Swift Introduction to Neural Networks

Roman Kaplan, November 2016

048874 Parallel Computing Architecture, Fall 2016

The description follows the online book http://neuralnetworksanddeeplearning.com/. Use it freely as a reference. A short description of its chapters:

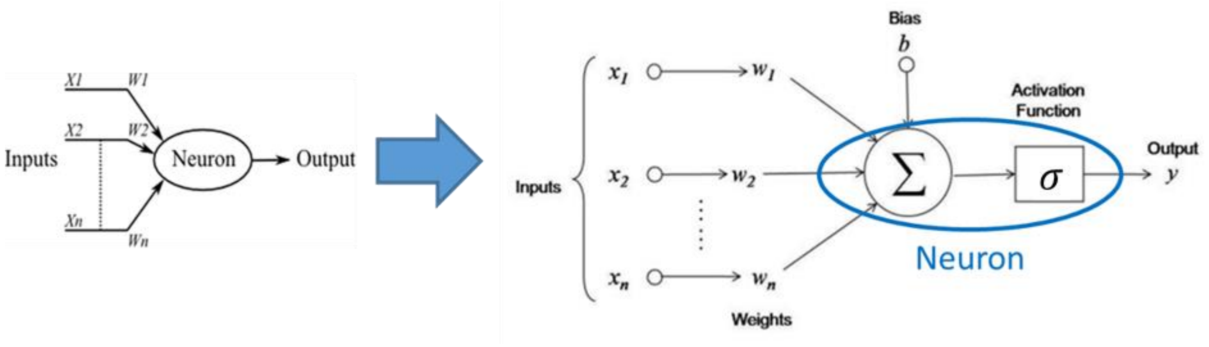
- Chapter 1 describes the example neural network we use and includes Python code to compute it (inference and training).
- Chapter 2 dwells on matrix NN and backpropagation.
- Chapter 3 revisits everything and gets deeper into mathematics.
- Chapter 4 addresses intellectual questions on NN.
- Chapter 5 discusses challenges of deep learning.
- Chapter 6 talks of convolutional (deep) NN.

1. Neuron

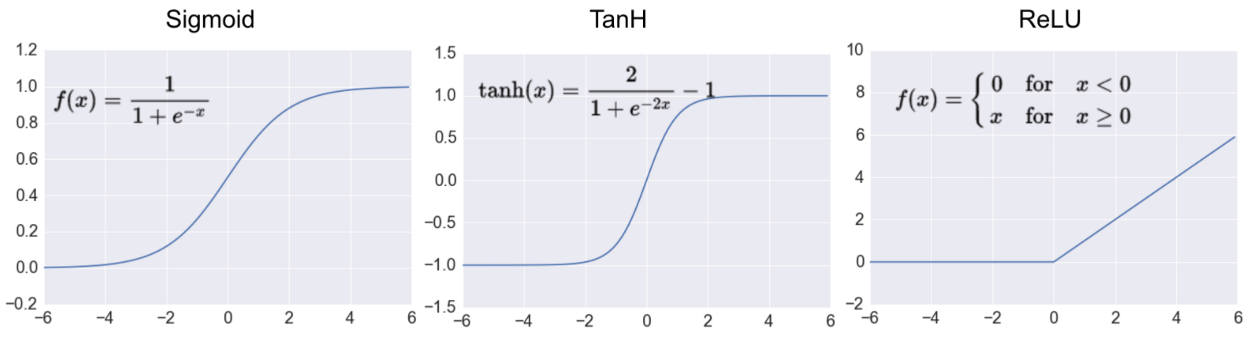
The most basic computational unit in a neural network. Has multiple inputs. Each input multiplied by a *weight* (real number). The weighted sum of inputs plus a bias is fed through a non-linear function $\sigma()$:

$$Output = \sigma(\sum w_i \cdot x_i + b) = \sigma(z)$$

The output (sometimes dented *a*) is also termed *activation* and $\sigma()$ is the *activation function,* $a = \sigma(z)$.



The most common activation functions are: 1) sigmoid (range 0,1), 2) tanh (-1,1) and 3) ReLU [0,∞) (Rectified Linear Unit).

