random instead of computing the full softmax.

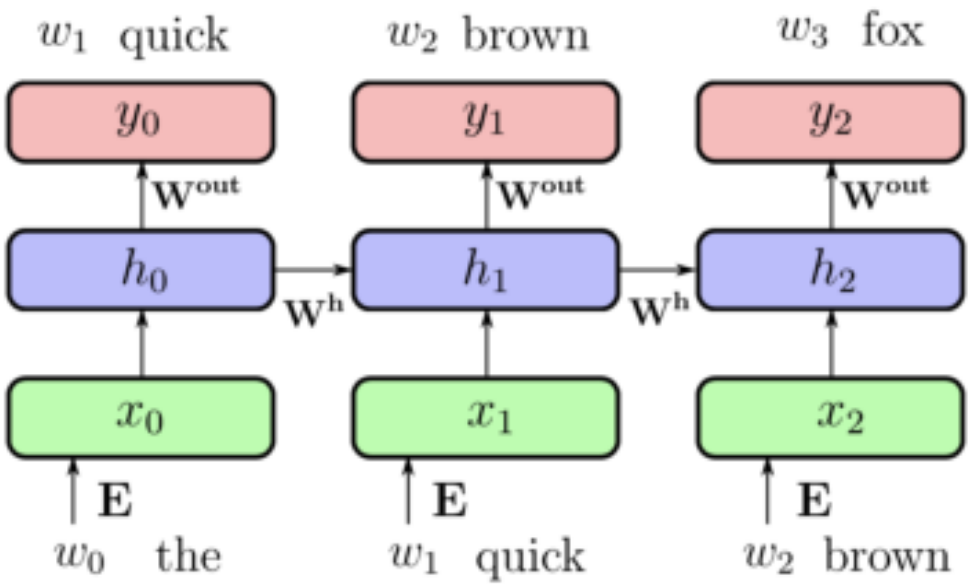
Learn embeddings from context: Skip Gram



[source : C. Ollion - O. Grisel]

- Given the central word, predict occurence of other words in its context.
- Widely used in practice
- Use negative sampling as a cheaper alternative to full softmax.
- Other popular method: GloVe.

Language Modeling and RNNs



$$x_t = \text{Emb}(w_t) = \mathbf{E}w_t \qquad \text{input projection } \mathsf{H}$$

$$h_t = g(\mathbf{W}^{\mathbf{h}} h_{t-1} + x_t + b^h) \qquad \text{recurrent connection } \mathsf{H}$$

$$y = \text{softmax}(\mathbf{W}^{\mathbf{o}} h_t + b^o) \qquad \text{output projection } \mathsf{K} = |\mathsf{V}|$$

[source : C. Ollion - O. Grisel]

- Use simple RNNs cells, LSTM cells, GRU cells...

Take home message

- For text applications, inputs of Neural Networks are Embeddings
  - Little training data and large vocabulary not well covered by training data: use transfer learning with pre-trained embeddings.
  - Large training data with labels: learn directly task-specific embedding in supervised mode.
- RNNs together with embeddings are now very efficient for NLP tasks.
- Going further:
  - Attention mechanism: The attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important.
  - The transformer (Attention Is All You Need): a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output.
  - Application to NLP : BERT Pre-training of Deep Bidirectional Transformers for Language Understanding

