An idea is inspired by the science of the brain
How computer works
How brain works: neurons

*Neuron* is an electrically excitable cell that processes and transmits information by electrical and chemical signaling.
Mathematical model of a neuron (McCulloch and Pitt, 1943)

\[ \text{Input neurons (x)} \]

\[ x_1, x_2, \ldots, x_i \]

\[ w_1, w_2, \ldots, w_i \]

\[ \text{“Neuron”} \]

\[ \text{IN} = \sum (i) x_i w_i \]

\[ a = g(\text{IN}) \]

\[ y \]

Output neuron
An input vector \( \mathbf{x} \) is the data given as one input to the processing “neuron” (corresponds to afferent neurons that transmit information to the brain).
How real neurons communicate

• The signal is transmitted to other neurons through synapses.

• The physical and neurochemical characteristics of each synapse determine the strength and polarity of the new input signal.

• This is where the brain is the most flexible: neuroplasticity.
Real neurons: signal summation

- **Dendrite(s)** receive an electric charge.
- The strengths of all the received charges are added together (spatial and temporal **summation**).
- The aggregate value is then passed to the soma (cell body) to **axon hillock**.
Real neurons: activation threshold

- If the aggregate input is greater than the axon hillock's **threshold** value, then the neuron **fires**, and an output signal is transmitted down the axon.
Real neurons: the output signal is constant

• **The strength of the output is constant**, regardless of whether the input was just above the threshold, or a hundred times as great.

• This uniformity is critical in an analogue device such as a brain where small errors can snowball, and where error correction is more difficult.
Modeling brain with networks

• The complicated biological phenomena may be modeled by a very simple model: nodes model neurons and edges model connections.

• The input nodes each have a weight that they contribute to the neuron, if the input is active. This corresponds to the strength of a synaptic connection.
Model: signal strength (weights)

Input vector \((\mathbf{x})\)

```
\[ x_1, w_1 \]
\[ x_2, w_2 \]
\[ x_i, w_i \]
```

"Neuron"

\[ \mathbf{IN}=\sum (i)x_iw_i \]

\[ a=g(\mathbf{IN}) \]

Output: 0 or 1

- Weights \(w_i\), are the weighted connections between input neurons and the processing neuron (these weights model the strength of synaptic connections in the brain).
The summation function $IN$ sums all the signals from the input vector multiplied by weights, and feeds the result into activation function $g$. 
The output $y$, shows the resulting action of processing neuron: neuron fires(1) or not(0).

We can write $y(x, W)$ to remind that the output depends on the inputs to the algorithm and the current set of weights of the network.
The activation function $g(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs.

As in real brain, this is a threshold function: neuron either fires, or not.
Simple threshold: \textit{sign}

Input vector \((x)\)

Output: 0 or 1

\[ \text{IN} = \sum_{i} x_i w_i \]

Activation function should be \textbf{threshold} function

The simplest threshold function: \textit{sign}
\[ g(x) = 0 \text{ if } x \leq 0 \]
\[ g(x) = 1 \text{ if } (x > 0) \text{ (neuron fires)} \]
Model: the goal – predict $y$

The model can be used to **predict a target variable $y$ given input vector $x$**.

Each input dimension (attribute) can be considered a separate input “neuron”

Processing happens in the “axon” and based on the result the output neuron “fires” (or not)
Conceptually there is no difference between input and output neurons. So the same input vector can be used to activate multiple output "neurons", using a different set of weights.
Let’s build some neural networks

Networks that know the meaning of lights
Predicting smiles

- We record people’s reaction to lights into a table (dataset).
- Can we set up a single network which when presented with a combination of lights will correctly predict if a person will smile?

