Training neural networks

Training: supervised learning

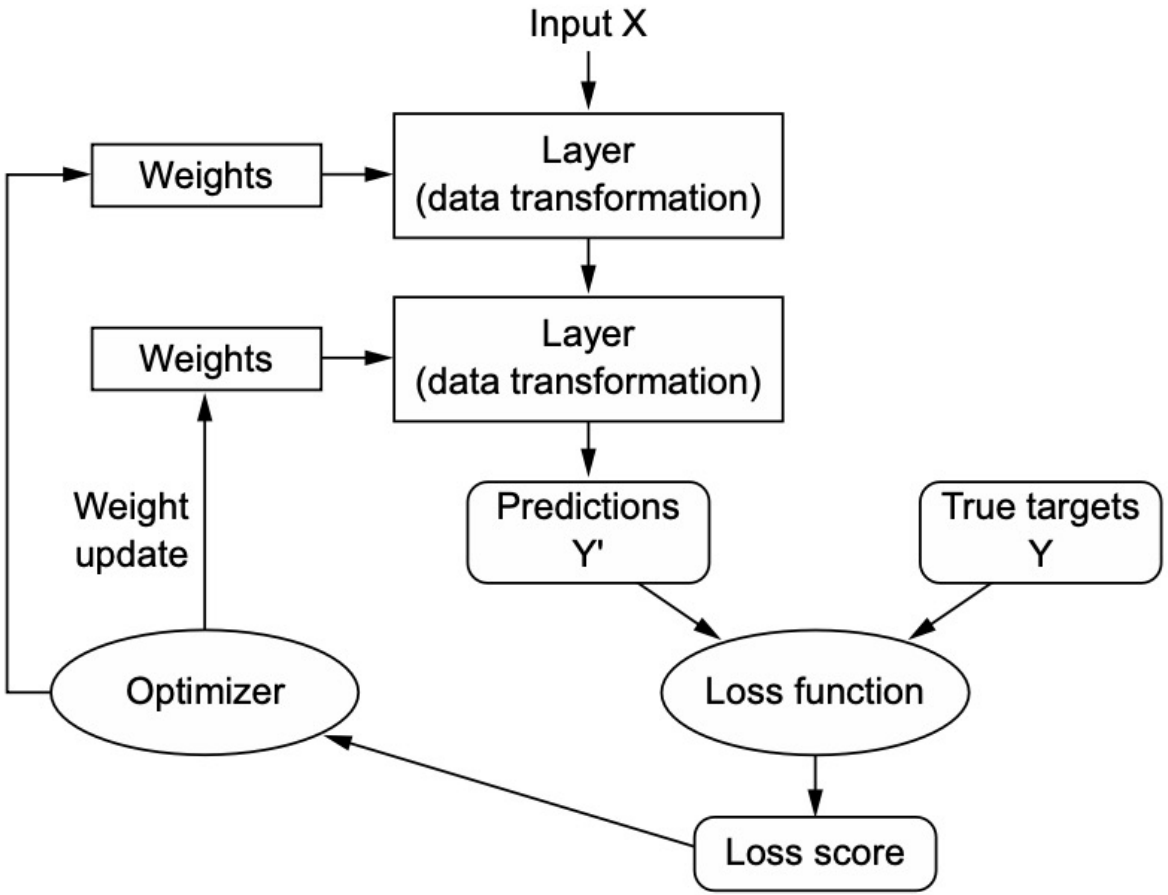
Data: Training examples consisting of **inputs** and their desired **outputs**

Objective: To set the values of the **connection weights** that minimize a cost function

Multiple **gradient-descent** algorithms exist:

- The most used historically is ***Backpropagation*** and derivatives
- Other algorithms: Marquardt-Levenberg, Rprop, Quickprop
- Most recent: RMSprop, Adam, SGD, Adagrad

Training: supervised learning



Overall scheme of the training/ learning process

Backpropagation steps

Forward propagation – calculates the output value(s) for the respective input vector and the error (given the known outputs);

Backpropagation – given the error on that sample, it is propagated back, adjusting the weights of the connections to reduce this error.

