Artificial Neural Networks Part II

Machine Learning

Cesare Alippi, Jürgen Schmidhuber

Michael Wand, Paulo Rauber

TAs: Róbert Csordás, Krsto Proroković, Xingdong Zuo, Francesco Faccio, Louis Kirsch

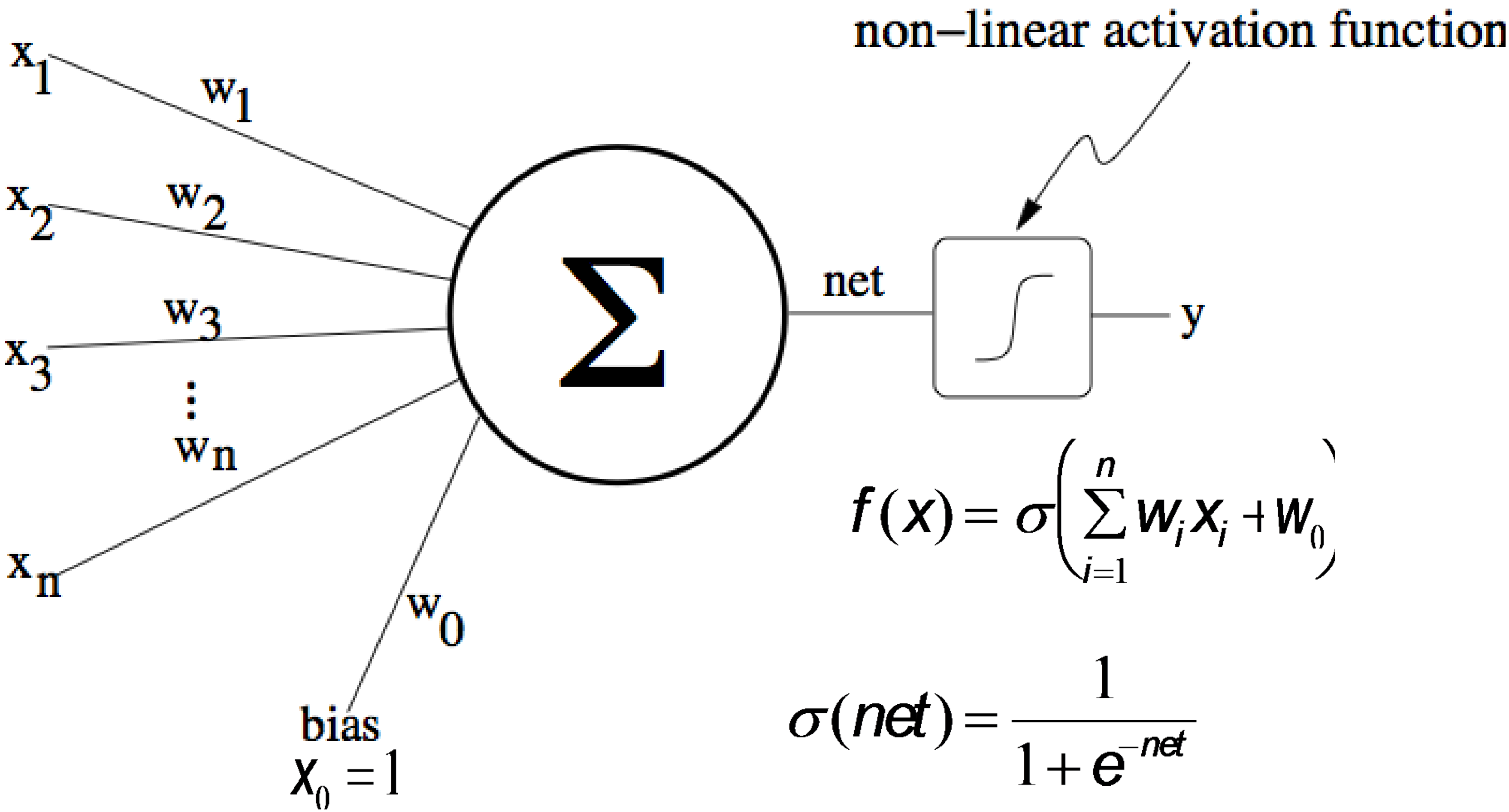
Calculus review

Chain rule:

$$(f \circ g)'(x) = f'(g(x))g'(x)$$

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Non-Linear Neuron



$$f(x) = \sigma\left(\sum_{i=1}^{n} w_i x_i + w_0\right)$$

$$\sigma(net) = \frac{1}{1 + e^{-net}}$$

Delta Rule (gradient descent) for **non-linear** neuron

- Calculate squared error between output and target

$$E = \frac{1}{2}(d - \sigma(net))^2 = \frac{1}{2}(d - \sigma(\sum_{i=1}^{n} w_i x_i))^2$$

