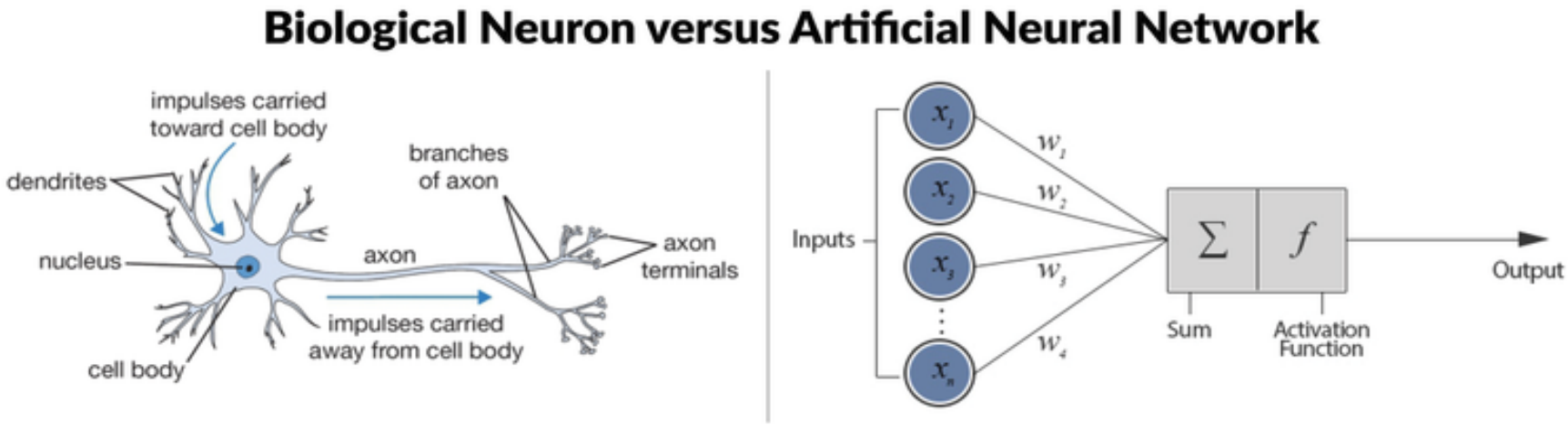
Neural Networks and Deep Learning

Dr Alfred Krzywicki
University of Adelaide

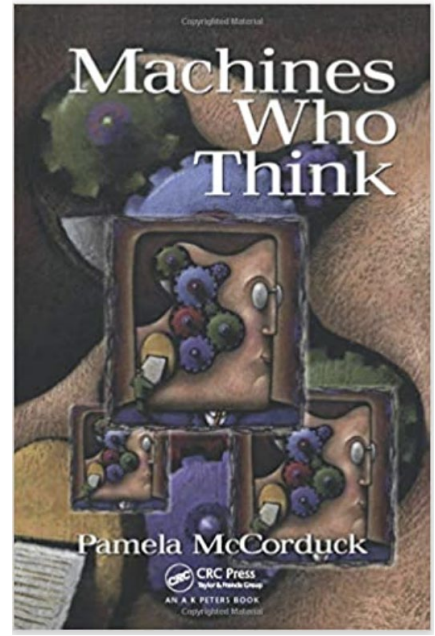
Natural Model for NN



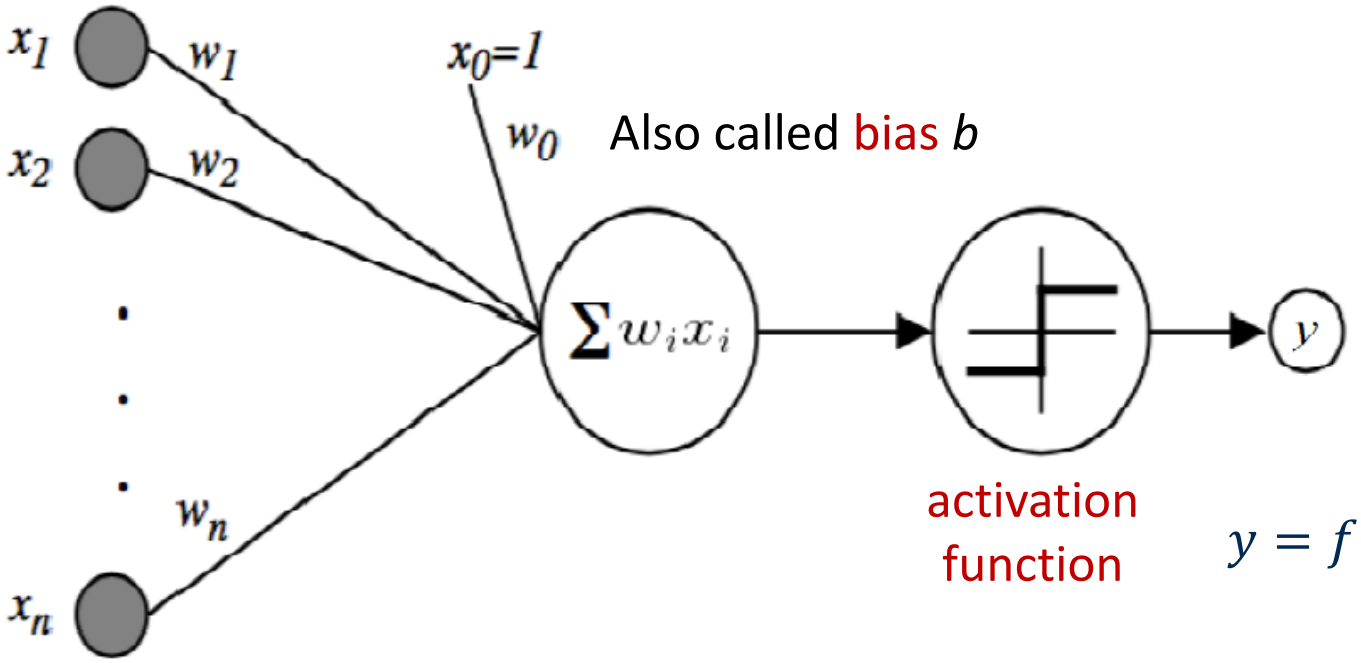
- Each **neuron** has multiple **dendrites** and a single **axon**. The neuron receives its **inputs from its dendrites** and transmits its **output through its axon**. Both inputs and outputs take the form of **electrical impulses**. The neuron sums up its inputs, and if the total electrical impulse strength exceeds the neuron's firing threshold, the **neuron fires off a new impulse along its single axon**. The **axon**, in turn, **distributes the signal along** its branching synapses which collectively reach thousands of neighbouring neurons.
- https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b