- Setting up the network means labeling the edges with correct weights.
Bias node

- When we are presenting the network with combination [0, 0] - then the weights do not matter: the data vector [0,0] is ignored by the network.
- To prevent this information loss, we add to the input a special **bias node** which always has a **constant** value, and we assign to it weight \( b \)

\[
\begin{array}{c|c|c|c}
\text{red} & \text{orange} & \text{smile} \\
\hline
\text{neg} & \text{pos} & \text{pos} \\
\text{pos} & \text{pos} & \text{pos} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{x}_1 & \text{x}_2 & \text{y} \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]
Network that predicts smiles

Assigning sample weights

<table>
<thead>
<tr>
<th>red</th>
<th>orange</th>
<th>smile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>neg</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pos</td>
</tr>
<tr>
<td>red</td>
<td></td>
<td>pos</td>
</tr>
<tr>
<td>red</td>
<td>orange</td>
<td>pos</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Bias: -0.5

$y$: smile

$x_1$: red

$x_2$: orange
Network that predicts smiles

Checking correctness of predictions

\[ y([0, 0]) = -0.5 \]  
\[ y([0, 1]) = 1 - 0.5 = 0.5 \]  
\[ y([1, 0]) = 1 - 0.5 = 0.5 \]  
\[ y([1, 1]) = 2 - 0.5 = 1.5 \]

\[ x_1 \quad x_2 \quad y \]
\[ 0 \quad 0 \quad 0 \]
\[ 0 \quad 1 \quad 1 \]
\[ 1 \quad 0 \quad 1 \]
\[ 1 \quad 1 \quad 1 \]
Predicting two outputs

There is no conceptual difference between input and output nodes
Predicting both smiles and stops

Assigning sample weights for $y_1$

We already know that this prediction is correct:

$y_1([0,0]) = -0.5 (-)$
$y_1([0,1]) = 1 - 0.5 = 0.5 (+)$
$y_1([1,0]) = 1 - 0.5 = 0.5 (+)$
$y_1([1,1]) = 2 - 0.5 = 1.5 (+)$
Predicting both smiles and stops

Assigning sample weights for $y_2$

We already know that this prediction is correct:

$y_1([0,0]) = -0.5$ (-)
$y_1([0,1]) = 1 - 0.5 = 0.5$ (+)
$y_1([1,0]) = 1 - 0.5 = 0.5$ (+)
$y_1([1,1]) = 2 - 0.5 = 1.5$ (+)
Predicting both smiles and stops

Checking $y_2$

- $y_1([0,0]) = -0.5 (-)$
- $y_1([0,1]) = 1 - 0.5 = 0.5 (+)$
- $y_1([1,0]) = 1 - 0.5 = 0.5 (+)$
- $y_1([1,1]) = 2 - 0.5 = 1.5 (+)$

- $y_2([0,0]) = -1.5 (-)$
- $y_2([0,1]) = 1 - 1.5 = -0.5 (-)$
- $y_2([1,0]) = 1 - 1.5 = -0.5 (-)$
- $y_2([1,1]) = 2 - 1.5 = 0.5 (+)$
We have built the system that recognizes OR and AND

Apply \textit{sign} function to the output

\begin{align*}
y_1([0,0]) &= \text{sign}(-0.5) = \text{true} \\
y_1([0,1]) &= \text{sign}(1 - 0.5) = \text{true} \\
y_1([1,0]) &= \text{sign}(1 - 0.5) = \text{true} \\
y_1([1,1]) &= \text{sign}(2 - 0.5) = \text{true} \\
y_2([0,0]) &= \text{sign}(-1.5) = \text{false} \\
y_2([0,1]) &= \text{sign}(1 - 1.5) = \text{false} \\
y_2([1,0]) &= \text{sign}(1 - 1.5) = \text{false} \\
y_2([1,1]) &= \text{sign}(2 - 1.5) = \text{true}
\end{align*}

Function \( g: \text{sign} \)

\( g(x) = 0 \) if \( x \leq 0 \)
\( g(x) = 1 \) if \( x > 0 \)

Truth table for OR
\begin{tabular}{|c|c|c|}
\hline
\( x_1 \) & \( x_2 \) & \( y_1 \) \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\hline
\end{tabular}

Truth table for AND
\begin{tabular}{|c|c|c|}
\hline
\( x_1 \) & \( x_2 \) & \( y_1 \) \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\hline
\end{tabular}
Can we build a system that recognizes: $x_1 \text{ AND NOT } x_2$?

