Neural Networks for Classification

An Introduction to Supervised ML

Joseph Le Roux

06/08/26

Notes

Outline

Notes

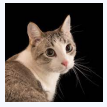
1 Goal

Notes

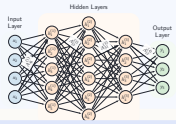
1 Working Example: Build a Classifier for Images

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

A small detour

Before we use NN for classification, let us see how they can model functions in general.

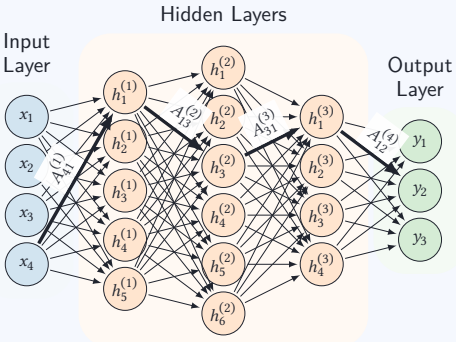
Notes

2 Neural Networks

Notes

2 General Form (0)

Mental Picture of a Neural Network



from https://youtu.be/MiHrd2rtCZQ?si=NUJHmjTRsaGYc4QE

a neural network example

- implements a function $\mathbb{R}^4 \to \mathbb{R}^3$ (i/o)
- has 3 hidden layers
- is *fully connected*
- no loop
- operations:
  1. multiply by coeff on outgoing edge
  2. sum over all incoming edges

Notes

2 General Form (1)

Functions transforming vectors of size i to vectors of size o

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

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

2 General Form (2)

Elementary Activations

We call elementary activation the application of a function η in $\mathbb{R} \longrightarrow \mathbb{R}$ 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 $\mathbb{R}^i \longrightarrow \mathbb{R}^o$, constructed as follows:

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

- $h^l_{i,j}$ 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

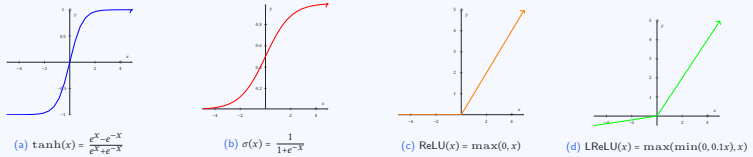
Alias : multi-layer perceptron (hence *MLP*)

Notes

2 Elementary Non-linearities

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

Usual Non-linear Activations



(a) $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (b) $\sigma(x) = \frac{1}{1+e^{-x}}$ (c) $\text{ReLU}(x) = \max(0, x)$ (d) $\text{LReLU}(x) = \max(\min(0, 0.1x), x)$

Figure 1: Examples of common non-linear activations: hyperbolic tangent, sigmoid, linear rectificator, *leaky* linear rectificator

The purpose is to create threshold effects

Notes

2 A Universal Approximator

Informal version

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

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

- 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 : $R(x) = h^2_{t,0} \circ \eta \circ h^1_{i,t}(x)$ but layer size t can be arbitrary large.

Idea of the proof

Divide the output space in intervals, return average value on interval.

- approximation quality increase with number of intervals
- see contruction on neuralnetworksanddeeplearning.com

Remarks

1. Above result theoretical
   - in practice number of parameters is limited
2. Still, MLP used to approximate (with error)

In practice

Depth matters more than width

Why Depth Wins

- Hierarchy Early layers spot basic shapes, deeper combine into objects.
- Efficiency Adding layers splits input space exponentially faster than widening
- Savings Deep networks approximate functions with a linear or polynomial scale, whereas wide networks need exponential neuron counts.

Notes

3
Learning: Optimising Loss

Notes

3 *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.

Error Approximation Function [cf. IMRL/Robotics INFO2]

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

- $E : \mathbb{R}^m \longrightarrow \mathbb{R}^+$ the result is a positive (or zero) scalar
- This is an example, other Error functions are possible (and better sometimes)

Good Approximation

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).

