Neural Networks & Deep Learning

Lecture 2:

Neural Architectures and Learning Algorithms-1

Prof George Magoulas
Email: g.magoulas@bbk.ac.uk

Outline

- Last week: neural computing fundamentals
- From the perceptron to multilayer neural networks
- The backpropagation algorithm
- Learning as error minimisation and the steepest descent method
- Gradient-based algorithms for supervised learning

Last week

In Machine Learning, model building is considered as approximation of some kind of "true" underlying function or distribution:

- Formulate a space of functions (implemented by a particular type of machine learning algorithm) in which you can search for a good function approximation easily; e.g. linear classifiers (a linear combination of some basis functions), or the space of functions induced by a neural network.
- Formulate a loss function which indicates how close are the predictions of the current function approximation. For neural networks, this is mean-squared or sum-squared error. For deep learning, it is often cross-entropy.
- Formulate a regulariser which allows you to vary the complexity of the function approximation so that overfitting/underfitting is avoided.
- Perform optimisation to minimize the sum of the loss and regularisation.

Neural computing assumptions about computation in the brain

1. **Neurons integrate information**. The neuron receives signals, either excitatory or inhibitory, from other neurons via synaptic connections onto the dendrites. If the sum of these signals exceeds a threshold, the neuron fires. This is communicated to other neurons by a signal passing down its axon. This signal acts as part of the input to the dendrites of the other neurons.

2. **Neurons pass information about the level of their input**. Each unit has an activity level, which is related to the input level- the higher the input, the higher the activity. The activity level is transmitted as a single value to all the units to which it is connected.

3. **Brain structure is layered**. Information is processed in the brain by a flow of activity passing through a sequence of physically independent structures. Example: the visual processing system.

4. **The influence of one neuron on another depends on the strength of the connection between them**. The effect of one neuron on another (whether it makes it much more or less likely to fire or whether it only slightly changes the probability) is determined by the strength of the synaptic connection between them, which is called the weight of the connection.

5. **Learning is achieved by changing the strengths of connections between neurons**. Experience can change the behaviour of an organism in response to a particular stimulus. Learning is implemented by rules, which determine how the weights of the connections between units are changed.

Applied Machine Learning module- how things work when using a package

Start with some labelled training data

1. Choose a differentiable loss function to minimise *(this is done in Keras choosing the Optimiser)*
2. Choose a network structure. Specifically determine how many layers and how many nodes in each layer.
3. Initialise the network's weights randomly *(this is done automatically in keras)*
4. Run the training data through the network to generate a prediction for each sample. Measure the overall error according to the loss function (the so-called forward propagation/pass)
5. Determine how much the current loss will change with respect to a small change in each of the weights. In other words, calculate the gradient of the loss function with respect to every weight in the network.
(the so-called backward propagation) *(binary cross entropy for binary output / categorical cross entropy for multiclass + softmax)*
6. Take a small "step" in the direction of the negative gradient *(depends on learning rate)*.
7. Repeat this process (from step 4) a fixed number of times or until the loss converges

Outline

- Last week: neural computing fundamentals
- **From the perceptron to multilayer neural networks**
- The backpropagation algorithm
- Learning as error minimisation and the steepest descent method
- Gradient-based algorithms for supervised learning

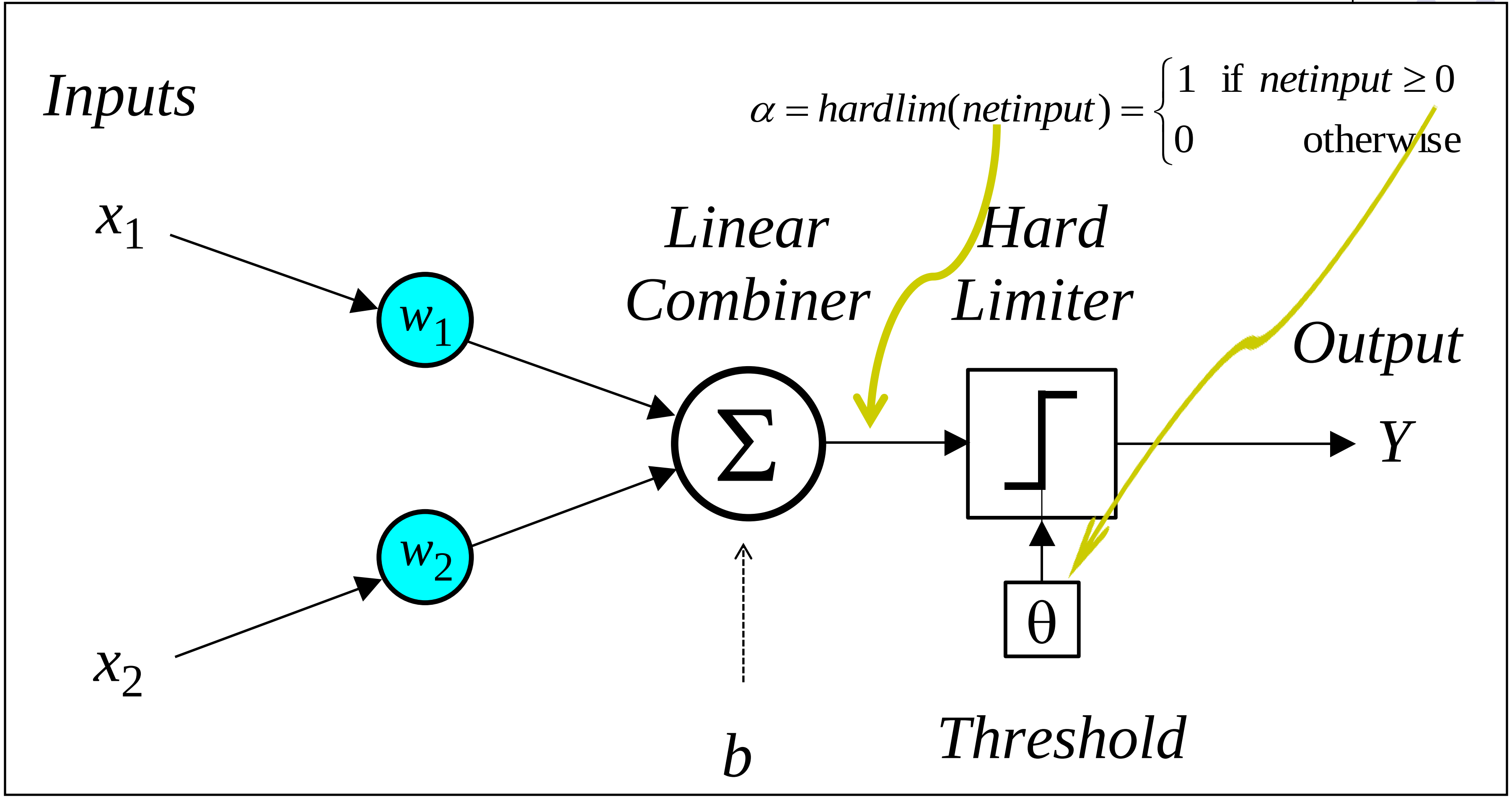
From perceptrons to Multilayer Networks

The perceptron and the perceptron rule: can a single neuron learn a task?

- In 1958, **Frank Rosenblatt** introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron**.
- The perceptron is the simplest form of a neural network. It consists of a single neuron with *adjustable* synaptic weights and a *hard limiter*.

Single-layer two-input perceptron

this node is still the main building block of modern networks (with some modifications/enhancements)



$$\alpha = hardlim(netinput) = \begin{cases} 1 & \text{if } netinput \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Let's have a look at a geometric interpretation of the node's training and operation…..→

The node divides the input space into two regions because it can only be in one of two states (i.e. 1 or 0- see threshold)

(1) Assume a node with only two inputs : $w_{11}x_1 + w_{12}x_2 = 0$
(2) Assume the node has a bias term *b:* $w_{11}x_1 + w_{12}x_2 + b = 0$
(3) Assume w_{11}=1; w_{12}=1; b=-1

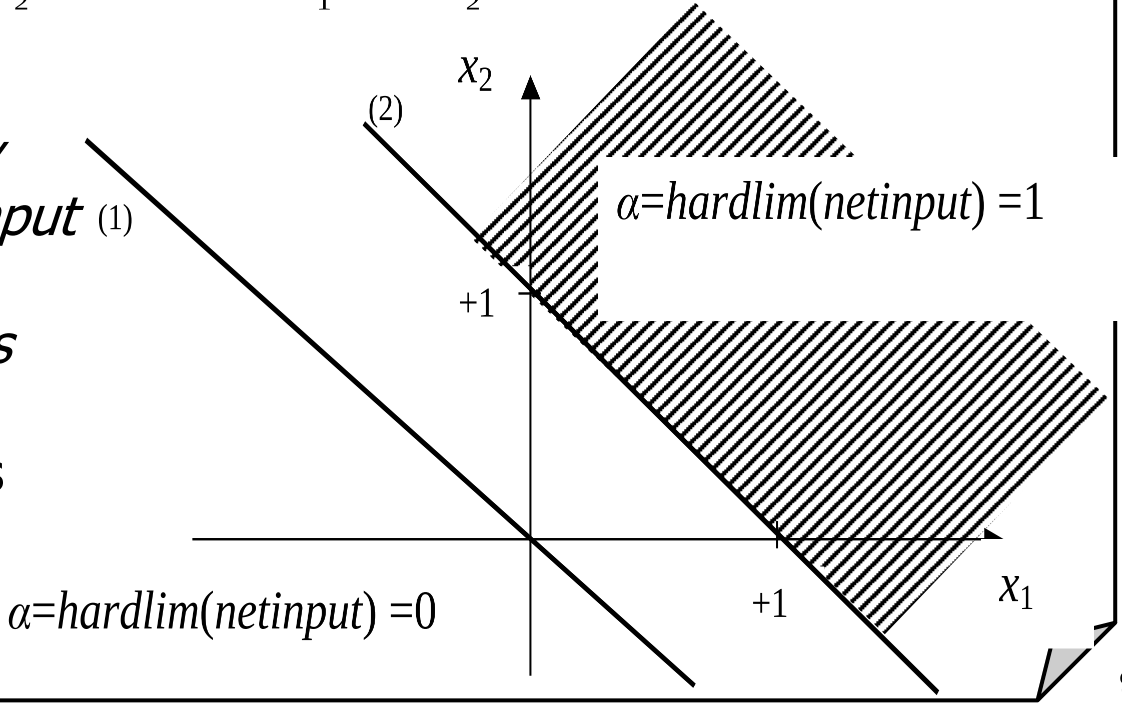
Then:

These are line equations→

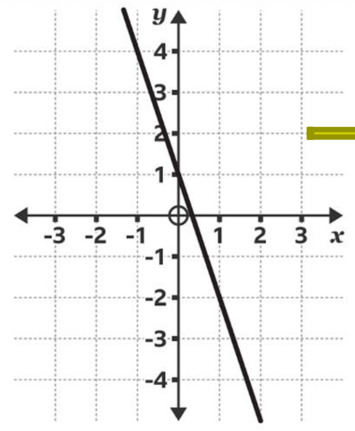
$x_1 + x_2 = 0 \Rightarrow x_1 = -x_2$ (1) No bias

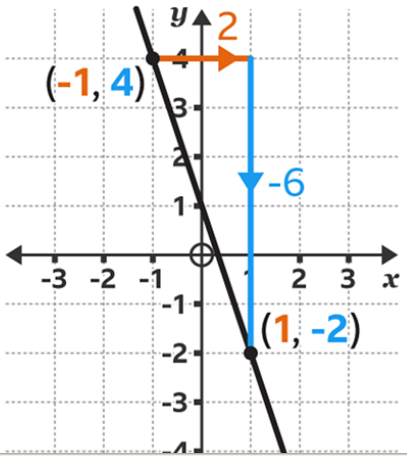
$x_1 + x_2 - 1 = 0 \Rightarrow x_1 = -x_2 + 1$ (2) With a bias

The line defines the boundary between regions where the input pattern produces a positive response (output) and regions where the response will be negative or zero. A line of this kind is also called *decision boundary*



For two coordinates on the line, (-1, 4) and (1, -2). Draw a triangle showing the horizontal movement to the right and the vertical movement down. Label the triangle with the change in the x-coordinate (from -1 to 1 is 2) and the change in the y-coordinate (from 4 to -2 is -6).

