Introduction to Deep Learning

Tools and deep learning of NNet

Alexandre Allauzen

ESPCI PARIS | PSL

Dauphine UNIVERSITÉ PARIS | PSL

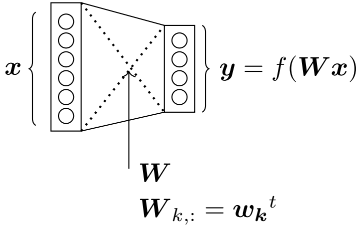
MILES Machine Intelligence and Learning Systems

27/01/20

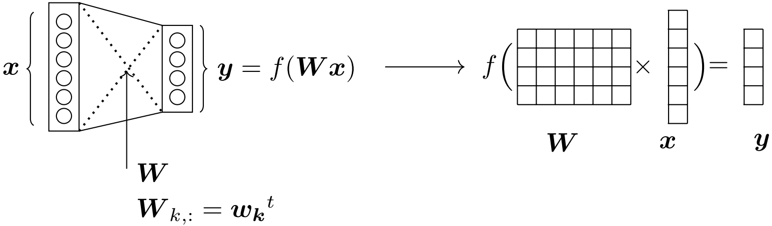
Outline

Outline

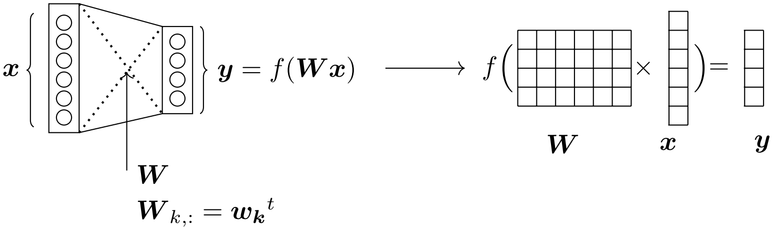
Two layers fully connected: a linear separation



Two layers fully connected: a linear separation



Two layers fully connected: a linear separation



Activation f:

- f is usually a non-linear function
- f is a component wise function
- tanh, sigmoid, relu, ...

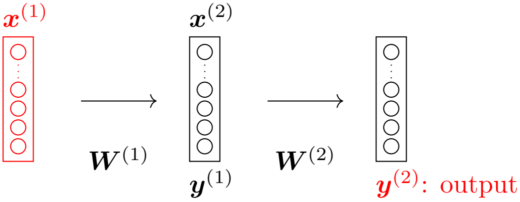
e.g the softmax function:

Dimensions:

- $\boldsymbol{x} : D \times 1$
- $\boldsymbol{W} : C \times D$
- $\boldsymbol{y} : (C \times \cancel{D) \times (D} \times 1) = C \times 1$

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

From linear to non-linear case



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

Trained by back-propagation of the gradient

Universal approximation theorem

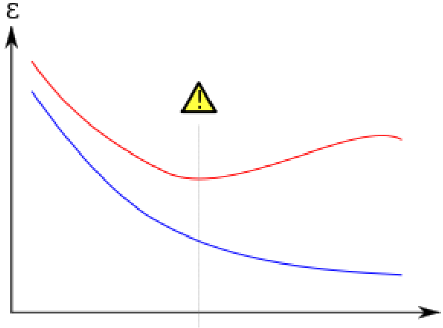
> *a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of* $\mathbb{R}^n$*, under mild assumptions on the activation function. (...)*

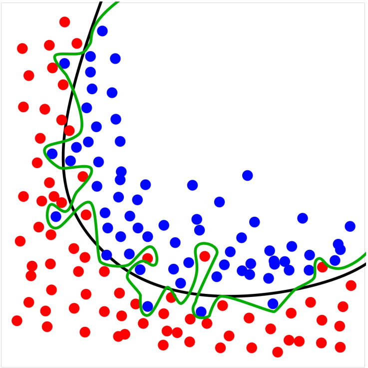
(Cybenko1989)

However, it does not touch upon the algorithmic learnability of those parameters.