NN: a Brief History

- Theoretical base started in 1873-1890 (A. Bain, W. James)
- In 1943 McCulloch and Pitts created a computational model (called “threshold logic”) for neural networks based on mathematics and algorithms.
- In 1958 Rosenblatt created the perceptron algorithm
- 1969: Minsky and Papert proved that linear perceptron cannot process XOR function, research on NN stopped.
- 1986: Rumelhart, Hinton & Williams: backpropagation, and NN research resumed.
- 1990s: more complex NN needed more power not available then
- 2000s: revival of NN, DL using power and distributed computing



Artificial Neuron Model



$$a(\boldsymbol{x}) = \sum_{i=0}^{n} x_i w_i$$

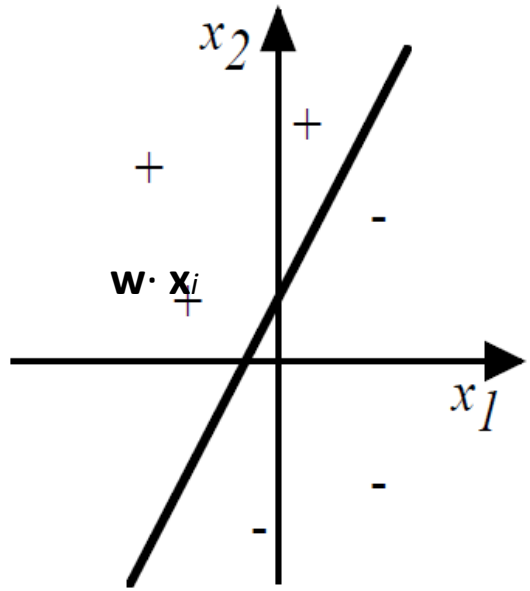
$$a(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w}$$

$$y = f(a(x)) = \begin{cases} 1 & if \;\; a(\boldsymbol{x}) \geq 0 \\ -1 & if \;\; a(\boldsymbol{x}) < 0 \end{cases}$$

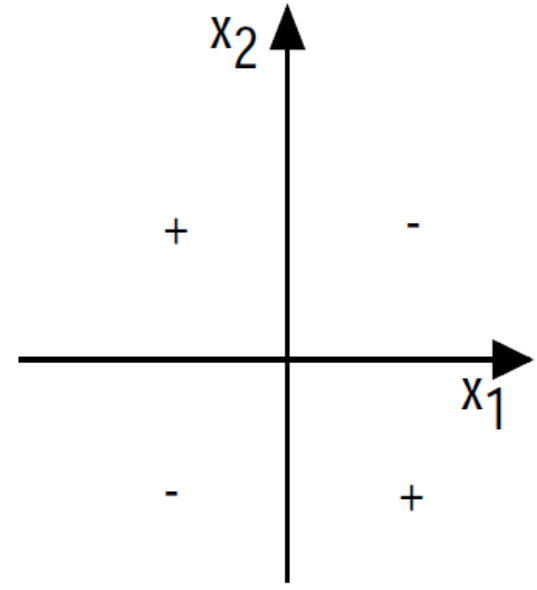
- Output $a(\boldsymbol{x})$ is sum of products of inputs and their weights. The f function is called activation function.

Main Limitation of Perceptron

- Single perceptron cannot represent functions that are not linearly separable



Linearly separable



Non-linearly separable (XOR)

Perceptron Learning Algorithm

Initialise the weight and the threshold values randomly.

For each data object in the training data set, check whether the perceptron predicts the correct class.

If the perceptron predicts the wrong class, adjust the weights and threshold value to **improve** the prediction

Repeat this until no changes occur

Perceptron Learning Algorithm cont.

The **delta rule** recommends to adjust the weight and the threshold values as:

$$a(\boldsymbol{x}) = \sum_{i=0}^{n} x_i w_i + b \qquad w_i^{new} = w_i^{old} + \Delta w_i$$

$$a(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w} + b \qquad b^{new} = b^{old} + \Delta b$$

- w_i: A weight of the perceptron
- b : The threshold value of the perceptron
- $(x_1, x_2, \ldots, x_n)$: An input vector

Perceptron Learning Algorithm cont.

$$\Delta w_i = \begin{cases} 0 & if\ y = t \\ +\eta x_i & if\ y = 0\ and\ t = 1 \\ -\eta x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ y = t \\ +\eta & if\ y = 0\ and\ t = 1 \\ -\eta & if\ y = 1\ and\ t = 0 \end{cases}$$

- y: the real output of the Perceptron for input vector $(x_1, x_2, \dots, x_n)$
- t: the desired output for input vector $(x_1, x_2, \dots, x_n)$
- $\eta > 0$: the Learning rate, selected by in cross-validation

