The backpropagation algorithm

Backpropagation algorithm

The backpropagation algorithm is simply the instantiation of the gradient descent technique to the case of neural networks.

Specifically, it provides iterative rules to compute partial derivatives of the loss function with respect to each parameter of the network.

In the following, we shall discuss it in the case of dense networks and shall hint to extensions to convolutional networks and recurrent networks at proper places.

Computing the gradient

In the previous lesson we computed the gradient by hand for a simple linear net.

But a neural network computes a complex function obtained by composition of many neural layers. How can we compute the gradient w.r.t. a specific parameter (weight) of the net?

We need a mathematical rule known as the chain rule (for derivatives).

The chain rule

Given two derivable functions f, g with derivatives f' and g', the derivative of the composite function $h(x) = f(g(x))$ is

$$h'(x) = f'(g(x)) * g'(x))$$

Equivalently, letting $y = g(x)$,

$$h'(x) = f'(g(x)) * g'(x)) = f'(y) * g'(x) = \frac{df}{dy} * \frac{dg}{dx}$$

Multivariate chain rule

Given a multivariable function $f(x, y)$ and two single variable functions $x(t)$ and $y(t)$

$$\underbrace{\frac{d}{dt}f(x(t), y(t))}_{\text{derivative of composition}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

In vector notation: let $\mathbf{v}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$, then

$$\underbrace{\frac{d}{dt}f(x(t), y(t))}_{\text{derivative of composition}} = \underbrace{\nabla f \cdot \mathbf{v}'(x)}_{\text{dot product of vectors}}$$

where ∇f is the gradient of f, i.e. the vector of partial derivatives.

The function computed by the net

logistic units at layer ℓ compute the function

$$a^\ell = \sigma(b^\ell + w^\ell \cdot x^\ell)$$

- a^ℓ is the activation vector at layer ℓ
- $z^\ell = b^\ell + w^\ell \cdot x^\ell$ is the weighted input at layer ℓ
- $x^{\ell+1} = a^\ell, x^1 = x$

The function computed by the neural net is

$$\sigma(b^L + w^L \cdot \ldots \sigma(b^2 + w^2 \cdot \sigma(b^1 + w^1 \cdot x^1)))$$

The dimensions of w^ℓ e b^ℓ depend on the number of neurons at layer ℓ (and $\ell - 1$).

All of them are parameters of the models.

Overall structure (single input)

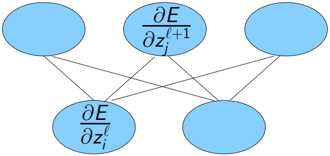
- Compute the gradient of the error

- Backpropagate error derivatives to activations and weighted input of hidden layers, using the chain rule.

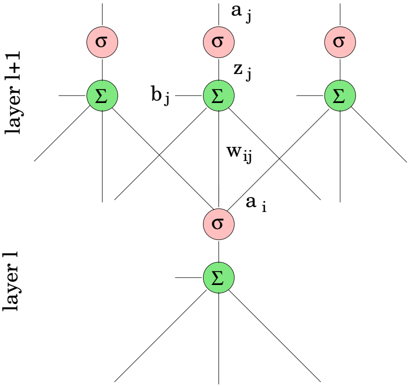
- derive error derivatives at layer ℓ w.r.t. each parameter of the layer

$$E = \frac{1}{2}\sum_j (a_j^L - t_j)^2$$

$$\frac{\partial E}{\partial a_j^L} = -({a_j}^L - t_j)$$



Backpropagation rules



$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial a_j}\frac{da_j}{dz_j} = \frac{\partial E}{\partial a_j}\sigma'(z_j)$$

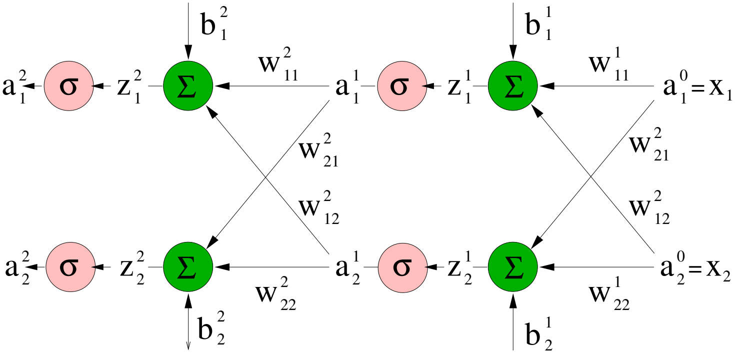
$$\frac{\partial E}{\partial a_i} = \sum_j \frac{\partial E}{\partial z_j}\frac{dz_j}{da_i} = \sum_j \frac{\partial E}{\partial z_j} w_{ij}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j}\frac{dz_j}{dw_{ij}} = \frac{\partial E}{\partial z_j} a_i$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial z_j}\frac{dz_j}{db_j} = \frac{\partial E}{\partial z_j}$$