Same as linear case but now with the sigmoid function squashing *net*

- Obtain partial derivative of error with respect to each weight

$$y = \sigma(net) \qquad \frac{\partial E}{\partial w_i} = \frac{\partial \frac{1}{2}(d-y)^2}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = (y - d)\frac{\partial y}{\partial net}\frac{\partial net}{\partial w_i}$$

Delta Rule (gradient descent) for **Non-linear** neuron

Recall for linear case

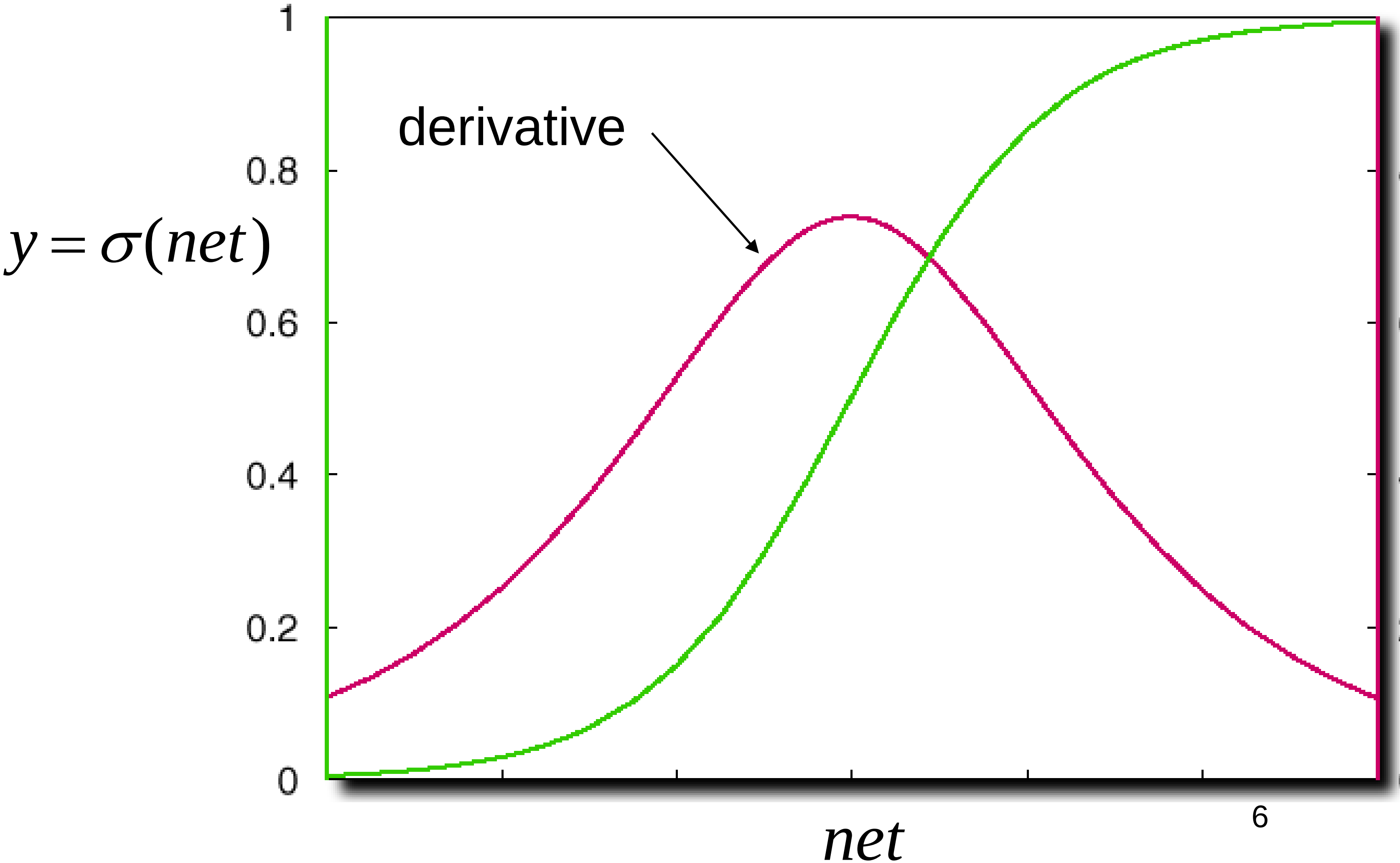
$$\frac{\partial E}{\partial w_i} = (net - d)\frac{\partial net}{\partial w_i}$$

