Neural Netwoks

Nicolás Pérez de la Blanca

CCIA-UGR

NN vs Nonlinear transformations

$$\mathbf{x} \rightarrow \mathbf{z} = \boldsymbol{\Phi}(\mathbf{x}) = [1, \phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_m(\mathbf{x})]^{\mathrm{T}}$$

$$h(\mathbf{x}) = \theta\left(w_0 + \sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right) \qquad \phi_j(\mathbf{x}) - \text{Basis function}$$

$$h(\mathbf{x}) = \theta\left(w_{01}^{(2)} + \sum_{j=1}^{m} w_{j1}^{(2)} \theta\left(\sum_{i=0}^{d} w_{ij}^{(1)} x_i\right)\right)$$

2-layer NN

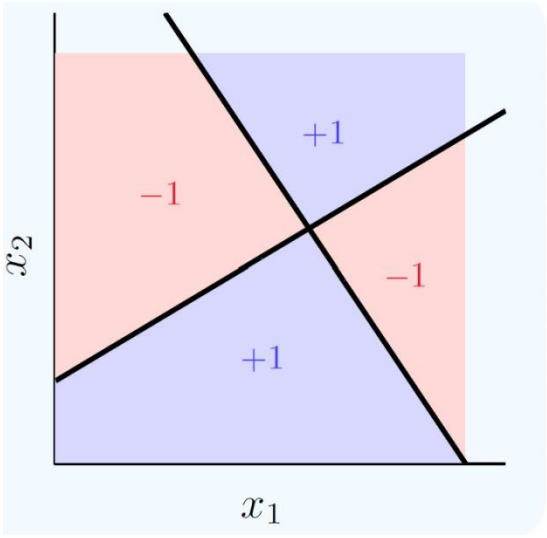
$$h(\mathbf{x}) = \theta\left(w_0 + \sum_{j=1}^{m} w_j \theta\left(\mathbf{v}_j^T \mathbf{x}\right)\right) \qquad \mathbf{v_j} = \mathrm{W}^{(1)}(:, \mathrm{j})$$

- This model can be seen as a linear model with augmented variables…..
- BUT , the dependence on the parameters is NOT linear
  - We will see again this case with the Radial Basis Function Model
- A NN is a Basis function Model where the Basis function to use are learnt from data

Multilayer perceptron: MLP

Neural networks are a generalization of the perceptron which uses a feature transform that is learned from data

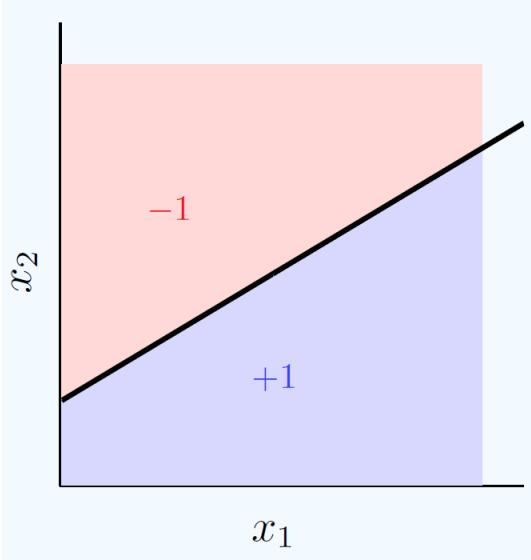
- The perceptron cannot implement simple classification functions: i.e XOR
  - f cannot be written as f=sign($\mathbf{w}^T\ \mathbf{x}$)



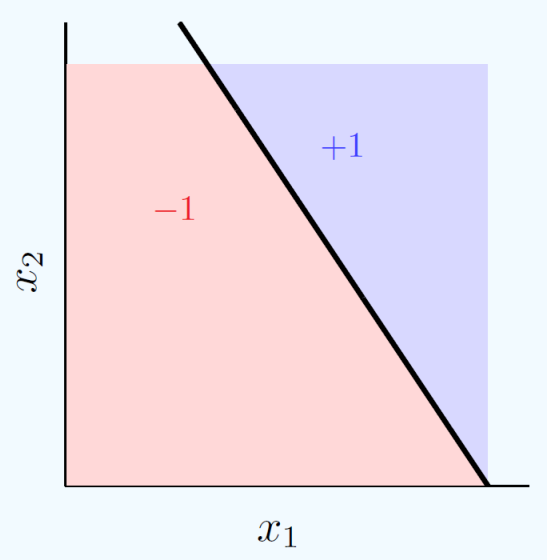
But we can decomposed f into two simple perceptrons, corresponding to the lines in the figure, and then combine the outputs of these two perceptrons in a simple way to get back f.

$$f = h_1\overline{h_2} + h_2\overline{h_1}$$

$h_1(x) = \text{sign}(w^t_1 x)$



$h_2(x) = \text{sign}(w^t_2 x)$

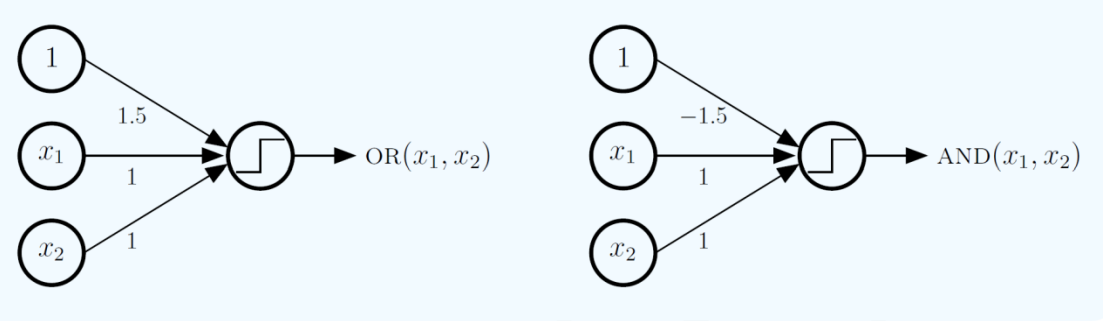


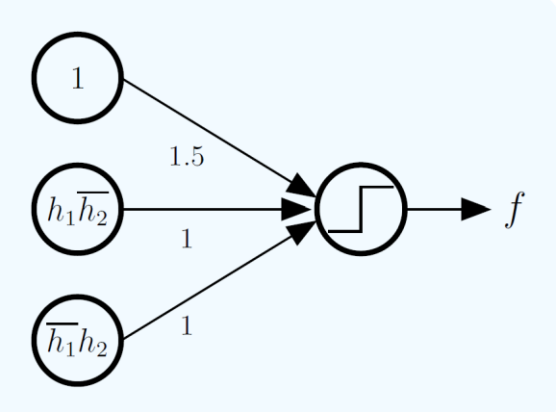
Multi Layer perceptron: MLP (cont`d)

- It can be shown that a complicated target, which is composed of perceptrons, is a **disjunction of conjunctions** (OR of ANDs) applied to the component perceptrons

- OR and AND can be implemented by the perceptron

$$OR(x_1, x_2) = sign(x_1 + x_2 + 1.5)$$
$$AND(x_1, x_2) = sign(x_1 + x_2 - 1.5)$$

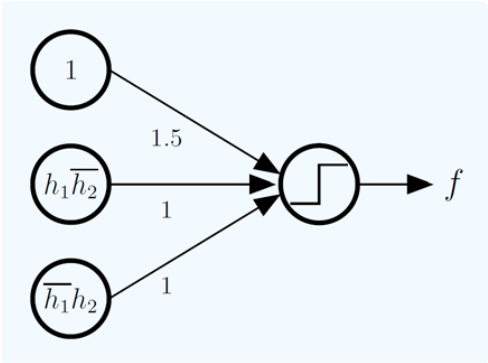




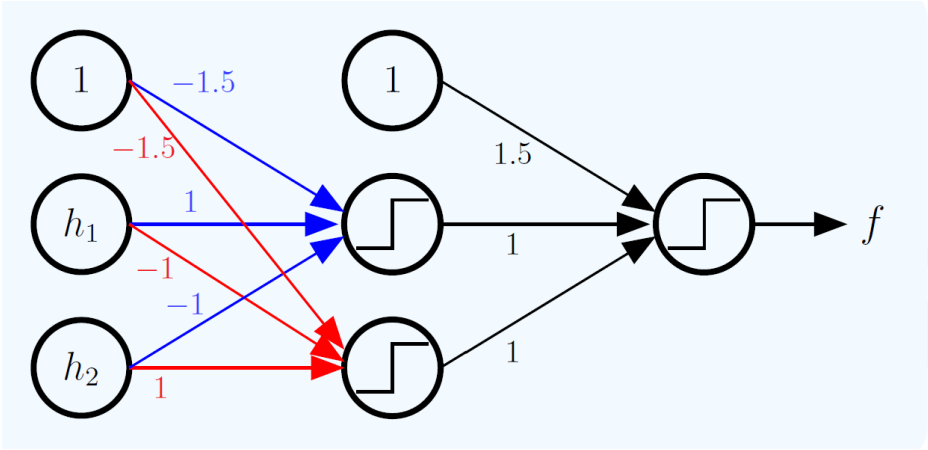
The target *f* equals +1 when exactly one of h_1 , h_2 equals +1

$$f = h_1\overline{h_2} + h_2\overline{h_1}$$

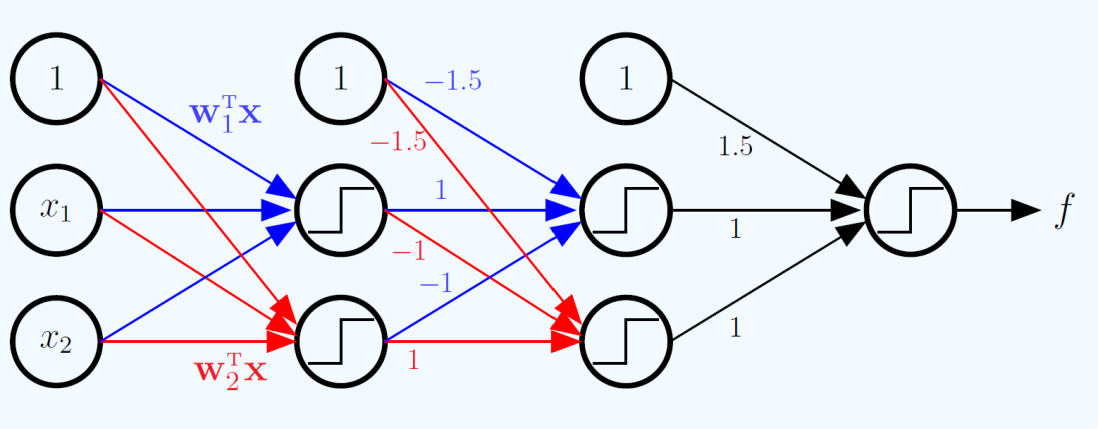
MLP: Complex target



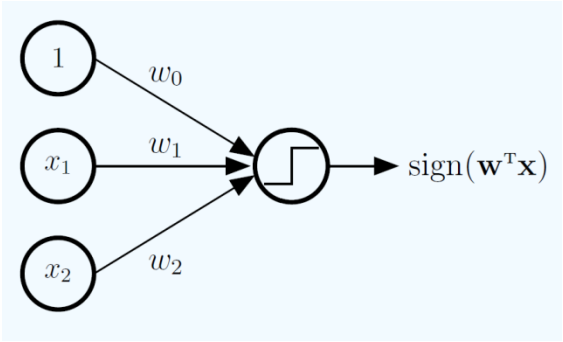
$$f = h_1\overline{h_2} + h_2\overline{h_1}$$

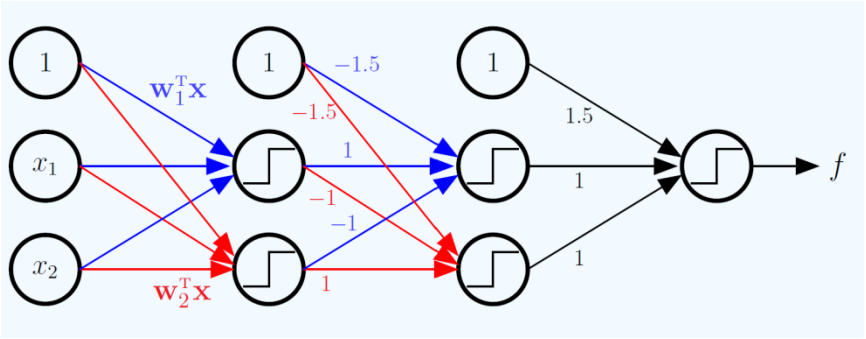


- The resulting graph representation of f is



Multi-Layer perceptron: MLP (cont`d)





Let's compare the graph function of a perceptron (left) with the graph function of *f* (right)

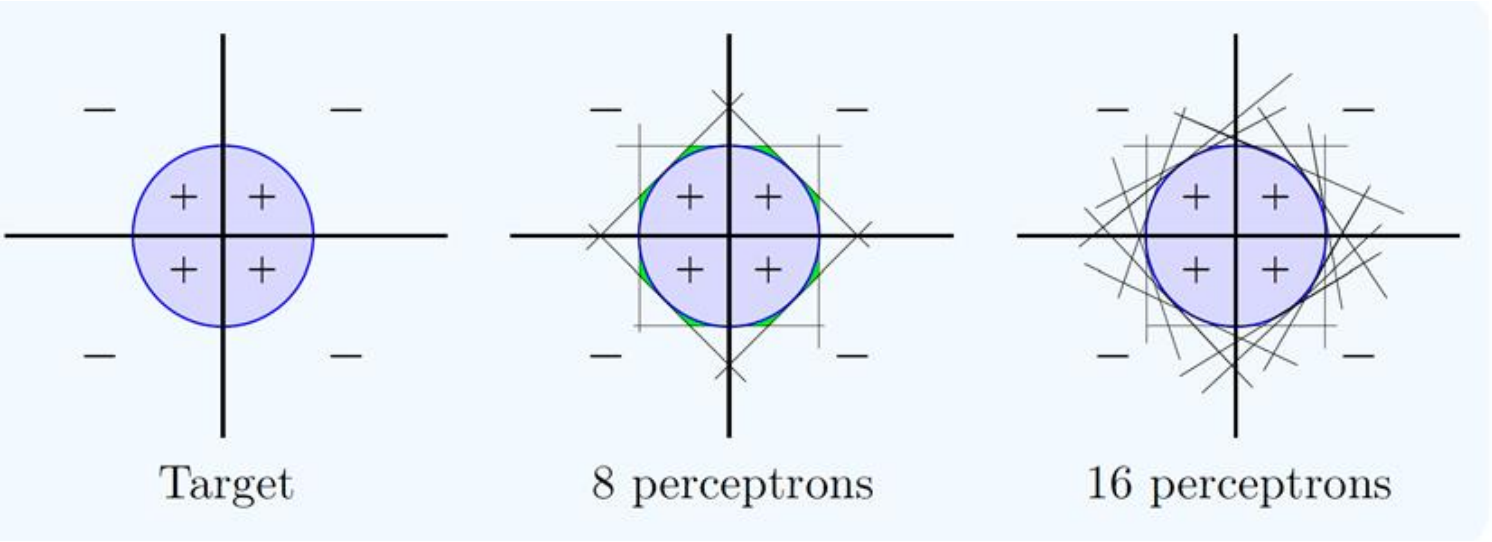
- The graph on the right has more layers: Multilayer perceptron
- The additional layers are called ***hidden layers***

The addition of hidden layers is what allowed us to implement the more complicated target.