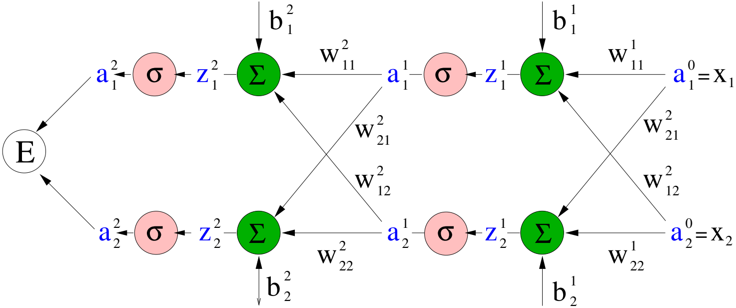
An example

Consider the following network



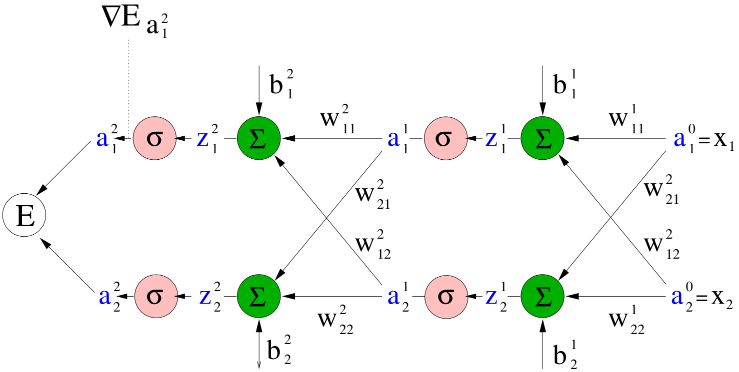
Forward pass

Let E be the error.
Take a sample $\langle x, y \rangle$ and compute the vectors $\mathbf{z}^l, \mathbf{a}^l$ at each layer l.



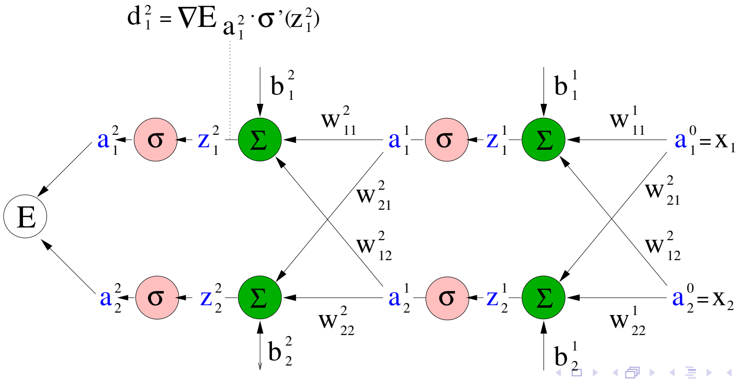
Backward pass (1)

First, we compute the partial derivative of the error, w.r.t the last activations:



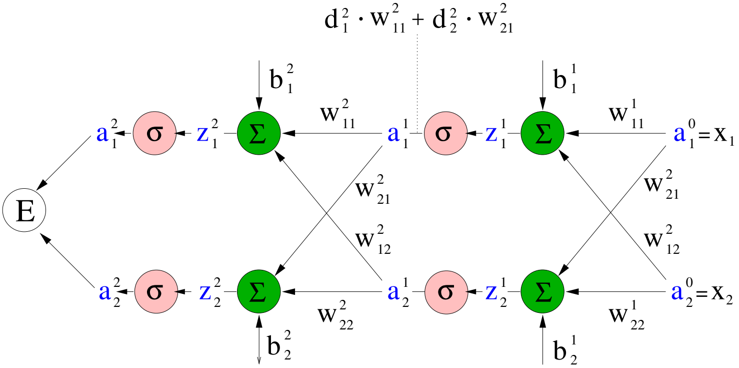
Backward pass (2)

Then, according to the chain rule, we backpropagate the derivative through a function at a time, multiplying by the partial derivative of the function we traversed:



Backward pass (2)