Truth table for AND NOT:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
System that recognizes:
$x_1$ AND NOT $x_2$

Truth table for AND NOT

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
System that recognizes:
$x_1$ AND NOT $x_2$

Truth table for AND NOT

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$y([0,0]) = \text{sign}(-0.5) = 0$
$y([0,1]) = \text{sign}(0 - 1 - 0.5) = 0$
$y([1,0]) = \text{sign}(1 + 0 - 0.5) = 1$
$y([1,1]) = \text{sign}(1 - 1 - 0.5) = 0$
How about: NOT (x₁ AND x₂)

Truth table for NOT AND

<table>
<thead>
<tr>
<th>x₁</th>
<th>x₂</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
System that recognizes: NOT \((x_1 \text{ AND } x_2)\)

\[
y([0,0]) = \text{sign}(1.5) = 1
y([0,1]) = \text{sign}(-1 + 1.5) = 1
y([1,0]) = \text{sign}(-1 + 1.5) = 1
y([1,1]) = \text{sign}(-2 + 1.5) = 0
\]

Truth table for NOT AND

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Our network is able to recognize linearly-separable binary classes

$AND \quad x_2 \quad x_1$

$OR \quad x_2 \quad x_1$

$NOT AND \quad x_2 \quad x_1$

$AND NOT \quad x_2 \quad x_1$
Why it works

- The network assumes that there is a linear correlation between the input vector $x$ and the output $y$

- We just need to discover the equation of the separating line (hyperplane) $y = wx + b$, which expresses this linear correlation
Can machines learn the network parameters for a given problem automatically?

Yes, by looking at the labeled dataset (supervised learning)

\[ \text{Predict} \rightarrow \text{Compare} \rightarrow \text{Learn} \text{ from errors \} \]
Neuron with learning capabilities: *Perceptron* (Rosenblatt, 1958)

- The network can learn its own weights
- It is presented with a set of inputs and known outputs
- Originally the predicted output is different from the actual output by some error
- We adjust the connection weights to produce a smaller error

![Most basic Perceptron diagram](image-url)
Adjusting the weights with gradient descent: error

Objective error function - in this case:
\[ E = \frac{1}{2} (y - t)^2 \]

where \( y = w \times x \) (predicted value), and \( t \) is the actual value of \( y \), known from the labeled dataset

\[ \frac{\partial E}{\partial y} = \frac{1}{2} \times 2 \times (y - t) = y - t \]
Adjusting the weights with gradient descent: derivative

\[ E = \frac{1}{2} (y - t)^2 \]

\[ y = w \cdot x \]

\[ \frac{\partial E}{\partial y} = y - t \]

To determine how to change weight \( w \) take derivative of \( E \) at point \( w \)

\[ \Delta = \frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial w} = (w \cdot x - t) \cdot x \]

If derivative is positive (function on the rise) we need to decrease the weight, if it is negative - we need to increase the weight.
Adjusting the weights with gradient descent: delta rule

\[ E = \frac{1}{2} (y - t)^2 \]
\[ y = w \cdot x \]
\[ \frac{\partial E}{\partial y} = y - t \]
\[ \Delta = \frac{\partial E}{\partial w} = (w \cdot x - t) \cdot x \]

**Delta rule**: adjust weight \( w \) by \( \Delta \)
\[ w = w - \frac{\partial E}{\partial w} = w - \Delta = w - (w \cdot x - t) \cdot x \]
More input dimensions - more weights to adjust

The network transforms input feature vector into target using two weights

\[
\begin{align*}
    w_1 \quad x_1 & \quad \text{y} \\
    w_2 \quad x_2 & \quad \text{y}
\end{align*}
\]

The principle is the same:

\[
E = \frac{1}{2} (w_1 x_1 + w_2 x_2 - t)^2
\]

Which weight contributed more to the error?

