A1- Deep Learning

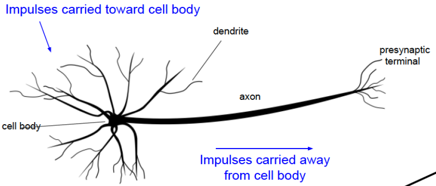
Melih Kandemir

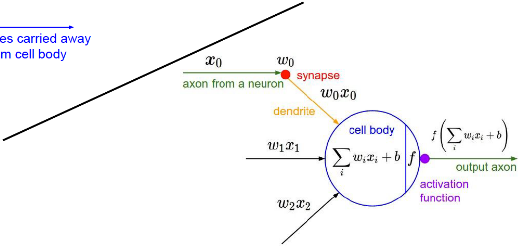
University of Southern Denmark
Department of Mathematics and Computer Science (IMADA)
kandemir@imada.sdu.dk

Fall 2022

Perceptron: The atom of intelligence

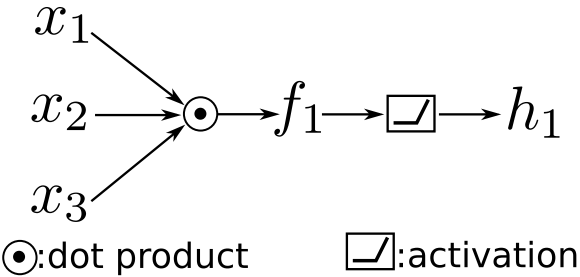
North-West: Biological neuron
South-East: Artificial neuron (perceptron)



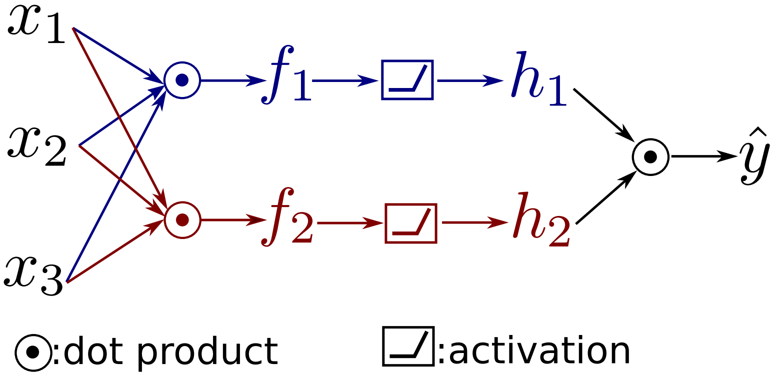


The computational graph of a perceptron

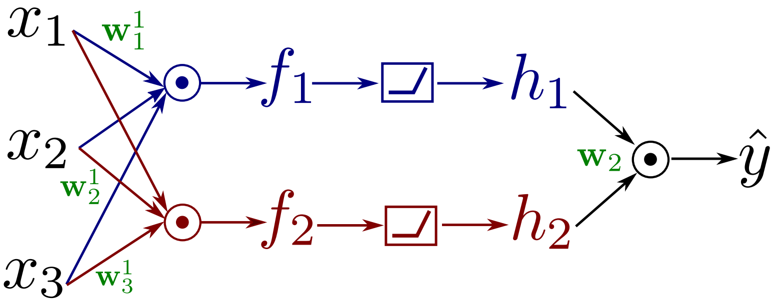
Computational graph: block diagram of mathematical operations.



Neural net: A group of connected perceptrons



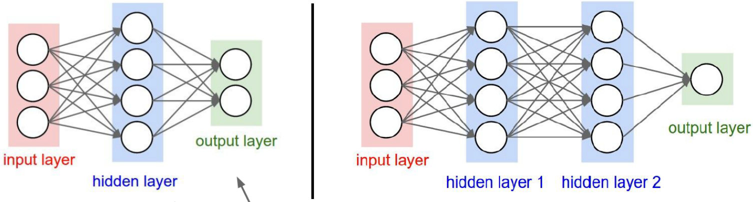
Neural net: A group of connected perceptrons



$\odot$:dot product :activation

$\mathbf{W} = \{\mathbf{w}_1^1, \mathbf{w}_2^1, \mathbf{w}_3^1, \mathbf{w}_2\}$:params

The Multi-Layer Perceptron (MLP)



"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"Fully-connected" layers

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

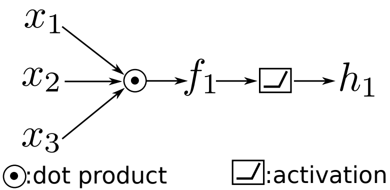
Formal definition of the perceptron

Define the input vector as $\mathbf{x} = \{x_1, x_2, x_3\}$ and the activation function as $v = \sigma(u)$. Then, the perceptron i is

$$f_i = \mathbf{w}_i^T \mathbf{x},$$
$$h_i = \sigma(f_i),$$

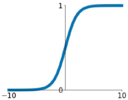
or in short hand

$$h_i = \sigma(\mathbf{w}_i^T \mathbf{x}).$$

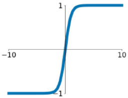


Activation functions

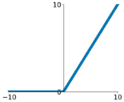
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



