Neural Networks - The essentials

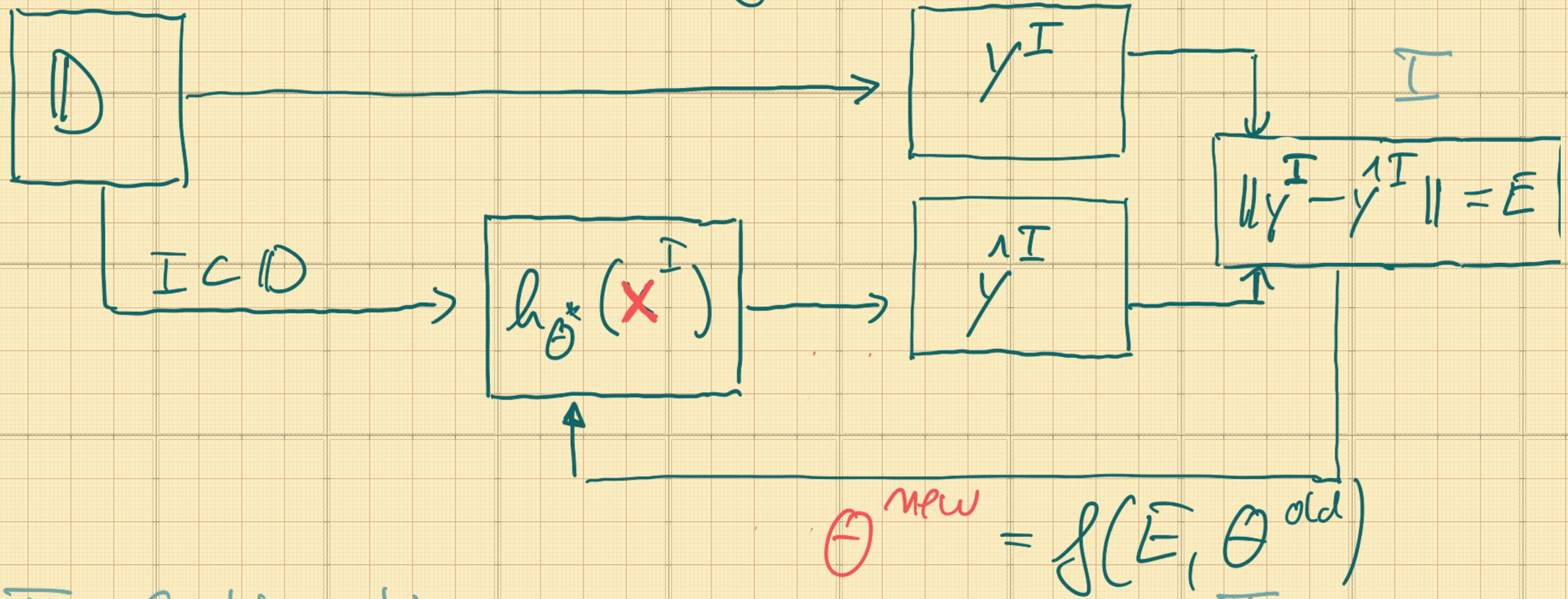
Context: Supervised learning for predictive models

given $\{x_i, y_i\}_{i=1}^{n} = D$, find $P(\hat{y} | x, D)$

or $f(x) = \hat{y} \approx y$, such that $\min_{\theta} \|y - \hat{y}\|$

Recap!

The Supervised Learning cycle: Fail better!



I: Cost function:
How to compute the error

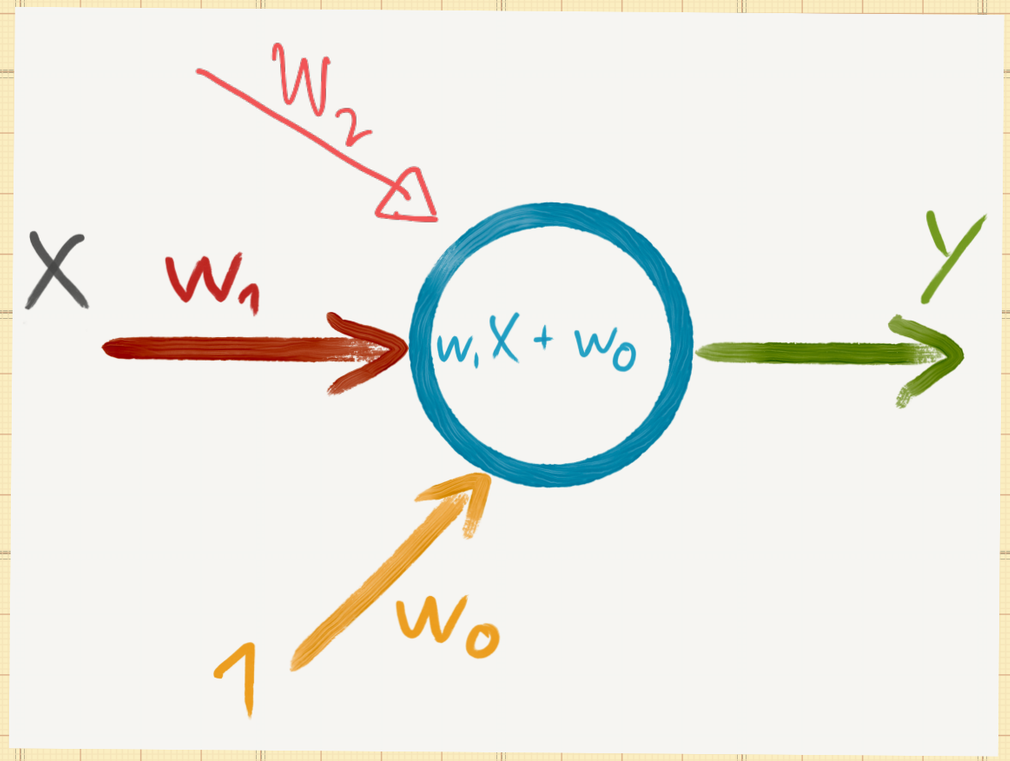
II: How to use knowledge from I to improve θ

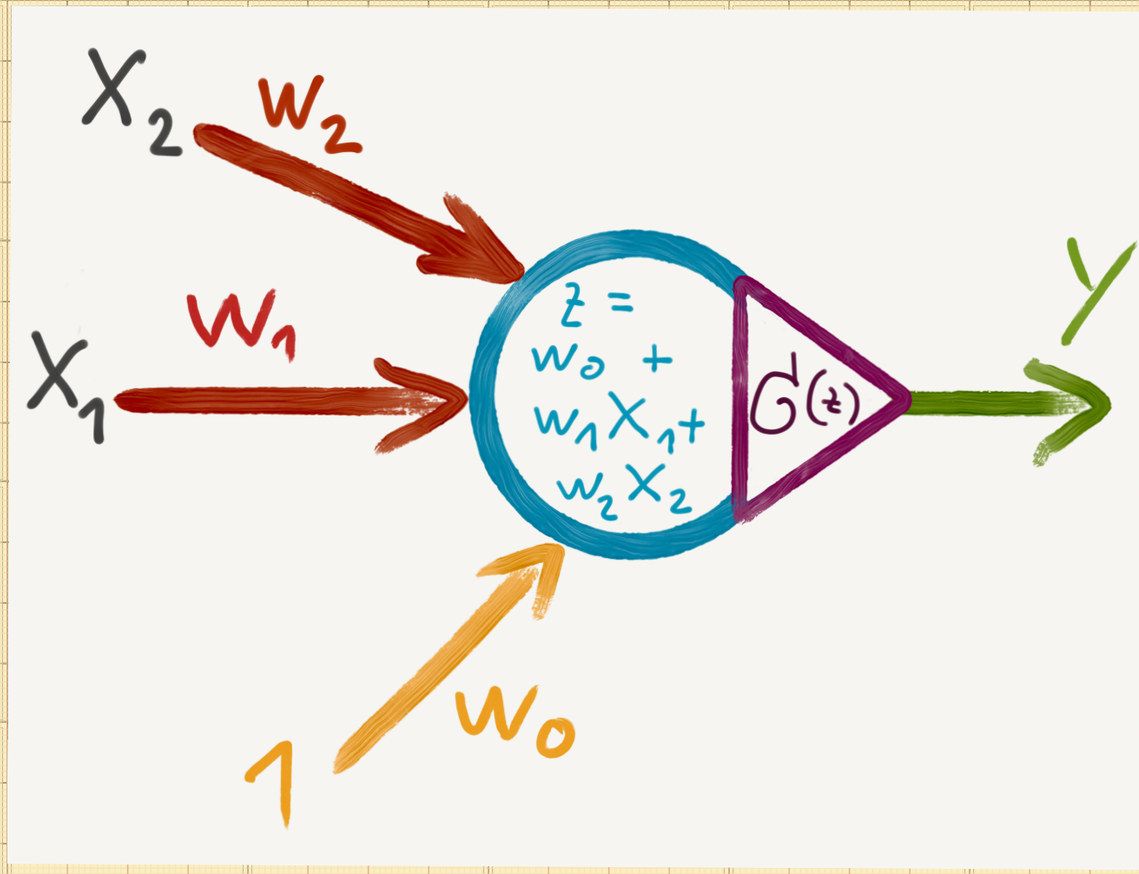
Recap!

So far : $h_\theta = w^T x$ or $h_\theta = G(w^T x)$

Linear model / neuron

Non linear model / logistic neuron





Recap !

Combining these leads to ANNs !

History of ANNs

- Some important publications:
  - McCulloch-Pitts (1943): First compute a weighted sum of the inputs from other neurons plus a bias: the perceptron
  - Rosenblatt (1958): First to generate MLP from perceptrons
  - Rosenblatt (1962): Perceptron Convergence Theorem
  - Minsky and Papert (1969): Limitations of perceptrons
  - Rumelhart and Hinton (1986): Backpropagation by gradient descent
  - LeCun (1995): "LeNet", convolutional networks
  - Hinton (2006): Speed-up of backpropagation
  - Krizhevsky (2012): Convolutional networks for image classification
  - Ioffe (2015): Batch normalization
  - He et al. (2016): Residual networks
  - AlphaGo, DeepMind...

"universal approximator"

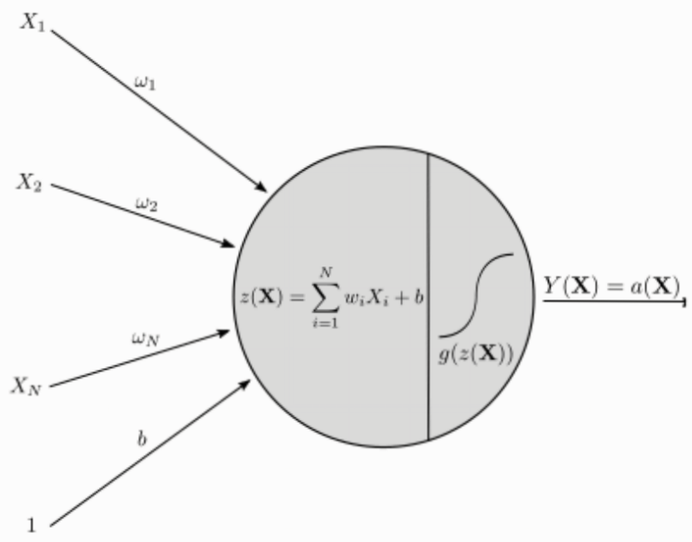
Cybenko 1989

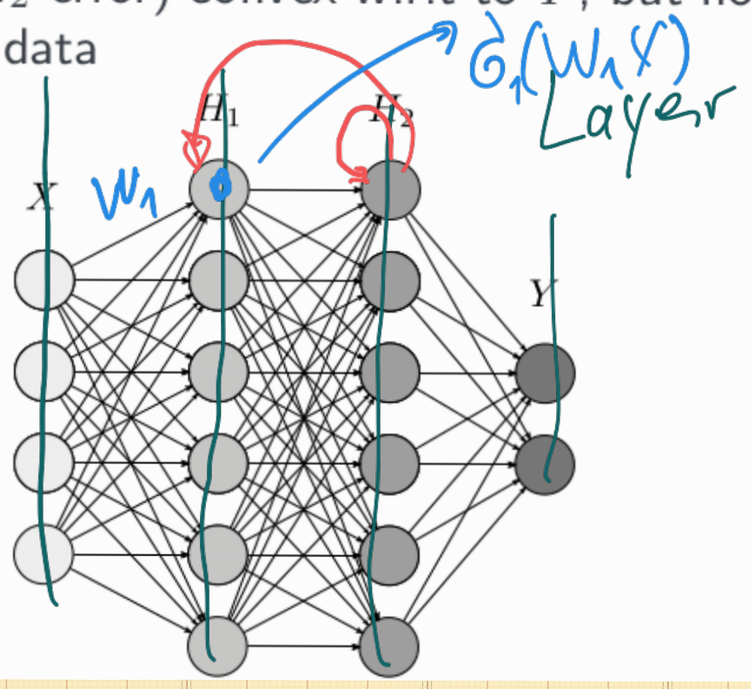
Neural Networks

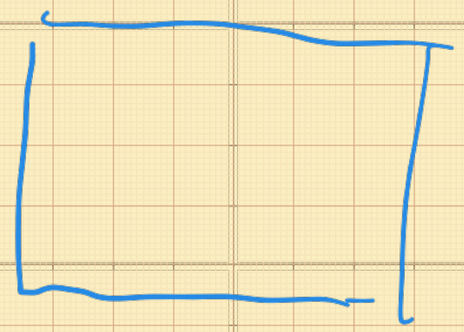
- Artificial Neural Network (ANN): A non-linear mapping from inputs to ouputs: $\mathbf{M} : \hat{X} \rightarrow \hat{Y}$
- An ANN is a nesting of linear and non-linear functions arranged in a directed acyclic graph:

$$\hat{Y} \approx Y = M(\hat{X}) = \sigma_L \left(W_L \left(\sigma_{L-1} \left(W_{L-1} \left(\sigma_{L-2} \left(\ldots \sigma_1 \left(W_1(\hat{X}) \right) \right) \right) \right) \right) \right), \tag{1}$$

  with W being an affine mapping and σ a non-linear function
- The entries of the mapping matrices W are the parameters or weights of the network: improved by training
- Cost function C as a measure for $\left| \hat{Y} - Y \right|$, (MSE / L_2 error) convex w.r.t to Y, but not w.r.t W: $\Rightarrow$ non-convex optimization problem requires a lot of data

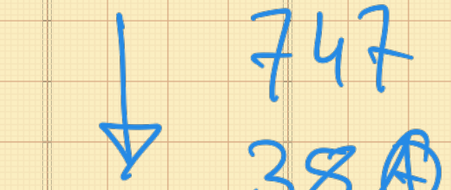






X

$f(x)$



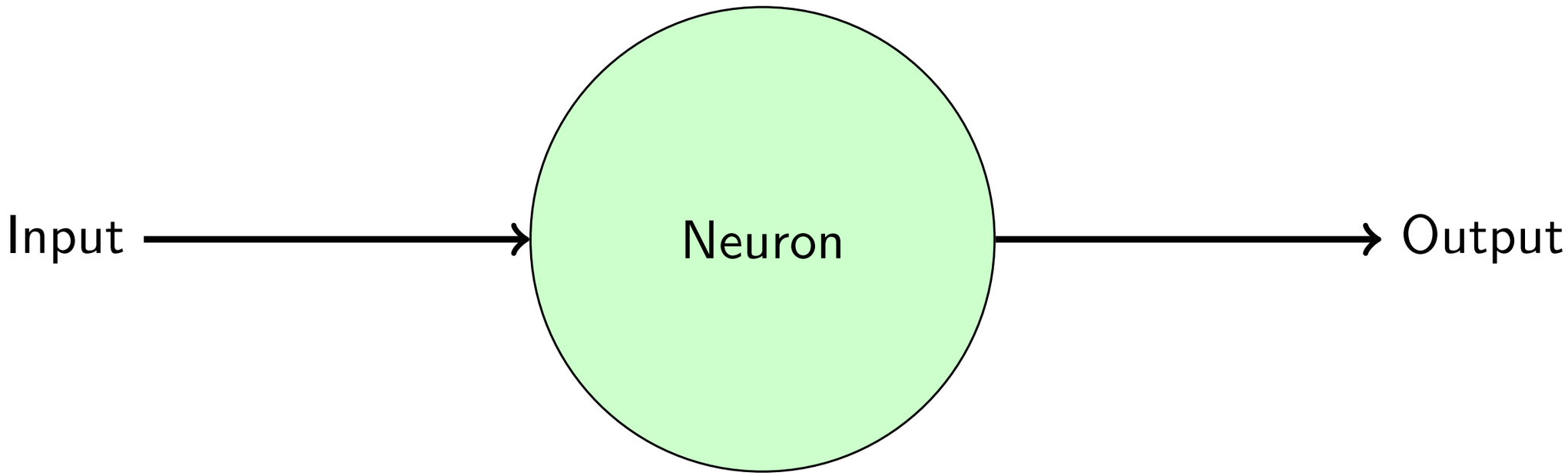
y

→ $f(x)$: general enough
: not too complicated
: trainable

$$\sigma_L(W_L(\sigma_{L-1}(W_{L-1}(\sigma_{L-2}\ldots(W_1 X)))\ldots$$

The smallest building block of an ANN: The Neuron

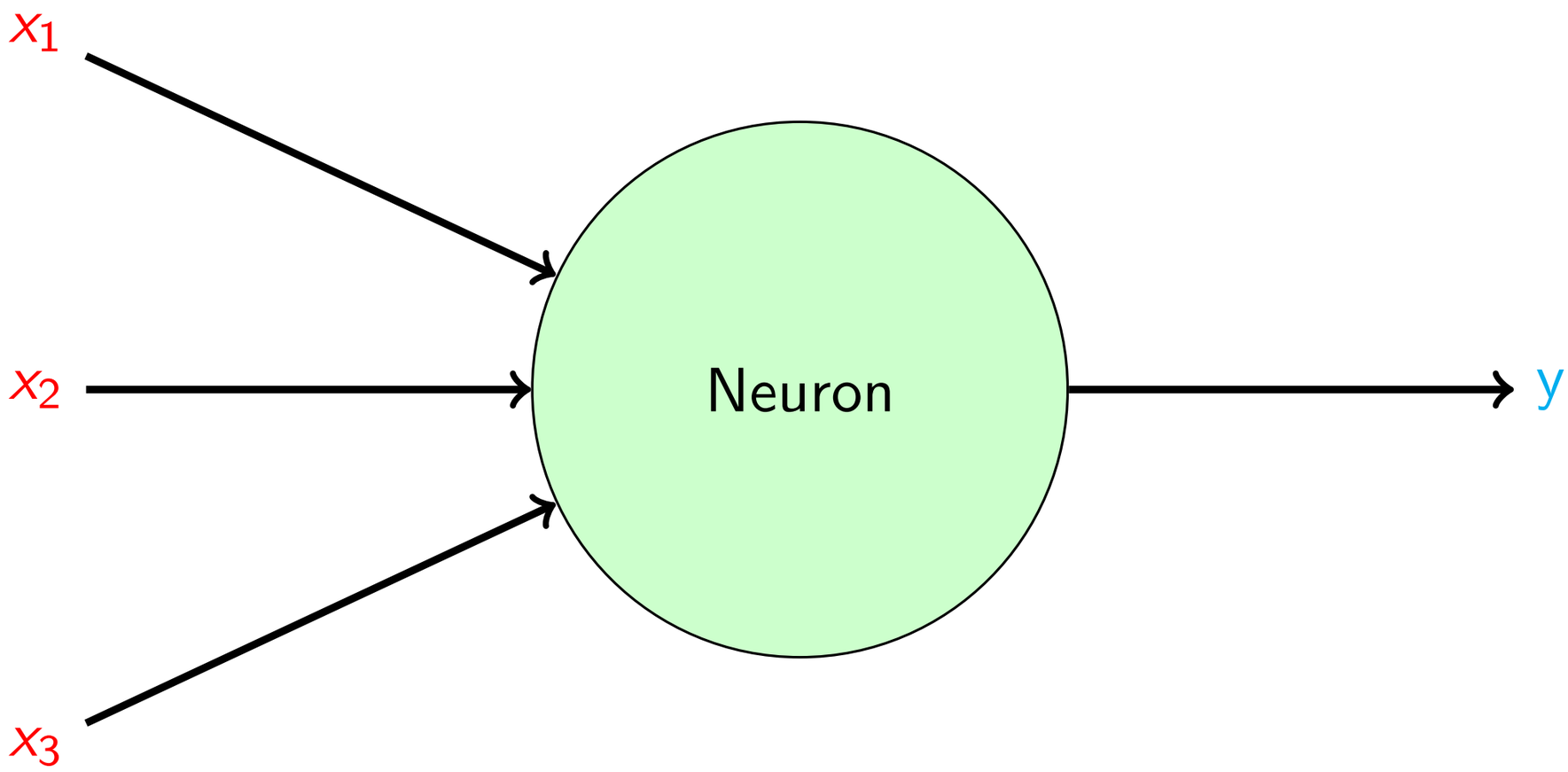
A single neuron is the smallest unit in an ANN. Combining and connecting many of them gives the Artificial Neural Network.



A neuron can be of any type, e.g. a linear or logistic neuron.

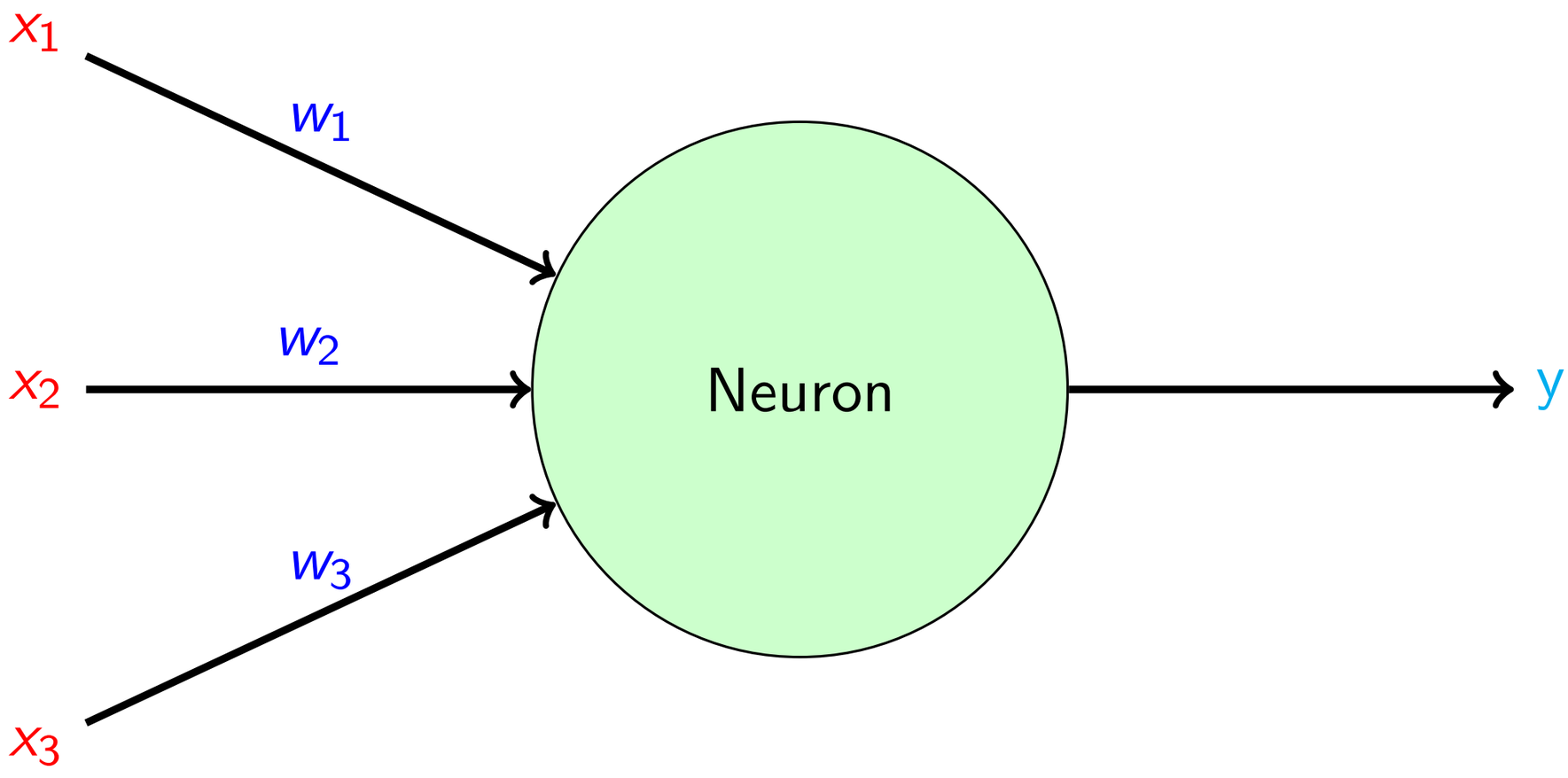
The smallest building block of an ANN: The Neuron

A single neuron can have multiple inputs.
The inputs can be weighted individually.

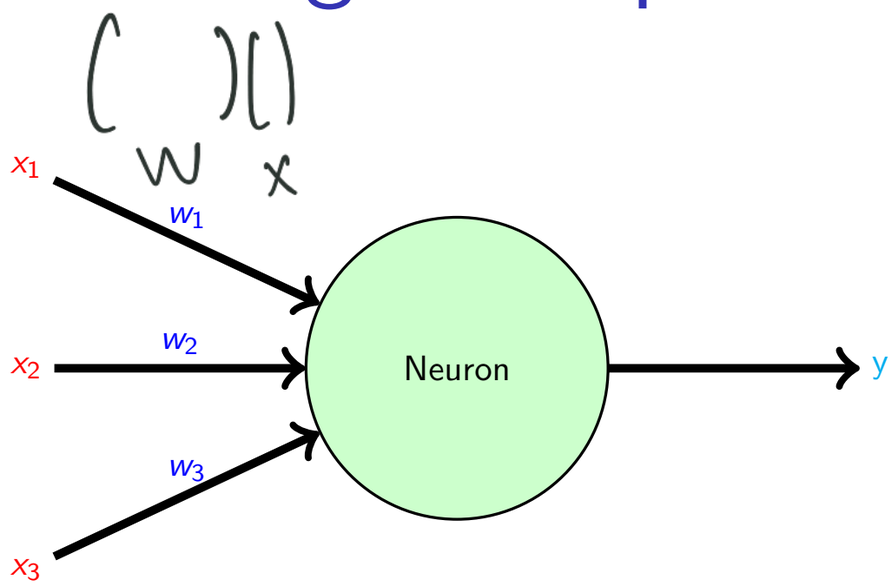


The smallest building block of an ANN: The Neuron

A single neuron can have multiple inputs.
The inputs can be weighted individually.



Combining the Inputs of a Neuron



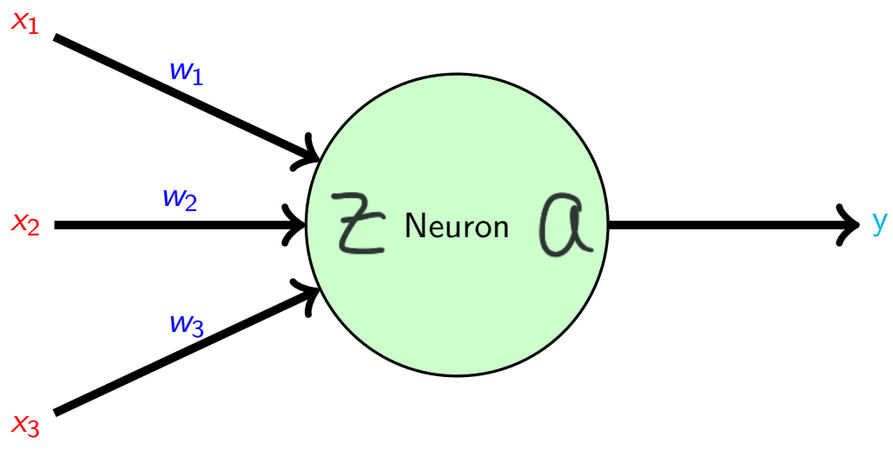
The input z of this neuron is comprised of the individual inputs x_i, weighted by their respective weights w_i.

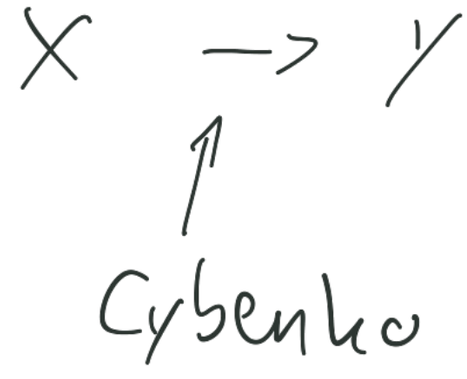
$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3$$

We can of course write this as the scalar / inner product of two vectors:

$$z = \begin{pmatrix} w_1 & w_2 & w_3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3$$

Activating the Neuron





The previous computations of getting z were linear. Adding a non-linear activation (componentwise) allows the neuron to approximate non-linear input/output relationships. This gives as the full output of the neuron y .

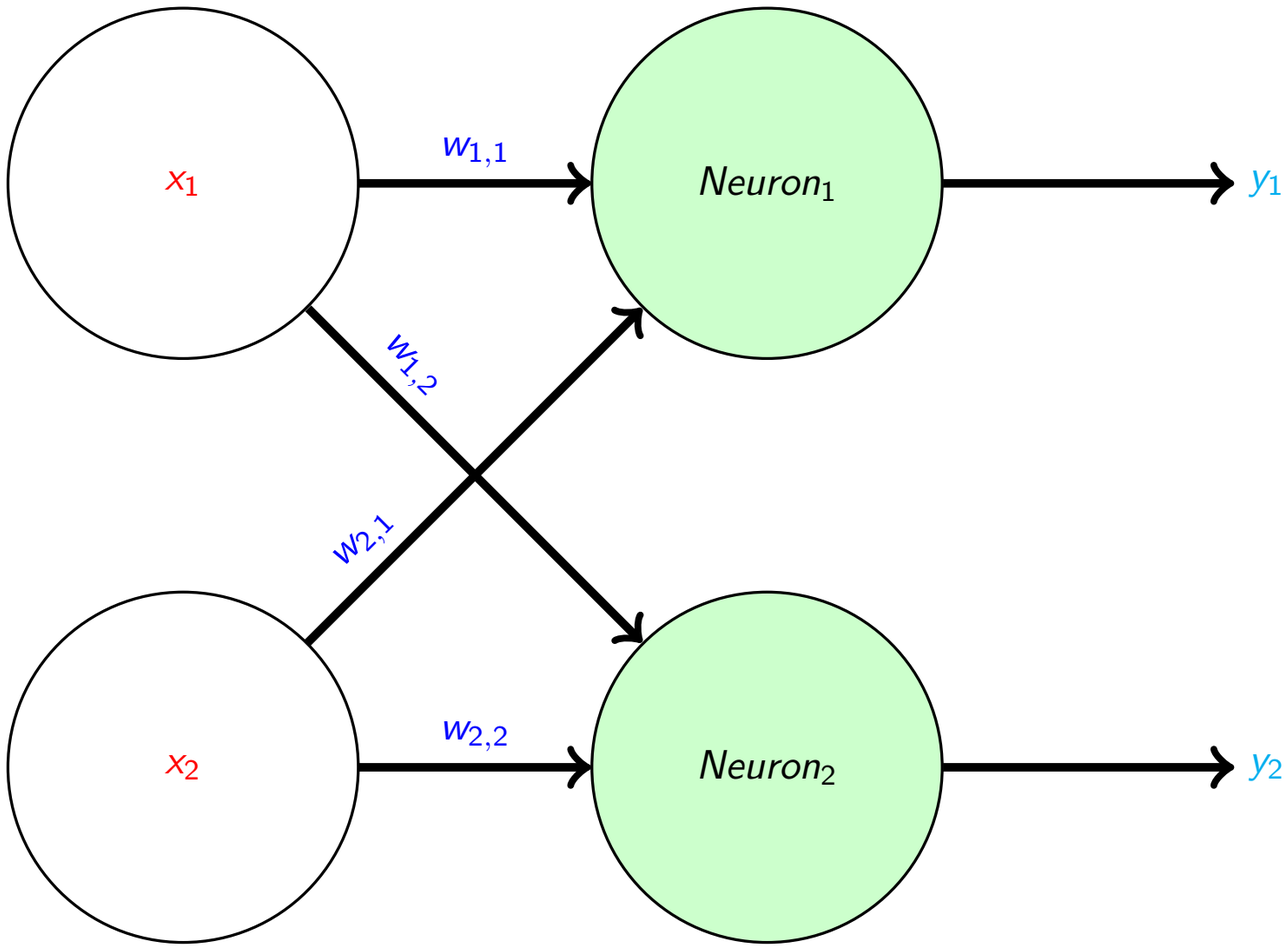
$$y = activationfunction(z)$$

For a sigmoid activation:

$$y = sigmoid(z)$$

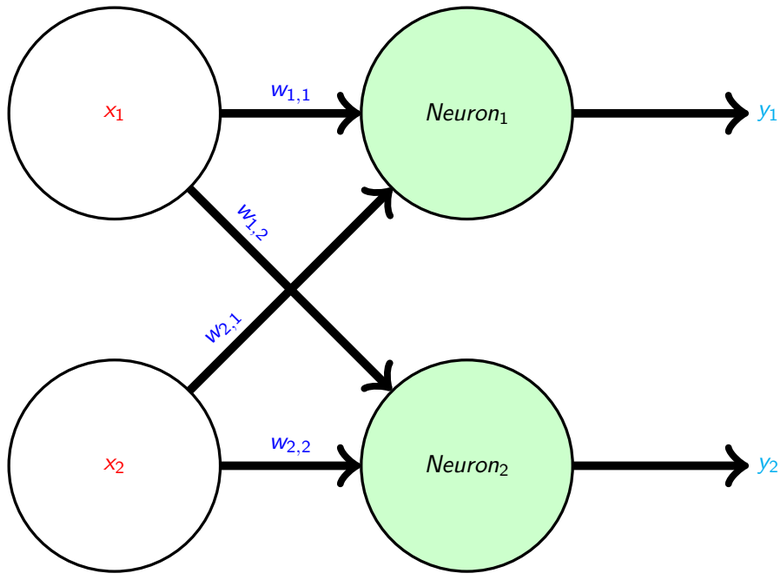
Combining the Neurons to a Network

ANNs get their expressive power from the combination of many neurons.



Nomenclature: weight $w_{1,2}$ runs from x_1 to $Neuron_2$.

Combining the Neurons to a Network



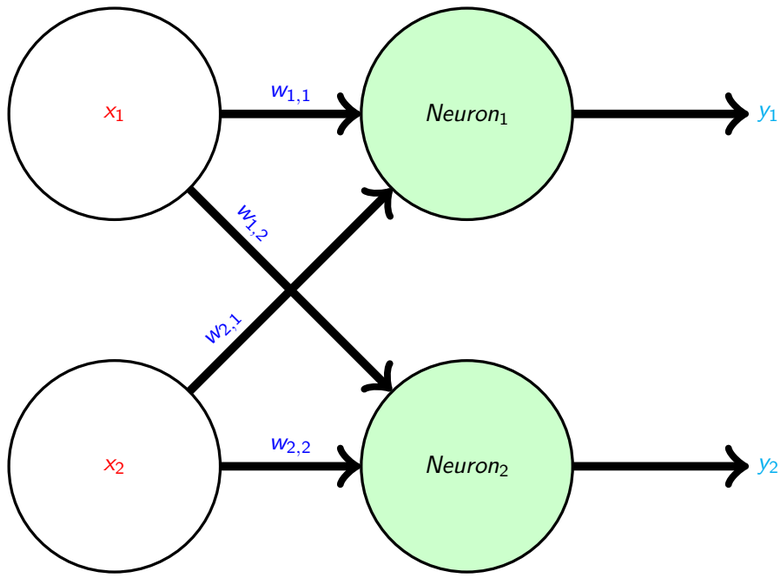
We can again compute the linear combinations for z_i for each neuron:

$$z_1 = w_{1,1} \cdot x_1 + w_{2,1} \cdot x_2$$
$$z_2 = w_{1,2} \cdot x_1 + w_{2,2} \cdot x_2$$

In short as a matrix - vector product for the linear system above:
$Z = \underline{W}X$:

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Combining the Neurons to a Network



The prediction or output of the network is given by the componentwise activation, using sigmoids here:

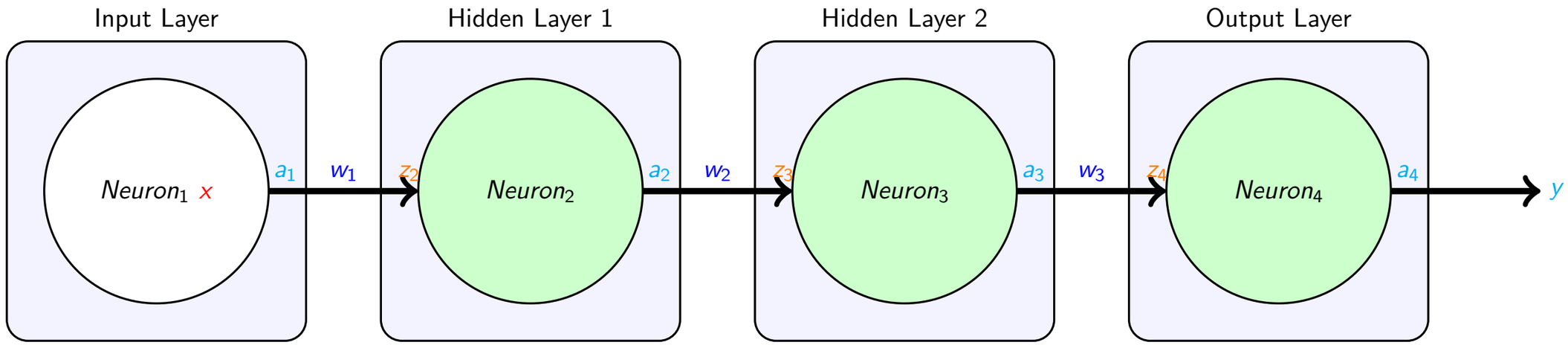
$$y_1 = sigmoid(z_1)$$
$$y_2 = sigmoid(z_2)$$

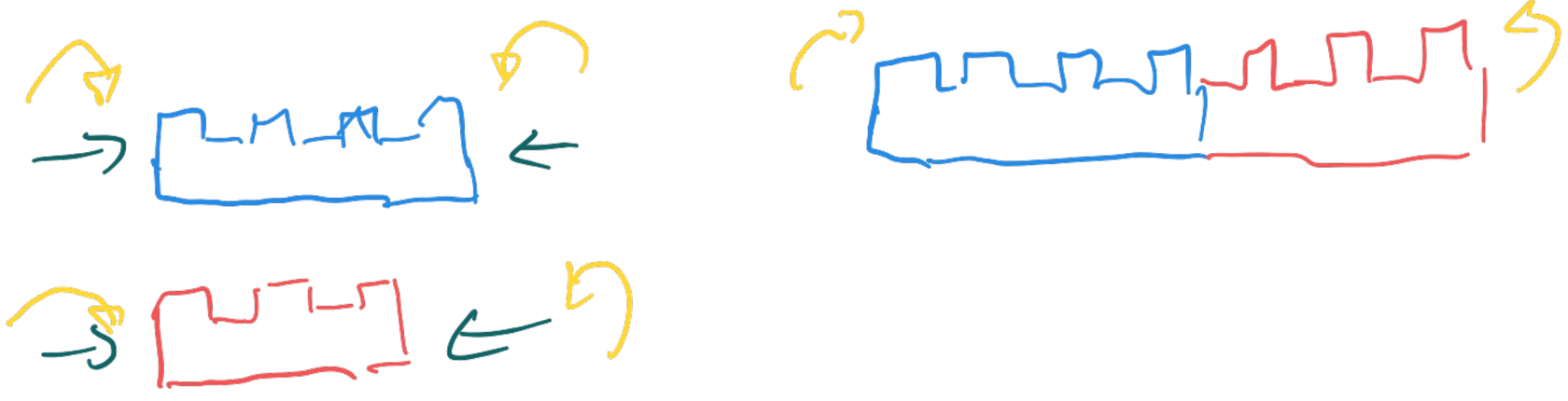
or shorthand:

$$Y = sigmoid(Z)$$

Adding Layers

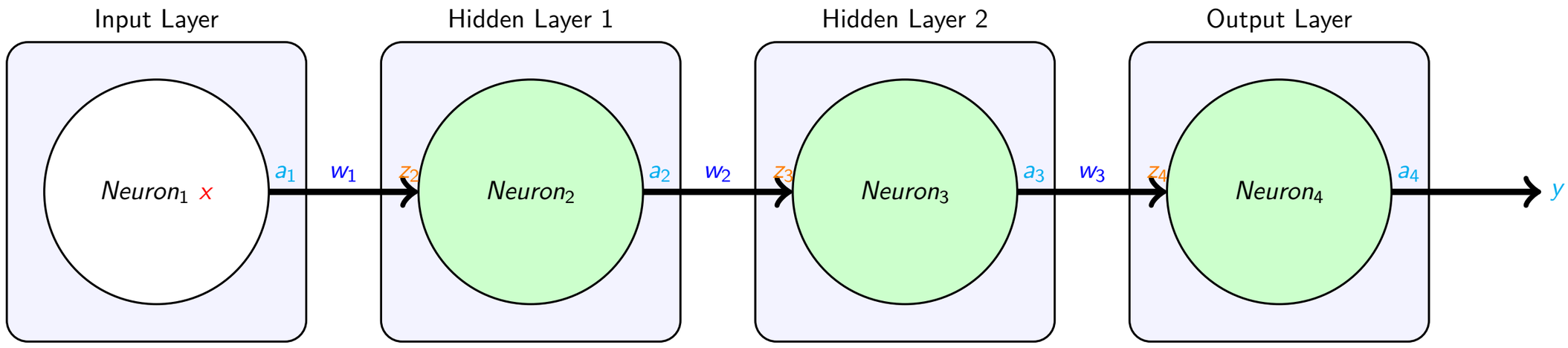
We can stack more layers with an arbitrary number of neurons to get deep ANNs.
Here an example with 2 hidden layers:





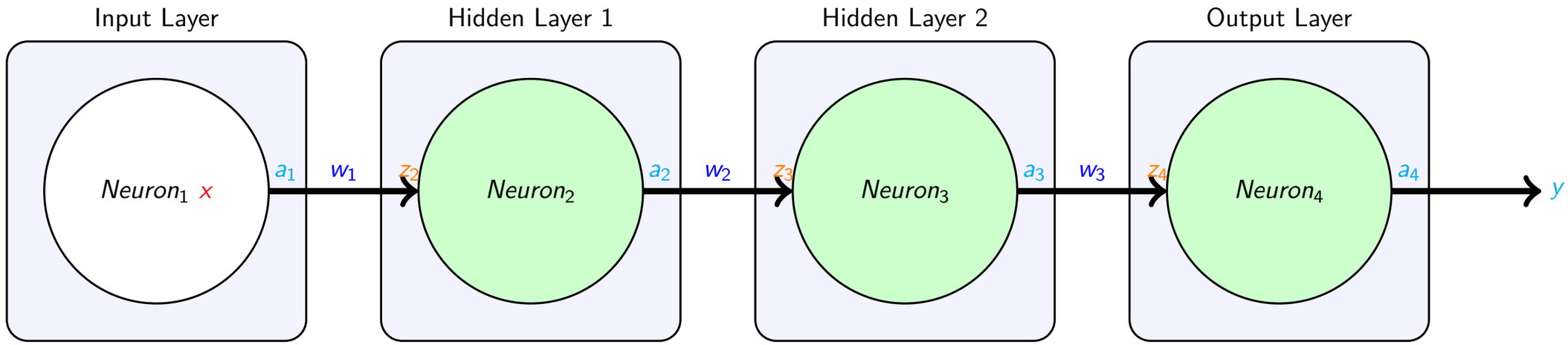
Adding Layers

We can stack more layers with an arbitrary number of neurons to get deep ANNs.
Here an example with 2 hidden layers:



Adding Layers

Here an example with 2 hidden layers:

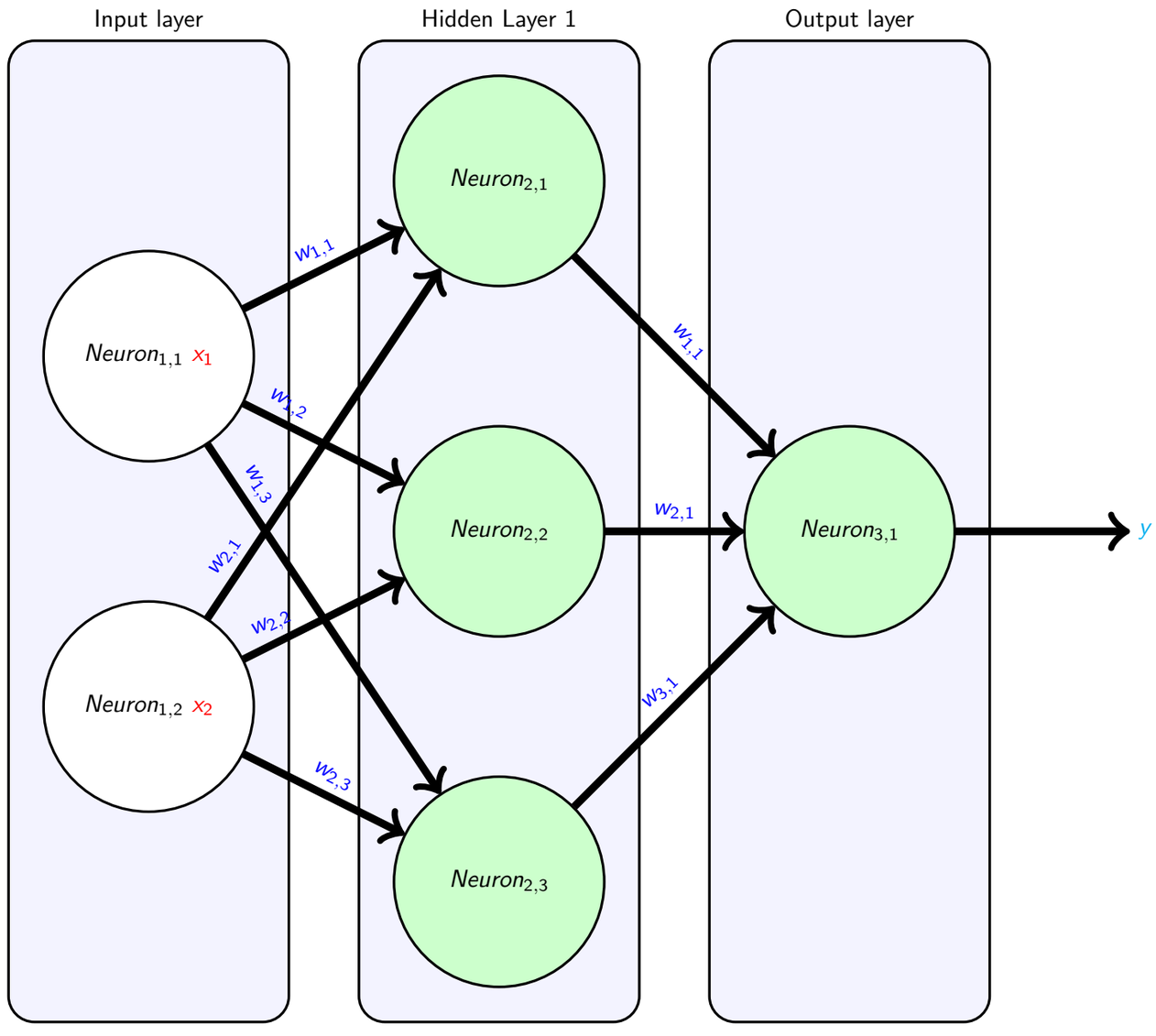


- $Neuron_2$ in hidden layer 1 gets the input $a_1 = x$
- With $z_2 = w_1 \cdot a_1$ we can compute the output of $Neuron_2$
  $a_2 = sigmoid(z_2)$
- This is repeated for hidden layer 2 and the output layer
- Output of this network: $y = a_4$

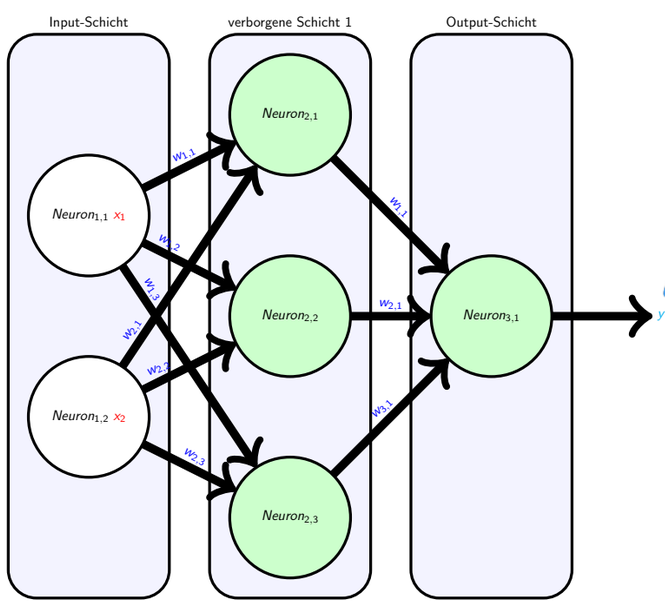
Multi-Layer-Perceptron (MLP)

Combining these options leads to the classical ANN architecture: The Multi-Layer-Perceptron (MLP):

- Multiple layers (fully connected)
- Multiple neurons per layer



Multi-Layer-Perceptron (MLP)



Forward pass:

$M(x) = y$

Using matrix-vector notation to describe the operations in this network:

- $A_1 = X$
- $Z_2 = \underline{W}_{1,2} \cdot A_1$
- $A_2 = sigmoid(Z_2)$
- $Z_3 = \underline{W}_{2,3} \cdot A_2$
- $A_3 = sigmoid(Z_3)$
- $Y = A_3$

Backpropagation

How does learning work?

- Remember the Assigment on linear regression: Iterative improvement of the weights by cost function gradient
- Wir haben an den Beispielen der Linearen und Logistischen Regression schon diskutiert, wie das funktioniert: Durch Ableitung der Kostenfunktion $\mathbb{C}$ nach den Parametern / Gewichten der Hypothese
- So we need the cost function and its gradient w.r.t the parameters
- We consider again the MSE:

$$\mathbb{C} = \frac{1}{n_{samples}} \sum_{n=1}^{n_{samples}} \frac{1}{2} (y - \hat{y})^2 \tag{1}$$

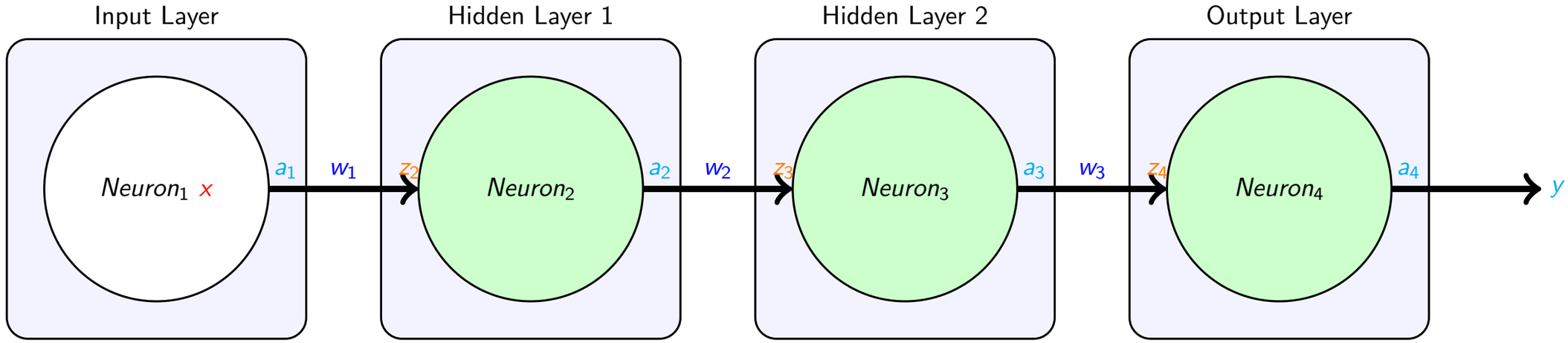
$\frac{\partial C}{\partial w}$

Backpropagation

↓

learning!

- How do we get the these gradients?
- A simplified "linear" network:



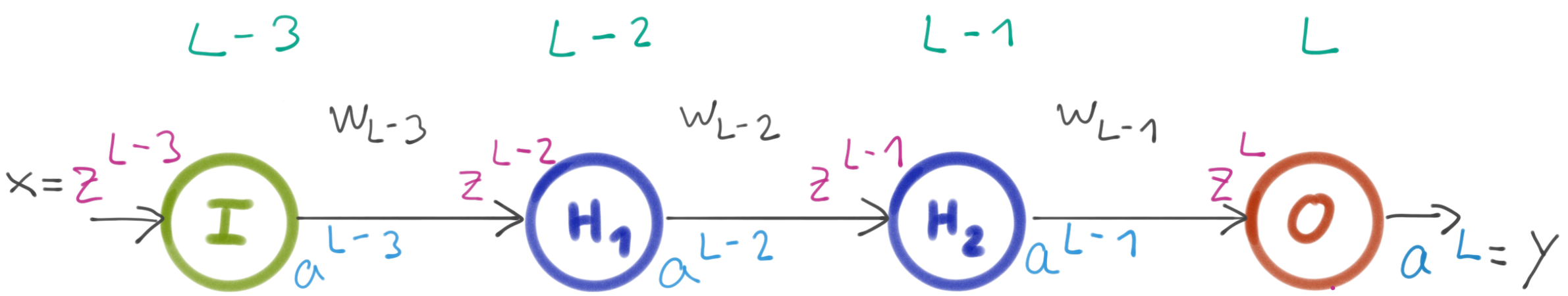
Backpropagation

Einfaches Modell eines MLP:

1 Input - Neuron

2 Hidden Layer mit jeweils einem Neuron

1 Output - Neuron



Nomenklatur

x: Input, y: Output, L: Gesamtzahl Layer

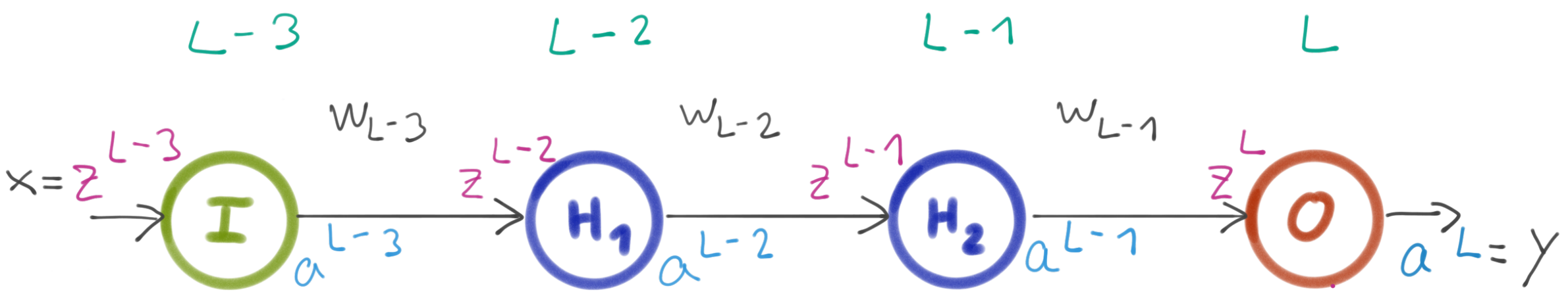
z^L: Input im Neuron des Layers L.

$$z^L = w^{L-1} a^{L-1} + b^{L-1}$$

a^L: Aktivierung / Output des Neurons im Layer L

$a^L = \sigma(z^L)$, mit σ: Aktivierungsfunktion

$\hat{y}$: Wahre Antwort / Label



Nomenklatur II

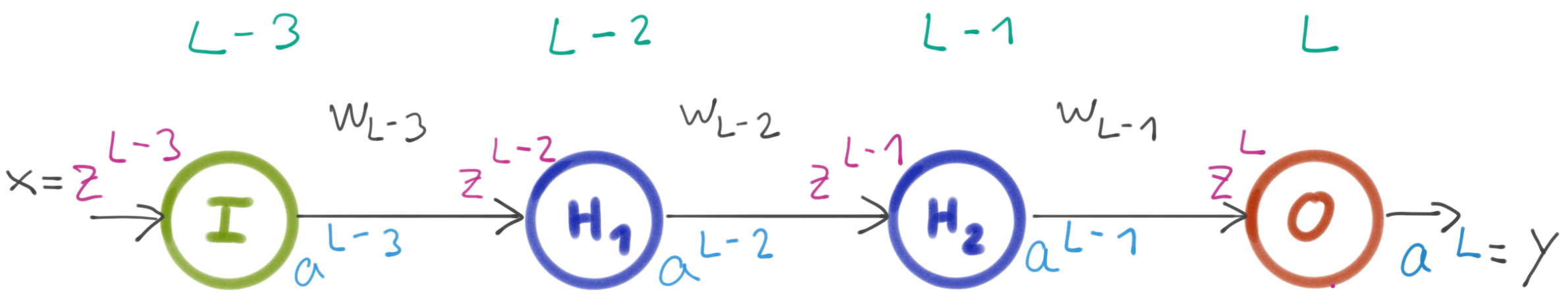
W_L, b_L: Gewichte / Bias zwischen Neuronen in Layern L und $L+1$

Inputlayer: keine Aktivierungsfunktion, $x = z^1 = a^1$

Outputlayer: $a^L = a^4 = y$

Kostenfunktion: $C = \frac{1}{2}(a^L - \hat{y})^2 = \frac{1}{2}(y - \hat{y})^2$

prediction true

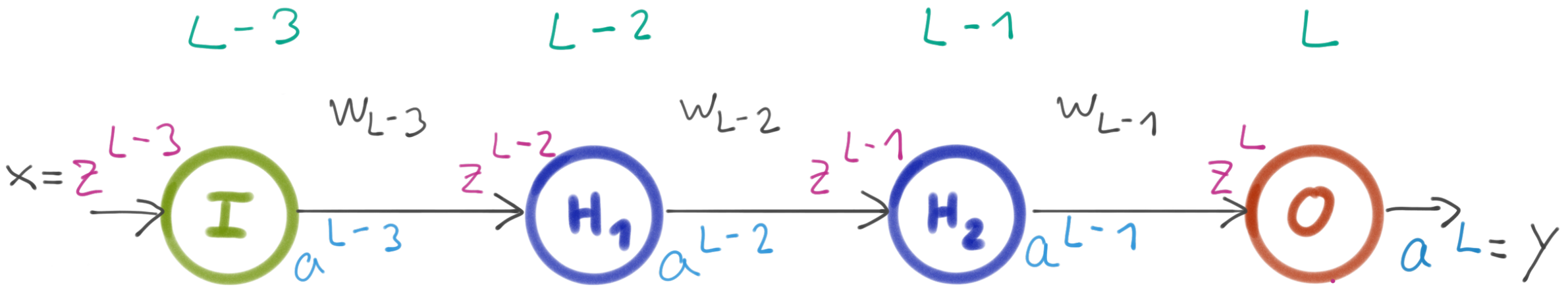


Gradientenberechnung

ges.: Update der Gewichte w_i und Biases b_i:
$\Rightarrow$ Gradienten $\frac{\partial C}{\partial w_i}$ und $\frac{\partial C}{\partial b_i}$, $i = 1, 2, 3$

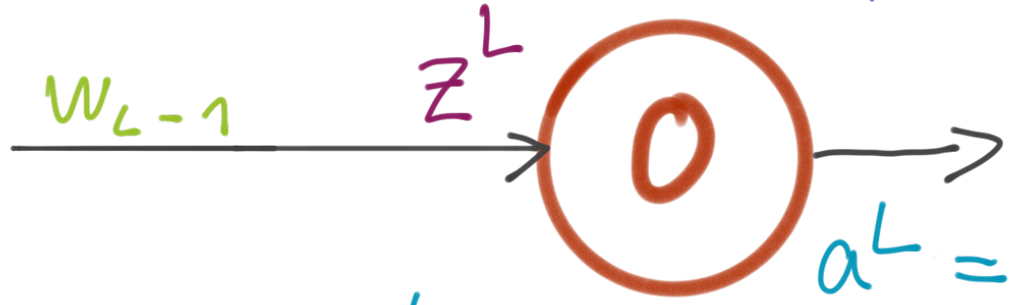
Wir betrachten nur die Gradienten der Gewichte, Biase werden analog behandelt!

$$\Rightarrow \left(\frac{\partial C}{\partial w_1} \quad \frac{\partial C}{\partial w_2} \quad \frac{\partial C}{\partial w_3} \right)^T = ?$$



Gradientenberechnung II

Generelle Strategie: von "rechts nach links", Kosten werden vom Output zum Input "back propagated"

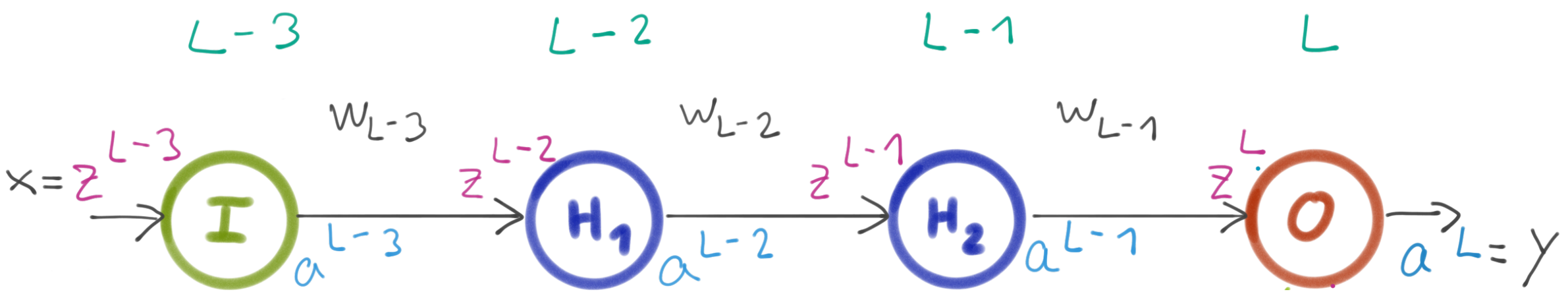


$a^L = y, C = \frac{1}{2}(a^L - \hat{y})^2$

$$\frac{\partial C}{\partial w_{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w_{L-1}}$$

↳ chain rule!

$\Rightarrow$ Kettenregel!



Gradientenberechnung III

$$\frac{\partial C}{\partial w_{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^{L-1}}$$

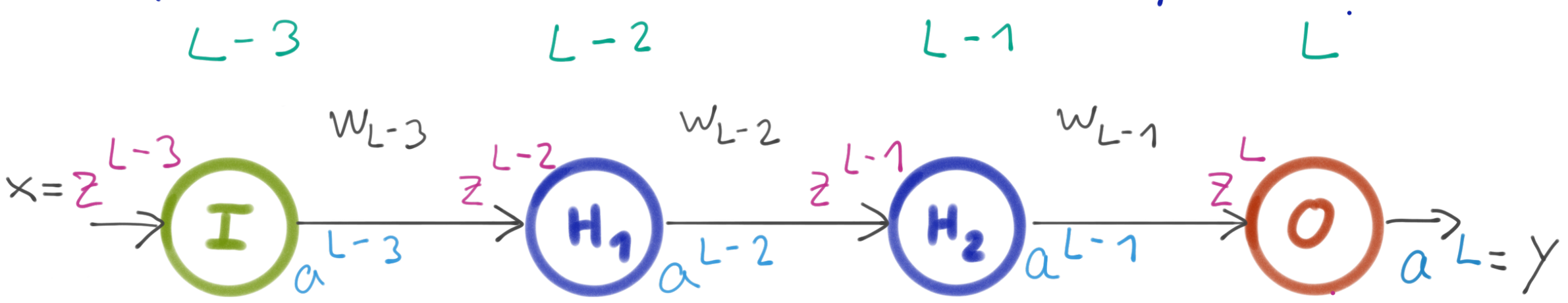
$$\underbrace{(a^L - \hat{y}) \cdot \sigma'(z^L)}_{\delta^{out}} \; \underbrace{a^{L-1}}_{a^{in}}$$

Erinnerung:

$a^L = \sigma(z^L)$

$z^L = w^{L-1} a^{L-1} + b^{L-1}$

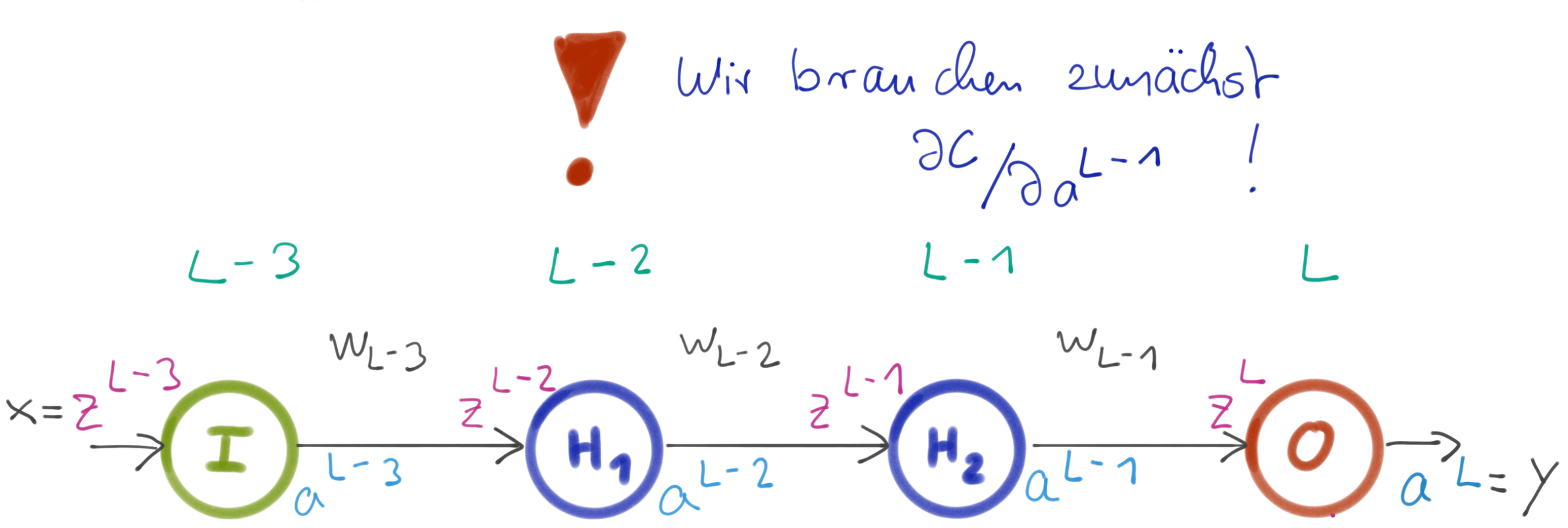
(analog: $\frac{\partial C}{\partial b_{L-1}} = (a^L - \hat{y}) \cdot \sigma'(z^L) \cdot 1$)



Gradientenberechnung IV

Wir haben also $\frac{\partial C}{\partial w_3}$ berechnet! Um die weiter "aufwärts" liegenden Gradienten zu berechnen, wieder holen wir die Kettenregel:

$$\frac{\partial C}{\partial w_{L-2}} = \frac{\partial C}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w_{L-2}}$$



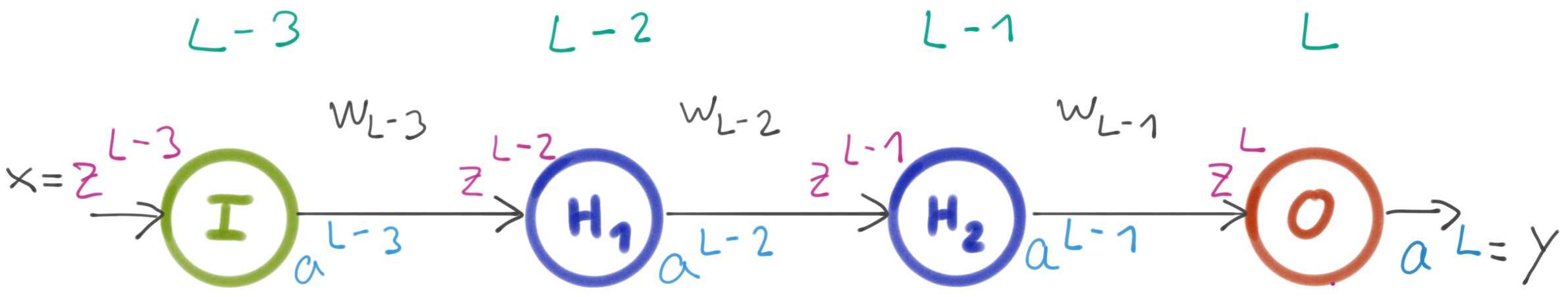
Backpropagated Error

„Stromaufwärts"-Fehler: Kettenregel

$$\frac{\partial C}{\partial a^{L-1}} = \frac{\partial C}{\partial a^{L}} \cdot \frac{\partial a^{L}}{\partial z^{L}} \cdot \frac{\partial z^{L}}{\partial a^{L-1}}$$

$$= (a^{L} - \vec{y}) \cdot \sigma'(z^{L}) \cdot w^{L-1}$$

Damit können wir die Kostenänderung als Funktion des Neuronenoutputs bestimmen.



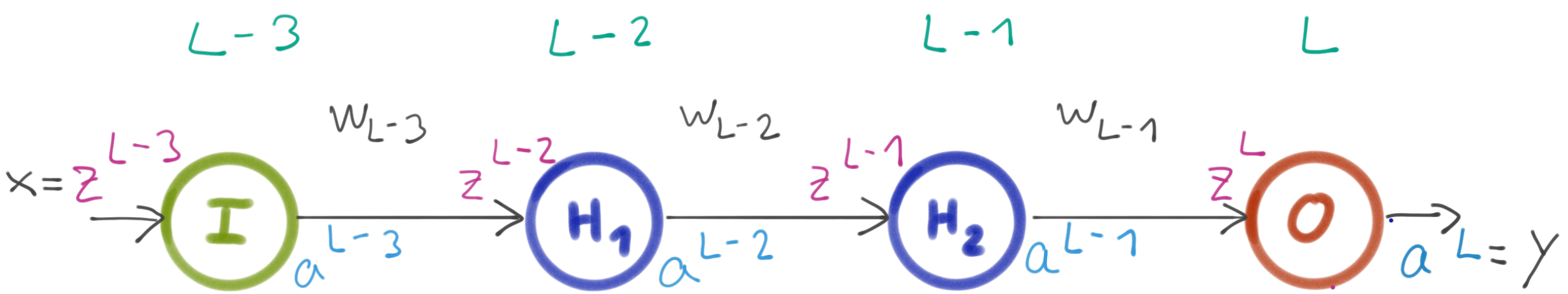
Gradientenberechnung V

$$\frac{\partial C}{\partial w_{L-2}} = \underbrace{(a^L - \vec{y}) \cdot \sigma'(z^L)}_{\partial C / \partial a^{L-1}} \cdot w^{L-1} \cdot \sigma'(z^{L-1}) \cdot \underbrace{a}_{a^{in}}$$