Notes

3 *Learning* Parameters: Approximate *unknown* f

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

Loss Function

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 vs. Reinforcement Learning

- here we have x and y, input and correct output ($\neq$ IMRL/Robotics INFO2)
- gradient methods are exact

Notes

3 Example: Computing MSE Loss

```
# 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

3 Learning Parameters: Generalisation vs. Overfitting

While we learn on given training set, we are interested in applying our model on new data. How good is it?

Overfitting

Can we expect a model to be accurate if training set loss is low (0)?

- No, with enough parameters, an MLP can learn the training set by heart, and still be unable to predict from new data.
- We need to evaluate on a disjoint set, mimickin new data, called the the test set.

Generalisation

We want to select the model that performs best on unseen data:

- We use a validation set, disjoint from both training set and test set, to choose when to stop gradient descent.

Notes

3 *Learning* Parameters: Where are They?

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)

Parametrised Error Function

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

Parametrised Loss Function

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

Notes

3 Learning and Optimisation

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

Learning Objective

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

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

Remarks

1. If differentiable elem. non-linearities (wrt θ), R differentiable too;
2. gradient of function f: vector of partial derivatives wrt 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

3 Learning by gradient descent (1/5)

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?

Intuition

We move the current estimate by moving it (a tiny bit) in the direction that reduces the loss most steeply

Notes

3 Learning by Gradient Descent (2/5)

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

3 Learning by Gradient Descent (3/5)

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 average gradient!
- too slow, and impossible to use with *big-data* (millions of examples)

Approximate (near-)infinte Average ? We saw that last year...

Computing average ≡ 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 approximate the expectation with MC sampling (cf. INFO2)
- (we could prove this is an unbiased estimator, so everything will be fine...)

Notes

3 Learning by Gradient Descent (4/5)

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

3 Learning by Stochastic Gradient Descent (5/5)

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

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 chain 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

4 Learning: Gradient Computation

Notes

4 Gradient Computation: How Does this Work?

NN toolkit implement efficient gradient computation

gradient of the loss function wrt 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

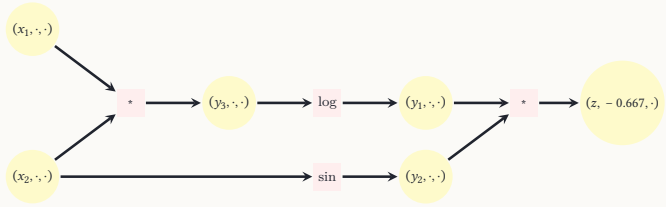
Example Computation (from the pytorch website)

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

We decompose as:

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

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



We know how to compute values i.e the *forward pass*: from input to output, apply operator semantics when incoming nodes are ready, and propagate.

Notes

4 Computing derivatives by hand: 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: $y_1 = \log\left(g_2(x1, x2)\right)$ $y_2 = \sin(x2)$ $y_3 = g_2(x1, x2)$

$$\frac{\partial z}{\partial x1} = \frac{\partial y_1}{\partial x_1}\frac{\partial g_1(y_1, y_2)}{\partial y_1} + \frac{\partial y_2}{\partial x_1}\frac{\partial g_1(y_1, y_2)}{\partial y_2} = \frac{\partial y_1}{\partial x_1}y_2 + \frac{\partial y_2}{\partial x_1}y_1 = 0.682\frac{\partial y_1}{\partial x_1} - 0.981\frac{\partial y_2}{\partial x_1} \quad (1)$$

$$\frac{\partial y_1}{\partial x1} = \frac{\partial y_3}{\partial x_1}\frac{\partial \log y_3}{\partial y_3} = \frac{\partial y_3}{\partial x_1}\frac{1}{y_3} = \frac{\partial y_3}{\partial x_1}\frac{1}{x_1x_2} = \frac{1}{0.375}\frac{\partial y_3}{\partial x_1} \quad (2)$$

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

$$\frac{\partial y_3}{\partial x1} = \frac{\partial g(x_1, x_2)}{\partial x_1} = x_2 = 0.75 \quad (4)$$

Putting it all together: $\frac{\partial z}{\partial x_1} = x_2\frac{1}{x_1x_2}\sin(x_2) + 0\log(g_2(x1, x2)) = \frac{sin(x_2)}{x_1} = \frac{0.682}{0.5} = 1.36$

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

4 Computing Derivatives on Computation Graph

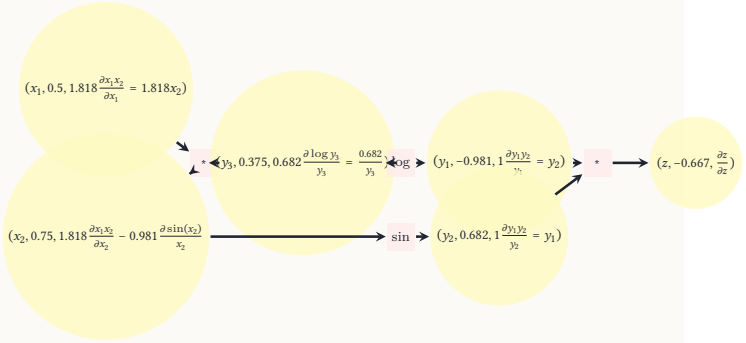
Example Computation (from the pytorch website)

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

We decompose as:

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

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



We can compute derivatives through a *backward pass*: from output to input, apply operator derivative semantics when outgoing nodes are ready, and propagate. This is backpropagation (reverse-mode automatic differentiation)

Notes

5 Learning: With Pytorch

Notes

5 Example: Computing gradient (1/2)