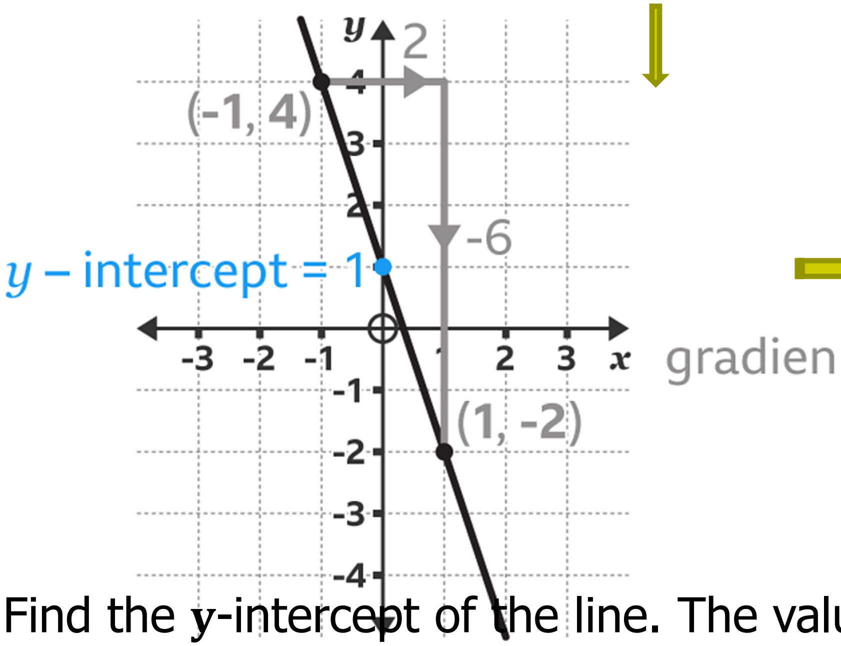


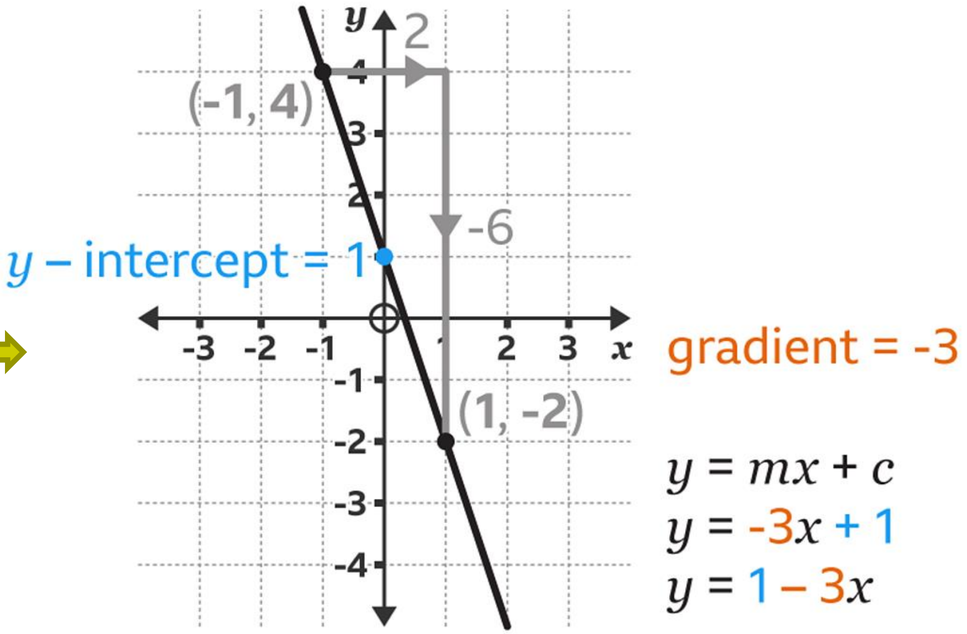


$-6 \div 2 = -3$

gradient = -3

As you move along a line from left to right, you might go up, you might go down or you might not change at all.





$y = mx + c$

$y = -3x + 1$

$y = 1 - 3x$

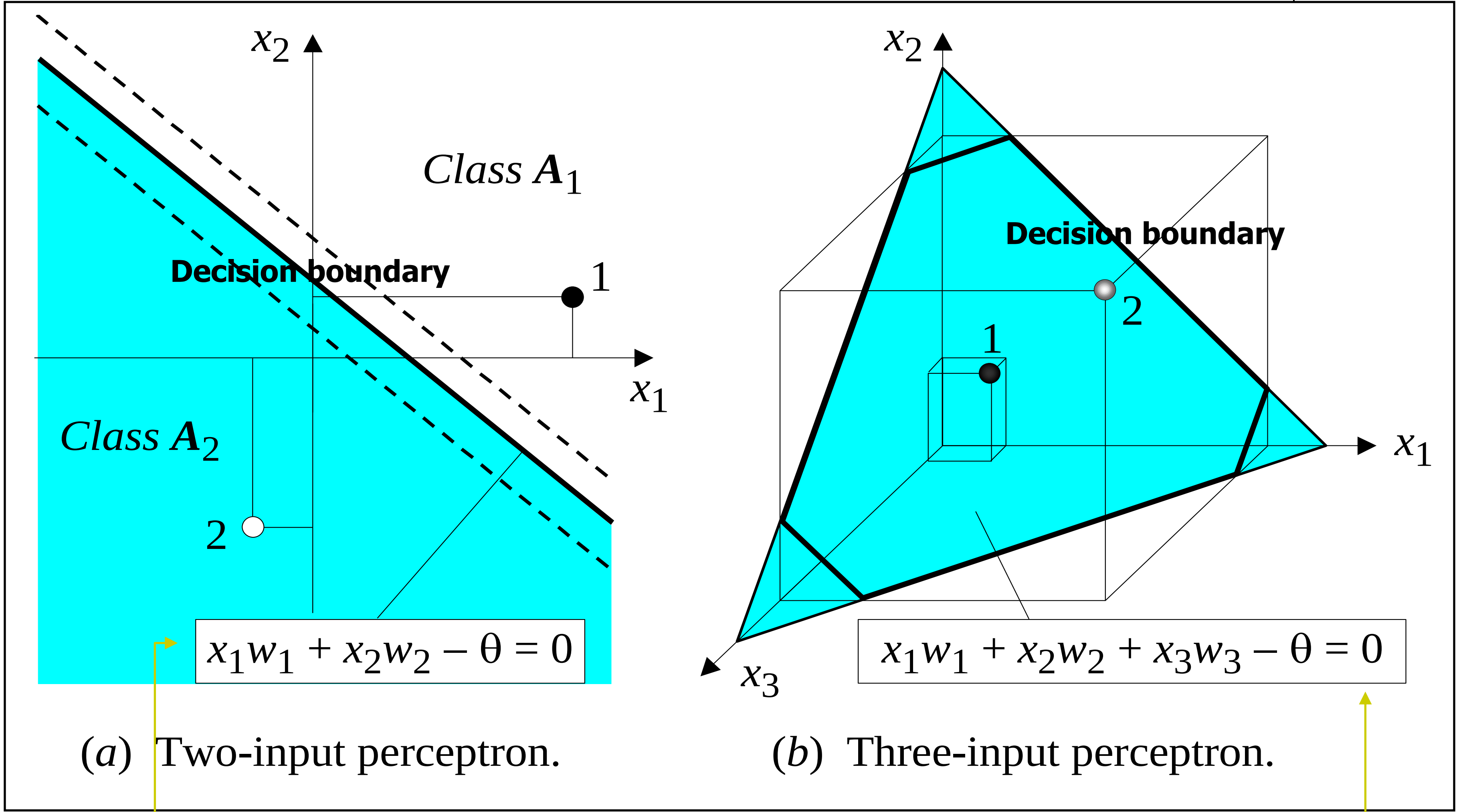
Find the y-intercept of the line. The value where the line crosses the y-axis is 1

https://www.bbc.co.uk/bitesize/subjects/z38pycw

- The aim of the perceptron is to classify inputs, $x_1, x_2, \ldots, x_n$, into one of two classes, say $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$.
- For an elementary perceptron, the n-dimensional space is divided by a *hyperplane* into two regions https://en.wikipedia.org/wiki/Hyperplane)
- The hyperplane is defined by the ***linearly separable*** **function**:

$$\boxed{\sum_{i=1}^{n} x_i w_i - \theta = 0}$$

Linear separability in the perceptron



(*a*) Two-input perceptron. (*b*) Three-input perceptron.

That's a line equation

That's a plane equation

https://en.wikipedia.org/wiki/Analytic_geometry

How does the perceptron learn its classification tasks?

This is done by making small adjustments in the weights* to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range [–0.5, 0.5], and then updated to obtain the output consistent with the training examples.

(*and other architectural hyperparameters, like biases, activations, thresholds etc in modern networks, if desirable)

- If at iteration *p*, the actual output is $Y(p)$ and the desired output is $Y_d(p)$, then the error is given by:

$$\boxed{e(p) = Y_d(p) - Y(p)}$$ where $p = 1, 2, 3, \ldots$

Iteration *p* here refers to the *p*th training example presented to the perceptron.

- If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

The perceptron learning rule

$$\boxed{w_i(p+1) = w_i(p) + \alpha \cdot x_i(p) \cdot e(p)}$$

where $p = 1, 2, 3, \ldots$

α is the **learning rate**, a positive constant less than unity.

The perceptron learning rule was first proposed by **Rosenblatt** in 1960. Using this rule we can derive the perceptron training algorithm for classification tasks.

Perceptron's training algorithm

<u>Step 1</u>: Initialisation

Set initial weights w_1, w_2,…, w_n and threshold θ to random numbers in the range $[-0.5, 0.5]$.

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$.

Perceptron's training algorithm (continued)

Step 2: Activation

Activate the perceptron by applying inputs $x_1(p)$, $x_2(p)$,…, $x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration $p = 1$

$$\boxed{Y(p) = step\left[\sum_{i=1}^{n} x_i(p)\, w_i(p) - \theta\right]}$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Perceptron's training algorithm (continued)

Step 3: Weight training

Update the weights of the perceptron

$$\boxed{w_i(p+1) = w_i(p) + \Delta w_i(p)}$$

where $\Delta w_i(p)$ is the weight correction at iteration *p*.