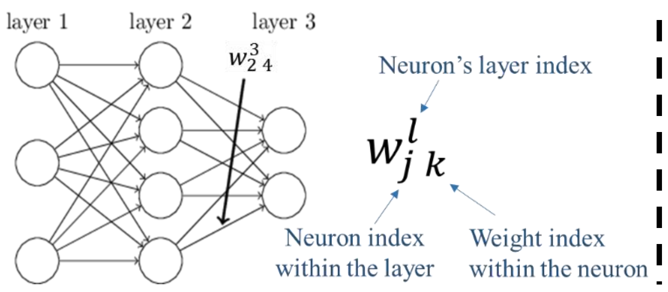


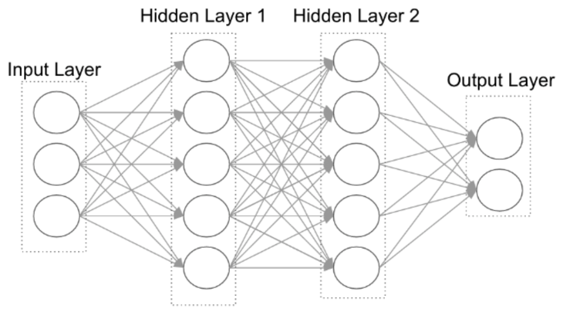
For simplicity, we ignore the bias term b in $z = \sum w_i \cdot x_i + b$ and assume $z = \sum w_i \cdot x_i$. Later we will show the full equations, including the bias term.

Further reading: http://neuralnetworksanddeeplearning.com/chap1.html#perceptrons

2. Neural Network (fully connected)

When connecting many neurons together, the complexity of implemented functions increases. The network is structured in “layers”, where a neuron output from one layer serves as an input to *all* neurons in the next layer (hence the term fully-connected neural network).





The output of each neuron in the *input layer* is constant. For example, if the input is an image consisting of pixels, then the input layer is a vector of pixels where each neuron in the input layer outputs the value of its pixel. Multiple hidden layers are possible.

Further reading:
http://neuralnetworksanddeeplearning.com/chap1.html#the_architecture_of_neural_networks

3. Calculating Network Output: Feedforward, or Inference

The input is a (constant) vector

$$input = (x_1, x_2, \dots, x_n)$$

The activation of neuron *k* in the first hidden layer is therefore

$$a_k^1 = \sigma(\sum_i w_{k,i}^1 \cdot x_i) \; = \sigma(z_k^1)$$

The weights of the first hidden layer (m-neuron layer with n inputs per neuron) can be represented as a matrix:

$$W_{m\times n}^1 = \begin{pmatrix} w_{1\,1} & \cdots (weights\ of\ neuron\ 1) \cdots & w_{1\,n} \\ \vdots & \vdots & \vdots \\ w_{k\,1} & \cdots (weights\ of\ neuron\ k) \cdots & w_{k\,n} \\ \vdots & \vdots & \vdots \\ w_{m\,1} & \cdots (weights\ of\ neuron\ m) \cdots & w_{m\,n} \end{pmatrix}$$

In matrix form,

$$a^1 = \sigma(W^1 \cdot input)$$

and the dimensions are $[m, 1] = [m, n] \times [n, 1]$. The activations of the first hidden layer serve as inputs to the next layer. Assuming a single hidden m-neuron layer and a q-neuron output layer, the output is

$$output = \sigma(W^{output} \cdot a^1)$$

with dimensions $[q, 1] = [q, m] \times [m, 1]$. This calculation is called *feedforward* because the input is propagated through all layers to the output. It is also termed *inference* because the output is inferred from the input by a given neural network. In contrast, *learning* (described below) modifies the neural network given many inputs and (possibly) respective desired outputs.

3.1 Softmax Layer – Solving a Classification Problem

If the network is used as a classifier, it is common to add 'softmax' stage at the end to force all outputs to [0,1] range. In this case, we consider output *k* to represent the probability that the input belongs to the *k'th* class. Softmax also assures that the sum of all outputs is 1. Softmax converts the output vector z (of arbitrary real values—the output layer may not need $\sigma()$ when followed by softmax), to vector $\sigma(z)$ of real values in the range (0,1) which add up to 1.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{q} e^{z_k}} \qquad \text{for } j = 1, \dots, q$$

Practical issue: Numeric stability. When writing code for computing the softmax function in practice, the intermediate terms e^{z_j} and $\sum_{k=1}^{q} e^{z_k}$ may be very large due to exponentials and may exceed the maximum magnitude allowed by the precision chosen for computation (for instance, if 16 bit fixed point format is employed, the maximum number that can be represented is $2^{15} - 1 = 32{,}767$). In softmax computation this problem may be address by certain normalization. One such normalization trick is to expand the fraction by a sufficiently small constant C. We can then "push" C to the exponents as follows:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{q} e^{z_k}} = \frac{C \cdot e^{z_j}}{C \cdot \sum_{k=1}^{q} e^{z_k}} = \frac{e^{z_j + logC}}{\sum_{k=1}^{q} e^{z_k + logC}}$$

One common choice for C is to set $logC = -\max_j z_j$. This choice implies shifting the values of the vector z so that the highest value is zero. All exponents are then fractions. This choice works well for floating point representation. Computing with fixed point numbers may require other choices.

4. Example: NN Recognition / Classification of the MNIST Hand-Written Digits Dataset

The MNIST data set contains 70,000 images of hand-written digits, 28×28=784 eight-bit pixels each. The goal of the NN is to classify each image into one of ten digits (each such classification is an inference). Consider a NN with a 784-neuron input layer (the 2D matrix of pixels is rearranged as a vector), one hidden layer of 100 neurons using ReLU activation and one output layer with 10 neurons (the number of neurons in the hidden layer, 100, is chosen arbitrarily). All bias values are assumed zero. The computation can be described as follows. The NN is applied to a single image at a time.

