Neural Networks for Classification

An Introduction to Supervised ML

Joseph Le Roux

18/10/25

Notes

Outline

Notes

Working Example: Build a Classifier for Images

From a set of images classified into predefined categories, we want to build: a system able to classify new images

Input

Output

dog 0.14

cat 0.85

hamburger 0.01

Two-step process

1. Compute a probability for each category
2. Return the most probable category

***Data Mining* perspective**

1. Use a Neural Network (NN) to compute probabilities
2. Learn NN parameters from examples

Notes

General Form (1)

Functions transforming vectors of size i to vectors of size o

Linear Functions $\mathbb{R}^i \to \mathbb{R}^o$

We write linear functions $h_{i,o} : \mathbb{R}^i \to \mathbb{R}^o$, $h_{i,o}(x) = Ax + b$, with

- A matrix in $\mathbb{R}^{o \times i}$,
- b (column) vector in $\mathbb{R}^o$ called the *bias*
- $x \in \mathbb{R}^i$ the input (column) vector

This returns a (column) vector $y \in \mathbb{R}^o$ such that:

$$y = Ax + b$$

We have for each output dimension $1 \leq p \leq o$:

$$y_p = (A_p \cdot x) + b_p = (\sum_{k=1}^{i} A_{pk} \times x_k) + b_p$$

Remarks

Yes, these are affine functions, not linear... but we call them linear anyway.

Notes

General Form (2)

Elementary Activations

We call elementary activation the application of a function η in $\mathbb{R} \to \mathbb{R}$ to each element of a tensor (matrix, vector...) We note the application by η itself.

Example:

$\eta(x) = x^2$. then we have $\eta(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}) = \begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$

Neural Network (MLP)

is a function $\mathbb{R}^i \to \mathbb{R}^o$, constructed as follows:

$$R(x) = h^L_{o_{L-1},o} \circ \eta \circ \dots \circ \eta \circ h^2_{o_1,o_2} \circ \eta \circ h^1_{i,o_1}(x)$$

- $h^l_{i,j}$ are linear functions
- η is the elementary application of a non-linear function
- L is called *the number of layers* of the network
- o_l is the *size* of layer l

Alias : multi-layer perceptron (hence *MLP*)

Notes

Elementary Non-linearities

Make NN able to approximate general functions (without, only linear transformations, cf next)

Usual Non-linear Activations

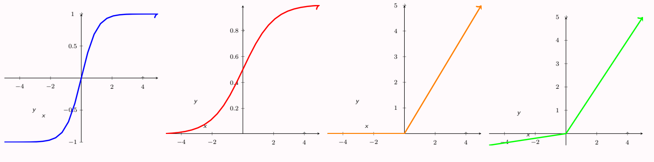


Figure: Examples of common non-linear activations: hyperbolic tangent, sigmoid, linear rectificator, *leeky* linear rectificator

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\text{ReLU}(x) = \max(0, x) \qquad \text{LReLU}(x) = \max(\min(0, 0.1x), x)$$

The purpose is to create threshold effects

Notes

A Universal Approximator

Informal version

For any function (∗) $f : \mathbb{R}^m \to \mathbb{R}^n$ *and all* $\epsilon > 0$, *there exists a MLP R s.t.* $|f(x) - R(x)| < \epsilon, \forall x$

- Actually, several theorems, depending on f, R and norm
- in these theorems R has either
  1. a possibly infinite number of layers,
  2. 2 layers : $R(x) = h^2_{t,o} \circ \eta \circ h^1_{i,t}(x)$ but layer size t can be arbitrary large.

Remark

1. Theoretical result, in practice number of parameters is limited
2. Still, MLP are used to approximate functions (with possibly error $> \epsilon$)

∗: abuse of language, there exist functions we cannot approximate well. . . (discontinuous, infinite values,. . .)

Notes

A Universal Approximator (II)

- *idea* of the proof: divide the output space in intervals and return the average value on this interval.
- see contruction and animations on `http://neuralnetworksanddeeplearning.com/chap4.html`

Notes

Example 1

```
import torch

# MLP for a function from  R^2 to R^1
class MLP2(torch.nn.Module):
  def __init__(self):
    super(MLP2, self).__init__()
    self.l1 = torch.nn.Linear(2,4) #lin transform R^2 -> R^4
    self.l2 = torch.nn.Linear(4,1) #lin transform R^4 -> R^1

  #definition of the function computed by the NN
  def forward(self, x):
    return self.l2(torch.tanh(self.l1(x)))

net = MLP2() # create object from class MLP2
#use net as function -> calls method  forward
print( net(torch.Tensor([2.,3.])) )
```

```
tensor([0.8033], grad_fn=<ViewBackward0>)
```

from something in R^2 we get something in R^1 and ... something else (see next slides)

Notes

Learning Parameters: What is a parameter?