Partial derivatives with respect to each weight:

\[
\begin{align*}
    \frac{\partial E}{\partial w_1} &= (w_1 x_1 - t)x_1 \\
    \frac{\partial E}{\partial w_2} &= (w_2 x_2 - t)x_2
\end{align*}
\]

Delta rules: update weights

\[
\begin{align*}
    w_1 &= w_1 - \frac{\partial E}{\partial w_1} \\
    w_2 &= w_2 - \frac{\partial E}{\partial w_2}
\end{align*}
\]
There is also a **bias** node, of course

Objective function: \( E = \frac{1}{2} (w_1 x_1 + w_2 x_2 + b - t)^2 \)

\[
\frac{\partial E}{\partial w_1} = (w_1 x_1 - t) x_1 \\
\frac{\partial E}{\partial w_2} = (w_2 x_2 - t) x_2 \\
\frac{\partial E}{\partial b} = (b c - t) c
\]

Delta rule:

\( w_1 = w_1 - \frac{\partial E}{\partial w_1} \)

\( w_2 = w_2 - \frac{\partial E}{\partial w_2} \)

\( b = b - \frac{\partial E}{\partial b} \)
Incorporating learning rate $\eta$ (eta)

\[ w_1 = w_1 - \eta \frac{\partial E}{\partial w_1} \]
\[ w_2 = w_2 - \eta \frac{\partial E}{\partial w_2} \]
\[ b = b - \eta \frac{\partial E}{\partial b} \]

where:

\[ \frac{\partial E}{\partial w_1} = (w_1 x_1 - t) x_1 \]
\[ \frac{\partial E}{\partial w_2} = (w_2 x_2 - t) x_2 \]
\[ \frac{\partial E}{\partial b} = (cb - t) c \]
Let’s try to build a perceptron that recognizes XOR

Truth table for XOR

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Let’s try to build a perceptron that recognizes XOR

\[
\begin{array}{ccc}
1 & x_1 & w_1 \\
& x_2 & w_2 \\
\downarrow & \downarrow & \downarrow \\
\sum & y & b \\
\end{array}
\]

Truth table for XOR

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

We can’t!

This failure caused a major delay in developing the idea of ANN in the 60s
Experiment with basic perceptron here
Idea:
express XOR through known solutions

\[ x_1 \text{ XOR } x_2 = (x_1 \text{ OR } x_2) \text{ AND } (\text{NOT}(x_1 \text{ AND } x_2)) \]
Add more layers!

\[ x_1 \text{ XOR } x_2 = (x_1 \text{ OR } x_2) \text{ AND } (\text{NOT}(x_1 \text{ AND } x_2)) \]
Importance of nonlinearity!

\[ x_1 \text{ XOR } x_2 = (x_1 \text{ OR } x_2) \text{ AND } (\text{NOT}(x_1 \text{ AND } x_2)) \]

Just combining linear separators would not work! We have to add some sort of nonlinearity or *sometimes-correlation* between the layers.

Truth table for XOR:

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example Calculations**:

\[
\begin{align*}
h_1([0,0]) &= -0.5 (-) \rightarrow 0 \\
h_1([0,1]) &= 1 - 0.5 = 0.5 (+) \rightarrow 1 \\
h_1([1,0]) &= 1 - 0.5 = 0.5 (+) \rightarrow 1 \\
h_1([1,1]) &= 2 - 0.5 = 1.5 (+) \rightarrow 1 \\
h_2([0,0]) &= 1.5 (+) \\
h_2([0,1]) &= -1 + 1.5 = 0.5 (+) \rightarrow 1 \\
h_2([1,0]) &= -1 + 1.5 = 0.5 (+) \rightarrow 1 \\
h_2([1,1]) &= -2 + 1.5 = -0.5 (-) \rightarrow 0
\end{align*}
\]

\[
\begin{align*}
y([0,0]) &= (\cdot) \rightarrow 0 \\
y([0,1]) &= (+) \rightarrow 1 \\
y([1,0]) &= (+) \rightarrow 1 \\
y([1,1]) &= (\cdot) \rightarrow 0
\end{align*}
\]
Conclusion: neurons can be combined into multiple layers to create complex shapes from linear separation boundaries.
Multi-layer Perceptron (MLP)