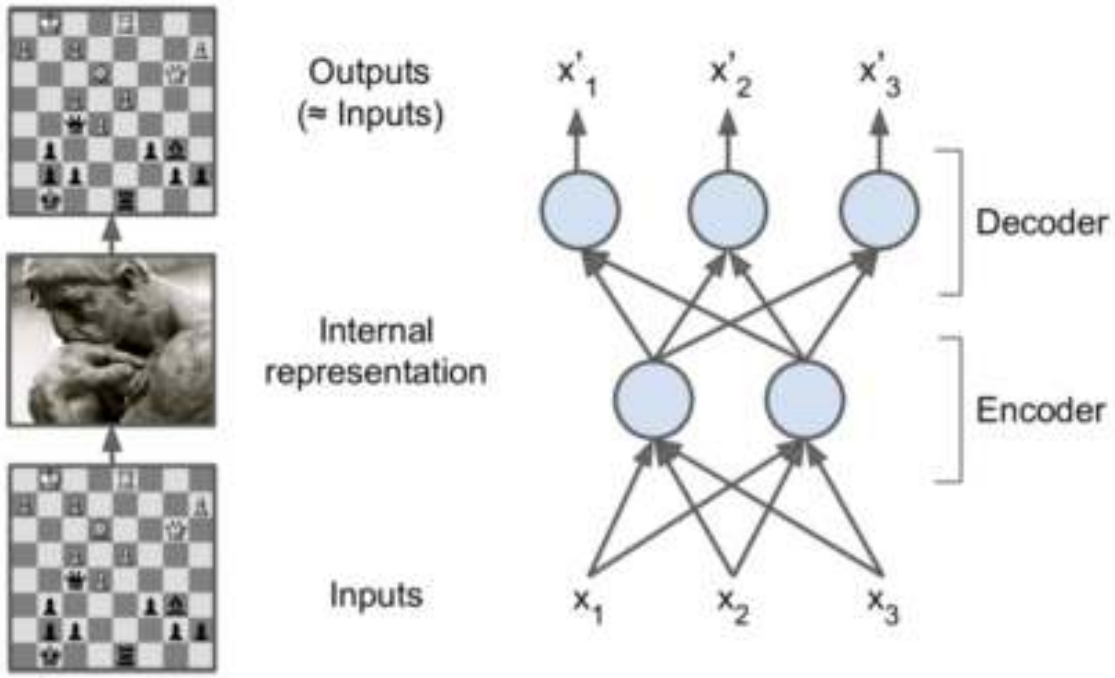
Outline

Autoencoders

- **Unsupervised learning** : Autoencoders learn efficient representations of the input data, called codings, without any supervision.
- **Dimensionality reduction** : These codings typically have a much lower dimensionality than the input data.
- **Featuring** : autoencoders act as powerful feature detectors, and they can be used for unsupervised pretraining of deep neural networks.
- **Generative**: can be used for randomly generating new data that looks very similar to the training data.

Autoencoder architecture and loss

- Same architecture as a MLP except that the number of neurons in the output layer must be equal to the number of inputs.