To approximate a function $f : \mathbb{R}^m \to \mathbb{R}^n$, we must decide the form of the network R. We distinguish:

hyper-parameters number of layers, size of each layer, type of non-linearities;

parameters values of elements in matrices and vectors for each linear application h in R.

Notes

Example 2

```
print(list(net.parameters()))
```

```
[Parameter containing:
tensor([[ 0.2860, -0.2742],
        [ 0.4483,  0.4352],
        [ 0.2364,  0.1122],
        [-0.3820, -0.1753]], requires_grad=True), Parameter containing:
tensor([ 0.2429, -0.2452, -0.1346,  0.0896], requires_grad=True), Parameter containing:
tensor([[-0.2535,  0.4431,  0.1331, -0.2847]], requires_grad=True), Parameter containing:
tensor([0.0600], requires_grad=True)]
```

Remark about Notation

- Matrix of `Linear(m,n)` is of dimension $n \times m$

Notes

Example (ct)

```
x = torch.Tensor([2.,3.])
print(net(x)) #MLP as function: calls forward

h1 = net.l1(x)
t1 = torch.tanh(h1)
l2 = net.l2(t1)
print(h1)
print(t1)
print(l2)
```

```
tensor([0.8033], grad_fn=<ViewBackward0>)
tensor([-0.0076,  1.9571,  0.6748, -1.2003], grad_fn=<ViewBackward0>)
tensor([-0.0076,  0.9609,  0.5881, -0.8337], grad_fn=<TanhBackward0>)
tensor([0.8033], grad_fn=<ViewBackward0>)
```


Notes

Learning Parameters: Approximate a function f

Hyperparameters being fixed, we search the parameters for which the difference between the two systems (f and the network) is zero.

To measure the quality of approximation we define an error function. An example of such error function is MSE:

$$E_{f,R}(x) = \frac{1}{n}\sum_{k=1}^{n}\frac{1}{2}(f(x)[k] - R(x)[k])^2$$

Remark: $E : \mathbb{R}^m \to \mathbb{R}^+$ the result is a positive (or zero) scalar

If $E_{f,R}(x) < \varepsilon$ for all input x we have effectively managed to approximate f with R (with positive error ε that we want close to zero).

[cf. prog. for AI/Robotics INFO2]

Notes

Learning Parameters: Approximate *unknown* function f

In general, we want to approximate a function that we do not know analytically, but for which we only a subset ot its graph $T = \{(x_i, y_i = f(x_i))\}_i$ (*i.e.* instances of input/output f).

We will assign parameters to approximate these examples. We define a new function, a *loss function* which averages errors made on these instances:

$$L = \frac{1}{|T|}\sum_{(x_i,y_i)\in T} E_{y_i,R}(x_i) = \frac{1}{|T|}\sum_{(x_i,y_i)\in T}\frac{1}{n}\sum_{k=1}^{n}\frac{1}{2}(y_i[k] - R(x_i)[k])^2$$

- If L is close to zero, R approximates f on instances.

Supervised Learning

here we have x and y, input and correct output ($\neq$ AI/Robotics INFO2)

Notes

Example 3: Computing a MSE Loss

```python
# T: set of examples (x,y)
# x vector in R^2, y scalar (in R)
# beware: not efficient (we will fix that later)
local_losses = [ torch.mean(torch.square(y - net(x))/2) for (x,y) in T]

# transform list to tensor and 'averaging'
loss = torch.mean(torch.stack(local_losses))
```


Notes

Learning Parameters: where are the parameters?

