Deep-Learning : the basics

A. Allauzen

Université Paris-Sud / LIMSI-CNRS

9 mai 2017

Outline

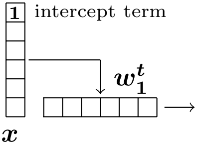
Outline

1. Neural Nets : Basics
   - Terminology
   - Training by back-propagation
2. Tools
3. Drop-out
4. Vanishing gradient

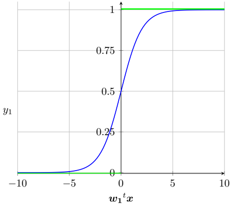
Outline

A choice of terminology

Logistic regression (binary classification)



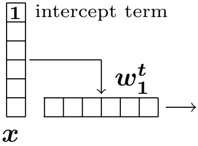
$$y_1 = f(\boldsymbol{w_1}^t \boldsymbol{x})$$



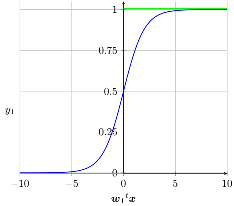
$$f(a = \boldsymbol{w_1}^t \boldsymbol{x}) = \frac{1}{1 + e^{-a}}$$

A choice of terminology

Logistic regression (binary classification)

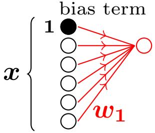


$\longrightarrow\ y_1 = f(\boldsymbol{w_1}^t \boldsymbol{x})$



$$f(a = \boldsymbol{w_1}^t \boldsymbol{x}) = \frac{1}{1 + e^{-a}}$$

A single artificial neuron

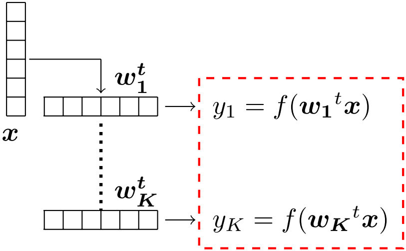


pre-activation : $a_1 = \boldsymbol{w_1}^t \boldsymbol{x}$

$y_1 = f(\boldsymbol{w_1}^t \boldsymbol{x})$, f is the activation function of the neuron

A choice of terminology - 2

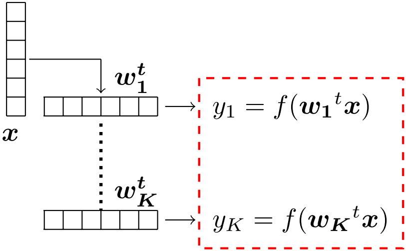
From binary classification to K classes (Maxent)



$$f(a_k = \boldsymbol{w_k}^t \boldsymbol{x}) = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}} = \frac{e^{a_k}}{Z(\boldsymbol{x})}$$

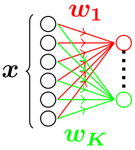
A choice of terminology - 2

From binary classification to K classes (Maxent)



$$f(a_k = \boldsymbol{w_k}^t\boldsymbol{x}) = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}} = \frac{e^{a_k}}{Z(\boldsymbol{x})}$$

A simple neural network

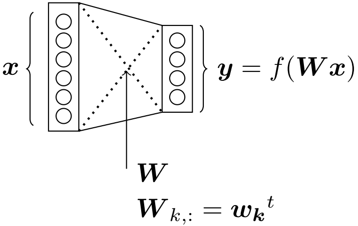


$y_1 = f(\boldsymbol{w_1^t}\boldsymbol{x})$

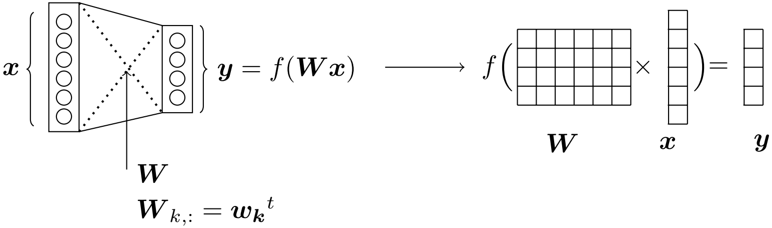
$y_K = f(\boldsymbol{w_K^t}\boldsymbol{x})$

- $\boldsymbol{x}$: *input layer*
- $\boldsymbol{y}$: *output layer*
- each y_k has its parameters $\boldsymbol{w}_k$
- f is the **softmax** function