But now we have to deal with the sigmoid

$$\frac{\partial E}{\partial w_i} = (y - d)\frac{\partial y}{\partial net}\frac{\partial net}{\partial w_i}$$

$$\frac{\partial \mathrm{y}}{\partial \mathrm{net}} = \frac{\partial \sigma(\mathrm{net})}{\partial \mathrm{net}} = \sigma(\mathrm{net})(1 - \sigma(\mathrm{net}))$$

Sigmoid and its derivative



Derivative of the Sigmoid

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial}{\partial x} \frac{1}{1+e^{-x}}$$

$$= (\frac{1}{1+e^{-x}})^2 e^{-x}$$

$$= (\frac{1}{1+e^{-x}})(\frac{1}{1+e^{-x}})(e^{-x})$$

$$= (\frac{1}{1+e^{-x}})(\frac{e^{-x}}{1+e^{-x}})$$

$$= \sigma(x)(1-\sigma(x))$$

Linear vs. non-linear gradient

Linear case:

$$\frac{\partial E}{\partial w_i} = (net - d)\frac{\partial net}{\partial w_i} = (net - d)x_i$$

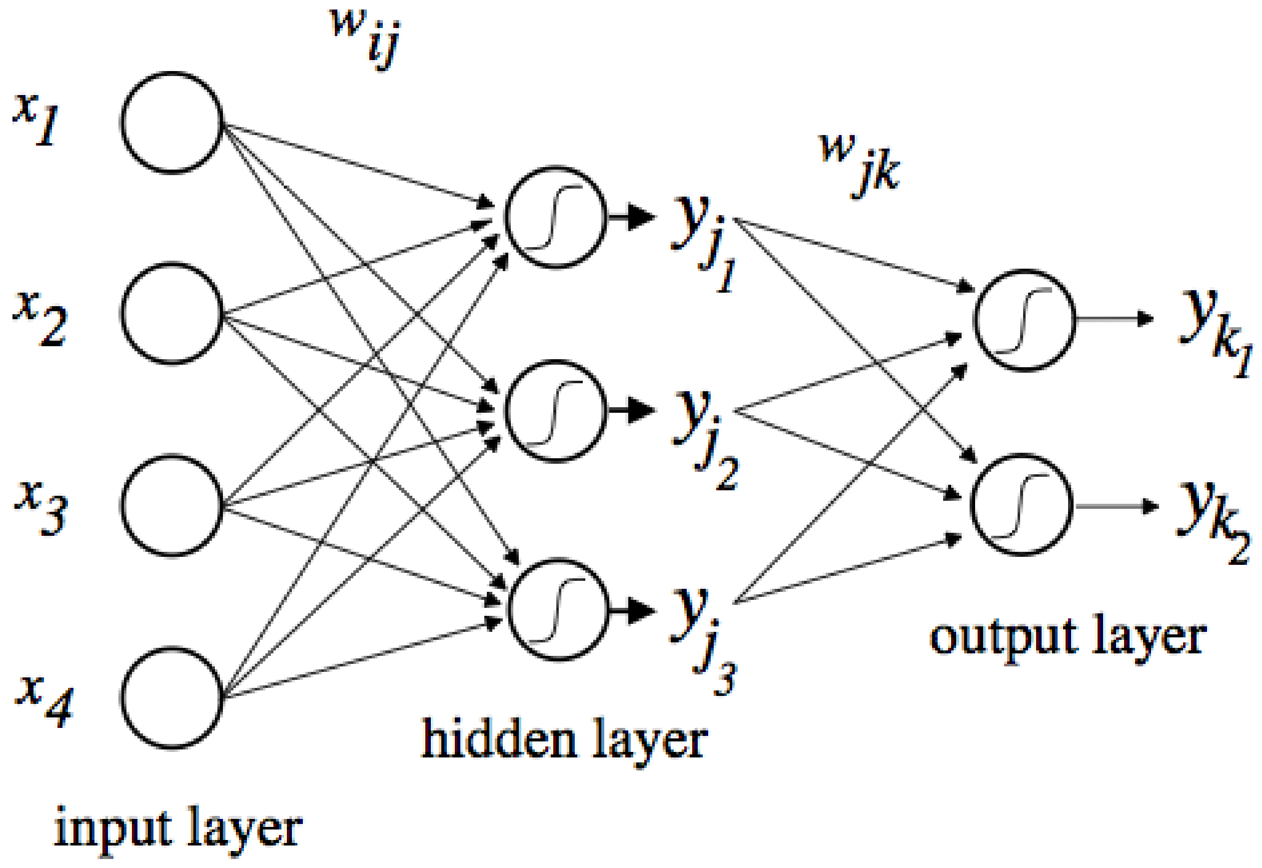
Non-linear case:

$$\frac{\partial E}{\partial w_i} = (y - d)\frac{\partial y}{\partial net}\frac{\partial net}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = (y - d)\overbrace{\sigma(net)(1 - \sigma(net))}\frac{\partial net}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = (y - d)\sigma(net)(1 - \sigma(net))x_i$$