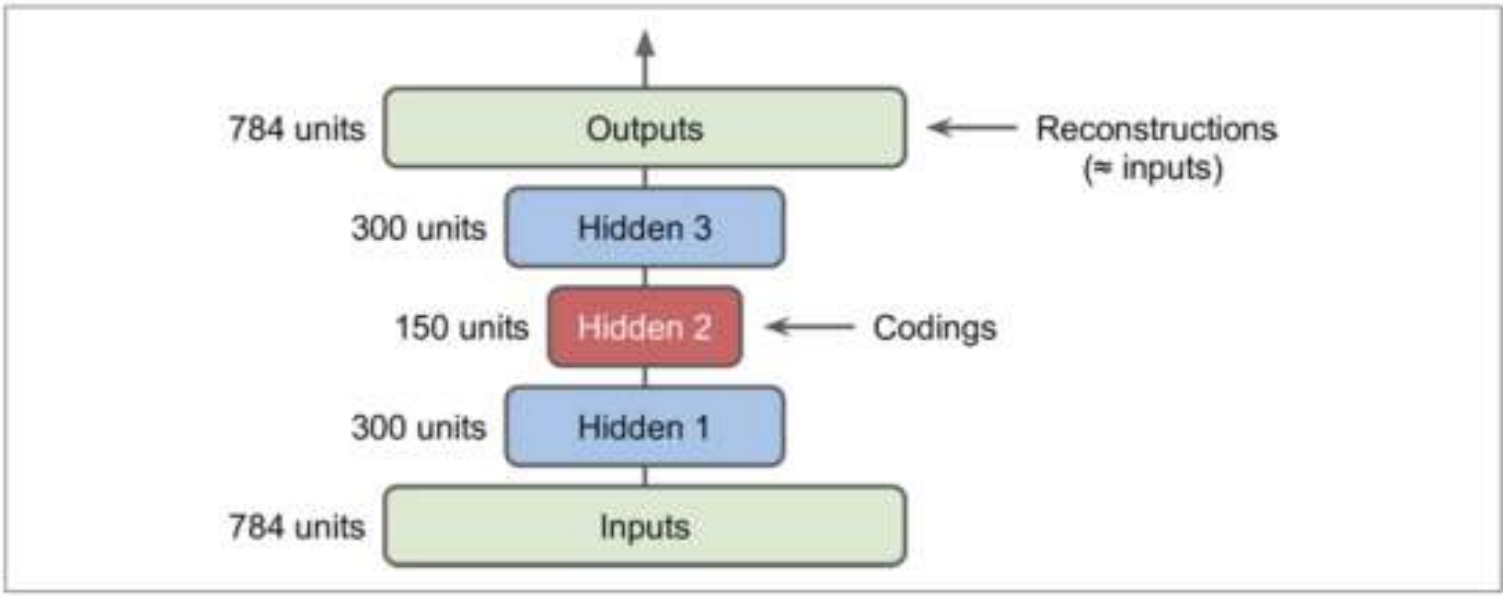


[source : Hands on ML]

- Outputs are called reconstructions, central is is called coding layer.
- Loss : ℓ_2, risk : *MSE*.
- Sparse autoencoder : additional regularization lead to a sparse representation.

Stacked Autoencoders

- Autoencoders with multiple hidden layers : **Deep Autoencoders**.
- **Sandwich architecture:** the architecture is typically symmetrical with regards to the central hidden.

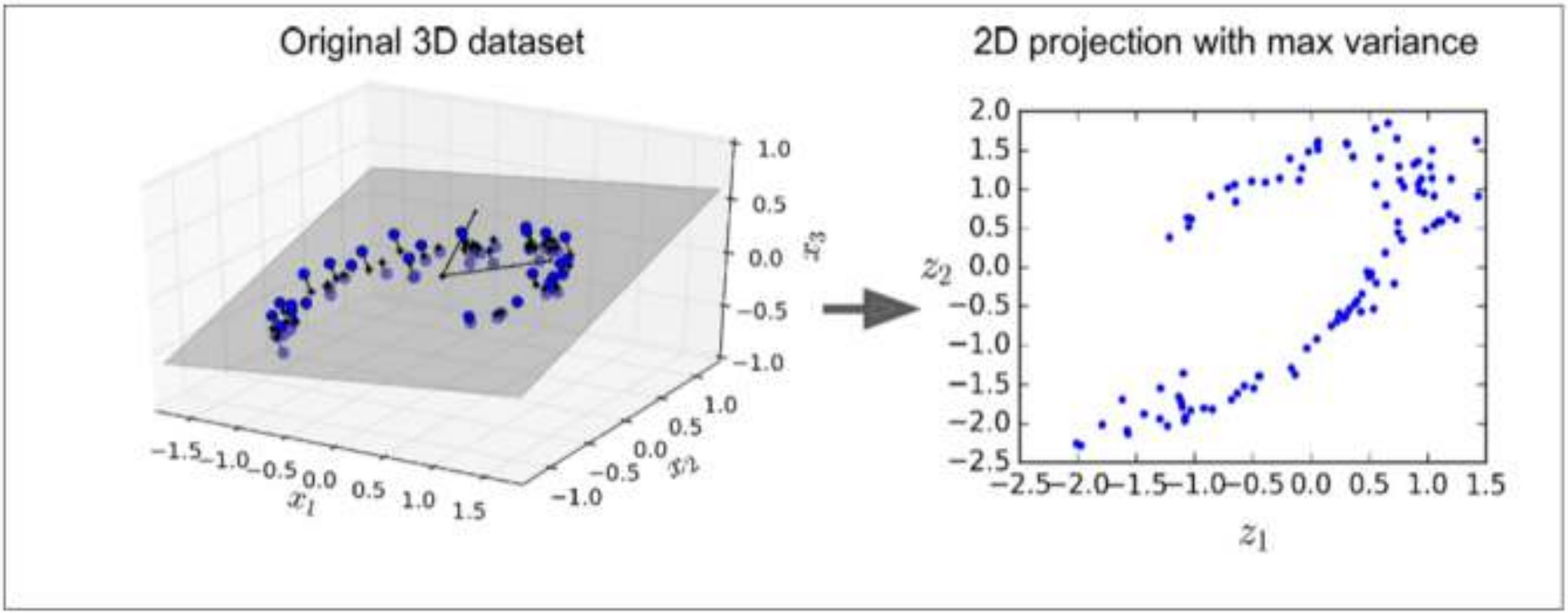


[source : Hands on ML]

- For symmetrical autoencoder we can tie the weights of the decoder layers to the weights of the encoder layers; Formally we impose $W_{N-L+1} = W'_L$.
- This speeds up training and limits overfitting.

