W2 - Neural Network Basics

Tuesday, June 2, 2020 12:06 AM

Binary Classification:

Logistic regression is an algorithm for binary classification

How does our computer store images?

- If we have a 64 x 64 pixel image, our computer stores 3 red-green-blue matrices that are 64 x 64 (rows x columns) corresponding to these colors pixel intensity value.

- a feature vector made from these matrices would be a single column with 64 x 64 x 3 = 12288 values.

dimension of the feature vector
n_x = 12288 (represents the input features x)

- In a binary image classifier, our goal is to be able to input a feature vector (x) and get a 1 or a 0 as an output, which represents cat or non-cat respectively.

Unrolling an image into a feature vector means converting the RGB values into a single feature vector

Notation

- Single training example: (x, y)
  - where $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
    - element of an n_x-dimensional feature vector

- m training examples (training set):
  $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
  - $m = m_{train}$
    - m_{test} = # test examples

- alternative way to write matrix:

Capital → $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ n_x ← feature vectors

m ← # of training examples

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

- in Python this comes out to be:
  X.shape (n_x, m)

$n_x \times m$ dimensional matrix

- abbreviated for the output:

$$Y = [y^{(1)}\ y^{(2)}\ y^{(3)}, \ldots, y^{(m)}]$$

$$Y \in \mathbb{R}^{1 \times m}$$

↳ in Python: Y.shape = (1, m) ~ columns matrix

Logistic Regression:

- Given x, want $\hat{y} = P(y=1|x)$
  ↳ if x is the cat picture, we want $\hat{y}$ to tell us the probability that it's a cat picture.

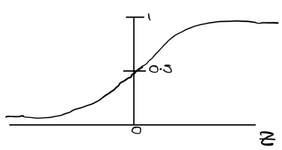
- $x \in \mathbb{R}^{n_x}$
  ↳ x is an n-dimensional vector given that:
  parameters: $w \in \mathbb{R}^{n_x}$ (dimensional vector) ~ weighting, $b \in \mathbb{R}$ ~ real-number (y-intercept) ~ bias

Given an input x and the parameters w and b how do we generate $\hat{y}$?

- traditional linear regression: $\hat{y} = w^T x + b$
  ↳ this does not work for binary classification

because you want $\hat{y}$ to be the chance that y = 1 ~ we want a value between 0 and 1
↳ this value can be greater than 1 or negative which doesn't make sense for a probability

- sigmoid function: σ ~ for binary classification



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

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

↳ if z is large $\sigma(z) = \dfrac{1}{1+0} \approx 1$ ~ the exponential factor decays

↳ if z is small (~ very negative) $\sigma(z) = \dfrac{1}{1+\text{big \#}} \approx 0$ ~ the exponential factor grows making the denominator large

- when programming a neural net, it is better to keep w and b as separate parameters and not in the same vector (as in other

(course)

- $\hat{y} = \sigma(w^T x + b)$

Logistic Regression cost function

- a cost function is used to train the parameters w and b of ~~linear~~ logistic regressions

- $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$ $\quad \sigma = \frac{1}{1+e^{-z}}$

  (i) describes data associated with the i-th training example

- Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$
  - this is not done in logistic regression because when you come to learn the parameters, you learn that the optimization problem becomes non-convex.
  - we end up with multiple local optima and no true global optima

- the loss function is used to define how good our $\hat{y}$ is when the true label is y
  - it measures the performance of our predictions on the given task

loss function used in logistic regression:

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

- it gives us an optimization problem which is convex
- Why this makes sense?
  - If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$
    - want $\log \hat{y}$ large, want $\hat{y}$ large
    - to reduce the loss
  - If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$
    - want $\log 1-\hat{y}$ large, want $\hat{y}$ small so that the loss is near zero

- Cost function for entire training sets:

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

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

- the cost function is the cost of your parameters

Correction 4:15

We want $-y \log(y\text{-hat})$ as small as possible

- when training with set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we want that our prediction $\hat{y}^{(i)}$ approximates to the true label $y^{(i)}$

- for the loss function we don't utilize the square error because it results in a non-convex shape where gradient descent can't find the global optima

- we want this error to be as small as possible

- $\ln(1) = 0$
- $\ln(0.0001)$ = large loss
  - negative
- $\hat{y}$ is to make loss positive as $\hat{y}$ range 0–1 which would give negative $\log$

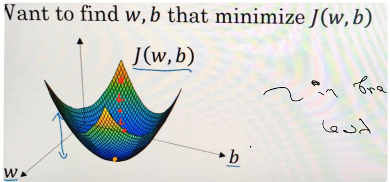
- measures how well we do for the entire training set

- in training the model, we are trying to find parameters w and b such that the cost function is minimized

- distinction: The loss function computes the error for a single training example; the cost function is the average loss function for the whole training set.

Gradient Descent

- the gradient descent algorithm is used to train / learn the b and w parameters on our training set. ~ making the J(w, b) cost function as small as possible



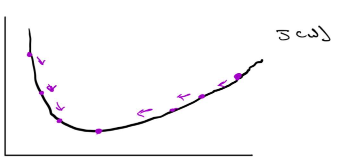
~ in practice, w can be higher dimensional but here it was been left as one for visualization.

we want to find the value (w, b) such that we minimize the cost function J. (orange) (minimum)

(this example is a straight forward convex function)

- the fact that the shape of our cost function is convex, as seen above, is one of the huge reasons we use it for logistic regressions.

- for gradient descent we start at a point and work our way down until we reach the optimum value. steepest downhill direction (cause rate of change (derivative))



Repeat $\{ w := w - \alpha \cdot \frac{\partial J(w)}{\partial w} \}$ ~ updating

↳ we do this until the algorithm converges

~ dw will be representative of this term when writing code $w := w - \alpha dw$

α: learning rate: controls how big a step we take each iteration

for gradient descent $J(w,b)$

$w := w - \alpha \frac{\partial J(w,b)}{\partial w}$ ~ dw

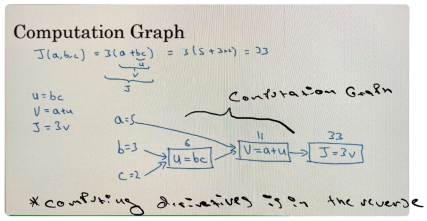
$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$ ~ db

* in code

* ∂ partial derivative symbol: is used when differentiating in respect to 2 variables eg.

Computation Graph

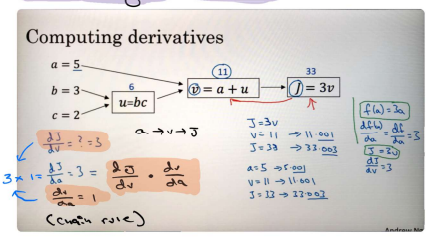
- the way neural networks are organized is a forward propagation step in which the outputs are calculated followed by a backward propagation step used to calculate gradients / derivatives to fine tune parameters.



* computing derivatives is in the reverse direction

right to left pass

Computing Derivatives

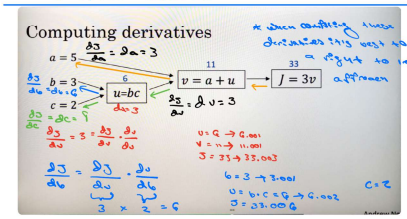


$\frac{dJ}{da}$ → "da" in code (syntax naming convention)

- By changing a, those changes get propagated to v and therefore to J

- A Final output variable (J) = dJ dvar = "d var" (variable)

* Lyman notation



Logistic Regression Gradient Descent

$\frac{da}{dz} = a(1-a)$

$x_1, w_1, x_2, w_2, b \rightarrow$ $z = w_1x_1 + w_2x_2 + b$ $\rightarrow$ $\hat{y} = a = \sigma(z)$ $\rightarrow$ $\mathcal{L}(a, y)$ ~ loss function (1 example)

$dz = \frac{dJ}{dz} = \frac{dJ}{da} \cdot \frac{da}{dz}$ → $a - y$

$\frac{d\mathcal{L}(a,y)}{d(a)} = -\frac{y}{a} + \frac{1-y}{1-a}$ ~ using the fractions of logs

~ look at derivation in towards DS

$\frac{da}{dz} = a(1-a) = \sigma(z) \cdot (1 - \sigma(z))$

$\frac{dL}{da} \cdot \frac{da}{dz} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \cdot (a(1-a)) = -y \cdot (1-a) + a \cdot (1-y)$

$= -y + ay + a - ay$

$= -y + a$

$dw_1 = x_1 \cdot dz$
$dw_2 = x_2 \cdot dz$
$db = dz$

→

$w_1 := w_1 - \alpha\, dw_1$
$w_2 := w_2 - \alpha\, dw_2$
$b := b - \alpha\, db$

$dw_1 = \frac{dz}{dw_1} \cdot \frac{dL}{dz} = x_1 \cdot dz$

$dw_2 = \frac{dz}{dw_2} \cdot \frac{dL}{dz} = x_2 \cdot dz$

$db = \frac{dz}{db} \cdot \frac{dL}{dz} = dz$

- In logistic regression we want to tweak these parameters (w, b) in order to reduce the loss function ~ (w_1, w_2, b)

Gradient descent on m examples

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

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

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$ ~ one training example

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

$dw_1^{(i)} - (x^{(i)}, y^{(i)})$ ~ this is what we computed before

- this would give us the overall gradient when we can
- the overall gradient $\frac{\partial J}{\partial w_1}$ is given by averaging the loss function for the m different training examples

use to implement gradient descent

Concrete algorithm of this...