Multi-layer Perceptron (MLP)



Choice of activation functions

- Hidden layers: sigmoidal functions
  - e.g. logistic sigmoid, tanh $(e^a-e^{-a})/(e^a+e^{-a})$
- Output layer:
  - Regression: identity function $y_k=net_k$
  - Binary classification: logistic sigmoid
  - Multiclass: softmax $\dfrac{\exp(a_k)}{\sum_j \exp(a_j)}$

Basic idea

- Attribute "blame" or "credit" to the weights from input to hidden layer, that contribute *indirectly* to the outputs.
- We can use the chain rule for this.

Backpropagation algorithm

Two steps:

1. Forward Pass: present training input pattern to network and activate network to produce output (can also do in batch: present all patterns in succession)

2. Backward Pass: calculate error gradient and update weights starting at output layer and then going back

Forward Pass

- Calculate *activation* of each hidden node and store them

$$y_j = \sigma\left(\sum_{i=1}^{n} w_{ij} x_i + w_{0j}\right)$$

- Then calculate *activation* of each output node

$$y_k = \sigma\left(\sum_{j=1}^{n} w_{jk} y_j + w_{0k}\right)$$

Backward Pass

1. Calculation of gradient

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}} = \delta_j y_i$$

$$\delta_j = \frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial y_j}\, f'(\mathrm{net}_j)$$

- For the output layer $\frac{\partial E}{\partial y_k} = (y_k - d_k)$
- For hidden layers we use the chain rule:

$$\frac{\partial E}{\partial y_j} = \sum_{k \in layer-after} \frac{\partial E}{\partial net_k}\frac{\partial net_k}{\partial y_j} = \sum_k \delta_k w_{jk}$$

Backward Pass for *output* node

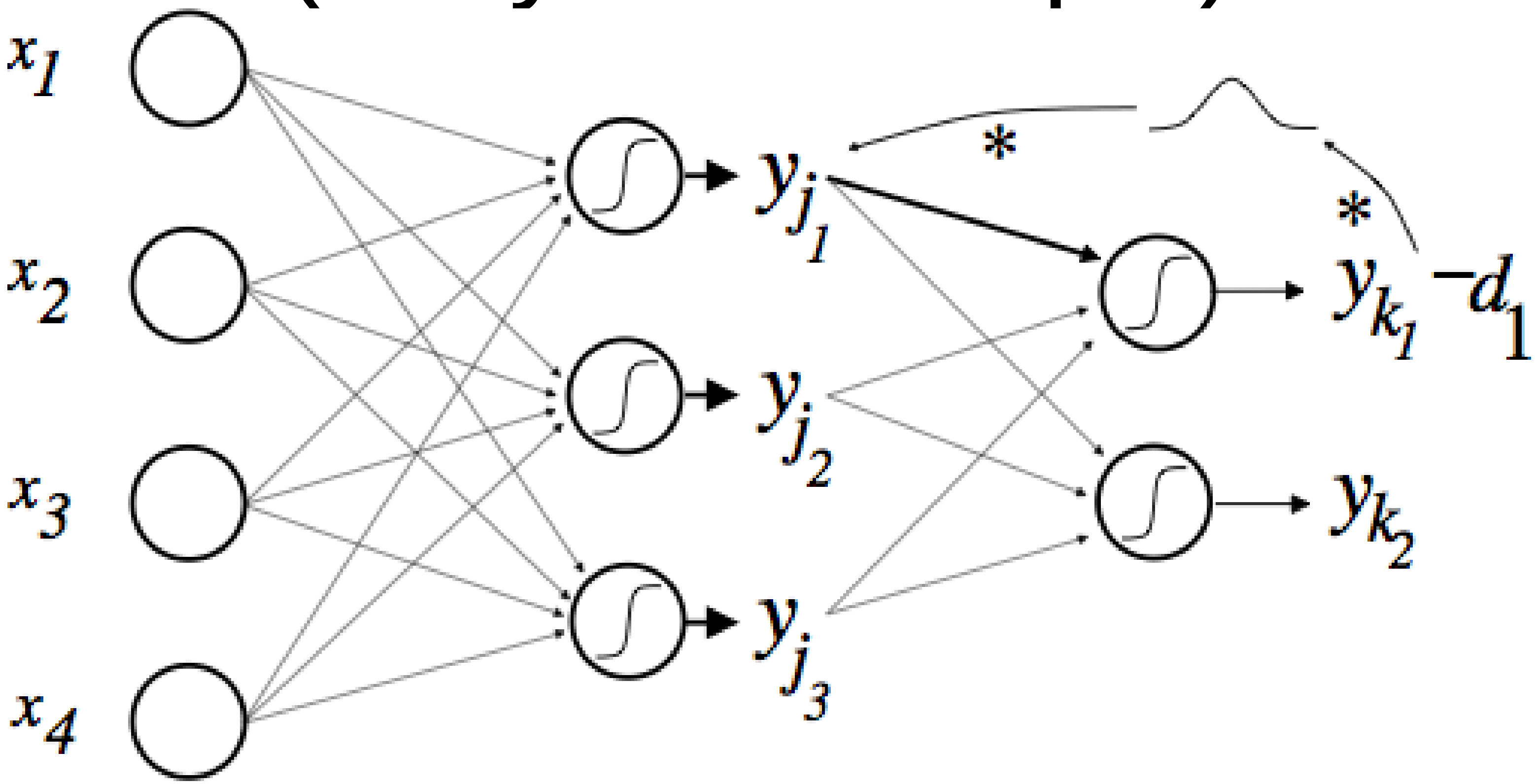
2. We update the weights:

$$w^{new}{}_{ij} = w^{old}{}_{ij} - \Delta w_{ij}$$

Where ,

$$\Delta \mathrm{w}_{ij} = \eta \mathrm{y}_{i} \delta_{j}$$

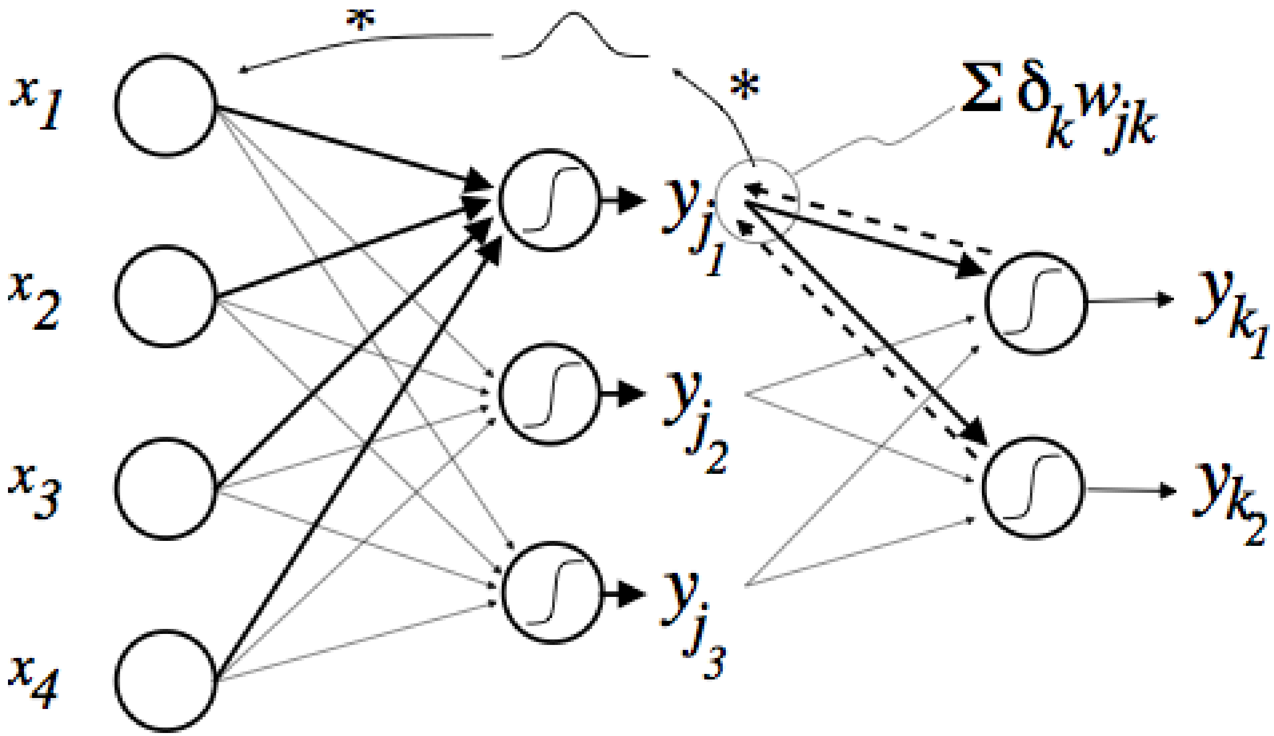
Updating output layer weight (3 layers example)



$w_{j_1k_1}$ only affects output y_{k_1}

$$\Delta w_{j_1k_1} = \eta y_{j_1}\delta_k = \eta y_{j_1}(y_{k_1} - d_1)\frac{\partial y_{k_1}}{\partial net_{k_1}}$$

Updating *hidden* layer weight (3 layers example)



Weight in input layer affects all outputs!

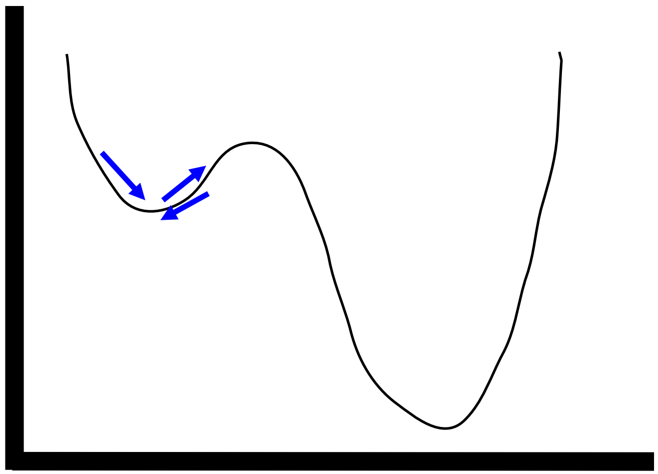
$$\Delta w_{ij} = \eta x_1 \delta_{j_1} = \eta x_1 \sum_{k \in output-layer} \delta_k w_{j_1 k} \frac{\partial y_{j_1}}{\partial net_{j_1}}$$

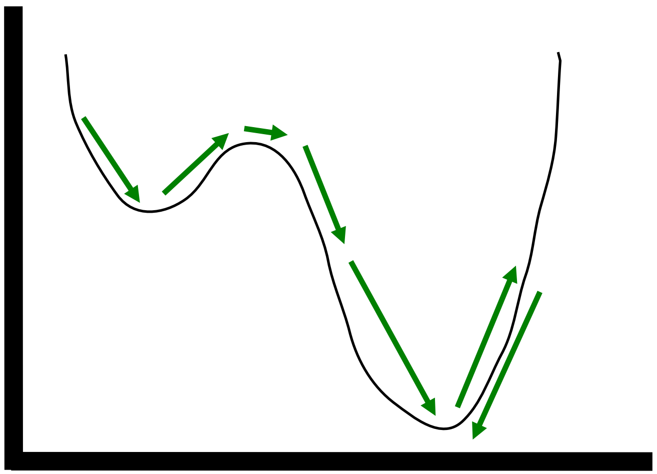
Local Minima

- Many dimensions make for many descent options
- Some believe that local minima are more common with very simple/toy problems, more rare with larger problems and larger nets
- If there are local minima problems, simply train multiple nets and pick the best
- Some algorithms add noise to the updates to escape from local minima

Enhancements To Gradient Descent

- Momentum
  - Adds a percentage of the last movement to the current movement





Backpropagation Learning Algorithm

- Until (low error or other stopping criteria) do
  - Present a training pattern
  - Calculate the error of the output nodes (based on $d_j - y_j$)
  - Calculate the error of the hidden nodes (based on the error of the output nodes which is propagated back to the hidden nodes)
  - Continue propagating error back until the input layer is reached
  - Update all weights based on the standard delta rule

Backpropagation Learning Equations

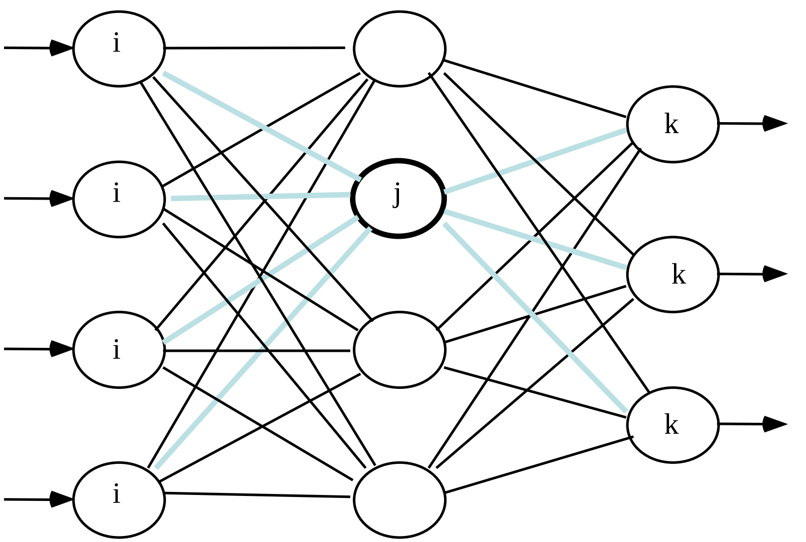
$$\Delta w_{ij} = \eta \delta_j y_i$$

$$\delta_j = (y_j - d_j) f'(net_j)$$

Output node, (j=k in 3 layers example)

$$\delta_j = \sum_{k \in layer-after} \delta_k w_{jk} f'(net_j)$$

Hidden nodes (j=i in 3 layers example)