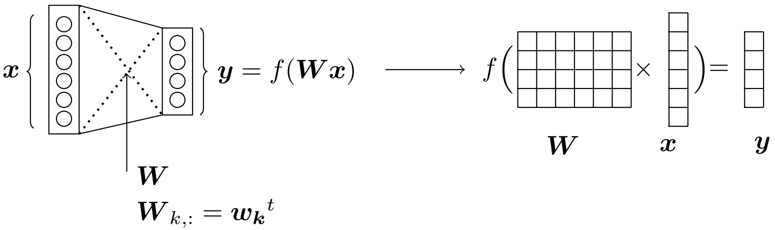
Two layers fully connected



Two layers fully connected



Two layers fully connected



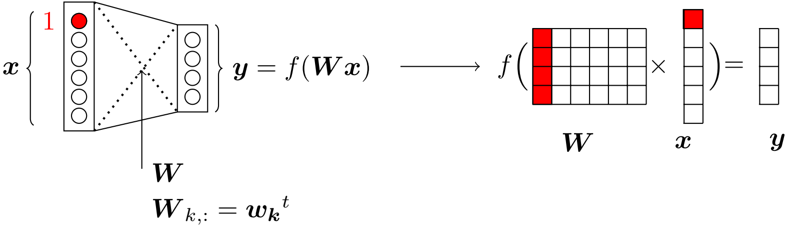
- f is usually a non-linear function
- f is a component wise function
- *e.g* the softmax function :

$$y_k = P(c = k|\boldsymbol{x}) = \frac{e^{\boldsymbol{w_k}^t \boldsymbol{x}}}{\sum_{k'} e^{\boldsymbol{w_{k'}}^t \boldsymbol{x}}} = \frac{e^{\boldsymbol{W}_{k,:}\boldsymbol{x}}}{\sum_{k'} e^{\boldsymbol{W}_{k',:}\boldsymbol{x}}}$$

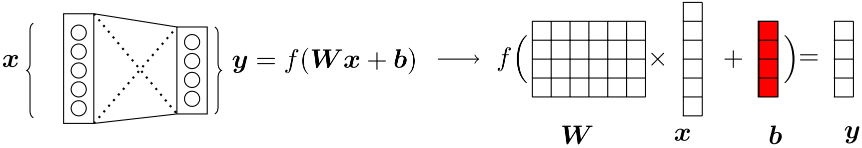
- tanh, sigmoid, relu, ...

Bias or not bias

Implicit Bias

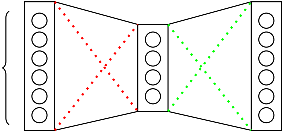


Explicit bias



With neural network : add a hidden layer

$\boldsymbol{x}$: raw input representation



$\boldsymbol{y} = f(\boldsymbol{W^{(2)}}\boldsymbol{h})$

$\boldsymbol{h} = f(\boldsymbol{W^{(1)}}\boldsymbol{x})$

the internal and tailored representation

Intuitions

- Learn an internal representation of the raw input
- Apply a non-linear transformation
- The input representation $\boldsymbol{x}$ is transformed/compressed in a new representation $\boldsymbol{h}$
- Adding more layers to obtain a more and more abstract representation

How do we learn the parameters ?

For a supervised single layer neural net

Just like a maxent model :

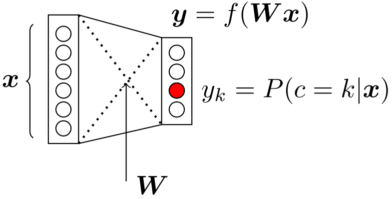
- Calculate the gradient of the objective function and use it to iteratively update the parameters.
- Conjugate gradient, L-BFGS, ...
- In practice : **Stochastic gradient descent (SGD)**

With one hidden layer

- The internal ("hidden") units make the function non-convex ... just like other models with hidden variables :
  - hidden CRFs (Quattoni et al.2007), ...
- But we can use the same ideas and techniques
- Just without guarantees ⇒ **backpropagation** (Rumelhart et al.1986)

Outline

Ex. 1 : A single layer network for classification



$\boldsymbol{\theta}$= the set of parameters, in this case :

$$\boldsymbol{\theta} = (\boldsymbol{W})$$

The log-loss (conditional log-likelihood)

Assume the dataset $\mathcal{D} = (\boldsymbol{x}_{(i)}, c_{(i)})_{i=1}^{N}$, $c_{(i)} \in \{1, 2, \ldots, C\}$

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)}) = \sum_{i=1}^{N} \Big(-\sum_{c=1}^{C} \mathbb{I}\big\{c = c_{(i)}\big\} \log(P(c|\boldsymbol{x}_{(i)})) \Big) \quad (1)$$

$$l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)}) = -\sum_{k=1}^{C} \mathbb{I}\big\{k = c_{(i)}\big\} \log(y_k) \quad (2)$$

