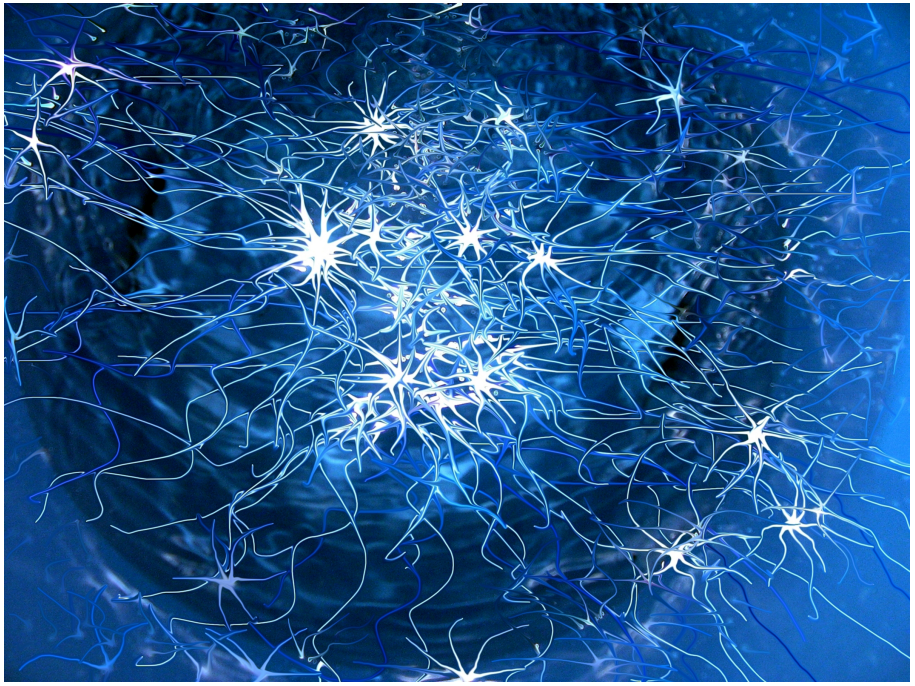
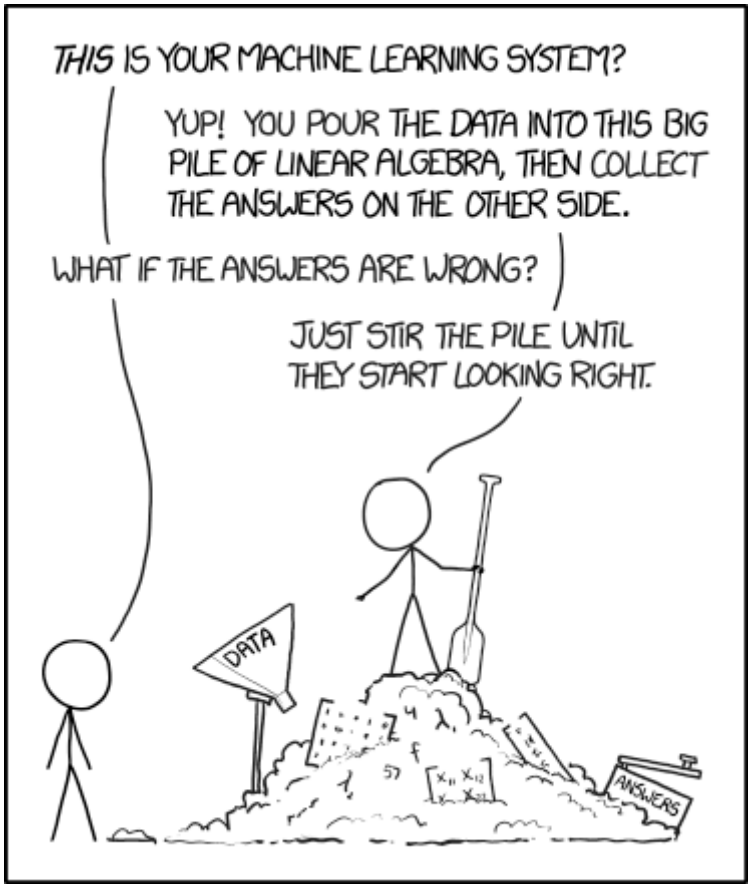
Neural Networks





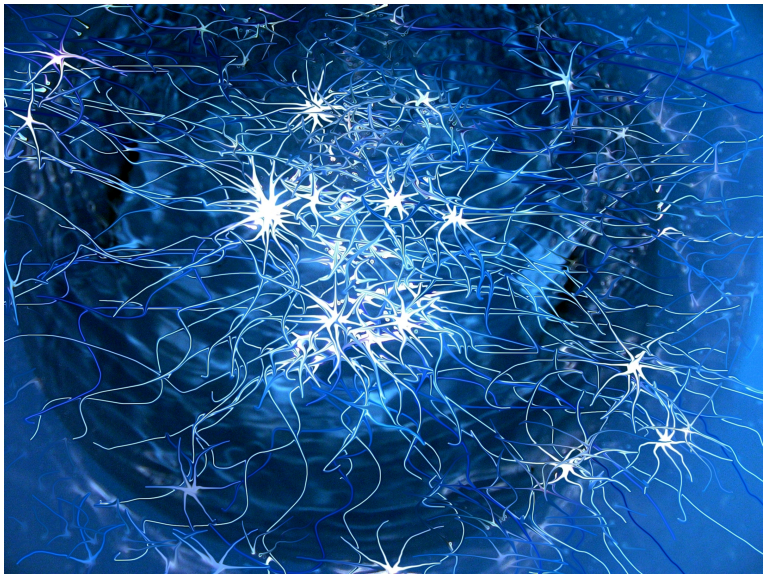
https://xkcd.com/1838/

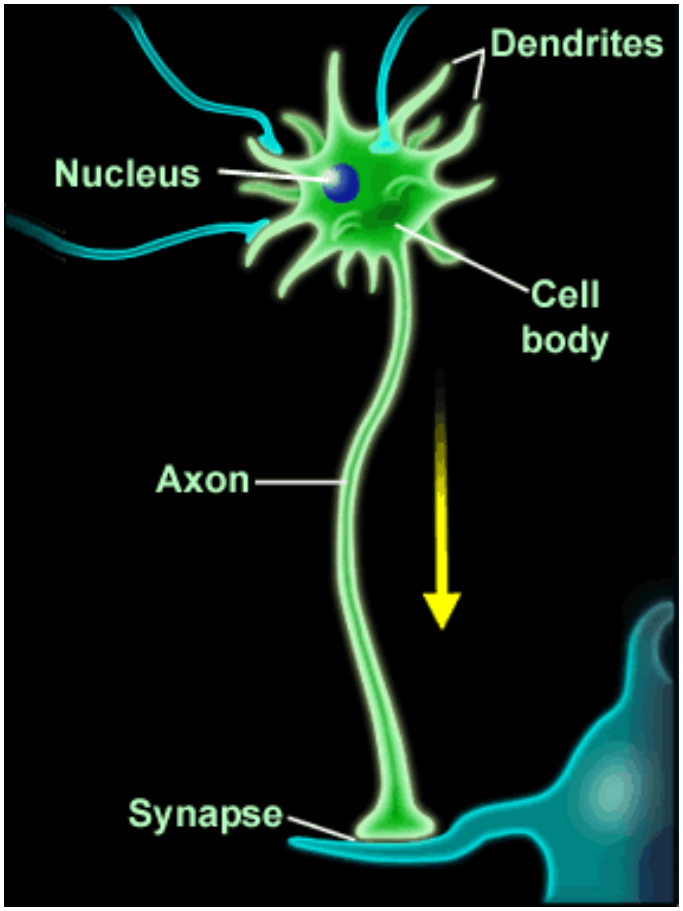
Neural networks

Artificial neural networks: computational models inspired by the brain

Properties:

- Highly interconnected
- Distributed computation/memory
- Robust to noise, failures

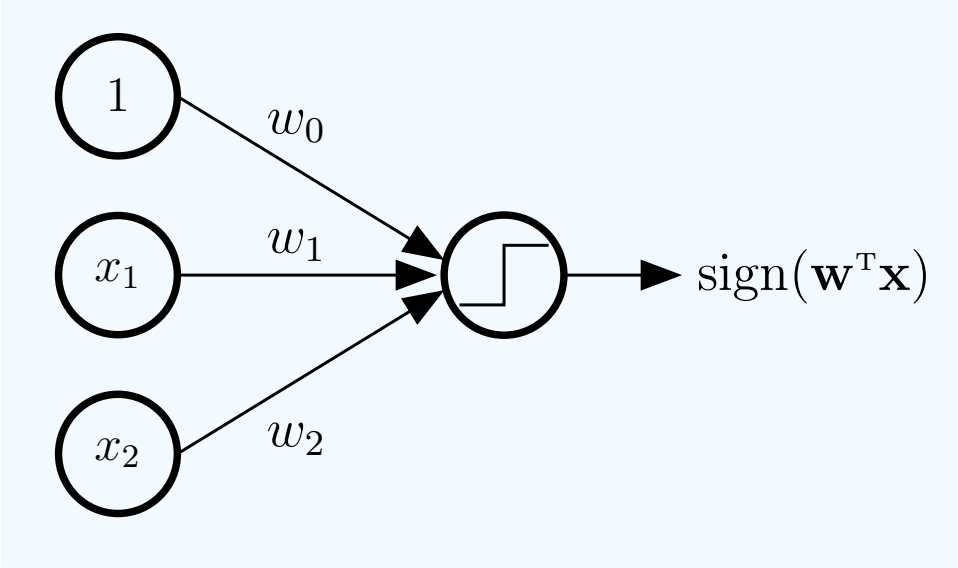




http://dragonoverwashington.blogspot.com/2013/03/the-blue-brain-project-making-human.html

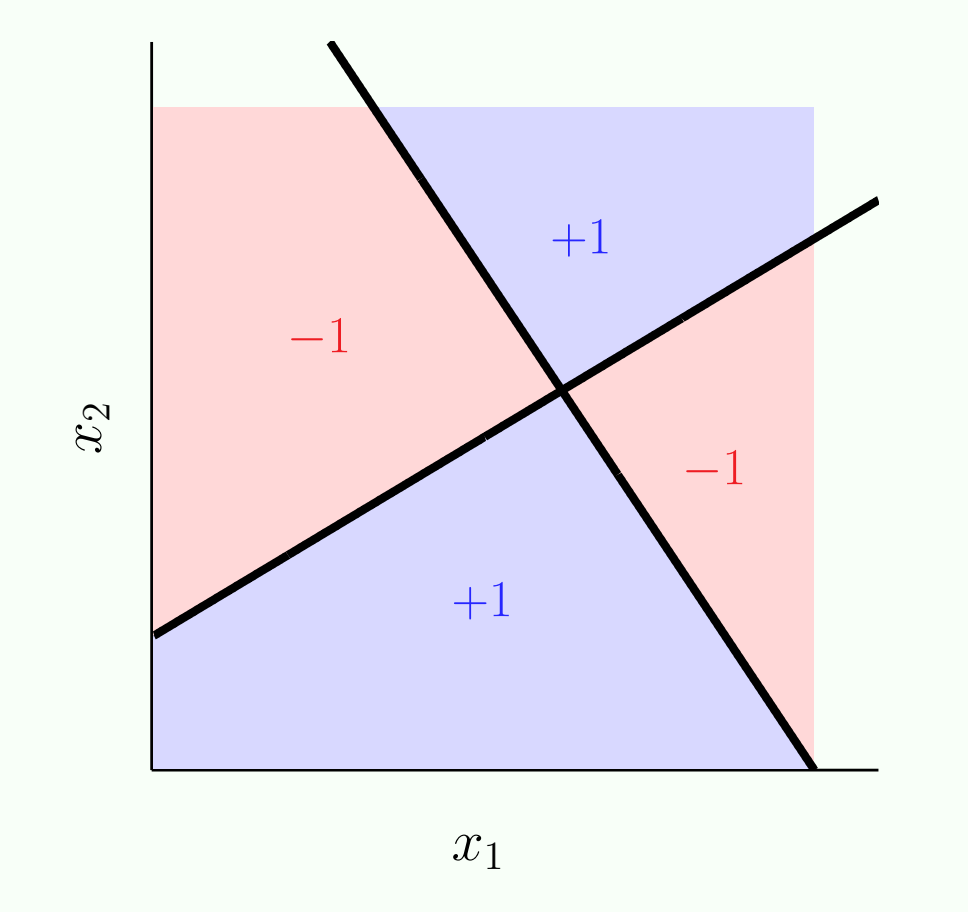
The perceptron as a neural network

Interpreting the perceptron as a single-layer neural network



Towards the multi-layer perceptron (MLP)