The input vector is $p_{[784,1]}$. The hidden layer, using ReLU activation, computes

$$a^1_{[100]} = ReLU(W^1_{[100,784]} \cdot p_{[784]})$$

Subscripts indicate dimensions. The output layer (without activation) computes

$$q_{[10]} = W^{output}_{[10,100]} \cdot a^1_{[100]}$$

To efficiently compute softmax, the following steps may be applied.

$$x_{[10]} = e^{q_{[10]}} = \begin{pmatrix} e^{q_1} \\ e^{q_2} \\ \cdots \\ e^{q_{10}} \end{pmatrix}$$

$$SX = \sum_{k=1}^{10} x(k)$$

$$\sigma_{[10]} = \frac{x_{[10]}}{SX} = \begin{pmatrix} e^{q_1}/_{SX} \\ e^{q_2}/_{SX} \\ \cdots \\ e^{q_{10}}/_{SX} \end{pmatrix}$$

Clearly, the two weight matrices must be pre-computed and known.

Further reading:
http://neuralnetworksanddeeplearning.com/chap1.html#a_simple_network_to_classify_handwritten_digits

5. Convolutional Neural Network

When processing very large inputs it may be impractical to connect all inputs to all neurons in the first hidden layer, as in the fully-connected NN. The same difficulty applies to the internal hidden layers. Instead, in convolutional neural networks (CNN) each neuron may process only a small subset of the previous layer. When the same set of weights (kernel) is used in many neurons of same layer, to cover the complete set of subsets of a previous layer, the computation is clearly a convolution.

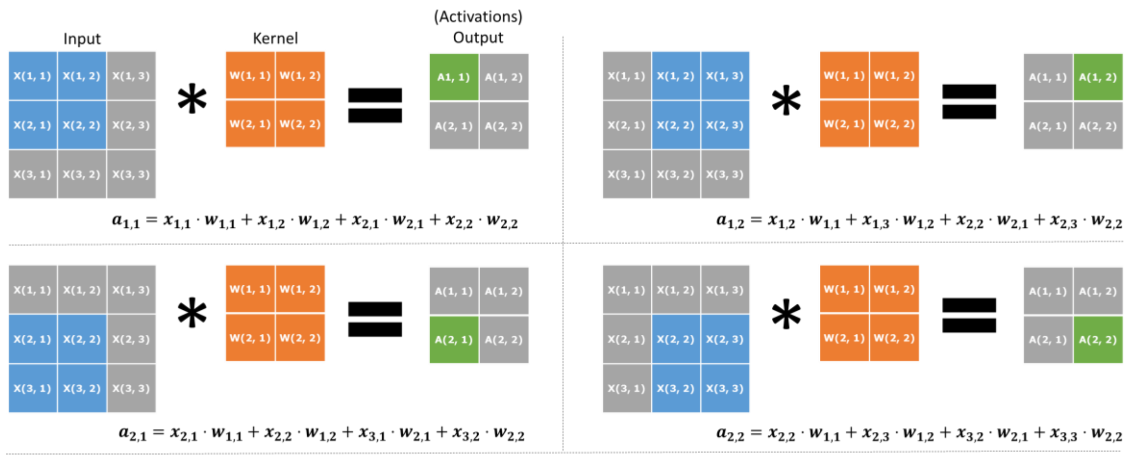
For instance, when the NN processes images, convolutions are typically employed to detect image features. Since multiple features may be sought in an image, a layer may comprise of many separate convolutions, all applied to the same previous layer.

Data reduction steps, such as pooling, are usually applied in between layers of convolutions. After several (or many) such layers, several fully-connected layers may be employed for final classification. The following sub-sections briefly describe these structures.

5.1 Convolution Layer

Each step of the convolution (one linear combination of values, applying one kernel) is performed by one neuron. The neuron receives only a local region of the previous layer. The neuron may be viewed as a filter. The idea is to apply a small filter on the entire input by sliding the filter across the width and height of the (two-dimensional) input.

Consider the following example of a 3×3 input convolved with a 2×2 filter kernel. The kernel is shifted by 1 pixel in either or both dimensions for other output values. The size of this shift is called *stride*.


