Neural Network Basics

Binary Classification

Case: Input an image, Output a label.

Notation

One training example: (x, y), $x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$.

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

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & ... & x^{(m)} \\ | & | & & | \end{bmatrix}$$

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

$Y = [y^{(1)}, y^{(2)}, ..., y^{(m)}]$

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

call `X.shape` and `Y.shape` in `python`.

Logistic Regression

Given x, want $\hat{y} = P(y = 1|x)$.

$X \in \mathbb{R}^{n_x}, 0 \leq y \leq 1$.

Parameters: $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$. For the output $w^T X + b$.

In order to constrain $\hat{y}$ between 0 and 1, we need a sigmoid function $\sigma(w^T X + b)$.

Sigmoid Function: $\sigma(z) = \dfrac{1}{1 + e^{-z}}$.

The goal of logistic regression is **trying to learn parameters w and b so that $\hat{y}$ has become a good estimate of the chase of y being equal to one.** i.e. Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

Simplify $w^T X + b$ to $w^T X$ by adding $x_0 = 1$ and b to be the first element of w.

Loss(error) function: $L(\hat{y}, y)$. -> defined to single training example.

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

Cost function $J(w, b)$ -> defined to the entity of training examples.

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

The difference between the cost function and the loss function for logistic regression? -> The loss function computes the error for a single training example while the cost function is the average of the loss functions of the entire training set.

Gradient Descent

$J(w,b) = -\frac{1}{m}\Sigma_{i=1}^{m}(y^{(i)}log\hat{y}^{(i)} + (1-y^{(i)})log(1-\hat{y}^{(i)}))$.

Want to find w, b that minimize $J(w,b)$.

After initializing, iterating:

$Repeat\{w := w - \alpha\frac{dJ(w,b)}{dw}; b := b - \alpha\frac{dJ(w,b)}{db}\}$ until they converge.

Computation Graph: One step of backward propagation on a computation graph yields derivative of final output variable.

An example of 2-dimensional weight vector: (single training example)

$\{x_1, w_1, x_2, w_2, b\} \rightarrow z = w_1x + w_2x + b \Longleftrightarrow a = \sigma(z) \Longleftrightarrow L(a,y)$

$da = \frac{dL(a,y)}{da} = \frac{d(-(yloga + (1-y)log(1-a)))}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

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

$dw_1 = \frac{dL}{dw_1} = x_1dz$

$dw_2 = \frac{dL}{dw_2} = x_2dz$

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

Update:

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

One single step of Gradient Descent *PseudoCode*:

```
// Given:
// x_1, x_2, ..., x_m as m training examples, each of them has n dimensions
// called by x_i[j].
// alpha: learning rate
// y_1, y_2, ..., y_m as m labels, each of them is a real number.

J = 0, dw_1 = 0, dw_2 = 0, ..., dw_n = 0, db = 0
initialize w and b // w is as [w_1, w_2, ..., w_n]

for i = 1 to m
    z_i = w.T * x_i + b
```

```
    a_i = sigmoid(z_i)
    J += -(y_i * log(a_i) + (1 - y_i)*log(1 - a_i))
    dz_i = a_i - y_i
    dw_1 += x_i[1] * dz_i
    dw_2 += x_i[2] * dz_i
    ...
    dw_n += x_i[n] * dz_i
    db += dz_i
// average
J = J / m
dw_1 = dw_1 / m
dw_2 = dw_2 / m
...
dw_n = dw_n / m
db = db / m
// update
w_1 = w_1 - alpha * dw_1
w_2 = w_2 - alpha * dw_2
...
w_n = w_n - alpha * dw_n
b = b - alpha * db
```

Weakness: too many for-loops, which cause inefficiency. -> Use vectorization.

vectorization

The art of getting rid of explicit folders in the code. Why the explicit for-loop can be so slow?

`np.dot(w, x) + b`

GPU and CPU both have SIMD - "single instruction multiple data".

Modify the code snippet to the vectorized version.

```
J = 0
dw = np.zeros((n_x, 1))
db = 0

for i = 1 to m
    z_i = w.T * x_i + b
    a_i = sigmoid(z_i)
    J += -(y_i * log(a_i) + (1 - y_i)*log(1 - a_i))
    dz_i = a_i - y_i
    dw += np.dot(x_i, dz_i)
    db += dz_i
# ...
```

Or even better, getting rid of the outer for-loop as well.

$[z^{(1)}, z^{(2)}, ..., z^{(m)}] = w^T X + [b, b, ..., b] = [w^T x^{(1)} + b, w^T x^{(2)} + b, ..., w^T x^{(m)} + b]$

```
z = np.dot(w.T, x) + b
a = sigmoid(z)
```

The dimension of X: (n_x, m).

vectorizing Logistic Regression

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

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

$\dots$

$dZ = [dz^{(1)}, dz^{(2)}, ..., dz^{(m)}]$

$A = [a^{(1)}, a^{(2)}, ..., a^{(m)}]$

$Y = [y^{(1)}, y^{(2)}, ..., y^{(m)}]$

$dZ = A - Y$

$db = \frac{1}{m}\Sigma_{i=1}^{m} dz^{(i)} = \frac{1}{m} np.sum(dZ)$

$dw = \frac{1}{m} X dZ^T$ $(\, n \times 1 \; matrix)$.

```
Z = np.dot(w.T, x) + b
A = sigmoid(Z)
dZ = A - Y
dw = 1/m * np.dot(X, dZ.T)
db = 1/m * np.sum(dZ)
w = w - alpha * dw
b = b - alpha * db
```