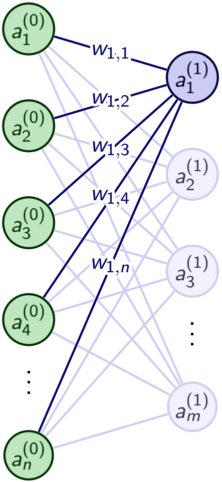
The backpropagation algorithm

Iwo Błądek

Poznan University of Technology

December 7, 2023



$$
\begin{aligned}
a_1^{(1)} &= \sigma\left(w_{1,1}a_1^{(0)} + w_{1,2}a_2^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right) \\
&= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)
\end{aligned}
$$

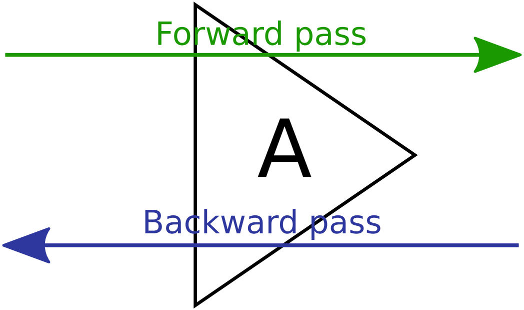
$$
\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix} \begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]
$$

$$
\mathrm{a}^{(1)} = \sigma\left(\mathrm{W}^{(0)}\mathrm{a}^{(0)} + \mathrm{b}^{(0)}\right)
$$

Objective: learn the weights $w_{i,j}$ and biases b_i so that the whole network generates correct predictions.

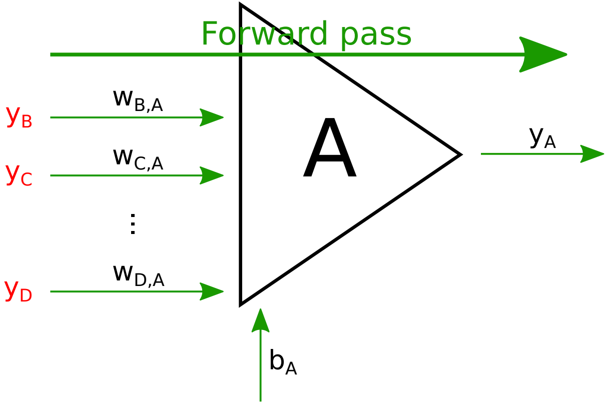
Source: https://tikz.net/neural_networks/

Information flow in neuron



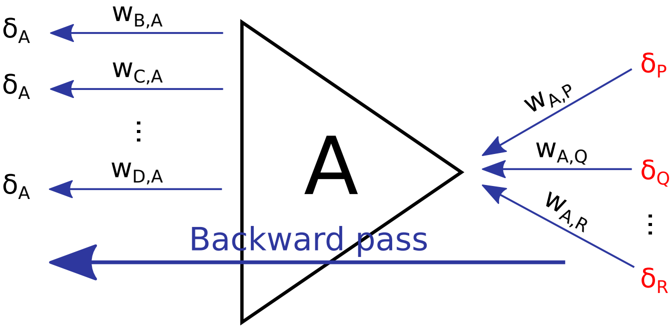
- **Forward pass** – aggregates inputs and after using activation function propagates the result to produce a **prediction**
- **Backward pass** – aggregates errors and (back)propagates them to the previous neurons so that network can **learn**
- Each neuron (or layer) can be treated as a separate module which receives activations/errors and simply propagates them further/backward after processing.

Forward pass



$z_A = b_A + w_{B,A} \cdot y_B + w_{C,A} \cdot y_C + \ldots + w_{D,A} \cdot y_D$ (aggregation)

$y_A = f(z_A)$ (activation function)

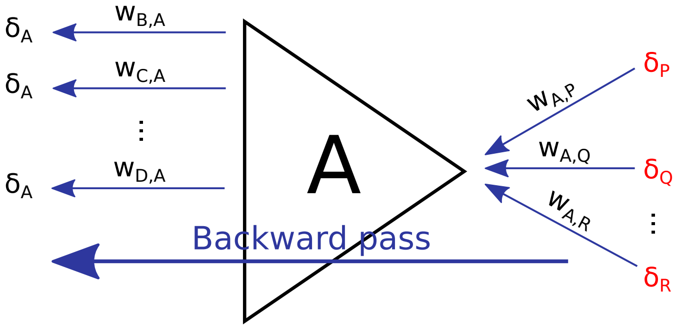


Step #1: Aggregation of errors (δ's) and computation of δ_A, which must also take into account derivative of the activation function f.

$$\delta_{PQR} = w_{A,P} \cdot \delta_P + w_{A,Q} \cdot \delta_Q + \ldots + w_{A,R} \cdot \delta_R \qquad \text{(aggregation)}$$