Simple Example: Learning AND

Training Data: {((0,0),0), ((0,1),0), ((1,0),0), ((1,1), 1)}

Learning rate $\eta = 1$, we also assume [0,1] activation function (not [-1,1])

Initialization: $w_1 = w_2 = b = 0$

$$y = f(a(x)) = \begin{cases} \mathbf{1} & \boldsymbol{if}\ \ \boldsymbol{a(x) \geq 0} \\ \mathbf{0} & \boldsymbol{if}\ \ \boldsymbol{a(x) < 0} \end{cases}$$

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
							0	0	0
Epoch 1	0 0	0	1	0	0	-1	0	0	-1
	0 1	0	0	0	0	0	0	0	-1
	1 0	0	0	0	0	0	0	0	-1
	1 1	1	0	1	1	1	1	1	0

Epoch: one cycle through the training data, *t* is target, *y* is actual output

Perceptron Convergence Theorem

For any finite set of linearly separable labelled examples, the delta rule will adjust the weights and the threshold after a finite number of steps in such way that all patterns are classified correctly.

BP and Gradient Descent (online)

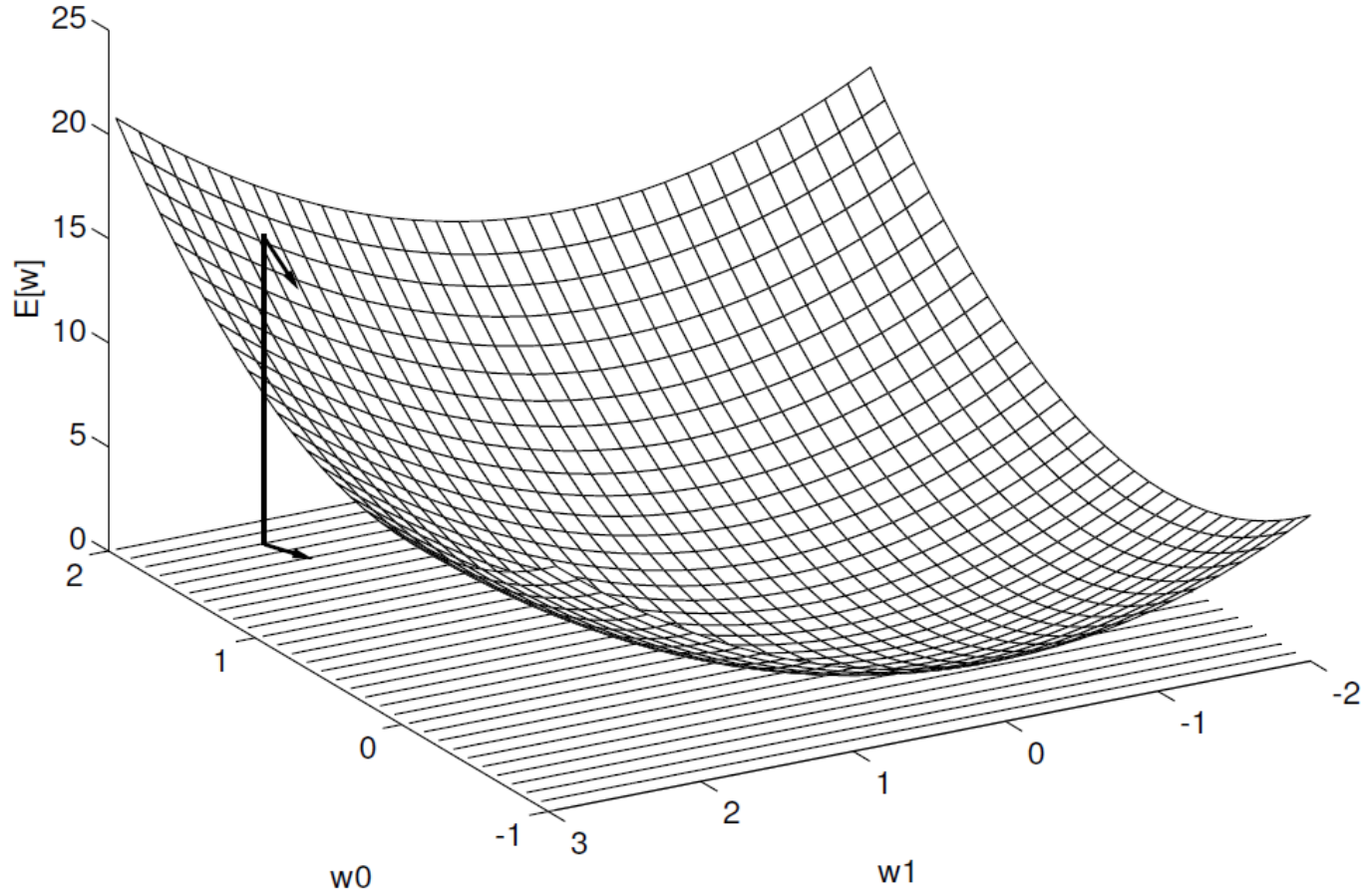
- Consider sigmoid function as the activation function

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

- Consider complex NN with hidden layers
- Need to update weights for all units using backpropagation algorithm (BP) calculated with gradient descent
- BP: forward pass: calculate outputs
- BP backward pass: update weights using square error (or different loss function) for each unit

Gradient Descent

- Weights need to change to get the min error on an error surface, which is unknown function of weights. Do it in small steps following EM.