```
# 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)
```

Notes

5 Example: Computing gradient (2/2)

```
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)
```


Notes

6 Example: MLP for XOR

Notes

6 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

6 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

6 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

6 What is going on? (1/2)

```
net1 = MLP1()

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

```
10000 0.3133946860417653
20000 0.3135637994147413
30000 0.3148711372495861
40000 0.3154004126131145
50000 0.3151257636393937
60000 0.3147700414181888
70000 0.3145736871188287
80000 0.3147026740897139
90000 0.314810704013109
100000 0.31475289222942765
```

Loss not really decreasing...

Notes

6 What is going on? (2/2)

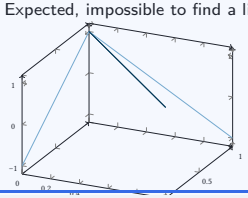
Tests

```
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([0.3702], grad_fn=<ViewBackward0>)
tensor([0.3041], grad_fn=<ViewBackward0>)
tensor([0.4604], grad_fn=<ViewBackward0>)
tensor([0.3943], grad_fn=<ViewBackward0>)
```

Linear MLP could not find useful parameters !!

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



Notes

6 With Non-Linear Function (1/2)

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


Notes

6 With Non-Linear Function (2/2)

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.])))
```

MLP2 found good parameters!

Notes

7 Learning: Variants of SGD

Notes

7 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

7 Parameter Update with `Optimizer` (1/3)

```
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

7 Parameter Update with `Optimizer` (2/3)

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

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

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


Notes

7 Parameter Update with `Optimizer` (3/3)

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

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

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


Notes

8 Neural Networks for Classification

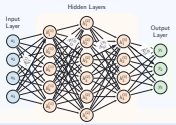
Notes

8 Remember our Goal: Build an Image Classifier

As we said in our very first slide, 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

8 Neural Networks for Classification (1/2)

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 $c^* = \operatorname{argmax}_c R(x)[c]$

Remarks

1. We are not really interested in class scores, but how they compare to each other!!
2. The input is a picture, not a vector:
   - we flatten the pixel grid (picture as a sequence of pixels)
   - We will discuss this in more details in the next lecture

Notes

8 Neural Networks for Classification (2/2)

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

8 Neural Networks and Categorical Probabilities

Turn scores into probabilities: the softmax function

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

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

- $\exp$ 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

8 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. $p(c_i|x_i) \longrightarrow 1$).

Remark

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

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 $\longrightarrow$ the product becomes a sum
- multiply by -1 $\longrightarrow$ maximization becomes a minimization with optimal zero: we have a loss

Finally the loss function to be minimized is:

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

Notes

8 Learning as Cross-Entropy (1/2)

In Pytorch

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

Definition: (Conditional) Cross Entropy

between 2 distribution q and p for one input:

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

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
- For a training example $\{x_i, c_i\}$, we can show:

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

Summing over training examples brings us back to the negative log-likelihood of the training set.

Notes

8 Learning as Cross-Entropy (2/2)

Proof of the derivation

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

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

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

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

Summing this for all examples gives the negative log likelihood !

Notes

8 Cross-Entropy in Pytorch (1/2)

```
# 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

8 Cross-Entropy in Pytorch (2/2)

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

9 Conclusion

Notes

9 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