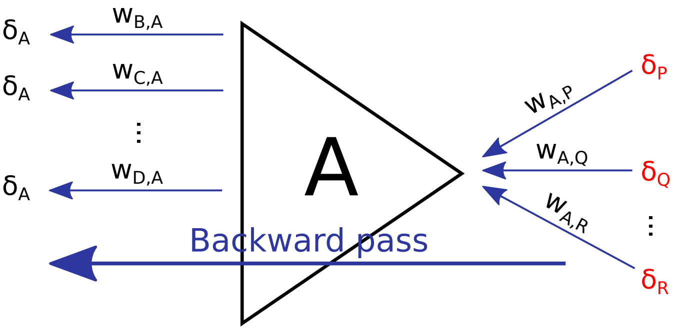
$$\delta_A = \delta_{PQR} \cdot \frac{\partial y_A}{\partial z_A} = \delta_{PQR} \cdot f'(z_A)$$

Backward pass



Step #2: Backpropagating δ_A to the connected neurons in the previous layer using the old values of weights $w_{x,A}$

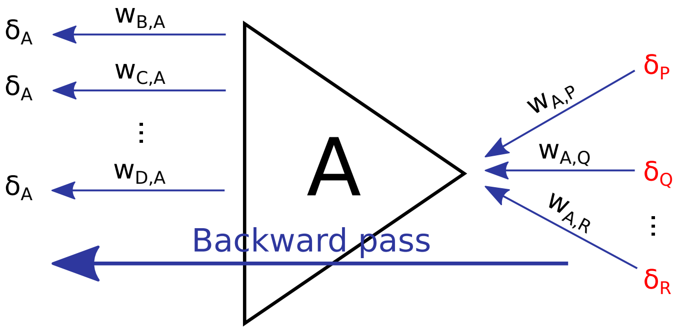
Backward pass



Step #3: Updating all weights and bias of A using δ_A, with μ as learning rate

$$w_{B,A} \coloneqq w_{B,A} - \mu \Delta_{w_{B,A}}$$

$$\Delta_{w_{B,A}} = \delta_A \cdot \frac{\partial z_A}{\partial w_{B,A}} = \delta_A \cdot y_B$$

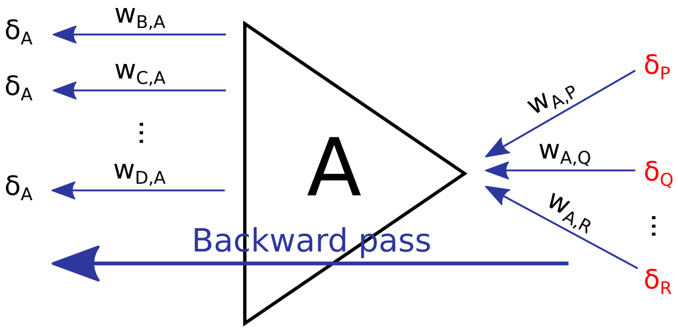


Step #3: Updating all weights and bias of A using δ_A, with μ as learning rate

$$w_{C,A} := w_{C,A} - \mu \Delta_{w_{C,A}}$$

$$\Delta_{w_{C,A}} = \delta_A \cdot \frac{\partial z_A}{\partial w_{C,A}} = \delta_A \cdot y_C$$

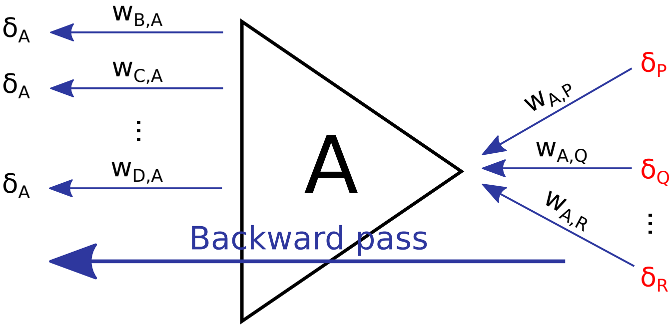
Backward pass



Step #3: Updating all weights and bias of A using δ_A, with μ as learning rate

$$w_{D,A} := w_{D,A} - \mu \Delta_{w_{D,A}}$$

$$\Delta_{w_{D,A}} = \delta_A \cdot \frac{\partial z_A}{\partial w_{D,A}} = \delta_A \cdot y_D$$



Step #3: Updating all weights and bias of A using δ_A, with μ as learning rate

$$b_A := b_A - \mu \Delta_{b_A}$$

$$\Delta_{b_A} = \delta_A \cdot \frac{\partial z_A}{\partial b_A} = \delta_A \cdot 1$$

Backward pass

Additional remarks:

- The backpropagation algorithm can be derived using multivariable chain rule.
- The error δ_{pred} of a prediction y returned by the last neuron is computed as follows:

$$L(y, \hat{y}) = MSE(y, \hat{y}) = (\hat{y} - y)^2$$
$$\delta_{\text{pred}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z} = -2(\hat{y} - y)f'(z),$$

  where $\hat{y}$ is the correct value for a particular training example, and $f'(z)$ is a derivative of activation function for the value of z computed during forward pass of the example.