Input
Kernel
(Activations) Output
X(1, 1) X(1, 2) X(1, 3)
X(2, 1) X(2, 2) X(2, 3)
X(3, 1) X(3, 2) X(3, 3)
W(1, 1) W(1, 2)
W(2, 1) W(2, 2)
A1, 1) A(1, 2)
A(2, 1) A(2, 2)
$a_{1,1} = x_{1,1} \cdot w_{1,1} + x_{1,2} \cdot w_{1,2} + x_{2,1} \cdot w_{2,1} + x_{2,2} \cdot w_{2,2}$
$a_{1,2} = x_{1,2} \cdot w_{1,1} + x_{1,3} \cdot w_{1,2} + x_{2,2} \cdot w_{2,1} + x_{2,3} \cdot w_{2,2}$
$a_{2,1} = x_{2,1} \cdot w_{1,1} + x_{2,2} \cdot w_{1,2} + x_{3,1} \cdot w_{2,1} + x_{3,2} \cdot w_{2,2}$
$a_{2,2} = x_{2,2} \cdot w_{1,1} + x_{2,3} \cdot w_{1,2} + x_{3,2} \cdot w_{2,1} + x_{3,3} \cdot w_{2,2}$

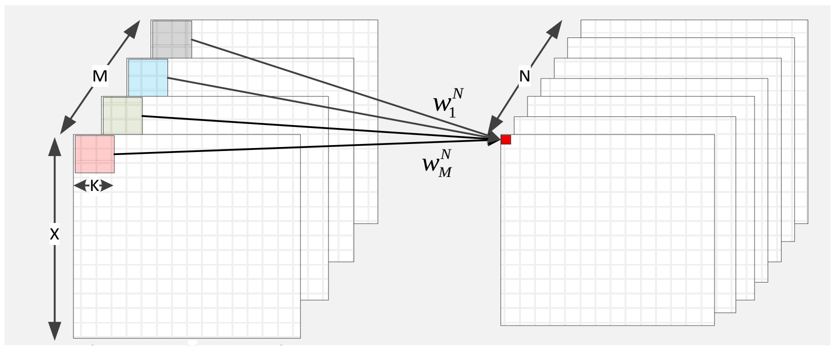
Convolution of a single 3×3 input array with a 2×2 size filter with stride of 1.

Mathematically, the convolution of a $K \times K$ kernel w with 2D input array x produces activation array a.

$$a_{i,j} = \sum_{p=1}^{K} \sum_{q=1}^{K} x_{i+p-1,j+q-1} \cdot w_{p,q}$$

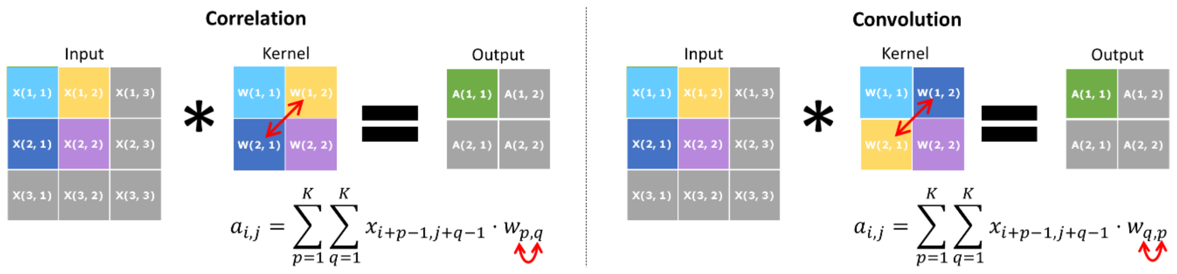
Care must be applied to array boundaries. Convolving an input with a kernel results in an output of somewhat smaller dimensions than the input, as can be seen in the above figure. For simplicity, it may be desirable for the output to be of the same size as the input. Therefore, a zero padding of the image may be applied by adding rows and columns of zeroes to the input boundaries.

In practice, multiple filters are applied to the same input, resulting in multiple outputs. These outputs are called feature maps. Every feature map uses a different filter (kernel matrix). Consequently, the input of a convolutional layer can also be composed of multiple feature maps. Every input feature map has its own filter. An output feature map equals the sum of combined filter results of all feature maps. Every output feature map has a different set of kernel to be applied on all input feature maps. Therefore, in a convolutional layer, usually there are multiple input feature maps and multiple output feature maps. The following figure demonstrates this concept.



General convolutional layer. M input and N output feature maps. Filter kernels are of size $K \times K$. Every output feature map has its set of kernel filter to be applied on the input feature maps.

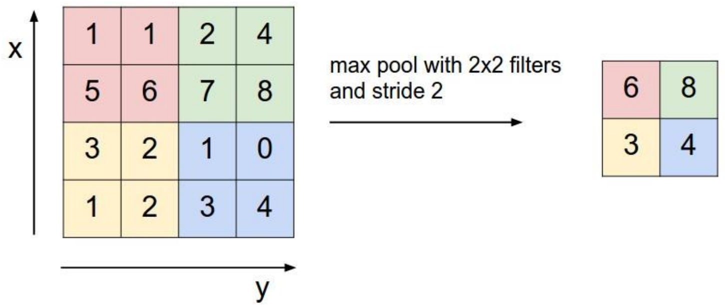
<u>*Side note*</u>: the definition of convolution is multiplying the transpose of the above kernel matrix by the matching input window. The above operation is actually correlation. Both operations, convolution and correlation, are similar and differ only in their indexing notation. For simplicity, despite calling the layer in the neural network “convolutional layer,” we use the correlation indexing notation. The following figure demonstrates the difference between convolution and correlation.