Overfitting

The danger of the over-parametrization

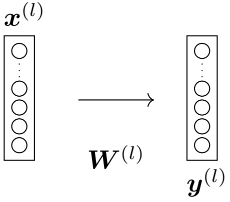




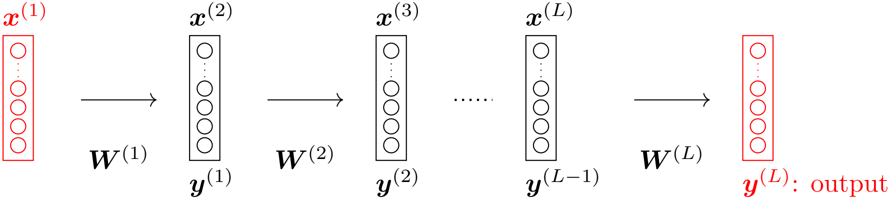
Source: Wikipedia

Multi-layer neural network (feed-forward)

One layer, indexed by l



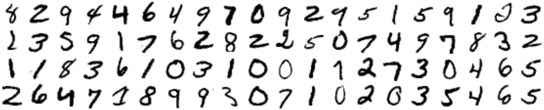
- $\boldsymbol{x}^{(l)}$: input of the layer l
- $\boldsymbol{y}^{(l)} = f^{(l)}(\boldsymbol{W}^{(l)}\ \boldsymbol{x}^{(l)})$
- stacking layers: $\boldsymbol{y}^{(l)} = \boldsymbol{x}^{(l+1)}$
- $\boldsymbol{x}^{(1)}$ = a data example



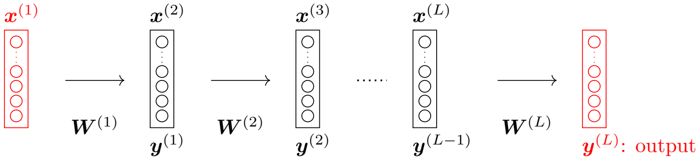
Outline

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

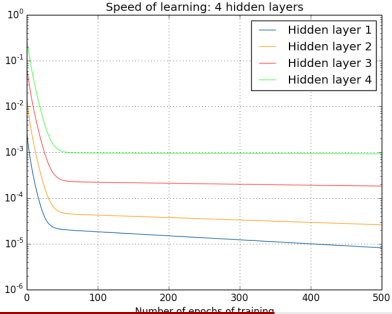
Varying the depth

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

Experimental observations (MNIST task) - 2

Varying the depth

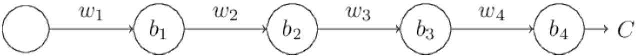
- Without hidden layer: $\approx 88\%$ accuracy
- 1 hidden layer (30): $\approx 96.5\%$ accuracy
- 2 hidden layers (30): $\approx 96.9\%$ accuracy
- 3 hidden layers (30): $\approx 96.5\%$ accuracy
- 4 hidden layers (30): $\approx 96.5\%$ accuracy



(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.

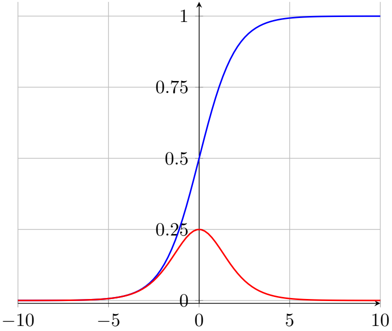
Compute the gradient of C *w.r.t* the bias b_1

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

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

Intuitive explanation - 2

The derivative of the activation function: σ'



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

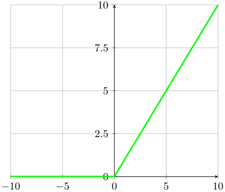
But weights are initialize around 0.

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.

A first Solution

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

Variants

- Leaky ReLU (Maas et al.2013)
- Soft-plus $log(1 + e^x)$

And many more, see `https://pytorch.org/docs/stable/nn.html`

More details

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

A question

Why adding a layer can lower the performance ?

A question

Why adding a layer can lower the performance ?

- Overfitting ? and what about the identity
- Vanishing gradient ? and with the *Relu* ?