Gradient Descent

Adjust the weights proportionally to the gradient of the error function

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \Delta\boldsymbol{w}(t)$$

with

$$\Delta\boldsymbol{w}(t) = -\eta \nabla(E(\boldsymbol{w}(t)))$$

with $\eta > 0$ a non-zero learning rate

$$\Delta\boldsymbol{w}(t) = -\eta \nabla\left(E\left(\boldsymbol{w}(t)\right)\right) = -\eta \left(\frac{\partial E(\boldsymbol{w}(t))}{\partial w_1}, \ldots, \frac{\partial E(\boldsymbol{w}(t))}{\partial w_m}\right)$$

We really need to determine:

$$\Delta w_{u,v} = -\eta \frac{\partial E}{\partial w_{u,v}}$$

for **each weight** of the network connecting any pair of nodes (u, v)

Gradient Descent Derivation (optional)

Assume we have $\sigma(x)$ activation function with its nice derivative

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial}{\partial x}\left(\frac{1}{1+e^{-x}}\right) = -\frac{e^{-x}}{(1+e^{-x})^2} = \boldsymbol{\sigma(x)\big(1-\sigma(x)\big)}$$

Assume for now we have only one output of the network

Assume we minimising squared error (similar to MSE)

$$E = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

Chain rule for derivatives:

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

e.g., $\frac{\partial (e^{4x^2})}{\partial x} = \frac{\partial e^{4x^2}}{\partial (4x^2)} \times \frac{\partial (4x^2)}{\partial x} = e^{4x^2} \times 8x$

Gradient Descent Derivation cont. (optional)

D is a set of training examples

Chain rule: $\frac{df(g(x)}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$

Target t_d does not depend on w

Target o_d is a function of the activation function σ and w_i is the sum of weights coming into node:

$$o_d = (\sigma(net_d(w_i))$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\
&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

Gradient Descent Derivation cont.(optional)

Derivative of activation function

$\frac{\partial \sigma(net_d)}{\partial x} = \sigma(net_d)\big(1 - \sigma(net_d)\big)$

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

All w other than w_i are constants, so derivative is 0

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\mathbf{w} \cdot \mathbf{x}_d)}{\partial w_i} = x_{i,d}$$

Finally:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

BP Algorithm

- Initialize all weights to small random numbers.
- Until satisfied*, Do
- For each training example, Do
  - Input the training example to the network and
  - compute the network outputs $o_k = f(\sum_{i=1}^{n} w_{ki} x_i)$ where f is activation function
  - For each output unit *k*:

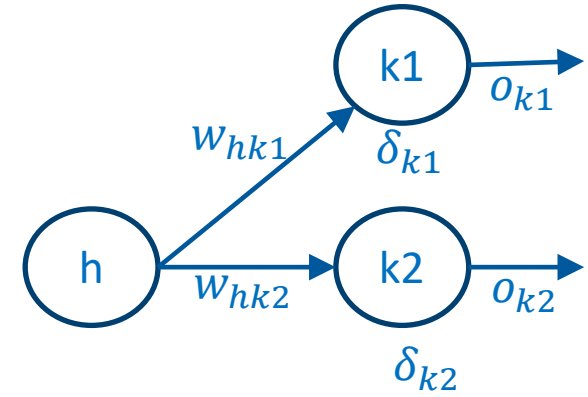
$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  - For each hidden unit *h*, δ_k comes from all k units $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k$ so here they are

$$o_h(1 - o_h)(w_{k1h}\delta_{k1} + w_{k2h}\delta_{k2})$$

  - Update each network weight from unit *j* to *i*, w_{ji}

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E}{w_{ji}} = w_{ji} + \eta \delta_j x_{ji}$$



* satisfied means complete given number of epochs, or satisfying another condition, e.g. validation accuracy

Offline and Online GD

Online training

```
loop maxEpochs times
  for each training data item
    compute weights and bias deltas for curr item
    adjust weights and bias values using deltas
  end for
end loop
```

Batch training

```
loop maxEpochs times
  for each training item
    compute weights and bias deltas for curr item
    accumulate the deltas
  end for
  adjust weights and bias deltas using accumulated deltas
end loop
```

https://visualstudiomagazine.com/articles/2014/08/01/batch-training.aspx

Offline and Online GD

Offline Training: Weight update after **all** training patterns

- correct
- computationally expensive and slow
- works with reasonably large learning rates (fewer updates!)

- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

Online/Stochastic Training: Weight update after **each** training pattern/each batch of random instances

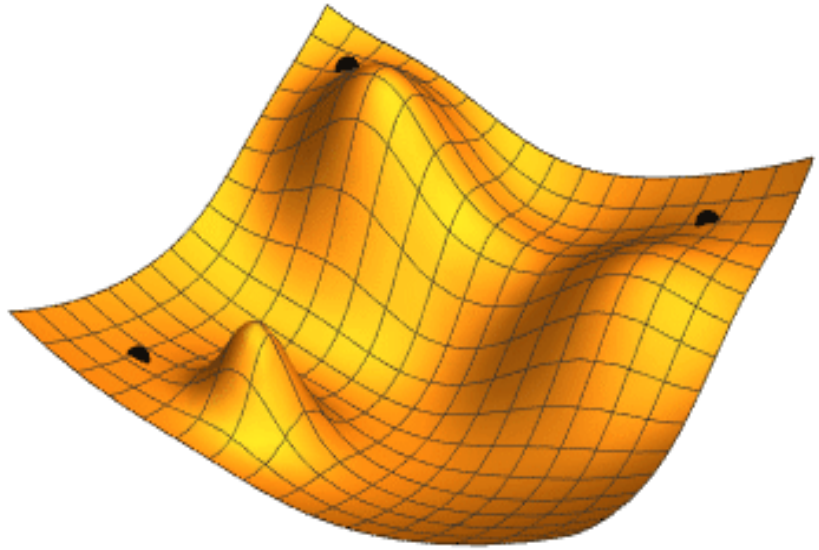
- approximation (can in theory run into oscillations)
- faster (fewer epochs!)
- smaller learning rates necessary

- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

Batch Training: Weight update after a batch of training patterns

- A compromise between the two

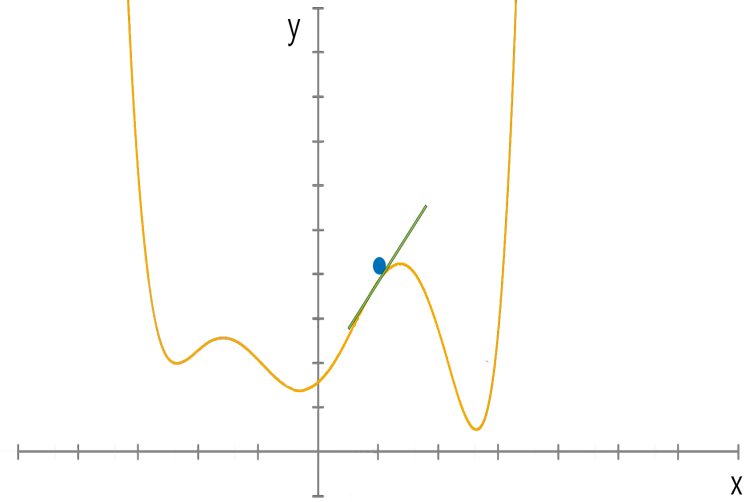
Gradient Descent in Action



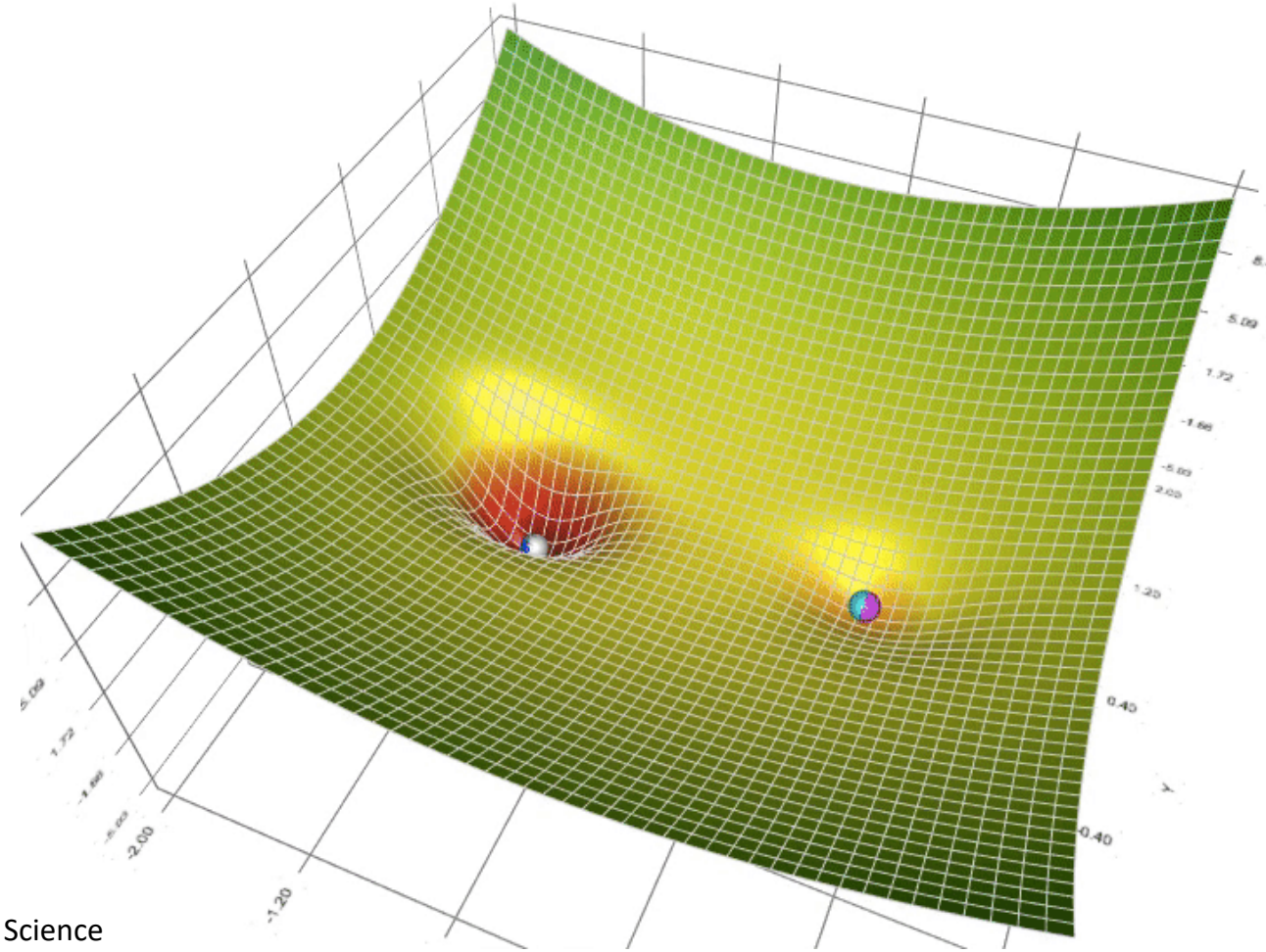
File:Gradient descent.gif
- Wikimedia Commons

- Often gradient descent ends up in a local min.