The weight correction is computed by the **delta rule**:

$$\boxed{\Delta w_i(p) = \alpha \cdot x_i(p) \cdot e(p)}$$

Step 4: Iteration

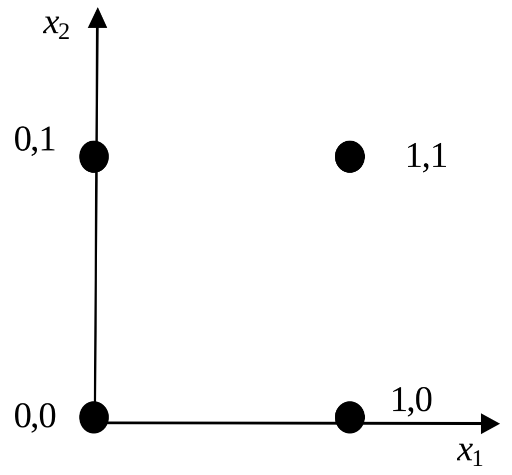
Increase iteration *p* by one, go back to *Step 2* and repeat the process until convergence.

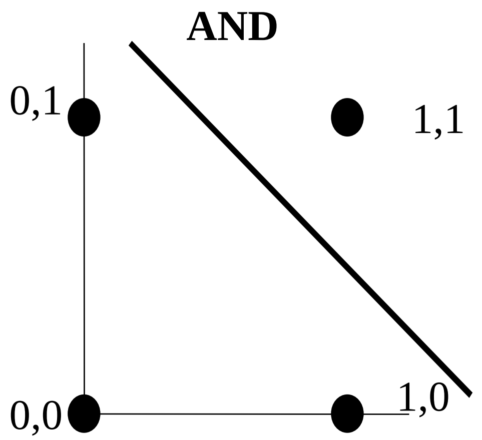
Perceptron and the perceptron rule (the limitations)

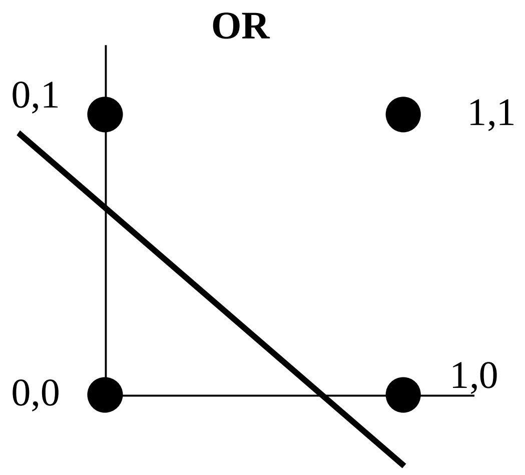
Linear separable problems:

Learning Boolean functions

Input		Output	
x_1	x_2	AND	OR
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

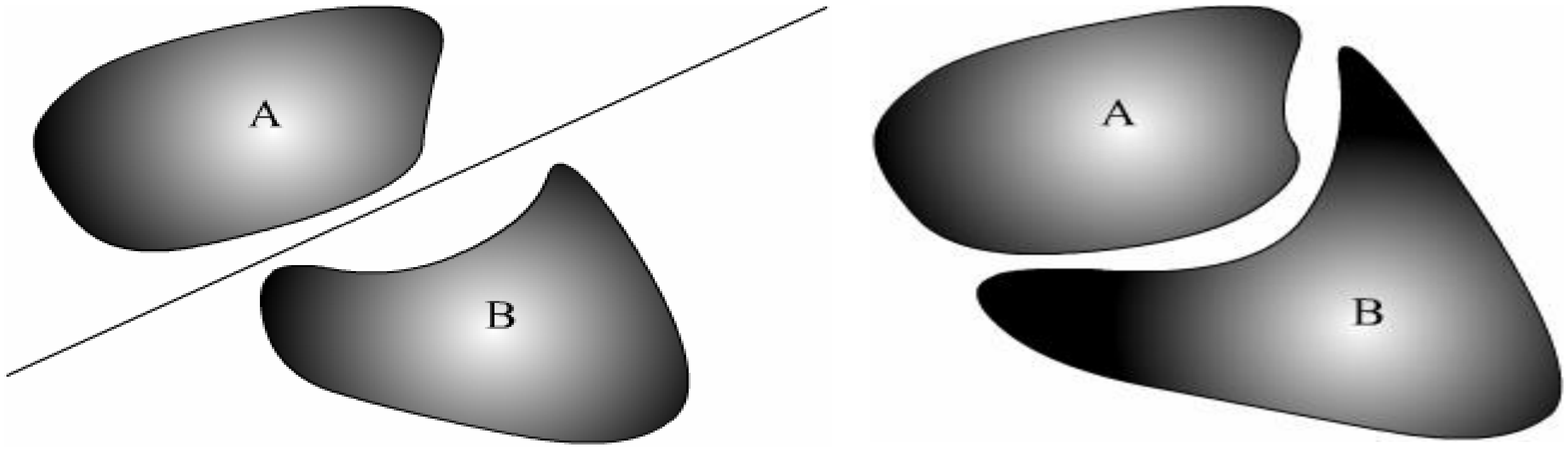






Multilayer perceptrons

Non-linear separable problems: Training patterns belonging to one output class cannot be separated from training patterns belonging to another class by a straight line, plane or hyperplane.

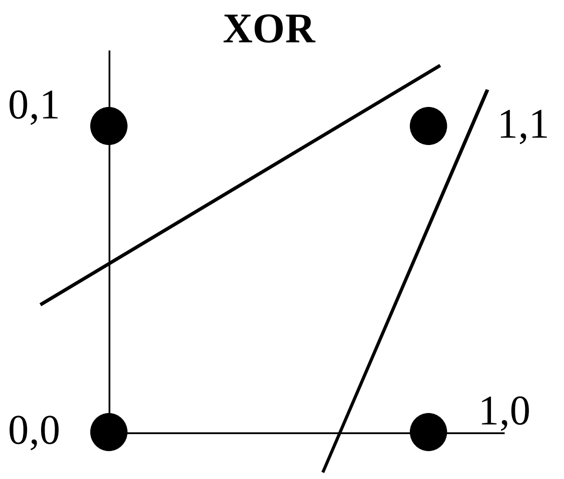


Linear separable **Nonlinear separable**

Multilayer perceptrons

Non-linear separable problems: Training patterns
be
tra
lin

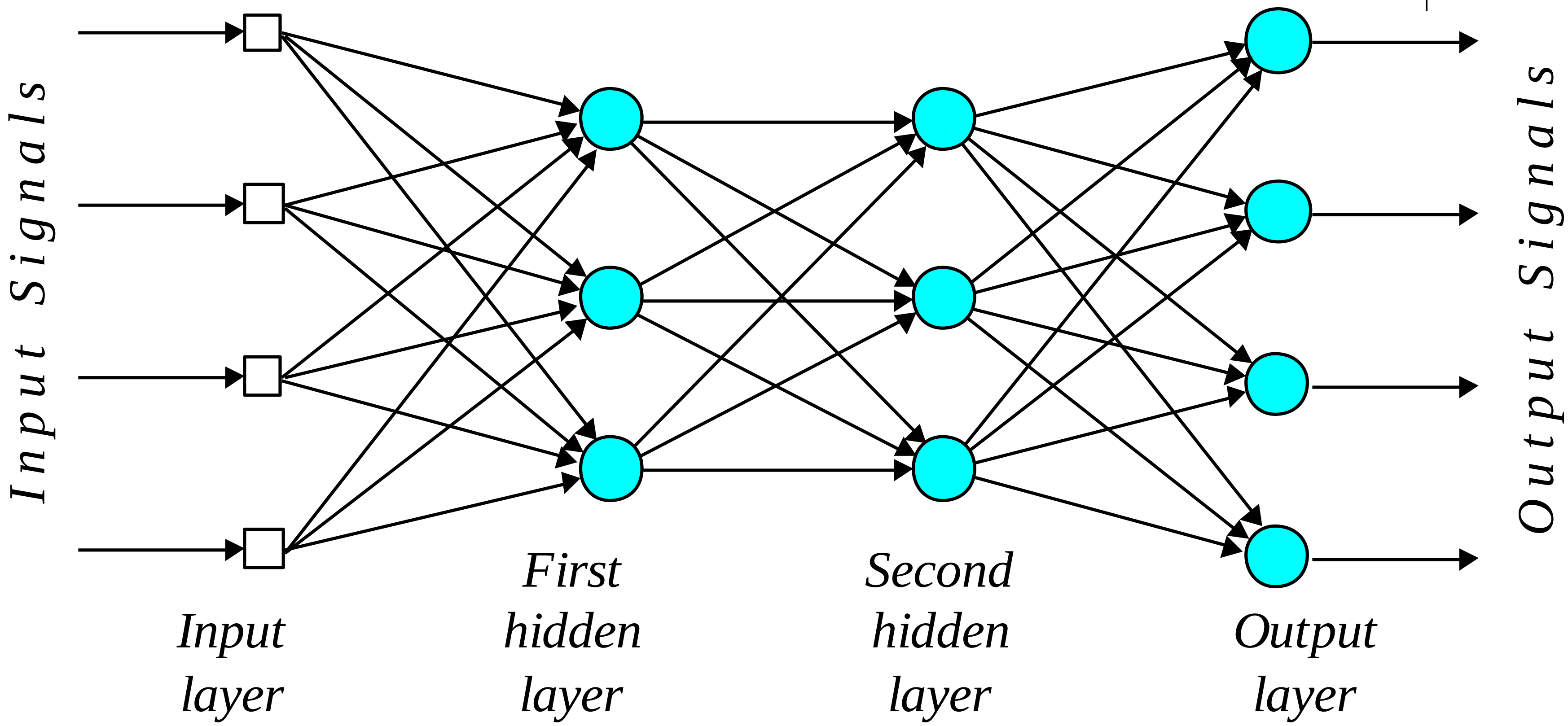
Input		Output
x_1	x_2	XOR
0	0	0
1	0	1
0	1	1
1	1	0



Multilayer neural networks

- A multilayer perceptron is a feedforward neural network with one or more hidden layers.
- The network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons.
- The input signals are propagated in a forward direction on a layer-by-layer basis.

Multilayer perceptron with two (or more) hidden layers



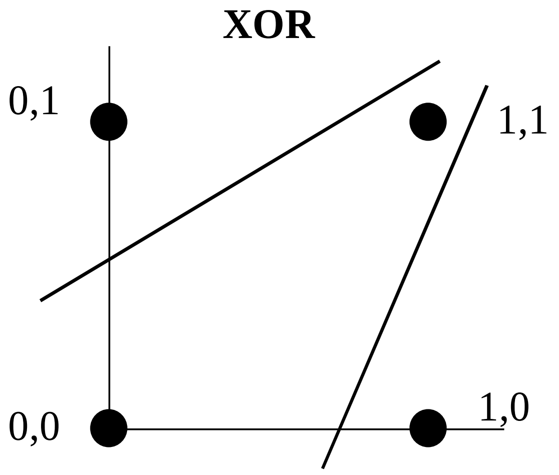
What does the middle layers hide?