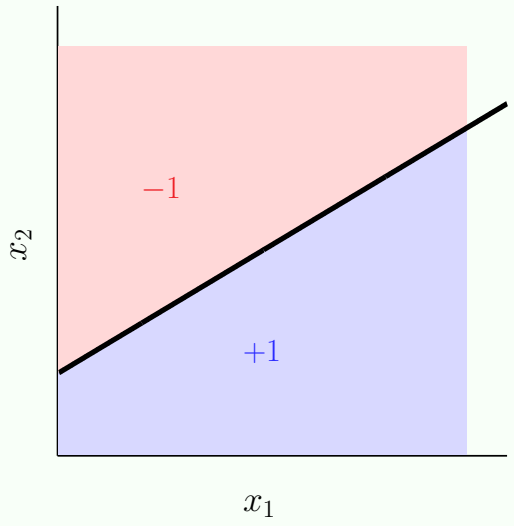
Consider the following classification problem:



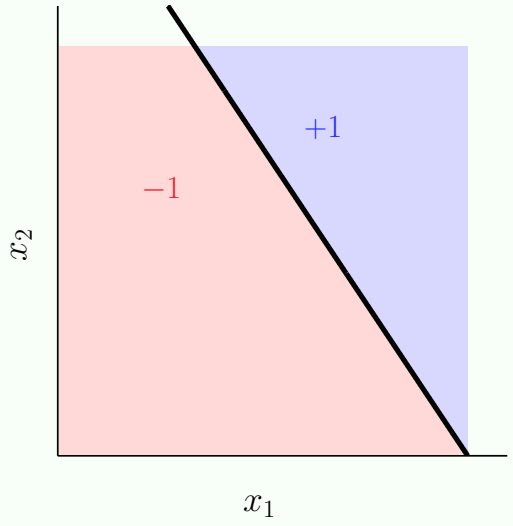
Clearly not solvable using a linear classifier!

Towards the multi-layer perceptron

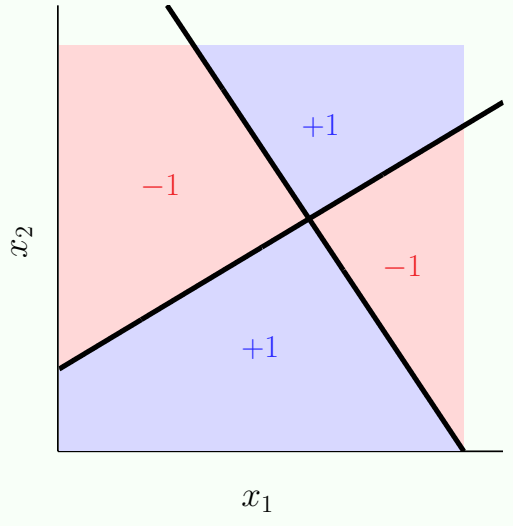
It can be addressed using a combination of multiple linear classifiers:



$h_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^{\text{T}}\mathbf{x})$



$h_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^{\text{T}}\mathbf{x})$

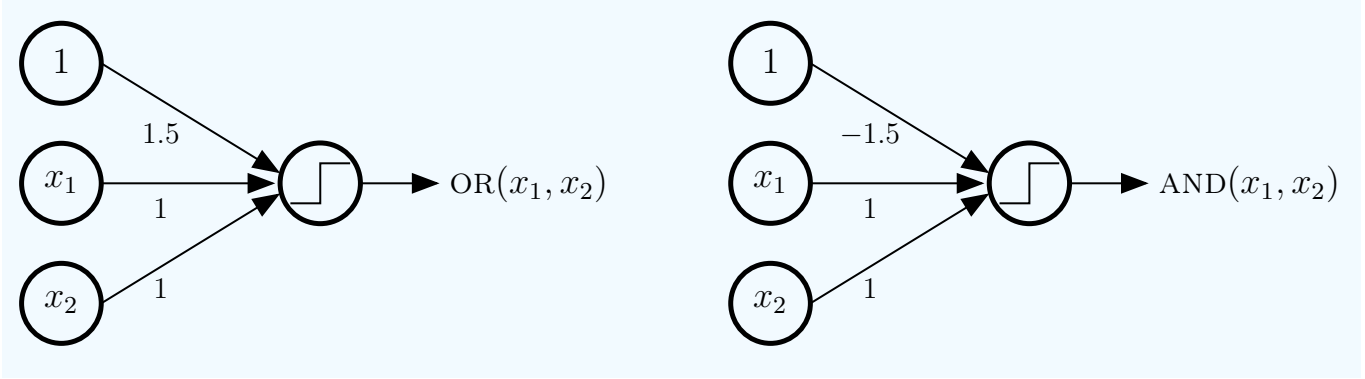


Our target function, f is XOR of h_1 and h_2, which can be expressed as: $f = h_1\overline{h_2} + \overline{h_1}h_2$

Towards the multi-layer perceptron

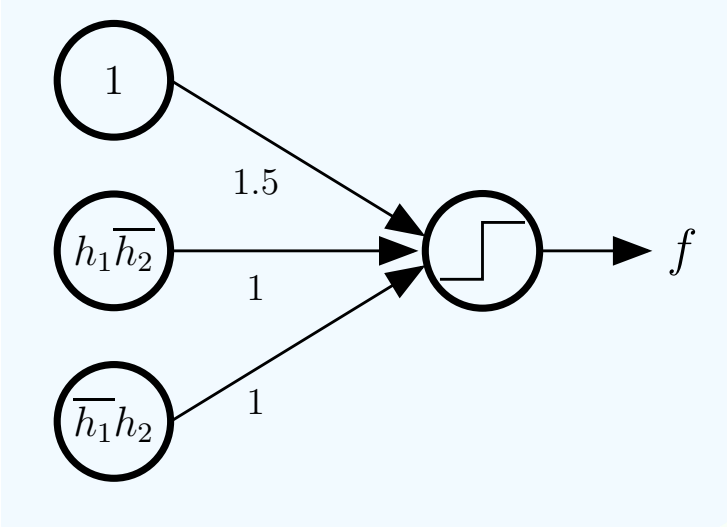
AND and OR gates can be implemented using a perceptron:

$$\begin{aligned} \mathrm{OR}(x_1, x_2) &= \mathrm{sign}(x_1 + x_2 + 1.5); \\ \mathrm{AND}(x_1, x_2) &= \mathrm{sign}(x_1 + x_2 - 1.5). \end{aligned}$$

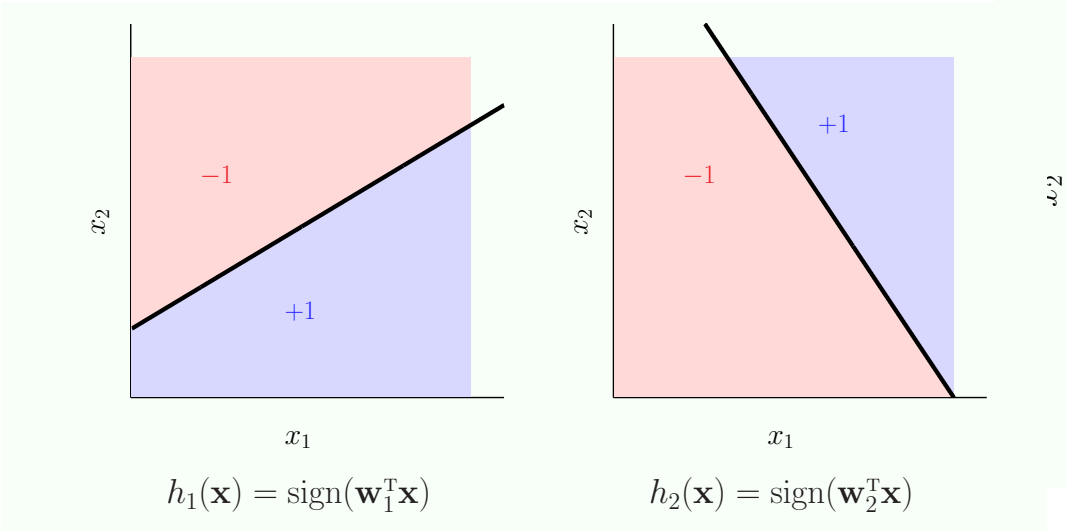


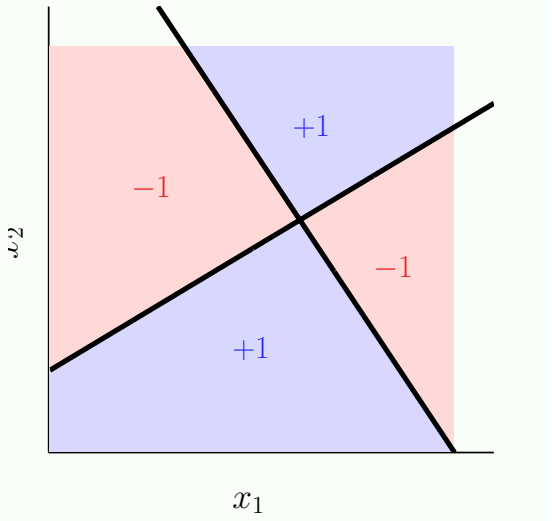
Towards the multi-layer perceptron

Towards implementing the target function:



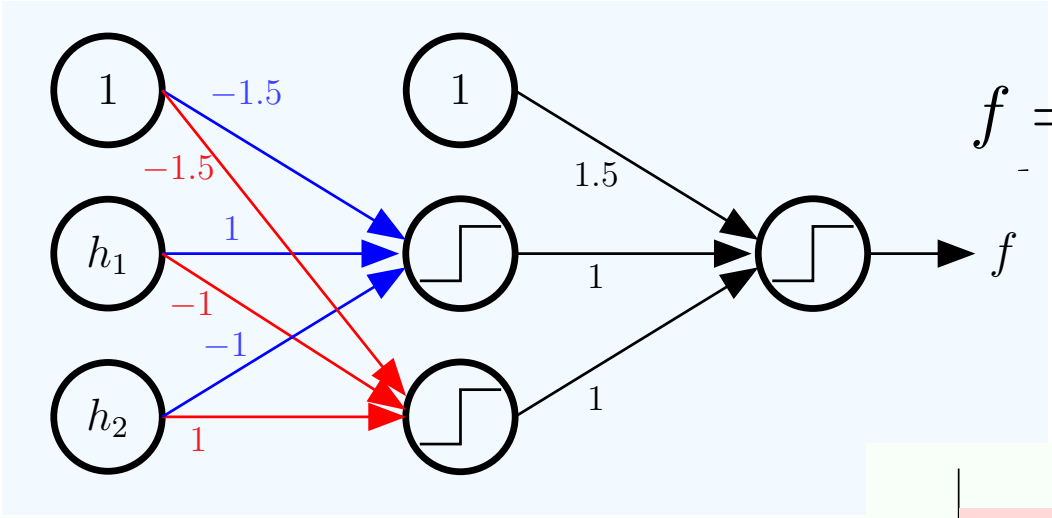
$$f = h_1\overline{h_2} + \overline{h_1}h_2$$



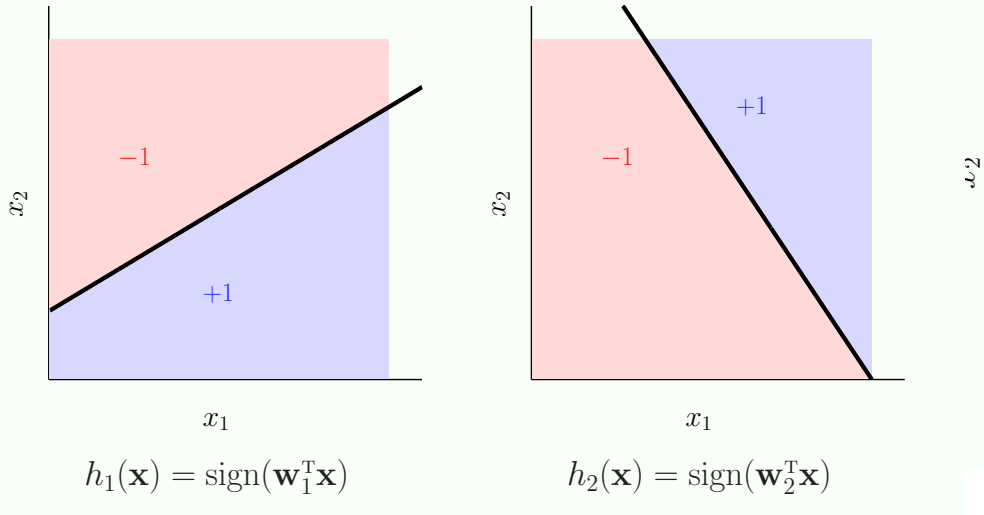


Towards the multi-layer perceptron

The target function is now represented as:

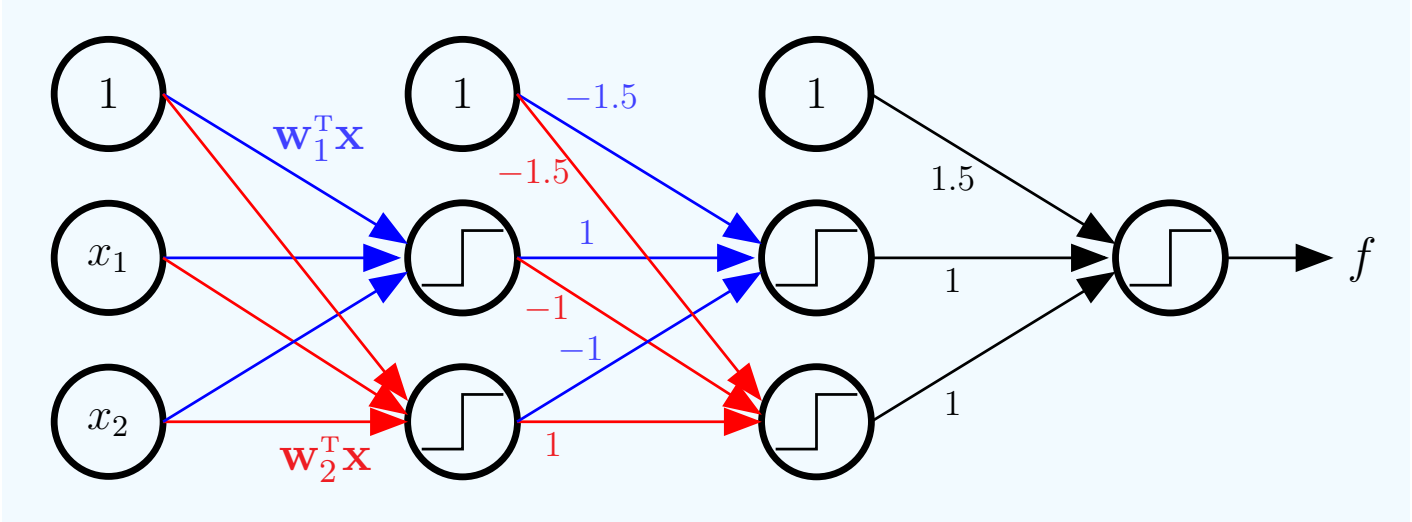


$$f = h_1\overline{h_2} + \overline{h_1}h_2$$



The multi-layer perceptron

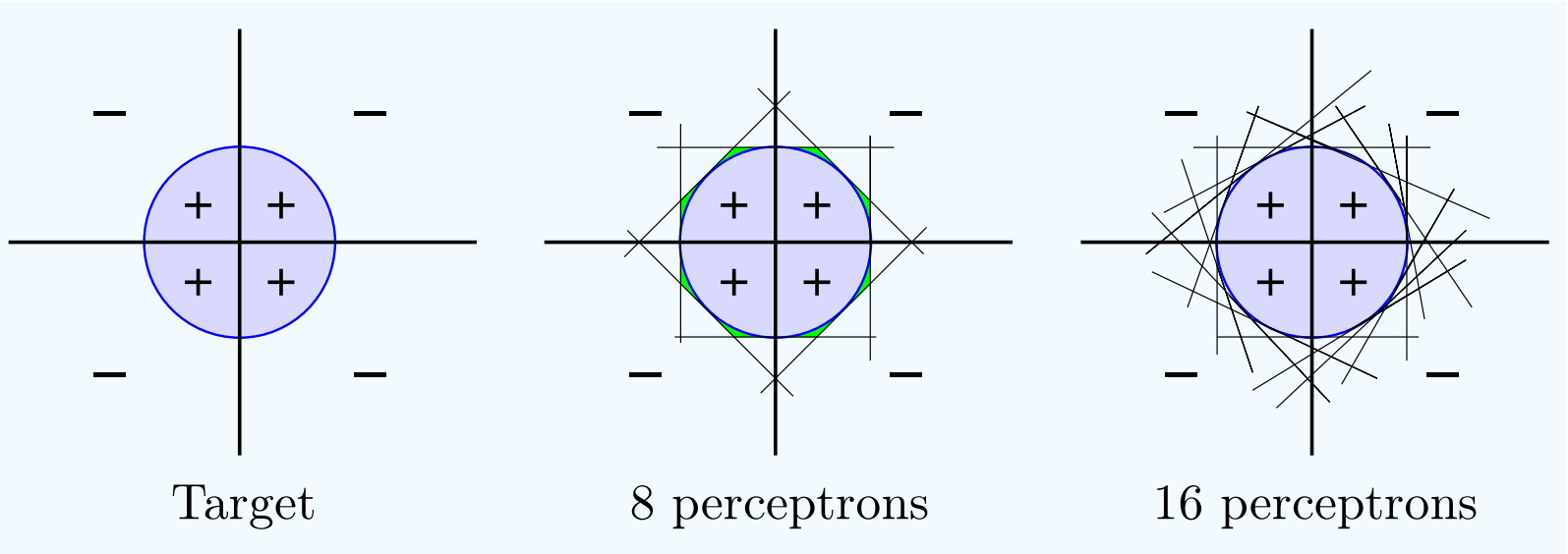
A graph representation of the target function:



More layers provide the flexibility required to represent the target function

The multi-layer perceptron

Given enough hidden neurons, it is possible to approximate arbitrary functions using this framework.

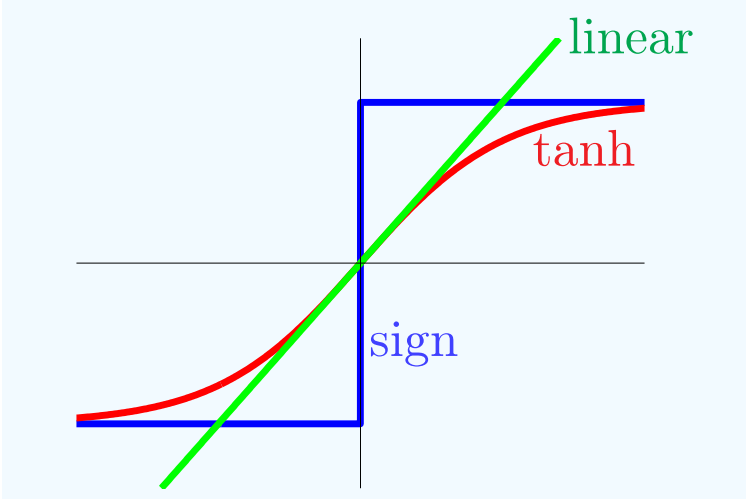


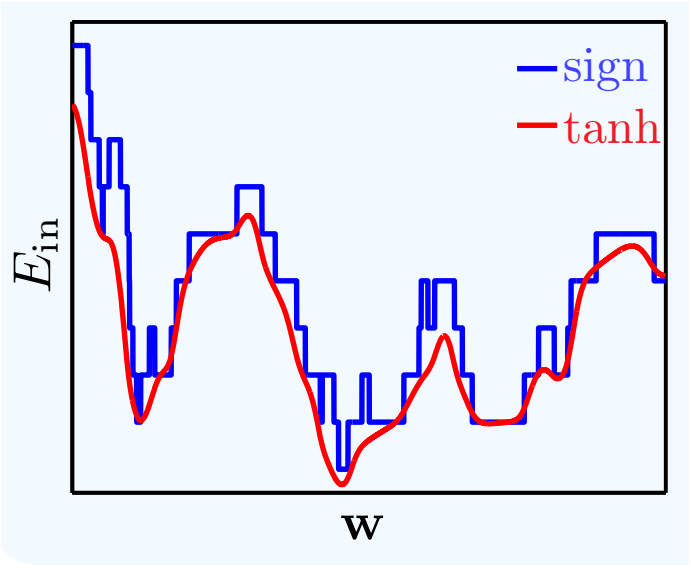
The problem: fitting the data is a combinatorial optimization problem (E_{in} is not a smooth function due to the sign function).

The multi-layer perceptron

The problem: fitting the data is a combinatorial optimization problem (E_{in} is not a smooth function due to the sign function).

Solution: replace the sign function with a smooth function

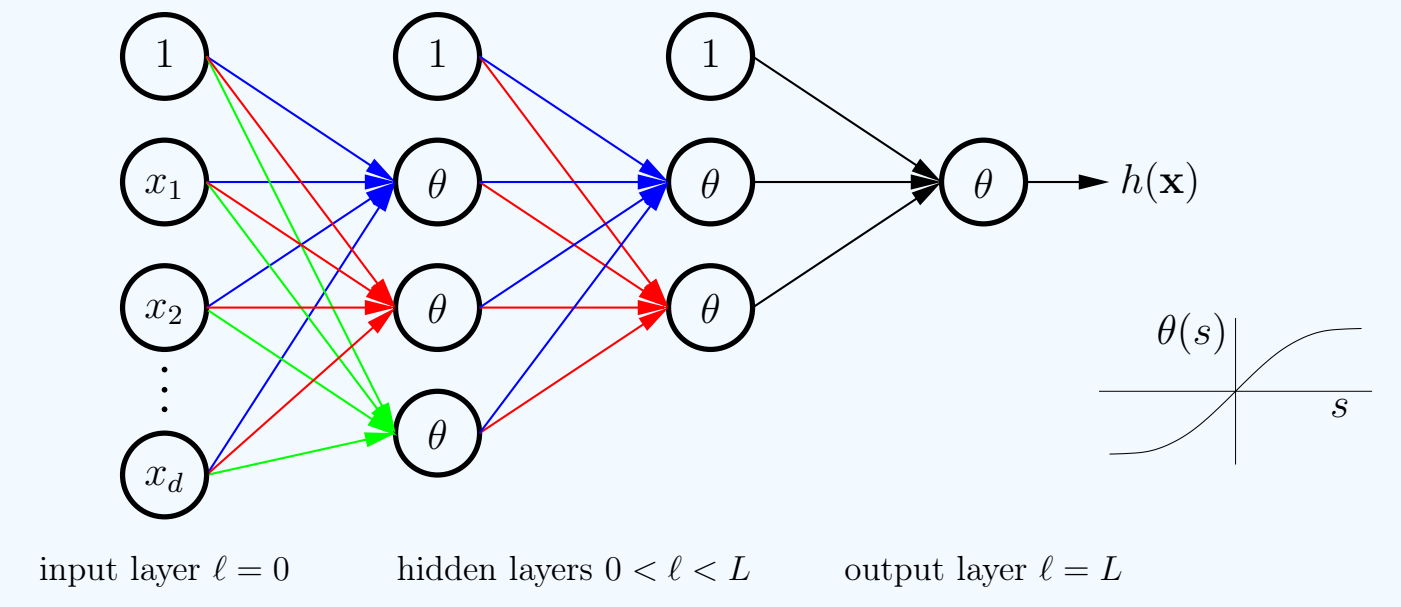




$$\theta(s) \;=\; \tanh(s) \;=\; \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$$

Feed forward neural networks

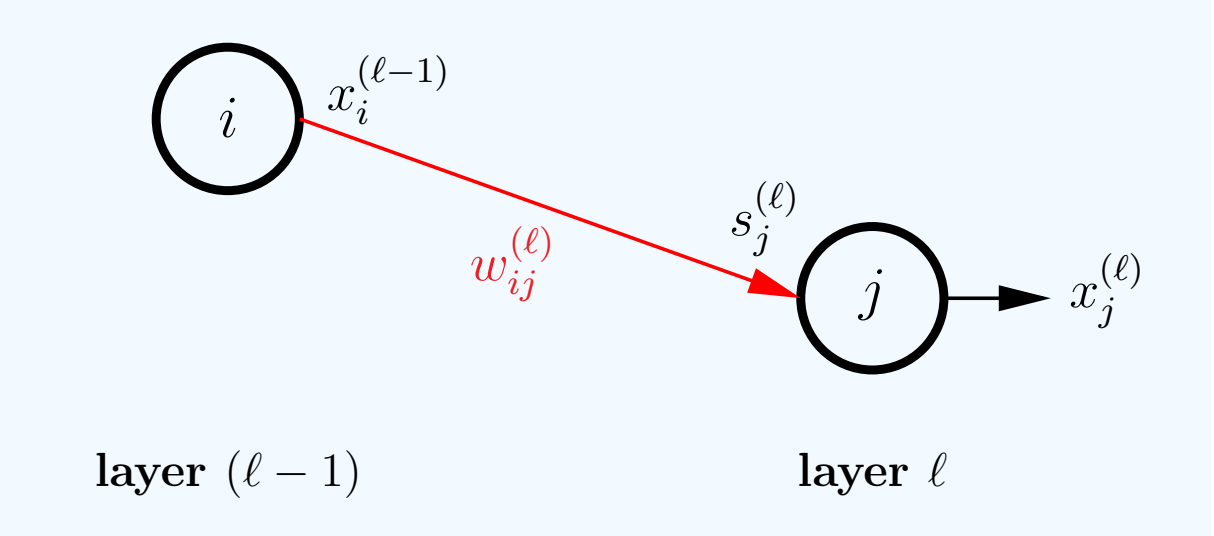
The architecture for a feed-forward neural network:



This network has three layers of neurons and weights (not counting the input layer).