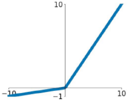
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

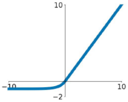


Leaky ReLU
$\max(0.1x, x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



Formal definition of the neural net

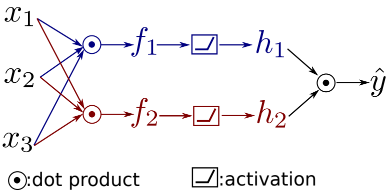
Define $\mathbf{h} = [h_1, h_2]$ and $\mathbf{v} = \sigma(\mathbf{u}) = [\sigma(u_1), \sigma(u_2)]$, then

$$\mathbf{f} = \mathbf{W}_1^T \mathbf{x},$$
$$\mathbf{h} = \sigma(\mathbf{f}),$$
$$\hat{y} = \mathbf{w}_2^T \mathbf{h},$$

or combined

$$\hat{y} = \mathbf{w}_2^T \sigma(\mathbf{W}_1^T \mathbf{x}).$$

Here, $\mathbf{h}$ is the output of the first (in this case the only) *hidden layer*. It is also referred to as the *activation map* of Layer 1.



Deep neural nets

- We can trivially extend the number of hidden layers.
- No agreement on how many layers make a neural net *deep*. Just take it as one with *many* layers, whatever many means.
- Nowadays 100-layer nets are deep, but not very deep.

Formally, a neural net with two hidden layers reads

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\hat{y} &= \mathbf{w}_3^T \mathbf{h}_2.
\end{aligned}$$

How does the computational graph now look like?

Learning with neural nets

Remember Mitchell's definition of learning. Maximize performance on experience

$$\underset{\mathbf{W}}{\operatorname{argmin}} \; \underbrace{\frac{1}{2}(y - \hat{y})^2}_{J(\mathbf{W})} \, .$$

Here,

- **Experience:** y.
- **Model:** $\hat{y}$.
- **Parameters:** $\mathbf{W}$ (collection of all weights in the net).
- **Performance:** J.

Learn as in linear regression: Find the point with minimum gradient

$\nabla_{\mathbf{W}} J \triangleq 0$ cannot be solved (i.e. no closed-form solution). Instead, start from a random point and take steps towards the gradient

$$\mathbf{W}^{(t+1)} \leftarrow \mathbf{W}^{(t)} - \alpha \nabla_{\mathbf{W}} J.$$

- This technique is called *gradient descent*.
- The step size α is called the *learning rate*.
- Each step (t) is called an *iteration*.

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

$\hat{y}$ is the predicted output of the model for a given input. The prediction can be computed by passing an input $\mathbf{x}$ through all the layers up to the output. This is called a *forward pass*.

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

$(y - \hat{y})$ is the prediction *error* of the model with the current parameter values.

Learning for neural nets

$$\nabla_{\mathbf{w}} J = (y - \hat{y}) \nabla_{\mathbf{w}} \hat{y}$$

$\nabla_{\mathbf{w}} \hat{y}$ is the gradient of the model wrt its parameters. Thanks to the chain rule, not as hard as it looks.

Chain rule: Given $y = f(u)$ and $u = g(x)$,

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}.$$

Remember the computational graphs!

Chain rule for vector-variate functions

Given $y = f(\mathbf{u})$ and $\mathbf{u} = g(\mathbf{x})$ where $\mathbf{u}$ and $\mathbf{x}$ are M and N dimensional vectors, respectively,

$$\frac{\partial y}{\partial x_i} = \sum_{j=1}^{M} \frac{\partial y}{\partial u_j} \frac{\partial u_j}{\partial x_i} = \frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial x_i}.$$

Applying this rule to all entries x_i of vector $\mathbf{x}$,

$$\begin{aligned}\frac{\partial y}{\partial \mathbf{x}} &= \left[\frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_N}\right] = \left[\frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial x_1}, \cdots, \frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial x_N}\right] \\ &= \frac{\partial y}{\partial \mathbf{u}}^T \left[\frac{\partial \mathbf{u}}{\partial x_1}, \cdots, \frac{\partial \mathbf{u}}{\partial x_N}\right] = \frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial \mathbf{x}},\end{aligned}$$

where $\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$ is the *Jacobian matrix*, which has the derivative $\frac{\partial u_i}{\partial x_j}$ on its (i, j)th element.