We will write R with additional arguments in order to make parameters explicit, *i.e.* $R(x;\theta)$ with θ the concatenation of all parameters (elements of vectors/matrices)

- For error:

$$E_{y,R}(x;\theta) = \frac{1}{n}\sum_{k=1}^{n}(y[k] - R(x;\theta)[k])^2$$

- For loss:

$$L(\theta) = \frac{1}{|T|}\sum_{i} E_{y_i,R}(x_i;\theta)$$

Notes

Learning and Optimization

Since function L is positive or null (if no error), learning parameters for R can be cast as the following problem:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L(\theta) = \underset{\theta}{\operatorname{argmin}}\, \frac{1}{|T|} \sum_{(x_i,y_i)\in T} E_{y_i,R}(x_i;\theta)$$

We search parameters which minimize L ($\rightarrow$ which zero L)

Remarks

1. If differentiable elem. non-linearities (wrt to θ), R differentiable too;
2. gradient of function f: vector of partial derivatives wrt to a vector of variables. For $L(\theta)$, we note its gradient wrt to θ as $\nabla_\theta L(\theta)$ or simply $\nabla L(\theta)$;
3. Solution to above problem is among solutions of $\nabla L(\theta) = \mathbf{0}$
4. L is not convex in general: several solutions.
5. Not important (What!!???): local minima will be good enough.
6. Problem: $\nabla L(\theta) = \mathbf{0}$ too difficult to solve analytically

Notes

Learning by gradient descent (1)

We will solve our problem (find a local minimum of $L(\theta)$) by an iterative method:

Gradient Descent Algorithm

- Initialize θ^0 randomly
- $t = 0$
- While $\|\nabla L(\theta^t))\|_2 \geq \epsilon$
  - Determine the size of step $\alpha^t > 0$ (more on that later)
  - $\theta^{t+1} = \theta^t - \alpha^t \nabla L(\theta^t)$
  - t = t+1
- return final θ

Simple, provided we can compute gradient $\nabla L(\theta^t)$, does this work?

Notes

Learning by Gradient Descent (2)

Easy part,each iteration improves the solution (less errors on examples). We start from the Taylor Expansion of L around θ^t:

$$\begin{aligned}
L(\theta^{t+1}) = L(\theta^t - \alpha^t \nabla L(\theta^t)) &= L(\theta^t) + \langle \nabla L(\theta^t), (\theta^t - \alpha^t \nabla L(\theta^t)) - \theta^t \rangle + o(|\alpha^t \nabla L(\theta^t)|) \\
&= L(\theta^t) + \langle \nabla L(\theta^t), -\alpha^t \nabla L(\theta^t) \rangle + o(|\alpha^t \nabla L(\theta^t)|) \\
&= L(\theta^t) - \alpha^t \|\nabla L(\theta^t)\|^2 + o(|\alpha^t \nabla L(\theta^t)|) \\
&\approx L(\theta^t) - \alpha^t \|\nabla L(\theta^t)\|^2 \text{ for small enough } \alpha^t \\
&\leq L(\theta^t)
\end{aligned}$$

Technical problems (for math people)

1. What does *small enough* α^t means?
2. What if $L(\theta^{t+1}) = L(\theta^t)$ (if $\alpha^t = 0$), does it work?

To prove (rate of) convergence we would need stronger assumptions on L (smoothness, strong convexity, blabla. . .)

Notes

Learning by Gradient Descent (3)

Technical problem for non-math people: too slow

$$\nabla L(\theta) = \nabla \frac{1}{T} \sum_{(x,y)\in T} E(y, R(x;\theta))$$

- hidden nested loop: iterate through all examples to compute the gradient!
- too slow, and impossible to use with *big-data* (millions of examples)

Notes

Learning by Gradient Descent (4)

Computing average $\equiv$ expectation

$$\nabla L(\theta) = \nabla \frac{1}{|T|} \sum_{(x,y)\in T} E(y, R(x;\theta)) = \sum_{(x,y)\in T} \frac{1}{|T|} \nabla E(y, R(x;\theta)) = \mathbb{E}_{(x,y)\sim p(x,y)}[\nabla E(y, R(x;\theta))]$$

with uniform distribution from T: $\forall (x,y) \in T, p(x,y) = \frac{1}{|T|}$.

- We can replace the expectation with MC sampling (cf. INFO2)

Stochastic Gradient Descent: update after each example

- Initialize θ^0 randomly; $t = 0$
- While θ^t *is different from* θ^{t+1}
  - pick up an example (x, y) randomly in T
  - Determine step size $\alpha^t > 0$
  - Update: $\theta^{t+1} = \theta^t - \alpha^t \nabla E(y, R(x;\theta^t))$ and $t = t + 1$

Intuition and Remarks

- Numerous small updates give useful parameters very early in training
- But: we do not have $L(\theta^{i+1}) < L(\theta^i)$ (reasoning *in expectation*)
- Solution: hybrid the 2 methods and perform updates on $k << |T|$ examples (gradient descent on *batches*).