Difference in indexing notation between correlation and convolution.

5.2 Pooling Layer (also called *Subsampling*)

The pooling operation takes a small region of the input and outputs a single value. The common max pooling outputs the maximal value of the region and discards the other values. Below is an image demonstrating max pooling. Other options for pooling include average pooling or L2-norm pooling. As with the convolutional layer, stride size is a parameter. The main purpose of pooling is to progressively reduce the data.



5.3 Putting it All Together – The *LeNet* Convolutional Neural Net

CNN, the precursor of *Deep Learning*, was originally developed in the 1980s. The LeNet CNN (see Yann Le Cun 1989[1] and 1998[2]) was applied to recognition of the handwritten digits database. We discuss here the complete architecture of LeNet-5 as presented in the 1998 paper. LeNet-5 comprises the following 7 layers, not including the input.

0. The input is a 32×32 pixel image.
1. C1: Convolution layer with 6 feature maps. Kernel sizes of 5×5 and a stride of 1.
2. S2: Subsampling (pooling) with filter sizes of 2×2 and a stride of 2.
3. C3: Convolutional layer with 16 feature maps. Kernel sizes of 5×5 and a stride of 1.
4. S4: Subsampling with filter sizes of 2×2 and a stride of 2.
5. C5: Convolutional layer with 120 feature maps. Kernel sizes of 5×5 (= input size).
6. F6: Fully connected layer with 84 units and $tanh$ activation.
7. Output: 10 neurons

Refer to the 1998 paper for details.

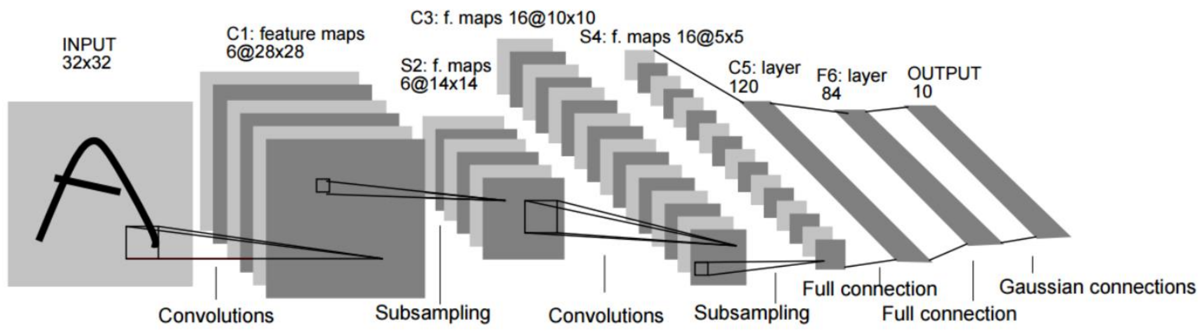


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

State-of-the-art CNNs (deep learning networks, such as Alex Net[3]) are larger in almost every aspect: larger inputs with RGB channels, more convolutional layers, larger fully connected layers and more output classes. In addition, LeNet uses activation and output functions which are no longer in widespread use. Nowadays, the fully connected layers are usually ReLU and the output layer is a softmax.

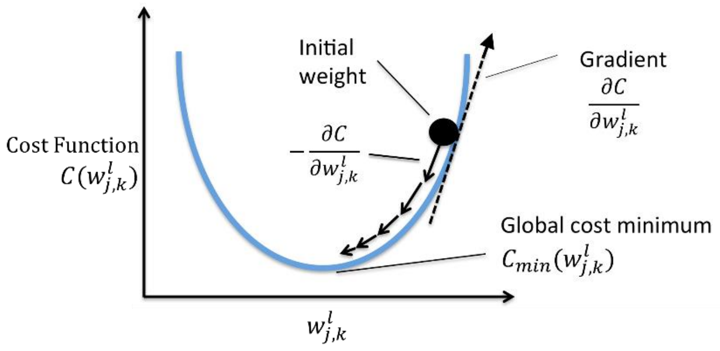
[1] Le Cun, Yann, et al. "Handwritten digit recognition: Applications of neural network chips and automatic learning." *IEEE Communications Magazine* 27.11 (1989): 41-46.
[2] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.
[3] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.

6. Learning: Setting Net Weights by Backward Propagation (Backpropagation)

Setting all net weights manually is impractical. Therefore we seek a "learning" algorithm to automatically determine the optimal weights. The learning algorithm should minimize some *cost function, C*. The cost function is a measure for the distance of the output from its target. Therefore, the cost is calculated from output layer activations (net's output) and the corresponding target values for each output. Target values mark the right class for that output, meaning they are known before learning has started. Every input sample (e.g., image) used for training has its known target. This is called a *supervised learning* process, where training is performed from a set of inputs with known target values.