Regular vs. stochastic gradient descent

- **(Regular) gradient descent** – computes gradient for all training examples at once and takes the average derivative of error. The update rule is defined as:

$$w_{B,A} := w_{B,A} - \mu \frac{1}{m} \sum_{i=1}^{m} \Delta_{w_{B,A}}(x_i),$$

  where m is the number of training examples in the dataset, and x_i is i'th training example.

- **Stochastic gradient descent (SGD)** – instead of m examples, we compute a gradient for a subset of $k << m$ randomly selected examples (called *batch*, or *minibatch*).

$$w_{B,A} := w_{B,A} - \mu \frac{1}{k} \sum_{i=1}^{k} \Delta_{w_{B,A}}(x_i),$$

Regular vs. stochastic gradient descent

Regular GD

$-$ Slow to compute

$-$ Can fall into bad local minima

$+$ Leads to better results if avoids bad local minima

? Final destination is entirely determined by the starting point

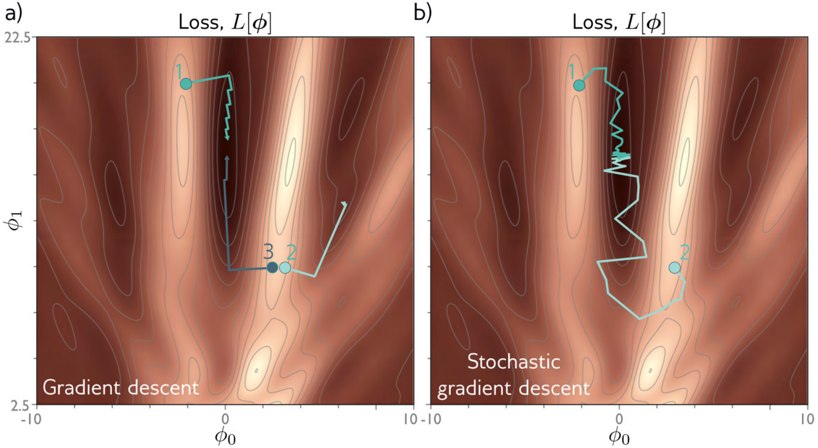
Stochastic GD

$+$ Fast to compute

$+$ Can escape from bad local minima

$-$ Potential problems with convergence to a good solution

Regular vs. stochastic gradient descent



Source: S.Prince, "Understanding Deep Learning"

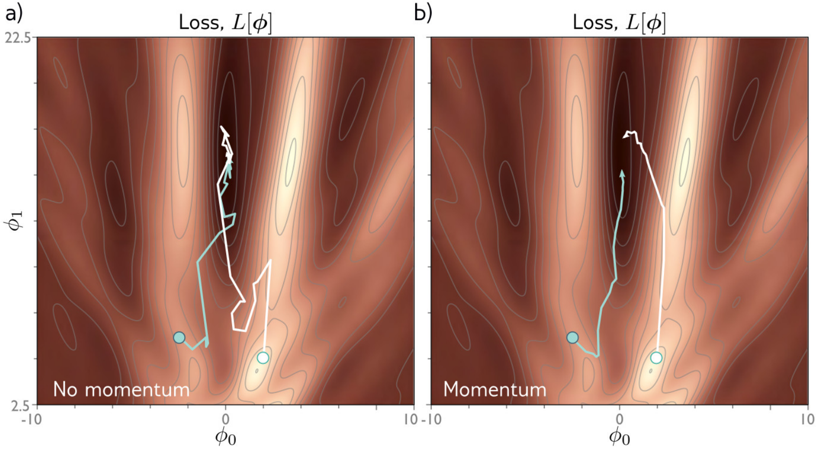
Momentum

To make SGD less chaotic, a modification called **momentum** is sometimes used. In agreement with its name, momentum makes changing direction of search harder by taking into account also part of the previously computed gradient. The update rule changes to:

$$
\begin{aligned}
m_{B,A} &:= \beta m_{B,A} - \mu \frac{1}{k} \sum_{i=1}^{k} \Delta_{w_{B,A}}(x_i) \\
w_{B,A} &:= w_{B,A} + m_{B,A},
\end{aligned}
$$

where $\beta \in [0, 1)$ is a parameter which specifies, how strong the momentum is (how quickly contributions of past gradient values exponentially decay).

Momentum



Source: S.Prince, "Understanding Deep Learning"