• Added: *hidden nodes*
• Organized nodes into *layers*. Edges are directed and carry weight
• No connections inside the layer!
Multi-layer Perceptron: learning

Objective of learning - did not change: determine the optimal values of weights to separate all labeled instances by a hyperplane
MLP: learning optimal weights

Because we need derivatives: instead of `sign` - use more complex nonlinear functions: **sigmoidal** functions
Because we need derivatives: instead of sign - use more complex nonlinear functions: sigmoidal functions
Non-linear activation functions

**Logistic function** (sigmoid)

\[ g(h) = \frac{1}{1 + e^{-2\beta h}}, \]

where \( \beta \) is a positive constant (we generally use \( 2\beta = 1 \) obtaining a standard logistic function)

Alternatively can use **tanh**:

\[ \tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \]

which has the same shape as sigmoid but in range -1 to 1.

More recently - **rectified linear units (ReLU)**: \( f(x) = x^+ = \max(0, x) \)

This function is 0 for negative argument values, and some units will yield activations 0, making networks sparse. Moreover, the gradient is particularly simple—either 0 or 1.
MLP learning algorithm

Training the MLP consists of two parts:

• Working out what the outputs are for the given inputs and the current weights – *Forward* phase

• Updating the weights according to the error, which is a function of the difference between the outputs and the targets – *Backward* phase
Forward: prediction
Forward phase:
1. input-to-hidden layer: summation

\[ h_1 = w_1 x_1 + w_2 x_2 + b_1 \]
\[ h_2 = w_3 x_1 + w_4 x_2 + b_2 \]
Forward phase:
2. input-to-hidden layer: activation

\[ h_1 = w_1 x_1 + w_2 x_2 + b_1 \]
\[ h_2 = w_3 x_1 + w_4 x_2 + b_2 \]
\[ g_1 = \sigma(h_1) \]
\[ g_2 = \sigma(h_2) \]
Forward phase:
3. hidden-to-output layer: prediction

\[ h_1 = w_1 * x_1 + w_2 * x_2 + b_1 \]
\[ h_2 = w_3 * x_1 + w_4 * x_2 + b_2 \]
\[ g_1 = \sigma(h_1) \]
\[ g_2 = \sigma(h_2) \]
\[ y = g_1 * w_5 + g_2 * w_6 + b_3 \]
Step-by-step example
initialize weights at random

The input vector $x = [1, 4]$, and the actual output $t = 0.1$
Step-by-step example
1. input to hidden layer: summation

\[ h_1 = w_1 x_1 + w_2 x_2 + b_1 = 0.5 + 0.1 \times 1 + 0.2 \times 4 = 1.4 \]
\[ h_2 = w_3 x_1 + w_4 x_2 + b_2 = 0.5 + 0.3 \times 1 + 0.4 \times 4 = 2.4 \]
Step-by-step example

2. input to hidden layer: activation

\[ h_1 = 1.4 \]
\[ h_2 = 2.4 \]
\[ g_1 = \sigma(h_1) = 0.8021838885585817481543 \approx 0.80 \]
\[ g_2 = \sigma(h_2) = 0.9168273035060776293371 \approx 0.92 \]
3. **hidden-to-output** layer: prediction

\[ h_1 = 1.4 \]
\[ h_2 = 2.4 \]
\[ g_1 = 0.80 \]
\[ g_2 = 0.91 \]

\[ y = g_1 \cdot w_5 + g_2 \cdot w_6 + b_3 = 0.80 \cdot 0.5 + 0.92 \cdot 0.6 + 0.5 \approx 1.45 \]
Step-by-step example

compute error

\[ h_1 = 1.4 \]
\[ h_2 = 2.4 \]
\[ g_1 = 0.80 \]
\[ g_2 = 0.91 \]
\[ y = 1.45 \]

\[ E = \frac{1}{2} (1.45 - 0.1)^2 = 0.845 \]