Updating Layer 4

$$\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\mathbf{f}_3 &= \mathbf{W}_3^T \mathbf{h}_2, \\
\mathbf{h}_3 &= \sigma(\mathbf{f}_3), \\
\hat{y} &= \mathbf{w}_4^T \mathbf{h}_3. \Rightarrow \text{Need to reach here}
\end{aligned}$$

The gradient wrt $\mathbf{w}_4$ reads

$$\nabla_{\mathbf{w}_4} \hat{y} = \nabla_{\mathbf{w}_4} \mathbf{w}_4^T \mathbf{h}_3 = \mathbf{h}_3.$$

Note that $\mathbf{h}_3$ needs to be stored during the forward pass!

Updating Layer 3

$$
\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\mathbf{f}_3 &= \mathbf{W}_3^T \mathbf{h}_2, \Rightarrow \text{Need to reach here} \\
\mathbf{h}_3 &= \sigma(\mathbf{f}_3), \\
\hat{y} &= \mathbf{w}_4^T \mathbf{h}_3.
\end{aligned}
$$

The gradient wrt $\mathbf{w}_r^3$, weights connecting Layer 2 neuron r to Layer 3 reads

$$
\nabla_{\mathbf{w}_r^3} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{w}_r^3}.
$$

Updating Layer 2

$$
\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \Rightarrow \text{Need to reach here} \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\mathbf{f}_3 &= \mathbf{W}_3^T \mathbf{h}_2, \\
\mathbf{h}_3 &= \sigma(\mathbf{f}_3), \\
\hat{y} &= \mathbf{w}_4^T \mathbf{h}_3.
\end{aligned}
$$

The gradient wrt $\mathbf{w}_r^2$, weights connecting Layer 1 neuron r to Layer 2 reads

$$
\nabla_{\mathbf{w}_r^2} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{w}_r^2}.
$$

Note how the factors in red can be reused from Layer 3!

Updating Layer 2

$$
\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \Rightarrow \text{Need to reach here} \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\mathbf{f}_3 &= \mathbf{W}_3^T \mathbf{h}_2, \\
\mathbf{h}_3 &= \sigma(\mathbf{f}_3), \\
\hat{y} &= \mathbf{w}_4^T \mathbf{h}_3.
\end{aligned}
$$

The gradient wrt $\mathbf{w}_r^2$, weights connecting Layer 1 neuron r to Layer 2 reads

$$
\nabla_{\mathbf{w}_r^2} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{w}_r^2}.
$$

Updating Layer 1

$$
\begin{aligned}
\mathbf{f}_1 &= \mathbf{W}_1^T \mathbf{x}, \Rightarrow \text{Need to reach here} \\
\mathbf{h}_1 &= \sigma(\mathbf{f}_1), \\
\mathbf{f}_2 &= \mathbf{W}_2^T \mathbf{h}_1, \\
\mathbf{h}_2 &= \sigma(\mathbf{f}_2), \\
\mathbf{f}_3 &= \mathbf{W}_3^T \mathbf{h}_2, \\
\mathbf{h}_3 &= \sigma(\mathbf{f}_3), \\
\hat{y} &= \mathbf{w}_4^T \mathbf{h}_3.
\end{aligned}
$$

The gradient wrt $\mathbf{w}_r^1$, weights connecting input neuron r to Layer 1 reads

$$
\nabla_{\mathbf{w}_r^1} \hat{y} = \frac{\partial \hat{y}}{\partial \mathbf{h}_3}^T \frac{\partial \mathbf{h}_3}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{h}_1} \frac{\partial \mathbf{h}_1}{\partial \mathbf{f}_1} \frac{\partial \mathbf{f}_1}{\partial \mathbf{w}_r^1}.
$$

Note how the factors in red can be reused from Layer 2!

Error Backpropagation

Put everything together, learn the parameters of Layer l following the update rule below:

$$\mathbf{w}_r^{l\,(t+1)} \leftarrow \mathbf{w}_r^{l\,(t)} - \alpha \underbrace{(y - \hat{y}) \nabla_{\mathbf{w}_r^l} \hat{y}}_{\nabla_{\mathbf{w}_r^l} \hat{y}}.$$

Looking closer, we basically update weights by rescaling the prediction error $(y - \hat{y})$ by the gradient of the model $\hat{y}$ wrt them. Hence, prediction error propagates from the top layer to bottom at different levels of importance. This is called **error backpropagation**.

Gradient Backpropagation

- The gradient at Layer $l + 1$ contains a portion of factors required to calculate the gradient at Layer l.
- Then update from top to bottom. Store the reusable factors of each gradient before moving down. This is called the *backward pass*.

Other things than the gradient can backpropagate as well (e.g. random variables or their moments!).

The Backprop Algorithm

Given an input $\mathbf{x}$ and a model $\hat{y}$ with L hidden layers.

- Do a forward pass (i.e. compute $\hat{y}(\mathbf{x})$). Store activation maps on the way $\mathbf{h}_1, \cdots, \mathbf{h}_L$.
- Do a backward pass (i.e. compute gradients $\nabla_{\mathbf{w}_r^l} \hat{y}$ for all r and l). Store the reusable factors of the gradients on the way.
- Perform the parameter update.