Adatis
Introduction to Artificial
Neural Networks part two

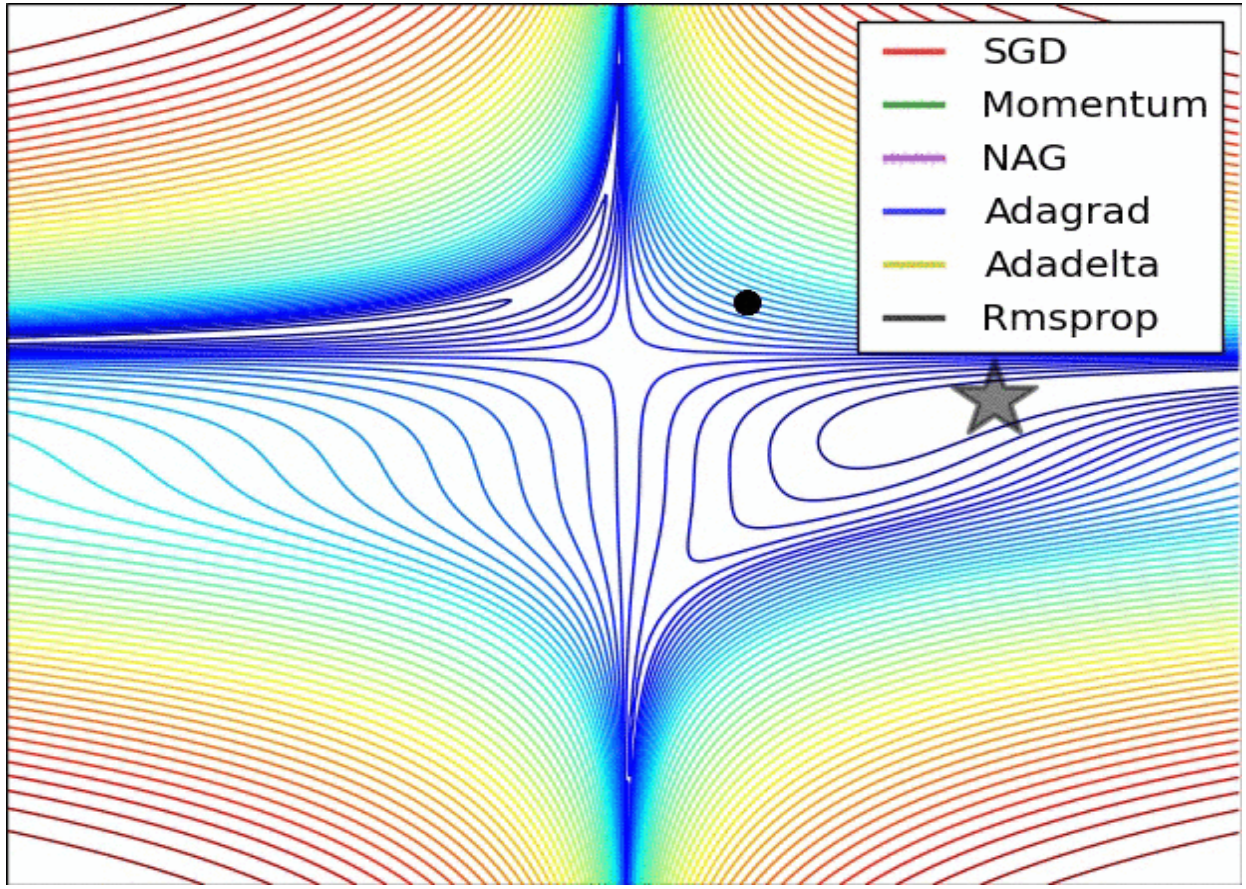


Gradient Descent in Action



Towards Data Science
A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp

Gradient Descent in Action



$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

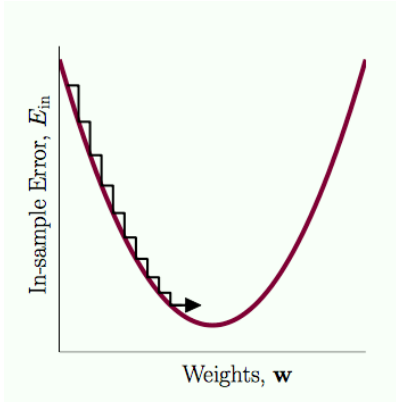
Nesterov accelerated gradient

- Which one of these
- performs best?

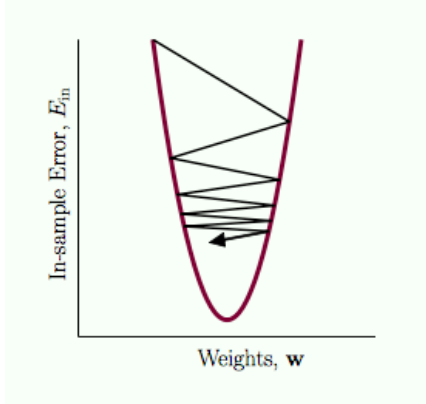
Excellent review of GD methods can be found here:
https://ruder.io/optimizing-gradient-descent/