We distinguish three types of layers: input, hidden and output

The number of layers in a network is the #hidden+1

Multi-Layer perceptron: MLP (cont`d)



- If *f* can be decomposed into perceptrons using an *OR* of *AND*s, then it can be implemented by a 3-layer perceptron.
- If *f* is not strictly decomposable into perceptrons, but the decision boundary is smooth, then a 3-layer perceptron can come arbitrarily close to implementing *f*.
- **Adding new nodes in the hidden layers is the mechanism to model more complex target functions**

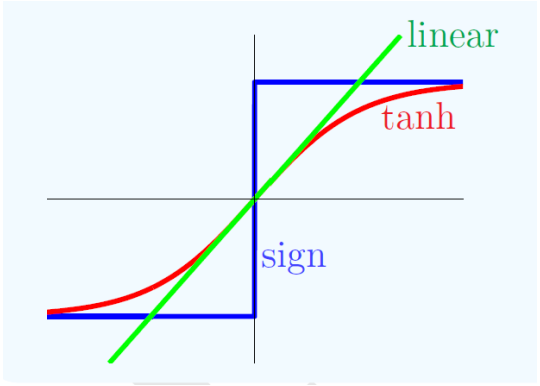
MLP: Activation function

- Number of layers and number of units by layer define the MLP architecture
- Once we fix the architecture (equivalent to fix $\mathcal{H}$) we have to estimate the parameters (weights on every link)

In the simple perceptron we have $f(\mathbf{x}) = \theta(\mathrm{w}^T\mathbf{x})$ where $\theta(x) = \mathrm{sign}(x)$

In the perceptron is a NP-problem, therefore is even harder for the MLP model

The solution is to aproach sign( ) by
a differentiable function: $\theta(x) = \tanh(x)$



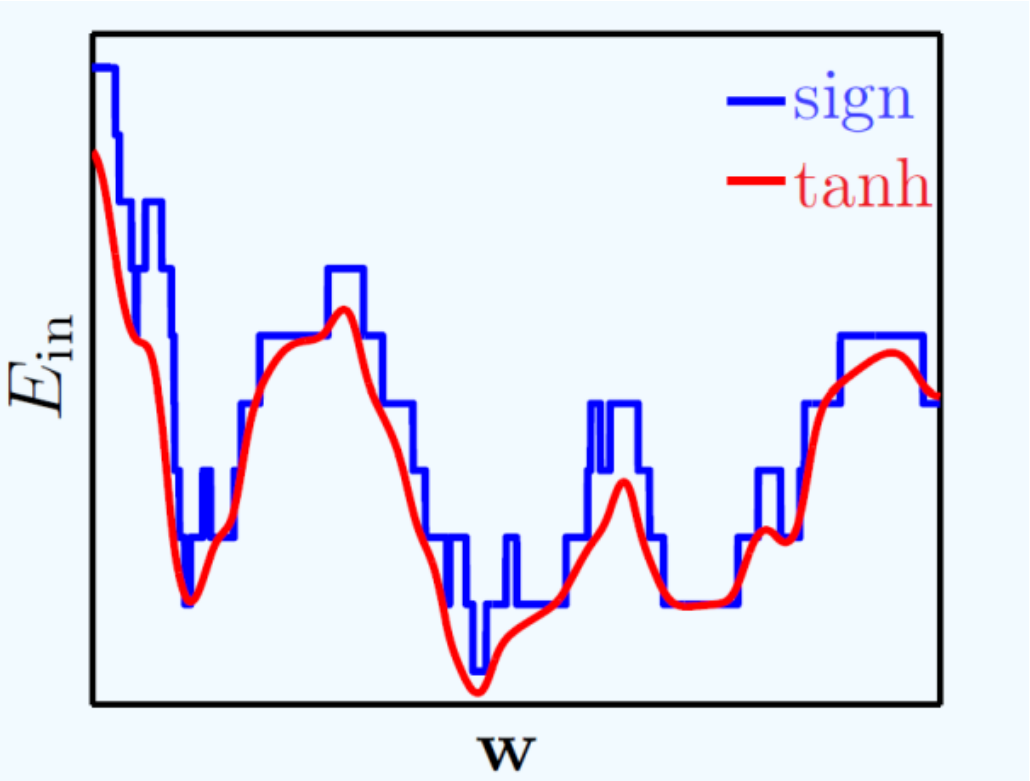
Weights learned using the sigmoidal neural network with tanh(·) activation function can be used for classification by replacing the output activation function with sign (·)

The networks using sigmoid or tanh are called sigmoidal neural networks

Sign() vs Tanh()

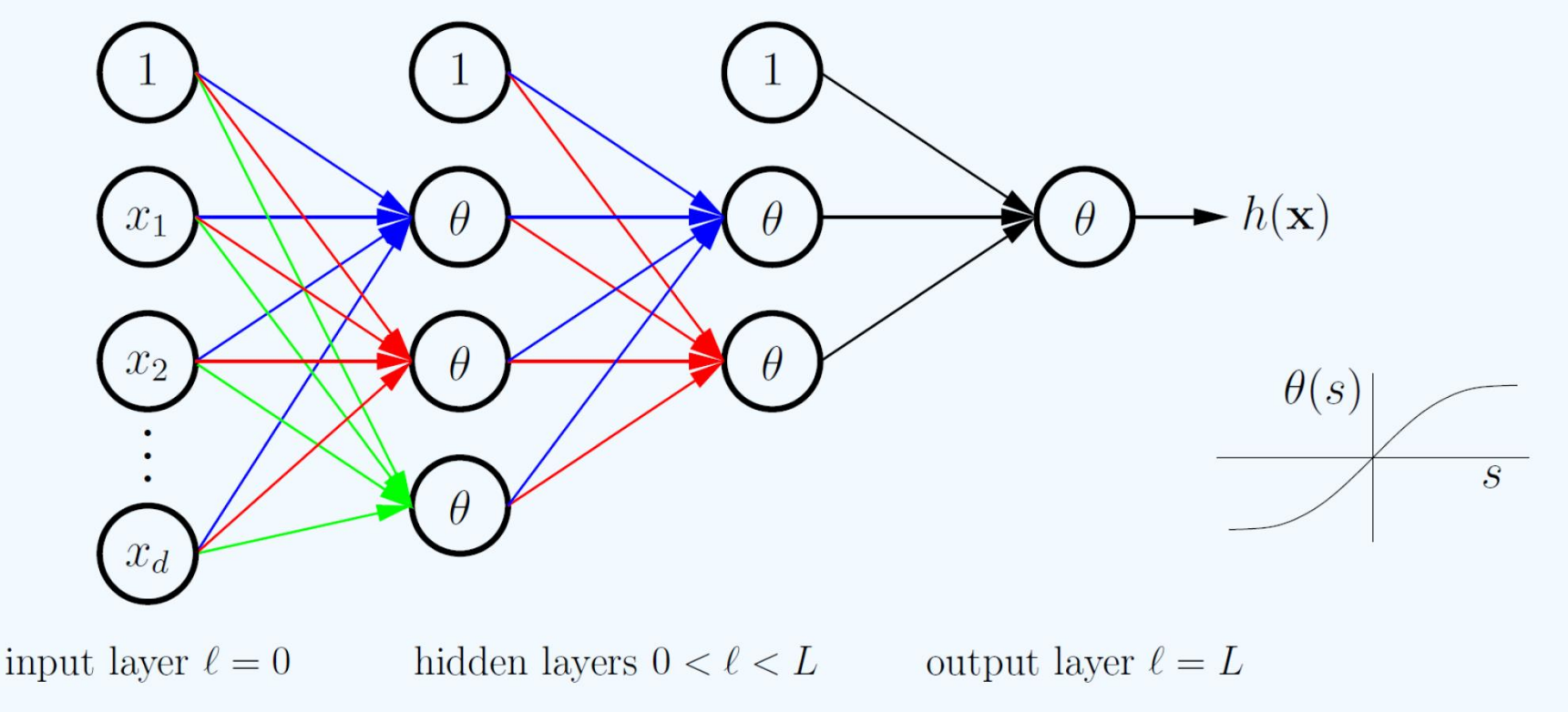
- The function **sign()** can be closely aproximated by the **tanh()** function (prove it)

- The figure shows as the In-sample error varies with one of the weights of **w** using the function sign() and the function tanh().



It is clear that the function **tanh()** capture the general form of the error, so that if we minimize the sigmoidal in-sample error, we get a good approximation to minimizing the in-sample classification error

Feed-Forward Neural Network



- The Neural Network is our “softened” version of the MLP
- **We only consider feed-forward models**
- Some notation is needed in order to understand how the function is computed

FF-NN: Notation

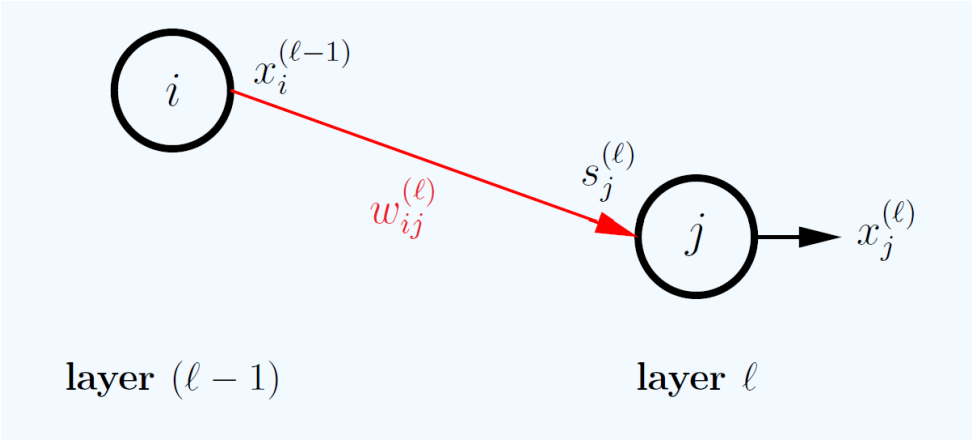
- Layers are labeled by $\ell = 0, 1, 2, \ldots, L$. In our example above, $L = 3$, i.e.
  - input layer $\ell = 0$. Usually not considered a layer
  - output layer : $\ell = L$, determine the value of the function
  - hidden layers : $0 < \ell < L$
  - We will use superscript(ℓ) to refer to a particular layer.
- Each layer ℓ has 'dimension' $d(\ell)$, which means that it has $d(\ell) + 1$ nodes,
  - labeled $0, 1, \ldots, d(\ell)$. The node labeled as 0 I called the bias node .
  - This bias node is set to have an output 1, which is analogous what happened in the linear models
- The output layer will use activation function according to the problem
  - Classification: sign ( )
  - Regression: Identity
  - Probability estimation: Sigmoidal or SoftMax (output vector)

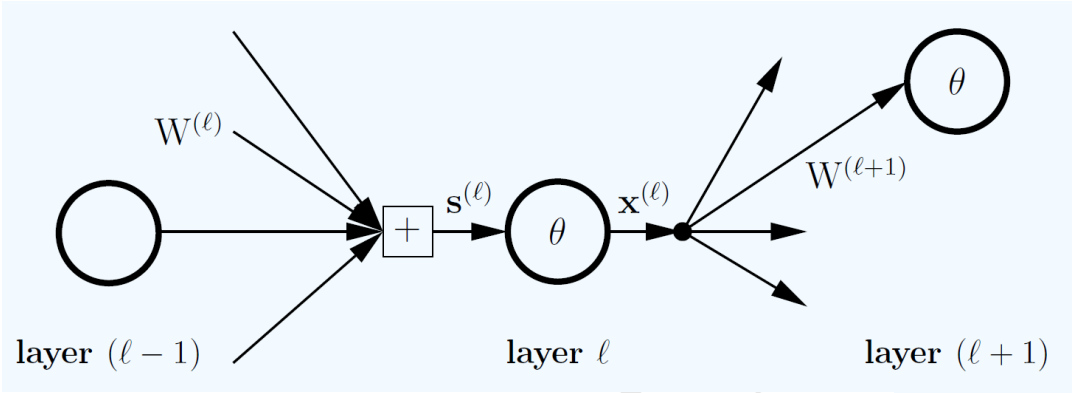
- The neural network model $\mathcal{H}_{nn}$ is specified once you determine the architecture of the neural network,
  - that is the dimension of each layer $\mathbf{d} = [d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$ (L is the number of layers).