Hidden Nodes

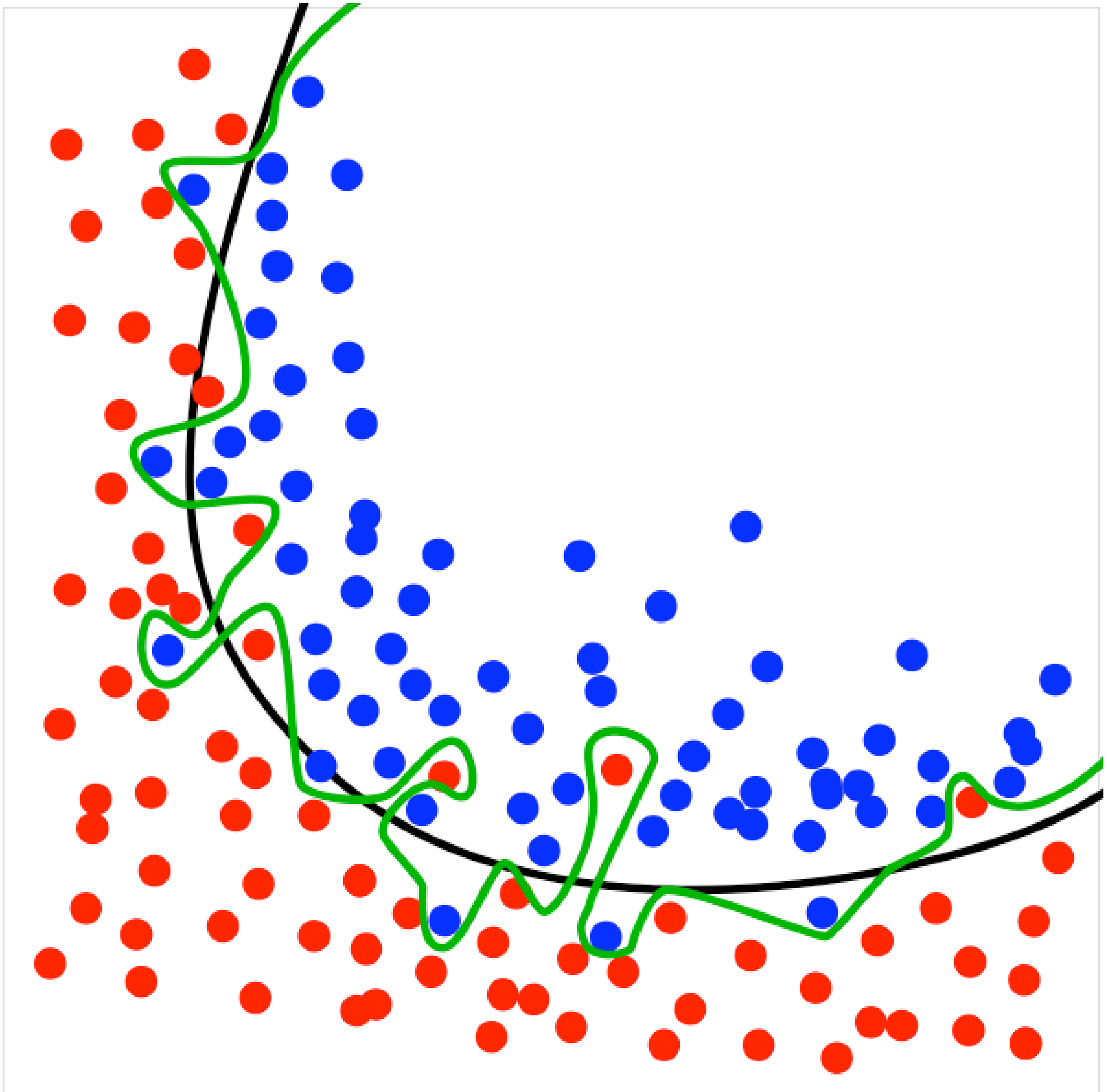
- Typically one fully connected hidden layer. Common initial number is 2*n* or 2log*n* hidden nodes where *n* is the number of inputs
- In practice train with a small number of hidden nodes, then keep doubling, etc. until no more significant improvement on test sets
- Hidden nodes discover new higher order features which are fed into the output layer

Enhancements To Gradient Descent

- Momentum
  - Useful to get over small bumps in the error function
  - Often finds a minimum in less steps
    - $w_{new} = -\eta\delta y + \alpha w_{old}$
    - **w** is the change in weight
    - **η** is the learning rate
    - **δ** is the error
    - **y** is different depending on which layer we are calculating
    - **α** is the momentum parameter

Overfitting / underfitting

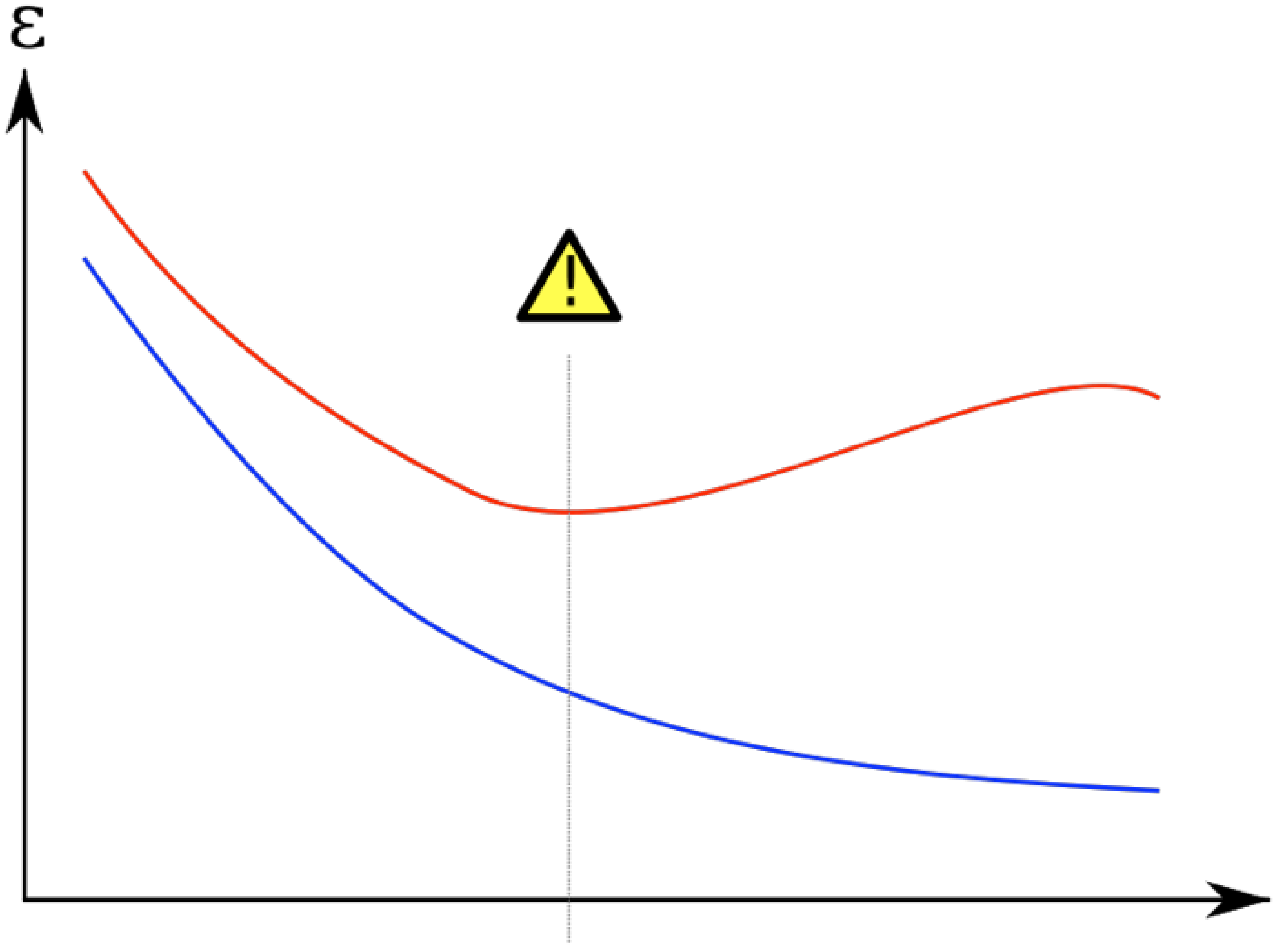
- Underfitting: the model performs badly even on the training set (high error)
- Overfitting: The model performs too nicely on the training set but not so well on the test one (noise and signal fitted)
- The following affects overfitting/underfitting: learning rate, number of hidden neurons, number of hidden layers, number of epochs...



Improving generalization 1

- Select a good network architecture (can e.g. be optimized by evolution - more on this later)
- Stop training when generalization gets worse, even if error on the training set still goes down! (*"early stopping"*)

To implement this, use a separate validation set, and test periodically


ε

Improving generalization 2

- Cross-validation: select new “validation sets” continuously
- Regularization: penalize large weights for smoother curve fitting