

Recall the situation, the deep feed forward neural net (also called MLP = multi-layered perceptron)

has the form

$$F(x, \eta) = F_L \circ \dots \circ F_1(x)$$

$$F_i(x) = \sigma(A_i x + b_i)$$

The parameters are $\eta = A_1, \dots, A_L, b_1, \dots, b_L$ with correct output

The training data is x_1, \dots, x_N
 y_1, y_2, \dots, y_N

The loss or error or objective function is (simplest version)

$$E(\eta) = \frac{1}{N} \sum_{i=1}^N \|F(x_i, \eta) - y_i\|^2$$

Learning consists of adjusting the parameters η so that the error diminishes.

42

• So Φ depends on F which depends on

$F_1 \dots F_n$ as a composition

• This is a job for the chain rule for multidimensional functions.

• First recall the derivative as a matrix

If $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ so

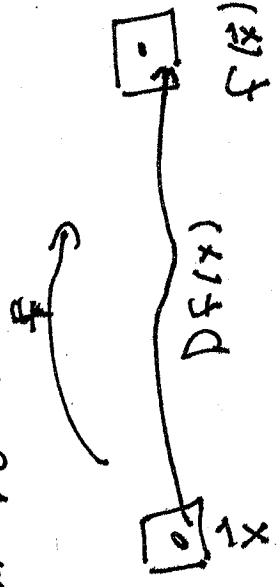
$$f(x_1, \dots, x_n) = (f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n))$$

$$\text{Then } Df = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \dots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

or $(Df)_{ij} = \frac{\partial f_i}{\partial x_j}$ Thus as a linear

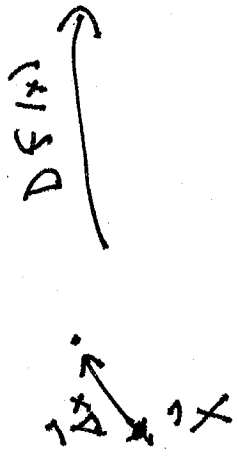
transformation $Df: \mathbb{R}^n \rightarrow \mathbb{R}^m$ also.

• The Derivative $Df(\vec{x})$ is the best local linear approximation to f at \vec{x}



• Taylor's Theorem

$$f(\vec{x} + \Delta \vec{x}) = f(\vec{x}) + Df(\vec{x}) \Delta \vec{x} + \text{h.o.t.}$$



$$Df(\vec{x}) \cdot \Delta \vec{x}$$

4

• Example $f(x_1, x_2) = (x_1^2 x_2, x_1 + x_2)$

$$Df(x_1, x_2) = \begin{bmatrix} 2x_1 x_2 & x_1^2 \\ 1 & 1 \end{bmatrix}$$

- The Chain Rule says that when taking the derivative, composition of functions becomes multiplication of matrices

$$D(g \circ f)(x) = Dg \left(\frac{f(x)}{A} \right) \cdot Df(x)$$

note Dg is evaluated at $f(x)$.

example

$$g(y_1, y_2) = y_1^2 + y_2^3. \text{ Let } h = g \circ f$$

$$f(x_1, y_1) = (x_1^2 x_2, x_1 + x_2) \quad Dg(y) = [2y_1, 3y_2^2]$$

\Rightarrow

$$Dh(x) = Dg(f(x)) \cdot Df(x)$$

$$= [2(x_1^2 x_2), 3(x_1 + x_2)^2]$$

$$= \begin{bmatrix} 2x_1 x_2 & x_1^2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 4x_1^3 x_2^2 + 3(x_1 + x_2)^2 \\ 2x_1^4 x_2 + 3(x_1 + x_2)^2 \end{bmatrix}$$

Now lets get back to the main job of

Understanding $\nabla \Phi$, which we want

to write using composition of functions
and the Chain Rule

Lc

- For simplicity, let's work with just one piece of training data at first so

$$\Phi(m) = \|F(x_1, m) - y_1\|^2$$

- To think about this using the chain rule we treat it as the composition of functions

$$\alpha(z) = \|z\|^2, \quad \beta(a) = a - y_1$$

~~$\Phi(m)$~~

$$\text{So } \Phi(m) = \alpha \circ \beta \circ F(x_1, m)$$

$$\text{then } \nabla \Phi(m) = D\Phi(m) = D\alpha(\beta \circ F(x_1, m)) \cdot D\beta(F(x_1, m)) \cdot DF(x_1, m)$$

- The first two pieces are easy

$$\alpha(z) = z_1^2 + \dots + z_p^2, \quad D\alpha = 2[z_1, z_2, \dots, z_p] = 2z$$

$$D\beta = Id$$

• so we have

$$\nabla \Phi(y) = \alpha (F(x_{l-1}) - y_1) \cdot DF(x_{l-1})$$

• so the real work is in computing DF .

• Now recall $F = F_L \circ \dots \circ F_1$ and so

(omitting arguments) $DF = DF_L \dots DF_1$

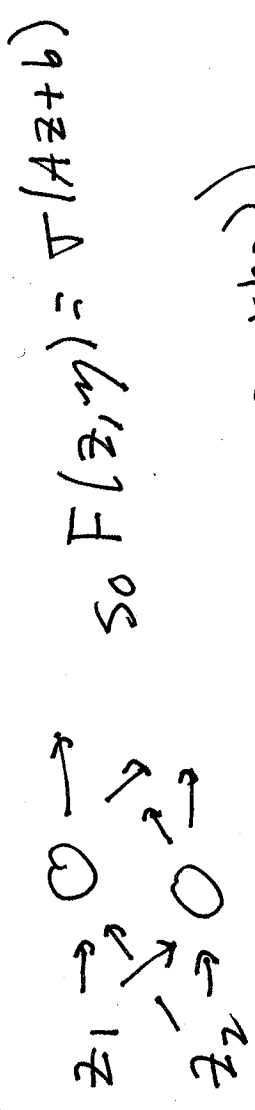
or more properly

$$DF(y) = DF_L(F_{L-1} \circ \dots \circ F_1(y)) DF_{L-1}(F_{L-2} \circ \dots \circ F_1(y)) \dots$$

$$\dots DF_1(y).$$

- now the $F_0 \circ \dots \circ F_1(y)$ are computed when we propagate x_1 through the network so the real issue is DF_j .

To simplify even further, we focus on a layer with just two neurons and two inputs



so $F(z, y) = \sigma(Az + b)$

$= (\sigma(A_{11}z_1 + A_{12}z_2 + b_1), \sigma(A_{21}z_1 + A_{22}z_2 + b_2))$

recalling that $\eta = A_{11}z_1 + A_{12}z_2 + b_1$ $\sigma_1 = A_{21}z_1 + A_{22}z_2 + b_2$

and letting $q_1 = A_{11}z_1 + A_{12}z_2 + b_1$ $q_2 = A_{21}z_1 + A_{22}z_2 + b_2$

$$DF = \begin{bmatrix} \sigma'(q_1)z_1 & \sigma'(q_1)z_2 & 0 & \dots & \sigma'(q_1)0 \\ 0 & \sigma'(q_2)z_1 & \sigma'(q_2)z_2 & \dots & 0 \end{bmatrix} \underbrace{\frac{\partial F}{\partial b_j}}$$

• This looks complicated, but is easy to

compute since each term like $A_{11}z_1 + A_{12}z_2 + b_1$

has been computed already while forward propagating

x_1 through the net

• Now we have a huge formula.

$$\nabla \Phi(y) = 2(F(x_1 - y) - y_1) DF_L DF_{L-1} \dots DF_1$$

and how do we compute it..?

• The answer is back propagation which means

from the left to the right in the formula.

which is from the output to the input in

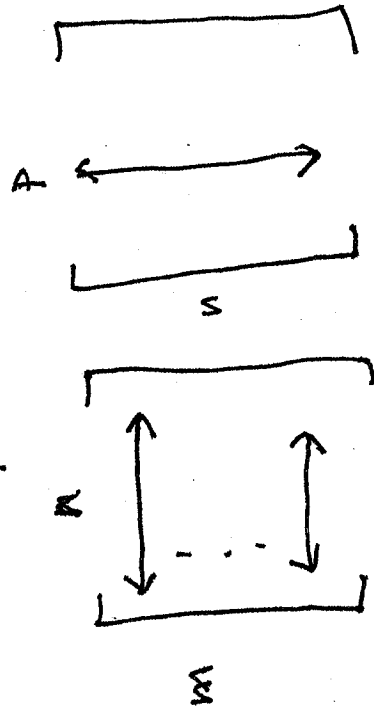
the net.

(10)

Let's see why that is by a simple example

First, if A is $m \times n$ and B is $n \times p$
how many multiplications are needed to

compute AB ?



each row of A
is n multiplications
and $n-1$ additions (so
we just track the mult)

There are n rows in A and p cols. in B so

$$\text{TOT \# MULT} = mnp$$

Now, in analogy to our formula for $\Delta\Phi$,

We examine ABC where

- A is 1×10^0
- B is 100×10^0
- C is 100×10^1

Then (AB) requires $1 \cdot 100 \cdot 100 = 10^4$ and is ~~100~~ $10^4 \times 100$

So (ABC) requires ~~100~~ $100 \times 100 \times 100 = 10^6$
So compuy from he left costs $10^4 + 10^6 = 2 \times 10^6$

on the other hand (BC) requires $100 \cdot 100 \cdot 100 = 10^6$ and is 100×10^0

So A(BC) requires $1 \cdot 100 \cdot 100 = 10^4$ so computing

from the right costs $1 \cdot 01 \times 10^6$
or about 50 times more than from the left

The difference gets even more pronounced with larger matrices.

Our whole discussion was based on a single input of training data, when there are N pieces it formally looks the same but ~~the~~ things become vectors and matrices. We save for the HW.

There is lots more to say even about just deep, feed forward nets, but we make just two more points.

• We have been using the quadratic, or least squares loss function. More common is to

use the cross entropy

$$C(m) = -\frac{1}{p} \sum_{i=1}^N y_i \ln(F(x_i, m)) + (1 - y_i) \ln(1 - F(x_i, m))$$

(after a normalization)

• There is a trick to make $Ax + b$ into

a single matrix multiplication. Let $\hat{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$

$$\hat{A} = \begin{bmatrix} 1 & 0 \\ b & A \end{bmatrix}$$

and augment the input x so the 2nd components become $\hat{A}\hat{x} = \begin{bmatrix} 1 \\ Ax + b \end{bmatrix}$ so the 2nd components become $Ax + b$ and the first remains 1

and note $\nabla_{\mathbf{r}}(1) = \nabla_{\mathbf{s}}(1) = \mathbf{1}$, so it stays 1 after activation.