Autoencoder as a dimension reduction method

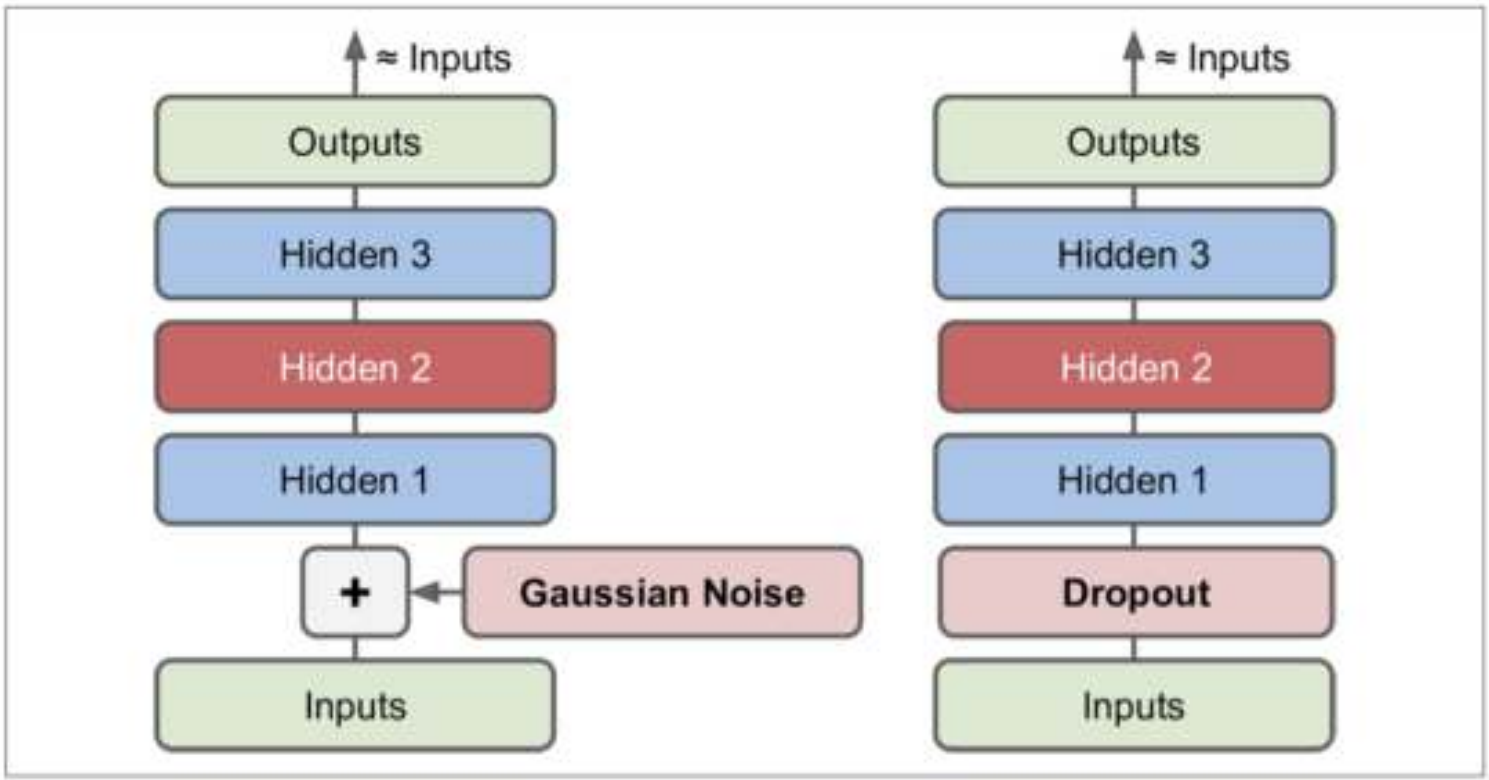
- The coding layer provides a representation of the data in a lower dimension than than the input dimension.