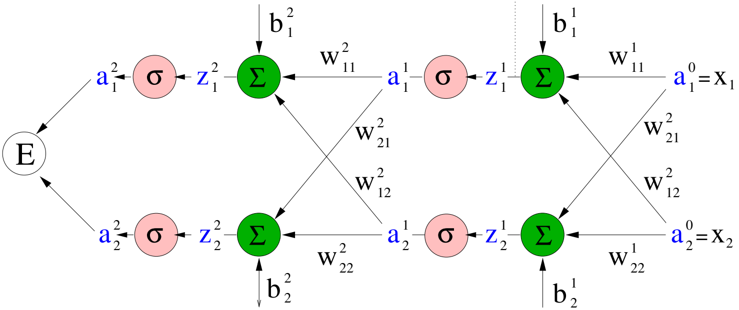
When an activation is shared by multiple units, we need to sum together the contributions of the partial derivatives along all directions:



Backward pass (3)

We reached the z of the previous layer, and we repeat the same computation through all layers:

$$d^1_1 = (d^2_1 \cdot w^2_{11} + d^2_2 \cdot w^2_{21}) \cdot \sigma'(z^1_1)$$



Backpropagation rules in vectorial notation

Given some error function E (e.g. euclidean distance) let us define the error derivative at l as the following vector of partial derivatives:

$$\delta^l = \frac{\partial E}{\partial z^l}$$

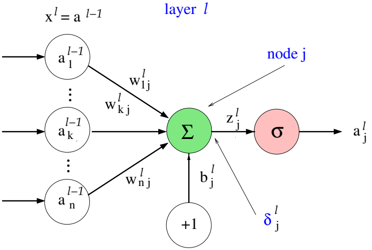
We have the following equations

$$(BP1) \quad \delta^L = \nabla_{a^L} E \odot \sigma'(z^L)$$

$$(BP2) \quad \delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$(BP3) \quad \frac{\partial E}{\partial b_j^l} = \delta_j^l$$

$$(BP4) \quad \frac{\partial E}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$



where $\odot$ is the Hadamard product (component-wise)

The backpropagation algorithm

- **input** $\langle x, y \rangle$: $a^0 = x$
- **feedforward:** for $l = 1, 2, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ e $a^l = \sigma(z^l)$
- **output error**: compute the vector $\delta^L = \nabla_a E \odot \sigma'(z^L)$
- **backpropagation**: per $l = L-1, L-2, \ldots, 1$ compute $\delta^l = ((w^l)^T \delta^{l+1}) \odot \sigma'(z^l)$
- **updating** for $l = 1, 2, \ldots, L$ update the parameters in the following way:
  - $w^l_{jk} \rightarrow w^l_{jk} + \mu a^{l-1}_k \delta^l_j$
  - $b^l_j = \!\!\rightarrow b^l_j + \mu \delta^l_j$

Derivatives of common activation functions

activation function	**derivative**
logistic function	
$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$
hyperbolic tangent	
$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$tanh'(x) = sech^2(x)$
rectified linear	
$relu(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$	$relu'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

An instance of the backpropagation algorithm

If σ is the logistic function, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
If $E(y) = \frac{(y-a)^2}{2}$, then $\nabla_a E = y - a$.

- **input** $\langle x, y \rangle$: $a^0 = x$
- **feedforward:** for $l = 1, 2, \ldots, L$ compute $z^l = w^l a^l + b^l$ e $a^l = \sigma(z^l)$
- **output error**: Compute the vector $\delta^L = \nabla_a E \odot \sigma'(z^L) = (y - a^L) \odot (a^L) \odot (1 - a^L)$
- **backpropagation**: for $l = L-1, L-2, \ldots, 1$ compute $\delta^l = ((w^l)^T \delta^{l+1}) \odot \sigma'(z^l) = ((w^l)^T \delta^{l+1}) \odot (a^l) \odot (1 - a^l)$
- **updating** for $l = 2, 3, \ldots, L$ update the parameters:
  - $w^l_{jk} \rightarrow w^l_{jk} + \mu a^{l-1}_k \delta^l_j$
  - $b^l_j = \rightarrow b^l_j + \mu \delta^l_j$

A neural network from scratch

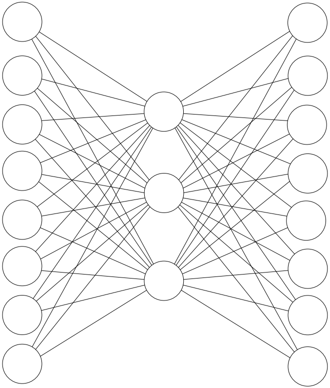
In the following slides we shall use the backpropagation rules to build a neural network (a simple autoencoder) **from scratch**.