- A hypothesis $h \in \mathcal{H}_{nn}$ is specified by selecting weights for the links

FF-NN: Notation (cont'd)

- A **node** has an **incoming signal s** and an **output x**.
- The weights are indexed by the layer into which they go.
- We use subscripts to index the nodes in a layer.
- $w^{(\ell)}_{ij}$ is the weight into node j in layer ℓ from node i in the previous layer,
- The signal going into node j in layer ℓ is $s^{(\ell)}_j$, and the output of this node is $x^{(\ell)}_j$.
- There are some special nodes in the network.
  - The zero nodes in every layer are constant nodes, set to output 1.
  - They have no incoming weight, but they have an outgoing weight.
  - The nodes in the input layer $\ell = 0$ are for the input values, and have no incoming weight or transformation function.





signals in	$\mathbf{s}^{(\ell)}$	$d^{(\ell)}$ dimensional input vector
outputs	$\mathbf{x}^{(\ell)}$	$d^{(\ell)}+1$ dimensional output vector
weights in	$\mathrm{W}^{(\ell)}$	$(d^{(\ell-1)}+1)\times d^{(\ell)}$ dimensional matrix
weights out	$\mathrm{W}^{(\ell+1)}$	$(d^{(\ell)}+1)\times d^{(\ell+1)}$ dimensional matrix

Forward propagation

Forward propagation to compute $h(\mathbf{x})$:

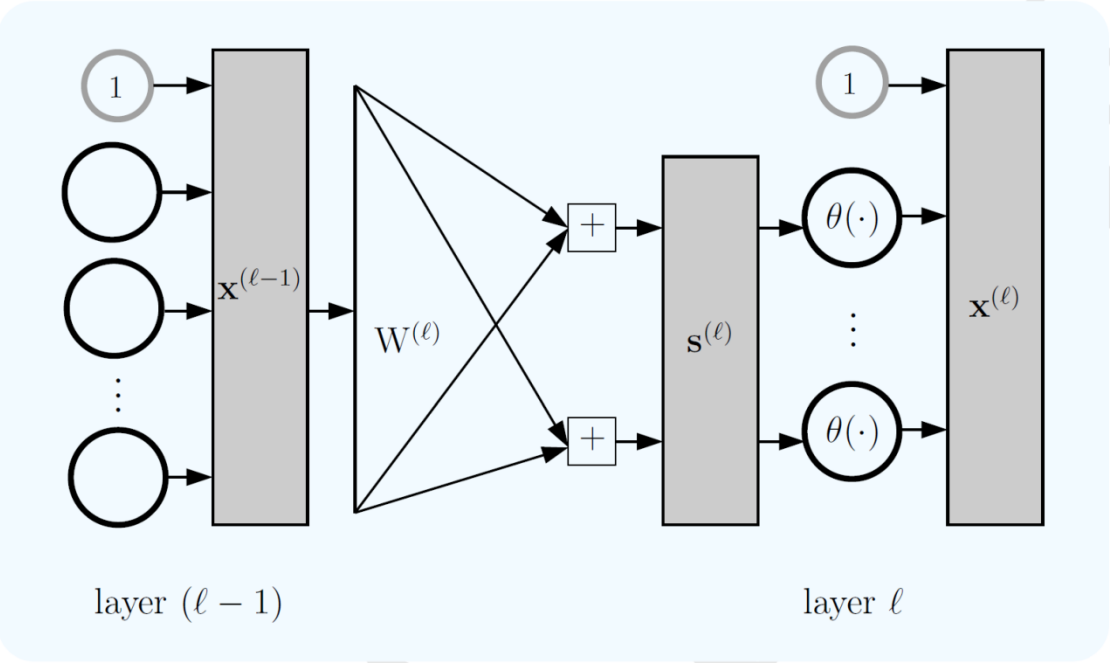
1: $\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**

2: **for** $\ell = 1$ to L **do** **[Forward Propagation]**

3: $\quad \mathbf{s}^{(\ell)} \leftarrow (\mathrm{W}^{(\ell)})^{\mathrm{T}} \mathbf{x}^{(\ell-1)}$

4: $\quad \mathbf{x}^{(\ell)} \leftarrow \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}$

5: $h(\mathbf{x}) = \mathbf{x}^{(L)}$ **[Output]**



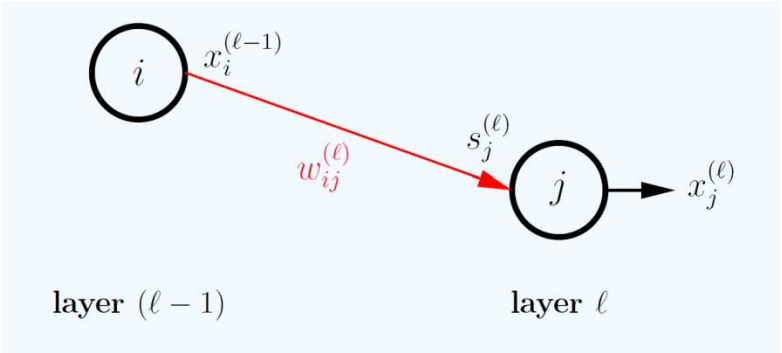
Forward propagation (cont´d)

- The neural network hypothesis h(x) is computed by the *forward propagation* algorithm

- Input and output of a node is given by the transformation function: $\boldsymbol{x}^{(l)} = \begin{pmatrix} 1 \\ \theta(\boldsymbol{s}^{(l)}) \end{pmatrix}$

$$s_j^l = \sum_{i=0}^{d^{(l-1)}} w_{ij}^l x_i^{l-1}$$

$$\mathbf{s}^{(l)} = \left(\mathbf{W}^{(l)}\right)^T \mathbf{x}^{(l-1)}$$



$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{\mathbf{W}^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{\mathbf{W}^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \longrightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x}).$$

$$E_{in} = \frac{1}{N}\sum_{n=1}^{N} (h(\mathbf{x}_n; \mathbf{w}) - \mathrm{y}_n)^2 = \frac{1}{N}\sum_{n=1}^{N} (\mathbf{x}^L - y_n)^2$$

Backpropagation

- We minimize E_{in} to obtain the learned weights using SGD
- Backpropagation is a special algorithm that computes the gradient efficiently

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{in}(\mathbf{w}(t))$$

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathbf{e}_n = \frac{1}{N}\sum_{n=1}^{N} \mathbf{e}(h(\mathbf{x}_n), y_n)$$

$$\mathbf{e}(h, y) = (h - y)^2$$

We have to compute $\dfrac{\partial E_{\text{in}}}{\partial \mathrm{W}^{(l)}} = \dfrac{1}{N}\displaystyle\sum_{n=1}^{N} \dfrac{\partial \mathbf{e}_n}{\partial \mathrm{W}^{(l)}}$ $\qquad l = 0,1,\cdots,L$

Backpropagation is based on several applications of the chain rule to write partial derivatives in layer ℓ using partial derivatives in layer (ℓ + 1).

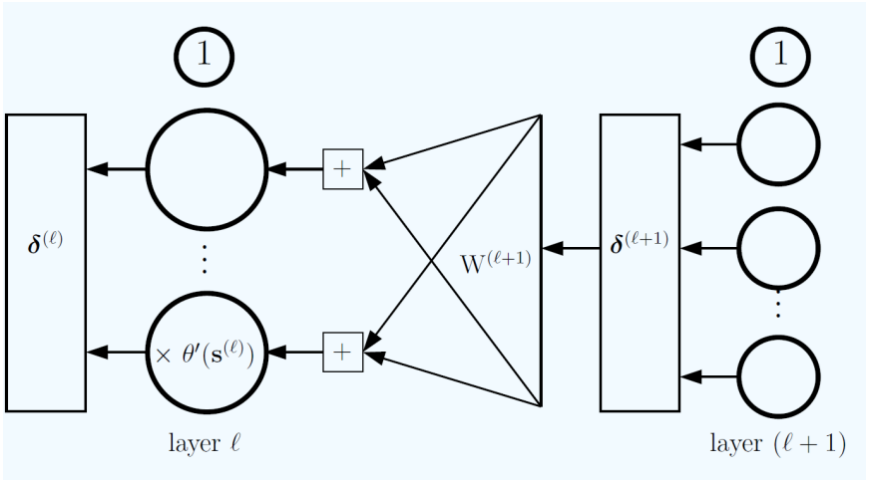
Backpropagation (cont´d)

$\boldsymbol{\delta}^{(l)} = \dfrac{\partial \mathbf{e}}{\partial \mathbf{s}^{(l)}}$: the *sensitivity (gradient)* of the error respect to the input signal to layer (l)

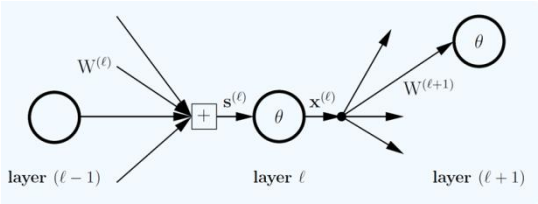
$$\frac{\partial \mathbf{e}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{e}}{\partial \mathbf{s}^{(l)}} \frac{\partial \mathbf{s}^{(l)}}{\partial \mathbf{W}^{(l)}} \qquad \mathbf{s}^{(l)} = \left(\mathbf{W}^{(l)}\right)^T \mathbf{x}^{(l-1)}$$

$$\boxed{\frac{\partial \mathbf{e}}{\partial \mathbf{W}^{(l)}} = \mathbf{x}^{(l-1)} \left[\boldsymbol{\delta}^{(l)}\right]^{\mathrm{T}}}$$

$$(d^{(l-1)}+1) \times d^l = \left[\left(d^{(l-1)}+1\right) \times 1\right][1 \times d^l]$$



$$\boxed{\frac{\partial \mathbf{e}}{\partial \mathbf{s}^{(l)}} = \frac{\partial \mathbf{e}}{\partial \mathbf{x}^{(l)}} \frac{\partial \mathbf{x}}{\partial \mathbf{s}^{(l)}}}$$



$$\boxed{\boldsymbol{\delta}^{(l)} = \frac{\partial \mathbf{e}}{\partial \mathbf{s}^{(l)}} = \theta'(\boldsymbol{s}^{(l)}) \odot \left[\left(\mathrm{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)}\right]_1^{d(l)}\right.}$$

$$\boxed{\frac{\partial \mathbf{x}}{\partial \mathbf{s}^{(l)}}} \qquad \boxed{\frac{\partial \mathbf{e}}{\partial \mathbf{x}^{(l)}} = \frac{\partial \mathbf{e}}{\partial \boldsymbol{s}^{(l+1)}} \frac{\partial \boldsymbol{s}^{(l+1)}}{\partial \mathbf{x}^{(l)}}}$$

$$\boldsymbol{\delta}^{(1)} \leftarrow \boldsymbol{\delta}^{(2)} \ldots \boldsymbol{\delta}^{(L-1)} \leftarrow \boldsymbol{\delta}^{(L)}$$

$\odot$ - element-wise product

Backpropagation (cont´d)

Activation Layer: tanh

$$\theta'(\boldsymbol{s}^{(l)}) = \tanh'(\boldsymbol{s}^{(l)}) = \mathbf{1} - \tanh(\boldsymbol{s}^{(l)})^{2} = \mathbf{1} - \mathbf{x} \odot \mathbf{x}$$

$\odot$ - element-wise product

Let's compute the layer *L*:

$$\boldsymbol{\delta}^{L} = \frac{\partial \boldsymbol{e}}{\partial \boldsymbol{s}^{L}} = \frac{\partial}{\partial \boldsymbol{s}^{L}}(\boldsymbol{x}^{L} - y)^{2} = 2(\boldsymbol{x}^{L} - y)\frac{\partial \boldsymbol{x}^{L}}{\partial \boldsymbol{s}^{L}}$$
$$= 2(\boldsymbol{x}^{L} - y)\theta'(\boldsymbol{s}^{(L)})$$

$$\boldsymbol{x}^{(l)} = \begin{pmatrix} 1 \\ \theta(\boldsymbol{s}^{(l)}) \end{pmatrix}$$

Backpropagation (cont´d)

Backpropagation to compute sensitivities $\boldsymbol{\delta}^{(\ell)}$.
Input: a data point $(\mathbf{x}, y)$.

0: Run forward propagation on $\mathbf{x}$ to compute and save:

$$\begin{array}{ll} \mathbf{s}^{(\ell)} & \text{for } \ell = 1, \ldots, L; \\ \mathbf{x}^{(\ell)} & \text{for } \ell = 0, \ldots, L. \end{array}$$

1: $\delta^{(L)} \leftarrow 2(x^{(L)} - y)\theta'(s^{(L)})$ **[Initialization]**

$$\theta'(s^{(L)}) = \begin{cases} 1 - (x^{(L)})^2 & \theta(s) = \tanh(s); \\ 1 & \theta(s) = s. \end{cases}$$

2: **for** $\ell = L - 1$ to 1 **do** **[Back-Propagation]**

3: Let $\theta'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \odot \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$.