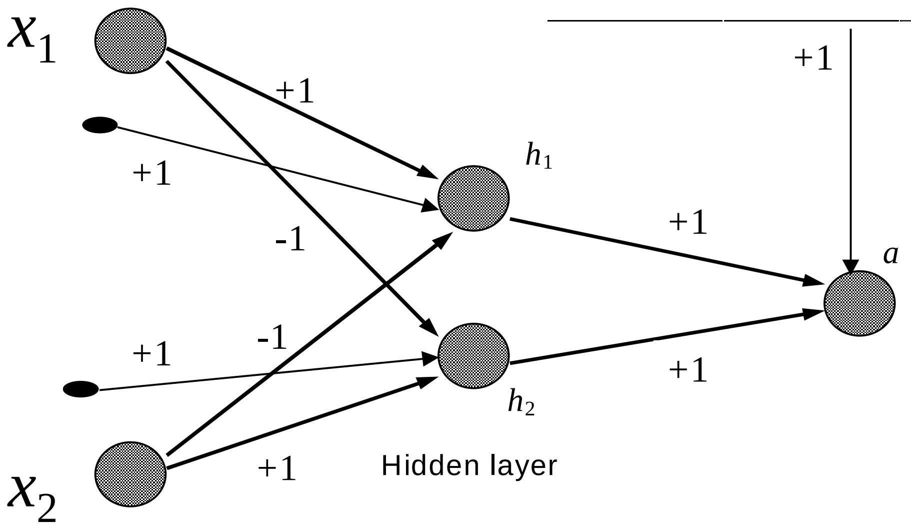
- A hidden layer "hides" its desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network. There is no obvious way to know what the desired output of the hidden layer should be.
- Shallow ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Deep ANNs may have five or more layers and utilise millions of neurons.

Multilayer perceptron

Example of nonlinear separable problem

Input		Output
x_1	x_2	XOR
0	0	0
1	0	1
0	1	1
1	1	0





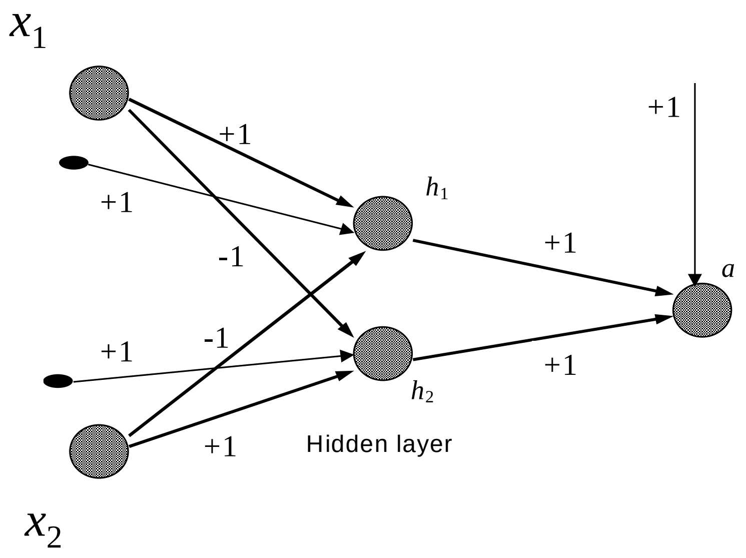
Each node will have its own decision boundary

A single node can classify input vectors into two categories.

$$\text{step}(netinput) = \begin{cases} 1 & \text{if } netinput > 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{step}(netinput) = \begin{cases} 1 & \text{if } netinput > 1 \\ 0 & \text{otherwise} \end{cases}$$

Hidden nodes' activities for the XOR problem



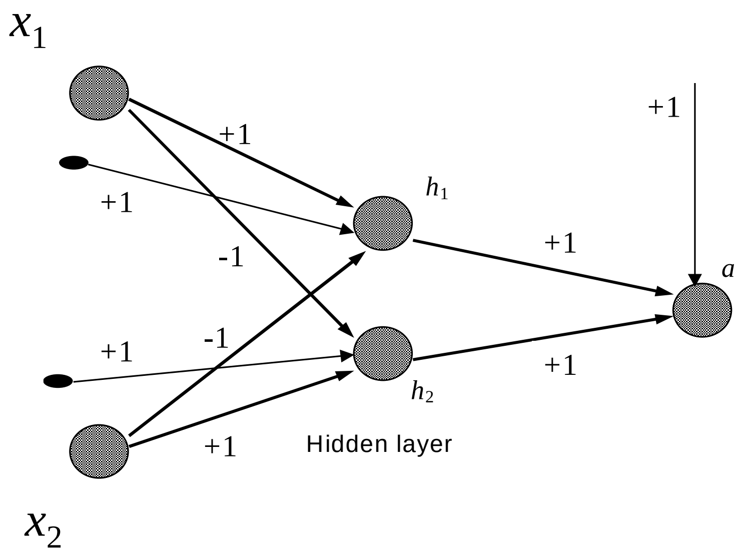
Input		Hidden		Targets
x_1	x_2	h_1	h_2	
0	0	0	0	0
1	0	1	0	1
0	1	0	1	1
1	1	0	0	0

$$h_1 = step[\,(+1) * x_1 + (-1) * x_2 + 1]$$

$$h_2 = step[\,\underbrace{(-1) * x_1 + (+1) * x_2 + 1}_{\text{That's a line equation: } -x_1 + x_2 + 1 = 0}]$$

$$\text{step}(netinput) = \begin{cases} 1 & \text{if } netinput > 1 \\ 0 & \text{otherwise} \end{cases}$$

Hidden nodes' activities for the XOR problem



Input		Hidden		Targets
x_1	x_2	h_1	h_2	
0	0	0	0	0
1	0	1	0	1
0	1	0	1	1
1	1	0	0	0

$- 0^* x_1 + 1^* x_2 + 1 = 2 > 1 \Rightarrow h_2 = 1$

$$h_1 = step[\,(+1) * x_1 + (-1) * x_2 + 1]$$

$$h_2 = \underbrace{step[\,(-1) * x_1 + (+1) * x_2 + 1]}$$

That's a line equation: $- x_1 + x_2 + 1 = 0$

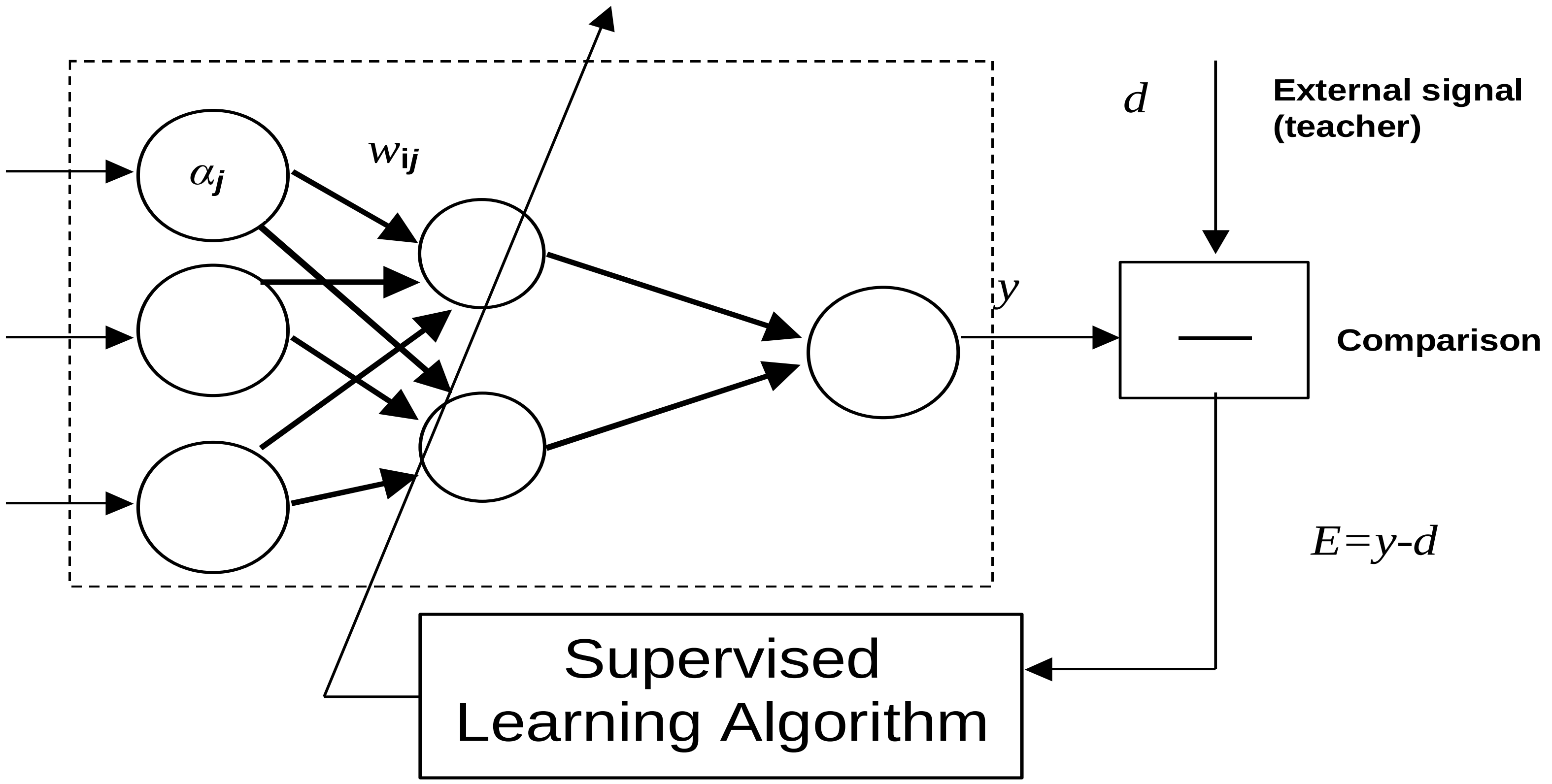
Outline

- Last week: neural computing fundamentals
- From the perceptron to multilayer neural networks
- **The backpropagation algorithm**
- Learning as error minimisation and the steepest descent method
- Gradient-based algorithms for supervised learning

The back-propagation algorithm

- Learning in a multilayer network proceeds the same way as for a perceptron.
- A training set of input patterns is presented to the network.
- The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