Error directly depends on the weights \( w_5, w_6, \) and \( b_3 \)

\[ E = \frac{1}{2} (0.80 \times w_5 + 0.92w_6 + b_3 - 0.1)^2 \]

We try to make it smaller by simultaneously adjusting \( w_5, w_6, \) and \( b_3 \)
Backward phase:
4. output-to-hidden weight updates

\[ E = \frac{1}{2}(y - t)^2 \]
\[ y = g_1 * w_5 + g_2 * w_6 + b_3 \]

To find how to update \( w_5, w_6, \) and \( b_3 \)
Partial derivatives:
\[ \frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y} * \frac{\partial y}{\partial w_5} = (y - t) * g_1 \]
\[ \frac{\partial E}{\partial w_6} = (y - t) * g_2 \]
\[ \frac{\partial E}{\partial b_3} = (y - t) * 1 \]
Step-by-step example

4. **output-to-hidden** weight updates

\[ h_1 = 1.4 \]
\[ h_2 = 2.4 \]
\[ g_1 = 0.80 \]
\[ g_2 = 0.91 \]
\[ y = 1.45 \]

\[ \frac{\partial E}{\partial w_5} = (y - t) g_1 = (1.45 - 0.1) \times 0.80 = 1.08 \]
\[ \frac{\partial E}{\partial w_6} = (y - t) g_2 = (1.45 - 0.1) \times 0.92 = 1.24 \]
\[ \frac{\partial E}{\partial b_3} = (y - t) \times 1 = 1.45 - 0.1 = 1.35 \]

This tells us how much to update \( w_5, w_6, \) and \( b_3 \)
Step-by-step example
4. output-to-hidden weight updates

1. \( h_1 = 1.4 \)
2. \( h_2 = 2.4 \)
3. \( g_1 = 0.80 \)
4. \( g_2 = 0.91 \)
5. \( y = 1.45 \)

\[ \frac{\partial E}{\partial w_5} = 1.08 \]
\[ \frac{\partial E}{\partial w_6} = 1.24 \]
\[ \frac{\partial E}{\partial b_3} = 1.35 \]

Update weights \( \eta = 0.1 \):

\[ w_5 = 0.5 - 1.08 \times 0.1 = 0.39 \]
\[ w_6 = 0.6 - 1.24 \times 0.1 = 0.48 \]
\[ b_3 = 0.5 - 1.35 \times 0.1 = 0.37 \]
Step-by-step example
4. output-to-hidden weight updates

\[ h_1 = 1.4 \]
\[ h_2 = 2.4 \]
\[ g_1 = 0.80 \]
\[ g_2 = 0.91 \]
\[ y = 1.45 \]

Update weights \( \eta = 0.1 \):

\[ \frac{\partial E}{\partial w_5} = 1.08 \quad \Rightarrow \quad w_5 = 0.5 - 1.08 \times 0.1 = 0.39 \]
\[ \frac{\partial E}{\partial w_6} = 1.24 \quad \Rightarrow \quad w_6 = 0.6 - 1.24 \times 0.1 = 0.48 \]
\[ \frac{\partial E}{\partial b_3} = 1.35 \quad \Rightarrow \quad b_3 = 0.5 - 1.35 \times 0.1 = 0.37 \]

Note that this step is exactly the same as in a single-layer perceptron!
Backward phase:
5. hidden-to-output weight updates

Error function $E$ indirectly depends on $w_1$, $w_2$, $w_3$, $w_4$, $b_1$, $b_2$

To find the contribution of each variable: partial derivatives

For example:
$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial g_1} \cdot \frac{\partial g_1}{\partial w_1}$$

Chain rule!
Backward phase:  
5. hidden-to-output weight updates  

Computing delta for $w_1$
\[
\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial g_1} \cdot \frac{\partial g_1}{\partial w_1}
\]