4: Compute the sensitivity $\boldsymbol{\delta}^{(\ell)}$ from $\boldsymbol{\delta}^{(\ell+1)}$:

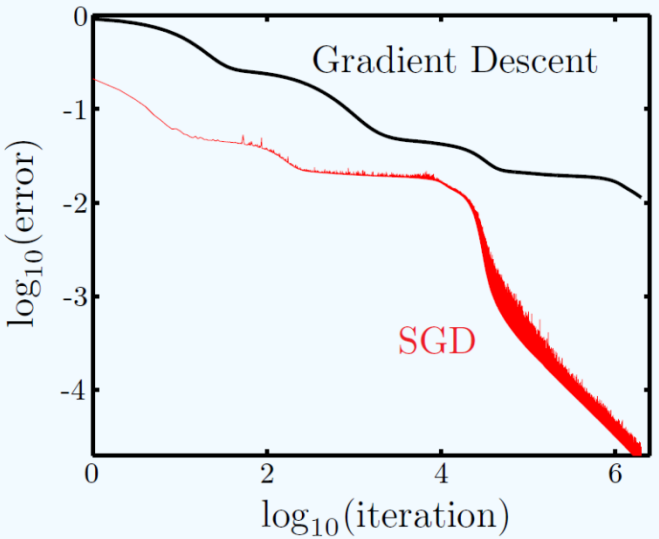
$$\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \odot \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$$

Backpropagation (cont´d)

Algorithm to Compute $E_{\text{in}}(\mathbf{w})$ and $\mathbf{g} = \nabla E_{\text{in}}(\mathbf{w})$.
Input: $\mathbf{w} = \{\mathrm{W}^{(1)}, \ldots, \mathrm{W}^{(L)}\}$; $\mathcal{D} = (\mathbf{x}_1, y_1) \ldots (\mathbf{x}_N, y_n)$.
Output: error $E_{\text{in}}(\mathbf{w})$ and gradient $\mathbf{g} = \{\mathrm{G}^{(1)}, \ldots, \mathrm{G}^{(L)}\}$.

1: Initialize: $E_{\text{in}} = 0$ and $\mathrm{G}^{(\ell)} = 0 \cdot \mathrm{W}^{(\ell)}$ for $\ell = 1, \ldots, L$.
2: **for** Each data point $(\mathbf{x}_n, y_n)$, $n = 1, \ldots, N$, **do**
3: Compute $\mathbf{x}^{(\ell)}$ for $\ell = 0, \ldots, L$. [forward propagation]
4: Compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = L, \ldots, 1$. [backpropagation]
5: $E_{\text{in}} \leftarrow E_{\text{in}} + \frac{1}{N}(\mathbf{x}^{(L)} - y_n)^2$.
6: **for** $\ell = 1, \ldots, L$ **do**
7: $\mathrm{G}^{(\ell)}(\mathbf{x}_n) = [\mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}]$
8: $\mathrm{G}^{(\ell)} \leftarrow \mathrm{G}^{(\ell)} + \frac{1}{N}\mathrm{G}^{(\ell)}(\mathbf{x}_n)$

- This scheme shows the batch gradient descent version of the algorithm. (NOT RECOMMENDED)
- SGD must be used in order to update the gradients more frequently and improve the fitting error



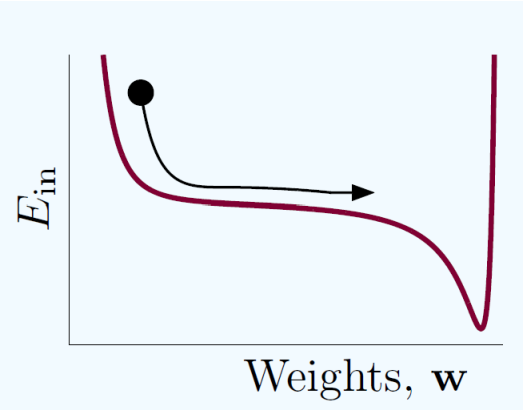
- A comparison of batch gradient descent with SGD is shown to the right.
- 500 training examples from the digits data and a 2-layer neural network with 5 hidden units and learning rate η = 0.01.
- The erratic behavior of SGD is mainly due to the use of a fix value of learning rate

Initialization and termination

- Two very important steps in learning NN models are:
  - Weight initialization
  - Termination criteria

- Weight initialization:
  - **Never initialize to zero or all weights the same value** (NO motion to the local optimum)
  - **Never initialize with large values** (large value saturates the sigmoidal fucntion and the gradiente propagation goes to zero)
  - Initialize with small random values: i.e $\text{Gausian}(0, \sigma_\varepsilon^2), \quad \sigma_\varepsilon^2 \cdot \max_{\mathbf{x}_i} \|\mathbf{x}_i\|^2 \ll 1$
  - For classification a very good initialization can be done using the weights after fitting a regression model (the same as with linear models)

- Termination criteria:
  - Number of iterations
  - Size of the gradient
  - The value E_{in}
  - The best is a combination of criteria



Regularization & Validation

- Multilayer neural networks fitted with SGD is prone to overfitting

- Weight-decay regularization:

$$E_{\text{aug}} = E_{\text{in}} + \frac{\lambda}{N}\sum_{l,i,j}\left(w_{ij}^{(l)}\right)^2 \qquad \frac{\partial E_{\text{aug}}(\mathbf{w})}{\partial \mathrm{W}^{(l)}} = \frac{\partial E_{\text{in}}(\mathbf{w})}{\partial \mathrm{W}^{(l)}} + \frac{2\lambda}{N}\mathrm{W}^{(l)}$$

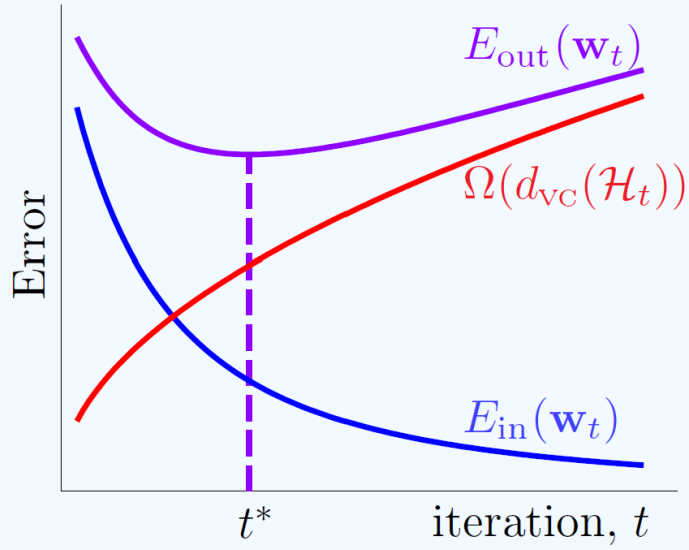
- Weight elimination regularization:

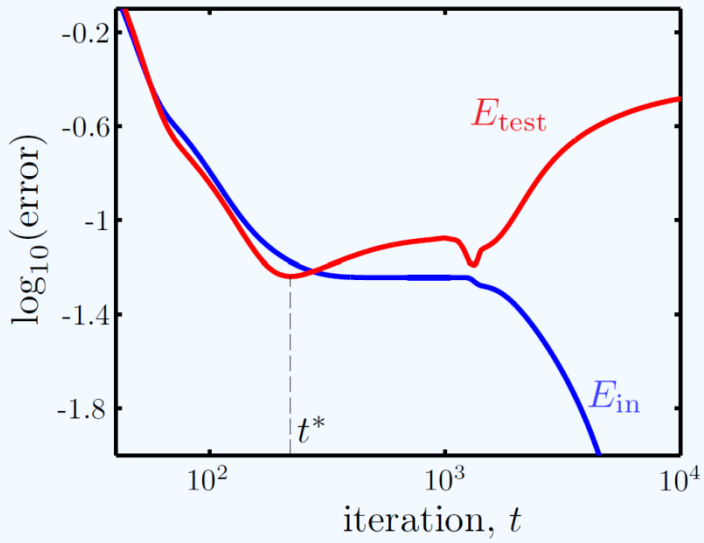
$$E_{\text{aug}} = E_{\text{in}} + \frac{\lambda}{N}\sum_{l,i,j}\frac{(w_{ij}^{(l)})^2}{1+(w_{ij}^{(l)})^2}$$

What is the difference with weight-decay ?

Early stopping regularization

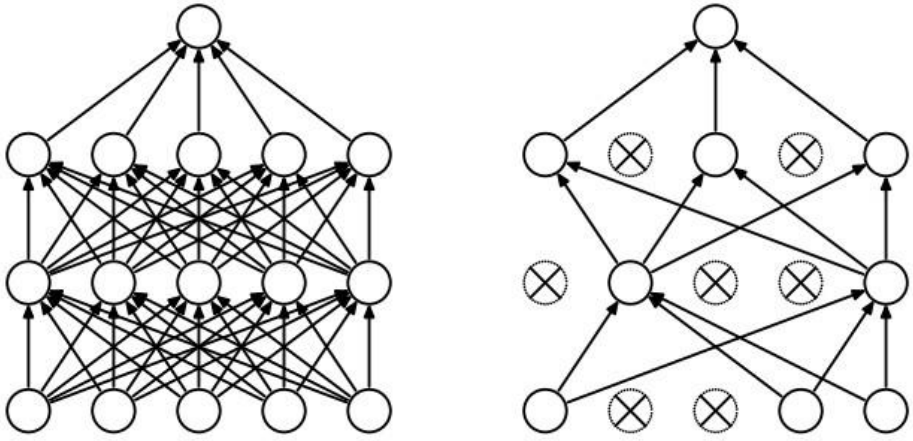
- An iterative method such as gradient descent does not explore your full hypothesis set all at once. With more iterations, more of your hypothesis set is explored.
- This means that by using fewer iterations, you explore a smaller hypothesis set and should get better generalization
- Early stopping implements weight-decay since it move to solution near to the initial values.
- Early stopping estimates a final model using verification.
- Unfortunately, adding back the validation data and training for t* iterations can lead to a completely different set of weights.
- But 'early stopping' by mistake can appear due to flat zones in the error function (see blue curve in figure)





'1' versus all other digits, with 70 randomly selected data points and a small sigmoidal neural network with a single hidden unit and tanh(·) output node.

Dropout Regularization



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

Regularization: Dropout (Hinton y col 2014)

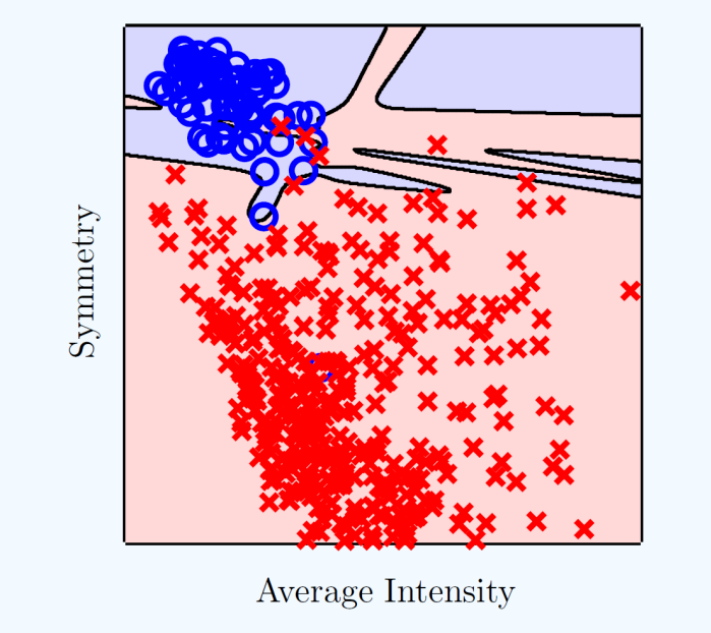
Dropout **prevents overfitting due to complex co-adaptations of feature detectors**

While training, dropout is implemented by only keeping a neuron active with some probability **p** (a hyperparameter), or setting it to zero otherwise.

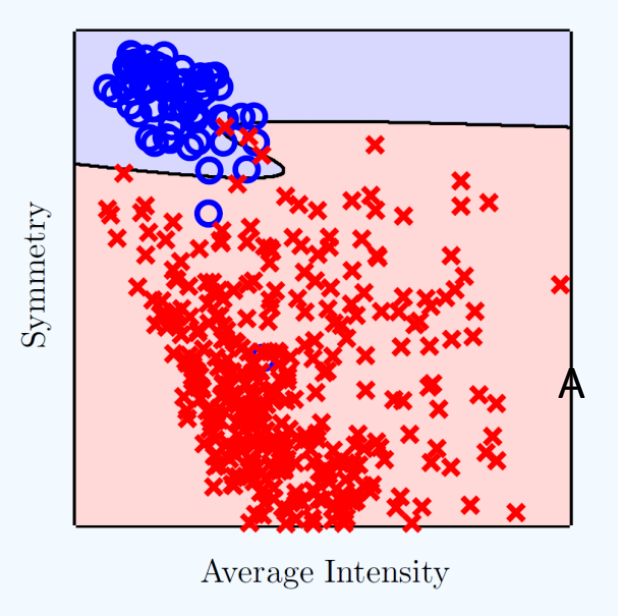
During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks

There is also the possibility of applying Dropout in Training and Test but paying the a computational cost

