Neural Network Step by Step - Part I

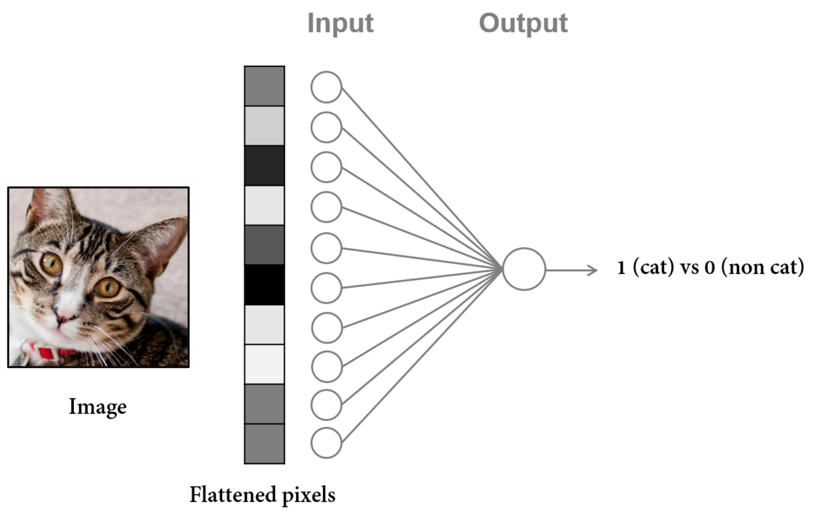
by Timothy (tq.wang93@hotmail.com)

First draft: Dec, 2020

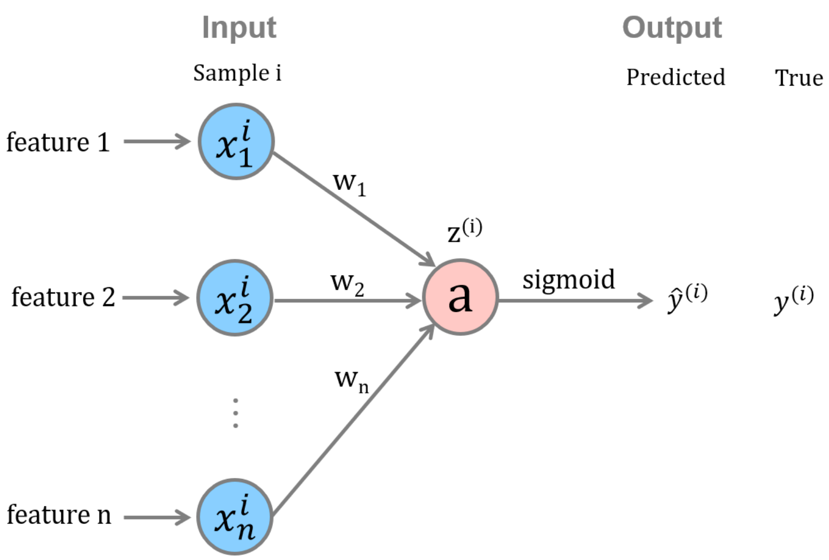
Last update: Dec, 2020

Logistic regression from scratch

In part I, we will work on the simplest neural network, which includes an ***input layer*** with ***n*** nodes (each representing a ***feature*** of the input) and ***one*** node in the ***output layer***. This type of neural network can be used to identify whether the item belongs to a certain category (Yes = 1; No = 0).

> For example, given a series of features $(x_1, x_2, ... x_n)$ of a picture, the model can judge whether it is a cat (Yes = 1; No = 0).
>
> 
>

We will train this neural network using ***m*** samples. Before showing codes, let's go through the calculation of ***feed-forward*** and ***backward propagation*** process.



Feed-forward

- **For sample i**

  Let's first define the vector representation of sample i and weight.

$$x^{(i)} = \begin{pmatrix} x_1^i \\ x_2^i \\ \vdots \\ x_n^i \end{pmatrix}_{\color{red}{n\times 1}} \qquad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}_{\color{red}{n\times 1}}$$

  We can then calculate z for sample i and represent it in a more concise way.

$$\begin{aligned} z^{(i)} &= x_1^i w_1 + x_1^i w_1 + \cdots + x_n^i w_n + b_i \\ &= (w_1, w_2, \ldots, w_n) \begin{pmatrix} x_1^i \\ x_2^i \\ \vdots \\ x_n^i \end{pmatrix} + b_i \\ &= w^T x^{(i)} + b_i \end{aligned}$$