Anatomy of a neuron

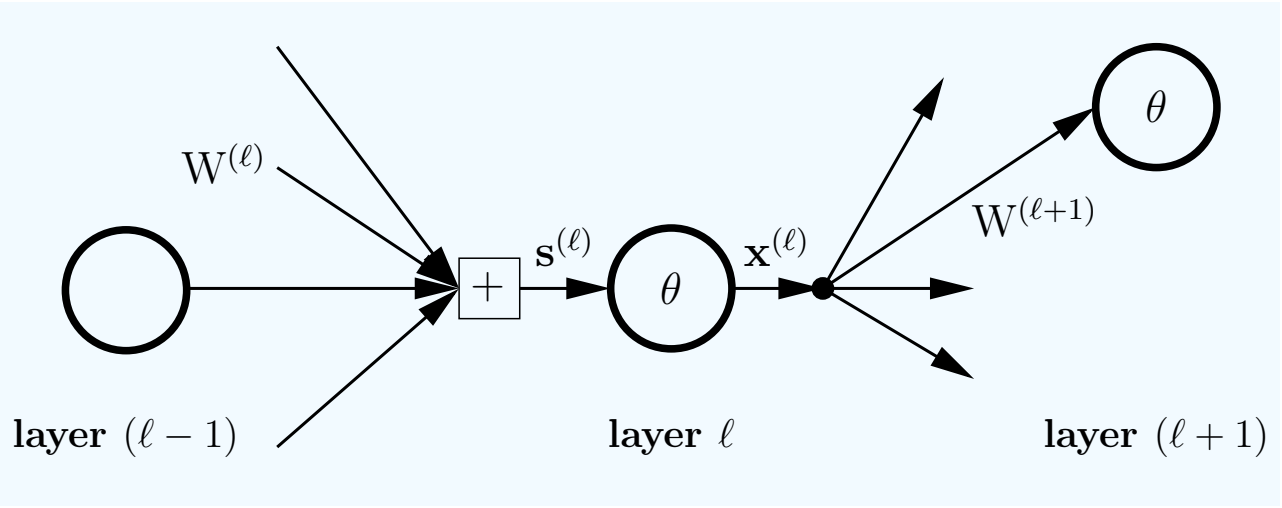
Let's look more closely at a pair of neurons



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

Anatomy of a neuron

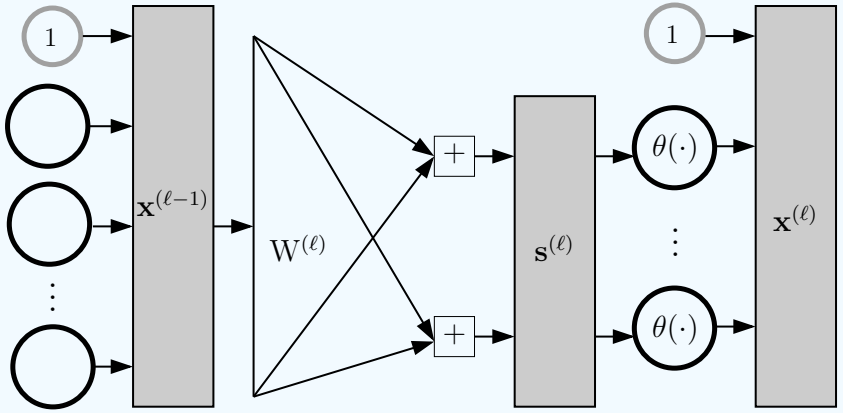
The interconnections of layer l:



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

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



$$s_j^{(\ell)} = (\mathbf{w}_j^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$$

$$\mathbf{s}^{(\ell)} = (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$$

$$\mathbf{x}^{(\ell)} = \begin{bmatrix} 1 \\ \theta(\mathbf{s}^{(\ell)}) \end{bmatrix}.$$

forward propagation

Computing the hypothesis h(x):

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

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

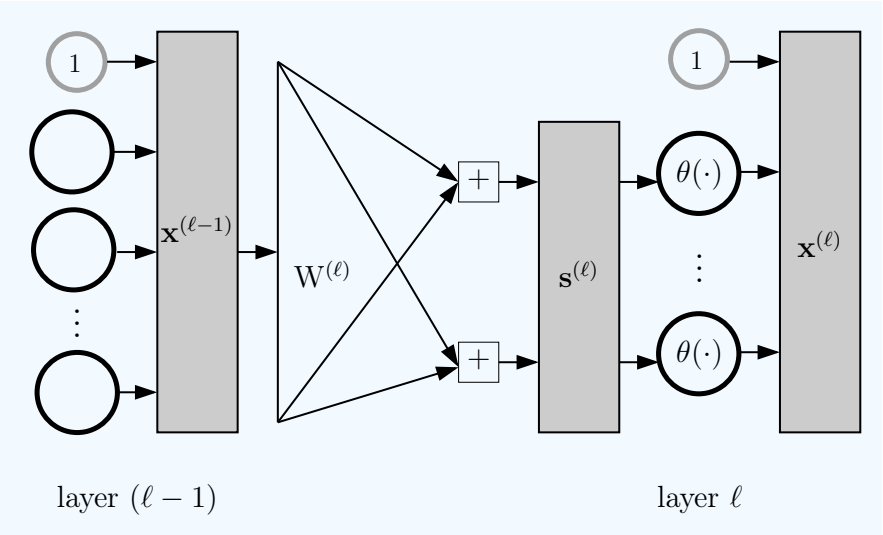
1: $\mathbf{x}^{(0)} \leftarrow \mathbf{x}$ **[Initialization]**

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

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

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

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



in-sample error

Computing the hypothesis h(x):

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

Now we are ready to compute the in-sample error:

$$\begin{aligned} E_{\text{in}}(\mathbf{w}) &= \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n;\mathbf{w}) - y_n)^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n^{(L)} - y_n)^2. \end{aligned}$$

If we use a smooth activation function, the in-sample error is differentiable, and we can use gradient descent:

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

Minimizing the in-sample error

Let's express the in-sample error as:

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathsf{e}_n.$$

where: $\mathsf{e}_n = \mathsf{e}\big(h(\mathbf{x}_n), y_n\big).$

And for the squared error $\mathsf{e}(h, y) = (h - y)^2$

We need the derivative with respect to each weight matrix:

$$\frac{\partial E_{\text{in}}}{\partial \mathbf{W}^{(\ell)}} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial \mathsf{e}_n}{\partial \mathbf{W}^{(\ell)}}$$

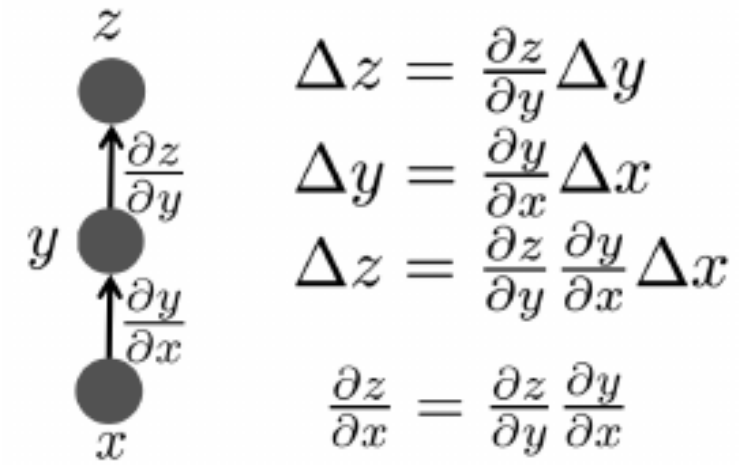
So we need $\dfrac{\partial \mathsf{e}(\mathbf{x})}{\partial \mathbf{W}^{(\ell)}}$

Digression: the chain rule

We need to take the derivative $\frac{\partial \mathrm{e}(\mathbf{x})}{\partial \mathrm{W}^{(\ell)}}$

However, the error is not a direct function of the weights.

To find it, first let's consider a simpler case:



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

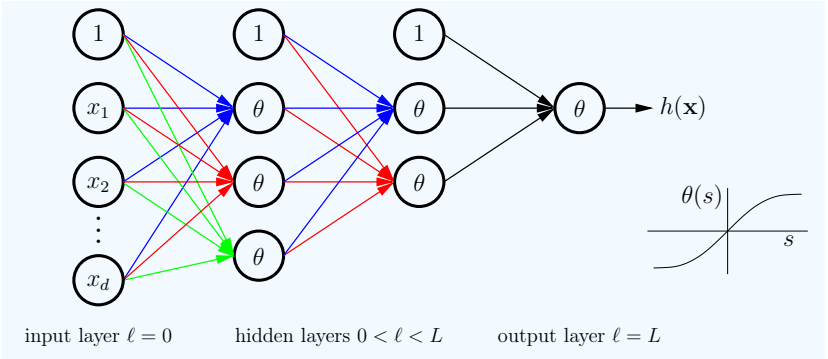


Image from http://www.iro.umontreal.ca/~bengioy/dlbook/

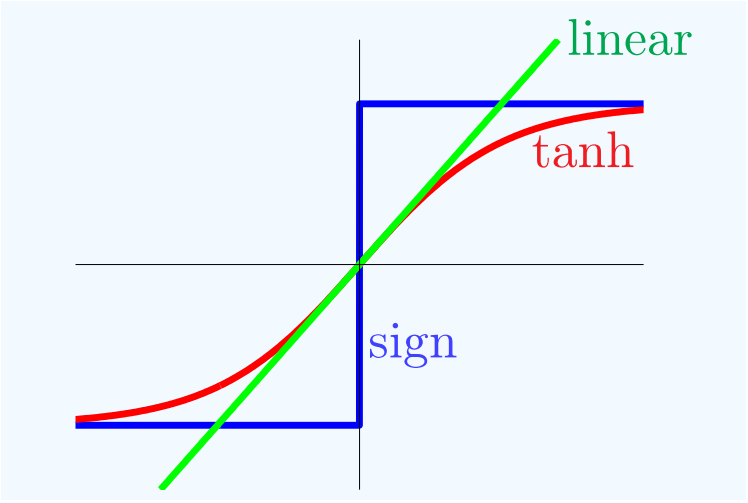
Example: the perceptron

In the case of a perceptron with a single neuron and tanh activation function:

For the sigmoidal perceptron, $h(\mathbf{x}) = \tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, let the in-sample error be $E_{\mathrm{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)^2$. Show that

$$\nabla E_{\mathrm{in}}(\mathbf{w}) = \frac{2}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)(1 - \tanh^2(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n))\mathbf{x}_n.$$

What happens when components of **w** are large?



$$\theta(s) \;=\; \tanh(s) \;=\; \frac{e^{s} - e^{-s}}{e^{s} + e^{-s}}$$

Example: the perceptron

In the case of a perceptron with a single neuron and tanh activation function:

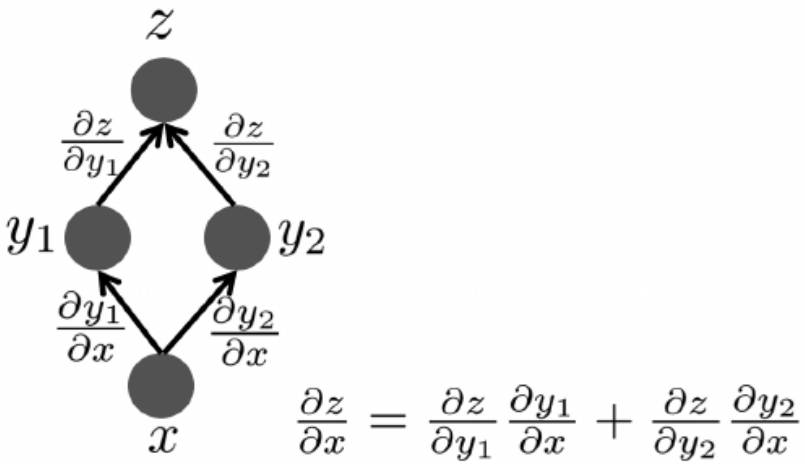
For the sigmoidal perceptron, $h(\mathbf{x}) = \tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, let the in-sample error be $E_{\mathrm{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)^2$. Show that

$$\nabla E_{\mathrm{in}}(\mathbf{w}) = \frac{2}{N}\sum_{n=1}^{N}(\tanh(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n) - y_n)(1 - \tanh^2(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n))\mathbf{x}_n.$$

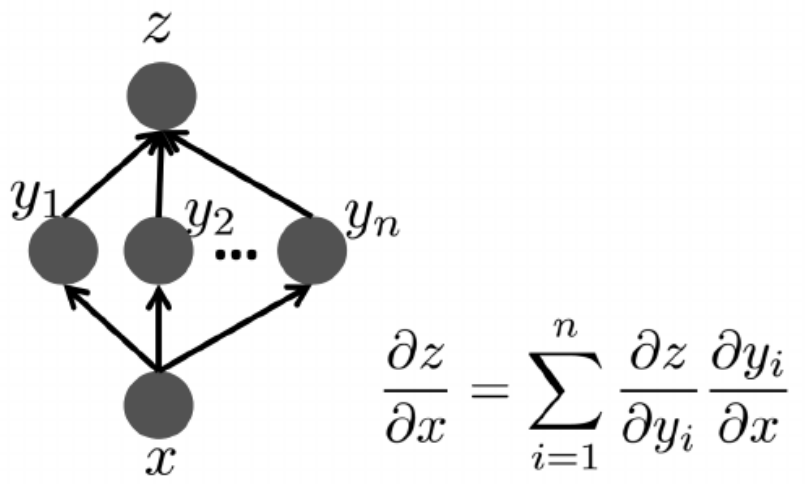
For multi-layer architectures there is no closed form expression for the gradient

Digression: the chain rule

When there are multiple intermediate variables we need to sum the influence of each one:



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

Image from http://www.iro.umontreal.ca/~bengioy/dlbook/

Computing the gradient

$\mathrm{e}(\mathbf{x})$ is a function of $\mathbf{s}^{(\ell)}$ and $\mathbf{s}^{(\ell)} = (\mathrm{W}^{(\ell)})^{\mathrm{T}}\mathbf{x}^{(\ell-1)}$

Therefore we can express the gradient of the error as:

$$\frac{\partial \mathrm{e}}{\partial \mathrm{W}^{(\ell)}} = \frac{\partial \mathbf{s}^{(\ell)}}{\partial \mathrm{W}^{(\ell)}} \cdot \left(\frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(\ell)}}\right)^{\mathrm{T}}$$

$$= \mathbf{x}^{(\ell-1)}(\boldsymbol{\delta}^{(\ell)})^{\mathrm{T}}$$

(by the chain rule)

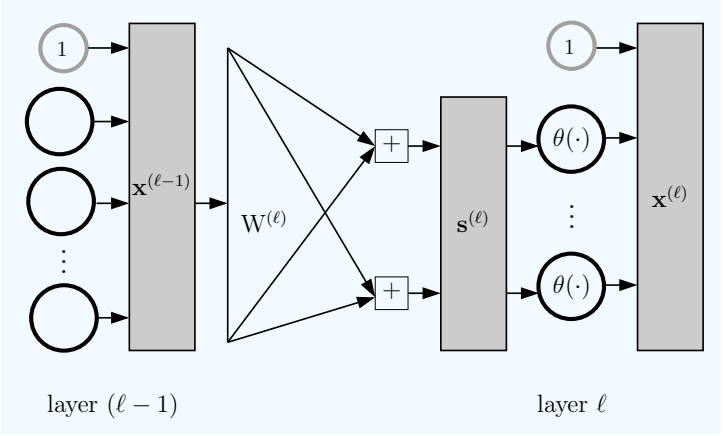
Dimensions:

$$(d^{(\ell-1)} + 1) \times d^{(\ell)}$$

where

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathrm{e}}{\partial \mathbf{s}^{(\ell)}}$$

is the sensitivity vector for layer *l*



Backpropagation

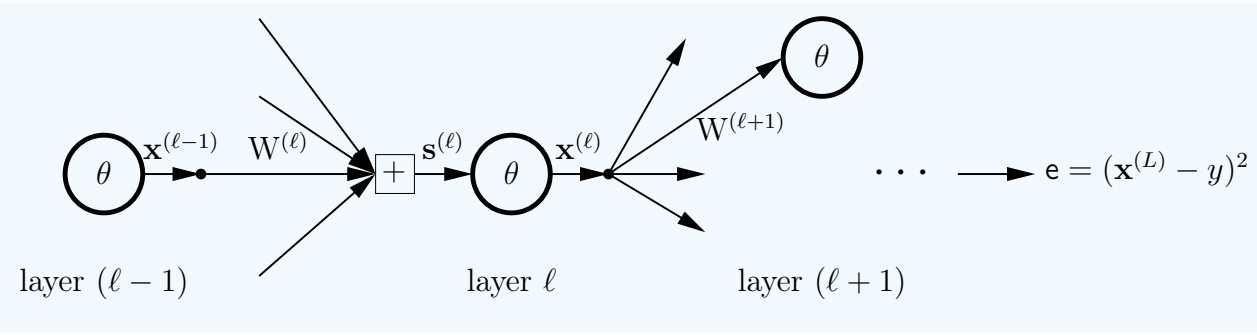
Computing the sensitivities:

$$\boldsymbol{\delta}_j^{(\ell)} = \frac{\partial \mathsf{e}}{\partial \mathbf{s}_j^{(\ell)}} = \frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathbf{x}_j^{(\ell)}}{\partial \mathbf{s}_j^{(\ell)}} = \theta'\left(\mathbf{s}_j^{(\ell)}\right) \cdot \frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}}$$

(since e depends on $s^{(l)}$ only through $x^{(l)}$)

$$\frac{\partial \mathsf{e}}{\partial \mathbf{x}_j^{(\ell)}} = \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial \mathbf{s}_k^{(\ell+1)}}{\partial \mathbf{x}_j^{(\ell)}} \cdot \frac{\partial \mathsf{e}}{\partial \mathbf{s}_k^{(\ell+1)}} = \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}.$$

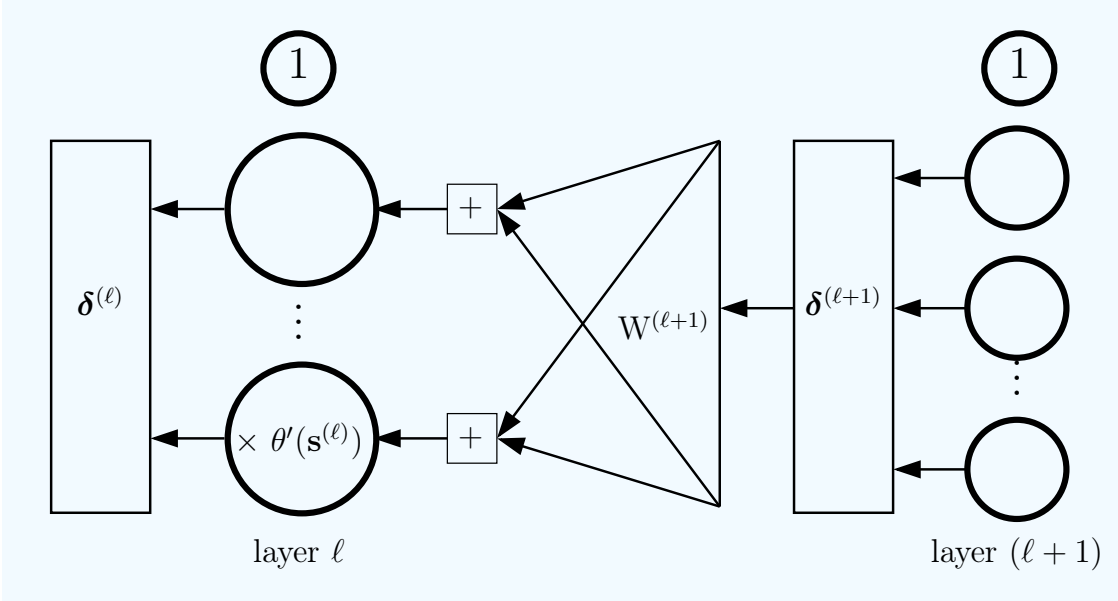
(because a change in component j of $x^{(l)}$ affects every component of $s^{(l+1)}$)



Backpropagation

We can compute the gradient by "backpropagating" the sensitivity vectors:

$$\boldsymbol{\delta}_j^{(\ell)} = \theta'(\mathbf{s}_j^{(\ell)}) \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}$$



$$\boldsymbol{\delta}^{(1)} \longleftarrow \boldsymbol{\delta}^{(2)} \cdots \longleftarrow \boldsymbol{\delta}^{(L-1)} \longleftarrow \boldsymbol{\delta}^{(L)}$$

Backpropagation

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

Backpropagation to compute sensitivities $\boldsymbol{\delta}^{(\ell)}$:
(Assume $\mathbf{s}^{(\ell)}$ and $\mathbf{x}^{(\ell)}$ have been computed for all ℓ)

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

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

3: Compute (for tanh hidden node):

$$\theta'(\mathbf{s}^{(\ell)}) = \left[1 - \mathbf{x}^{(\ell)} \otimes \mathbf{x}^{(\ell)}\right]_1^{d^{(\ell)}}$$

4: $\boldsymbol{\delta}^{(\ell)} \leftarrow \theta'(\mathbf{s}^{(\ell)}) \otimes \left[\mathrm{W}^{(\ell+1)}\boldsymbol{\delta}^{(\ell+1)}\right]_1^{d^{(\ell)}}$ $\longleftarrow$ componentwise multiplication

5: **end for**

$$\boldsymbol{\delta}_j^{(\ell)} = \theta'(\mathbf{s}_j^{(\ell)}) \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(\ell+1)} \boldsymbol{\delta}_k^{(\ell+1)}$$

Backpropagation

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

1: Initialize: $E_{\text{in}} = 0$; for $\ell = 1, \ldots, L$, $\mathrm{G}^{(\ell)} = 0 \cdot \mathrm{W}^{(\ell)}$.
2: **for** Each data point $\mathbf{x}_n$ $(n = 1, \ldots, N)$ **do**
3: Compute $\mathbf{x}^{(\ell)}$ for $\ell = 0, \ldots, L$. [forward propagation]
4: Compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = 1, \ldots, L$. [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: $\mathrm{G}^{(\ell)} \leftarrow \mathrm{G}^{(\ell)} + \frac{1}{N}\mathrm{G}^{(\ell)}(\mathbf{x}_n)$.
9: **end for**
10: **end for**

Using the gradient for gradient descent: $\mathrm{W}^{(\ell)} \leftarrow \mathrm{W}^{(\ell)} - \eta \mathrm{G}^{(\ell)}$

History of backpropagation

The method was proposed independently several times:

Rumelhart, David E.; Hinton, Geoffrey E., Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". *Nature* **323** (6088): 533–536.