Examples: digit data

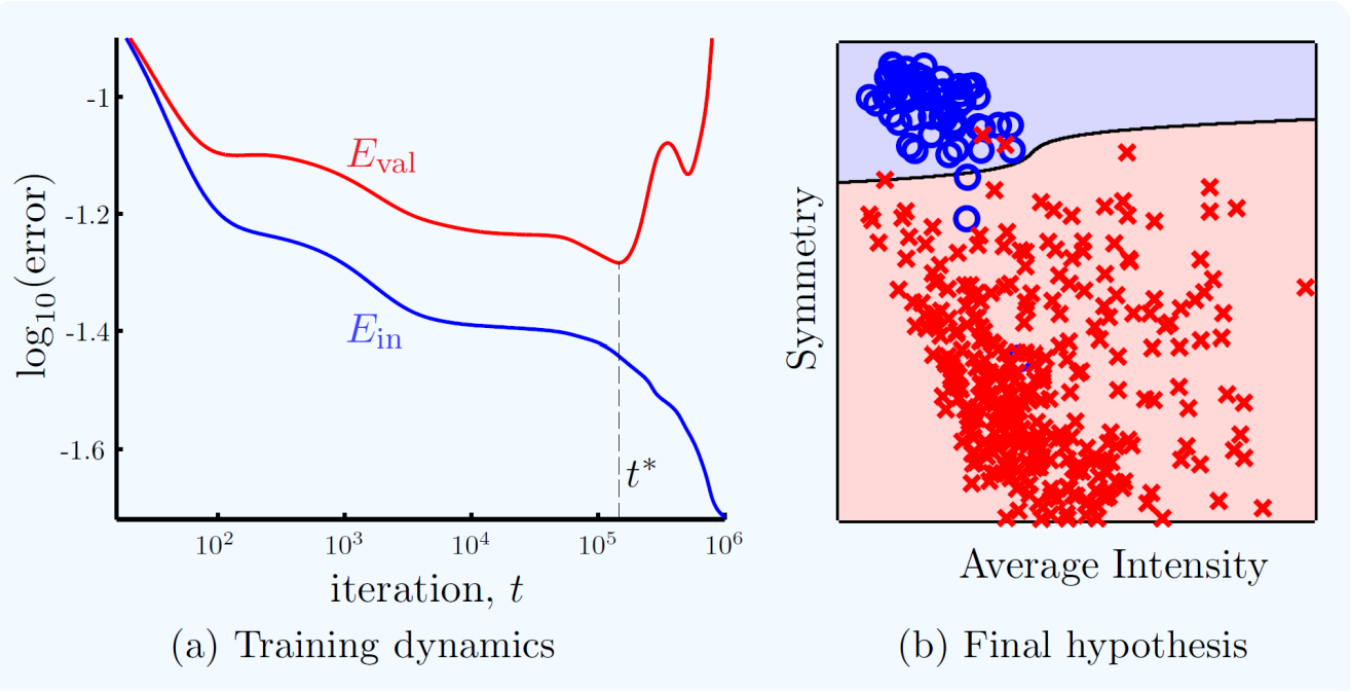


10 hidden unit neural network trained with gradient descent on 500 examples from the digits data (no regularization). Blue circles are the digit '1' and the red x's are the other digit



Regularized weight-decay solution λ=0.01

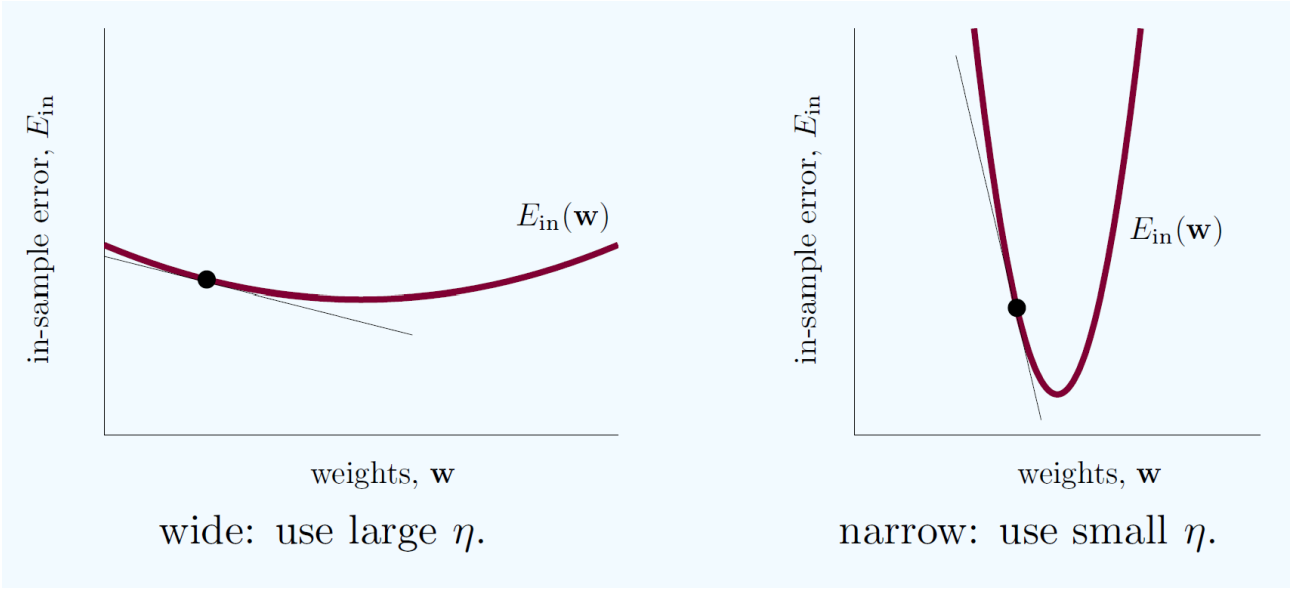
Examples: digit data (cont'd)



Early stopping with a validation set of size 50 (one-tenth of the data); so the training set will now have size 450

	E_{train}	E_{val}	E_{in}	E_{out}
No Regularization	–	–	0.2%	3.1%
Weight Decay	–	–	1.0%	2.1%
Early Stopping	1.1%	2.0%	1.2%	2.0%

Adapting the learning rate



Goal: To obtain the best possible optimun in a reasonable time

Variable Learning Rate Gradient Descent:

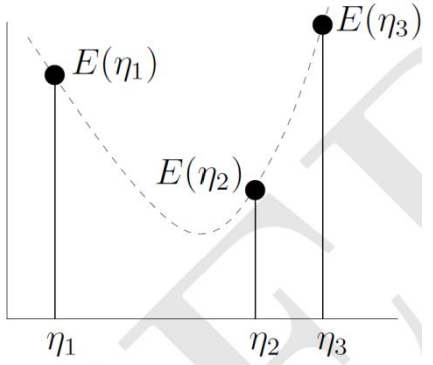
1: Initialize $\mathbf{w}(0)$, and η_0 at $t = 0$. Set $\alpha > 1$ and $\beta < 1$.
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.
4: **if** $E_{\text{in}}(\mathbf{w}(t) + \eta_t \mathbf{v}(t)) < E_{\text{in}}(\mathbf{w}(t))$ **then**
5: accept: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta_t \mathbf{v}(t)$; $\eta_{t+1} = \alpha \eta_t$.
6: **else**
7: reject: $\mathbf{w}(t+1) = \mathbf{w}(t)$; $\eta_{t+1} = \beta \eta_t$.
8: Iterate to the next step, $t \leftarrow t + 1$.

$\alpha \approx 1.05 - 1.1$
$\beta \approx 0.5 - 0.8$

Adapting the learning rate (cont'd)

Steepest Descent (Gradient Descent + Line Search):

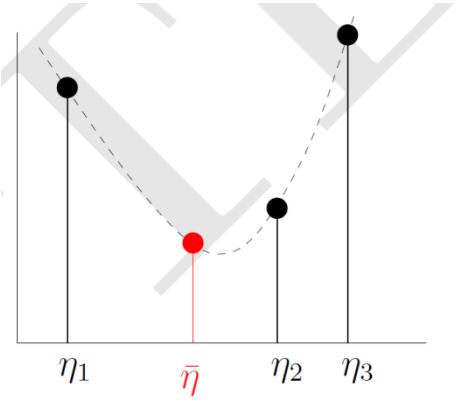
1: Initialize $\mathbf{w}(0)$ and set $t = 0$;
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.
4: Let $\eta^* = \text{argmin}_\eta E_{\text{in}}(\mathbf{w}(t) + \eta \mathbf{v}(t))$.
5: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^* \mathbf{v}(t)$.
6: Iterate to the next step, $t \leftarrow t + 1$.



We have to solve a new optimization problem to obtain η*

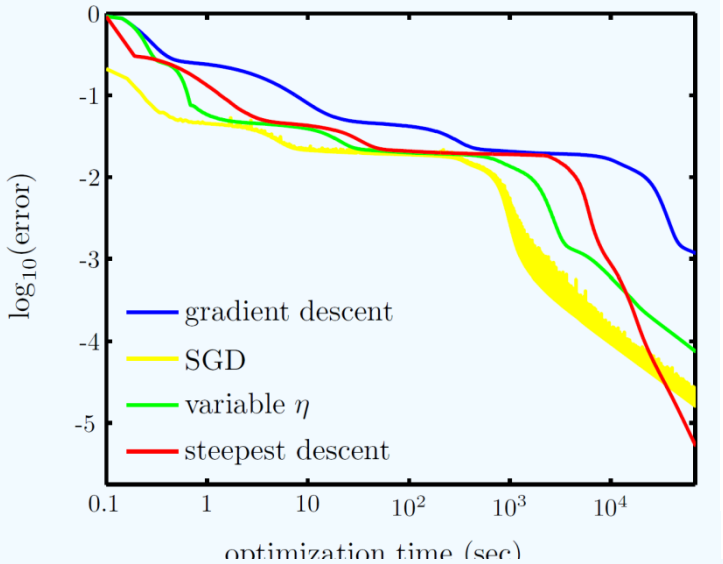
$E(\eta_2) < \min\{E(\eta_1), E(\eta_3)\}$.

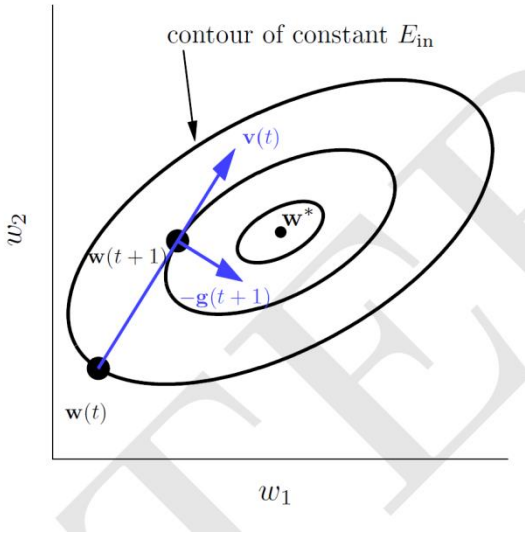
Method	Optimization Time		
	10 sec	1,000 sec	50,000 sec
Gradient Descent	0.079	0.0206	0.00144
Stochastic Gradient Descent	**0.0213**	**0.00278**	0.000022
Variable Learning Rate	0.039	0.014	0.00010
Steepest Descent	0.043	0.0189	**0.000012**

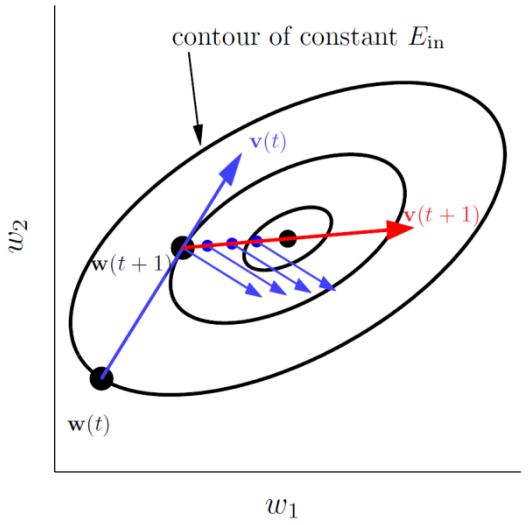


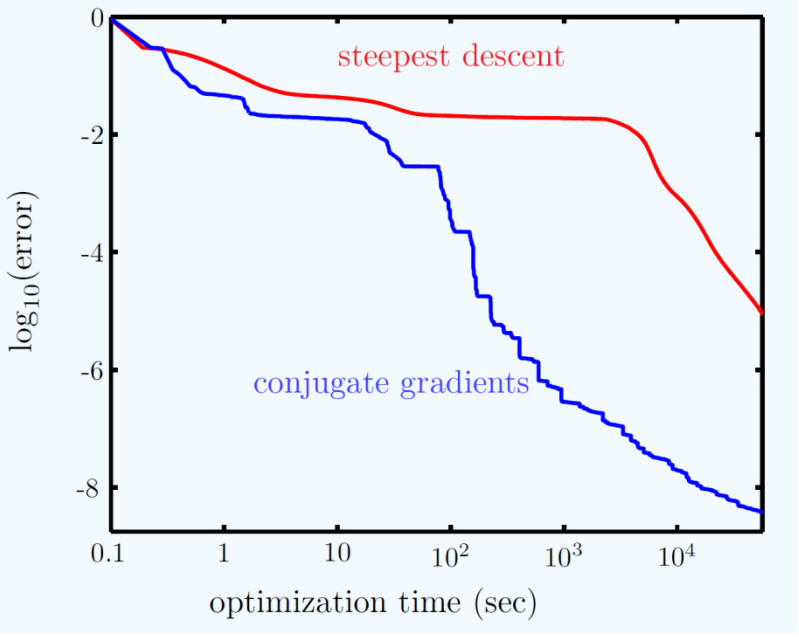
$$\bar{\eta} = \frac{1}{2}(\eta_1 + \eta_3)$$

Adapting the learning rate (cont'd)









Conjugate Gradient Descent:

1: Initialize $\mathbf{w}(0)$ and set $t = 0$; set $\mathbf{v}(-1) = \mathbf{0}$
2: **while** stopping criterion has not been met **do**
3: Let $\mathbf{v}(t) = -\mathbf{g}(t) + \mu_t \mathbf{v}(t-1)$, where

$$\mu_t = \frac{\mathbf{g}(t+1)^{\mathrm{T}}(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{g}(t)^{\mathrm{T}}\mathbf{g}(t)}.$$