- **For all samples**

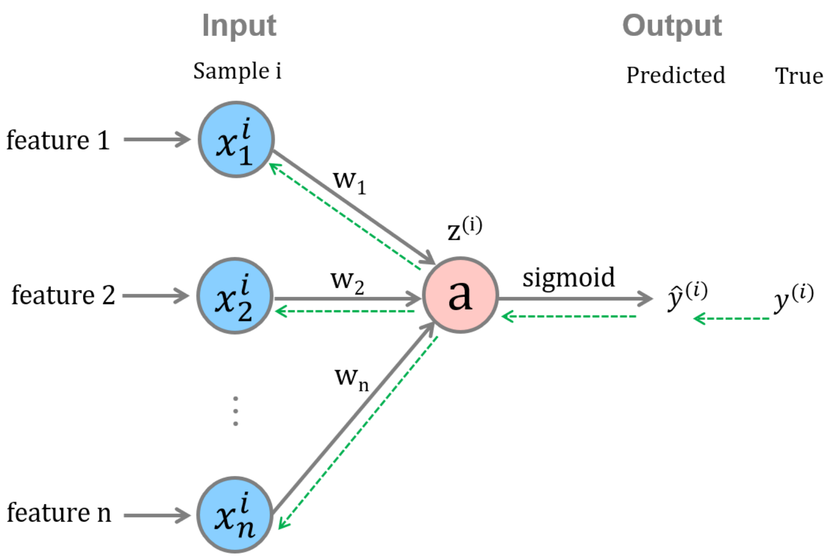
$$\begin{aligned} Z &= [z^{(1)}, z^{(2)}, \ldots, z^{(m)}]_{\color{red}{1\times m}} \\ &= [w^T x^{(1)} + b_1, w^T x^{(2)} + b_2, \ldots, w^T x^{(m)} + b_m] \\ &= w^T [x^{(1)}, x^{(2)}, \ldots, x^{(m)}] + [b_1, b_2, \ldots, b_m] \\ &= w^T X + b \end{aligned}$$

$$w^T = [w_1, w_2, \ldots, w_n]_{\color{red}{1\times n}} \qquad X = \begin{pmatrix} x_1^1 & x_1^2 & \ldots & x_1^m \\ x_2^1 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \vdots \\ x_n^1 & x_n^2 & \ldots & x_n^m \end{pmatrix}_{\color{red}{n\times m}}$$

$$\hat{Y} = sigmoid(Z) = [\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(m)}]_{\color{red}{1\times m}}$$

Back propagation

Back propagation starts by comparing the ***predicted value*** $\hat{y}$ to the ***true value*** y, which produces the **loss**. It then updates the weights w and bias b using ***gradient descent***. Through iterations, the weights and bias are optimized.



- **Loss function** $L(a, y)$

  Here, we use the ***cross entropy*** loss function. There are other options such as ***mean squared error***.

$$L(a, y) = -ylog\,a - (1-y)log(1-a)$$

- **Cost function** $J(a, y)$

  As defined, cost function is the ***mean*** of the loss of all samples.

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

- **Gradient descent**

  Based on chain rules,

$$\frac{\partial L(a,y)}{\partial w} = \frac{\partial L(a,y)}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

$$\frac{\partial L(a,y)}{\partial b} = \frac{\partial L(a,y)}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b}$$

  Note: the *sigmoid* function is

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

$$\begin{aligned}\frac{\partial L(a,y)}{\partial a} &= -y\,\frac{1}{a} - (1-y)\,\frac{1}{1-a}(-1) \\ &= -\frac{y}{a} + \frac{1-y}{1-a}\end{aligned}$$

$$\begin{aligned}\frac{\partial a}{\partial z} &= \frac{0 - 1 \cdot e^{-z} \cdot (-1)}{(1+e^{-z})^2} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\ &= \frac{1}{1+e^{-z}} \cdot (1 - \frac{1}{1+e^{-z}}) \\ &= a \cdot (1-a)\end{aligned}$$

  Therefore,

$$\begin{aligned}\frac{\partial L(a,y)}{\partial a} \cdot \frac{\partial a}{\partial z} &= (-\frac{y}{a} + \frac{1-y}{1-a})\,a(1-a) \\ &= -y(1-a) + a(1-y) \\ &= a - y\end{aligned}$$

  Moving on, for the partial derivatives of $z = w^T + b$ to w and b,

  - In the case of **sample i**, because

$$\frac{\partial z}{\partial w} = x_{n\times 1} \qquad \frac{\partial x}{\partial b} = 1_{1\times 1}$$

    we can get

$$\frac{\partial L(a,y)}{\partial w} = x_{n\times 1}(a-y)_{1\times 1}$$

$$\frac{\partial L(a,y)}{\partial b} = (a-y)_{1\times 1} \cdot 1_{1\times 1}$$

  - In the case of **m samples**,

$$\frac{\partial L(A,Y)}{\partial w_{n\times 1}} = X_{n\times m}(A-Y)^T{}_{m\times 1}$$

$$\frac{\partial L(A,Y)}{\partial b_{1\times 1}} = A - Y_{1\times m}$$
$$= np.sum(A-Y)_{1\times 1}$$