An example cost function is the *Mean Squared Error (MSE)*: $C(x) = \frac{1}{2}\sum_i (target(x)_i - a_i^L(x))^2$.

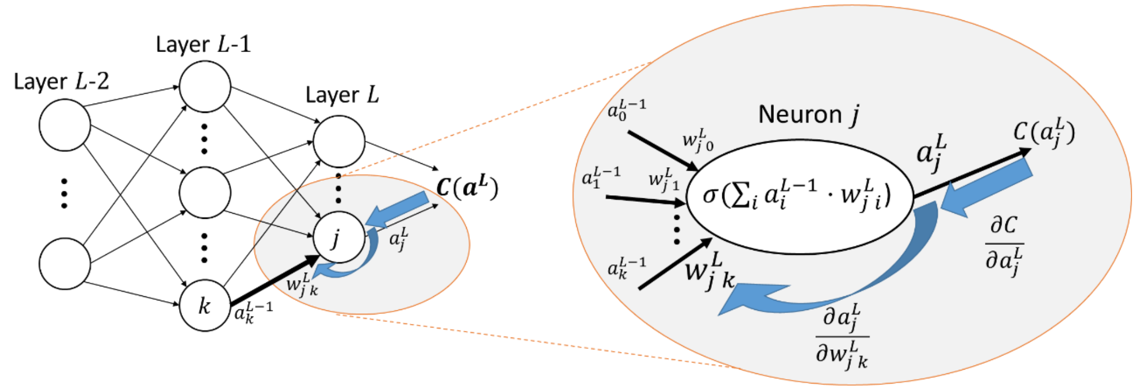
During backpropagation, each weight should be updated to minimize the cost function. For each weight, $w_{j,k}^l$, the partial derivative of the cost function with respect to that weight, $\frac{\partial C}{\partial w_{j,k}^l}$, sets the direction of update. This direction should be opposite to the derivative. The following figure provides intuition into the learning process.


Initial weight
Gradient $\frac{\partial C}{\partial w_{j,k}^l}$
Cost Function $C(w_{j,k}^l)$
$-\frac{\partial C}{\partial w_{j,k}^l}$
Global cost minimum $C_{min}(w_{j,k}^l)$
$w_{j,k}^l$

Notes: Descending the gradient towards the desired target is appropriately called "Gradient Descent" (GD). Usually, this process achieves only a local minimum, if there are multiple minima. Selecting the steepest gradient (when multiple gradients are possible) is called "Steepest Gradient Descent." In large NNs, a Stochastic Gradient Descent (SGD) is typically employed.

Calculating $\frac{\partial C}{\partial w_{j\,k}^{l}}$ requires to know the relation between C and $w_{j\,k}^{l}$. For a hidden layer neuron this relation is hard to reach due to the distribution of neuron's activation to all neurons in the next layer. The simplest $\frac{\partial C}{\partial w_{j\,k}^{l}}$ to calculate is of a weight in the output layer L.This calculation relies on the chain rule.

Assuming a MSE cost function, then $\frac{\partial C}{\partial w_{j\,k}^{L}}$ with respect to some output neuron weight, $w_{j\,k}^{L}$, can be visually explained as follows:



The full calculation:

	Clarification
$\frac{\partial C}{\partial w_{j\,k}^{L}} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial w_{j\,k}^{L}}$ (chain rule)	
$= \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial \sigma\left(\sum_i w_{j\,i}^{L} \cdot a_i^{L-1}\right)}{\partial w_{j\,k}^{L}}$	$a_j^L = \sigma\left(\sum_i w_{j\,i}^{L} \cdot a_i^{L-1}\right) = \sigma(z_j^L)$
$= \left(target_j - a_j^L\right) \cdot \sigma'\left(z_j^L\right) \cdot a_k^{L-1}$	$C = \frac{1}{2}\sum_i \left(target(x)_i - a_i^L(x)\right)^2$, but when $i = j$ then: $\frac{\partial\left(target(x)_j - a_j^L(x)\right)}{\partial w_{j\,k}^{L}} \neq 0$
$= \delta_j^L \cdot a_k^{L-1}$	• $\left(target_j - a_j^L\right) \cdot \sigma'\left(z_j^L\right) = \delta_j^L$ • $\sigma'\left(z_j^L\right)$ can be calculated from the activation function • a_k^{L-1} is the activation of neuron k in layer L-1

Notes:

1. $\boldsymbol{\left(target_j - a_j^L\right) \cdot \sigma'\left(z_j^L\right) = \delta_j^L}$ is the *output error*. It marks the difference between output neuron activation and its target value.
2. $\boldsymbol{\sigma'\left(\sum_i w_{j\,i}^{L} \cdot a_i^{L-1}\right) = \sigma'\left(z_j^L\right)}$ is activation function's derivative. For example, if $\sigma()$ is the ReLU function, then $\sigma'(\boldsymbol{z_j^L} \leq 0) = 0$ and $\sigma'\left(\boldsymbol{z_j^L} > 0\right) = 1$.