Improving BP

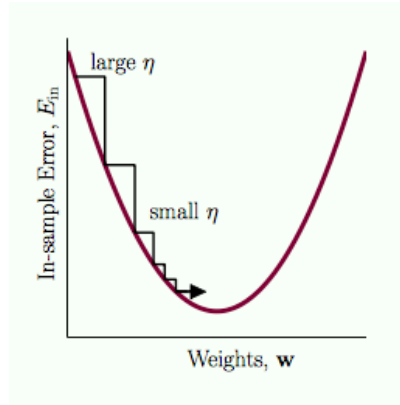
η too small



η too large



η just right

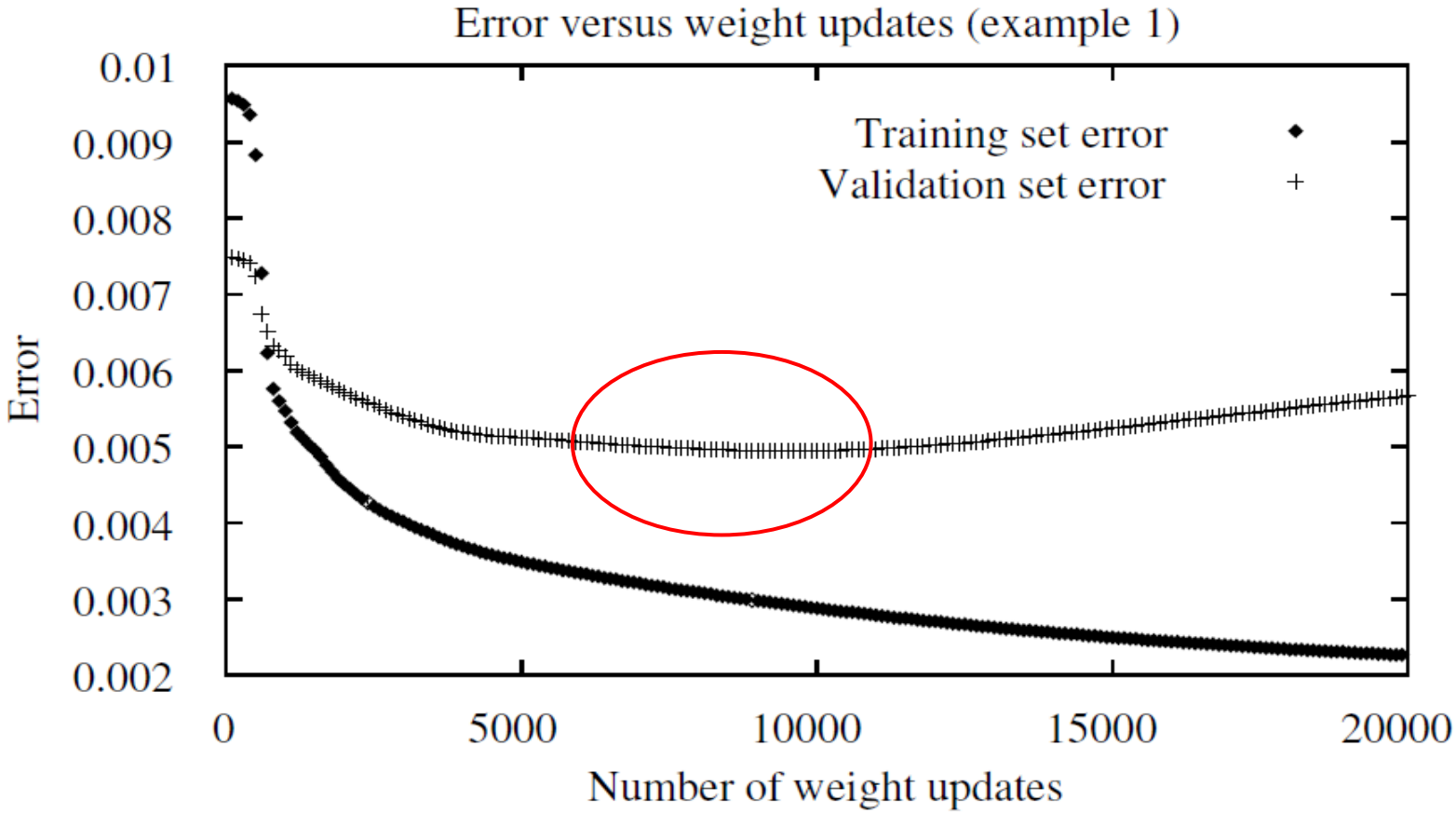


- Can be selected using cross-validation, usually <0.5
- Use momentum for smoother descent

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- Select initial random weights few times, sometimes the start point may be closer to better (global) minimum.

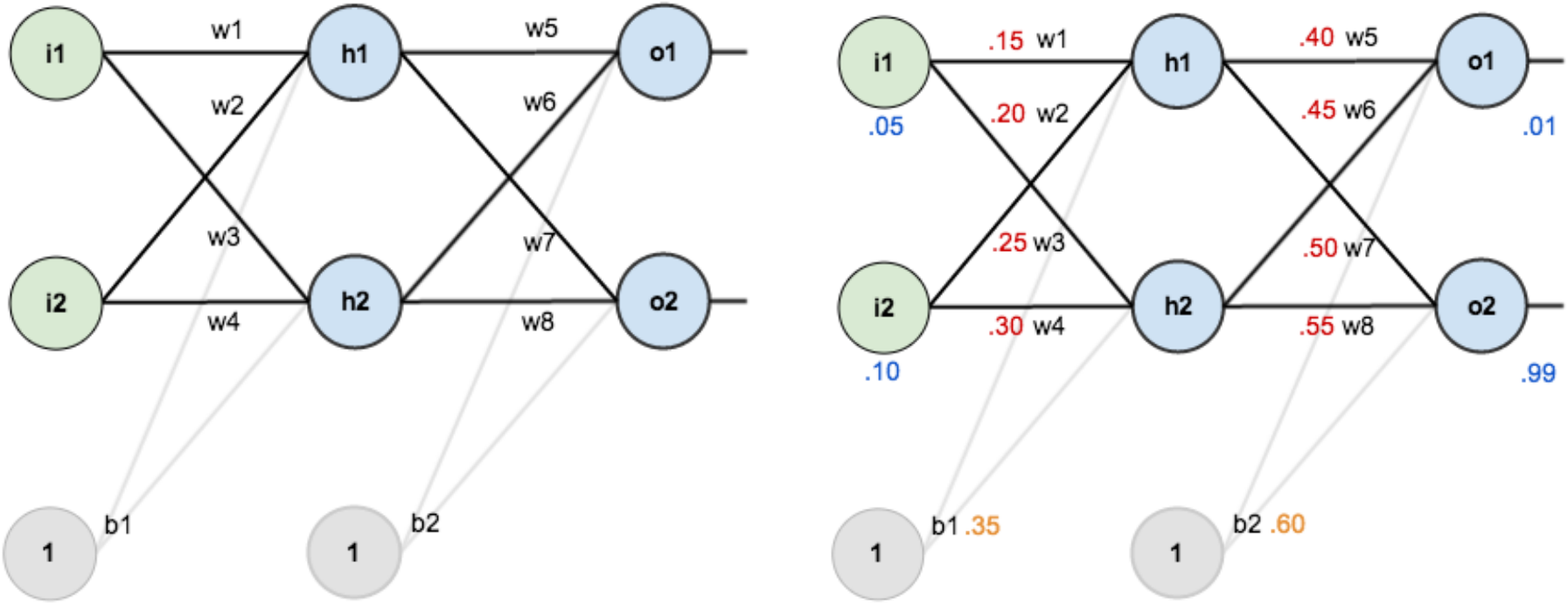
How much training is needed?