[source : Hands on ML]

Denoising Autoencoders

- Learn useful features by adding noise to its inputs and training it to recover the original (noise-free) inputs:
  - Gaussian noise
  - Drop-out
- Add noise only for the training, not when next applying the trained network.



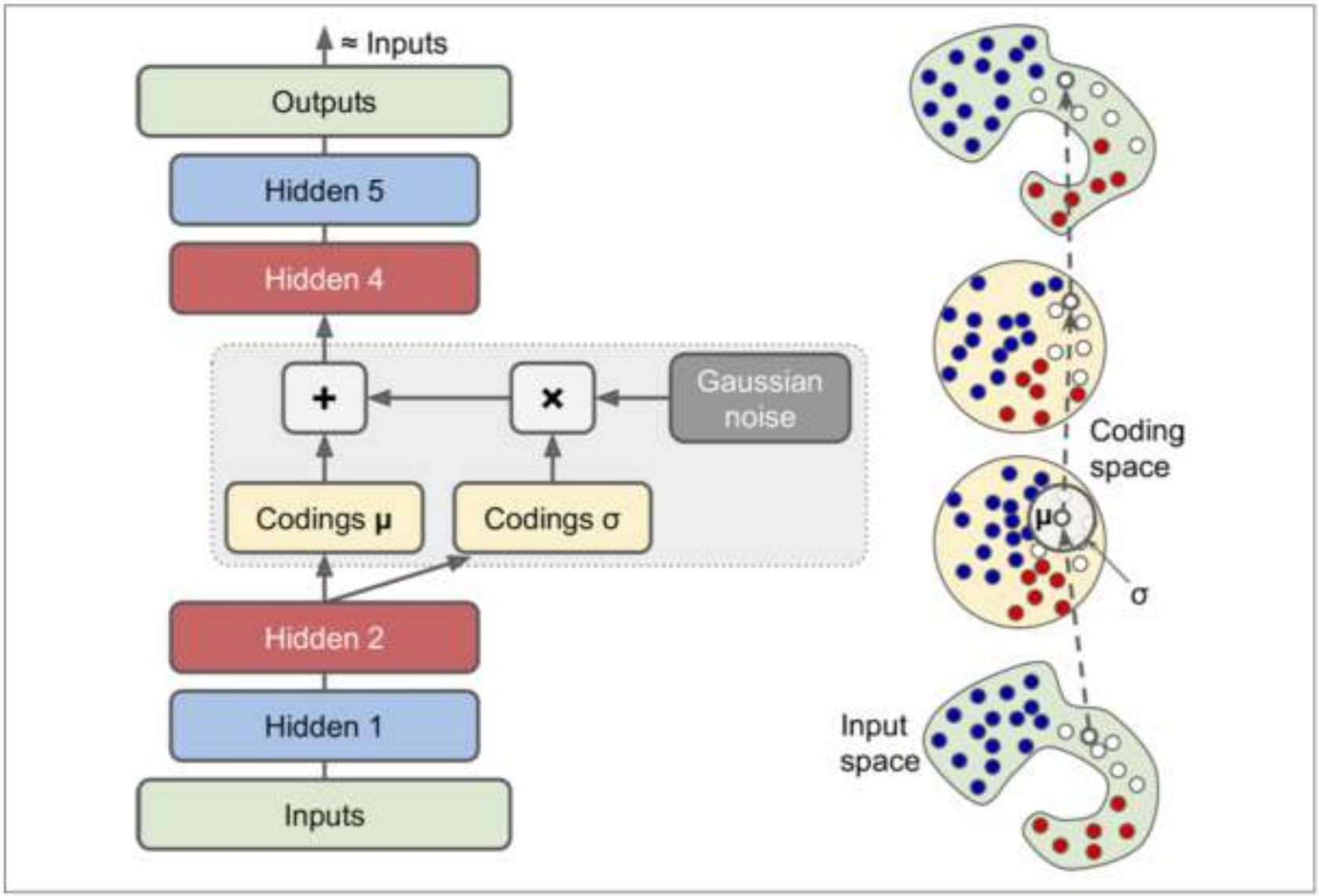
[source : Hands on ML]

Variational Autoencoders

Variational Autoencoders are

- **Probabilistic autoencoders**: outputs of variational Autoencoders are the result of random simuations (non only for training as for denoising autoencoders).
- **Generative autoencoders**: the coding layer learn a distribution and then simulate from this distribution in order to produce output that “looks like" the inputs.