Ex. 1 : optimization method

Stochastic Gradient Descent (Bottou2010)

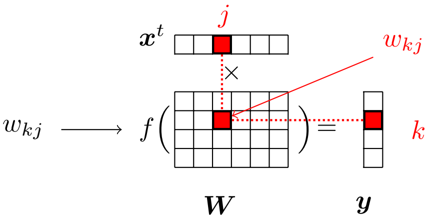
For ($t = 1$; until convergence ; $t++$) :

- Pick randomly a sample $(\boldsymbol{x}_{(i)}, c_{(i)})$
- Compute the gradient of the loss function w.r.t the parameters ($\nabla_{\boldsymbol{\theta}}$)
- Update the parameters : $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta_t \nabla_{\boldsymbol{\theta}}$

Questions

- convergence : what does it mean ?
- what do you mean by η_t ?
  - convergence if $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$
  - $\eta_t \propto t^{-1}$
  - and lot of variants like Adagrad (Duchi et al.2011), Down scheduling, ... see (LeCun et al.2012)

Ex. 1 : compute the gradient - 1



Inference chain :

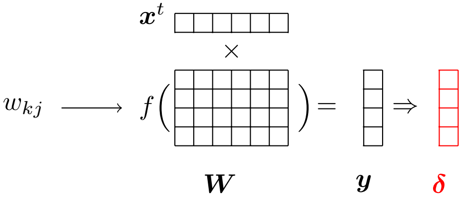
$$x_{(i)} \longrightarrow (a = Wx_{(i)}) \longrightarrow (y = f(a)) \longrightarrow l(\theta, x_{(i)}, c_{(i)})$$

The gradient for w_{kj}

$$\nabla_{w_{kj}} = \frac{\partial l(\theta, x_{(i)}, c_{(i)})}{\partial w_{kj}} = \frac{\partial l(\theta, x_{(i)}, c_{(i)})}{\partial y} \times \frac{\partial y}{\partial a} \times \frac{\partial a}{\partial w_{kj}}$$

$$= -(\mathbb{I}\{k = c_{(i)}\} - y_k) x_j = \delta_k x_j$$

Ex. 1 : compute the gradient - 2

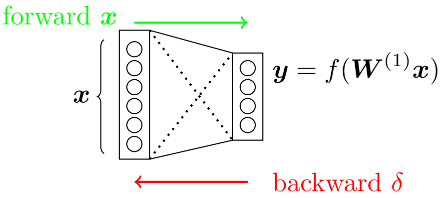


Generalization

$$\nabla_{\boldsymbol{W}} = \boldsymbol{\delta}\boldsymbol{x}^t$$
$$\delta_k = -(\mathbb{I}\{k = c_{(i)}\} - y_k)$$

with $\boldsymbol{\delta}$ the gradient at the pre-activation level.

Ex. 1 : Summary



Inference : a forward step

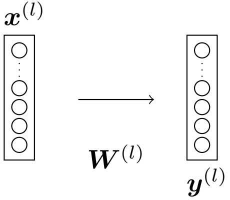
- matrice multiplication with the input $\boldsymbol{x}$
- Application of the activation function

One training step : forward and backward steps

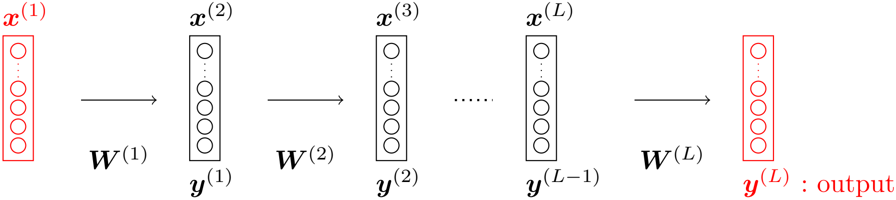
- Pick randomly a sample $(\boldsymbol{x}_{(i)}, c_{(i)})$
- Compute $\boldsymbol{\delta}$
- Update the parameters : $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta_t \boldsymbol{\delta}\boldsymbol{x}^t$

Notations for a multi-layer neural network (feed-forward)

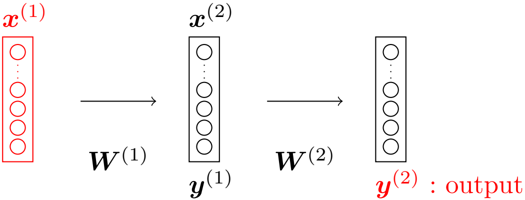
One layer, indexed by l



- $\boldsymbol{x}^{(l)}$: input of the layer l
- $\boldsymbol{y}^{(l)} = f^{(l)}(\boldsymbol{W}^{(l)}\ \boldsymbol{x}^{(l)})$
- stacking layers : $\boldsymbol{y}^{(l)} = \boldsymbol{x}^{(l+1)}$
- $\boldsymbol{x}^{(1)}$ = a data example



