Multilayered Perceptrons

Classification of non-linearly separable patterns

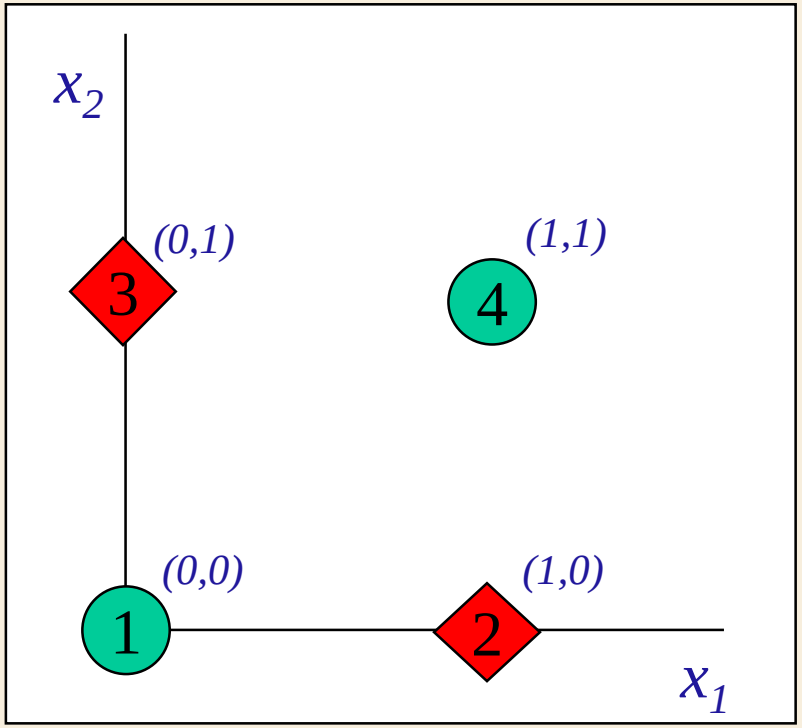
- A single-layered perceptron can achieve linear separation of patterns
- In the late 1960s the prevailing view on whether solution of non-linearly separable problems is practical was pessimistic (following criticism by Minsky and Papert).
- It is necessary to show that:

➢ There exist architectures capable of non-linear separation of patterns

➢ There exist effective training algorithms capable of delivering the synaptic weights needed to achieve non-linear separation of patterns in the general case.

Classification of non-linearly separable patterns – The XOR problem (1)

A/A	x_1	x_2	Class
1	0	0	0
2	1	0	1
3	0	1	1
4	1	1	0

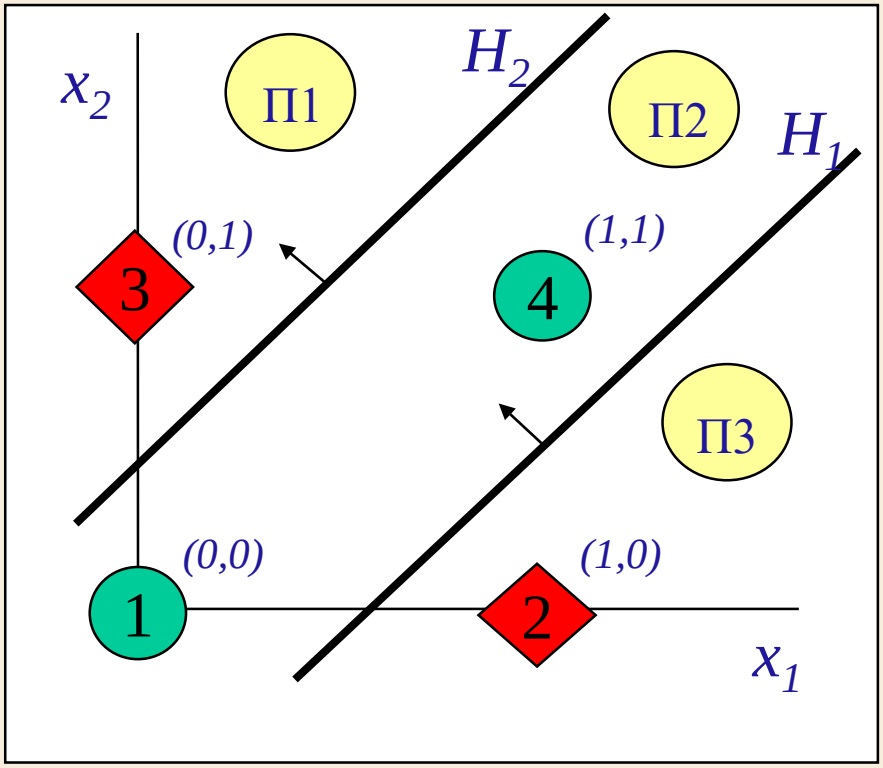


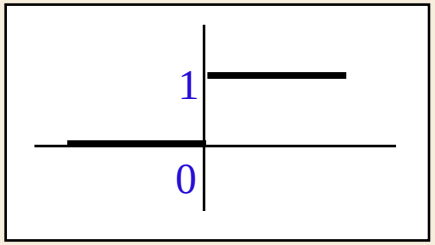
- Linear separation of the classes with a single-layered perceptron is impossible
- Proof: System of 4 linear inequalities has no solution

Classification of non-linearly separable patterns – The XOR problem (2)

Let us try to construct a solution:

- We need 2 hidden neurons (with intermediate outputs z_1 και z_2) that separate the plane of the 3 original inputs into 3 regions Π1, Π2, Π3.
- Consequently, we map the pattens in a new 2-dimensional space.
- For example, we wish patterns 1 και 4 to yield the same output. This will render the problem linearly separable in the new 2-dimensional space!

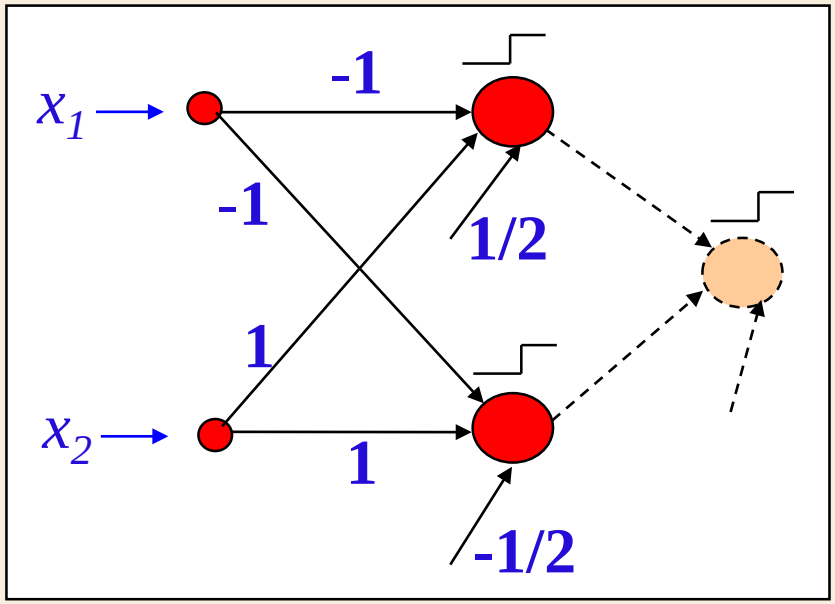


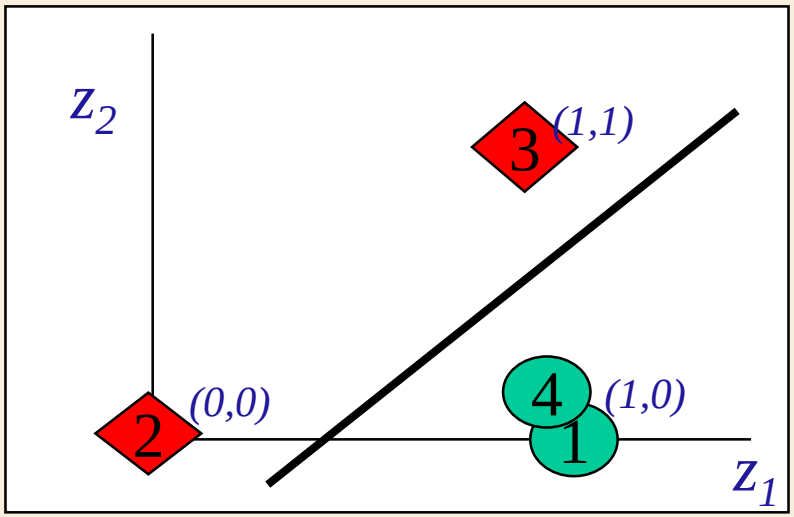


$$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Region	z_1	z_2
Π1	1	1
Π2	1	0
Π3	0	0

Classification of non-linearly separable patterns – The XOR problem (3)