Variational Autoencoders

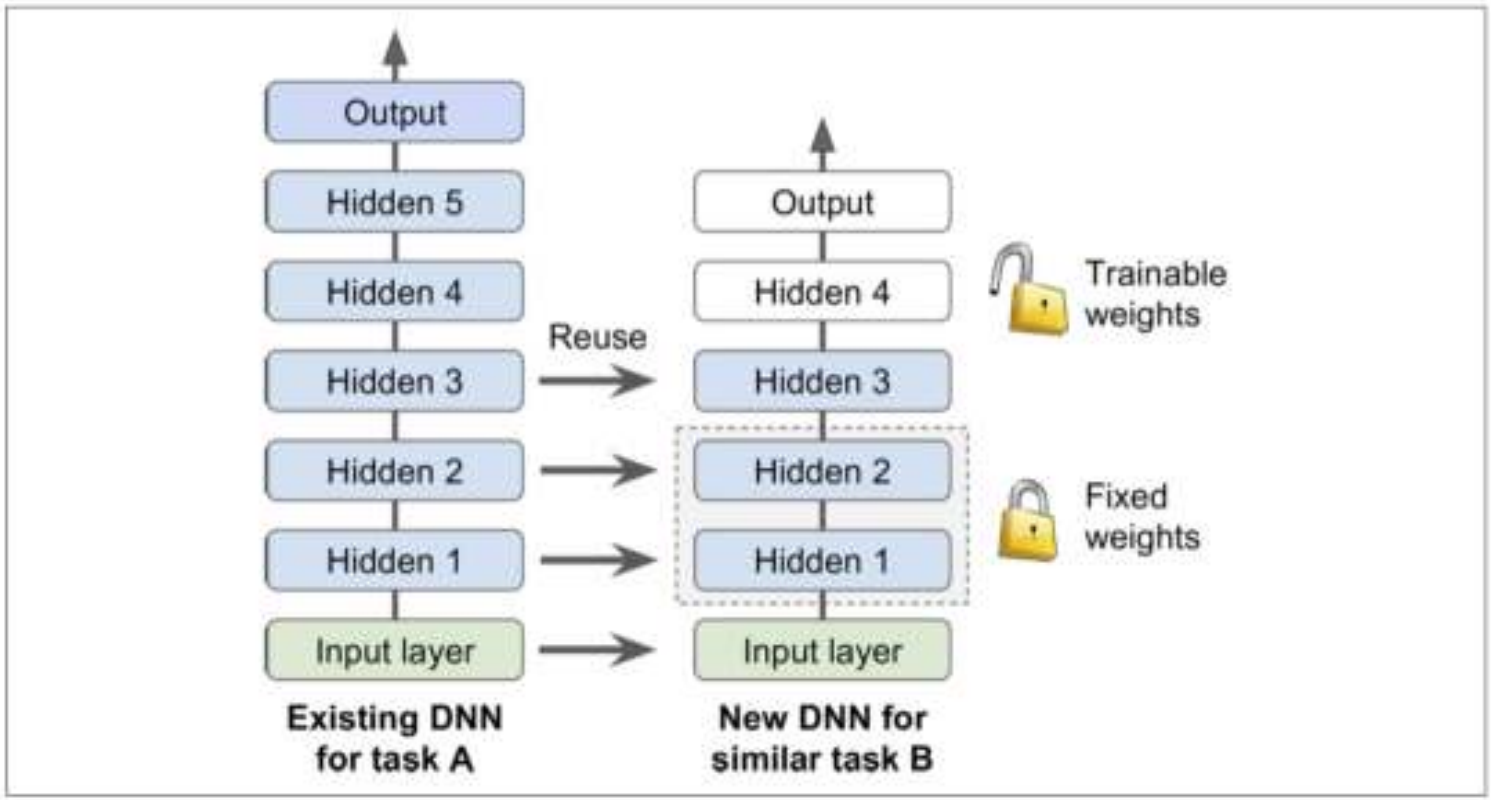


[source : Hands on ML]

Outline

Reusing Pre-trained Networks

- Not a good idea to train a very large Deep NN from scratch.
- **Transfer learning** : find an existing NN that accomplishes a similar task and reuse the lower layers of this network:
  - Speed up training considerably
  - Require much less training data.