Ex. 2 : with one hidden layer



$$\theta = (W^{(1)}, W^{(2)})$$

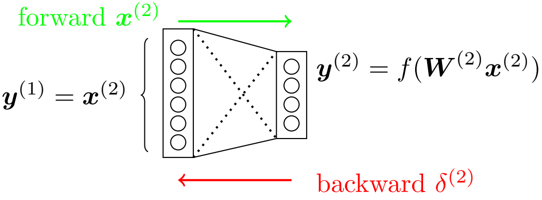
To learn, we need the gradients for :

- the output layer : $\nabla_{W^{(2)}}$
- the hidden layer : $\nabla_{W^{(1)}}$

Back-propagation of the loss gradient

For the output layer

As in the Ex. 1 :



$$
\begin{aligned}
\nabla_{\boldsymbol{W}^{(2)}} &= \boldsymbol{\delta}^{(2)}{\boldsymbol{x}^{(2)}}^t, \text{ with} \\
\delta_k^{(2)} &= -(\mathbb{I}\{k = c_{(i)}\} - y_k) \\
\boldsymbol{y} &\rightarrow \boldsymbol{y}^{(2)} \\
\boldsymbol{W} &\rightarrow \boldsymbol{W}^{(2)} \\
\boldsymbol{x} &\rightarrow \boldsymbol{x}^{(2)} = \boldsymbol{y}^{(1)}
\end{aligned}
$$

Back-propagation of the loss gradient

For the hidden layer - 1

The goal : compute $\boldsymbol{\delta}^{(1)}$

Inference (/forward) chain from $\boldsymbol{a}^{(1)}$ to the output :

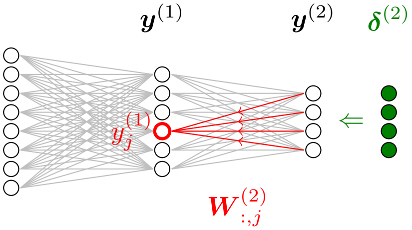
$$\boldsymbol{y}^{(1)} = f^{(1)}(\boldsymbol{a}^{(1)}) \rightarrow \left(\boldsymbol{a}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{y}^{(1)}\right) \rightarrow \left(\boldsymbol{y}^{(2)} = f^{(2)}(\boldsymbol{a}^{(2)})\right) \rightarrow l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})$$

Backward / Back-propagation :

$$\delta_j^{(1)} = \nabla_{a_j^{(1)}} = \frac{\partial l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})}{\partial a_j^{(1)}} = \frac{\partial l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})}{\partial \boldsymbol{y}^{(2)}} \times \frac{\partial \boldsymbol{y}^{(2)}}{\partial \boldsymbol{a}^{(2)}} \times \frac{\partial \boldsymbol{a}^{(2)}}{\partial y_j^{(1)}} \times \frac{\partial y_j^{(1)}}{\partial a_j^{(1)}}$$

Back-propagation of the loss gradient

For the hidden layer - 2



Backward / Back-propagation :

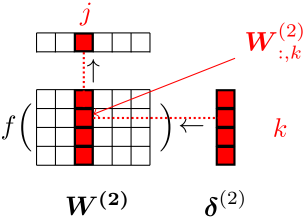
$$\delta_j^{(1)} = \nabla_{a_j^{(1)}} = \frac{\partial l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})}{\partial a_j^{(1)}} = \frac{\partial l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})}{\partial \boldsymbol{y}^{(2)}} \times \frac{\partial \boldsymbol{y}^{(2)}}{\partial \boldsymbol{a}^{(2)}} \times \frac{\partial \boldsymbol{a}^{(2)}}{\partial y_j^{(1)}} \times \frac{\partial y_j^{(1)}}{\partial a_j^{(1)}}$$

$$= f'^{(1)}(a_j)\left(\boldsymbol{W}_{:,j}^{(2)\,t}\boldsymbol{\delta}^{(2)}\right)$$