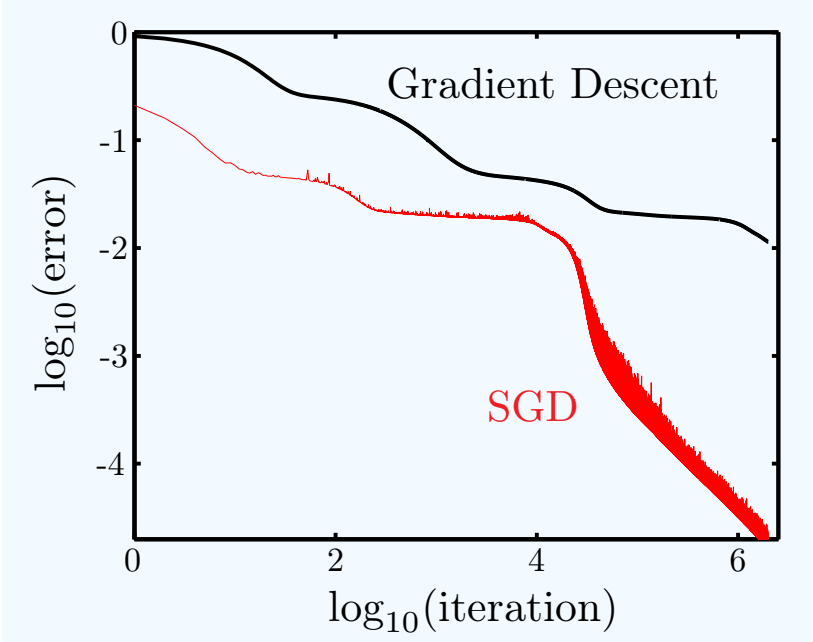
Paul J. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974

Batch vs stochastic update

Stochastic gradient descent: update the weight vector after the gradient with respect to a given training example has been computed.

Batch gradient descent: aggregate the gradients across all training examples before updating the weight vector.

SGD is more effective:



Batch vs stochastic update

Batch training is typically slower than stochastic.

Why?

Assume there are several duplicates of a given example.

The update made from the duplicates is not contributing to convergence.

In real data you will have examples that are correlated.

There are methods for accelerating batch training (e.g. conjugate gradient method), and it's easier to determine convergence.

Batch vs stochastic update

Batch training is typically slower than stochastic.

There are methods for accelerating batch training (e.g. conjugate gradient method), and it's easier to determine convergence.

Stochastic learning often leads to better local minima (the cost function is not monotonically decreasing, and can sometimes help escape to a basin of a deeper local minimum).

Can smooth the progress of stochastic learning by processing examples in batches, and varying batch size and learning rate.

More tricks

Neural networks benefit from the same types of normalization as other geometrical methods (SVMs/perceptron)

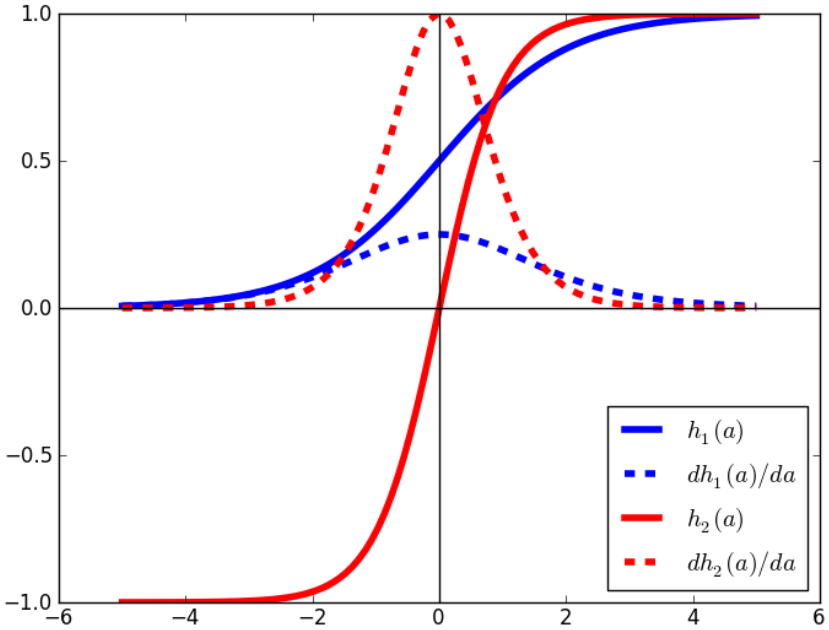
Activation functions

Common activation functions:

$$\tanh(x)$$

Logistic function

$$\frac{1}{1+\exp(x)}$$



Linear
(useful for regression problems)

Gaussian (aka radial basis function)

The input to a neuron should be in the region where the derivative is non-flat

Activation functions

The output layer should use the logistic function if you want probability-like values

Comments about gradient-based training

What can we say about the error surface?

Can have multiple local minima. Therefore gradient-based training doesn't necessarily finds the global minimum.

Is this a problem? Not necessarily.

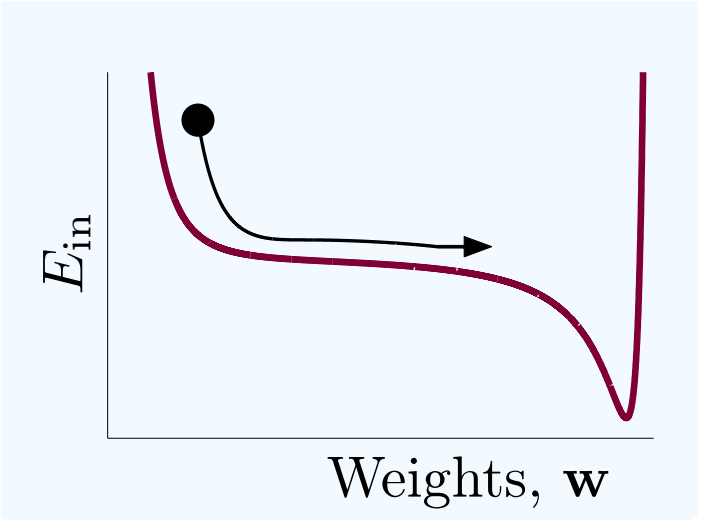
Compare to SVM which have a globally optimal solution, and typically faster training times.

Comments about gradient-based training

What can we say about the error surface?

Can have multiple local minima. Therefore gradient-based training doesn't necessarily finds the global minimum.

Another issue: plateaus – regions where the error is more or less constant

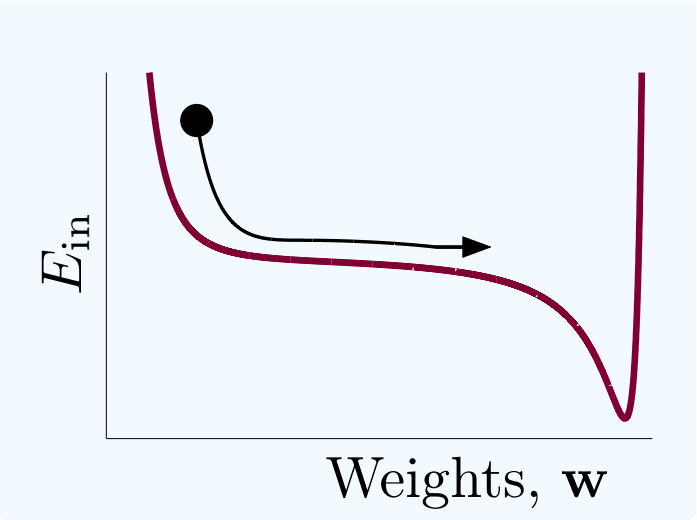


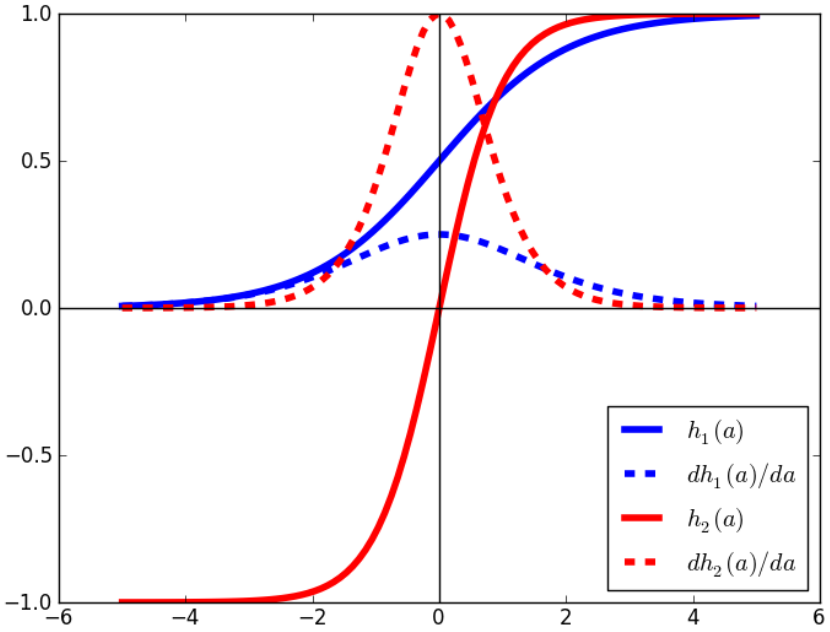
Comments about gradient-based training

Plateaus arise because of saturation of the activation function.

How to avoid them?

Initialize with small weights





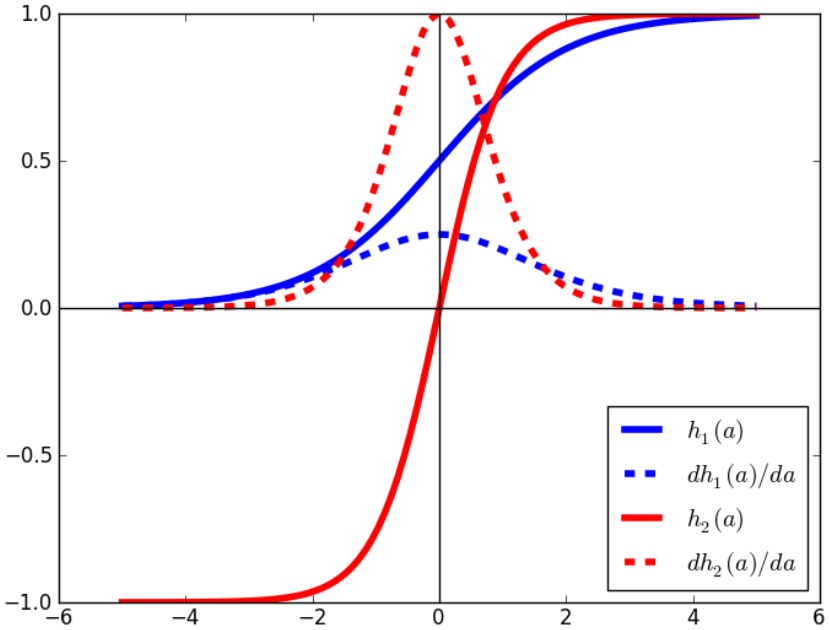
Initialize weights to be inversely proportional to the fan-in of a unit

Comments about gradient-based training

Plateaus arise because of saturation of the activation function.

How to avoid them?

Momentum!



$$\mathbf{w}(m+1) = \underbrace{\mathbf{w}(m) + \Delta\mathbf{w}(m)}_{\text{gradient descent}} + \underbrace{\alpha\Delta\mathbf{w}(m-1)}_{\text{momentum}}$$

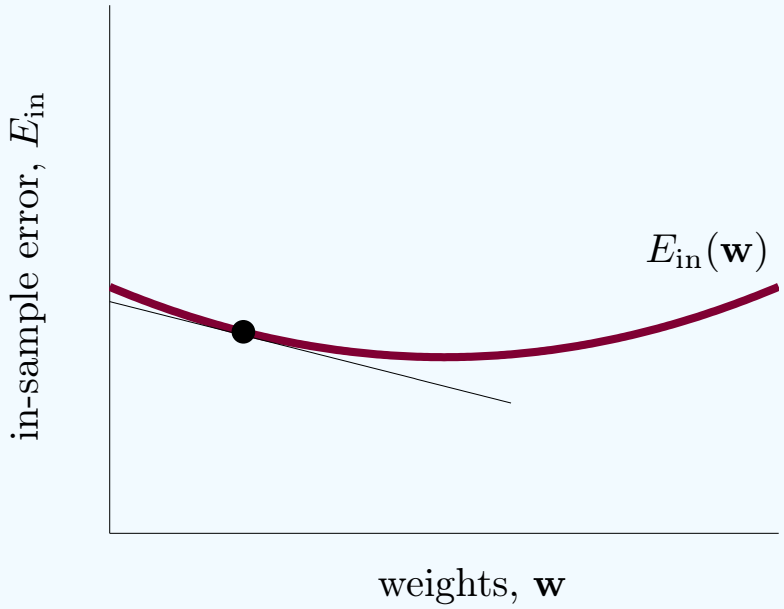
Beefing up gradient descent (section 7.5)

There are some methods for improving gradient descent.