From the equations of the separating lines H_1 and H_2

$$-x_1 + x_2 + \tfrac{1}{2} = 0, \quad -x_1 + x_2 - \tfrac{1}{2} = 0$$

we can read off the synaptic weights of the hidden layer:

$$z_1 = f\left[(-1)x_1 + (+1)x_2 + \tfrac{1}{2}\right]$$

$$z_2 = f\left[(-1)x_1 + (+1)x_2 - \tfrac{1}{2}\right]$$

$$\text{Pattern 1}: z_1 = f\left[(-1)0 + (+1)0 + \tfrac{1}{2}\right] = f\left[\tfrac{1}{2}\right] = 1$$

$$z_2 = f\left[(-1)0 + (+1)0 - \tfrac{1}{2}\right] = f\left[-\tfrac{1}{2}\right] = 0$$

$$\text{Pattern 2}: z_1 = f\left[(-1)1 + (+1)0 + \tfrac{1}{2}\right] = f\left[-\tfrac{1}{2}\right] = 0$$

$$z_2 = f\left[(-1)1 + (+1)0 - \tfrac{1}{2}\right] = f\left[-\tfrac{3}{2}\right] = 0$$

$$\text{Pattern 3}: z_1 = f\left[(-1)0 + (+1)1 + \tfrac{1}{2}\right] = f\left[\tfrac{3}{2}\right] = 1$$

$$z_2 = f\left[(-1)0 + (+1)1 - \tfrac{1}{2}\right] = f\left[\tfrac{1}{2}\right] = 1$$

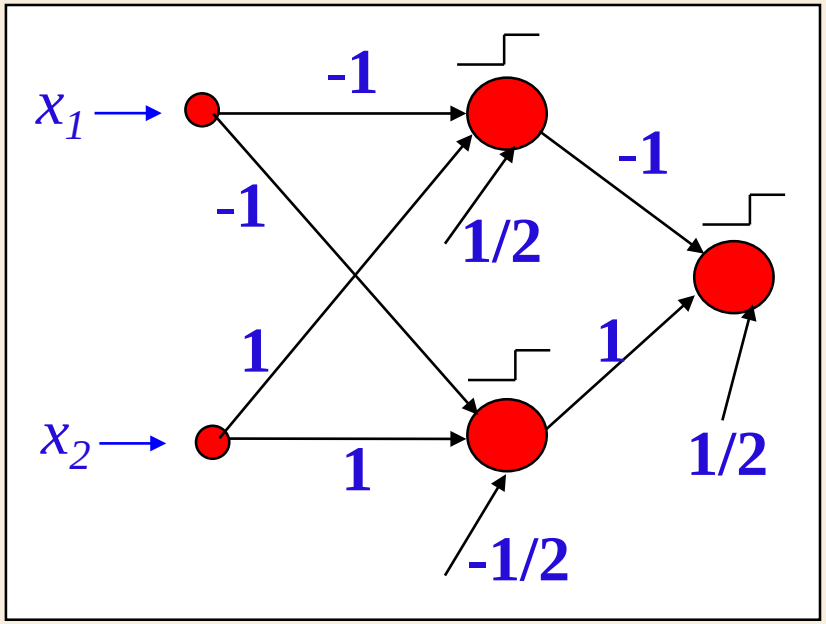
$$\text{Pattern 4}: z_1 = f\left[(-1)1 + (+1)1 + \tfrac{1}{2}\right] = f\left[\tfrac{1}{2}\right] = 1$$

$$z_2 = f\left[(-1)1 + (+1)1 - \tfrac{1}{2}\right] = f\left[-\tfrac{1}{2}\right] = 0$$

The representations of the patterns in the hidden layer are linearly separable!

Classification of non-linearly separable patterns – The XOR problem (4)

A/A	z_1	z_2	Class
1	1	0	0
2	0	0	1
3	1	1	1
4	1	0	0



From the equation of the line separating the outputs of the hidden layer:

$$-z_1 + z_2 + \tfrac{1}{2} = 0$$

we can read off the synaptic weights of the output layer:

$$y = f\left[(-1)z_1 + (+1)z_2 + \tfrac{1}{2}\right]$$

$$\text{Pattern 1}: y = f\left[(-1)1 + (+1)0 + \tfrac{1}{2}\right] = f\left[-\tfrac{1}{2}\right] = 0$$

$$\text{Pattern 2}: y = f\left[(-1)0 + (+1)0 + \tfrac{1}{2}\right] = f\left[\tfrac{1}{2}\right] = 1$$

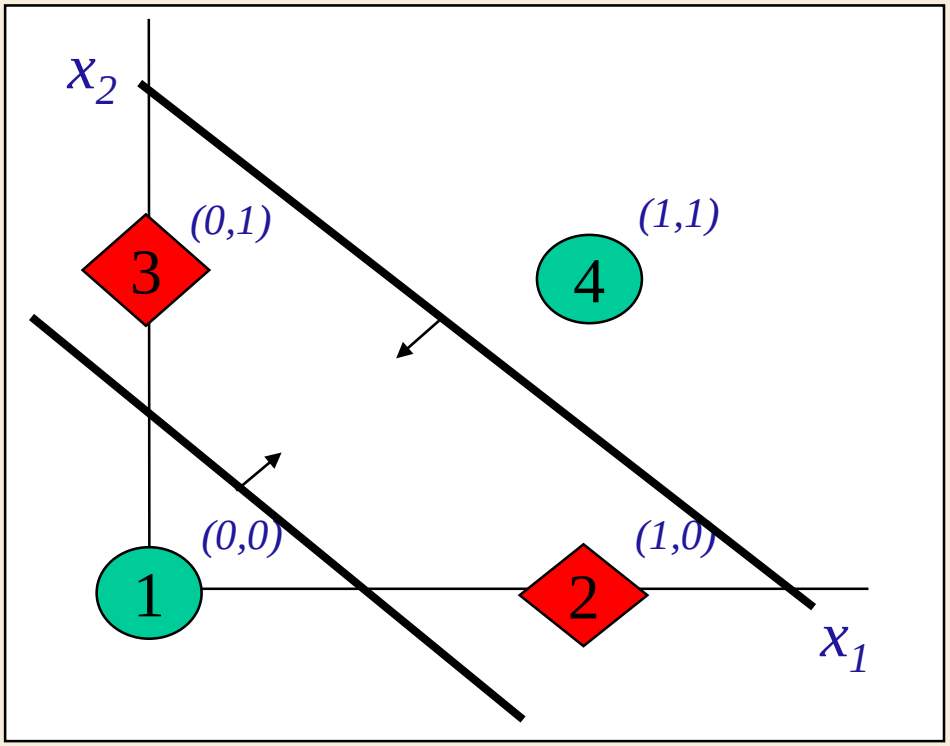
$$\text{Pattern 3}: z_1 = f\left[(-1)1 + (+1)1 + \tfrac{1}{2}\right] = f\left[\tfrac{1}{2}\right] = 1$$

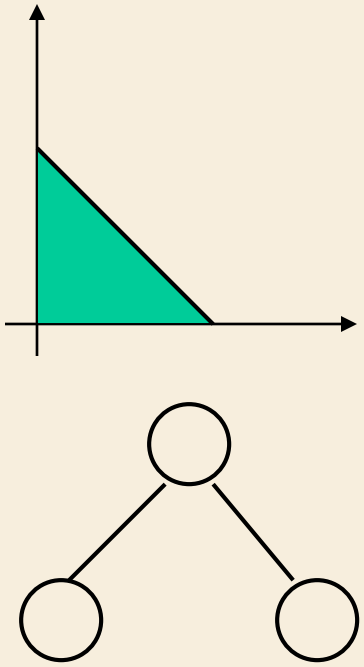
$$\text{Pattern 4}: y = f\left[(-1)1 + (+1)0 + \tfrac{1}{2}\right] = f\left[-\tfrac{1}{2}\right] = 0$$

Therefore, our network implements the XOR function!

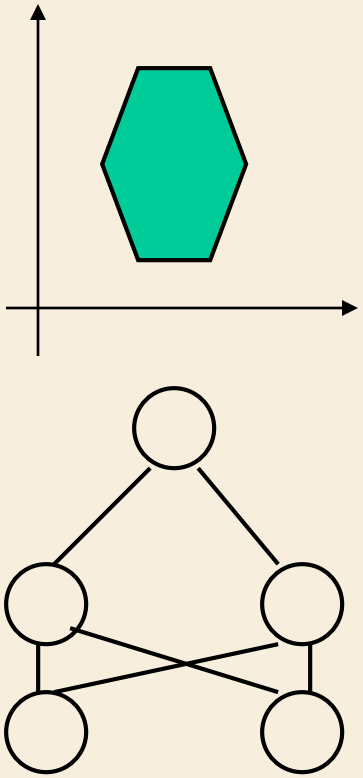
Classification of non-linearly separable patterns – The XOR problem (5)

EXERCISE: For the XOR problem, find another mapping of the inputs to the hidden layer neurons, using a different partition of the 2-dimensional input space, as shown in the diagram below. Draw the resulting network and find its synaptic weights and thresholds.