Back-propagation of the loss gradient

For the hidden layer - 3

the back-propagated signal $\nabla_{\boldsymbol{y}^{(1)}} \leftarrow$

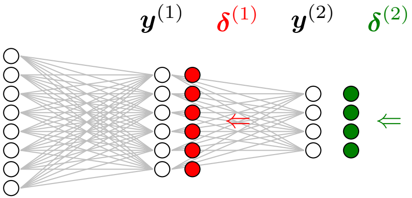


$$\nabla_{\boldsymbol{y}^{(1)}} = \boldsymbol{W}^{(2)^t} \boldsymbol{\delta}^{(2)}, \text{ then}$$

$$\boldsymbol{\delta}^{(1)} = \nabla_{\boldsymbol{a}^{(1)}} = f^{(1)'}(\boldsymbol{a}^{(1)}) \circ \left(\boldsymbol{W}^{(2)^t} \boldsymbol{\delta}^{(2)}\right)$$

Back-propagation of the loss gradient

For the hidden layer - 4



As for the output layer, the gradient is :

$$\nabla_{\boldsymbol{W}^{(1)}} = \boldsymbol{\delta}^{(1)} {\boldsymbol{x}^{(1)}}^t, \text{ with}$$

$$\delta_j^{(1)} = \nabla_{a_j^{(1)}}$$

$$\boldsymbol{\delta}^{(1)} = {f'}^{(1)}(\boldsymbol{a}^{(1)}) \circ ({\boldsymbol{W}^{(2)}}^t \boldsymbol{\delta}^{(2)})$$

The term $({\boldsymbol{W}^{(2)}}^t \boldsymbol{\delta}^{(2)})$ comes from the upper layer.

Back-propagation : generalization

For a hidden layer l :

- The gradient at the pre-activation level :

$$\boldsymbol{\delta}^{(l)} = {f'}^{(l)}(\boldsymbol{a}^{(l)}) \circ \left(\boldsymbol{W}^{(l+1)^t}\boldsymbol{\delta}^{(l+1)}\right)$$

- The update is as follows :

$$\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \eta_t \boldsymbol{\delta}^{(l)} {\boldsymbol{x}^{(l)}}^t$$

The layer should keep :

- $\boldsymbol{W}^{(l)}$: the parameters
- $f^{(l)}$: its activation function
- $\boldsymbol{x}^{(l)}$: its input
- $\boldsymbol{a}^{(l)}$: its pre-activation associated to the input
- $\boldsymbol{\delta}^{(l)}$: for the update and the back-propagation to the layer $l-1$

Back-propagation : one training step

Pick a training example : $\boldsymbol{x}^{(1)} = \boldsymbol{x}_{(i)}$

Forward pass

For $l = 1$ to $(L-1)$

- Compute $\boldsymbol{y}^{(l)} = f^{(l)}(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)})$
- $\boldsymbol{x}^{(l+1)} = \boldsymbol{y}^{(l)}$

$\boldsymbol{y}^{(L)} = f^{(L)}(\boldsymbol{W}^{(L)}\boldsymbol{x}^{(L)})$

Backward pass

Init : $\boldsymbol{\delta}^{(L)} = \nabla_{\boldsymbol{a}^{(L)}}$
For $l = L$ to 2 // all hidden units

- $\boldsymbol{\delta}^{(l-1)} = f'^{(l-1)}(\boldsymbol{a}^{(l-1)}) \circ \left(\boldsymbol{W}^{(l)^t}\boldsymbol{\delta}^{(l)}\right)$
- $\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \eta_t \boldsymbol{\delta}^{(l)}\boldsymbol{x}^{(l)^t}$

$\boldsymbol{W}^{(1)} = \boldsymbol{W}^{(1)} - \eta_t \boldsymbol{\delta}^{(1)}\boldsymbol{x}^{(1)^t}$

Initialization recipes

A difficult question with several empirical answers.

One standard trick

$$\boldsymbol{W} \sim \mathcal{N}(0, \frac{1}{\sqrt{n_{in}}})$$

with n_{in} is the number of inputs

A more recent one

$$\boldsymbol{W} \sim \mathcal{U}\Big[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\Big]$$

with n_{in} is the number of inputs

Outline

Some useful libraries

Theano

Written in python by the LISA (Y. Bengio and I. Goodfellow)

TensorFlow

The Google library with python API

Keras

A high-level API, in Python, running on top of either TensorFlow or Theano.