A question

Why adding a layer can lower the performance ?

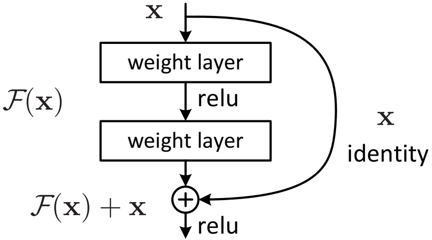
- Overfitting ? and what about the identity
- Vanishing gradient ? and with the *Relu* ?

Residual block

From (He et al.2015)

- Add a skip connection
- The model learn the "residual"

$$\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}) = \boldsymbol{x} + \mathcal{R}(\boldsymbol{x})$$



A simple version of highway networks (Srivastava et al.2015)

Residual block

Forward

$$\begin{aligned} \boldsymbol{y} &= \mathcal{F}(\boldsymbol{x}) = \boldsymbol{x} + \mathcal{R}(\boldsymbol{x}), \text{ or} \\ \boldsymbol{y} &= \boldsymbol{W}_s\boldsymbol{x} + \mathcal{R}(\boldsymbol{x}), \text{ to adapt the dimension} \end{aligned}$$

Backward

Assume a residual block for the layer l in the network. Training requires:

- $\frac{\partial l}{\partial \boldsymbol{W}^{(l)}}$ for the update of the layer
- $\frac{\partial l}{\partial \boldsymbol{x}^{(l)}}$ for the backpropagation

$$\begin{aligned} \frac{\partial l}{\partial \boldsymbol{x}^{(l)}} &= \frac{\partial l}{\partial \boldsymbol{y}^{(l)}} \times \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{x}^{(l)}} \\ &= \frac{\partial l}{\partial \boldsymbol{y}^{(l)}} \times (1 + \frac{\partial \mathcal{R}(\boldsymbol{x}^{(l)})}{\partial \boldsymbol{x}^{(l)}}) \end{aligned}$$

Outline

Regularization l^2 or gaussian prior or weight decay

The basic way:

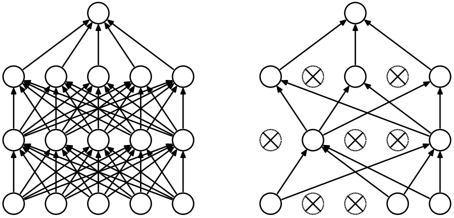
$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \sum_{i=1}^{N} l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)}) + \frac{\lambda}{2} ||\boldsymbol{\theta}||^2$$

- 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

Some useful libraries

Theano

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

TensorFlow and Keras

The Google library with python API + high level API

pyTorch

The Facebook library with python API

And others

Caffe, MXNet, CNTK, Chainer, ...

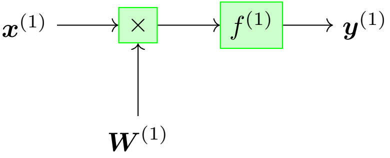
- CPU/GPU
- Automatic differentiation based on computational graph

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.

Building a computation graph

Variables (eq. Tensors) flow through a D.A.G

A variable is a *Tensor*

A *Tensor* stores:

- the values (as *numpy.array*);
- a link to its creator;
- (optionally) the gradient values (as a *numpy.array* of same size);

The creator is a function or tensor operation

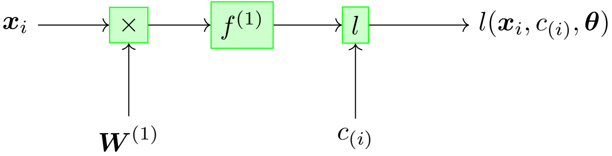
Function

A tensor operation that takes:

- several (or zero) input tensors,
- and output one new tensor as a result.