δ^L_{out}

δ^{L-1}_{out}

$L-2$

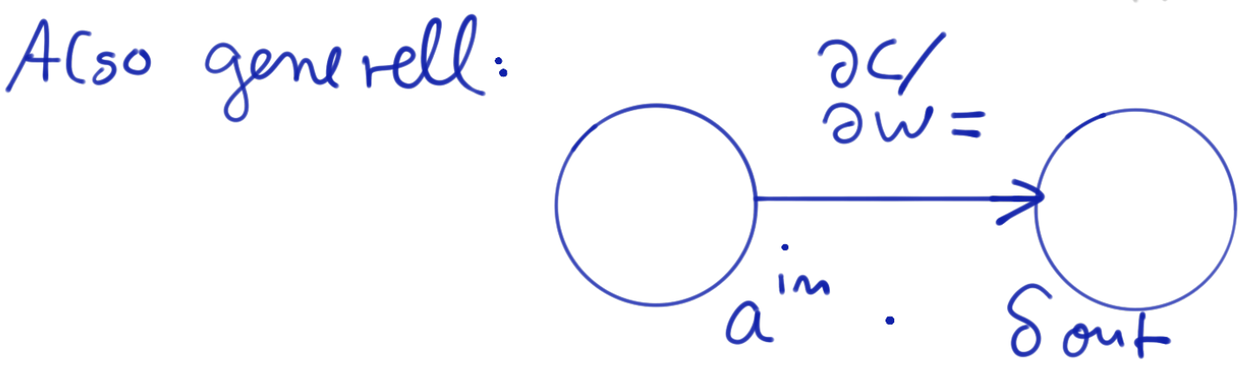


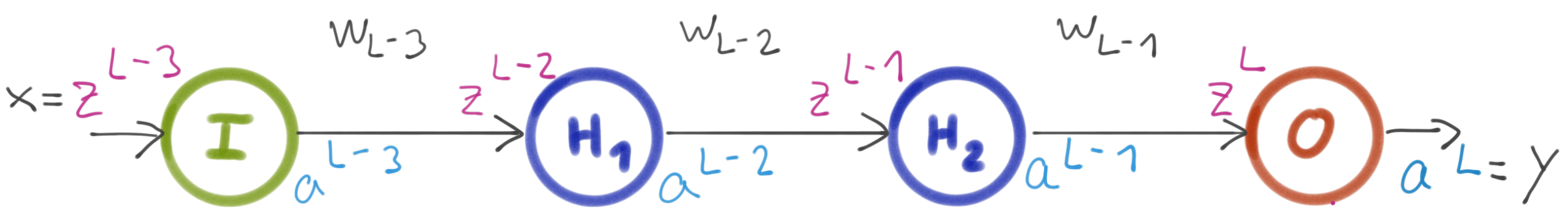
Gradientenberechnung VI

Ganz analog für $\frac{\partial C}{\partial w_n}$:

$$\frac{\partial C}{\partial w_{L-3}} = \underbrace{\delta_{out}^{L-1} \cdot w^{L-2} \cdot \sigma'(z^{L-2})}_{\delta_{out}^{L-2}} \cdot a^{L-3}$$

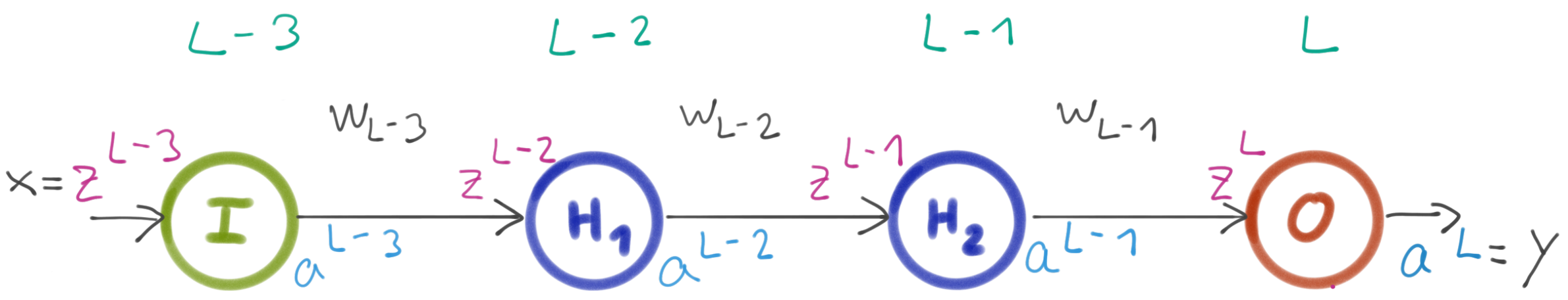
Also generell:





Bemerkungen

- Ein Gradient lässt sich also berechnen aus der "Stromaufs" - Aktivierung a^{in} und einem Term δ^{out}, der nur aus Stromab - Informationen besteht.
- a^{in} ist aus dem Vorwärts/Anfragen/Prediction-step bekannt
- δ^{out} entsteht durch die Anwendung der Kettenregel und propagiert die Kosten stromaufs.



Zusammenfassung

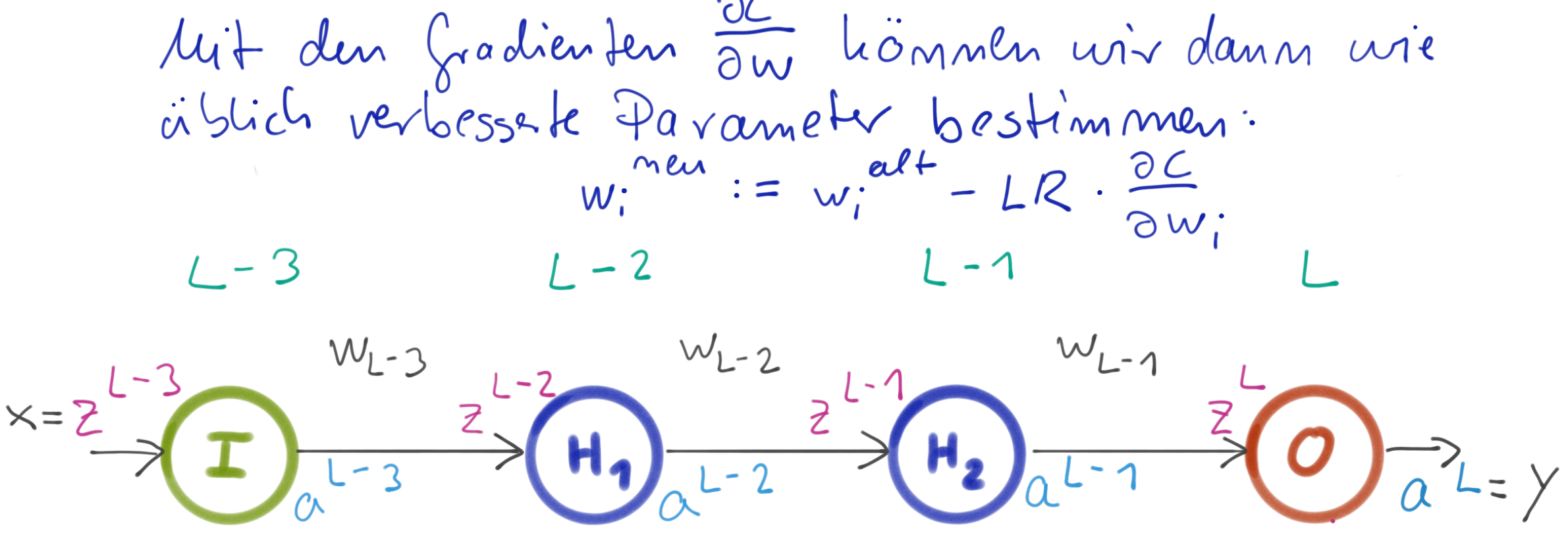
Letztes Layer: $\delta^L = (a^L - \hat{y}) \cdot \sigma'(z^L)$

vorherige Layer: $\delta^l = \delta^{l+1} \cdot w^l \cdot \sigma'(z^l)$

so dass: $\frac{\partial C}{\partial w_{L-i}} = \delta^{L-i+1} \cdot a^{L-i}, \quad i = 1, \dots, L-1$

Mit den Gradienten $\frac{\partial C}{\partial w}$ können wir dann wie üblich verbesserte Parameter bestimmen:

$$w_i^{neu} := w_i^{alt} - LR \cdot \frac{\partial C}{\partial w_i}$$



Backpropagation

Backpropagation for this simple network

- We now know how to update all the weights of the NN, also the ones deep in the net.
- "backpropagation": Chain rule
- For more complex networks with more than one neuron per layer, the algorithm remains the same - we just need to account for vectorial / matrix qunatities
- This is exactly where GPUs shine.

Backpropagation für generelle MLPs

- Wir haben den Backpropagation Algorithmus für ein sehr einfaches Netzwerk kennengelernt
- Für allgemeine Netze verläuft er analog!

1.) Forward pass / prediction für 1 sample:
Berechne alle Aktivierungen a_ℓ und z_ℓ (für jedes Layer ℓ, und für alle Neuronen in diesem Layer)

2.) Berechne δ^L (des Output Layers) als

$$\delta^L = (a^L - \hat{y}) \cdot \sigma'(z^L) \quad \left(\begin{pmatrix} a^L \end{pmatrix} - \begin{pmatrix} \hat{y} \end{pmatrix} \right) \rightarrow \begin{matrix} \sigma'(\,) \\ \sigma'(\,) \\ \sigma'(\,) \\ \vdots \end{matrix}$$

Backpropagation allgemein II

$$\delta^L = (a^L - \vec{y}) \boxdot G'(z^L)$$

elementweise Multiplikation (numpy.multiply)

Vektor der Länge n_outputs

Aktivierungen des letzten Layers, $a^L = y$, Vektor der Länge n_outputs

Wahre Labels, Vektor der Länge n_outputs

Input im letztes Layer

Backpropagation allgemein III

3.) Berechne alle weiteren Fehler δ^l von rechts nach links als

$$\delta^l = (w^l)^T \delta^{l+1} \odot \sigma'(z^l)$$

$w^l : (l+1)_N \times (l)_N$ Matrix

$(w^l)^T : (l)_N \times (l+1)_N$ Matrix

Matrix-Vektor-Produkt, ergibt Vektor d. Länge $(l)_N$

$\odot$ elementweise Multiplikation

Hinweis: δ^1 wird nicht benötigt!

Backpropagation allgemein IV

4.) Berechne alle benötigten Ableitungen als

$$\frac{\partial C}{\partial w_{ij}^{\ell}} = a_j^{\ell}\, \delta_i^{\ell+1}$$

$\delta_i^{\ell+1}$ → i-te Komponente des Vektors δ^{ℓ}

w_{ij}^{ℓ}: Position in der Matrix w^{ℓ}

a_j^{ℓ}: j-ter Eintrag des Vektors a^{ℓ}

oder

$$\left(\nabla_w C\right)^{\ell} = \delta^{\ell+1} \boxdot \left(a^{\ell}\right)^T$$

" Gradient von C für Schicht ℓ "

$\boxdot$: dyadisches Produkt

Backpropagation allgemein V

5.) Update der Parameter

$$w^l := w^l - LR \cdot (\nabla_w C)^l$$

Damit können wir die Parameter des Netzes in beliebig tiefen Schichten verbessern.