4: Let $\eta^* = \operatorname{argmin}_\eta E_{\text{in}}(\mathbf{w}(t) + \eta \mathbf{v}(t))$.
5: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^* \mathbf{v}(t)$;
6: Iterate to the next step, $t \leftarrow t+1$;

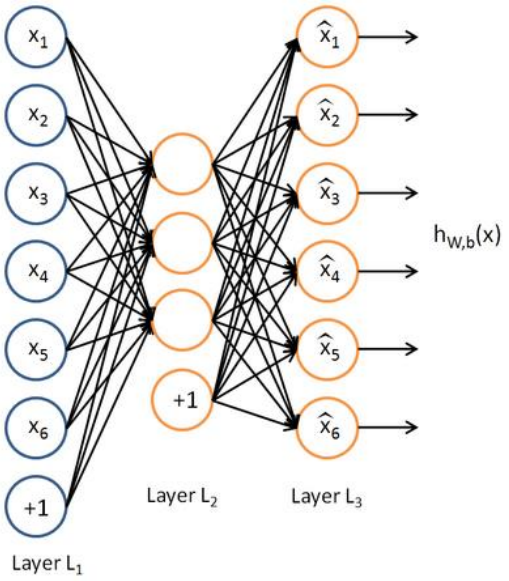
Deep Learning motivation

- Universal Approximation Result: "A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. "

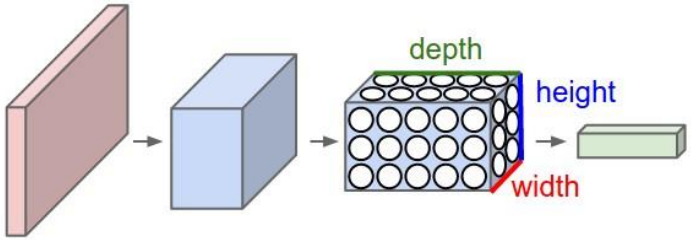
- Empirical findings:
  - In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.
  - There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value d, but which require a much larger model if depth is restricted to be less than or equal to d.
  - In many cases, the number of hidden units required by the shallow model is exponential in input dimension.
  - Greater depth does seem to result in better generalization for a wide variety of tasks
  - Many AI problems are today identified to function needing deep architectures to be learned (Y. Bengio, 2009)

Practical FeedForward Deep Networks

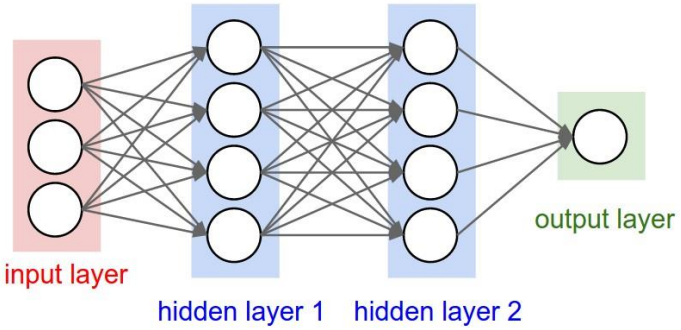
- Autoencoder: (Lecun 1987)
  - Full connected models
  - Defines a mappings bettween two different representation spaces
  - Unsupervised models (mostly)
  - Two stages : encoder and decoder
  - Connections with the Generative Learning Model



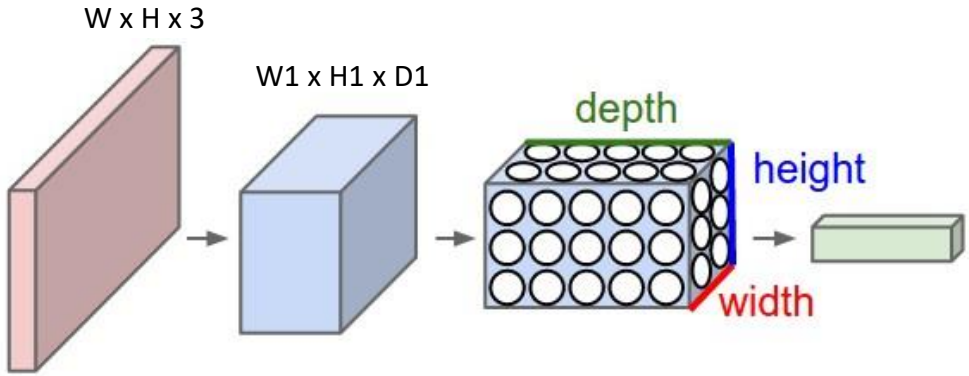
- **Convolutional Networks:** (Lecun 1998 and others)
  - Use prior information
    - Spatial layout of the signal: local dependences
    - Weights are spatially stationary
  - Sharing parameters, sparse connections
  - Feature extraction architecture
  - Composition of different types of layers



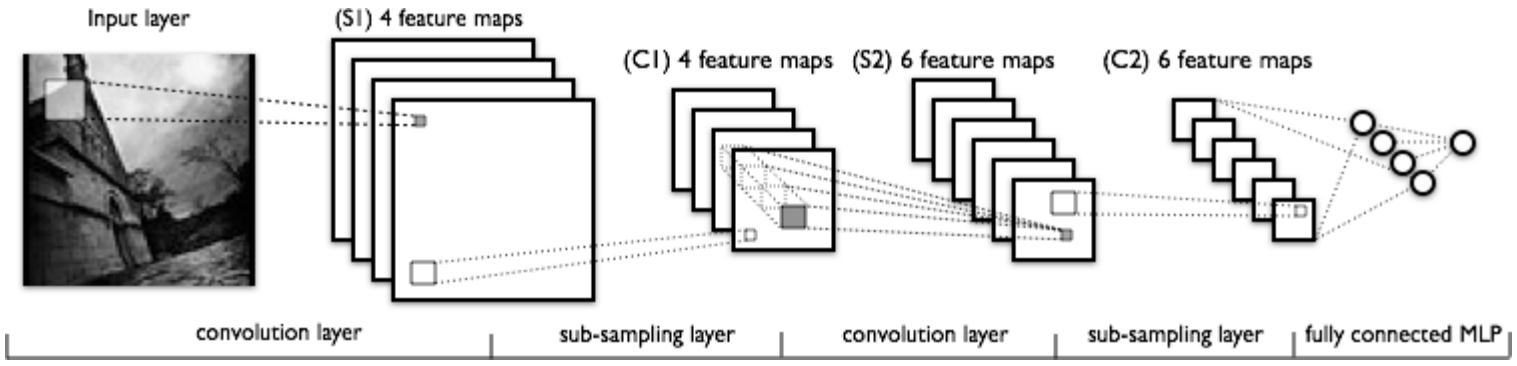
Convolutional Neural Network (CNN)



3-layers standard dense FF-NN



3 layers CNN with 3D blocks of neurons



Simple example of CNN

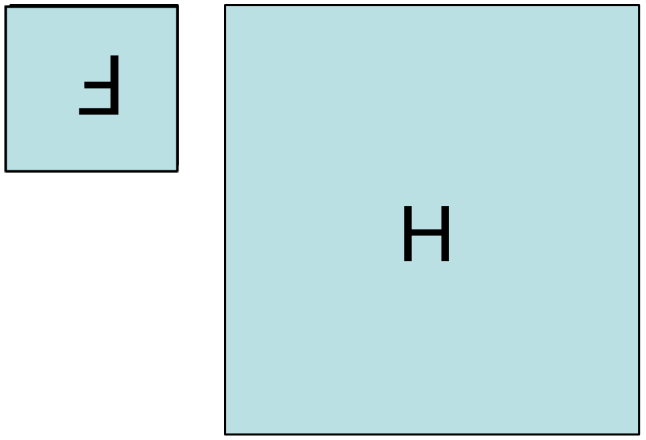
Convolution

- Convolution:
  - Flip the filter in both dimensions (bottom to top, right to left)
  - Then apply cross-correlation

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u,j-v]$$

$$G = H \star F$$

Notation for convolution operator



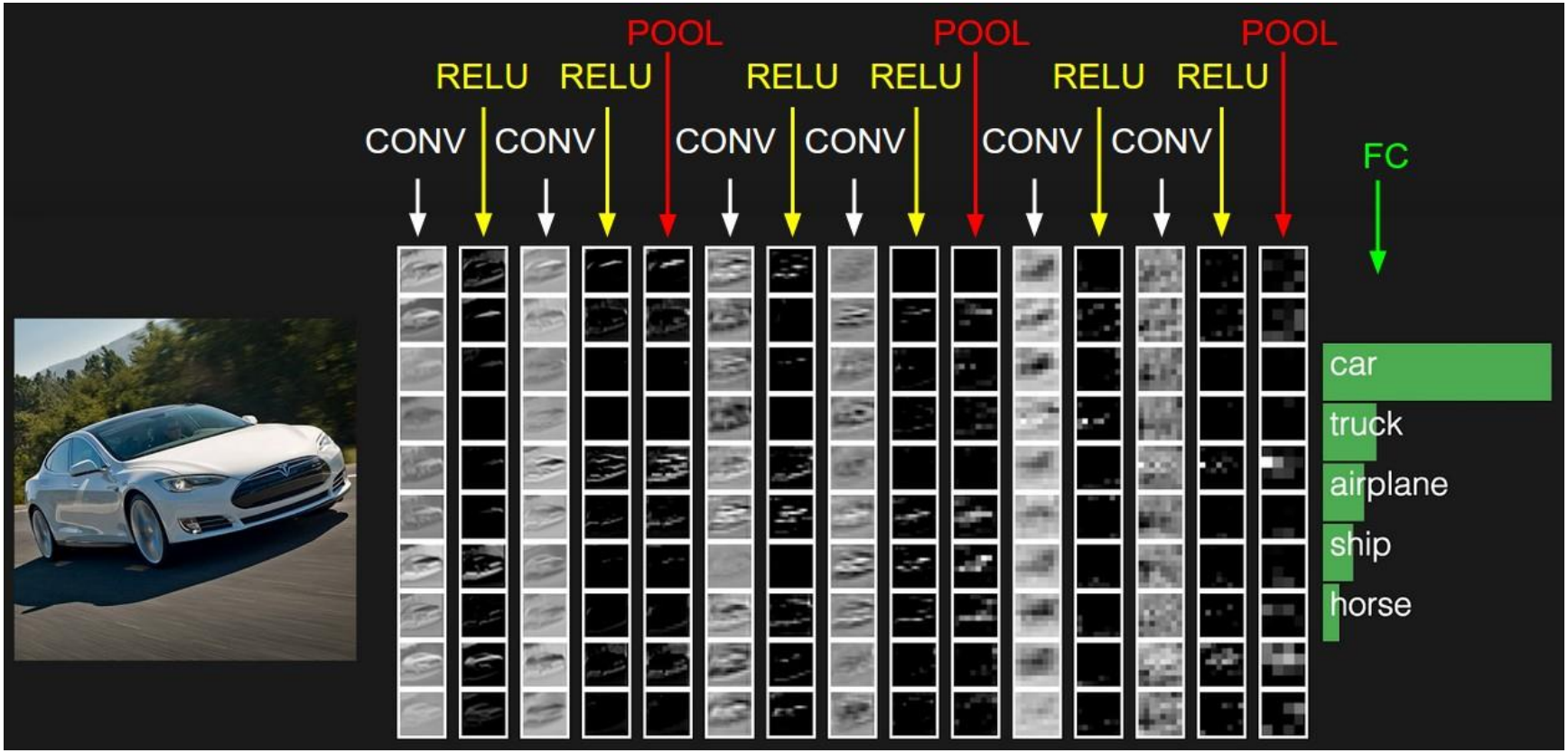
Layers of the CNN architecture

- Type of layers:
  - Convolutional (Conv)
  - Pooling (POOL)
  - Full Connected (FC)
  - RELU: max(0,x)
  - Some others : normalization, regularization, etc

- **Conv** and **FC** performs transformations that are a function of not only the activations in the input volume, but also of the parameters.
- **POOL/RELU** layers will implement a fixed function
- **Others:**

- The parameters in the CONV/FC layers will be trained with gradient descent (SGD) using backpropagation.

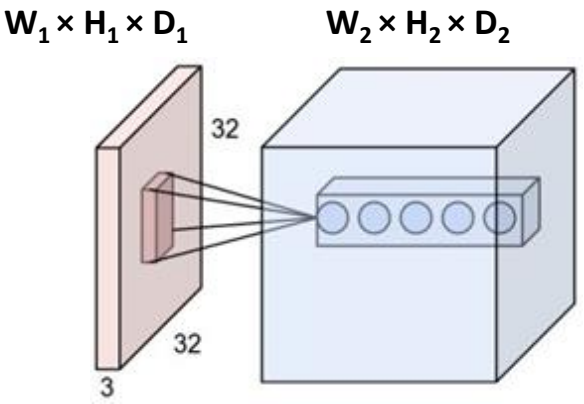
Designing a CNN (Stanford)



The activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one. The full web-based demo is shown in the header of http://cs231n.stanford.edu/. The architecture shown here is a tiny VGG Net, which we will discuss later.