- **Updating weights and bias**

  The weight w and bias b are updated by subtracting the ***mean of cost*** multiplied by the ***learning rate*** α.

$$w \leftarrow w - \alpha \cdot \frac{1}{m} \cdot X(A-Y)^T$$

$$b \leftarrow b - \alpha \cdot \frac{1}{m} \cdot np.sum(A-Y)$$

- **Revisiting the cost function J**

  Usually, we print/plot the change of cost as a function of iterations.

$$\begin{aligned} J &= \frac{1}{m}\sum_{i=1}^{m}[-y log\, a - (1-y) log(1-a)] \\ &= -\frac{1}{m}\sum_{i=1}^{m}[y log\, a + (1-y) log(1-a)] \\ &= -\frac{1}{m}[Y log\, A + (1-Y) log(1-A)] \\ &= -\frac{1}{m} np.sum[Y log\, A + (1-Y) log(1-A)] \end{aligned}$$

  Note: the $Y log\, A + (1-Y)log(1-A)$ is calculated using dot product.

Coding a logistic regression

Code: sigmoid function

$$a = \frac{1}{1+e^{-z}}$$

```
# define sigmoid function
def sigmoid(z):
    a = 1 / (1 + np.exp(-z))
    return a
```

Code: feed-forward, cost function and gradient descent

- **Feed-forward**

$$Z = w^T X + b$$

$$A = \sigma(Z)$$

- **Cost function**

$$J = -\frac{1}{m} np.sum[Y log\, A + (1-Y) log(1-A)]$$

- **Gradient descent**

$$\frac{\partial L(A,Y)}{\partial w} = X(A-Y)^T$$

$$\frac{\partial L(A,Y)}{\partial b} = np.sum(A-Y)$$

```
# initialize parameters
n_dim = train_data_sta.shape[0] # number of rows in training data
w = np.zeros((n_dim, 1))
b = 0

# propagate
def propagate(w, b, X, Y):

    # feed-forward function
    Z = np.dot(w.T, X) + b # np.dot -> matrix multiplication
    A = sigmoid(Z)

    # cost function
    m = X.shape[1]
    J = -1/m * np.sum(Y * np.log(A) + (1-Y) * np.log(1-A))

    # gradient descent (Note: mean)
    dw = 1/m * np.dot(X,(A-Y).T)
    db = 1/m * np.sum(A-Y)

    grands = {'dw': dw, 'db': db}

    return grands, J
```

Code: optimization

$$w \leftarrow w - \alpha \cdot \frac{1}{m} \cdot X(A-Y)^T$$

$$b \leftarrow b - \alpha \cdot \frac{1}{m} \cdot np.sum(A-Y)$$

```
# Optimization
def optimize(w, b, X, Y, alpha, n_iters):
    costs = []
    for i in range(n_iters):
        grands, J = propagate(w, b, X, Y)
        dw = grands['dw']
        db = grands['db']

        w = w - alpha * dw
        b = b - alpha * db

        if i % 100 == 0:
            costs.append(J)
            print('Epoch %d: cost = %.4f' % (i+1, J))

    grands = {'dw': dw, 'db': db}
    params = {'w': w, 'b': b}

    return grands, params, costs
```

Code: prediction

```
# Prediction
def predict(w, b, X_test):

    Z = np.dot(w.T, X_test) + b
    A = sigmoid(Z)

    m = X_test.shape[1]
    Y_pred = np.zeros((1, m))

    for i in range(m):
        if A[:, i] > 0.5:
            Y_pred[:, i] = 1
        else:
            Y_pred[:, i] = 0

    return Y_pred
```

Code: integrating previous steps

```
# integrating previous steps
def model(w, b, X_train, X_test, Y_train, Y_test, alpha, n_iters):
    grands, params, costs = optimize(w, b, X_train, Y_train, alpha, n_iters)
    w = params['w']
    b = params['b']

    Y_pred_train = predict(w, b, X_train)
    Y_pred_test = predict(w, b, X_test)

    print('Train accuracy: %.2f' % np.mean(y_pred_train == y_train))
    print('Test accuracy: %.2f' % np.mean(y_pred_test == y_test))

    dic = {
           'w': w,
           'b': b,
           'costs': costs,
           'y_pred_train': y_pred_train,
           'y_pred_test': y_pred_test,
           'alpha': alpha
    }

    return dic
```

Code: train and test

- Train and test

```
dic = model(w, b,
            train_data_sta, train_labels_tran,
            test_data_sta, test_labels_tran,
            alpha = 0.005, n_iters = 2000
           )
```

- Plot the change of cost as a function of iterations

```
plt.plot(b['costs'])
plt.xlabel('per hundred iterations')
plt.ylabel('cost')
```

Code: predict a picture

```
index = 1
print('True label: %d' % test_labels_tran[0, index])
print('Pred label: %d' % int(b['y_pred_test'][0, index]))

# show the picture
plt.imshow(test_data_org[index])
```