$E(y) = \frac{1}{2}(y - t)^2$  \rightarrow  \frac{\partial E}{\partial y} = y - t  

$y(g_1) = g_1w_5 + g_2w_6 + b_3$  \rightarrow  \frac{\partial y}{\partial g_1} = w_5  

$g_1(w_1) = \sigma(h_1) = \sigma(w_1x_1 + w_2x_2 + b_1)$  \rightarrow  \frac{\partial g_1}{\partial w_1} = g_1(1 - g_1)x_1$

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$  
sigmoid derivative
Backward phase:
5. hidden-to-output weight updates

Computing delta for $w_1$

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial g_1} \cdot \frac{\partial g_1}{\partial w_1}$$

$$\Delta = \frac{\partial E}{\partial w_1} = (y - t) \cdot w_5 \cdot g_1 \cdot (1 - g_1) \cdot x_1$$

$$w_1 = w_1 - \eta \Delta$$
Step-by-step example
5. hidden-to-output weight update for $w_1$

$\Delta = \frac{\partial E}{\partial w_1} = (y - t) * w_5 * g_1 * (1 - g_1) * x_1$

$w_1 = w_1 - \eta \Delta$

$\Delta = (1.45 - 0.1) * 0.5 * 0.80 * 0.20 * 1 = 0.108$

Update $w_1$ using $\eta=0.1$:

$w_1 = 0.1 - 0.1 * 0.108 = 0.0892$
Role of nonlinearity

- Somewhere inside the hidden layer we **must** have a mechanism which will ignore some correlations.
- Otherwise the network will serve as a basic linear separator and be no better than a single-layer perceptron.
- You **may** also add nonlinearity to transform the output.

![Diagram](image)
Experiment with multi-layer-perceptron here
Multi-layer perceptron: vanilla (basic) neural networks

Some useful computations

Inputs

Hidden layer 1

Hidden layer 2

Outputs

Normal computing

Computing with MLP
What do we gain from the extra layers

1st layer draws linear boundaries

2nd layer combines the boundaries

3rd layer can generate arbitrarily complex boundaries
Very powerful model

• With sigmoidal activation function we can show that a 3-layer net can approximate any function to arbitrary accuracy: property of *Universal Approximation*

• Proof by thinking of superposition of sigmoids

• Not practically useful as we might need arbitrarily large number of neurons - more of an existence proof

• Same is true for a 2-layer net providing function is continuous and from one finite dimensional space to another
Universal Approximation Theorem

For any given constant $\varepsilon$ and continuous function $h(x_1,\ldots,x_m)$, there exists a three-layer ANN with the property that

$$\| h(x_1,\ldots,x_m) - H(x_1,\ldots,x_m) \| < \varepsilon$$

where $H(x_1,\ldots,x_m) = \sum_{i=1}^{k} a_i f\left( \sum_{j=1}^{m} w_{ij} x_j + b_i \right)$
Applications of ANNs

• Credit card frauds
• Kinect – gesture recognition
• Facial recognition
• Self-driving cars
• ...

Example: breast cancer diagnosis


• Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass

• Diagnosing breast cancer from mammograms is a very hard non-trivial task

Run and see how MLP learns to diagnose breast cancer
Make computers as capable as humans?

Brain is a highly complex, non-linear, massively-parallel system

• Response of integrated response circuit:
  1 nanosecond = 10^{-9} sec

• Response of neuron:
  1 millisecond = 10^{-3} sec

The only advantage of the brain: massively parallel – 10 billion neurons with 60 trillions of connections working together
Artificial neural network is an abstract idea – media-independent

- To simulate the brain we could theoretically construct thousands of circuits working in parallel
- We can simulate them using a program that is executed on a conventional serial processor
- The solutions are *theoretically* equivalent
- We can simulate the neural behavior by a virtual machine which is functionally identical to a real machine that currently is prohibitively complex and expensive to build