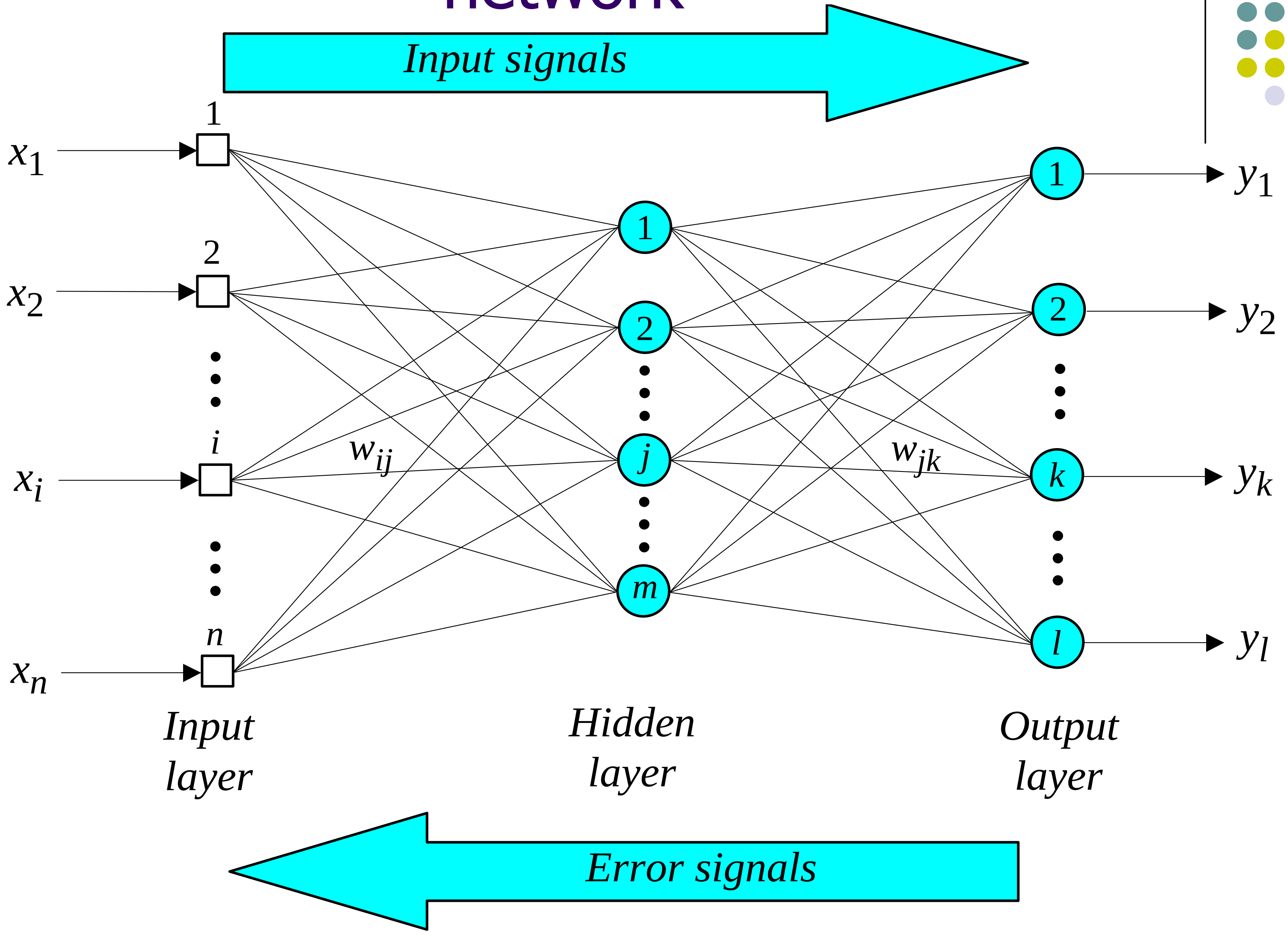
This form of training is called **Supervised learning:** The response that the backpropagation network is required to learn is presented to the network during training. The desired response of the network acts as an explicit teacher signal.



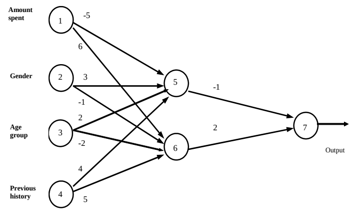
N. Abramson and D. Braverman, Learning to recognize patterns in a random environment, IRE Trans. Information Theory, vol. IT-8, pp. S58S63, September 1962.

- In a back-propagation neural network, the learning algorithm has two phases.
- First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

Three-layer back-propagation neural network

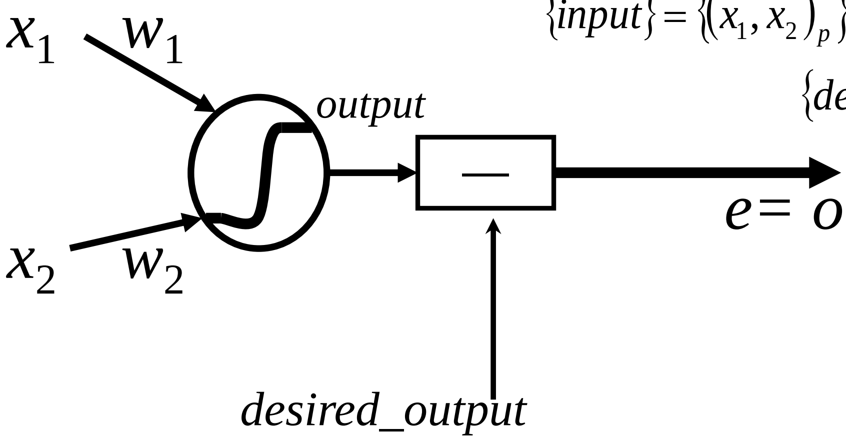


Nonlinear neuron



$$\{input\} = \{(x_1, x_2)_p\}_{p=1}^{8} = \{(-6,1),(-6.1,1),(-4.1,1),(-4,1),(4,1),(4.1,1),(6,1),(6.1,1)\}$$

$$\{desired\_output\} = \{0,0,0.97,0.99,0.01,0.03,1,1\}$$



$$e = output - desired\_output$$

$$output = \frac{1}{1 + e^{(w_1 x_1 + w_2 x_2)}}$$

$$E = \sum_{i=1}^{p} e_i^2$$

Sum squared error function

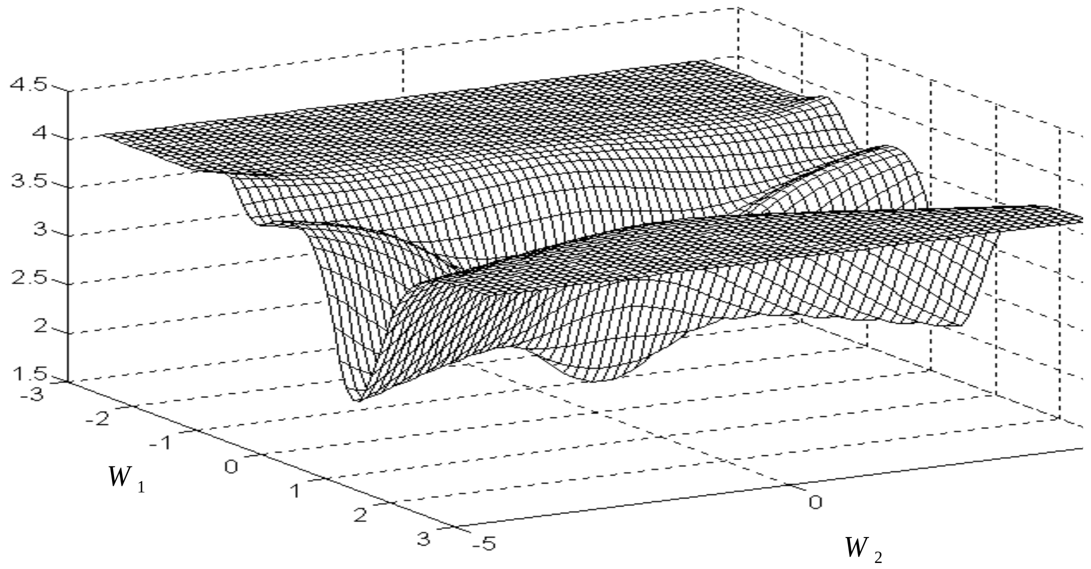


Figure from Effective backpropagation training with variable step size, Neural Networks, 1997

Outline

- Last week: neural computing fundamentals
- From the perceptron to multilayer neural networks
- The backpropagation algorithm
- **Learning as error minimisation and the steepest descent method**
- Gradient-based algorithms for supervised learning

Learning as minimisation of the error and the steepest descent method

$$\min_{\mathbf{w}} E(\mathbf{w}).$$

General weight update rule:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{d}_k,$$

$\mathbf{d}_k$: search direction

α_k : stepsize

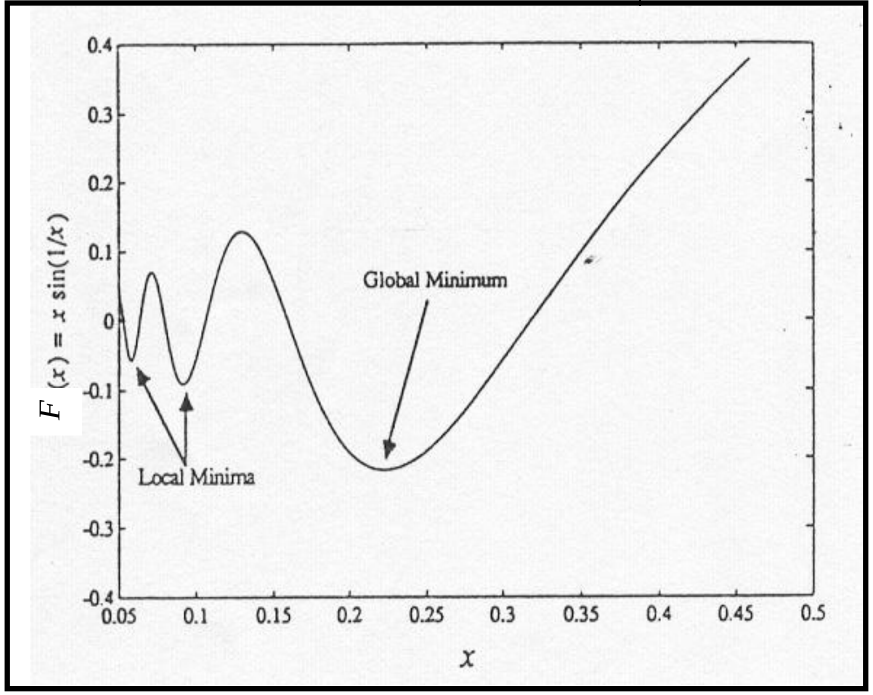
A. Cauchy. Methode generale pour la resolution des systemes d’equations simultanees. C. R. Acad. Sci. Paris, 25:536–538, 1847.