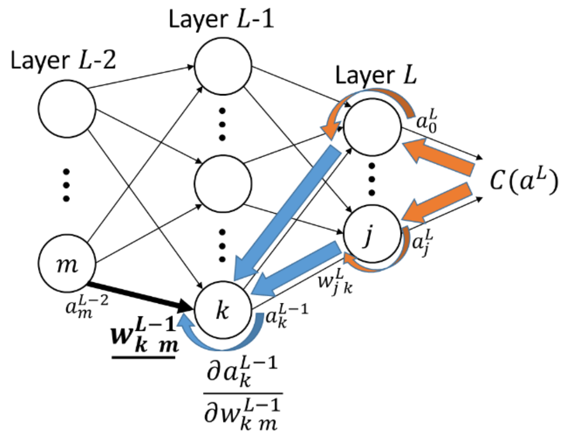
In matrix notation, all partial derivatives of the cost for an output neuron j is:

$$\frac{\partial C}{\partial w_j^L} = \delta_j^L \cdot a^{L-1}, \quad \text{where } \delta_j^L = \nabla_a C \odot \sigma'(z_j^L)$$

The 'Hadamard Product' operator $\odot$ is applied as follows: $\begin{bmatrix}2\\3\end{bmatrix} \odot \begin{bmatrix}5\\6\end{bmatrix} = \begin{bmatrix}2\cdot 5\\3\cdot 6\end{bmatrix}$ ("$.*$" in Matlab).

The $\nabla_a C$ operator is a vector whose j-th component is the partial derivative $\frac{\partial C}{\partial a_j^L}$: $\nabla_a C = \begin{pmatrix}\frac{\partial C}{\partial a_1^L}\\ \frac{\partial C}{\partial a_2^L}\\ \cdots \\ \frac{\partial C}{\partial a_N^L}\end{pmatrix}$

In case of a weight in a hidden layer neuron, neuron's activation serves as an input to all neurons in the following layer. Consider a neuron in the last hidden layer (layer L-1). Assume neuron's index is k and the weight index within the neuron is m: $w_{k\ m}^{L-1}$. The figure below presents the main terms of $\frac{\partial C}{\partial w_{j\,k}^{L-1}}$. Partial derivatives belong to the orange arrows were computed in $\frac{\partial C}{\partial w_j^L}$, blue arrows mark the currently computed terms.



Calculating $\frac{\partial C}{\partial w_{k\,m}^{L-1}}$ is similar to the equivalent computation in the output layer:

$$\frac{\partial C}{\partial w_{k\,m}^{L-1}} = \frac{\partial C}{\partial a_k^{L-1}} \cdot \frac{\partial a_k^{L-1}}{\partial w_{k\,m}^{L-1}} \quad \text{(chain rule)}$$

$$= \frac{\partial C}{\partial a_k^{L-1}} \cdot \frac{\partial \sigma\left(\sum_i w_{k\,i}^{L-1} \cdot a_i^{L-2}\right)}{\partial w_{k\,m}^{L-1}}$$

$$= \sum_i \left(\frac{\partial C}{\partial a_i^L} \cdot \frac{\partial a_i^L}{\partial a_k^{L-1}}\right) \cdot \sigma'\left(z_k^{L-1}\right) \cdot a_m^{L-2}$$

$$= \sum_i \left(\delta_i^L \cdot w_{i\,k}^L\right) \cdot \sigma'\left(z_k^{L-1}\right) \cdot a_m^{L-2}$$

$$= \boldsymbol{\delta_k^{L-1}} \cdot \boldsymbol{a_m^{L-2}}$$

Clarification

- $z_k^{L-1} = \sum_i w_{k\,i}^{L-1} \cdot a_i^{L-2}$
- $\frac{\partial a_k^{L-1}}{\partial w_{k\,m}^{L-1}} = \sigma'\left(z_k^{L-1}\right) \cdot a_m^{L-2}$
- a_k^{L-1} is an input in all output layer neurons, therefore $\frac{\partial C}{\partial a_k^{L-1}} = \sum_i \left(\frac{\partial C}{\partial a_i^L} \cdot \frac{\partial a_I^L}{\partial a_I^{L-1}}\right)$

- $\frac{\partial C}{\partial a_i^L} = target_i - a_i^L$ from layer L.
- $\frac{\partial a_i^L}{\partial a_k^{L-1}} = \frac{\partial \sigma\left(\sum_t w_{i\,t}^L \cdot a_t^{L-1}\right)}{\partial a_k^{L-1}} = \sigma'\left(z_k^{L-1}\right) \cdot w_{i\,k}^L$

