Neural Net: Deep Learning basics - 2

A. Allauzen

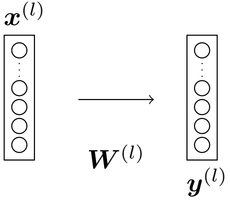
Université Paris-Sud / LIMSI-CNRS

Outline

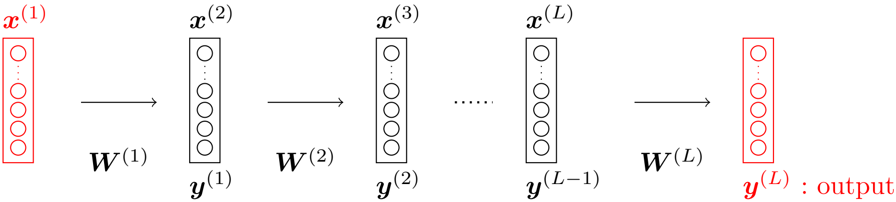
Outline

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



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

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

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

Outline

Some useful libraries

Theano

Written in python by the LISA (Y. Bengio and I. Goodfellow), low-level API.

TensorFlow

The Google library with python API

(py)Torch

The Facebook library with Lua/python API

Keras

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

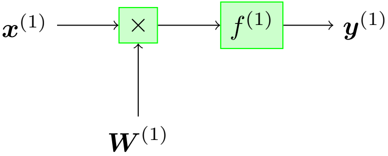
- CPU/GPU
- Automatic differentiation based on computational graph

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



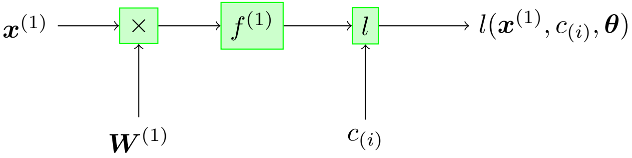
- Setting $\boldsymbol{x}^{(1)}$ and $\boldsymbol{W}^{(1)}$
- Forward pass $\rightarrow \boldsymbol{y}^{(1)}$

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

Remark

Some toolkit refers to variable as node, and function as edge.

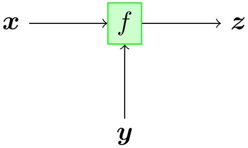
Computation graph for training



- 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

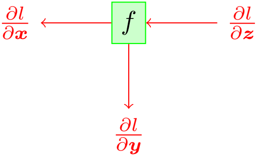


This node implements :

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

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 :

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

Summary of a function node

f :

$$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \qquad \text{\# store the values}$$

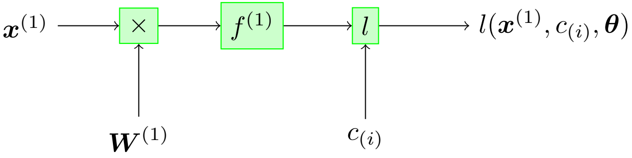
$$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y}) \qquad \text{\# forward}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \rightarrow \frac{\partial f}{\partial \boldsymbol{x}} \qquad \text{\# local gradients}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \rightarrow \frac{\partial f}{\partial \boldsymbol{y}}$$

$$\left(\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right), \left(\frac{\partial l}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}\right) \qquad \text{\# backward}$$

Example of a single layer network



Forward

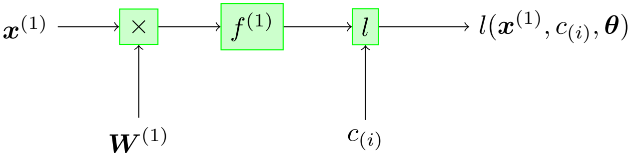
For each function node in topological order

- 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



Backward

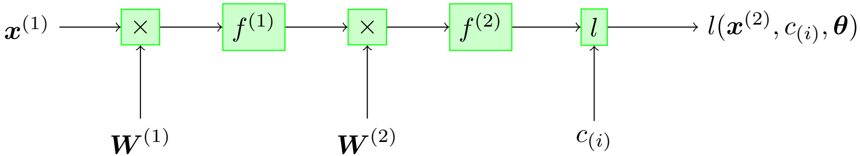
For each function node in reversed topological order

- 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 your own function node or by
- Wrapping a layer in a module

pytorch in three keywords

torch.Tensor

Similar to numpy's *ndarrays*, but can be used on a GPU to accelerate computing.