This process is based on the calculation of the **gradient** of the cost function, using the **chain rule** of partial derivatives for composite functions

This process is repeated multiple times in order to improve the network's performance.

Back-propagation algorithm

It is based on the gradient of the error surface that defines the direction of maximum descent (**gradient descent**)

Important parameter: **Learning rate-** sets the size of the steps in that direction

A sequence of these movements leads to a **minimum** (desirably global)

The training takes place over a number of **epochs** (iterations)

Initial network configuration (connection weights) - randomly generated

Stopping criterion: **fixed number of epochs**, elapsed time or convergence criterion based on a subset of validation examples

Training: chain rule and computation graphs

As we have already seen, a NN implements the **composition of functions (layers):**

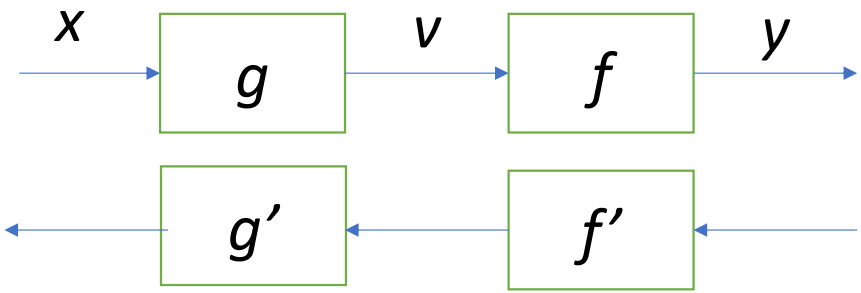
- from the calculation of the activation from the weights and inputs of each neuron
- through the application of the activation functions
- by the interconnection of neurons in successive layers, according to the topology of the network

Thus, the cost function of a NN includes many parameters (weights) defined at **various levels** in the network

Training: chain rule and computation graphs

To facilitate the calculation of the derivatives of this cost function, in relation to the weights to be optimized, the chain rule is used to calculate derivatives of composite functions

Computationally, the process of calculating the output of the NN and the cost function is organized in a **computation graph**, used for the calculation of derivatives by reversing the direction of calculation and allowing to work layer by layer



$$v = g(x)$$
$$y = f(g(x))$$

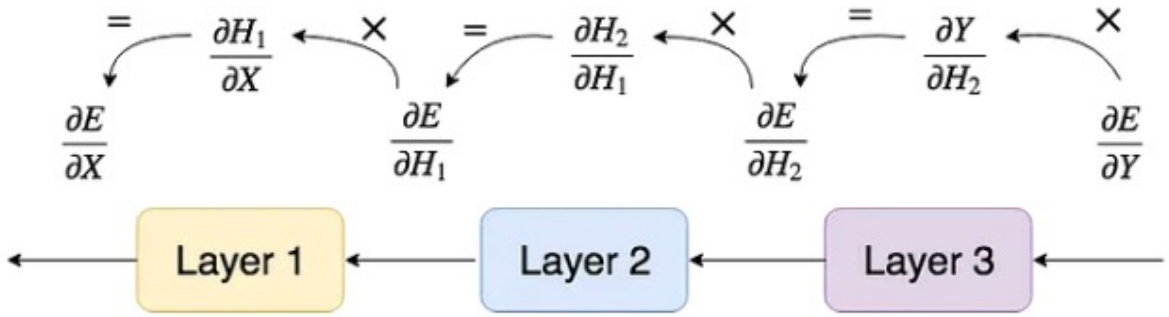
$$\frac{dy}{dx} = \frac{dy}{dv} \cdot \frac{dv}{dx}$$

