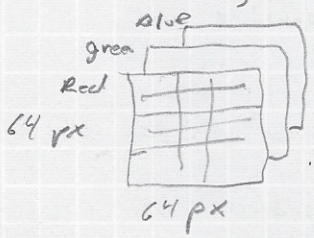
Neural Networks and Deep Learning
Week 2 Neural Networks Basics

Binary Classification

Have an input image and an output label 1 or 0, cat or non cat

y

eg. Computer stores 3 matricies corresponding to the Red, Green, and Blue channels of each image.



→ unroll these into a single feature vector x

assuming 64px × 64px image our feature vector has $64 \times 64 \times 3 = 12288$

$n = n_x = 12288$

$x \rightarrow y$

Notation

a single pair (x, y) $\quad x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$

m training examples: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$

$m = m_{train}$ = # examples $\quad m_{test}$ = # test examples

to put all of the traing examples into a more compact notation, we've going to take the training set inputs and stack them in columns to define matrix X

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \uparrow n_x \downarrow$$

$\leftarrow m \rightarrow$

has m columns

number of rows is equal to the number of values for each example of your input data

This convention makes implementation in python much easier.

$X \in \mathbb{R}^{n_x \times m}$ $\quad X.shape = (n_x, m)$

It is also helpful to stack your outputs $\quad Y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$

$Y \in \mathbb{R}^{1 \times m}$ $\quad Y.shape = (1, m)$

Logistic Regression

A learning algorithm you use when the the output label, Y, is either 0 or 1, so a binary classification problem.

Given input feature vector X, want $\hat{y}$ = your estimate of Y

You want $\hat{y}$ to be the probability of the chance that Y is equal to one given the input features X.

$$\hat{y} = P(y=1|x)$$

$X \in \mathbb{R}^{n_x}$

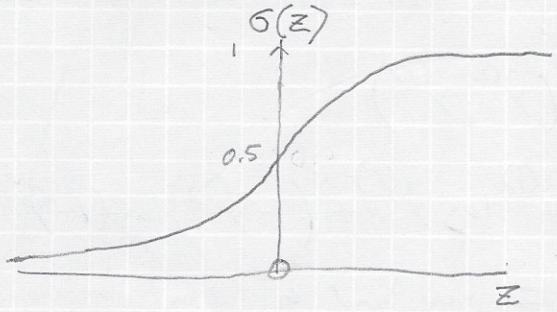
Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

(X and w are n_x dimensional vectors, b is a real number)

Output: $\hat{y} = ?$ how do you get $\hat{y}$

One option would be $= w^T x + b$, which is linear regression, but this isn't a good option for binary classification because you want $\hat{y}$ ("Y hat") to be the chance that Y is equal to one. So $\hat{y}$ should be between 0 and 1 and it's difficult to enforce that because w transpose X can be much bigger than one or it can be negative, which doesn't make sense for probability.

Output $\hat{y} = \sigma(w^T x + b)$ ← sigmoid function



Sigmoid of z crosses the vertical axis at 0.5

Sigmoid function:

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

notice:
- if z is very large e^{-z} will be close to zero, so $\frac{1}{1+0} = 1$
- if z is very small e^{-z} or a large negative number e^{-z} is a very big number so $\frac{1}{1+\text{big number}} \approx 0$

Next, in order to train the parameters w and b you need a cost function.

Logistic Regression Cost Function

Recap:

For a single training example $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$ (want your prediction to be close to ground truth)

Superscript (i) means the i^{th} training example.

we could define the loss in terms of the square error, so:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y}-y)^2$$

But in logistic regression people dont usually do this because the optimization function becomes non-convex ∿ where you have multiple local optimum so gradient descent might not find the global optimum.

Loss function is a measure we define to measure how good $\hat{y}$ is when the true label is y.

In L.R. this is the loss function we use which does give a convex ∪ shape that will perform well in gradient descent:

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$$

You want this to be as small as possible.

If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ ← (because second term is zero (1-1))
want $\log \hat{y}$ to be large, want $\hat{y}$ large

If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ ← first term will be zero
want $\log 1-\hat{y}$ to be large, want $\hat{y}$ small

Because we are using sigmoid $\hat{y}$ will never be greater than 1
This is one of many functions that if $\hat{y}=1$ we try to make it very large, and if zero we try to make it really small.