Variable

A thin wrapper around a Tensor object holding :

- that also holds the gradient w.r.t. to it,
- a reference to a function that created it.

Function

Records operation history and defines formulas for differentiating ops. Every operation performed on Variables :

- creates a new function object, that performs the computation,
- records what happened.
- The history is retained in the form of a DAG of functions, with edges describing data dependencies (input ¡- output).

And more

Module

Modules $\sim$ neural network layers.

- A Module receives input Variables and computes output Variables,
- may also hold internal state such as Variables containing learnable parameters.

Sequential

Feed-forward container

Optimizer

Take care of the gradient descent

An example in pytorch

```
import torch from torch.autograd import Variable

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H,  D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs,
# and wrap them in Variables.
x = Variable(torch.randn(N, D_in))
y =  Variable(torch.randn(N, D_out), requires_grad=False)

# Use the nn package to define our model
# as a sequence of layers.
model = torch.nn.Sequential(
   torch.nn.Linear(D_in, H),
   torch.nn.ReLU(),
   torch.nn.Linear(H, D_out),
   )
```

An example un pytorch - 2

```
loss_fn = torch.nn.MSELoss(size_average=False)
# Optimizer will update  the weights of the model.
lr0 = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
lr=lr0)
for t in  range(10):
   # Forward pass: compute predicted y by passing x.
   y_pred = model(x)
   # Compute and print loss.
   loss = loss_fn(y_pred, y)
   print(t, loss.data[0])
   # Optim in two steps
   optimizer.zero_grad()
   # Backward pass: compute gradient of the loss wrt parameters
   loss.backward()
   # Calling the step function on an Optimizer makes an update
   optimizer.step()
```

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

```python
# 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)]
```

Outline

Objective function for regression

- Assume $\mathcal{D} = (\boldsymbol{x}_{(i)}, t_i)$, with $\boldsymbol{x}_{(i)} \in \mathbb{R}^K$.
- A neural network with parameters $\boldsymbol{\theta}$ output a real value $y_i(\boldsymbol{\theta}, \boldsymbol{x}_{(i)})$.
- The goal is to learn to approximate t_i, with a feed-forward network.
- Consider target values are corrupted by a gaussian noise.
- To model this uncertainty the NNet is assumed to predict the mean value of t instead of the targeted value :

$$P(t_i | \boldsymbol{x}_{(i)}, \boldsymbol{\theta}, \beta) = \mathcal{N}(t_i | y(\boldsymbol{x}_{(i)}, \boldsymbol{\theta}), \beta^{-1}),$$

  $i.e$ the predicted value has a normal distribution centered in $y(\boldsymbol{x}_{(i)}, \boldsymbol{\theta})$ with a variance of β^{-1} (concentration).

Questions : the loss function

1. Illustrate this setup in the case of a scalar input.
2. Write the log-likelihood of one training example.
3. To learn $\boldsymbol{\theta}$, we maximize the log-likelihood, Write the loss function.
4. Give an interpretation of the loss function when optimizing it w.r.t $\boldsymbol{\theta}$.
5. Sketch how you could learn $\boldsymbol{\theta}$. The results is known as $\boldsymbol{\theta}_{ML}$.
6. Knowing $\boldsymbol{\theta}_{ML}$, provide an estimate of β.
7. With a single layer NNet, which kind of approximate can we get ? Can you explain β ?

Questions : prior on parameters

Assume now a prior distribution over $\boldsymbol{\theta}$. For simplicity, we assume a gaussian prior, centered at the origin, with a covariance matrix such as $\sum = \alpha^{-1}\boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix.

1. Write explicitely this prior distribution over $\boldsymbol{\theta}$: $P(\boldsymbol{\theta}|\alpha)$
2. Write the posterior distribution of $\boldsymbol{\theta}$, $P(\boldsymbol{\theta}|\mathcal{D}, \alpha, \beta)$.
3. Provide an interpretation.