Notes

Stochastic Gradient Descent (5)

WLOG, we assume that R approximates f a function in $\mathbb{R}^m \to \mathbb{R}$

How to compute a gradient ?

For our approximation error E (MSE):

$$\begin{aligned}
\nabla E_{y,R}(x;\theta)) &= \nabla \frac{1}{2}(y - R(x;\theta))^2 \\
&= -y\nabla R(x;\theta) + R(x;\theta)\nabla R(x;\theta) \\
&= (R(x;\theta) - y)\nabla R(x;\theta)
\end{aligned}$$

Of course, this must be adapted if we use a different error function.

But... how to compute $\nabla R(x;\theta)$?

- R is a composition of functions
- use the chaining rule (remember $f(g(x))' = f'(g(x)) \cdot g'(x)$)
- can be done automatically, computable in $O(n)$ where n is the number of parameters : this is called *backpropagation*
- no need to compute gradient by hand ☻

Notes

Example 4 (1)

```
# assume example ([2,3], 5):
local_loss = torch.square(5. - net(torch.tensor([2.,3.])))/2
print(local_loss)
#print matrix R^2 -> R^4 and its gradient
print("weight ", net.l1.weight.data)
print("grad ", net.l1.weight.grad)
```

```
tensor([8.8063], grad_fn=<DivBackward0>)
weight  tensor([[ 0.2860, -0.2742],
        [ 0.4483,  0.4352],
        [ 0.2364,  0.1122],
        [-0.3820, -0.1753]])
grad  None
```


Notes

Example 4 (2)

```
local_loss = torch.square(5. - net(torch.tensor([2.,3.])))/2
print(local_loss)
local_loss.backward() #compute gradient for this example
print("grad ", net.l1.weight.grad)
#update via SGD: (alpha = 0.001)
net.l1.weight.data = net.l1.weight.data - 0.001 * net.l1.weight.grad
net.l1.weight.grad=None #reset gradient
print("weight" ,net.l1.weight.data)
print(torch.square(5. - net(torch.tensor([2.,3.])))/2)
```

```
tensor([8.8063], grad_fn=<DivBackward0>)
grad  tensor([[ 2.1277,  3.1915],
        [-0.2854, -0.4281],
        [-0.7308, -1.0962],
        [ 0.7284,  1.0926]])
weight tensor([[ 0.2839, -0.2774],
        [ 0.4486,  0.4356],
        [ 0.2371,  0.1133],
        [-0.3827, -0.1764]])
tensor([8.7879], grad_fn=<DivBackward0>)
```


Notes

XOR

We want to learn function XOR (exclusive or):

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

- $T = \{((0,0),0), ((0,1),1), ((1,0),1), ((1,1),0)\}$
- we use for error *squared difference*

Notes

XOR

with linear MLP!!

```
import torch

#Input Data
Xdata = torch.tensor([[0,0],[0,1],[1,0],[1,1]], dtype=torch.float)
Ydata = torch.tensor([0,1,1,0], dtype=torch.float)

# A linear function
class MLP1(torch.nn.Module):
  def __init__(self):
    super(MLP1, self).__init__()
    self.l1 = torch.nn.Linear(2,1) # linear transform from R^2 to R^1

  #definition of the computation
  def forward(self, x):
    return self.l1(x)
```


Notes

Learning

```
def train(network, nb_exp=1000, frq_eval=10, alpha=0.1):
  mean_error_score = 0
  for m in range(nb_exp):
    # example selection: 1 sample   0 <=s < nb examples
    i = torch.randint(0, Xdata.size(0),(1,))
    # error computation
    # we do not divide by 2 and the number of examples
    # the learning rate alpha will be adapted
    error = torch.square(Ydata[i] - network(Xdata[i]))

    mean_error_score += error.item()

    # gradient computation
    error.backward()

    #gradient descent for all parameters
    for param in network.parameters():
      param.data.copy_(param.data - alpha * param.grad)
      param.grad = None #reset gradient

    #displayinformation
    if ((m+1) % frq_eval) == 0:
      print(m+1, mean_error_score/(m+1))
```


Notes

What is going on? (1)

```
net1 = MLP1()

train(net1, nb_exp=100000,frq_eval=10000)
```

```
10000 0.31219633601758673
20000 0.31260662023698654
30000 0.3135818249658544
40000 0.3136580531996927
50000 0.3141801899289425
60000 0.3140056242947169
70000 0.31357454647844923
80000 0.3139633190718281
90000 0.31428038918919554
100000 0.314269295872414
```

Loss not really decreasing. . .