Ex: the logistic regression model

The computation graph



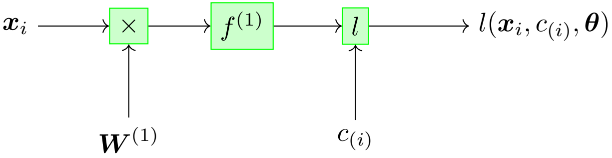
- $f^{(1)}$ is the sigmoid (σ) function
- the loss is the binary log-loss (a.k.a binary cross entropy):

$$l(\boldsymbol{x}, c, \boldsymbol{\theta} = \boldsymbol{W}) = c_{(i)} \log y + (1 - c_{(i)}) \log(1 - y),$$

- with y, a scalar, the output of $f^{(1)}$.

Ex: the logistic regression model

In numpy *vs* pytorch

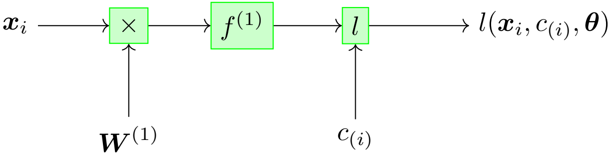



```python
import numpy as np
# explicite bias
W = np.random.randn(1,D+1)
# inference
# x the input:  np.array
a = W@x
y = 1/(1+np.exp(-a))
# a and y are np.array
# loss: c (target) np.array~~~
l = -c*np.log(y)
    - (1-c)*np.log(1-y)
```

```python
import torch as th
# explicite bias
W = th.randn(1,D+1,requires_grad=True)
# inference
# x is a th.Tensor
a = W@x # or W.matmul(x)
y = 1/(1+th.exp(-a))
# a and y are th.Tensor
# loss: c (target), a th.Tensor
l = -c*th.log(y)
    - (1-c)*th.log(1-y)
```

Ex: the logistic regression model

Compute the gradient



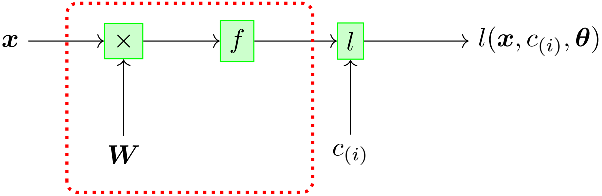
The goal :

$$\frac{\partial l}{\partial \boldsymbol{W}^{(1)}} = \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}^{(1)}}$$

Gradient computation:

$$\boxed{l} \rightarrow \frac{\partial l}{\partial \boldsymbol{y}} \rightarrow \boxed{f} \rightarrow \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{a}} \rightarrow \boxed{\times} \rightarrow \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}^{(1)}}$$

The computation graph in both directions



Forward (inference)	Backward (gradient)
$l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)}) \leftarrow \boldsymbol{y}$	$\dfrac{\partial l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})}{\partial \boldsymbol{W}} = \dfrac{\partial l(\boldsymbol{\theta}, \boldsymbol{x}_{(i)}, c_{(i)})}{\partial \boldsymbol{y}}$
$\boldsymbol{y} = f(\boldsymbol{a})$	$\times \dfrac{\partial \boldsymbol{y}}{\partial \boldsymbol{a}}$
$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x}$	$\times \dfrac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}$

Illustration in 3 steps

Initialization (inputs of the graph)

```
import torch as th
x = th.ones(3,1)
c = th.ones(1)
W = th.randn(1,3,requires_grad=True)
  tensor([[-0.3999,  0.1500,  0.3771]], requires_grad=True)
```

Forward: build the graph

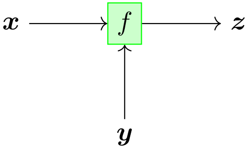
```
h = 1/(1+th.exp(-W@x))
  tensor([[0.4682]], grad_fn=<MulBackward0>)
l =  -y*th.log(h) - (1-y)*th.log(1-h)
  tensor([[0.7588]], grad_fn=<SubBackward0>)
  W.grad:  None
```

backward

```
l.backward()
  W.grad:  tensor([[0.5318, 0.5318, 0.5318]])
```

A function node

Forward pass

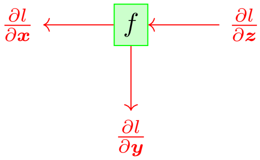


This node implements:

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

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:

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

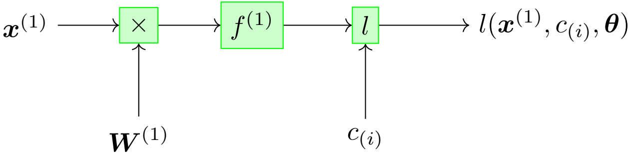
$$\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y}) \qquad \text{\# forward}$$

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

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

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

Example of a single layer network



Forward

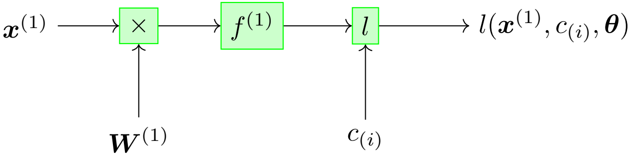
For each function node in topological order

- 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

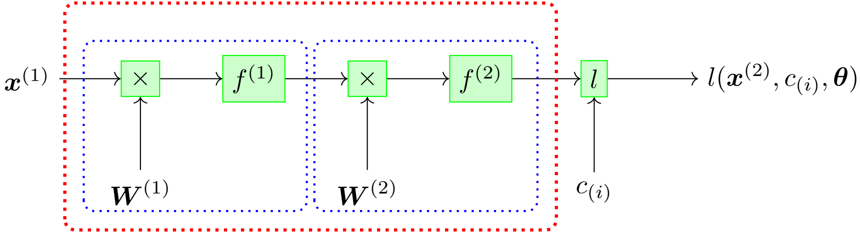
For each function node in reversed topological order

- 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

Outline

pytorch in three concepts

A *Tensor* is a tensor

- Similar to numpy's *ndarrays*, but can be used on a GPU to accelerate computing.
- A node of a computation graph, holding:
  - the gradient w.r.t to it self (back-propagation)
  - a reference to its creator

Autograd

Package for building computational graphs out of Tensors, and automatically computing gradients

Module

A neural network layer, may store state or learnable Function (i.e with parameters)

An example in pytorch

```
import torch as th
# The model
D_in=2  # input size : 2
D_out=1 # output size: one value
model = th.nn.Sequential(
   th.nn.Linear(D_in, D_out),
   th.nn.Sigmoid()
   )
loss_fn = th.nn.BCELoss()
# Optimizer will update  the weights of the model.
lr0 = 1e-4
optimizer = torch.optim.SGD(model.parameters(),
                          lr=lr0)
```

An example in pytorch - 2

```
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()
```

Three kinds of *Module*

Linear

- *Linear* is a *Module* for a linear transformation.
- The parameters: a *Tensor* $\boldsymbol{W}$
- Forward $\boldsymbol{x} \rightarrow \boldsymbol{W}\boldsymbol{x}$

Sigmoid

- *Sigmoid* is a *Module* for a pointwise function
- No parameters

Sequential

- *Sequential* is a container *Module*
- It contains a sequence of *Module*, i.e a feed-forward NNet

Look at the *torch.nn* doc for many examples

From CPU to GPU

```
# This vector is stored on cpu (+any operation you do on it)
a = torch.DoubleTensor([1., 2.])
# The same for GPU
a = torch.FloatTensor([1., 2.]).cuda()
a = torch.cuda.FloatTensor([1., 2.])
# it will be on the default device:
torch.cuda.current_device()
#####
# For the model:
model = model.cuda()
```

Define your module

```
class LogisticRegression(th.nn.Module):
    def __init__(self,D_in):
        super(LogisticRegression, self).__init__()
        self.lin = th.nn.Linear(D_in, 1)
        self.out = th.nn.Sigmoid()

    def forward(self, x):
        a = self.lin(x)
        return self.out(a)

mod = LogisticRegression(D_in=2)
```

G. Cybenko.
1989.
Approximation by superpositions of a sigmoidal function.
Mathematics of Control, Signals, and Systems (MCSS), 2(4):303–314, December.

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.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun.
2015.
Deep residual learning for image recognition.

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.

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng.
2013.
Rectifier nonlinearities improve neural network acoustic models.
In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*.

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

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber.
2015.
Training very deep networks.
CoRR, abs/1507.06228.