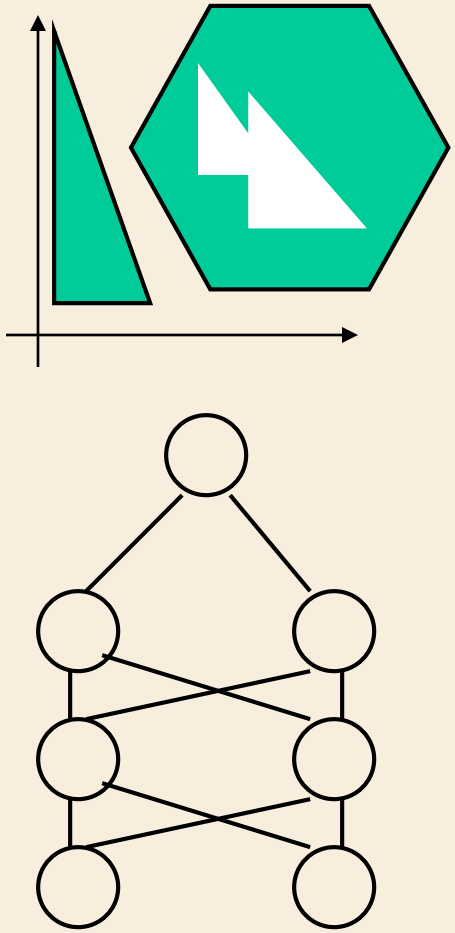




1st layer: Linear category boundaries

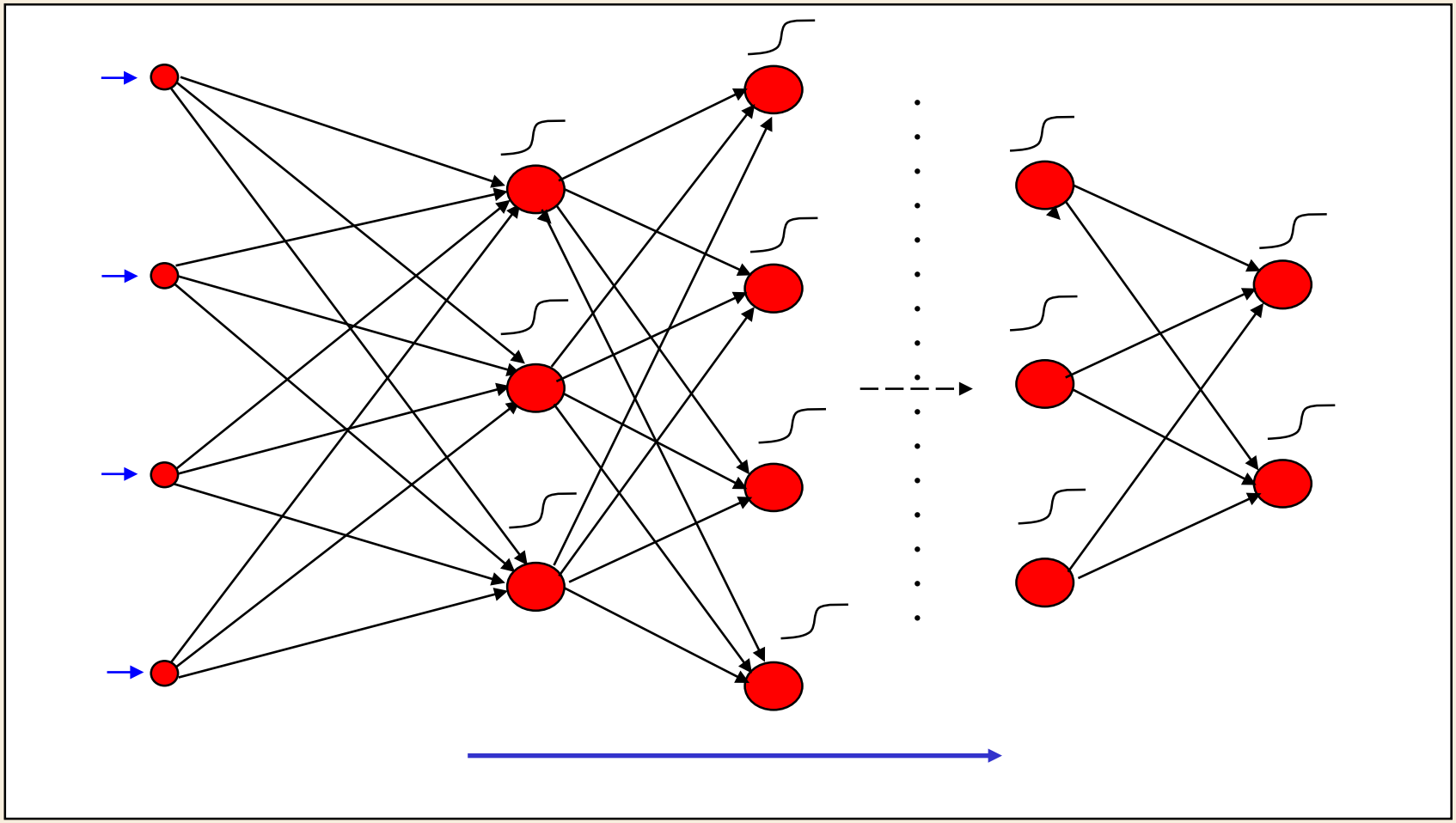


2nd layer: Convex piecewise linear bourndaries



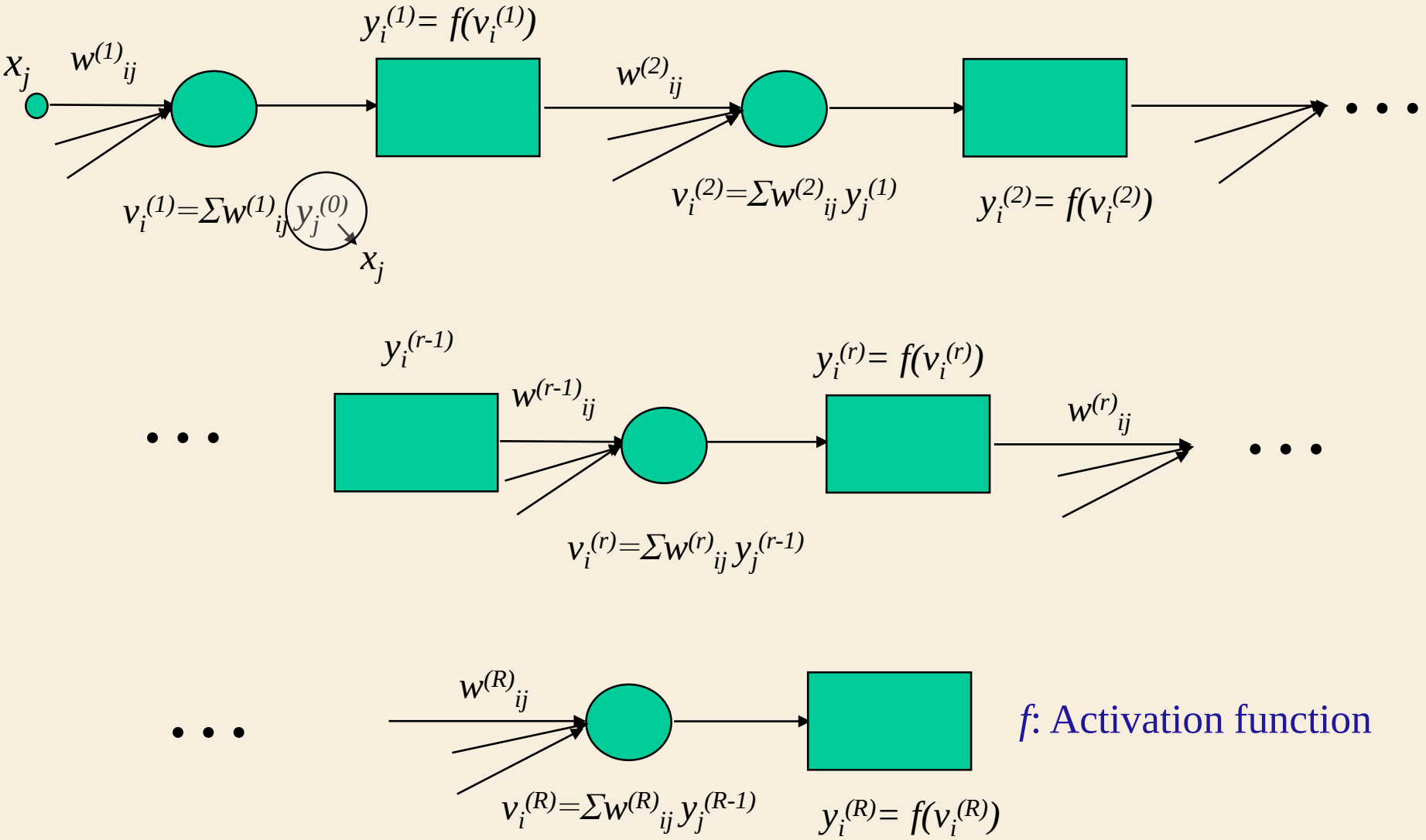
3rd layer: Any piecewise linear boundaries

Multi-layered Perceptron



Multi-layered Perceptron

This is a diagram showing the information flow in a perceptron with *R* layers



f: Activation function

* Note: Input to each neuron is augmented by a constant input component equal to +1 (to account for the presence of the threshold)

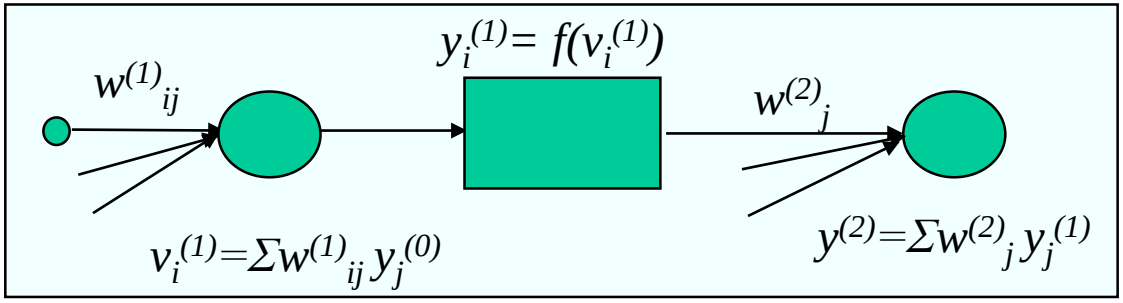
Universal approximation (1)

Question: The XOR problem can be solved with 2 layers of neurons. Can we solve a general non-linear classification or regression problem using multi-layered networks?

Answer: A two-layered perceptron suffices in principle, because it is a universal approximator, i.e.

- Let $g(\mathbf{x})$ be a continuous function of N variables, defined on a compact set $S \subset \mathbb{R}^N$.
- Let $f(t)$ be a non-constant, bounded and increasing as well as continuous function of a single variable.

Then given $\varepsilon>0$ there exists an integer $M=M(\varepsilon)$ and a two-layered perceptron with N inputs, M hidden neurons and one (linear) output neuron:



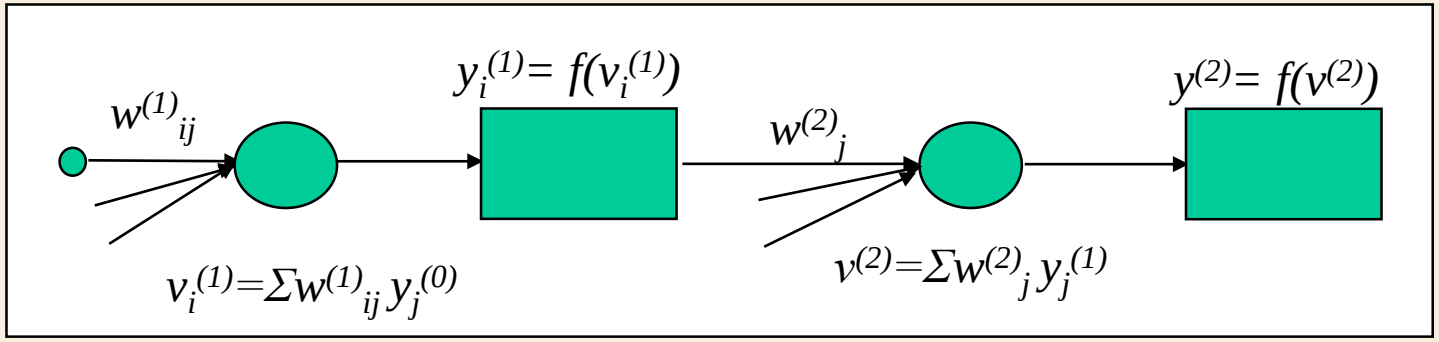
so that the following inequality holds: $\left\| y^{(2)}(\mathbf{x}) - g(\mathbf{x}) \right\| < \varepsilon, \quad \forall \mathbf{x} \in S$

Universal approximation (2)

If the output neuron is non-linear, as in the following diagram, with the function f bounded:

$$L_1 \leq f \leq L_2$$

we can approximate any continuous function taking values between L_1 and L_2.



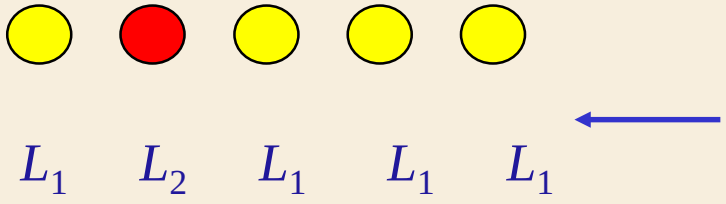
- Linear activation functions in the output layer are more convenient for solving regression problems.
- Non-inear activation functions in the output layer are more convenient for solving classification problems.

Desired outputs (1)

Classification problem with K classes:

Encoding of desired outputs:

1st method (diagonal encoding): K output neurons. The desired outputs for a pattern belonging to class number k (k=1,2,…,K) are all equal to L_1 except the k-th output, which is equal to L_2.



Diagonal encoding of the 2nd class in a problem of 5 classes

Desired outputs (2)

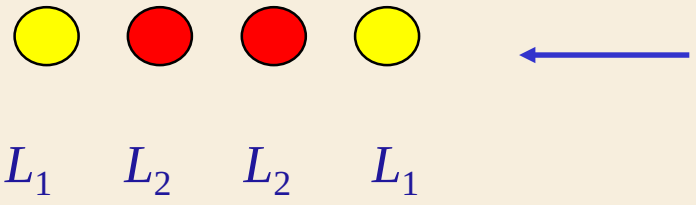
2nd method (binary encoding): Consider a pattern belonging to class k and the binary representation (a succession of 0s and 1s) using as many bits, as we need to represent K-1. The desired outputs are:

- L_1 wherever the binary representation is 0
- L_2 wherever the binary representation is 1

Example: Problem with 10 classes: We need 4 bits.

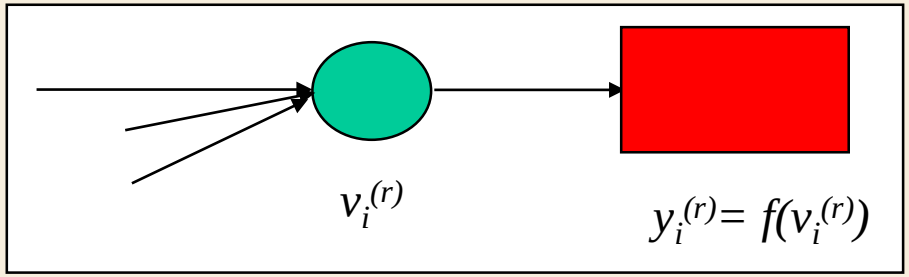
Representation of the 7th class: Binary representation of 7-1=6:

0 1 1 0.



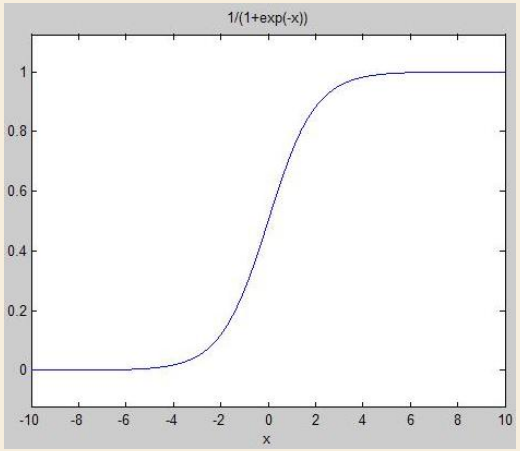
Binary representation of the 7ης class in a 10-class problem

Activation functions



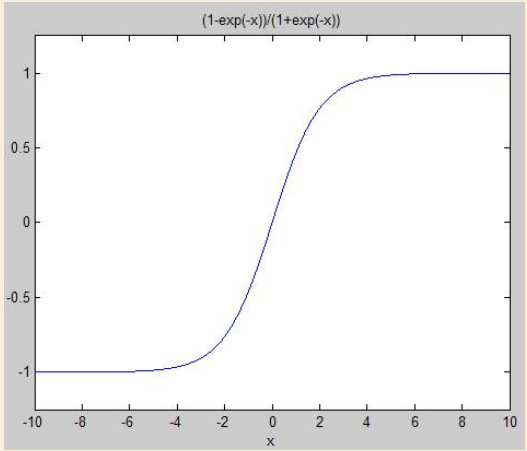
1st case:

$$f(x) = \frac{1}{1+\exp(-x)}, \qquad L_1 = 0, \quad L_2 = 1$$



2nd case:

$$f_2(x) = \tanh(x/2) = \frac{1-\exp(-x)}{1+\exp(-x)}, \qquad L_1 = -1, \quad L_2 = 1$$



The Backpropagation Algorithm (1)

Cost function:

Desired outputs

$$E = \frac{1}{2}\sum_{k\mu}\left(e_{k\mu}\right)^2 \qquad e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}$$

μ: Labels the patterns in the training set

GRADIENT DESCENT:

- We update the synaptic weights moving opposite to the gradient of E with respect to these weights:

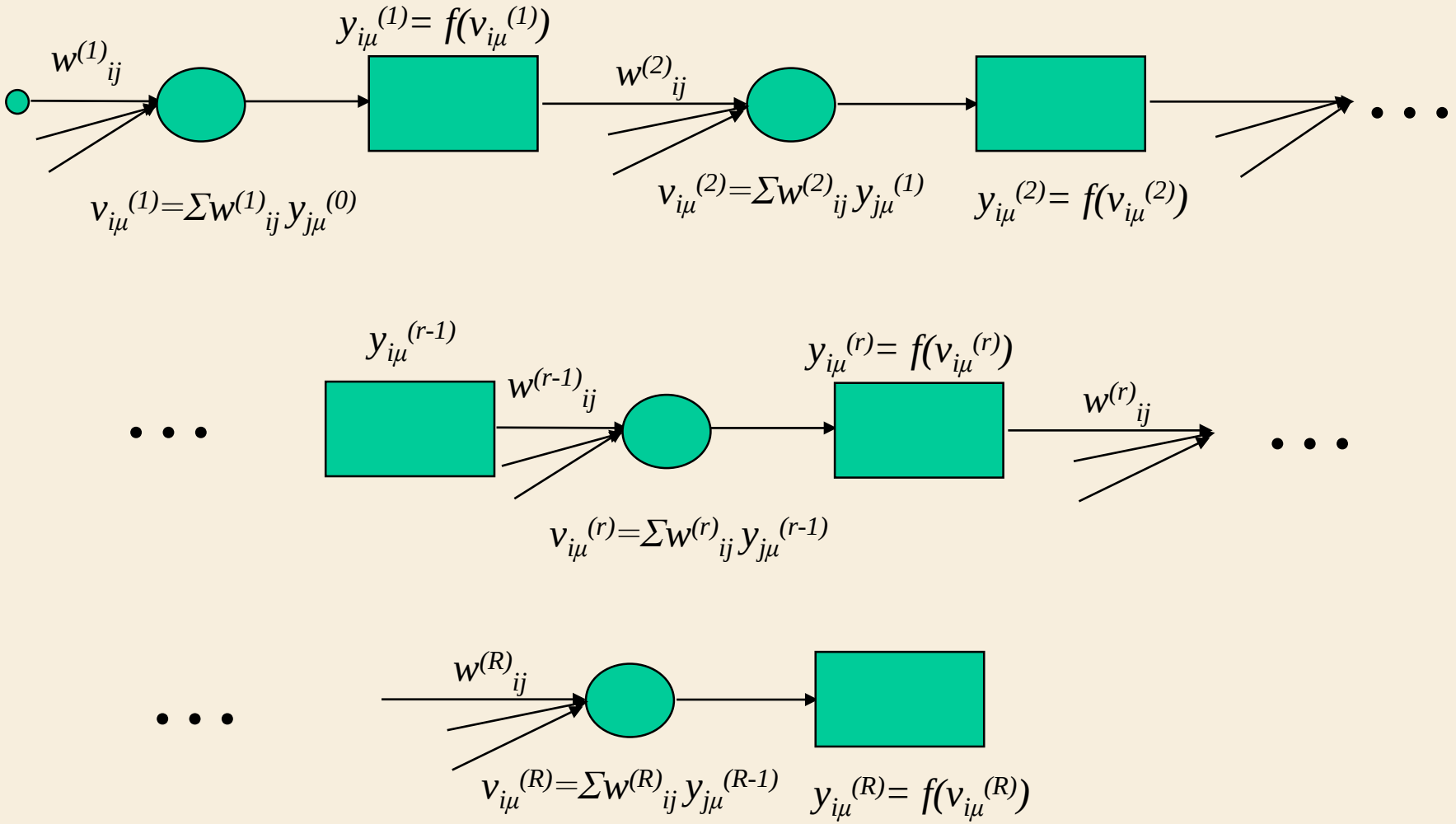
$$\delta \mathbf{w} = -\varepsilon \nabla E$$

- We need a systematic computationally efficient method of calculating the gradient.
- We organize our calculations by layer:

$$\delta w_{ij}^{(r)} = -\varepsilon \frac{\partial E}{\partial w_{ij}^{(r)}}$$

The Backpropagation Algorithm (2)

Information flow diagram (R layers of neurons)

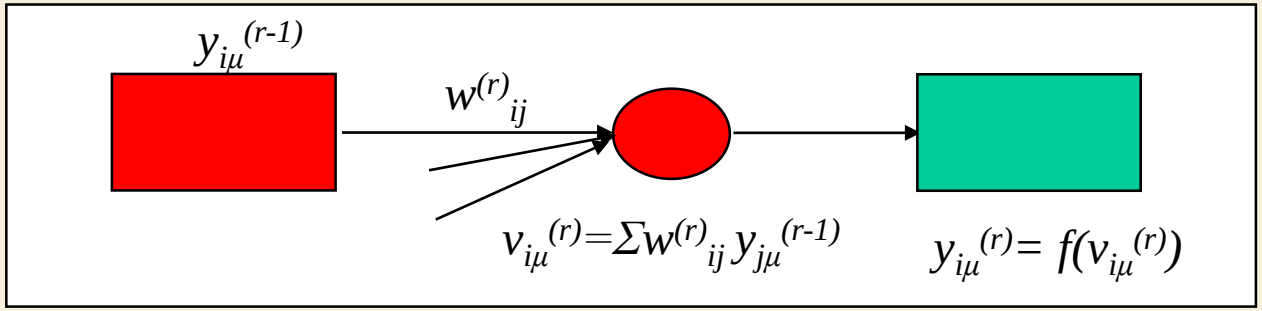


Calculation of derivatives: Extensive use of the chain rule:

$$\frac{\partial \phi(\sigma(x))}{\partial x} = \frac{\partial \phi}{\partial \sigma}\frac{\partial \sigma}{\partial x}$$

The Backpropagation Algorithm (3)

We focus on parts of the flow diagram (coloured red):



$$\frac{\partial E}{\partial w_{ij}^{(r)}} = \sum_{\mu} \frac{\partial E}{\partial v_{i\mu}^{(r)}} \frac{\partial v_{i\mu}^{(r)}}{\partial w_{ij}^{(r)}} = \sum_{\mu} \delta_{i\mu}^{(r)} y_{j\mu}^{(r-1)}$$

$$\delta_{i\mu}^{(r)} = \frac{\partial E}{\partial v_{i\mu}^{(r)}}$$

What have we achieved so far?

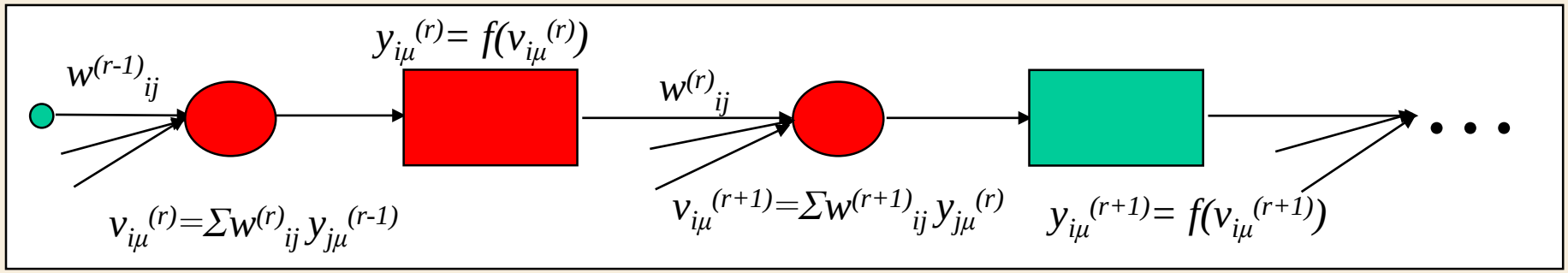
For a given layer *r* we have written the derivative of the cost function with respect to the weights as a sum of terms, with each term depending on:

- Already known local information, originating strictly from the previous layer.
- Information depending on the given layer and all subsequent layers, which is unknown to us and calls for additional calculations. Let us concentrate on these.

The Backpropagation Algorithm (4)

$$\delta_{i\mu}^{(r)} = \frac{\partial E}{\partial v_{i\mu}^{(r)}} = \sum_k \frac{\partial E}{\partial v_{k\mu}^{(r+1)}} \frac{\partial v_{k\mu}^{(r+1)}}{\partial v_{i\mu}^{(r)}} = \sum_k \delta_{k\mu}^{(r+1)} \frac{\partial v_{k\mu}^{(r+1)}}{\partial v_{i\mu}^{(r)}}$$

Look at the diagram:



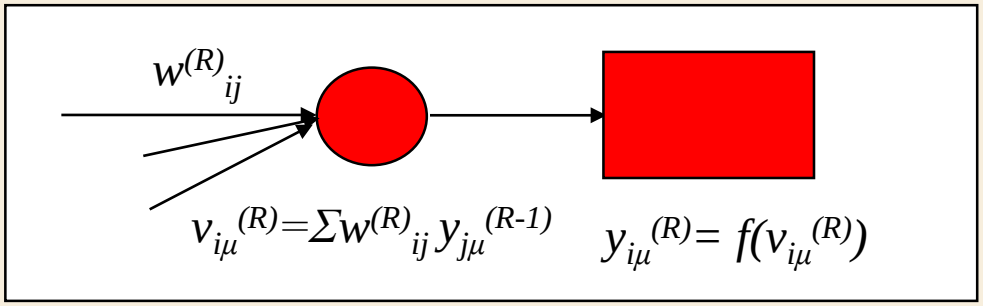
$$\frac{\partial v_{k\mu}^{(r+1)}}{\partial v_{i\mu}^{(r)}} = \frac{\partial v_{k\mu}^{(r+1)}}{\partial y_{i\mu}^{(r)}} \frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^{(r)}} = \frac{\partial v_{k\mu}^{(r+1)}}{\partial y_{i\mu}^{(r)}} f'(v_{i\mu}^{(r)}) = w_{ki}^{(r+1)} f'(v_{i\mu}^{(r)})$$

$$\delta_{i\mu}^{(r)} = \sum_k \delta_{k\mu}^{(r+1)} w_{ki}^{(r+1)} f'(v_{i\mu}^{(r)})$$

The Backpropagation Algorithm (5)

What have we achieved?

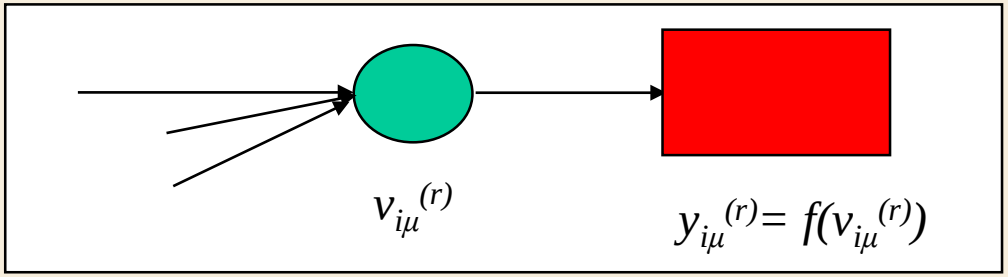
- We have reduced the unknown information from the layer r, to known information plus information that depends only on layer r+1 and subsequent layers.
- The deltas can be evaluated starting from the last layer and moving towards previous layers (backpropagation).
- To complete the picture, we only need to "initialize" the deltas, i.e. to calculate their values for the last layer *R*:



$$\delta_{i\mu}^{(R)} = \frac{\partial E}{\partial v_{i\mu}^{(R)}} = \frac{\partial E}{\partial y_{i\mu}^{(R)}} \frac{\partial y_{i\mu}^{(R)}}{\partial v_{i\mu}^{(r)}} = f'(v_{i\mu}^{(R)})(y_{i\mu}^{(R)} - T_{i\mu})$$

The Backpropagation Algorithm (6)

Calculation of activation function derivatives:



1st case:

$$f(x) = \frac{1}{1+\exp(-x)} \Rightarrow f'(x) = \frac{\exp(-x)}{\left[1+\exp(-x)\right]^2} = \frac{1}{1+\exp(-x)}\frac{\exp(-x)}{1+\exp(-x)} = f(x)\left[1-f(x)\right]$$

2nd case:

$$f_2(x) = \tanh(x/2) = \frac{1-\exp(-x)}{1+\exp(-x)} = 2\frac{1}{1+\exp(-x)} - 1 = 2f_1(x) - 1 \Rightarrow f_1(x) = \frac{1}{2}[1+f_2(x)]$$

$$f_2'(x) = 2f_1'(x) = 2f_1(x)[1-f_1(x)]$$

$$\Rightarrow f_2'(x) = \frac{1}{2}[1+f_2(x)][1-f_2(x)]$$

In both cases, the derivative is a function of the activation function itself.

The Backpropagation Algorithm (7)

Initialization: With small weights uniformly distributed between $-\alpha$ and α, where α is a small positive real number.

Iterative steps:

-Forward calculations:

Calculation of the outputs of every neuron in the network, starting from the first layer and progressing towards the output layer:

$$y_{i\mu}^{(r)} = f\left(\sum_k w_{ij}^{(r)} y_{j\mu}^{(r-1)}\right)$$

Cost function calculation: $E = \frac{1}{2}\sum_{k\mu}\left(e_{k\mu}\right)^2 \qquad e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}$

-Backward calculations:

Calculation of all deltas for each layer starting from the output layer R according to the relations:

$$\delta_{i\mu}^{(R)} = f'(v_{i\mu}^{(R)})(y_{i\mu}^{(R)} - T_{i\mu}) \qquad \delta_{i\mu}^{(r)} = \sum_k \delta_{k\mu}^{(r+1)} w_{ki}^{(r+1)} f'(v_{i\mu}^{(r)})$$

-Weight updates:

$$w_{ij}^{(r)}(\text{new}) = w_{ij}^{(r)}(\text{old}) + \delta w_{ij}^{(r)}, \quad \delta w_{ij}^{(r)} = -\varepsilon \sum_\mu \delta_{i\mu}^{(r)} y_{j\mu}^{(r-1)}$$

Termination:

- When the cost drops below a given threshold
- Or the norm of the cost function gradient drops below a given threshold
- *To avoid overfitting, apply termination criterion on validation set

The Backpropagation Algorithm (8)

REMARK:

For weight updates, all patterns μ are taken into account:

$$w_{ij}^{(r)}(\text{new}) = w_{ij}^{(r)}(\text{old}) + \delta w_{ij}^{(r)}, \quad \delta w_{ij}^{(r)} = -\varepsilon \sum_{\mu} \delta_{i\mu}^{(r)} y_{j\mu}^{(r-1)}$$

The update takes place after all patterns in the training set have been presented to the network (batch mode).

Variation: Update takes place after presenting each training pattern (incremental mode):

$$w_{ij}^{(r)}(\text{new}) = w_{ij}^{(r)}(\text{old}) + \delta w_{ij}^{(r)}, \quad \delta w_{ij}^{(r)} = -\varepsilon \delta_{i\mu}^{(r)} y_{j\mu}^{(r-1)}, \forall\ \mu$$

Weight Pruning – Regularization (1)

- How to decide about the number of weights? Too many weights mean unnecessary complexity for our model, which inevitably leads to overfitting.

- The usual practice is to start with a reasonably large number of weights and eliminate the least useful ones.

- Modified cost functions are used for training, which help push the least informative weights towards very low values.

- After training ends, these are altogether eliminated (pruned) and the pruned network is used for testing.

Weight Pruning – Regularization (2)

Weight decay: Minimize regularized cost function (as in ridge regression):

$$E'(\mathbf{w}) = E(\mathbf{w}) + \lambda \|\mathbf{w}\|^2$$

- It is better to use a different λ for each layer of weights.
- It is not good to include biases (thresholds) in the norm, as these have a distinct function to play in the network and it is not wise to eliminate them.

Weight elimination: Minimize modified cost function:

$$E'(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_k \frac{w_k^2}{w_k^2 + T^2}$$

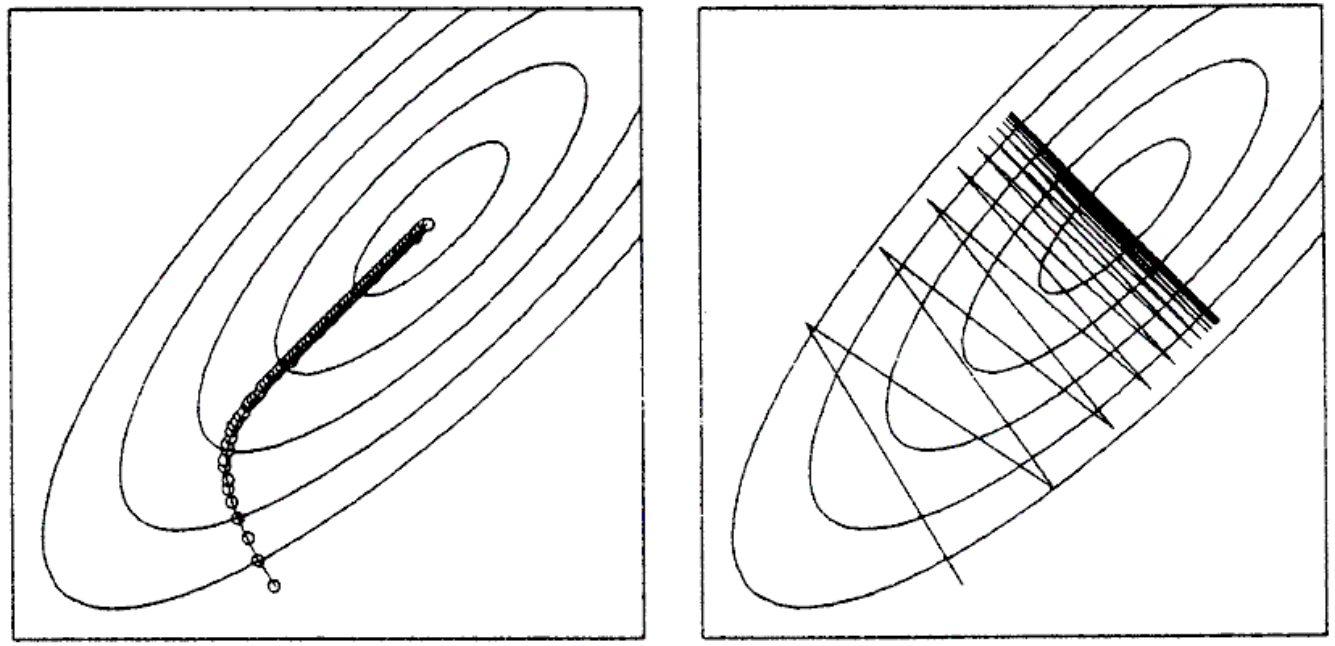
- When a weight drops below T, the corresponding regularizing term goes to zero fast. For weights much larger than T, the term is close to 1. Less significant weights are pushed towards zero.

1st order algorithms (1)

Gradient descent-back propagation:

$$\delta \mathbf{w} = -\varepsilon \nabla E$$

(All synaptic weights of the network are considered as components of a single weight vector)



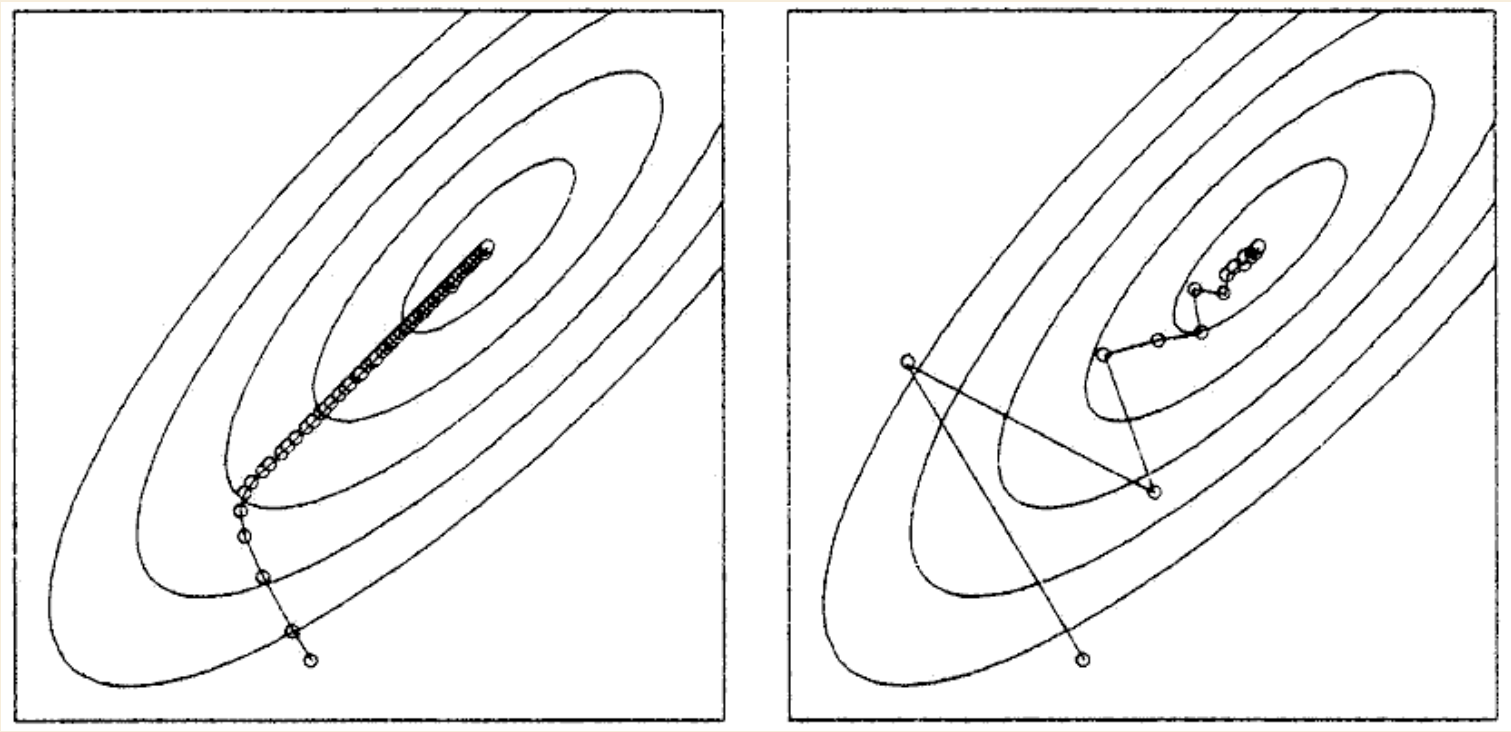
Small step ε: Very slow progress

Large step: Oscillations (zig-zags)

1st order algorithms (2)

Gradient descent with momentum:

$$\delta \mathbf{w} = -\varepsilon \nabla E + a \delta \mathbf{w}_{t-1}, \quad \delta \mathbf{w}_{t-1} = \mathbf{w} - \mathbf{w}_{t-1}$$



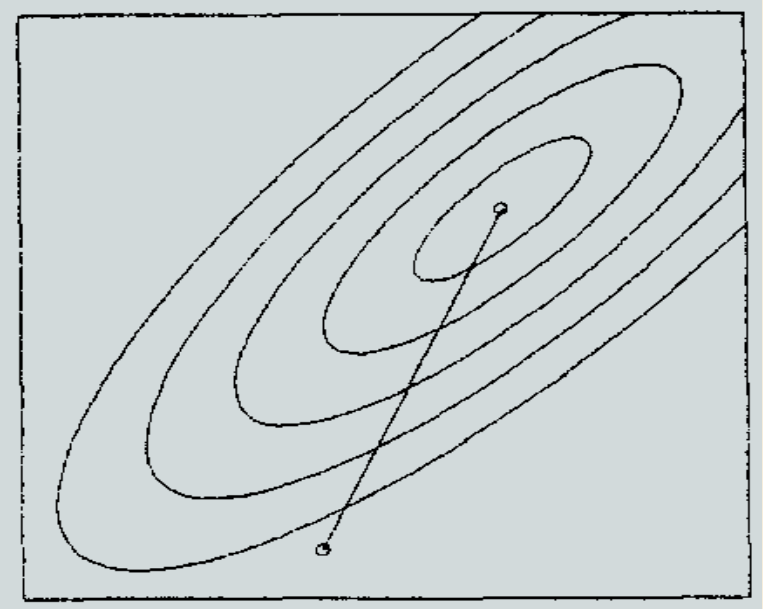
The second term brings about a partial alignment of the old and new weight updates, so that oscillations are somewhat diminished.