This has only a **didactical interest**.

In practice, we have **domain specific languages** (pytorch, tensorflow, keras, . . .) that allows us to build complex neural networks in a very simple way.

A simple autoencoder

input	output
10000000	10000000
01000000	01000000
00100000	00100000
00010000	00010000
00001000	00001000
00000100	00000100
00000010	00000010
00000001	00000001



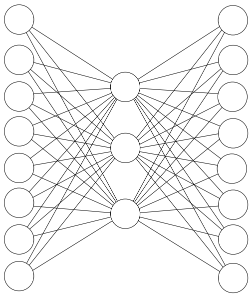
Can we learn this function with this net?
Can we learn the identity function?

The function computed by the net

The function computed by the net is:

$o(x) = \sigma(b^2 + W^2 \cdot \sigma(b^1 + W^1 \cdot x))$

parameters	b^1	b^2	W^1	W^2
dimensions	3	8	8×3	3×8



The net has 59 parameters

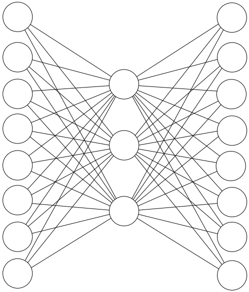
Demo!

python code

```
def update(x,y):
  #input
  a[0] = x
  #feed forward
  for i in range(0,l-1):
    z[i] = np.dot(a[i],w[i])+b[i]
    a[i+1] = np.vectorize(activate)(z[i])
  #output error
  d[l-2] = (y - a[l-1])*np.vectorize(actderiv)(a[l-1])
  #back propagation
  for i in range(l-3,-1,-1):
    d[i]=np.dot(w[i+1],d[i+1])*np.vectorize(actderiv)(z[i+1])
  #updating
  for i in range(0,l-1):
    for k in range (0,dim[i+1]):
      for j in range (0,dim[i]):
        w[i][j,k] = w[i][j,k] + mu*a[i][j]*d[i][k]
      b[i][k] = b[i][k] + mu*d[i][k]
```

Learned representation

input	hidden values	output
10000000	.88 .05 .08	10000000
01000000	.02 .11 .88	01000000
00100000	.01 .96 .27	00100000
00010000	.95 .97 .71	00010000
00001000	.03 .06 .02	00001000
00000100	.22 .98 .99	00000100
00000010	.82 .01 .98	00000010
00000001	.63 .94 .01	00000001

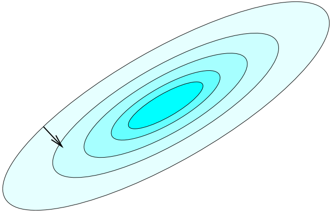


- the hidden layer learns a new representation of data
- the new features provides a compression of the input
- we cannot expect it to work well for any input (shannon theory)
- it may work well on available data, and “similar” inputs

Learning issues

- Why learning can be slow
- Vanishing gradient problem
- Optimization rules

Gradient descent can be slow



The gradient does not necessarily points to the direction of the local minimum

Issues with backpropagation

- $(BP4) \quad \frac{\partial E}{\partial w^{\ell}_{jk}} = a^{\ell-1}_k \delta^{\ell}_j$
  when activations are low, weights change (learn) slowly
- $(BP2) \quad \delta^l = (w^{\ell+1})^T \delta^{\ell+1} \odot \sigma'(z^{\ell})$
  For sigmoid activations, if $\sigma(z^{\ell}) \sim 0$ o $\sigma(z^{\ell}) \sim 1$, then $\sigma'(z^{\ell}) \sim 0$: in this case we say that the neuron is saturated.
  Similarly for BP1.

Summing up, a neuron learns slowly if either its input is low, or the output has saturated, i.e., it is either close to 1 or close to 0.

The vanishing gradient problem

$$(BP2) \quad \delta^l = (w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

For the first layers in the net, the gradient is the product of many factors of the form $\sigma'(z^\ell) \leq \frac{1}{4}$ for small values of z^ℓ (at least initially)

The first layers learn much more slowly than the last layers.

On the other side, if weights are small, the first layer loose most of the input information

Hence,last layers learn fast but on a highly deteriorate information.

A bit of history

The vanishing gradient problem blocked the progress on neural netwoks for almost 20 years (1990-2010).

It was first bypassed by network pre-training (e.g. with Boltzmann Machines), and later by the introduction on new activation functions, such as Rectified Linear Units (RELU), making pre-training obsolete.

Still, fine-tuning starting from good network weights (e.g. VGG) is a viable approach for many problems (transfer learning).