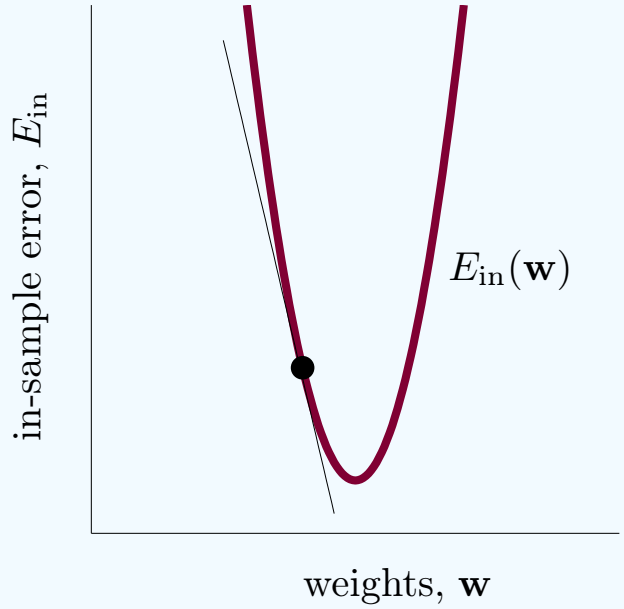
Alternative: go for more sophisticated methods altogether

Choosing a learning rate

The choice of the learning rate should depend on the shape of the error surface:



wide: use large η.



narrow: use small η.

Variable learning rate

A simple heuristic: if the error drops, increase, if it increases, reject the update and decrease

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: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta_t \mathbf{v}(t)$;
increment η: $\eta_{t+1} = \alpha \eta_t$. $\alpha \in [1.05, 1.1]$

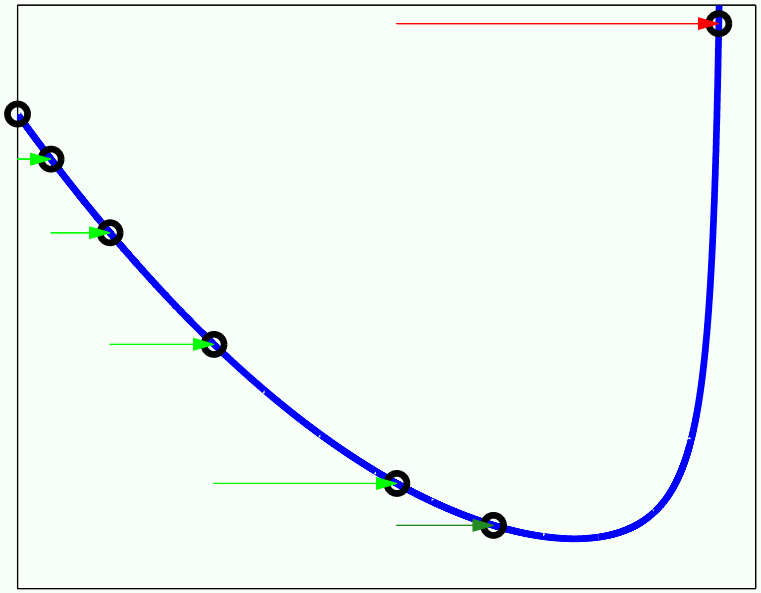
6: **else**

7: reject: $\mathbf{w}(t+1) = \mathbf{w}(t)$;
decrease η: $\eta_{t+1} = \beta \eta_t$. $\beta \in [0.7, 0.8]$

8: **end if**

9: Iterate to the next step, $t \leftarrow t + 1$.

10: **end while**



Choosing the parameters: $\alpha \approx 1.05 - 1.1$
$\beta \approx 0.5 - 0.8$.

Steepest descent

1: Initialize $w(0)$ and set $t = 0$;

2: **while** stopping criterion has not been met **do**

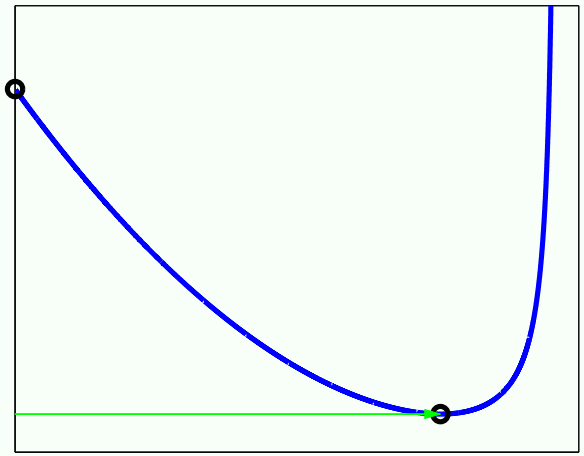
3: Let $\mathbf{g}(t) = \nabla E_{\text{in}}(\mathbf{w}(t))$, and set $\mathbf{v}(t) = -\mathbf{g}(t)$.

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, $t \leftarrow t + 1$.

7: **end while**



Use line search to find the optimal value of the learning rate

Line search

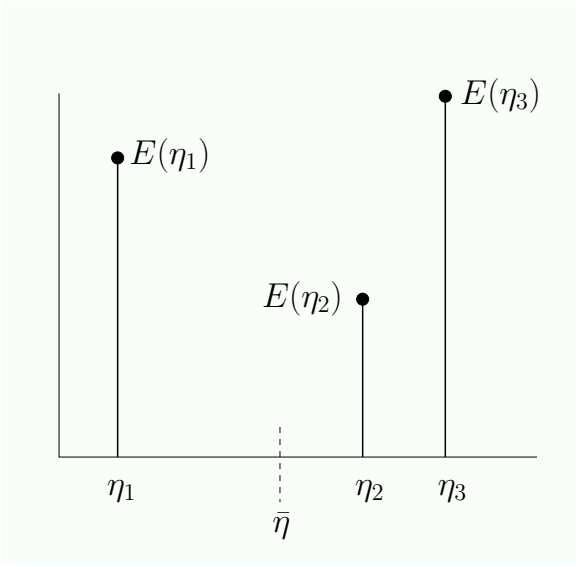
Find values $\eta_1 < \eta_2 < \eta_3$ with

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

Since E is continuous, there must be a local minimum in the interval $[\eta_1, \eta_3]$. Now, consider the midpoint of the interval,

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

Iterate this process.



Comparison of optimization methods

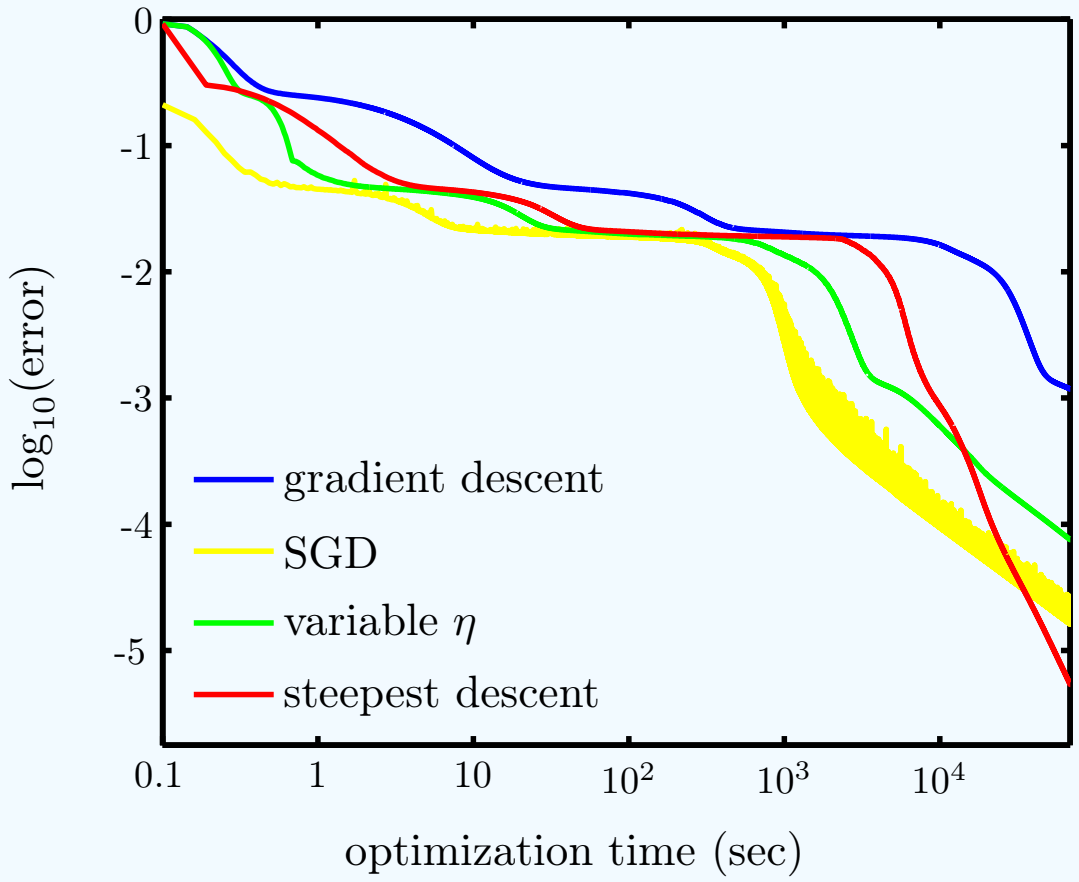
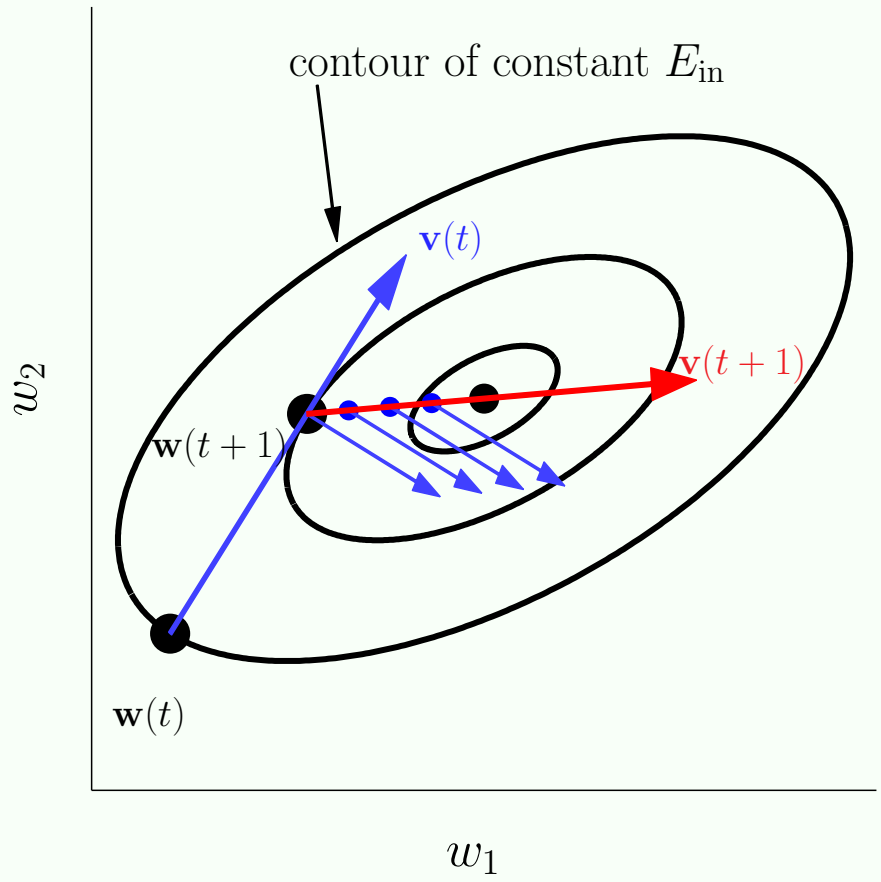
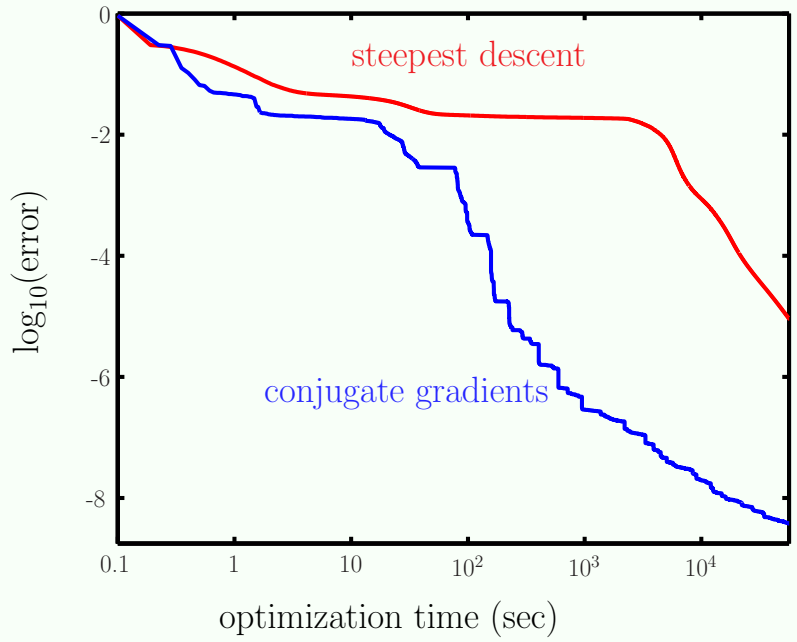


Figure 7.4: Gradient descent, variable learning rate and steepest descent using digits data and a 5 hidden unit 2-layer neural network with linear output. For variable learning rate, $\alpha = 1.1$ and $\beta = 0.8$.