Torch

The Facebook library with Lua python API

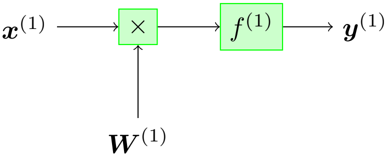
- CPU/GPU
- Automatic differentiation based on computational graph

Computation graph

A convenient way to represent a complex mathematical expressions :

- each node is an operation or a variable
- an operation has some inputs / outputs made of variables

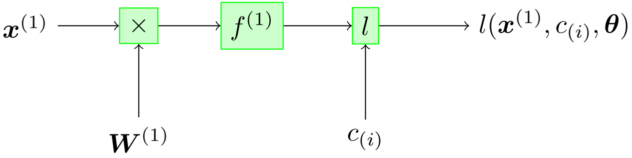
Example 1 : A single layer network



- Setting $\boldsymbol{x}^{(1)}$ and $\boldsymbol{W}^{(1)}$
- Forward pass $\rightarrow \boldsymbol{y}^{(1)}$

$$\boldsymbol{y}^{(1)} = f^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)})$$

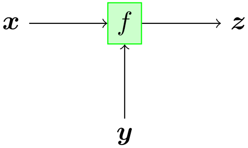
Training computation graph



- A variable node encodes the label
- To compute the output for a given input
  $\rightarrow$ forward pass
- To compute the gradient of the loss *wrt* the parameters ($\boldsymbol{W}^{(1)}$)
  $\rightarrow$ backward pass

A function node

Forward pass

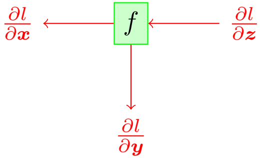


This node implements :

$$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y})$$

A function node - 2

Backward pass



A function node knows :

- the "local gradients" computation

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}, \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}$$

- how to return the gradient to the inputs :

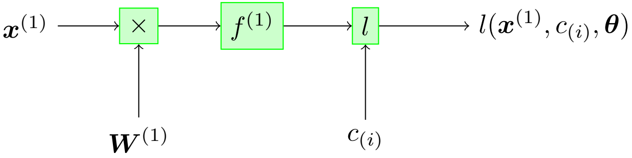
$$(\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}), (\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}})$$

Summary of a function node

$f:$

$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$	# store the values
$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y})$	# forward
$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \rightarrow \frac{\partial f}{\partial \boldsymbol{x}}$	# local gradients
$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \rightarrow \frac{\partial f}{\partial \boldsymbol{y}}$	
$(\frac{\partial l}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}),(\frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}})$	# backward

Example of a single layer network



Forward

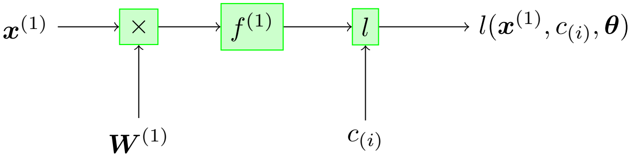
For each function node in topological order

- forward propagation

Which means :

1. $\boldsymbol{a}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)}$
2. $\boldsymbol{y}^{(1)} = f^{(1)}(\boldsymbol{a}^{(1)})$
3. $l(\boldsymbol{y}^{(1)}, c_{(i)})$

Example of a single layer network



Forward

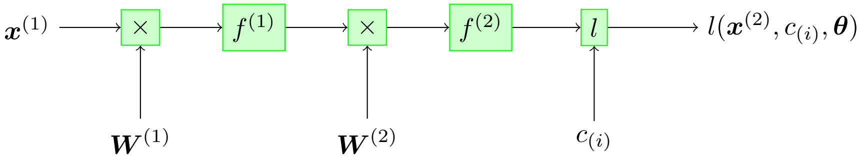
For each function node in reversed topological order

- backward propagation

Which means :

1. $\nabla_{\boldsymbol{y}^{(1)}}$
2. $\nabla_{\boldsymbol{a}^{(1)}}$
3. $\nabla_{\boldsymbol{W}^{(1)}}$

Example of a two layers network



- The algorithms remain the same,
- even for more complex architectures
- Generalization by coding the function node

Example in Theano - 1

```
import theano
import theano.tensor as T
# Define the input
x = T.fvector('x')
# The parameters of the hidden layer
H = 100 # hidden layer size
n_in=im.shape[0] # dimension of inputs
n_out=H
Wi = uniform(shape=[n_out,n_in], name="Wi")
bi=shared0s([n_out],name="bi")
# parameters for the output layer
n_in=H
n_out=NLABELS
Wo = uniform(shape=[n_out,n_in], name="Wo")
bo=shared0s([n_out],name="bo")
```

Example in Theano - 2

```
# define the hidden layer
h = T.nnet.relu(T.dot(Wi,x)+bi)
# output layer and related variables:
p_y_given_x = T.nnet.softmax(T.dot(Wo,h)+bo)
y_pred = T.argmax(p_y_given_x)
# Compute the cost function
ygold = T.iscalar('gold_target')
cost = -T.log(p_y_given_x[0][ygold])
# 1/ Store all the learnt parameters:
params = [Wi, bi, Wo, bo]
# 2/ Get the gradients of everyone
gradients = T.grad(cost,params)
# 3/ Collect the updates
upds = [(p, p - (learning_rate * g))
            for p, g in zip(params, gradients)]
```