Formulation as a minimisation of the error

<u>Gradient vector</u>

<u>of function $F(\mathbf{x})$:</u>

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \dots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$



First derivative of $F(\mathbf{x})$ with respect

to x_i (ith element of gradient vector):

$$\partial F(\mathbf{x}) / \partial x_i$$

<u>Optimality Condition</u>

$$\nabla F(\mathbf{x}) \Big|_{\mathbf{x} = \mathbf{x}^*} = \mathbf{0}$$

Steepest Descent

Gradient descent

Choose the next $\mathbf{x}$ so that: $F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$

Maximise the decrease by choosing: $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$

where $\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x} = \mathbf{x}_k}$

Curry, Haskell B. (1944). The Method of Steepest Descent for Non-linear Minimization Problems. Quart. Appl. Math. 2 (3): 258–261. doi:10.1090/qam/10667

Steepest Descent

$$F(\mathbf{x}) = x_1^2 + 2x_1x_2 + 2x_2^2 + x_1$$

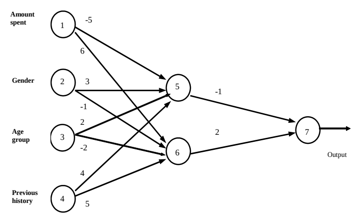
$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \qquad \alpha = 0.1$$

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + 1 \\ 2x_1 + 4x_2 \end{bmatrix} \qquad \mathbf{g}_0 = \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \mathbf{g}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} - 0.1 \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$

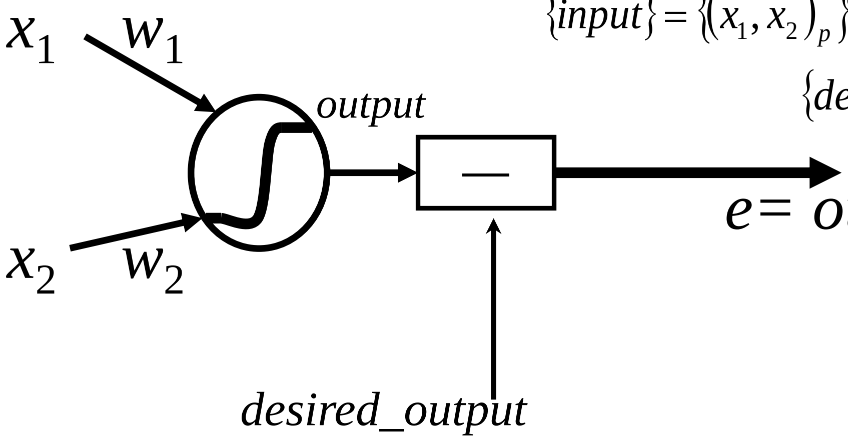
$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha \mathbf{g}_1 = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix} - 0.1 \begin{bmatrix} 1.8 \\ 1.2 \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.08 \end{bmatrix}$$

Nonlinear neuron



$$\{input\} = \{(x_1, x_2)_p\}_{p=1}^{8} = \{(-6,1),(-6.1,1),(-4.1,1),(-4,1),(4,1),(4.1,1),(6,1),(6.1,1)\}$$

$$\{desired\_output\} = \{0,0,0.97,0.99,0.01,0.03,1,1\}$$



$$e = output - desired\_output$$

$$output = \frac{1}{1+e^{(w_1x_1+w_2x_2)}}$$

$$E = \sum_{i=1}^{p} e_i^2$$

Sum squared error function

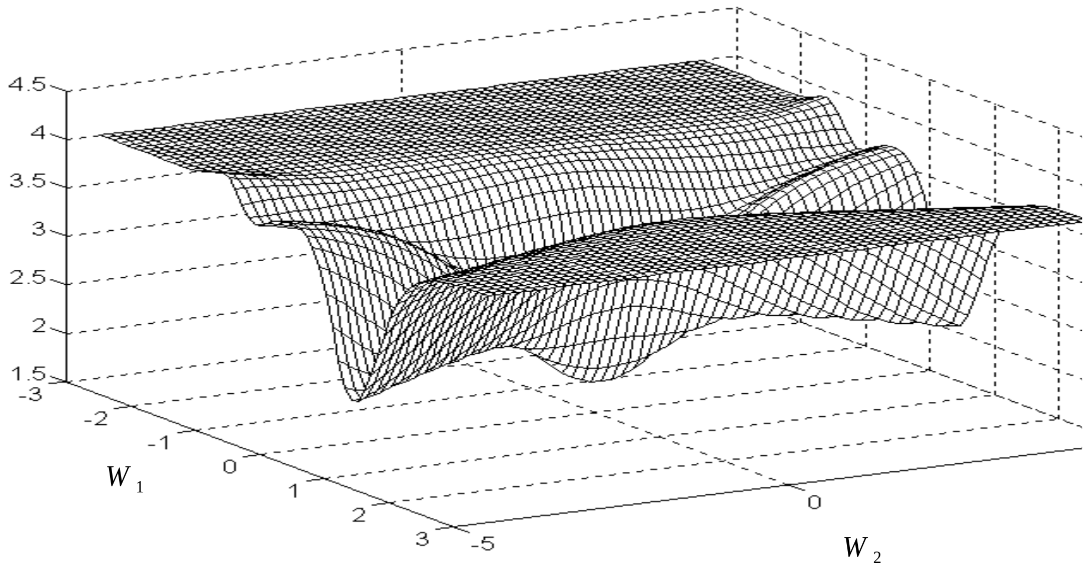
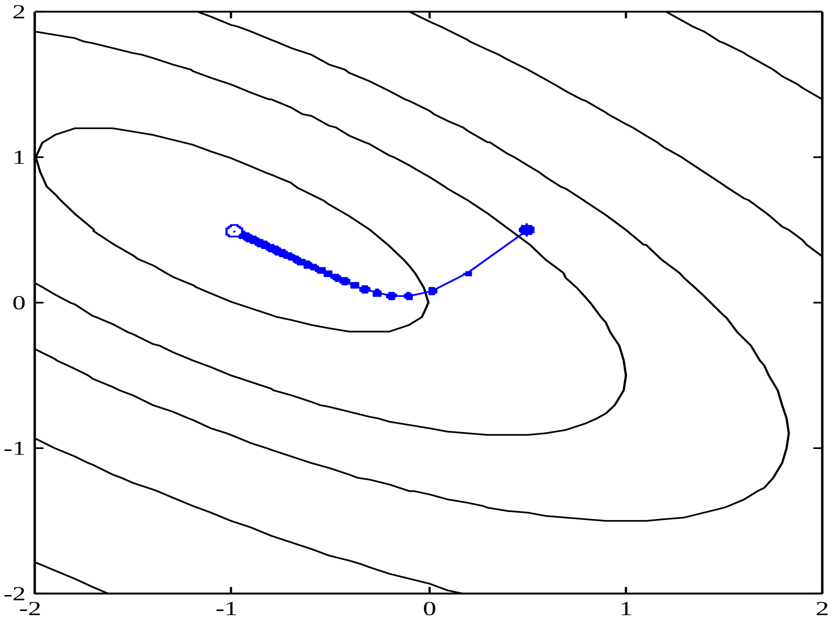
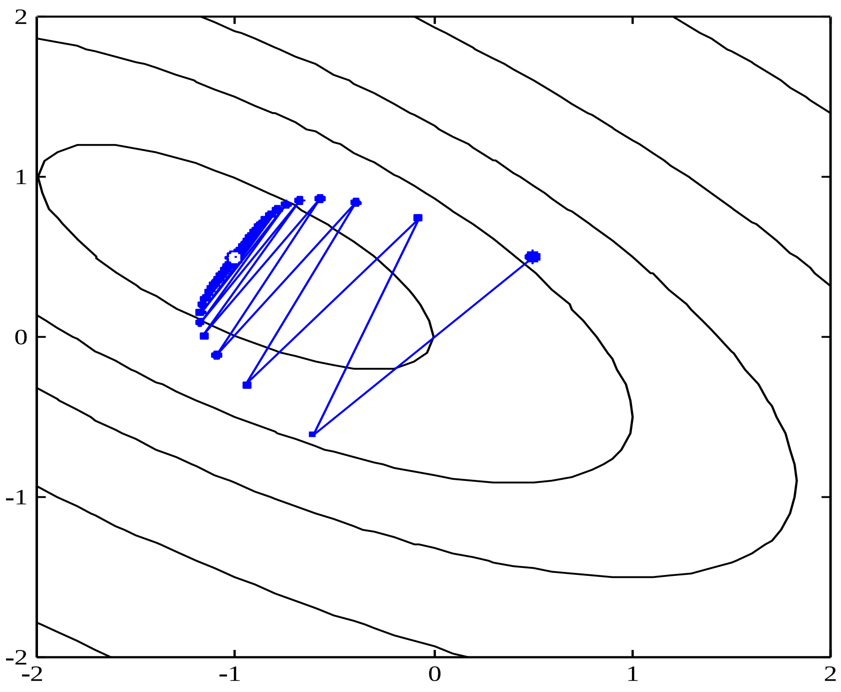


Figure from Effective backpropagation training with variable step size, Neural Networks, 1997

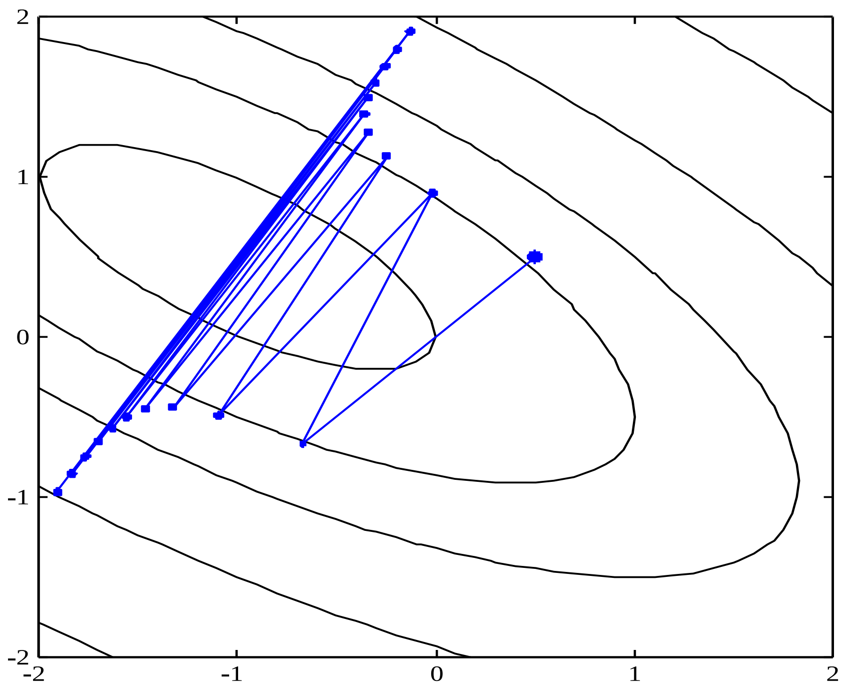


α = learning rate

$\alpha = 0.37$

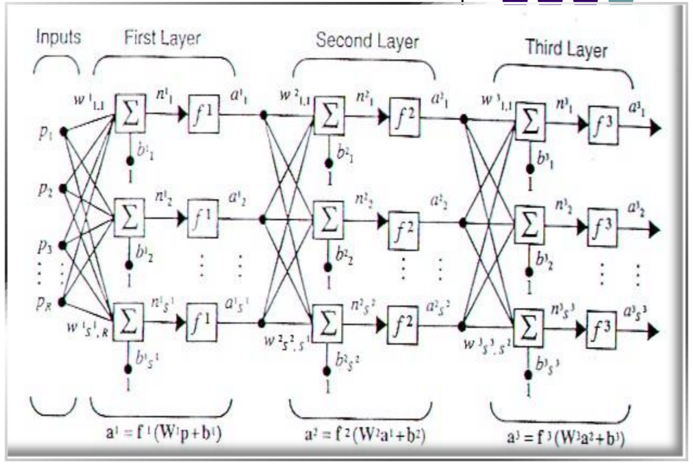


$\alpha = 0.39$



The Chain Rule

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}$$



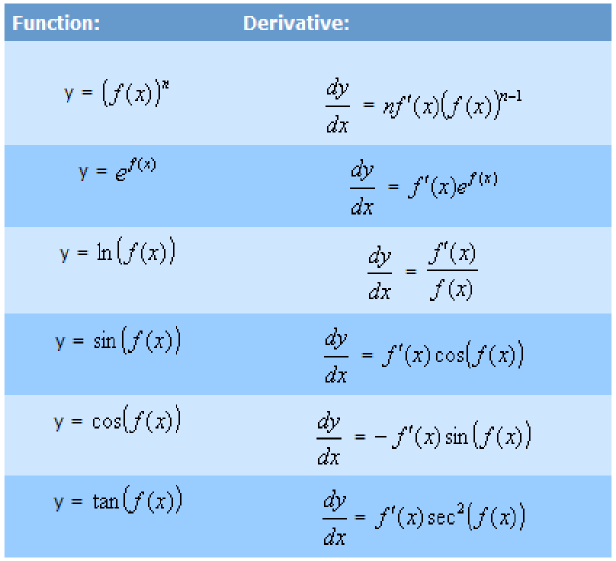
Example **composite function**

$$f(n) = \cos(n) \qquad n = e^{2w} \qquad f(n(w)) = \cos(e^{2w})$$

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (-\sin(n))(2e^{2w}) = (-\sin(e^{2w}))(2e^{2w})$$