Conjugate gradients

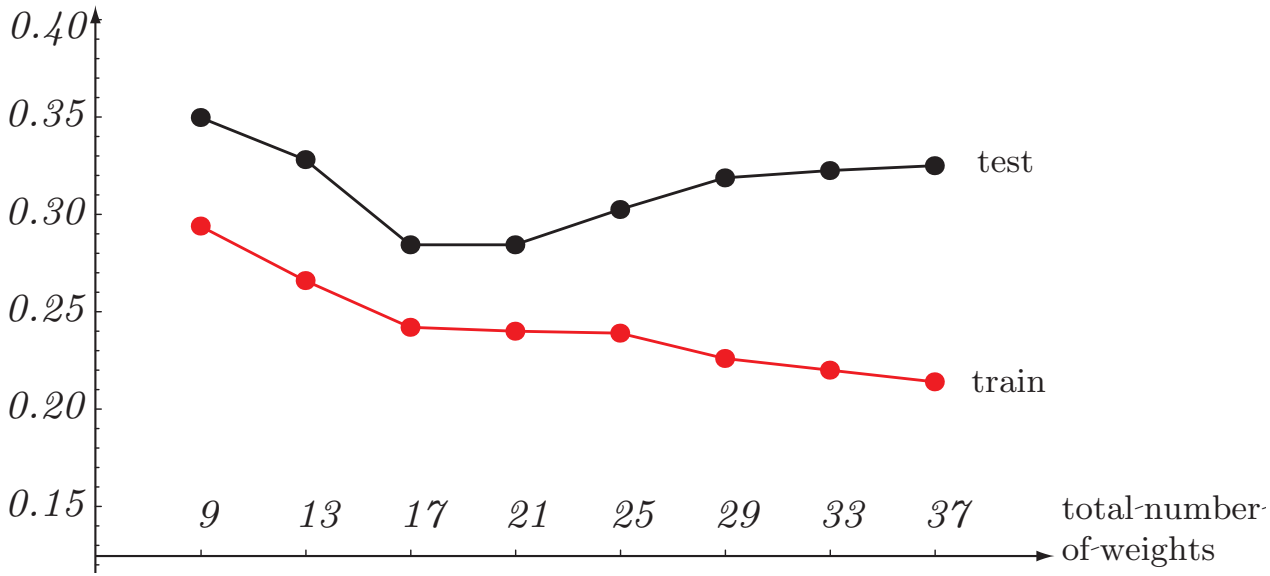
Choose a better direction than the gradient





Accuracy depends on the number of hidden units

The number of hidden units governs the expressive power of the network.



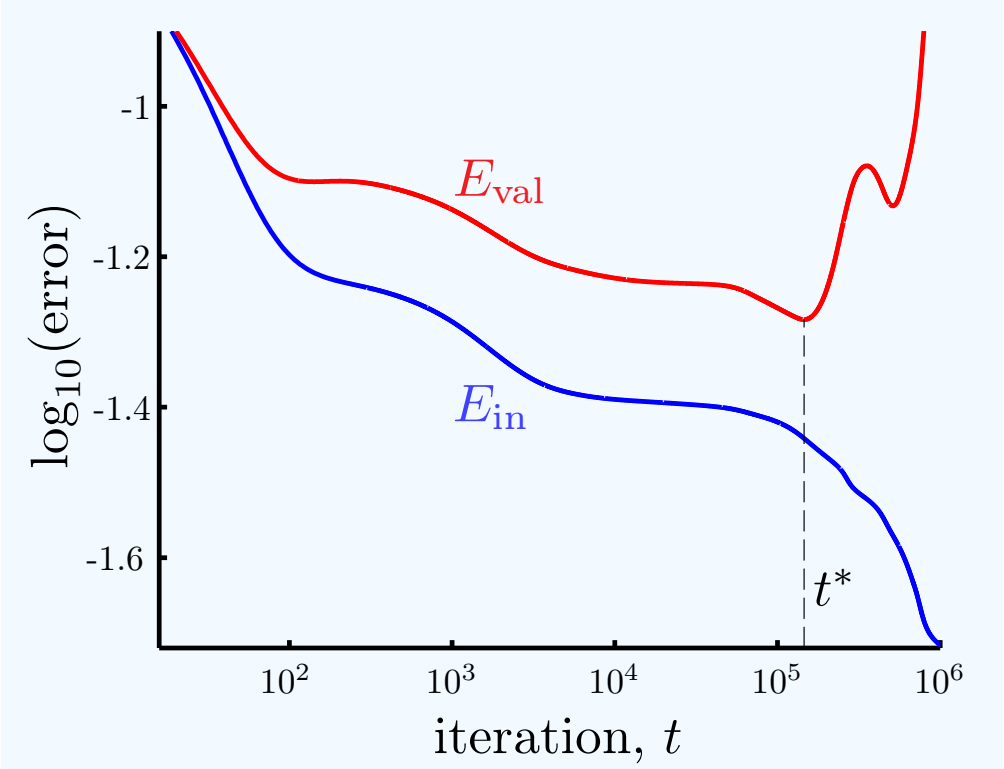
Regularization (section 7.4.1)

Can add a regularizer to the error function, e.g. $||\mathbf{w}||^2$

In the context of NNs this is called weight decay

$$E_{\text{aug}}(\mathbf{w}) \quad = \quad E_{\text{in}}(\mathbf{w}) + \frac{\lambda}{N} \sum_{\ell,i,j} (w_{ij}^{(\ell)})^2$$

Early stopping (section 7.4.2)



Early stopping (halting before a local minimum is reached) is a method for avoiding overfitting.

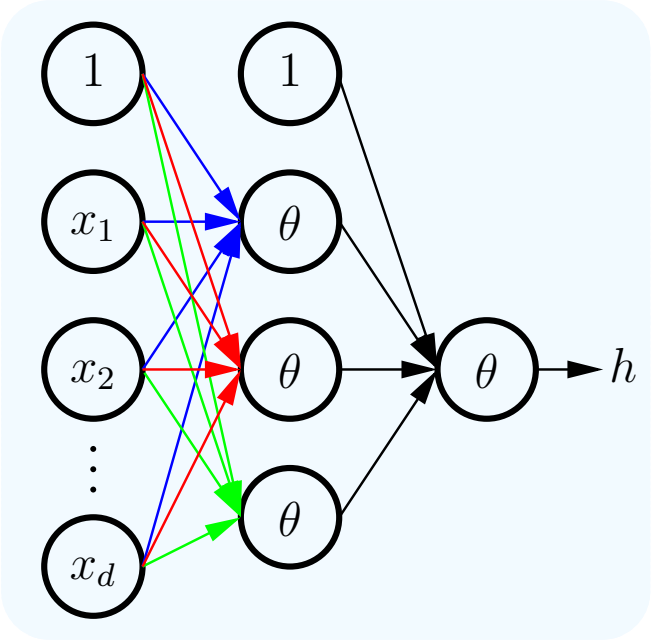
This is a form of regularization: not allowing the algorithm to to fully explore the hypothesis space reduces the effective VC dimension

How many hidden layers? (Section 7.3)

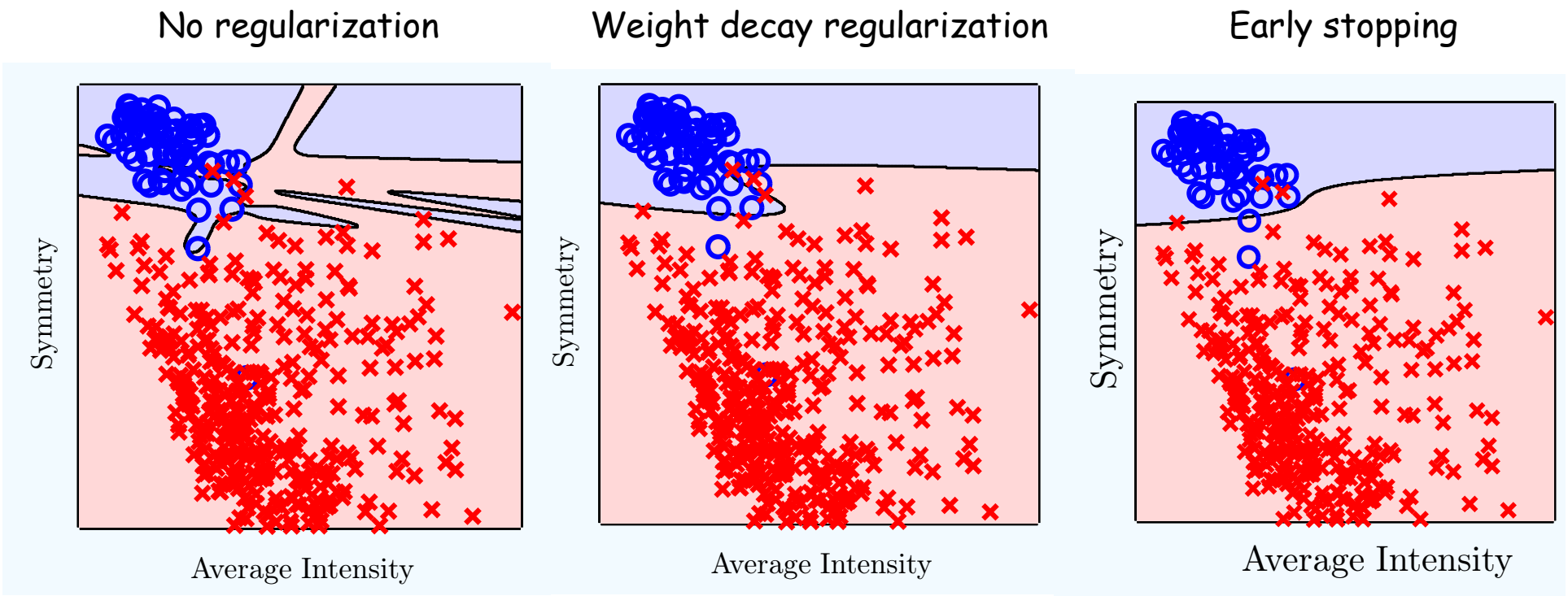
"Theorem": A neural network with a single hidden layer can approximate any function arbitrarily well.

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

But more layers can potentially do so more efficiently!



Comparison on the digit recognition data
(Section 7.4.3)



All networks are with 10 hidden units

	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%

Figures from section 7.4.3 in chapter e-7

Multi-class problems

How would you use a neural network to solve a multi-class classification problem with c classes?

Train a network with c output units.

Encode the labels using the "one-of-c" encoding: a vector with one element equal to 1 and the rest 0.

Outputs as probabilities

When using a logistic activation function for a single-output network, it can be interpreted as a probability.

With multiple output units the individual values won't sum to 1.

The fix is to use the softmax activation function, which is a generalization of the logistic function that normalizes outputs to sum to 1.

Neural networks in practice

Training a neural network requires a lot of decisions:

- Number of layers
- Activation function
- Number of hidden units per layer
- Learning rate
- Regularization parameter
- Initialization
- Number of epochs for training

Not feasible to perform grid search!

Neural networks vs SVMs

SVMs – no local minima; more scalable; easier model selection

Both require normalization/standardization for optimal performance

So why use NNs?