Convolutional Layer

- Prior information: Local dependence
- Brain architecture: Local connectivity



Spatial arrangement:

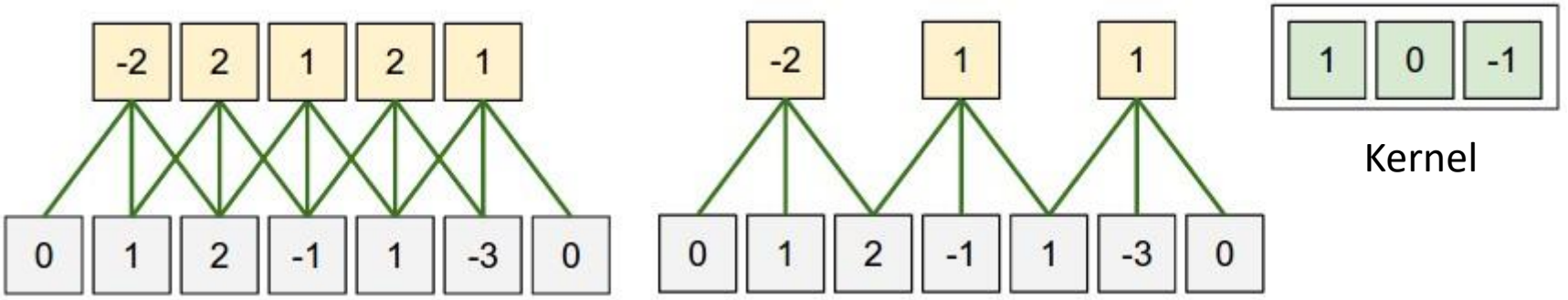
- Parameters used to generate the next block of neurons
  - **Receptive Field:** spatial region of the input (**F**)
  - **Depth**: Number of output layers (**D**)
  - **Stride**: mechanism to reduce the spatial dimension (**S**)
  - **Zero-Padding**: mechanism to effectively compute a fixed output spatial dimension (**P**)

Parameters sharing:

- this scheme is used in Convolutional Layers to control the number of parameters
  - Each output unit is assigned with the convolution value of a 3D kernel and a local region of the input block
  - Parameters sharing refers to that all the local regions share the same kernel (parameters)

Convolutional Layer: Stride

- The spatial size of the output volume as a function of **W**, **F**, **S**, **P** .
- The correct formula given by (**W**−**F**+**2P**)/**S**+1.
- For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output.



In this example there is only one spatial dimension (x-axis), one neuron with **F** = 3, **W** = 5, and **P** = 1.

- **Left**: The neuron strided across the input in stride of **S** = 1, giving output of size (5 - 3 + 2)/1+1 = 5.
- **Right**: The neuron uses stride of **S** = 2, giving output of size (5 - 3 + 2)/2+1 = 3.
- Notice that stride **S** = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5 - 3 + 2) = 4 is not divisible by 3.

Summarizing the Conv Layer

Summary.

- Accepts a volume of size $\mathbf{W_1 \times H_1 \times D_1}$
- Requires four hyperparameters:
  - Number of filters **K**, (depth)
  - their spatial extent **F**, (kernel size)
  - the stride **S**, (skip)
  - the amount of zero-padding **P**

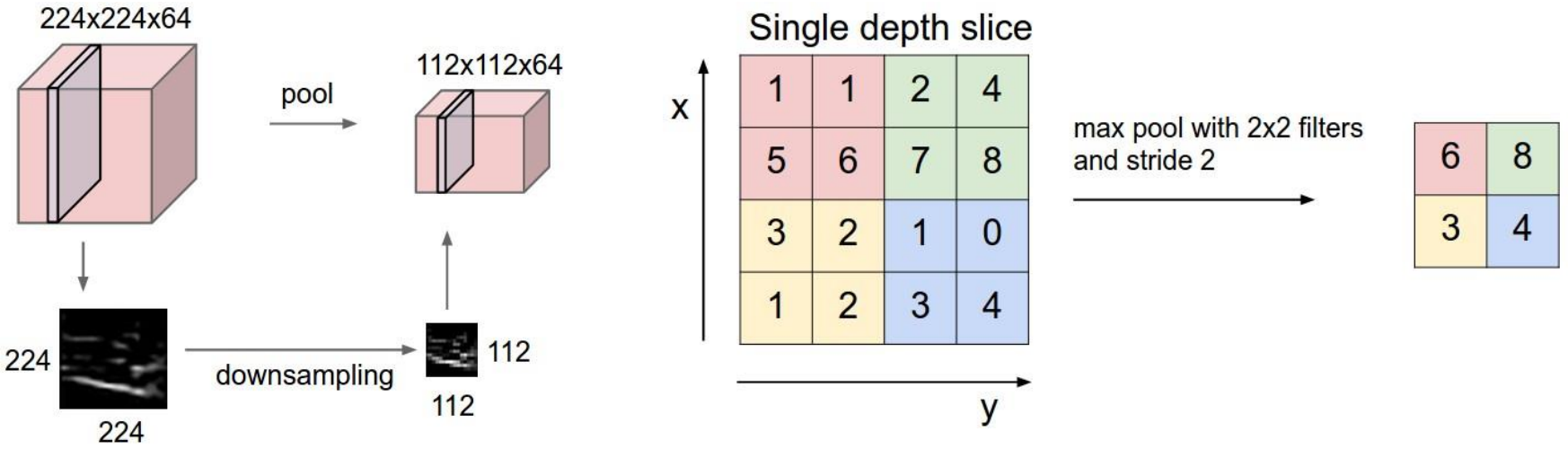
- Produces a volume of size $\mathbf{W_2 \times H_2 \times D_2}$ where:
  - $\mathbf{W_2} = (\mathbf{W_1} - \mathbf{F} + 2\mathbf{P})/\mathbf{S} + \mathbf{1}$
  - $\mathbf{H_2} = (\mathbf{H_1} - \mathbf{F} + 2\mathbf{P})/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $\mathbf{D_2} = \mathbf{K}$

- With parameter sharing, it introduces **F** x **F** x $\mathbf{D_1}$ weights per filter, for a total of (**F** x **F** x $\mathbf{D_1}$) x **K** weights and **K** biases.

- In the output volume, the **d**-th depth slice (of size $\mathbf{W_2 \times H_2}$) is the result of performing a valid convolution of the **d**-th filter over the input volume with a stride of **S**, and then offset by **d**-th bias.

- A common setting of the hyper-parameters is **F** = 3, **S** = 1, **P** = 1. However, there are common conventions and rules of thumb that motivate these hyperparameters.

Pooling layer



Common strategies: **F**={ 2, 3 } , **S** = 2, { Max-Pooling or Average }

Pooling provides spatial invariance

Nevertheless: The pooling layer is an open discussion

Summazing the Pooling Layer

- Accepts a volume of size $\mathbf{W_1 \times H_1 \times D_1}$
- Requires two hyperparameters:
  - their spatial extent **F**,
  - the stride **S**,
- Produces a volume of size $\mathbf{W_2 \times H_2 \times D_2}$ where:
  - $\mathbf{W_2 = (W_1 - F)/S + 1}$
  - $\mathbf{H_2 = (H_1 - F)/S + 1}$
  - $\mathbf{D_2 = D_1}$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

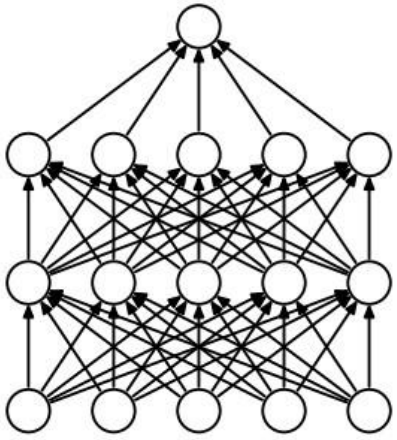
Regularization & Normalization

Regularization: Dropout (Hinton y col 2014)

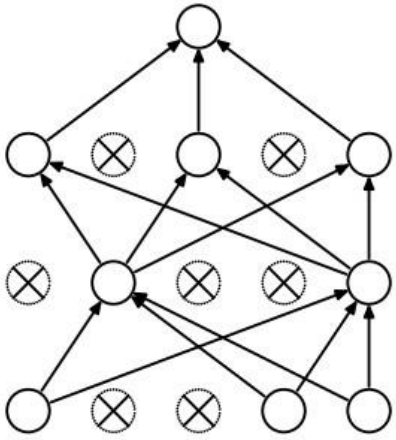
While training, dropout is implemented by only keeping a neuron active with some probability **p** (a hyperparameter), or setting it to zero otherwise.

During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks

In CNN is only applied to Full Connected layers



(a) Standard Neural Net

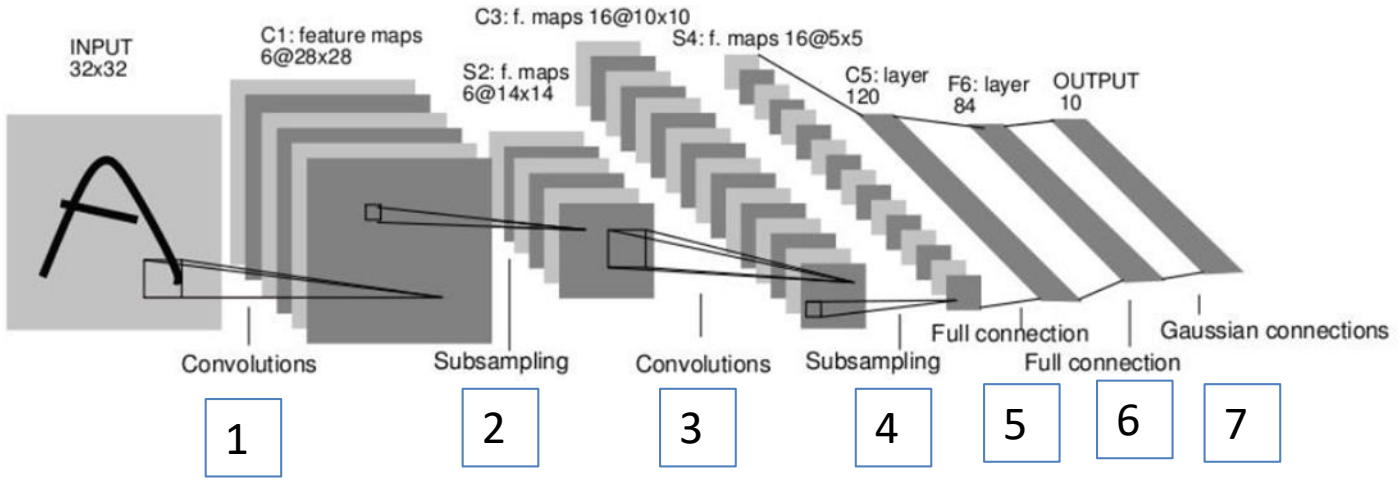


(b) After applying dropout.

Normalization;

- Many types of normalization layers have been proposed for use in ConvNet architectures.
- Biological inspiration:
  - Local response normalization layer (same map) : mean/sum of squared
  - Local response normalization layer (across map)
  - Local contrast normalization layer: mean/variance
- **Batch normalization layer:** VERY RELEVANT lo learn very deep network
  - Drawback: Introduce 2 new parameters per batch
- Normalization is an open issue …..

LENET



Feature extraction phase

1.- 6 filters 5x5 → 6@28x28
2.- Pooling 2x2 → 6@14x14
3.-16 filters 5x5 → 16@10x10
4.- Pooling 2x2 → 6@5x5
5.- Full connection 400→120
6.-Full connection 120→84

Parameters: (6+16)x5x5+
400x120+ 120x84+84x10

7.- Output layer: softmax 10 classes (84→ 10)

Message to take home:

- A CNN architecture fixes a class of functions
- The weights only choose one of the functions
- We use prior information on the functions

CNN is a smart way of defining a class of complex functions with very few parameters

Evolution of the Convolutional Networks Architectures

- **Lenet** : Y. Lecun (2001). Handwritten digits recognition

ILSVRC Challenge : 1.3 M images 1000 object

- **AlexNet**: (2012) First model showing tha potential of the convolutional nertworks (Alex Krizhevsky, Ilya Sutskever and Geoff Hinton)
- **ZF Net** (2013): Zeiler & Fergus Net , improve the AlexNet
- **GoogLeNet**: (2014,-) Introduce the inception module that reduces dramatically the number of parameters (from 60M (Alexnet) to 4M)
- **VGGNet** : (2014) Karen Simonyan and Andrew Zisserman. It show the importance of depth in object recognition. Only use 3x3 Conv Layers.
- **ResNet:** (2015) Kaiming He et al. It features special skip connections and a heavy use of batch normalization. It allows to learn very deep architectures.

