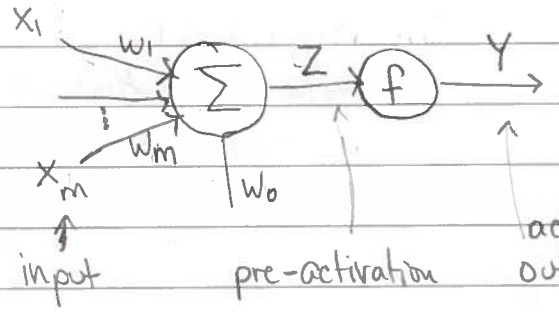
Neural networks

View 1: stochastic gradient descent for classification and regression with a potentially very rich hypothesis class

View 2: brain inspired network of neuron-like computing elements learn distributed representations

View 3: a method for building applications that make predictions based on huge amounts of data in very complex domains.

Basic element: "neuron" "unit" "node"



$$y = f\left[\sum_{j=1}^{m} x_j w_j + w_0\right] = f\left(w^T x + w_0\right)$$

{differentiable!}

f : "activation function"
w : "weights"
w_0 : "threshold" "bias"

Training

Given • a loss function $L(\text{guess}, \text{actual})$
• dataset $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$

Do (stochastic) gradient descent, adjusting W to minimize

$$\sum_i L\left(NN(x^{(i)}; W), y^{(i)}\right)$$

(We will discuss regularization next time.)

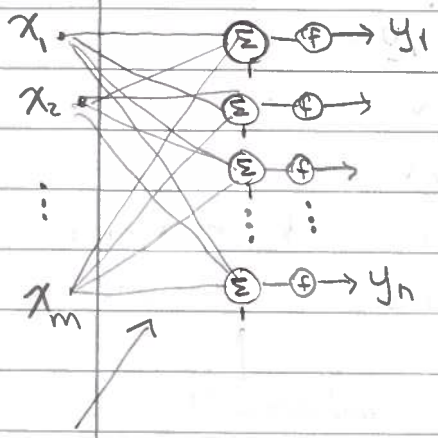
We already know two versions
- classifier w/ hinge loss
- regressor w/ quadratic loss

f = identity

L4-2

Networks!

A layer is several units, not connected to each other.



"fully connected" if all the inputs to each unit are the same

W, W_0

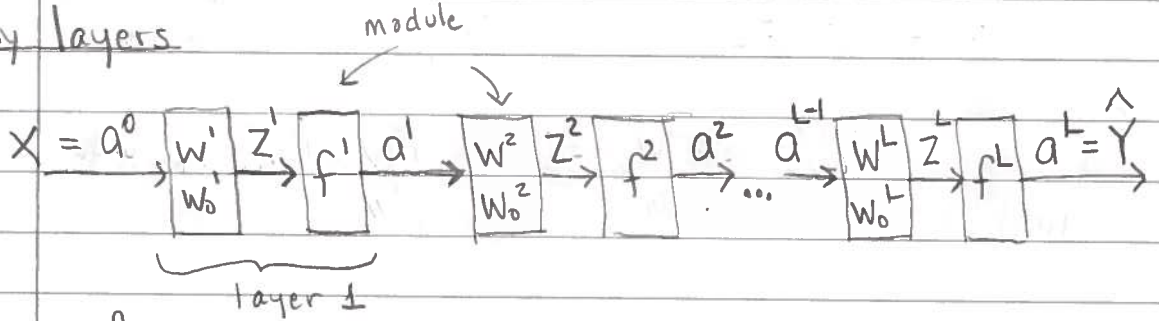
n units
n outputs
m inputs

W is an $m \times n$ matrix
W_0 is an $n \times 1$ column vec
X is an $m \times 1$ column vec
Y is an $n \times 1$ column vec

$$\underset{n\times 1}{y} = f(\underbrace{W^T X + W_0}_{n\times 1})$$

apply elementwise

Many layers



layer l

m^l inputs
$n^l = m^{l+1}$ outputs
$W^l : m^l \times n^l$
$W_0^l : n^l \times 1$
f^l : activation function

$$Z^l = W^{l^T} a^{l-1} + W_0^l \qquad (n^l \times 1)$$

$$a^l = f^l(Z^l) \qquad (n^l \times 1)$$

What the f!

What if we just let f be the identity? {Leaving off w0 for effect}

$$Y = a^L = W^{L^T} a^{L-1} = W^{L^T} W^{L-1^T} \cdots W^{1^T} X$$

So

$$Y = W^{\text{total}} X$$

Having all those layers did not change the representational capacity of the network: non-linearity is crucial.

Choices for f

$$\text{step}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

derivative is everywhere 0 or undefined

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ x & \text{otherwise} \end{cases} = \max(0, z)$$

"rectified linear unit"

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

sigmoid logistic

interpretable as a probability

tanh(z)

Could be anything, but these are the common ones, esp ReLU in internal ("hidden") layers and σ on output.

$$\text{Softmax}(z) = \begin{bmatrix} e^{z_1} / \sum_i e^{z_i} \\ \vdots \\ e^{z_n} / \sum_i e^{z_i} \end{bmatrix}$$

interpretable as a probability distn

↑ 1-hot encoding

L4-4

Loss for classification : NLL

An alternative to hinge loss which

- is okay without regularization (but we may still need it)
- extends nicely to multi-class

Assume a softmax output layer

$$P\left(\begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}, \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}\right) = a_i \text{ where } y_i = 1$$

$$= \prod_{i=1}^{n} a_i^{y_i}$$

probability assigned to the correct output

prob dist activations — one-hot training label — product

will be 1 when $y^i = 0$, a_i when $y^i = 1$

One view is that we would like to maximize the probability our network assigns to the whole dataset

$$\prod_{j=1}^{\text{dataset size}} \prod_{i=1}^{n \leftarrow \text{output dim}} a_i^{(j)\, y_i^{(j)}}$$

We can maximize the log of this function instead because log is monotonic. Log is

$$\sum_{j=1}^{\text{dataset size}} \sum_{i=1}^{\text{output dim}} y_i^{(j)} \log a_i^{(j)}$$

Hints at a loss function (which we want to minimize)

$$NLL(a, y) = -\sum_{i=1}^{\text{output dim}} y_i \log a_i$$

negative log likelihood
log loss
Cross entropy

L4-6

Modules with weights have to be able to compute

input, $\partial L/\partial$output $\rightarrow$ $\partial L/\partial$weights

Training

Initializing W is important

- Don't make magnitude too big for sigmoid / tanh activation fns
- Don't set to 0
  - no gradient for relu
  - generally better to randomize for symmetry breaking
- Rule of thumb:

$$W_{ij}^{l} \sim \text{Gaussian}(0, 1/m^{l})$$

drawn from prob dist'n; 0 mean; variance

Magnitude of Z will be independent of # of inputs

$W_{ij}^{l} \sim G(0, 1/m^{l})$ $\quad W_{0j}^{l} \sim G(0,1)$ $\quad$ or $W_{0j}^{l} = 0$

for $t = 1$ to T

  i = random draw from $\{1 .. n\}$ (n = # of data pts)

  $a^0 = x^{(i)}$

  for l from 1 to L:

    $z^l = W^{lT} a^{l-1} + W_0^l$

    $a^l = f^l(z^l)$

  loss $= L(a^L, y^{(i)})$

  $dL/dz^L = (\partial L/\partial a^L)(\partial a^L/\partial z^L)$

  $dL/dW^L = dL/dz^L \cdot dz/dW^L$

  for l from $L-1$ downto 1:

backprop:

    $dL/dA^l = dL/dZ^{l+1} \cdot dZ/dA^l$

    $dL/dZ^l = dL/dA^l \cdot dA/dZ^l$

weight grad:

    $dL/dW^l = dL/dZ^l \cdot dZ/dW^l$

    $dL/dW0^l = dL/dZ^l \cdot dZ/dW0^l$

sgd:

    $W^l = W^l - \eta(t)\, dL/dW^l$

    $W_0^l = W_0^l - \eta(t)\, dL/dW0^l$

* More about this later

$dL/dW0^L = dL/dz^L \cdot dz/dW0^L$

$W^L = W^L - \eta(t)\, dL/dW^L$

$W_0^L = W_0^L - \eta(t)\, dL/dW0^L$

Error backpropagation : backprop

To do SGD, focus on loss with respect to 1 (x, y)

Need to compute

$$\nabla_W \text{Loss}(NN(x; W), y)$$

↑ all the W_1^l, W_0^l

Seems terrifying, but actually easy and cool!

The chain rule is our hero

We will grossly abuse calc notation

Start with W^L.

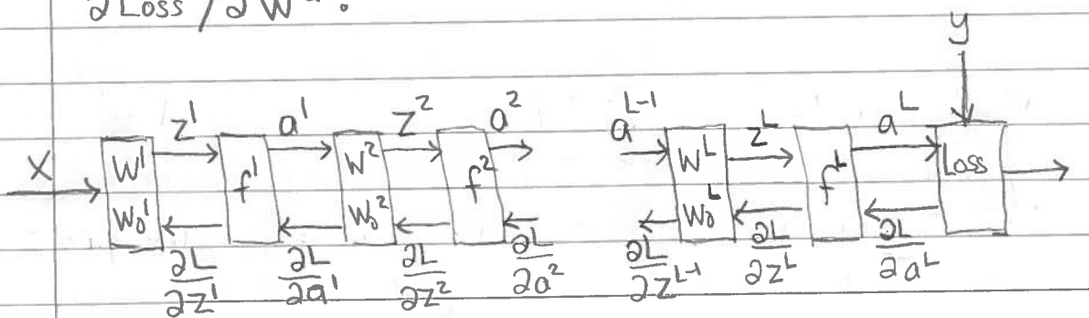
$$\frac{\partial \text{Loss}}{\partial W^L} = \frac{\partial \text{Loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial W^L}$$

↗ depends on loss function; ↑ $f^{L\prime}$; ↑ depends on a^{L-1}

$$\frac{\partial \text{Loss}}{\partial W^1} = \underbrace{\frac{\partial \text{Loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \cdots \cdot \frac{\partial a^2}{\partial z^2}}_{\partial \text{Loss}/\partial z^2} \cdot \frac{\partial z^2}{\partial a^1} \cdot \frac{\partial a^1}{\partial W^1}$$

$\partial \text{Loss}/\partial a^1$

So, we can do a systematic backward propagation of errors (I like to think of it as blame) to compute all the $\partial \text{Loss} / \partial W^l$:



Every module has to be able to compute

forward : input → output

backward : input, output, $\partial L / \partial \text{output}$ → $\partial L / \partial \text{input}$

L 4-7 (do NLL here)

Loss functions make assumptions about the type of input they're getting - so in our code we assume that f^L (activation function of last layer) and L (loss) go together:

Loss	f^L	
squared	linear	
hinge	linear	
NLL_1	sigmoid	(in homework)
NLL	softmax	

It turns out that directly computing

$$\frac{\partial L}{\partial z^L}$$ ~~is usually easier than separately doing~~

$$\frac{dL}{da^L} \text{ and } \frac{da^L}{dz^L}$$

especially for NLL. So we will ask our software implementation of a loss function to provide a backward method that computes $\partial L/\partial z$ directly.

The shape of things

What is $\partial Loss/\partial W^l$?

A matrix of the same shape as W^l $(m^l \times n^l)$ with entries: $\partial Loss/\partial W^l_{ij}$

Similarly:

- $\partial Loss/\partial a^l$ is a vector $(n^l \times 1)$ with entries $\partial Loss/\partial a^l_i$
- $\partial Loss/\partial z^l$ is a vector $(n^l \times 1)$ with entries $\partial Loss/\partial z^l_i$