Notes

What is going on? (2)

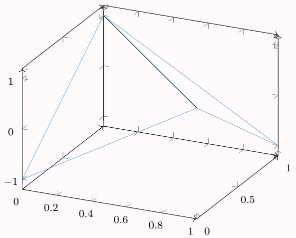
Tests

```
print(net1(torch.tensor([0.,0.])))
print(net1(torch.tensor([0.,1.])))
print(net1(torch.tensor([1.,0.])))
print(net1(torch.tensor([1.,1.])))
```

```
tensor([0.4548], grad_fn=<ViewBackward0>)
tensor([0.5784], grad_fn=<ViewBackward0>)
tensor([0.4645], grad_fn=<ViewBackward0>)
tensor([0.5880], grad_fn=<ViewBackward0>)
```

Linear MLP could not find useful parameters !!

Expected, impossible to find a linear function to approximate XOR (draw picture)



Notes

With Non-Linear Function (1)

```
net2 = MLP2()
train(net2, nb_exp=100000,frq_eval=10000)
```

```
10000 0.0077581390788330935
20000 0.0038790695394183457
30000 0.0025860463596134287
40000 0.0019395347697109737
50000 0.0015516278157694807
60000 0.0012930231798084924
70000 0.0011083055826935054
80000 0.0009697673848572572
90000 0.0008620154532068437
100000 0.0007758139078865102
```


Notes

With Non-Linear Function (2)

Tests

```
print(net2(torch.tensor([0.,0.])))
print(net2(torch.tensor([0.,1.])))
print(net2(torch.tensor([1.,0.])))
print(net2(torch.tensor([1.,1.])))


tensor([0.], grad_fn=<ViewBackward0>)
tensor([1.], grad_fn=<ViewBackward0>)
tensor([1.0000], grad_fn=<ViewBackward0>)
tensor([0.], grad_fn=<ViewBackward0>)
```

MLP2 found good parameters!

Notes

Gradient Descent

Different variants of SGD. Implemented as subclasses of `torch.optim.optimizer`

- `SGD` vanilla stochastic gradient descent
- `AdaGrad`, `Adam`, `AmsGrad` maintain average of previous gradients to adjust α steps for each parameter individually
- `LBFGS` uses 2nd-order derivatives to compute optimal α

Notes

Parameter Update with Optimizer (1)

```
def train(network, optimizer, nb_exp=1000, frq_eval=100, alpha=0.1):
  mean_error_score = 0
  for m in range(nb_exp):
    optimizer.zero_grad() #reset gradient

    # example selection
    i = torch.randint(0, Xdata.size(0),(1,))
    #error computation
    error = torch.square(Ydata[i] - network(Xdata[i]))
    mean_error_score += error.item()

    # gradient computation
    error.backward()
    #gradient descent
    optimizer.step()

    #display some info
    if ((m+1) % frq_eval) == 0:
      print(m+1, mean_error_score/(m+1))
```


Notes

Parameter Update with Optimizer (2)

```
net2 = MLP2()
opt = torch.optim.SGD(net2.parameters(), lr=0.1)

train(net2, opt, nb_exp=100000,frq_eval=25000)
```

```
25000 0.0033479650764059905
50000 0.0016739825382035844
75000 0.0011159883588027853
100000 0.0008369912691023847
```

```
print(net2(torch.tensor([0.,0.])))
print(net2(torch.tensor([0.,1.])))
print(net2(torch.tensor([1.,0.])))
print(net2(torch.tensor([1.,1.])))
```

```
tensor([0.], grad_fn=<ViewBackward0>)
tensor([1.], grad_fn=<ViewBackward0>)
tensor([1.], grad_fn=<ViewBackward0>)
tensor([-8.9407e-08], grad_fn=<ViewBackward0>)
```