Loss = with respect to one training example
Cost = measures how you are doing on the entire training set (average of losses)

Cost function:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

expanding with loss function

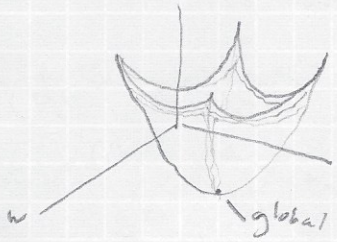
$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

We're going to try to find parameters w and b that minimize the cost function. $J(w,b)$

Gradient Descent Algorithm

Trains ("learns") the parameters w, b on your training set. By finding w, b that minimise $J(w,b)$

3D plot visualizes gradient decent with w as a single $\mathbb{R}$ b as a single $\mathbb{R}$



the height of the surface is the cost $J(w,b)$ at a certain point

the cost function is some surface above the axes w b.

global optimum, the minimum of cost function

so to find a good value for w, b we'll initialize them to some value For logistic regression almost any initialization method works, Usually you initialize to zero, random also works.

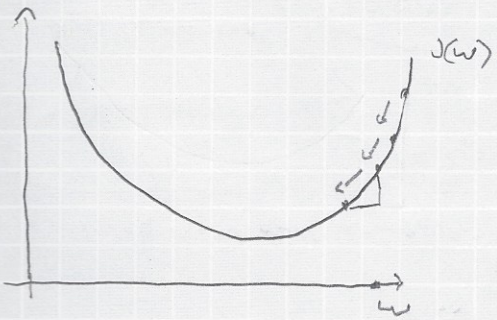
G.D. works by starting at the initialization and stepping in the direction of greatest descent, the steepest downhill direction. Then stepping further iterations down, until you converge on something close to the global optimum.

Repeat {

$$w := w - \alpha \frac{dJ(w)}{dw}$$

(α: learning rate)

(in code this derivative term will be written just "dw")



just looking at w gradient decent takes a step down as per the slope (the derivative) at that point in steps per the learning rate

"dw" in code

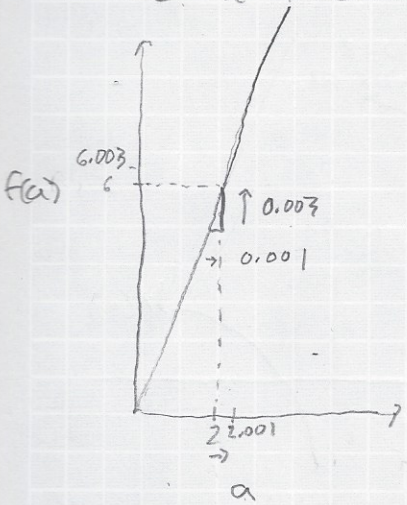
$J(w, b)$

$$w := \alpha \frac{dJ(w,b)}{dw}$$

$$b := \alpha \frac{dJ(w,b)}{db}$$ } "db" in code

Derivatives

we want to gain an intuitive understanding of derivatives.
we have a function $f(a) = 3a$, a straight line



$a = 2$	$f(a) = 6$
$a = 2.001$	$f(a) = 6.003$
$a = 5$	$f(a) = 15$
$a = 5.001$	$f(a) = 15.003$

if you nudge a over a little bit goes up three times as much so we'll say that the slope (derivative) of $f(a)$ at $a=2$ is 3, at $a=5$ also 3

slope is friendly term, derivative is scary sounding term

height/width of our triangle $\frac{0.003}{0.001}$

$\frac{df(a)}{da} = 3$ ← This says that the slope of $f(a)$ if you nudge a just a little bit is 3

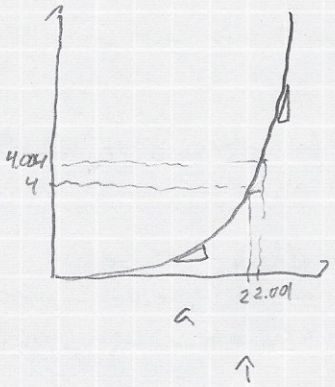
can also write this as $\frac{d}{da} f(a)$

all this says is that, if a nudge a a little bit I expect $f(a)$ to go up by three times as much

- the nudge you give in the formal definition is smaller than this 0.001, its actually infinitesible small amount, and infinitly small amount.

On a straight line, we can see that the derivative is the same everywhere, it doesn't change.

More Derivatives



$f(a) = a^2$

$a = 2$ $\quad f(a) = 4$

$a = 2.001$ $\quad f(a) = 4.004$ — goes up by 4x

$a = 5$ $\quad f(a) = 25$

$a = 5.001$ $\quad f(a) = 25.010$ — goes up by 10x

$\frac{d}{da} f(a) = 4 \quad$ at 2

$= 10 \quad$ at 5

↑ the ratio of the $\frac{\text{height}}{\text{width}}$ triangles are different at different points on the curve

Calculus textbooks list the formulas for various functions. Taking a Calc class will take you through the proofs that these are the correct derivative formulas for the respective functions. Here we're just going to give them.

$$\frac{d}{da} f(a) = \frac{d}{da} a^2 = 2a$$

More examples

$f(a) = a^3 \quad \frac{d}{da} f(a) = 3a^2$

↳ $3 \times 2^2 = 12$

$a = 2 \quad f(a) = 8$

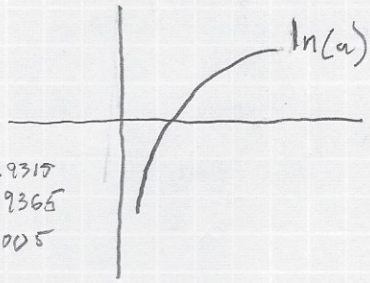
$a = 2.001 \quad f(a) = 8.012$

12x

$f(a) = \log_e(a)$

$= \ln(a)$

$\frac{d}{d(a)} = \frac{1}{a}$



$a = 2 \quad f(a) = 0.69315$

$a = 2.001 \quad f(a) = 0.69365$

$\Delta = 0.0005$

$\frac{1}{2}$ goes up by half as much

- The slope of the line varies depending on where you are for these functions.
- You can look up the formulas for the derivatives of common functions. you don't need to be able to prove them to train and use NN.

Computation Graph

Computations of a neural network are organized in terms of a forward pass (or forward propagation) in which we compute the output of the neural network, followed by a backward pass (or back propagation) step, in which we use to compute gradients (or compute derivatives.

The computation graph explains why it is organized this way.

E.g. Simple example $J(a,b,c) = 3(a+bc)$

u v J

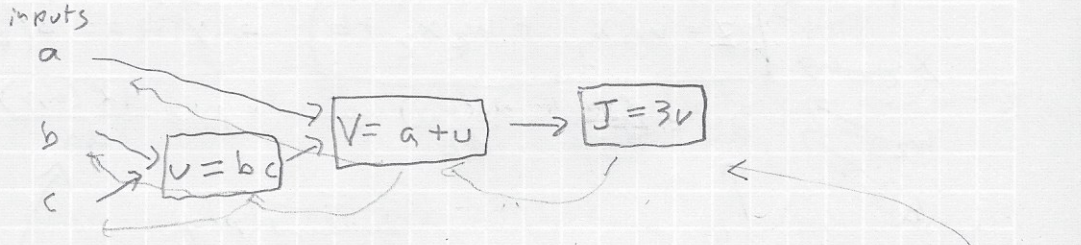
3 steps to solve

$u = bc$

$v = a + u$

$J = 3v$

Computation graph:



The computation graph comes in handy when you have some output variable (J) you want to optimize.

In logistic regression J will be the cost function.
When computing derivatives there will be a left to right pass in the direction of the arrows

Derivatives with Computation graph

$\frac{dJ}{dv} = ? = 3$ in the example above "dv"

$a \xrightarrow{\text{affects}} v \xrightarrow{\text{affects}} J$

$\frac{dJ}{da} = ? = 3 = \frac{dJ}{dv} \cdot \frac{dv}{da}$ ← chain rule in calculus

if you change a how much does it affect the final output J?

$\frac{dv}{da} = 1$

by having computed $\frac{dJ}{dv}$ it helps you compute $\frac{dJ}{da}$

when coding you want a final output variable $\frac{d\text{FinalOutputVar}}{d\text{var}}$
you always do this so we just call it "dvar"

$\frac{dJ}{du} = 3 = \frac{dJ}{dv} \cdot \frac{dv}{du} = dv$

$3 \quad 3 \quad 1$

$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 6 = db$

2

(chain rule in calc)

- most efficient way to compute derivatives is to follow the right to left arrows and use the derivatives you have already computed to help you by using calc's chain rule