Alexnet architecture

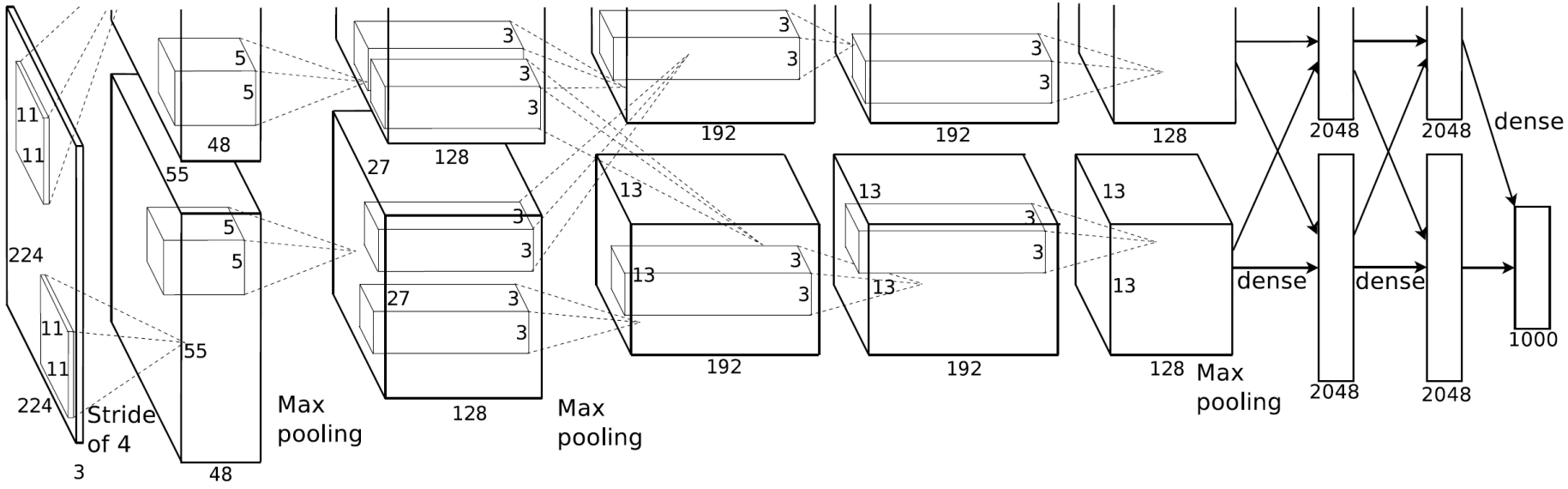


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Alexnet Results

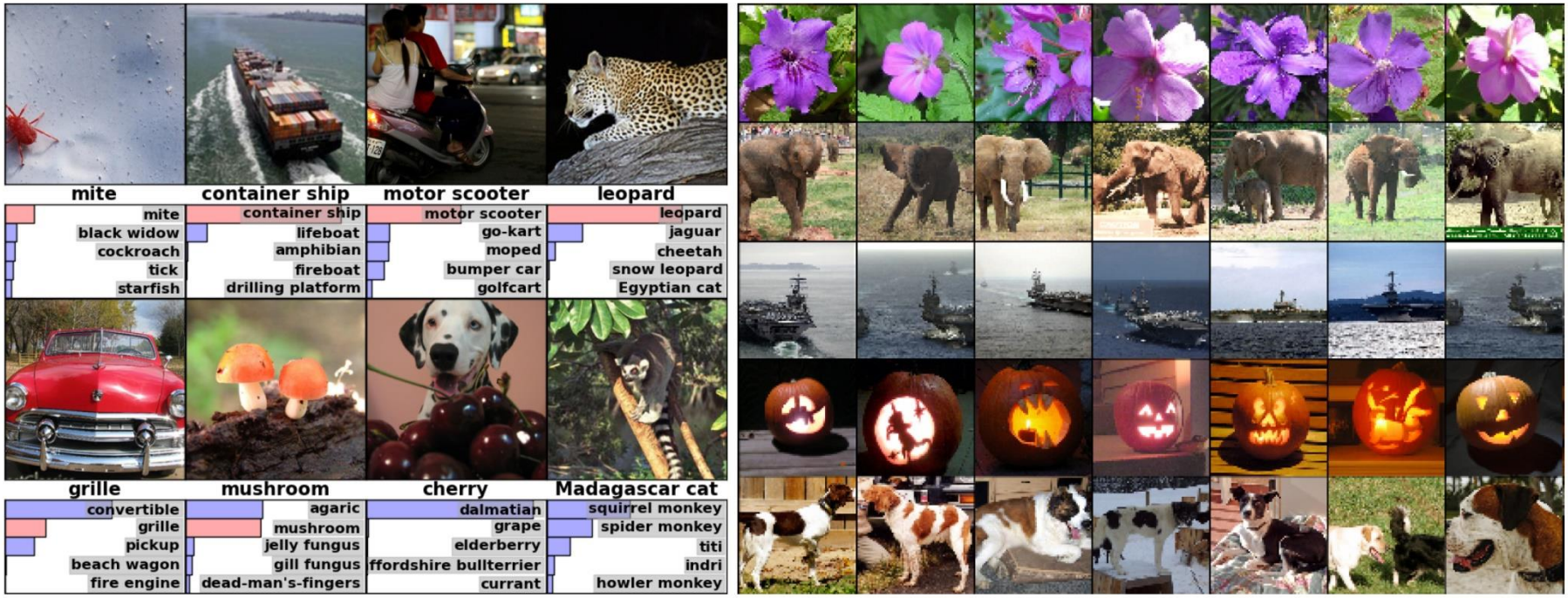
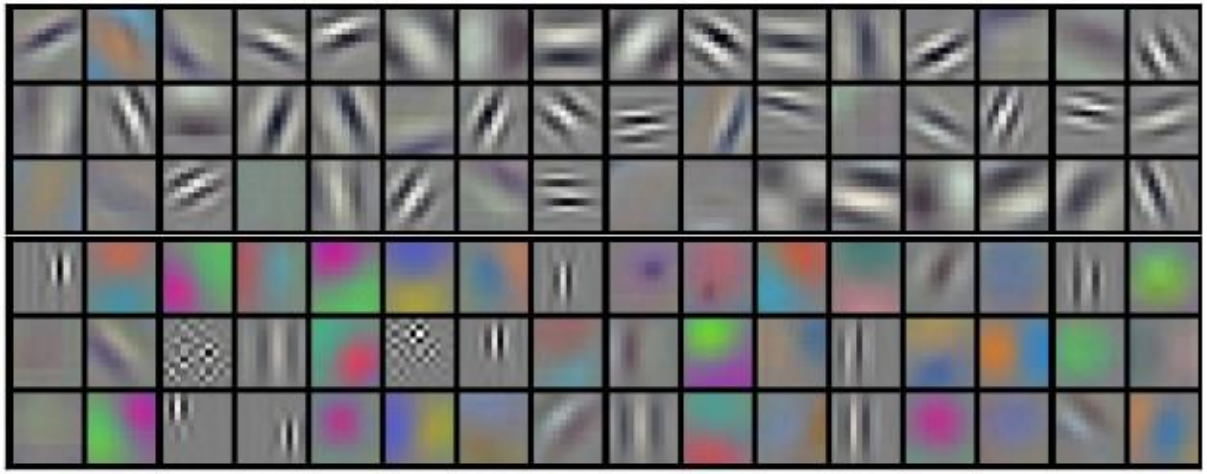


Figure 4: **(Left)** Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5). **(Right)** Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

Learned Weights

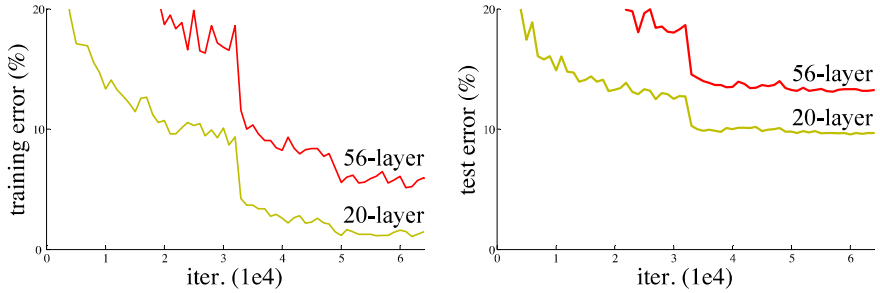


Example of filters learned by an AlexNet model in the first layer. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. (trained using ImageNet 2012: 1.3M of images spread over 1000 different classes)

Residual Networks

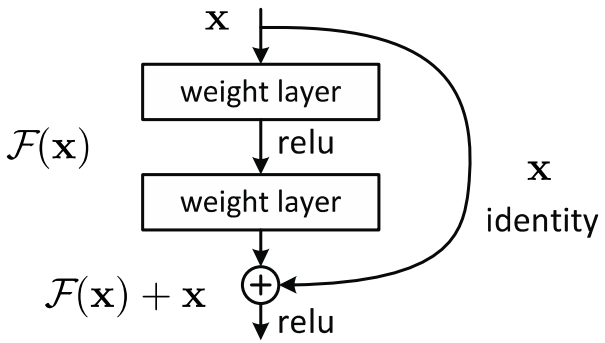
[He, Zhang, Ren, Sun, CVPR 2016]

Really, really deep convnets don't train well,
E.g. CIFAR10:



Key idea: introduce "pass through" into each layer

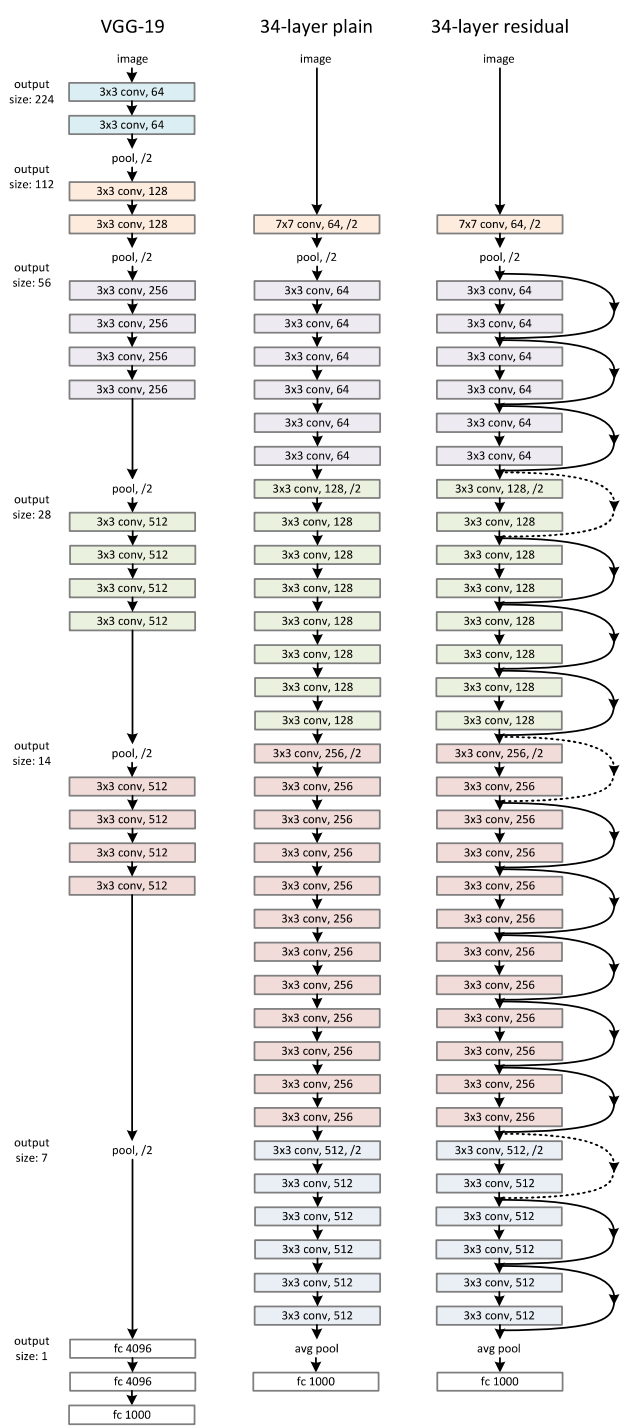
Thus only residual now needs to be learned



method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

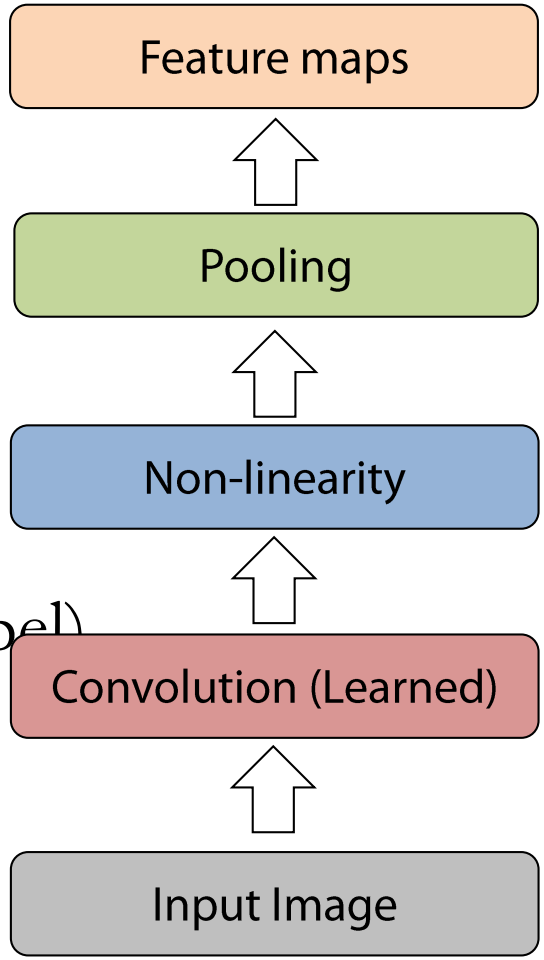
Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except † reported on the test set).

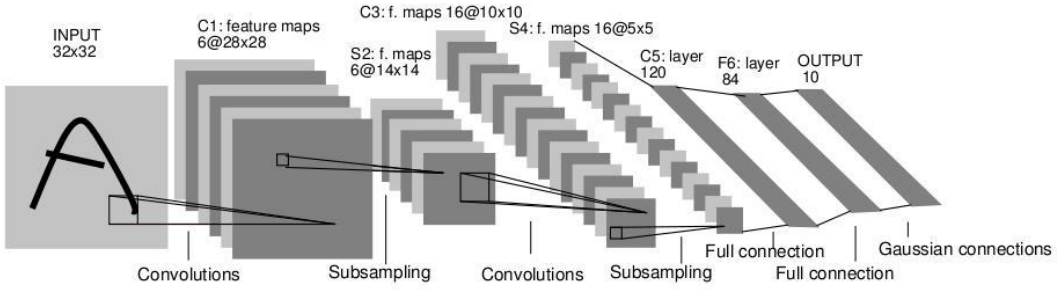
With ensembling, 3.57% top-5 test error on ImageNet



Summary of Convnet Model

- Feed-forward:
  - Convolve input
  - Non-linearity (rectified linear)
  - [Optional] Pooling (local max)
  - [Optional] Batch Normalization
- Fully-connected classifier layer at top
- Supervised loss function (uses image label)
- Train convolutional filters by back-propagating classification error





LeCun et al. 1998