Notes

Parameter Update with `Optimizer` (3)

```python
net2 = MLP2()
opt = torch.optim.Adam(net2.parameters(), lr=0.001)

print("Hello")
train(net2, opt, nb_exp=100000,frq_eval=25000)
print("World")
```

```
Hello
25000 0.056801105387168364
50000 0.028400552693584182
75000 0.018933701795722787
100000 0.014200276346792091
World
```

```python
print(net2(torch.tensor([0.,0.])))
print(net2(torch.tensor([0.,1.])))
print(net2(torch.tensor([1.,0.])))
print(net2(torch.tensor([1.,1.])))
```

```
tensor([0.], grad_fn=<ViewBackward0>)
tensor([1.], grad_fn=<ViewBackward0>)
tensor([1.], grad_fn=<ViewBackward0>)
tensor([0.], grad_fn=<ViewBackward0>)
```


Notes

Remember our Goal: Build a Classifier for Images

From a set of images classified into predefined categories, we want to build: a system able to classify new images

Input

Output

dog 0.14
cat 0.85
hamburger 0.01

Two-step process

1. Compute a probability for each category
2. Return the most probable category

***Data Mining* perspective**

1. Use a Neural Network (NN) to compute probabilities
2. Learn NN parameters from examples

Notes

Neural Networks for Classification (1)

Most tasks with NNs are classification tasks (even in the age of GenAI!)

From an input in $\mathbb{R}^d$, assign it a class c among C possibilities

- given a picture predict whether it's a dog, a cat...
- given a text predict whether it's about politics, sports...

With a MLP:

- define a MLP R which approximate a function from $\mathbb{R}^d \to \mathbb{R}^C$
- each output dimension c represents the score of the c^{th} class given the input
- return the dimension whose score is the highest: given $x \in \mathbb{R}^d$, return $c^* = \mathrm{argmax}_c\, R(x)[c]$

Remark

We are not really interested in class scores, but how they compare to each other!!

Notes

Neural Networks for Classification (2)

Probabilistic loss

Learn parameters of R for classification from examples

- Assume we have examples $\mathcal{T} = \{(x_i, c_i)\}_i$ (input,correct class)
- we want to train our NN to predict c_i when given x_i, or
- we want to train our NN to predict c_i the most probable class given x_i

2 Issues

1. How do we turn scores into probabilities
2. How do we formally express the learning intuition (most probable class)

Notes

Neural Networks and Categorical Probabilities

Turn scores into probabilities: the softmax function

We define the probability of class c given an input x as:

$$p(c|x) = \frac{\exp(R(x)[c])}{\sum_{c'} \exp(R(x)[c'])} = \mathsf{softmax}(R(x))[c]$$

- $\exp$ to get positive values
- $\sum$ in denominator to get values between 0 and 1, which all sum to 1

This has many names (Gibb's distribution, exponential family...), and is already implemented in pytorch!

Notes

Loss function for Classification: Learning as Likelihood Estimation

Goal

We want examples $\mathcal{T} = \{(x_i, c_i)\}_i$ to be as probable as possible (ie. $p(c_i|x_i) \to 1$).

Remark

if $\lim p(c|x) \to 1$, it means that for $c' \neq k$ we have $\lim p(c'|x) \to 0$

We want to maximize the *likelihood* of $\mathcal{T}$

$$\prod_i p(c_i|x_i)$$

This is a product... difficult to optimize (+ precision issues)

- work in the log domain $\to$ the product becomes a sum
- multiply by -1 $\to$ maximization becomes a minimization with optimal zero: we have a loss

Finally the loss function to be minimized is:

$$L(\theta) = -\sum_i \log p(c_i|x_i) = -\sum_i \log(\mathsf{softmax} R(x_i;\theta))[c_i]$$

this is called the *negative log-likelihood*

Notes

Learning as Cross-Entropy

In Pytorch

Negative log-likelihood is called cross-entropy loss, why?

Definition: (Conditional) Cross Entropy

between 2 distribution q and p for one input:

$$\mathtt{CE}(q,p) = -\sum_c q(c|x)\log p(c|x)$$

Equivalence

Let us compute the cross-entropy between for one input in $\mathcal{T}$

- q the empirical distribution (1 for examples, 0 otherwise)
- p the distribution parameterized by our MLP

Notes

Learning as Cross-Entropy (2)

$$\mathtt{CE}(q,p) = -\sum_c q(c|x_i)\log p(c|x_i)$$

$$\mathtt{CE}(q,p) = -q(c_i|x_i)\log p(c_i|x_i) - \sum_{c\neq c_i} q(c|x_i)\log p(c|x_i)$$

$$\mathtt{CE}(q,p) = -1\times\log p(c_i|x_i) - \sum_{c\neq c_i} 0\times\log p(c|x_i)$$

$$\mathtt{CE}(q,p) = -\log p(c_i|x_i)$$

Summing this for all examples gives the negative log likelihood !