Example in Tensorflow - 1

```
import tensorflow as tf

# x isn't a specific value. It's a placeholder,
# a value that we'll input to run a computation.
x = tf.placeholder(tf.float32, [None, 784])

# Define the parameters as variables
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# the prediction variable
y = tf.nn.softmax(tf.matmul(x, W) + b)

# the gold standard (a placeholder)
y_ = tf.placeholder(tf.float32, [None, 10])
```

Example in Tensorflow - 2

```
# the loss function
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduct

# SGD
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_e

# Init. of all the variables
# This defines the operations but does not run it yet.
init = tf.initialize_all_variables()

# open a session
sess = tf.Session()
sess.run(init)

# Training
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Outline

Regularization l^2 or gaussian prior or weight decay

The basic way :

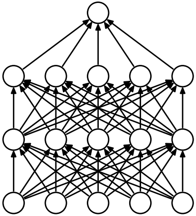
$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)}) + \frac{\lambda}{2}||\boldsymbol{\theta}||^2$$

- The second term is the regularization term.
- Each parameter has a gaussian prior : $\mathcal{N}(0, 1/\lambda)$.
- λ is a hyperparameter.
- The update has the form :

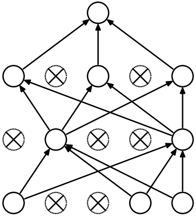
$$\boldsymbol{\theta} = (1 + \eta_t \lambda)\boldsymbol{\theta} - \eta_t \nabla_{\boldsymbol{\theta}}$$

Dropout

A new regularization scheme (Srivastava and Salakhutdinov2014)



(a) Standard Neural Net



(b) After applying dropout.

- For each training example : randomly turn-off the neurons of hidden units (with $p = 0.5$)
- At test time, use each neuron scaled down by p

- Dropout serves to separate effects from strongly correlated features and
- prevents co-adaptation between units
- It can be seen as averaging different models that share parameters.
- It acts as a powerful regularization scheme.

Dropout - implementation

The layer should keep :

- $\boldsymbol{W}^{(l)}$: the parameters
- $f^{(l)}$: its activation function
- $\boldsymbol{x}^{(l)}$: its input
- $\boldsymbol{a}^{(l)}$: its pre-activation associated to the input
- $\boldsymbol{\delta}^{(l)}$: for the update and the back-propagation to the layer $l-1$
- $\boldsymbol{m}^{(l)}$: the dropout mask, to be applied on $\boldsymbol{x}^{(l)}$

Forward pass

For $l = 1$ to $(L-1)$

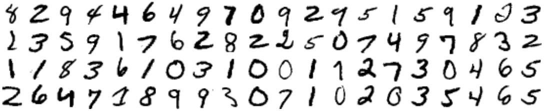
- Compute $\boldsymbol{y}^{(l)} = f^{(l)}(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)})$
- $\boldsymbol{x}^{(l+1)} = \boldsymbol{y}^{(l)} = \boldsymbol{y}^{(l)} \circ \boldsymbol{m}^{(l)}$

$\boldsymbol{y}^{(L)} = f^{(L)}(\boldsymbol{W}^{(L)}\boldsymbol{x}^{(L)})$

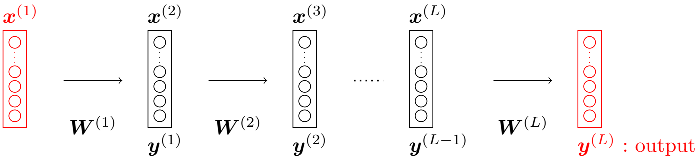
Outline

Experimental observations (MNIST task) - 1

The MNIST database



Comparison of different depth for feed-forward architecture



- Hidden layers have a sigmoid activation function.
- The output layer is a softmax.

Experimental observations (MNIST task) - 2

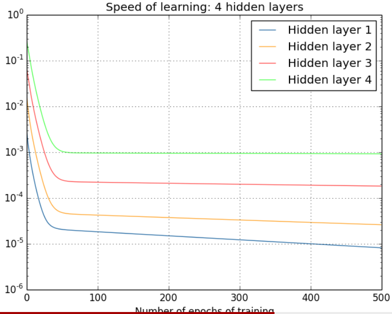
Varying the depth

- Without hidden layer : ≈ 88% accuracy
- 1 hidden layer (30) : ≈ 96.5% accuracy
- 2 hidden layer (30) : ≈ 96.9% accuracy
- 3 hidden layer (30) : ≈ 96.5% accuracy
- 4 hidden layer (30) : ≈ 96.5% accuracy

Experimental observations (MNIST task) - 2

Varying the depth

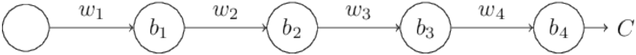
- Without hidden layer : ≈ 88% accuracy
- 1 hidden layer (30) : ≈ 96.5% accuracy
- 2 hidden layer (30) : ≈ 96.9% accuracy
- 3 hidden layer (30) : ≈ 96.5% accuracy
- 4 hidden layer (30) : ≈ 96.5% accuracy



(From `http://neuralnetworksanddeeplearning.com/chap5.html`)

Intuitive explanation

Let consider the simplest deep neural network, with just a single neuron in each layer.