[source : Hands on ML]

Frozen layers

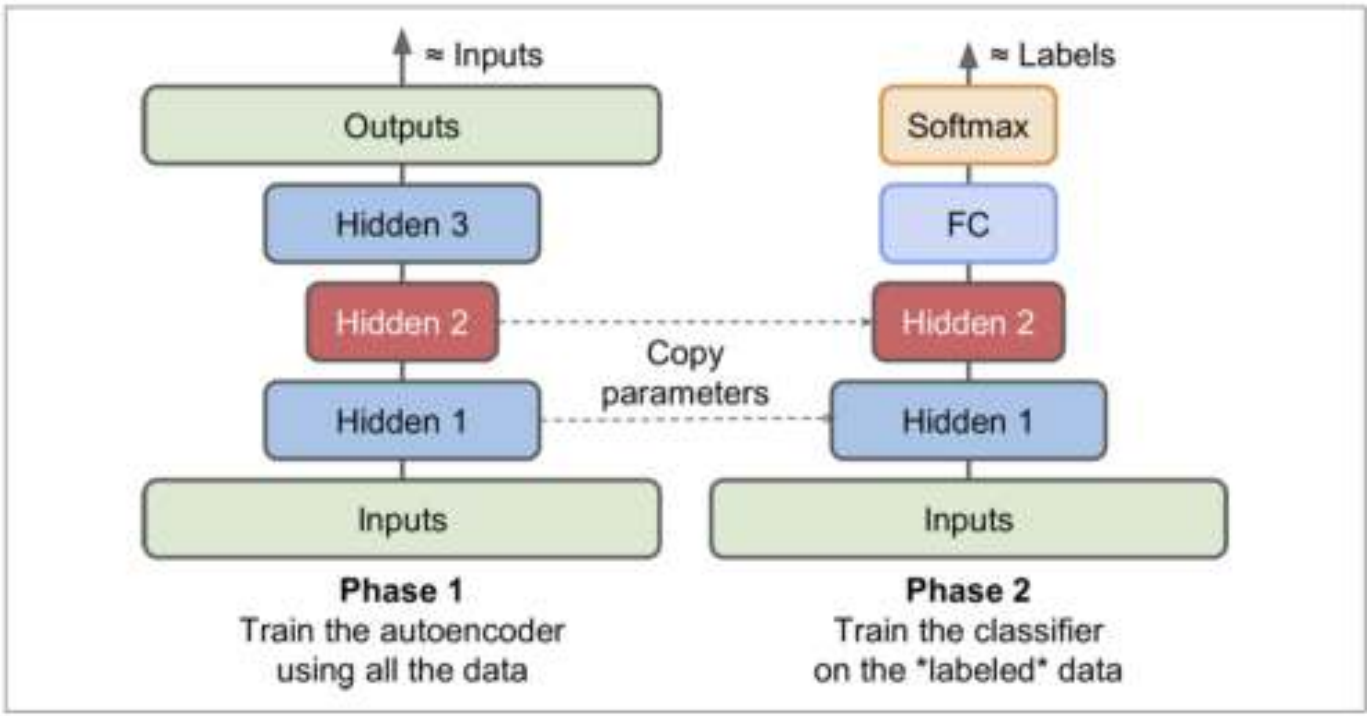
- Suppose that you have trained a Deep NN for a given task.
- For for a new problem, the output layer and upper hidden layers have to be adapted.
- Lower layers have learned to detect low-level features that could be used for other tasks.
- Reuse these layers as they are : “freeze" their weights when training the new DNN.
- Try freezing all the copied layers first, then train your model and see how it performs.
- Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.
- The more training data you have, the more layers you can unfreeze.

Model Zoos

- Many people train Machine Learning models for various tasks and kindly release their pretrained models to the public :
- Model zoo for deep learning
  - modelzoo.co
  - keras.io

Unsupervised pretraining using Autoencoders

- Context : large dataset but most of it is unlabeled.
- Pretraining using autoencoders is an alternative solution to transfer learning;
  - Train a Deep autoencoder using all the data
  - Reuse the lower layers to create a NN for your actual task
  - Train it using the labeled data.



[source : Hands on ML]