Outline

Regularization l^2 or gaussian prior or weight decay

The basic way :

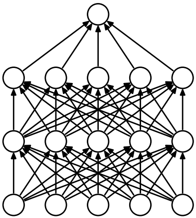
$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)}) + \frac{\lambda}{2}||\boldsymbol{\theta}||^2$$

- 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 :

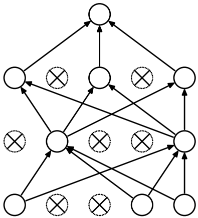
$$\boldsymbol{\theta} = (1 + \eta_t\lambda)\boldsymbol{\theta} - \eta_t \nabla_{\boldsymbol{\theta}}$$

Dropout

A new regularization scheme (?)



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

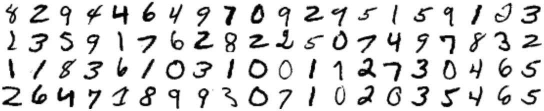
$\boldsymbol{y}^{(L)} = f^{(L)}(\boldsymbol{W}^{(L)}\boldsymbol{x}^{(L)})$

Outline

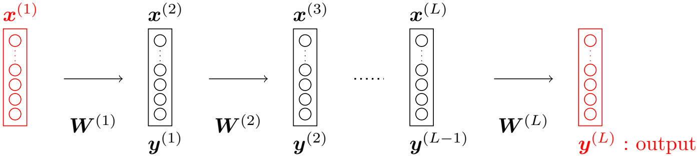
1. Reminder
2. Tools for deep-learning
3. Regression break (exercise)
4. Regularization and Dropout
5. Vanishing gradient issue

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

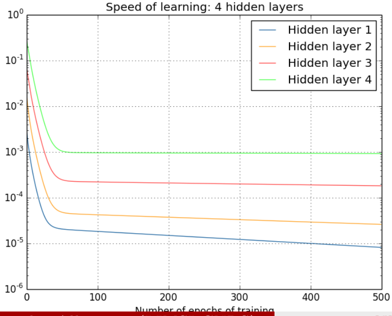
Varying the depth

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

Experimental observations (MNIST task) - 2

Varying the depth

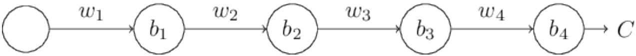
- Without hidden layer : $\approx 88\%$ accuracy
- 1 hidden layer (30) : $\approx 96.5\%$ accuracy
- 2 hidden layer (30) : $\approx 96.9\%$ accuracy
- 3 hidden layer (30) : $\approx 96.5\%$ accuracy
- 4 hidden layer (30) : $\approx 96.5\%$ accuracy



(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.

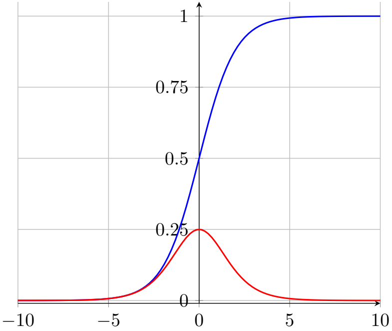
Compute the gradient of C *w.r.t* the bias b_1

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial y_4} \times \frac{\partial y_4}{\partial a_4} \times \frac{\partial a_4}{\partial y_3} \times \frac{\partial y_3}{\partial a_3} \times \frac{\partial a_3}{\partial y_2} \times \frac{\partial y_2}{\partial a_2} \times \frac{\partial a_2}{\partial y_1} \times \frac{\partial y_1}{\partial a_1} \times \frac{\partial a_1}{\partial b_1} \tag{1}$$

$$= \frac{\partial C}{\partial y_4} \times \sigma'(a_4) \times w_4 \times \sigma'(a_3) \times w_3 \times \sigma'(a_2) \times w_2 \times \sigma'(a_1) \tag{2}$$

Intuitive explanation - 2

The derivative of the activation function : σ'



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

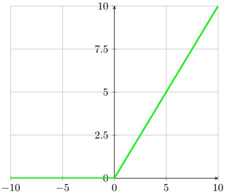
But weights are initialize around 0.

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 (?) for more details

Do pre-training when it is possible

See (?; ?) :

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

More details

See (?; ?; ?)