Chain Rule

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$$

Training: example with feedforward NN

Forward propagation



Backward propagation

Training: example with feedforward NN: dense layer

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Dense layer

$$y_j = b_j + \sum_i x_i w_{ij}$$

Forward propagation

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = XW + B$$

Training: example with feedforward NN: dense layer

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Backward propagation

STEP 1: compute the layer input error for layer i (the output error from the layer i+1), **dE/dX**, to pass on to the layer i-1 (this will be the dE/dY of the next layer)

$$\frac{\partial E}{\partial X} = \left[\frac{\partial E}{\partial x_1} \quad \frac{\partial E}{\partial x_2} \quad \dots \quad \frac{\partial E}{\partial x_i}\right]$$

Chain rule

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \dots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$
$$= \frac{\partial E}{\partial y_1} w_{i1} + \dots + \frac{\partial E}{\partial y_j} w_{ij}$$

$$\frac{\partial E}{\partial X} = \left[\left(\frac{\partial E}{\partial y_1} w_{11} + \dots + \frac{\partial E}{\partial y_j} w_{1j}\right) \quad \dots \quad \left(\frac{\partial E}{\partial y_1} w_{i1} + \dots + \frac{\partial E}{\partial y_j} w_{ij}\right)\right]$$
$$= \left[\frac{\partial E}{\partial y_1} \quad \dots \quad \frac{\partial E}{\partial y_j}\right] \begin{bmatrix} w_{11} & \dots & w_{i1} \\ \vdots & \ddots & \vdots \\ w_{1j} & \dots & w_{ij} \end{bmatrix}$$
$$= \frac{\partial E}{\partial Y} W^t$$

Training: example with feedforward NN: dense layer

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Backward propagation

STEP 2: compute the weights' errors: **dE/dW** = X.T * dE/dY

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

Chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ij}} + \ldots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} x_i$$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1}x_1 & \cdots & \frac{\partial E}{\partial y_j}x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1}x_i & \cdots & \frac{\partial E}{\partial y_j}x_i \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} = \boxed{X^t \frac{\partial E}{\partial Y}}$$

Training: example with feedforward NN: dense layer

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Backward propagation

STEP 3: compute the biases' errors: **dE/dB** = dE/dY

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} & \frac{\partial E}{\partial b_2} & \cdots & \frac{\partial E}{\partial b_j} \end{bmatrix}$$

Chain rule

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j} = \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} = \boxed{\frac{\partial E}{\partial Y}}$$

Training: example with feedforward NN: dense layer

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Backward propagation

STEP 4: update weights and biases using gradient descent

$$w_i \leftarrow w_i - \alpha \frac{\partial E}{\partial w_i}$$

$$b_i \leftarrow b_i - \alpha \frac{\partial E}{\partial b_i}$$

Training: activation functions

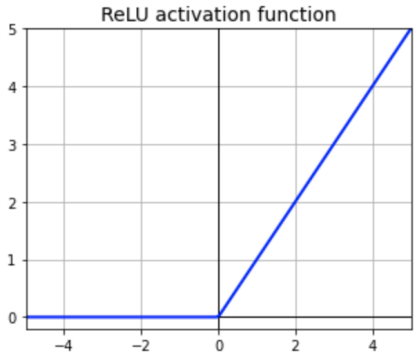
Activation functions will work similarly to other layers, being defined their forward and backpropagation computation graphs

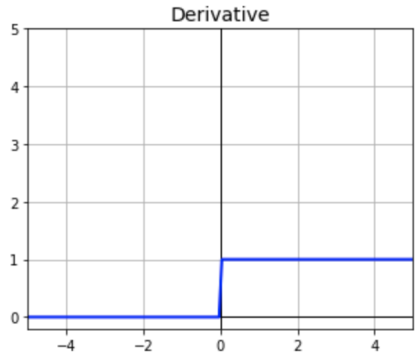
Since they do not have weights, only the error propagation step (step 1) is done for these cases