Notes

Cross-Entropy in Pytorch (3)

```
# classifier for input of size 10, 5 classes:
net = MLP2(10,8,5)

# let us pretend the following tensor contains
# the input for 3 examples
inputs = torch.rand(3,10)

# we obtain the 5 scores for all 3 examples
preds = net(inputs)

# let us suppose that the correct classes were:
empirical = torch.tensor([0,2,1], dtype=torch.long)

loss_function = torch.nn.CrossEntropyLoss()

# note that we do not compute log softmax (inside CE)
loss = loss_function(preds, empirical)

loss.backward() # and the rest...
```


Notes

Cross-Entropy in Pytorch (2)

At testing time:

```
# classifier for input of size 10, 5 classes:
net = MLP2(10,8,5)

# let us pretend the following tensor contains
# the input for 3 examples
inputs = torch.rand(3,10)

# we obtain the 5 scores for all 3 examples
predictionss = net(inputs)

#apply the argmax for each line:
outputs = torch.argmax(predictionss, dim=1)
```

no cross-entropy, no softmax: why?

Notes

Gradient Computation: How Does this Work?

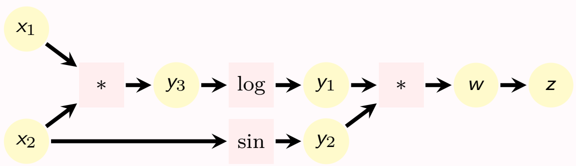
NN toolkit implement efficient gradient computation

gradient of the loss function wrt to NN parameters θ. Based on:

- the chain rule $(f(g(x)))' = f'(g(x)) \times g'(x)$
- the *Computation Graph* a graph describing the computation: parameters are leaf nodes

Example Computation (from the pytorch website)

```
x1, x2 = 0.5, 0.75
z = log(x1 * x2) * sin(x2)
```




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```

We want to compute the gradient of `z` wrt parameters `x1,x2`

Notes

Computing derivatives by hand using the chain rule

We want to derive $z = g_1(\log(g_2(x1, x2)), sin(x2))$ where $g_1(x, y) = g_2(x, y) = xy$

We note: $y_1 = \log\left(g_2(x1, x2)\right)$ $\quad y_2 = \sin(x2)$ $\quad y_3 = g_2(x1, x2)$

$$\frac{\partial z}{\partial x1} = \frac{\partial y_1}{\partial x_1}\frac{\partial g_1(y_1, y_2)}{\partial y_1} + \frac{\partial y_2}{\partial x_1}\frac{\partial g_1(y_1, y_2)}{\partial y_2} = \frac{\partial y_1}{\partial x_1}y_2 + \frac{\partial y_2}{\partial x_1}y_1 \quad (1)$$

$$\frac{\partial y_1}{\partial x1} = \frac{\partial y_3}{\partial x_1}\frac{\partial \log y_3}{\partial y_3} = \frac{\partial y_3}{\partial x_1}\frac{1}{y_3} = \frac{\partial y_3}{\partial x_1}\frac{1}{x_1 x_2} \quad (2)$$

$$\frac{\partial y_2}{\partial x1} = \frac{\partial x_2}{\partial x_1}\frac{\partial \sin x2_2}{\partial x_2} = 0\cos(x_2)) = 0 \quad (3)$$

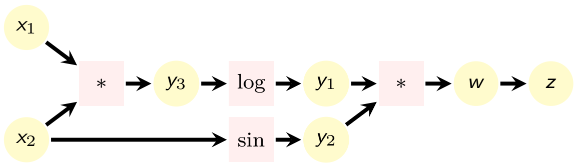
$$\frac{\partial y_3}{\partial x1} = \frac{\partial g(x_1, x_2)}{\partial x_1} = x_2 \quad (4)$$