- Stop when validation error gets to min

Example of BP calculation (optional)

Example of BP calculation (1) (optional)



Dataset: input [0.05, 0.10]. Target [0.1, 0.99]

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Example of BP calculation (2) forward pass

Calculate h1 unit input to activation sigmoid:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

Calculate h1 unit output after sigmoid:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

The same for h2 gives

$$out_{h2} = 0.596884378$$

Example of BP calculation (3) forward pass

Calculate o1 unit:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

The same for o2 gives:

$$out_{o2} = 0.772928465$$

Example of BP calculation (4) error derivatives

Error total is the sum of errors from the two output units. We need to calculate error derivatives with respect to weights, e.g., w5

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

Only out_o1, because out_o2 does not depend on w5

Let's start from derivative with respect of o1 output:

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

* -1, because (–Out/dOut)=-1

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Example of BP calculation (5) error derivatives

Next derivative:

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Finally:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

Example of BP calculation (5) adjusting output layer weights

Weight for w5:

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

Similarly, for w6, w7, w8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

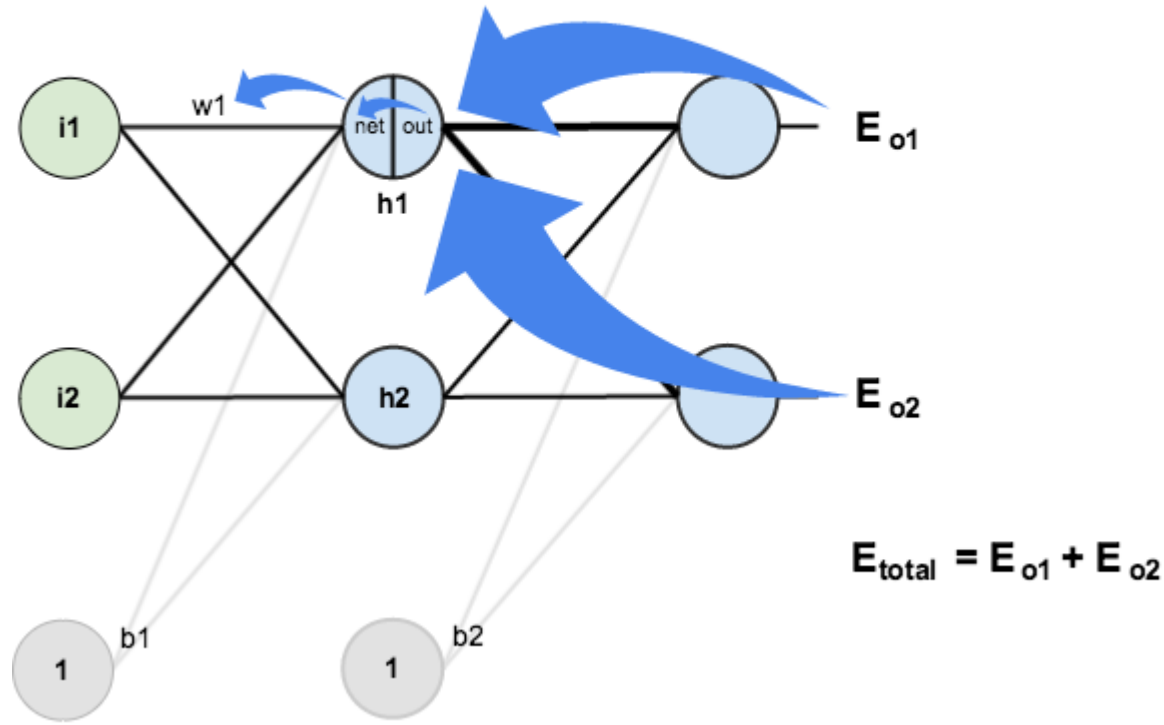
Example of BP calculation (6) adjusting hidden layer weights

Starting with weight for w1:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



Example of BP calculation (7) adjusting hidden layer weights

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

From previous calculations, we have:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

Now $\frac{\partial net_{o1}}{\partial out_{h1}}$

Since $net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$

Then $\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$

So $\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$

Example of BP calculation (8) adjusting hidden layer weights

In similar way:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Now, that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w}$ for each weight

Example of BP calculation (9) adjusting hidden layer weights

In similar way:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

Example of BP calculation (10) adjusting hidden layer weights

Now we can update w1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Repeating this process for w2, w3 and w4:

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$