RelU

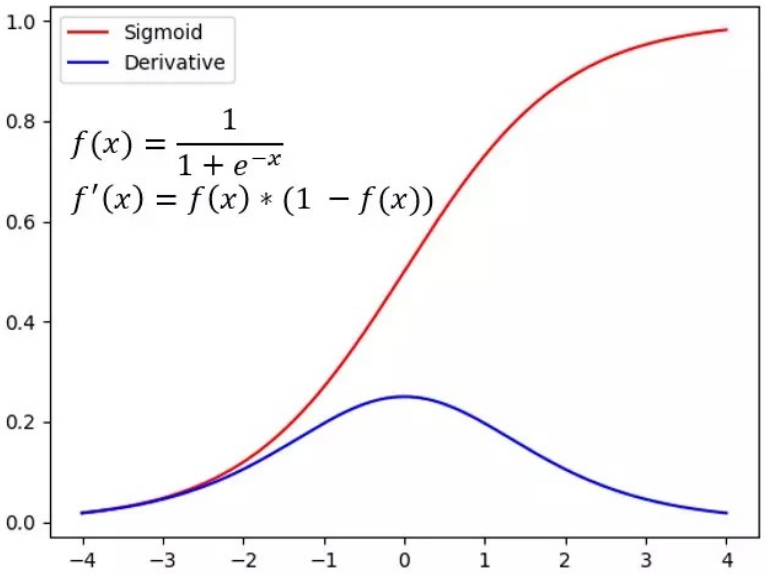
$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$



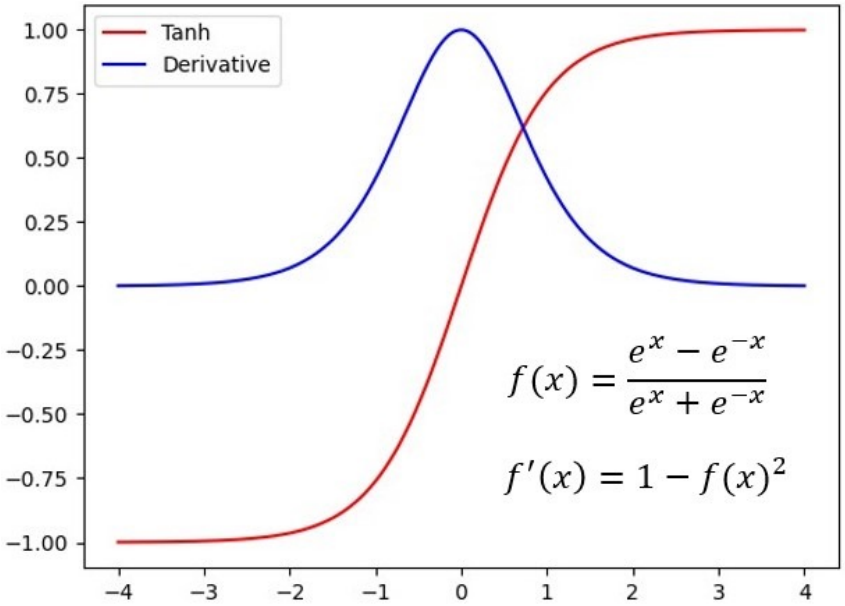


Sigmoid



Training: activation functions

Tanh



Softmax

$$f(x_i) = \frac{e^{x_i}}{\sum_j^n e^{x_j}} = \frac{e^x}{\sum e^x}$$

$$f'(x) = f(x)(1 - f(x))$$

Training: cost function

The cost function is the one that will be minimized in backpropagation

Used functions in NN are similar to those used in linear and logistic regression, being the sum of the error function for the set of individual examples

The error function (***loss function***) for each example will typically be given by:

- Sum/ mean of the squared errors (SSE/ MSE), for regression problems
- (Binary) cross entropy (used in logistic regression) for binary classification
- A generalization of the previous one to handle multiple classes