$J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$ ~ these are being used as accumulators at the start ~ initialization

*execute for loop over training set (comments)

(1) for i = 1 to m ~ for loop over training set to calculate all derivatives for example and then add them up

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

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

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

$dz^{(i)} = a^{(i)} - y^{(i)}$ (in respect to one example)

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$
$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$] n = 2] a for loop required over n feature
$db \mathrel{+}= dz^{(i)}$

(2)
$J = \frac{J}{m}$
$dw_1 = \frac{dw_1}{m}$
$dw_2 = \frac{dw_2}{m}$
$db = \frac{db}{m}$

we are computing averages so we divide by m

In the end... $dw_1 = \frac{\partial J}{\partial w_1}$

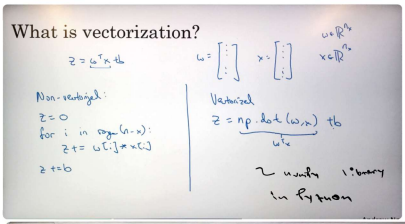
↳ concerning one loop of the gradient descent of the algorithm.

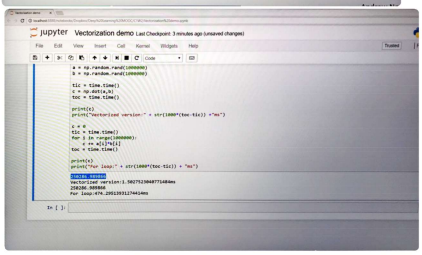
and $w_1 := w_1 - \alpha dw_1$
$w_2 := w_2 - \alpha dw_2$
$b := b - \alpha db$
} these will be updated in the end

* this concrete algorithm implements just one step in gradient descent so we would need to repeat everything here multiple times to take multiple steps of gradient descent.

* there are two weaknesses to this way of implementing the algorithm:
↳ two for loops required
(initial one and another one for features / parameters w_j)
↳ explicit for loops in deep learning makes algorithms run less efficiently
↳ vectorization techniques let us get rid of these explicit for loops

Vectorization





• avoid explicit for and if loops wherever possible.

Vectorizing logistic Regression's Gradient Computation

Vectorizing Logistic Regression:

making a prediction on the first example:

Vectorizing Logistic Regression:

making a prediction on the first example

forward:

$z^{(1)} = w^T x^{(1)} + b$

$a^{(1)} = \sigma(z^{(1)})$

• second example

$z^{(2)} = w^T x^{(2)} + b$

$a^{(2)} = \sigma(z^{(2)})$

} might have to do this m times for m training examples.

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x, m) dimensions (features, examples)

training data matrix

how do we compute all the z's in one step?

$[z^{(1)}\ z^{(2)} \ldots z^{(m)}] = w^T X + [b\ b \ldots b] = [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \ldots \quad w^{(T)} x^{(m)} + b]$ (z_1, z_2, z_m)

Z — row vector

$w^T X$: $[w^T x^{(1)}\ w^T x^{(2)} \ldots w^T x^{(m)}]$

$[b\ b \ldots b]$: 1xm vector — adding the bias of the layer to each computation (that's why they're all the same b)

* to implement this in numpy,

`Z = np.dot(w.T, X) + b`

b — broadcasting in python: transforms b into appropriate vector for computation.

how to compute $a^{(1)}$ through $a^{(m)}$,

$A = [a^{(1)}\ a^{(2)} \ldots a^{(m)}]$

* implemented in the programming assignment

Vectorizing Logistic Regression's Gradient Computation

$dz^{(1)} = a^{(1)} - y^{(1)}$

$dz^{(2)} = a^{(2)} - y^{(2)}$

} how do we compute this for all m training examples

$dZ = [dz^{(1)}\ dz^{(2)} \ldots dz^{(m)}]$ — 1xm matrix

$A = [a^{(1)} \ldots a^{(m)}]$ $\quad Y = [y^{(1)} \ldots y^{(m)}]$

↓

$dZ = A - Y$

Programmatic Implementation

loops before

$dw = 0$

$dw \mathrel{+}= x^{(1)} dz^{(1)}$

$dw \mathrel{+}= x^{(2)} dz^{(2)}$

⋮

$dw \mathrel{/}= m$

$db = 0$

$db \mathrel{+}= dz^{(1)}$

$db \mathrel{+}= dz^{(2)}$

⋮

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

$db \mathrel{/}= m$

previous implementation (for loop over m examples)

Non vectorized implementation

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$= \frac{1}{m} \cdot np.sum(dz)$$ Python implementation

$$dw = \frac{1}{m} \cdot X \cdot dz^T = \frac{1}{m} \cdot \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix} = \frac{1}{m} \left[x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)} \right]$$

$n_x \times 1$ — # of features; 2 vectors

Code Implementation of vectorization

$$Z = w^T X + b = np.dot(w.T, X) + b$$

$$A = \sigma(Z)$$

$$dZ = A - Y$$

$$dw = \frac{1}{m} X dZ^T$$

$$db = \frac{1}{m} np.sum(dZ)$$

all of A, Y, Z are capitalized as dictated by convention

$$w := w - \alpha dw$$
$$b := b - \alpha db$$ } update to the weighting

*this is a single iteration of gradient descent for logistic regression

↳ i.e. we need multiple iterations of gradient descent, we still require multiple for loops over the # of iterations