w_i, b_i are resp. the weight and bias of neuron i and C some cost function.

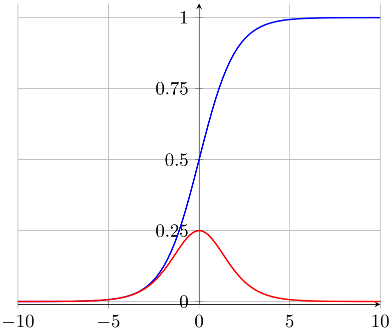
Compute the gradient of C *w.r.t* the bias b_1

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial y_4} \times \frac{\partial y_4}{\partial a_4} \times \frac{\partial a_4}{\partial y_3} \times \frac{\partial y_3}{\partial a_3} \times \frac{\partial a_3}{\partial y_2} \times \frac{\partial y_2}{\partial a_2} \times \frac{\partial a_2}{\partial y_1} \times \frac{\partial y_1}{\partial a_1} \times \frac{\partial a_1}{\partial b_1} \quad (3)$$

$$= \frac{\partial C}{\partial y_4} \times \sigma'(a_4) \times w_4 \times \sigma'(a_3) \times w_3 \times \sigma'(a_2) \times w_2 \times \sigma'(a_1) \quad (4)$$

Intuitive explanation - 2

The derivative of the activation function : σ'



$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

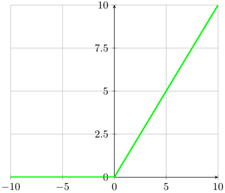
But weights are initialize around 0.

The different layers in our deep network are learning at vastly different speeds :

- when later layers in the network are learning well,
- early layers often get stuck during training, learning almost nothing at all.

Solutions

Change the activation function (Rectified Linear Unit or ReLU)



- Avoid the vanishing gradient
- Some units can "die"

See (Glorot et al.2011) for more details

Do pre-training when it is possible

See (Hinton et al.2006; Bengio et al.2007) :

when you cannot really escape from the initial (random) point, find a good starting point.

More details

See (Hochreiter et al.2001; Glorot and Bengio2010; LeCun et al.2012)

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle.
2007.
Greedy layer-wise training of deep networks.
In B. Schölkopf, J.C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 153–160. MIT Press.

Léon Bottou.
2010.
Large-scale machine learning with stochastic gradient descent.
In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of COMPSTAT'2010*, pages 177–186. Physica-Verlag HD.

John Duchi, Elad Hazan, and Yoram Singer.
2011.
Adaptive subgradient methods for online learning and stochastic optimization.
J. Mach. Learn. Res., 12 :2121–2159, July.

Xavier Glorot and Yoshua Bengio.
2010.
Understanding the difficulty of training deep feedforward neural networks.
In *JMLR W&CP : Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 249–256, May.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio.

2011.

Deep sparse rectifier neural networks.

In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings.

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh.

2006.

A fast learning algorithm for deep belief nets.

Neural Computation, 18(7) :1527–1554, JUL.

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber.

2001.

Gradient flow in recurrent nets : the difficulty of learning long-term dependencies.

In Kremer and Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press.

Yann LeCun, Léon Bottou, Genevieve Orr, and Klaus-Robert Müller.

2012.

Efficient backprop.

In Grégoire Montavon, GenevièveB. Orr, and Klaus-Robert Müller, editors, *Neural Networks : Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 9–48. Springer Berlin Heidelberg.

Ariadna Quattoni, Sybor Wang, Louis-Philippe Morency, Michael Collins, and Trevor Darrell.
2007.
Hidden conditional random fields.
IEEE Trans. Pattern Anal. Mach. Intell., 29(10) :1848–1852, October.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams.
1986.
Learning representations by back-propagating errors.
Nature, 323(6088) :533–536, 10.

Nitish Srivastava and Ruslan Salakhutdinov.
2014.
Multimodal learning with deep boltzmann machines.
Journal of Machine Learning Research, 15 :2949–2980.