Putting it all together: $\frac{\partial z}{\partial x_1} = x_2\frac{1}{x_1 x_2}\sin(x_2) + 0\log(g_2(x1, x2)) = \frac{sin(x_2)}{x_1}$

The same thing applies to $\frac{\partial z}{\partial x_2}$, we could do them in parallel by computing the vector of derivatives at each time

Notes

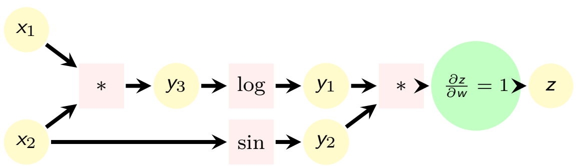
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

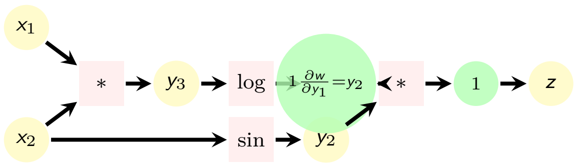
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

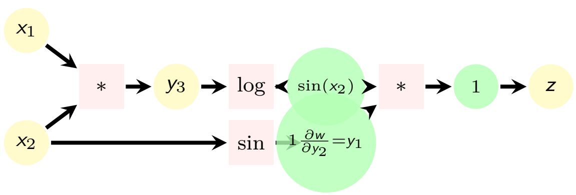
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

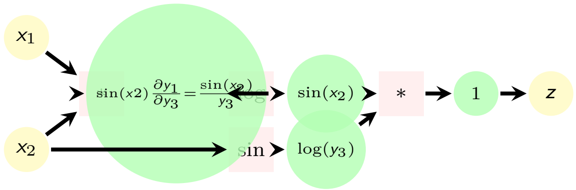
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

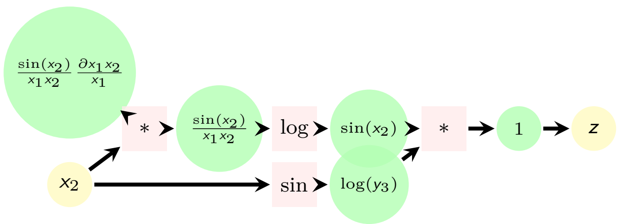
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

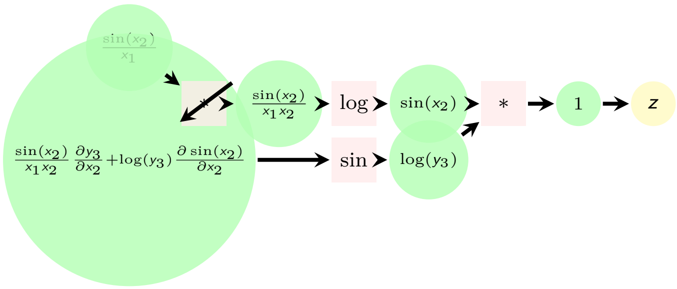
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

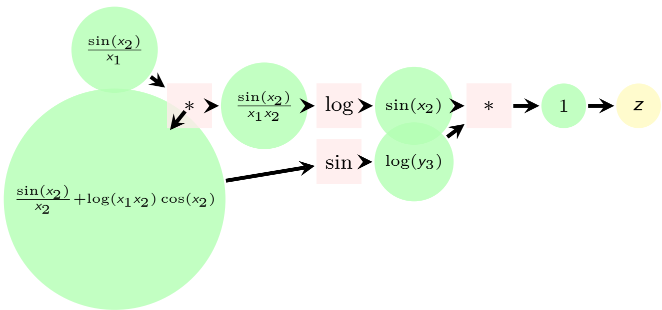
Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```


Notes

Computing derivatives with a computation graph




```
y3  = x1 * x2
y1 = log(y3)
y2 = sin(x2)
w = y1 * y2
z = w
```

In practice, variables are replaced by their values computed during *the forward pass* (the computation of z).

Notes

Conclusion

Neural Networks

- composition of linear mappings and elementary non-linear activations
- can be *learned* (parametrized) by reviewing labeled examples
- learning amounts to gradient descent of a loss function

Classification

- Output a score for each class
- Transform into a probability with softmax
- Learn with Negative Log-Likelihood loss

Back Propagation

- method to implement gradient descent efficiently
- works on a *computation graph* (DAG)
- share computation: linear complexity in the depth of the DAG

Notes

Notes