- $\delta_k^{L-1} = \sum_i \left(\delta_i^L \cdot w_{i\,k}^L\right) \cdot \sigma'\left(z_k^{L-1}\right)$

By applying the process on all layers, starting from the output layer L, we can reach a general iterative formula for each layer l:

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

  It can also be written as $\delta^l = (\left(\delta^{l+1}\right)^T \cdot w^{l+1}) \odot \sigma'(z^l)$ to avoid transposing the weights matrix.
- $\frac{\partial C}{\partial w_{j\,k}^l} = \delta_j^l \cdot a_k^{l-1}$

For the output layer, the output error is calculated as follows:

- $\delta^L = \nabla_a C \odot \sigma'(z^L)$

Backpropagation with a Softmax Output Layer

In case of a softmax output layer, MSE cost function does not fit well. A good cost function is the cross entropy:

$$C = -\sum_{i \in output} target_i \cdot \log(a_i)$$

The output activations equal to $a_j^L = \sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{q} e^{z_k}}$. The above backpropagation equations will then be changed to:

- $\delta_i^L = a_i^L - target_i$
- $\frac{\partial C}{\partial w_{i\,j}^L} = \delta_i^L \cdot a_j^{L-1}$

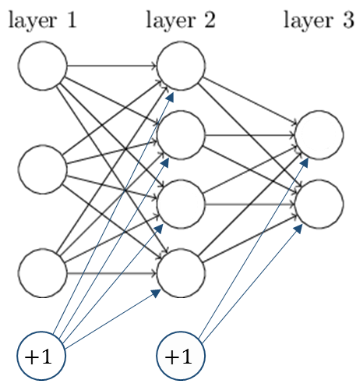
When calculating δ^L note that softmax is a classifier, therefore only a single target value will equal to 1, the rest will equal to 0.

The above softmax terms apply only for the output layer since the hidden layers don't use softmax.

The Bias Term

In Section 1 we have seen the bias term in $z = \sum w_i \cdot x_i + \boldsymbol{b}$, but omitted it. Since it is a constant scalar which also requires to update during learning, similar to the weights, it can be treated as a constant 1 neuron input, which also has a weight.

In this case, we can rewrite z as follows: $z = w_i \cdot x_i + \boldsymbol{w_{N+1}} \cdot 1$. Since the bias weight input is constant 1, the partial derivative of $\frac{\partial C}{\partial w_{j\,N+1}^l} = \delta_j^l$.



The Backpropagation algorithm

1. **Input $\boldsymbol{x}$**: Set the corresponding activation a^1 for the input layer.
2. **Feedforward**: For each $l = 2,3, \ldots, L$ compute: $z^l = w^l a^{l-1}$ and $a^l = \sigma(z^l)$
3. **Output error $\boldsymbol{\delta^L}$**: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. **Backpropagate the error**: For each $l = L\text{-}1, L\text{-}2, \ldots, 2$ compute:
$$\delta^l = \left(\left(w^{l+1}\right)^T \cdot \delta^{l+1}\right) \odot \sigma'(z^L)$$
5. **Output**: The cost function gradient is given by $\frac{\partial C}{\partial w_{j,k}^l} = a_k^{l-1} \cdot \delta_j^l$

7. Putting It All Together – Training the Net

Training the net requires both algorithms, feedforward and backpropagation. In practice, it is common to combine backpropagation with a learning algorithm such as a stochastic gradient descent, in which we compute the gradient for a group of training examples. Such a group is called a *mini-batch*, marked m. The *learning rate* ratio η sets the amount of change for each weight in the opposite direction to the gradient.

The learning algorithm based on stochastic gradient descent with a mini-batch of size m:

1. For each sample x out of the total m samples, do:
   i) **Feedforward**: For each $l = 2,3,..,L$ compute
   $$z^l(x) = w^l \cdot a^{l-1}(x) \text{ and } a^l(x) = \sigma(z^l(x))$$
   ii) **Output error $\boldsymbol{\delta^L(x)}$**: Compute the vector
   $$\delta^L(x) = \nabla_a C \odot \sigma'(z^L(x))$$
   iii) **Backpropagation**: For each $l = L\text{-}1, L\text{-}2, \dots, 2$ compute
   $$\delta^l(x) = \left(\left(w^{l+1}\right)^T \cdot \delta^{l+1}(x)\right) \odot \sigma'\left(z^l(x)\right)$$

2. **Gradient Descent**: For each $l = L, L\text{-}1, \dots 2$, update the weights according to the rule:
$$w^l \leftarrow w^l - \frac{\eta}{m}\sum_x \delta^l(x) \cdot \left(a^l(x)\right)^T$$

In the last step, we update the weights in the opposite direction to the gradient. For each weight, all m mini-batch gradients are averaged and multiplied by the learning rate. To achieve cost function convergence, the learning rate is usually small, such as $\eta = 0.05$.