Training: loss functions in backprop

Mean squared error

$$L = \frac{1}{n}\sum_{i}^{n}\left(y_i - y_i^*\right)^2$$

$$\frac{\partial L}{\partial Y} = \begin{bmatrix} \frac{\partial L}{\partial y_1} & \cdots & \frac{\partial L}{\partial y_i} \end{bmatrix}$$

$$= \frac{2}{n}\begin{bmatrix} y_1^* - y_1 & \cdots & y_i^* - y_i \end{bmatrix}$$

$$= \frac{2}{n}(Y^* - Y)$$

Training: loss functions in backprop

Binary cross entropy

$$L = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log(y_i) + (1 - y_i)\log(1 - y_i))$$

$$\frac{\partial L}{\partial Y} = \frac{-Y}{Y^*} - \left(\frac{1 - Y}{1 - Y^*} * -1\right)$$

$$= \frac{-Y}{Y^*} + \frac{1 - Y}{1 - Y^*}$$

Categorical cross entropy

$$L = -\sum_{i=1}^{i=N} y_i \log(y_i^*)$$

$$\frac{\partial L}{\partial Y} = -\frac{Y}{Y^*}$$

Improvements on training algorithms: batches

Doing the training process using **mini-batches** (batches with part of the examples) instead of considering all available examples to calculate errors and their derivatives and update weights

Batch sizes to consider change typically between 64 and 512; *batch* should be of a size to allow keeping it as a whole in GPU memory

Batch size can be tuned/ optimized

If batch = 1 -> ***stochastic gradient descent***

If batch = number of examples in the dataset -> ***batch gradient descent***

Improvements on training algorithms: momentum

Recent algorithms use **momentum** – calculating moving average of the error gradients for the last N updates

Way to remove update noise, allowing to use higher learning rates and accelerate the convergence of the training process

Adds a new parameter (β) defining the weight to the previous value (new batch contributes with 1 – β) – typical value of 0.9

Retained gradient $\longrightarrow V_t = \beta V_{t-1} + (1-\beta)\frac{\partial L}{\partial W}$

$$W_t = W_{t-1} - \alpha V_t$$

Improvements on training algorithms: RMSProp e Adam

RMSProp also uses moving averages, but here of the squared errors (uses also a parameter β – typical value of 0.999)

Weight update divides the derivative by the square root of the moving average (adding a small value ε to avoid zero values)

Adam combines momentum with the strategy of RMSProp, so having two parameters for the update - β1 e β2 - and also ε

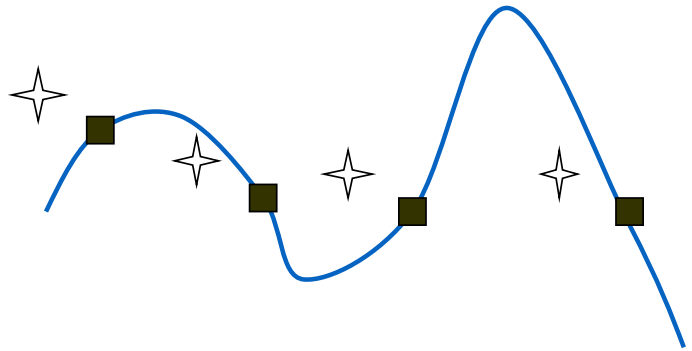
Overfitting and generalization in NNs

To train the network many epochs (training iterations) can prevent generalization and promote *overfitting*

It is preferable to stop the training in these cases – **early stopping**; error is measured in a validation dataset; if the validation error improves for a number of consecutive epochs, the training is stopped

The probability of overfitting increases if:

- Few training cases
- Many weights (complexity)



Overfitting and generalization

In DL, overfitting has been addressed using various techniques, such as **regularization** (L1 or L2, similar to linear/logistic regression, called **decay** in NNs), or explicit control of model **capacity** (e.g. number of hidden layers and number of neurons in each)

A specific technique developed for DL models is **dropout**: in each iteration of the training the output of some randomly selected neurons is put to zero. The proportion of zeroed neurons in each iteration is a parameter defined by each layer where dropout applies