Application to Gradient Calculation

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \qquad \frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$



Function:	Derivative:
$y = (f(x))^n$	$\frac{dy}{dx} = nf'(x)(f(x))^{n-1}$
$y = e^{f(x)}$	$\frac{dy}{dx} = f'(x)e^{f(x)}$
$y = \ln(f(x))$	$\frac{dy}{dx} = \frac{f'(x)}{f(x)}$
$y = \sin(f(x))$	$\frac{dy}{dx} = f'(x)\cos(f(x))$
$y = \cos(f(x))$	$\frac{dy}{dx} = -f'(x)\sin(f(x))$
$y = \tan(f(x))$	$\frac{dy}{dx} = f'(x)\sec^2(f(x))$

Outline

- Last week: neural computing fundamentals
- From the perceptron to multilayer neural networks
- The backpropagation algorithm
- Learning as error minimisation and the steepest descent method
- **Gradient-based algorithms for supervised learning**

Gradient-based algorithms for supervised learning

Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986). Learning representations by back-propagating errors. Nature, 533–536.

Step 1: Initialisation

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range:

$$\boxed{\left(-\frac{2.4}{F_i},\ +\frac{2.4}{F_i}\right)}$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: Activation

Activate the back-propagation neural network by applying inputs $x_1(p)$, $x_2(p)$,…, $x_n(p)$ and desired outputs $y_{d,1}(p)$, $y_{d,2}(p)$,…, $y_{d,n}(p)$.

(a) Calculate the actual outputs of the neurons in the hidden layer:

$$\boxed{y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \cdot w_{ij}(p) - \theta_j\right]}$$

where n is the number of inputs of neuron j in the hidden layer, and *sigmoid* is the *sigmoid* activation function.

Step 2: Activation (continued)

(*b*) Calculate the actual outputs of the neurons in the output layer:

$$\boxed{y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \cdot w_{jk}(p) - \theta_k\right]}$$

where *m* is the number of inputs of neuron *k* in the output layer.

Step 3: Weight training

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(*a*) Calculate the error gradient for the neurons in the output layer:

$$\boxed{\delta_k(p) = y_k(p) \cdot [1 - y_k(p)] \cdot e_k(p)}$$

where $\boxed{e_k(p) = y_{d,k}(p) - y_k(p)}$

Calculate the weight corrections:

$$\boxed{\Delta w_{jk}(p) = \alpha \cdot y_j(p) \cdot \delta_k(p)}$$

Update the weights at the output neurons:

$$\boxed{w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)}$$

Step 3: Weight training (continued)

(*b*) Calculate the error gradient for the neurons in the hidden layer:

$$\boxed{\delta_j(p) = y_j(p)\cdot[1-y_j(p)]\cdot\sum_{k=1}^{l}\delta_k(p)\, w_{jk}(p)}$$

Calculate the weight corrections:

$$\boxed{\Delta w_{ij}(p) = \alpha\cdot x_i(p)\cdot\delta_j(p)}$$

Update the weights at the hidden neurons:

$$\boxed{w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)}$$

Step 4: Iteration

Increase iteration *p* by one, go back to *Step 2* and repeat the process until the selected error criterion is satisfied.

If you want to learn more about how this has been implemented in modern packages (Pytorch, Keras, Tensoflow): Automatic Differentiation in Machine Learning: a Survey

Reminder-How things work when using a package

Start with some labelled training data

1. Choose a differentiable loss function to minimise *(this is done in Keras choosing the Optimiser)*
2. Choose a network structure. Specifically determine how many layers and how many nodes in each layer.
3. Initialise the network's weights randomly *(this is done automatically in keras)*
4. Run the training data through the network to generate a prediction for each sample. Measure the overall error according to the loss function (the so-called forward propagation/pass)
5. Determine how much the current loss will change with respect to a small change in each of the weights. In other words, calculate the gradient of the loss function with respect to every weight in the network.
(the so-called backward propagation) *(binary cross entropy for binary output / categorical cross entropy for multiclass + softmax)*
6. Take a small "step" in the direction of the negative gradient *(depends on learning rate)*.
7. Repeat this process (from step 4) a fixed number of times or until the loss converges

Gradient-descent variants

-*Batch learning*: gradient of E over the full training set
-*Online learning*: gradient estimation E_p based on only one training point/pattern
-*Mini-batch learning*: better gradient estimation than E_p as based on subset of training data depending on size of the mini-batch.

Steepest descent rule: $w^{new} = w^{old} + \Delta w$

$$w^{k+1} = w^k - \eta \nabla E(w^k)$$

Notation:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial x_n} F(\mathbf{x}) \end{bmatrix}$$

Consists of the first derivatives of $F(\mathbf{x})$ with respect to x_i (*i*th element of gradient vector): $\partial F(\mathbf{x}) / \partial x_i$

In multilayer networks is:

$$\frac{\partial E(w)}{\partial w_i} = \frac{\partial \left(\sum_{p=1}^{P} e_p^2 \right)}{\partial w_i}$$

Steepest descent with momentum:

$$w^{k+1} = w^k - \eta \nabla E(w^k) + m \left(w^k - w^{k-1} \right)$$

Online learning

```
for several epochs of training do
    for each training example in the data do
        Calculate gradients of the loss
        Update the weights based on the calculated gradient
    end for
end for
```

Batch learning

```
for several epochs of training do
  for each training example in the data do
      Calculate and accumulate gradients of the loss
  end for
  Update the weights according to the accumulated gradient
end for
```

Mini-batch learning

```
for several epochs of training do
    for each training example in the mini-batch do
       Calculate and accumulate gradients of the loss
    end for
    Update the weights according to the gradient accumulated in this mini-batch
end for
```

Gradient-based algorithms for supervised learning

- **The Rprop method :** help to eliminate harmful influences of derivatives' magnitude on the weight updates.

  Basic Idea: the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update.

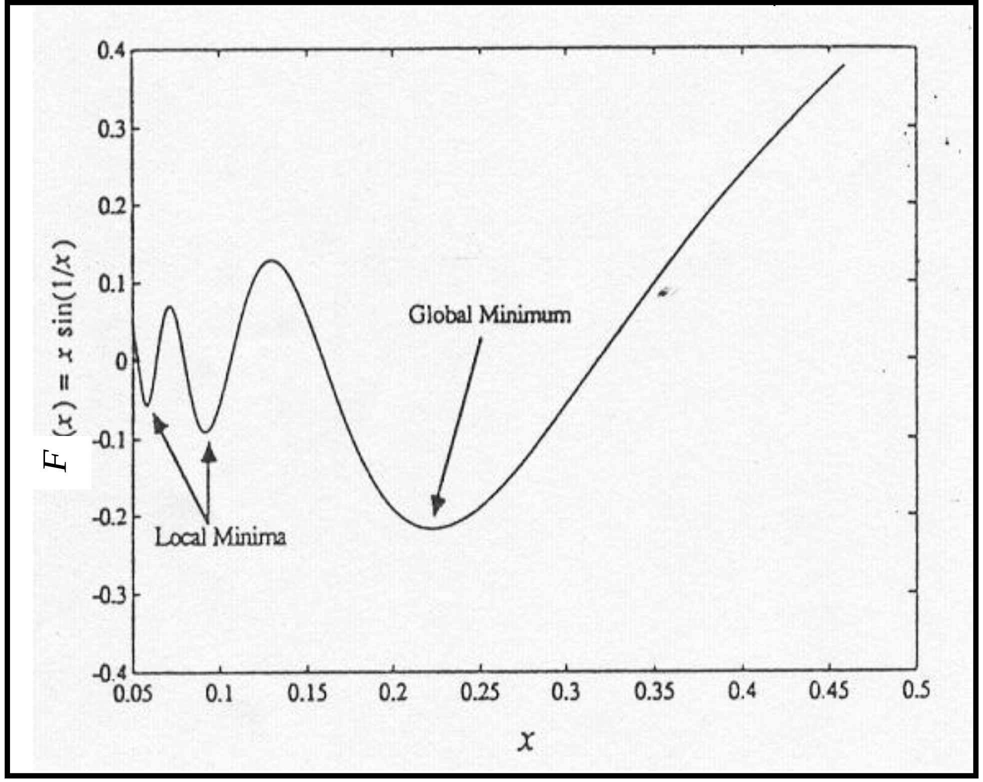
***The Resilient propagation update rule*:**

$$w^{k+1} = w^k - diag\{\eta_1^k, \cdots, \eta_i^k, \ldots \eta_n^k\} \cdot sign\left(g(w^k)\right)$$

$$\begin{aligned}
&if \quad \left(g_m(w^{k-1}) \cdot g_m(w^k) > 0\right) \quad then \quad \eta_m^k = \min\left(\eta_m^{k-1} \cdot \eta^+, \Delta_{\max}\right)\\
&if \quad \left(g_m(w^{k-1}) \cdot g_m(w^k) < 0\right) \quad then \quad \eta_m^k = \max\left(\eta_m^{k-1} \cdot \eta^-, \Delta_{\min}\right)\\
&if \quad \left(g_m(w^{k-1}) \cdot g_m(w^k) = 0\right) \quad then \quad \eta_m^k = \eta_m^{k-1}
\end{aligned}$$

Rprop

```
for each $w_k$ do{
    if $g_k^T * g_{k-1} > 0$ then{
        $\Delta_k = \min\left(\Delta_{k-1} \cdot \eta^+, \Delta_{\max}\right);$
        $\Delta w_k = -\mathrm{sign}\left(g_k\right) \cdot \Delta_k;$        }
    elseif $g_k^T * g_{k-1} < 0$ then{
        $\Delta_k = \max\left(\Delta_{k-1} \cdot \eta^-, \Delta_{\min}\right);$
        $\Delta w_k = -\Delta w_{k-1};$
        $g_k = 0;$                                   }
    elseif $g_k^T * g_{k-1} = 0$ then{.
        $\Delta_k = \Delta_{k-1};$
        $\Delta w_k = -\mathrm{sign}\left(g_k\right) \cdot \Delta_k;$        }
 $w_{k+1} = w_k + \Delta w_k;$
```