2nd order algorithms: Newton's method (1)

$$E(\mathbf{w}+\delta\mathbf{w}) = E(\mathbf{w}) + \nabla E(\mathbf{w})^T \delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^T [\nabla^2 E(\mathbf{w})]\delta\mathbf{w} + \ldots$$

$$\nabla E(\mathbf{w}+\delta\mathbf{w}) = \nabla E(\mathbf{w}) + \delta\mathbf{w}^T \nabla^2 E(\mathbf{w}) + \ldots$$

- We use information about the gradient and curvature. Assuming that the cost function is quadratic, we ignore terms of higher than second order
- We seek to find the minimum in a SINGLE step



$$\nabla E(\mathbf{w}+\delta\mathbf{w}) = \nabla E(\mathbf{w}) + \delta\mathbf{w}^T \nabla^2 E(\mathbf{w}) = 0$$

Newton step:

$$\delta\mathbf{w} = -(\nabla^2 E)^{-1} \nabla E$$

2nd order algorithms: Newton's method (2)

– Problems:

- Complexity of evaluating and inverting the Hessian Matrix

$$\mathbf{H} = \nabla^2 E$$

- To ensure local convexity of the cost function, **H** must be positive definite in order to ensure local convexity of the cost function (this does not hold necessarily)
- Convergence issues

– Simplification:

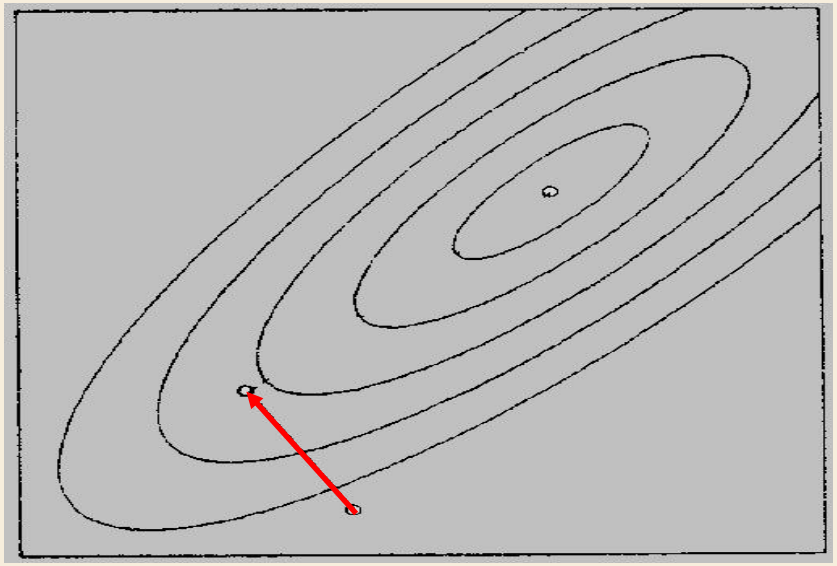
- The main question: Do methods exist, that use second order information without explicit evaluation and inversion of the Hessian?

2nd order algorithms: The conjugate gradient method (1)

We still assume that the cost function is quadratic in the weights:

$$E(\mathbf{w}+\delta\mathbf{w}) = E(\mathbf{w}) + \nabla E(\mathbf{w})^T \delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^T [\nabla^2 E(\mathbf{w})]\delta\mathbf{w}$$

We initialize the weight vector and pick a random direction for our first update. We choose the weight update vector $\delta\mathbf{w}$ so as to minimize the cost function along this direction.

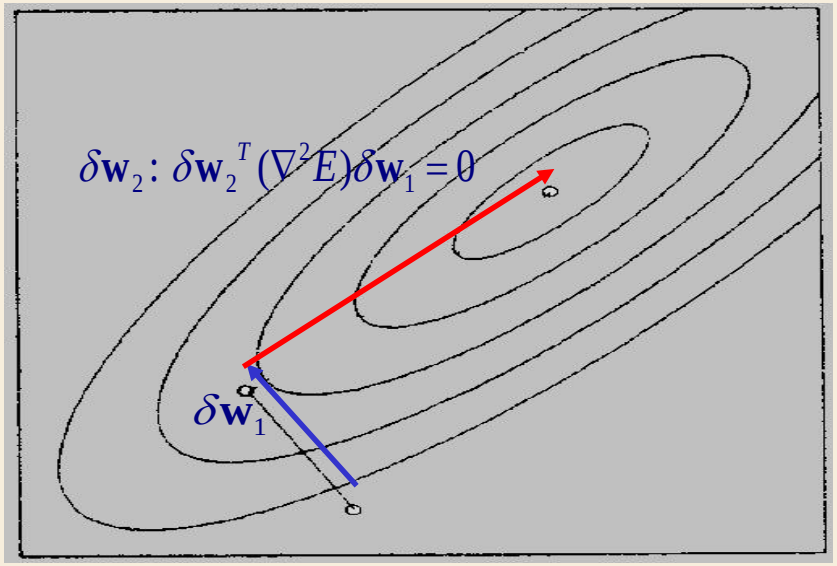


2nd order algorithms: The conjugate gradient method (2)

- Question: Is it possible to choose the successive directions so that the minimum of the cost function can be reached after a succession of line minimizations along each individual direction?
- Answer: Yes! We must choose directions that are conjugate with respect to the Hessian:

$$\delta \mathbf{w}^T (\nabla^2 E) \delta \mathbf{w}_{t-1} = 0$$

With this choice, and provided that the quadratic approximation is valid, it can be shown that the minimum of E can be found in a number of steps equal to the total number of weights.



2nd order algorithms: The conjugate gradient method (3)

- Question: Can we evaluate the successive weight updates without explicitly evaluating the Hessian?
- Answer: Yes! The weight update rule is:

$$\delta \mathbf{w} = -\nabla E + \beta_t \delta \mathbf{w}_{t-1}$$

where β_t can be evaluated using any of the following formulas:

$$\beta_t = \frac{\|\nabla E(\mathbf{w})\|^2}{\|\nabla E(\mathbf{w}_{t-1})\|^2}$$ (Fletcher-Reeves method)

$$\beta_t = \frac{\nabla E(\mathbf{w})^T [\nabla E(\mathbf{w}) - \nabla E(\mathbf{w}_{t-1})]}{\|\nabla E(\mathbf{w}_{t-1})\|^2}$$ (Polak-Ribiere method)

$$\beta_t = \frac{\nabla E(\mathbf{w})^T [\nabla E(\mathbf{w}) - \nabla E(\mathbf{w}_{t-1})]}{\delta \mathbf{w}_{t-1}{}^T [\nabla E(\mathbf{w}) - \nabla E(\mathbf{w}_{t-1})]}$$ (Hestenes-Stiefel method)

2nd order algorithms: The Levenberg-Marquardt algorithm (1)

$$E = \frac{1}{2}\sum_{\mu}\left(e_{k\mu}\right)^2 = \frac{1}{2}\left\|\mathbf{e}\right\|^2, \quad e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}$$

$$\nabla E = \sum_{k\mu} e_{k\mu}\frac{\partial e_{k\mu}}{\partial \mathbf{w}} = \mathbf{e}^T\nabla\mathbf{e}$$

$\mathbf{J} = \nabla\mathbf{e}$: Jacobian

$$\nabla^2 E = \nabla(\mathbf{e}^T\nabla\mathbf{e}) = \nabla\mathbf{e}^T\nabla\mathbf{e} + \mathbf{e}^T\nabla^2\mathbf{e} = \mathbf{J}^T\mathbf{J} + \mathbf{e}^T\nabla^2\mathbf{e}$$

$$\nabla^2 E \simeq \mathbf{J}^T\mathbf{J}$$

When the norm of the error vector **e** is already small (close to the local minimum, where a quadratic approximation for the cost function makes sense).

2nd order algorithms: The Levenberg-Marquardt algorithm (2)

$$\delta \mathbf{w} = -(\mathbf{H} + \mu \mathbf{I})^{-1} \nabla E \longrightarrow$$

Levenberg:

- We keep μ small close to minima and large far away from minima
- Close to a minimum the weight update follows a Newton step
- Away from a minimum, if μ is large enough, emphasis is placed on gradient descent

$$\delta \mathbf{w} = -(\mathbf{J}^T \mathbf{J} + \mu \mathbf{I})^{-1} \nabla E \longrightarrow$$

A valid approximation, since the Hessian term is dominant close to the minimum, where the quadratic approximation for the cost function is valid

$$\delta \mathbf{w} = -[\mathbf{J}^T \mathbf{J} + \mu \, \mathrm{diag}(\mathbf{H})]^{-1} \nabla E \rightarrow$$

Variation (Marquardt): Weighted gradient components according to curvature, so that zig-zagging is avoided even more

2nd order algorithms: The Levenberg-Marquardt algorithm (3)

$$\delta \mathbf{w} = -(\mathbf{J}^T \mathbf{J} + \mu \mathbf{I})^{-1} \nabla E$$

$$\delta \mathbf{w} = -[\mathbf{J}^T \mathbf{J} + \mu \, \mathrm{diag}(\mathbf{H})]^{-1} \nabla E$$

- Criterion for updating μ: increase or decrease of the cost function in the previous step.
- If the cost has increased, our quadratic approximation is not reasonable, so we boost μ to place emphasis on the gradient descent term.
- If the cost has decreased, the quadratic approximation is valid, so we place emphasis on the Hessian term by decreasing μ.

2nd order algorithms: The Levenberg-Marquardt algorithm (4)

1. Initialization with small weights and small μ
2. Weight adaptation according to the rule

$$\delta \mathbf{w} = -(\mathbf{J}^T \mathbf{J} + \mu \mathbf{I})^{-1} \nabla E \qquad \text{(Levenberg)} \quad \text{or}$$

$$\delta \mathbf{w} = -[\mathbf{J}^T \mathbf{J} + \mu \, \text{diag}(\mathbf{H})]^{-1} \nabla E \qquad \text{(Marquardt)}$$

3. If the error grew:
   - Ignore the update (we return to the previous weight vector)
   - Increase μ: $\mu = \rho * \mu$ (ρ of the order of 10)
   - Return to step 2
4. If the error dropped:
   - Keep the new weights
   - Decrease μ: $\mu = \mu / \rho$
   - Return to step 2