$\frac{dJ}{dc} = \frac{dJ}{dv} \cdot \frac{dv}{dc} = 9$

3×3

Logistic Regression Gradient Descent

Recap:

$$z = w^T x + b$$

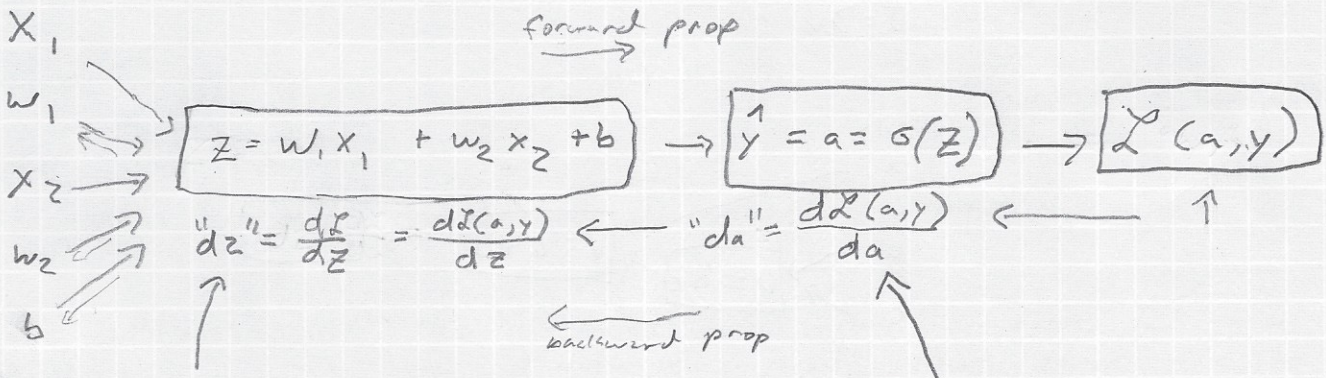
$$\hat{y} = a = \sigma(z)$$

$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

*with respect to one training example

For this example let's say we have only two features x_1 and x_2

Computation graph



In L.R. we modify params w + b in order to reduce this loss.

The formulas you need to calculate the derivatives will be provided in this course. The derivative of our loss function is

$$-\frac{y}{a} + \frac{1-y}{1-a} =$$

$$= a - y$$

Again, you can derive these from the formulas above using calculus, but we're just going to look them up.

The final step in backprop is compute how much you want to change (update) w and b. Similarly, we can show that these are the equations for dw and db.

$$\frac{\partial \mathcal{L}}{\partial w_1} = "dw_1" = x_1 \cdot dz$$

$$"dw_2" = x_2 \cdot dz$$

$$"db" = dz$$

$$w_1 := w_1 - \alpha\, dw_1$$

$$w_2 := w_2 - \alpha\, dw_2$$

$$b := b - \alpha\, db$$

Then you can perform the updates.

This is all just for one example.

Gradient Descent on m examples

Recap:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y)$$

$$\rightarrow \quad a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

$$dw_1^{(i)}, \; dw_2^{(i)}, \; db^{(i)}$$

$$(x^{(i)}, y^{(i)})$$

The above is for a single example denoted by superscript (i). Just as the overall cost function is an <u>average</u> of the losses, the derivative of w_1 with respect to the cost function is also going to the <u>average</u> of derivatives with respect to w_1 of the individual loss terms.

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})}_{dw_1^{(i)} - (x^{(i)}, y^{(i)})}$$

Symbol just means partial derivative because it is one of several terms

compute these derivatives and average them

Gives you the overall gradient that you can use to implement gradient descent

To wrap this all up into an algorithm that you can use:

Initialize variables

$J = 0, \; dw_1 = 0, \; dw_2 = 0, \; db = 0$ ← we will use these as accumulators

Use a for loop over the training set and compute derivatives with respect to each training example and then also add them up.

For $i = 1$ to m:

$$z^{(i)} = w^T x^{(i)} + b$$

$$a^{(i)} = \sigma z^{(i)}$$

$$J \mathrel{+}= -\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log (1 - a^{(i)})\right]$$

$$dz = a^{(i)} - y^{(i)}$$

$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

$$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$

$$db \mathrel{+}= dz^{(i)}$$

$n = 2$ doing this assuming only two features. Otherwise you would do this for every feature

Divide by m

$J \mathrel{/}= m$

$dw_1 \mathrel{/}= m; \; dw_2 \mathrel{/}= m; \; db \mathrel{/}= m$

So then to implement one step of gradient descent, you then

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

everything in this example would then get run again multiple times to perform multiple steps of gradient descent.

However, the algorithm above uses two For-loops which is very inefficient. Vectorization techniques help you avoid these. This is really important now in the era of big data.