....& GRprop →

→ GRprop

Theorem 1. *Suppose that assumptions* (i) (iii) *are fulfilled, then for any* $w^0 \in \mathbb{R}^n$ *and any sequence* $\{w^k\}_{k=0}^{\infty}$ *generated by the Rprop scheme*

$$w^{k+1} = w^k - \tau^k \operatorname{diag}\{\eta_1^k, \dots, \eta_i^k, \dots, \eta_n^k\} \operatorname{sign}(g(w^k)), \quad k = 0, 1, \dots, \tag{5}$$

where $\operatorname{sign}(g(w^k))$ *denotes the column vector of the signs of the components of* $g(w^k) = (g_1(w^k), g_2(w^k), \dots, g_n(w^k))$, $\tau^k > 0$ *satisfying Wolfe's conditions,* $\eta_m^k (m = 1, 2, \dots, i-1, i+1, \dots, n)$ *are small positive real numbers generated by Rprop's learning rates schedule:*

$$\textit{if } (g_m(w^{k-1}) \cdot g_m(w^k) > 0) \quad \textit{then } \eta_m^k = \min(\eta_m^{k-1} \cdot \eta^+, \Delta_{\max}), \tag{6}$$

$$\textit{if } (g_m(w^{k-1}) \cdot g_m(w^k) < 0) \quad \textit{then } \eta_m^k = \max(\eta_m^{k-1} \cdot \eta^-, \Delta_{\min}), \tag{7}$$

$$\textit{if } (g_m(w^{k-1}) \cdot g_m(w^k) = 0) \quad \textit{then } \eta_m^k = \eta_m^{k-1}, \tag{8}$$

where $0 < \eta^- < 1 < \eta^+$, $\Delta_{\max}$ *is the learning rate upper bound,* $\Delta_{\min}$ *is the learning rate lower bound and*

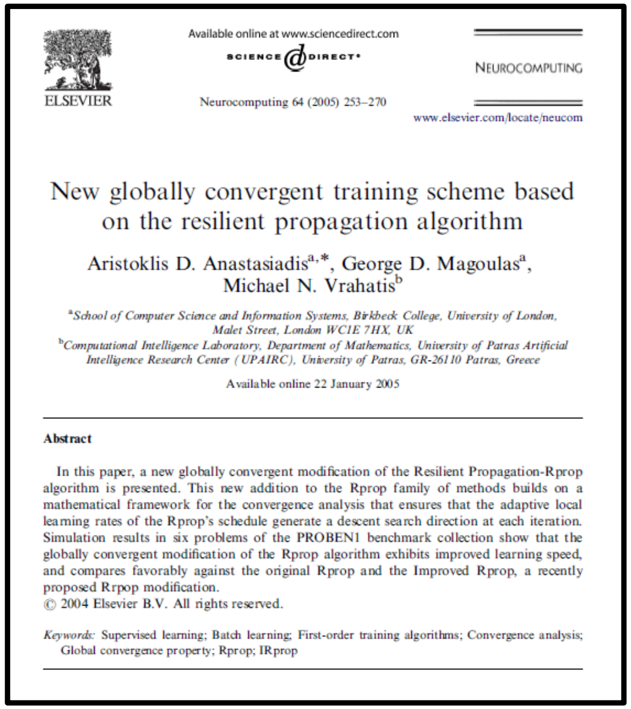
$$\eta_i^k = -\frac{\sum_{\substack{j=1 \\ j \neq i}}^{n} \eta_j^k g_j(w^k) + \delta}{g_i(w^k)}, \quad 0 < \delta \ll \infty, \quad g_i(w^k) \neq 0, \tag{9}$$

it holds that $\lim_{k \to \infty} g(w^k) = 0$.

$$f(x^k + \tau^k d^k) - f(x^k) \leqslant \sigma_1 \tau^k g(x^k)^\top d^k, \tag{3}$$

$$g(x^k + \tau^k d^k)^\top d^k \geqslant \sigma_2 g(x^k)^\top d^k, \tag{4}$$

Included in **R neuralnet**- the R neural networks package by Frauke Günther and Stefan Fritsch.



Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Neurocomputing 64 (2005) 253–270
NEUROCOMPUTING
www.elsevier.com/locate/neucom

New globally convergent training scheme based on the resilient propagation algorithm

Aristoklis D. Anastasiadis[a,*], George D. Magoulas[a], Michael N. Vrahatis[b]

[a]School of Computer Science and Information Systems, Birkbeck College, University of London, Malet Street, London WC1E 7HX, UK
[b]Computational Intelligence Laboratory, Department of Mathematics, University of Patras Artificial Intelligence Research Center (UPAIRC), University of Patras, GR-26110 Patras, Greece

Available online 22 January 2005

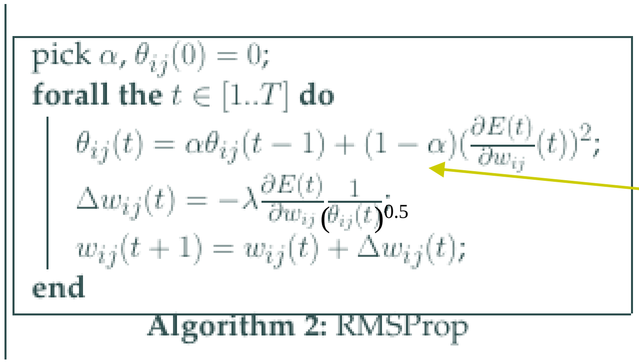
Abstract

In this paper, a new globally convergent modification of the Resilient Propagation-Rprop algorithm is presented. This new addition to the Rprop family of methods builds on a mathematical framework for the convergence analysis that ensures that the adaptive local learning rates of the Rprop's schedule generate a descent search direction at each iteration. Simulation results in six problems of the PROBEN1 benchmark collection show that the globally convergent modification of the Rprop algorithm exhibits improved learning speed, and compares favorably against the original Rprop and the Improved Rprop, a recently proposed Rrpop modification.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Supervised learning; Batch learning; First-order training algorithms; Convergence analysis; Global convergence property; Rprop; IRprop

Rprop for deep learning- see [Adapting Resilient Propagation for Deep Learning]

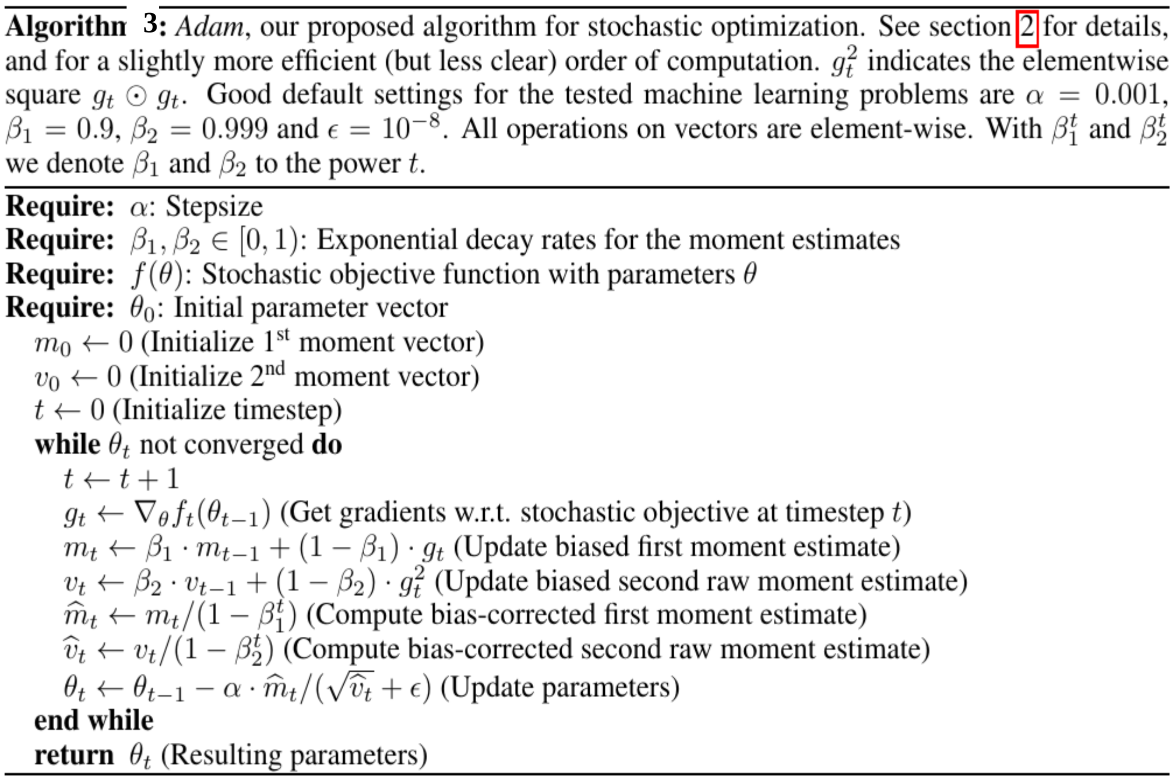
From Rprop to RMSProp and Adam



```
pick $\alpha$, $\theta_{ij}(0) = 0$;
forall the $t \in [1..T]$ do
    $\theta_{ij}(t) = \alpha\theta_{ij}(t-1) + (1-\alpha)(\frac{\partial E(t)}{\partial w_{ij}}(t))^2$;
    $\Delta w_{ij}(t) = -\lambda \frac{\partial E(t)}{\partial w_{ij}} \frac{1}{(\theta_{ij}(t))^{0.5}}$;
    $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$;
end
```

Algorithm 2: RMSProp

Exponentially weighted moving averaging (decay rate α)



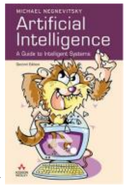
Algorithm 3: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

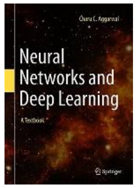
```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Summary

- Presented assumptions for the development of neural architectures and approaches for training.
- Introduced models of neurons, neural network architectures, and supervised learning.
- Described the operation of multilayer perceptons and gradient-based supervised learning rules.
- Revision questions at: https://moodle.bbk.ac.uk/mod/resource/view.php?id=2240305

Useful Reading





Negnevitsky, Artificial Intelligence: a Guide to Intelligent Systems, Chapters 6.1-6.5 (available at the BBK e-Library -Kortext).

Aggarwal, Neural Networks and Deep Learning, Chapters 1, 3 (available at the BBK e-Library- Kortext)

Du K-L, Leung C-S, Mow WH, Swamy MNS. Perceptron: Learning, Generalization, Model Selection, Fault Tolerance, and Role in the Deep Learning Era

Rojas R. (1996), Neural Networks-A Systematic Introduction, chapters 7-8.

Hagan Martin T., Demuth Howard B., Beale Mark H. (1996), Neural Network Design, chapter 2, 9, 11, 12

Anastasiadis A., Magoulas G.D., and Vrahatis M.N., New Globally Convergent Training Scheme Based on the Resilient Propagation Algorithm, Neurocomputing.

Kingma D.P., Ba J., Adam: A Method for Stochastic Optimization

Next week

Neural architectures and learning algorithms – to be continued

From simple recurrent nodes to Recurrent Neural Networks, the BBTT, Deep recurrent CNN, Deep residual learning

Preparation

Werbos, P.J., (1990) Backpropagation through time: What it does and How to do it, Proceedings of the IEEE, vol. 78, 10.

Ronald J. Williams and Jing Peng (1990), An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories.