Deep Learning 吴恩达深度学习笔记—精简版

引言

1. 神经网络基础编程

1.1. 逻辑回归

（1）训练样本

带标签的样本（m个）：

$$(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)})\cdots(x^{(i)},y^{(i)})\cdots(x^{(m)},y^{(m)})$$

（2）逻辑回归函数及其导数

逻辑回归函数：

$$\hat{y}=\sigma(z)=\frac{1}{1+e^{-z}},\text{其中}z=w^Tx+b$$

对w求偏导数，根据链式法则，首先计算：

$$\frac{\partial\hat{y}}{\partial z}=\frac{e^{-z}}{(1+e^{-z})^2},\frac{\partial z}{\partial w}=x$$

进一步可得，逻辑回归函数对w的偏导数：

$$\frac{\partial\hat{y}}{\partial w}=\frac{\partial\hat{y}}{\partial z}\frac{\partial z}{\partial w}=\frac{xe^{-(w^Tx+b)}}{(1+e^{-(w^Tx+b)})^2}$$

对b求偏导数，根据链式法则，首先计算：

$$\frac{\partial z}{\partial b}=1$$

进一步可得，逻辑回归函数对b的偏导数：

$$\frac{\partial \hat{y}}{\partial b}=\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b}=\frac{e^{-(w^Tx+b)}}{(1+e^{-(w^Tx+b)})^2}$$

（3）损失函数及其导数

损失函数，用于衡量单个样本的预测值与训练数据之间的误差：

$$L(\hat{y}^{(i)},y^{i})=-y^{(i)}\log\hat{y}^{(i)}-(1-y^{(i)})\log(1-\hat{y}^{(i)})$$

对w求偏导数，根据链式法则，首先计算：

$$\frac{\partial L}{\partial \hat{y}}=\frac{-y^{(i)}}{\hat{y}^{(i)}}+\frac{1-y^{(i)}}{1-\hat{y}^{(i)}}$$

进一步可得，损失函数对w的偏导数：

$$\frac{\partial L}{\partial w}=\frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w}=[\frac{-y^{(i)}}{\hat{y}^{(i)}}+\frac{1-y^{(i)}}{1-\hat{y}^{(i)}}]\frac{xe^{-(w^Tx+b)}}{(1+e^{-(w^Tx+b)})^2}$$

化简得：

$$\frac{\partial L}{\partial w}=x(\hat{y}-y)$$

进一步可得，损失函数对b的偏导数：

$$\frac{\partial L}{\partial b}=\frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}=[\frac{-y^{(i)}}{\hat{y}^{(i)}}+\frac{1-y^{(i)}}{1-\hat{y}^{(i)}}]\frac{e^{-(w^Tx+b)}}{(1+e^{-(w^Tx+b)})^2}$$

化简得：

$$\frac{\partial L}{\partial b}=\hat{y}-y$$

（4）代价函数及其导数

代价函数，用于衡量整个训练集上，所有样本预测值与训练数据误差的平均，也就是损失函数的平均：

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

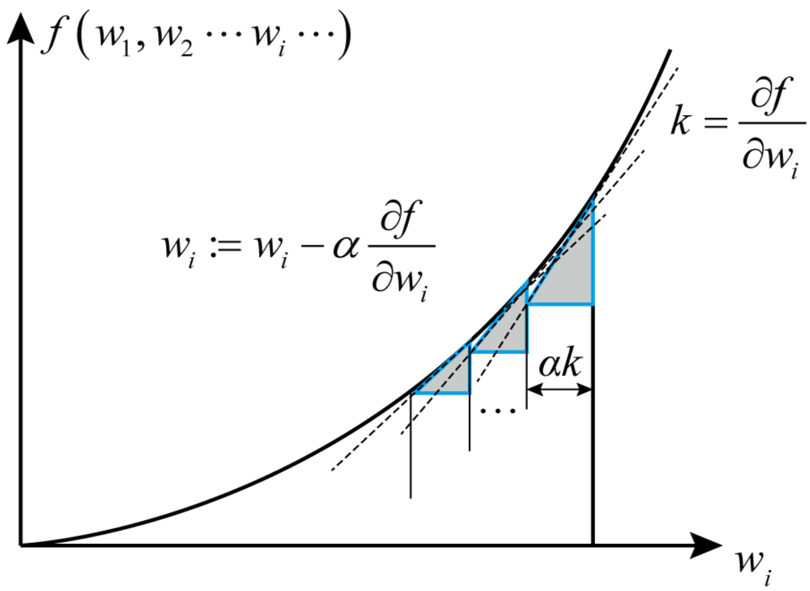
代价函数对w的偏导数：

$$\frac{\partial J}{\partial w}=\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial w}=\frac{1}{m}\sum_{i=1}^{m}\{x(\hat{y}-y)\}$$

代价函数对b的偏导数：

$$\frac{\partial J}{\partial b}=\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L}{\partial b}=\frac{1}{m}\sum_{i=1}^{m}(\hat{y}-y)$$

1.2 梯度下降法



（1）迭代求解w：

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

（2）迭代求解b：

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

1.3 逻辑回归算法Python实现：

（1）循环嵌套编程

```
import numpy as np
import time
import matplotlib.pyplot as plt

#计时开始
```

```python
tic = time.time()

# 输入数据：样本
x = np.array([-2, -1, 0, 1, 2])  # 样本x
y = 1/(1+np.exp(-x))  # 样本y：1/(1+np.exp(-x))
m = len(x)  # 样本容量

# 赋初值
alpha = 0.01  # 学习率
w = 1
b = 1
i = 0
n_max = 1000
L = np.empty(n_max)  # 损失函数
J = np.empty(n_max)  # 代价函数

# 第一层循环：采用梯度下降法计算w，b值
while (i<n_max):
    sumdL_w = 0  # 损失函数对w的导数赋初值
    sumdL_b = 0  # 损失函数对b的导数赋初值
    sumJ = 0  # 代价函数赋初值

    # 第二层循环：遍历样本，计算损失函数、代价函数及其导数
    for j in range(m):
        # 计算假设函数及其导数
        z = w * x[j] + b
        yhat = 1 / (1 + np.exp(-z))  # 逻辑回归预测值
        dyhat_z = np.exp(-z) / ((1 + np.exp(-z)) ** 2)  # 
yhat对z的偏导数
        dz_w = x[j]  # z对w的偏导数，以下命名规则类似
        dz_b = 1
        dyhat_w = dyhat_z * dz_w
        dyhat_b = dyhat_z * dz_b

        # 计算损失函数及其导数
        L = -y[j]*np.log(yhat) - (1-y[j])*np.log(1-yhat)
        dL_yhat = -y[j] / yhat + (1 - y[j]) / (1 - yhat)
        dL_w = dL_yhat * dyhat_w
        dL_b = dL_yhat * dyhat_b

        # 计算代价函数及其导数
```

```
        sumJ = sumJ + L
        sumdL_w = sumdL_w + dL_w
        sumdL_b = sumdL_b + dL_b

    J[i] = sumJ/m
    dJ_w = sumdL_w / m
    dJ_b = sumdL_b / m

    # 梯度下降迭代计算w,b
    w = w - alpha * dJ_w
    b = b - alpha * dJ_b
    print('i=',i, 'w=',w, 'b=',b, 'J[i]=',J[i])
    i = i + 1

# 计时结束
toc = time.time()
tictoc = toc - tic
print('total time of CPU is', tictoc, 's')
plt.plot(J)
plt.show()
```

(2) 向量化编程

```
#!usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os

'''
Logistic regression algorithm to identify the apple
picture.
The training set consists of ten 196pixel × 196pixel
images of apples.
The results show that initialization, learning rate, and
normalization are key factors in
achieving algorithm convergence.
Using a training dataset with both positive and negative
labels will improve prediction.
```

```
This is a standard program in vectorization.
'''

# Load images, generate training or test dataset, and
preprocessing

def load_image_data(dir_data):
    """
    This function load the image database and create data
set of numpy arrays

    Argument:
    names -- a set of image names in dir_data
    data_set_x_orig -- initialized numpy array for storing
all images
    img -- image opened by PIL library
    index -- from 0 to number of names
    data_set_x_orig_flatten -- flatten numpy array of
data_set_x_orig

    Return:
    data_set_x_normalized -- normalized numpy array of
data_set_x_orig_flatten
    """

    names = os.listdir(dir_data)
    data_set_x_orig = np.zeros([len(names),196,196,3])

    # (1)Get data set - 4 dimentional
    index = 0

    for img_name in (names):
        if os.path.splitext(img_name)[1] == '.jpg':  #
only for images with .jpg
            img = Image.open(dir_data + '\\' + img_name)
            data_set_x_orig[index] = np.array(img)

            index = index + 1
            print('data_set shape = ',
data_set_x_orig.shape, 'img size = ', img.size,
'img_dpi=', img.info['dpi'])
```

```
            #plt.imshow(img_arr)
            #plt.show()

    # (2)Reshape data set - 2 dimensional
    data_set_x_orig_flatten =
data_set_x_orig.reshape(data_set_x_orig.shape[0],-1).T

    # (3) Normalize data set - 2 dimensional
    data_set_x_normalized = data_set_x_orig_flatten/255

    return data_set_x_normalized


# Graded function: sigmoid

def sigmoid(z):
    """
    This function compute the sigmoid of z

    Argument:
    z -- a scalar or numpy array

    Return:
    s -- sigmoid(z)
    """

    s = 1 / (1 + np.exp(-z))

    return s


# Graded function: Initialization

def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim,
1) for w and initializes b to 0.

    Argument:
```

```
    dim -- size of the w vector we want (or number of
parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """

    w = np.zeros((dim, 1))
    b = 0

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b


# Graded function: propagate

def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the
propagation explained above

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px *
3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of
examples)
    Y -- true "label" vector (containing 0 if non-apple, 1
if apple) of size (1, number of examples)

    Return:
    cost -- negative log-likelihood cost for logistic
regression
    dw -- gradient of the loss with respect to w, thus
same shape as w
    db -- gradient of the loss with respect to b, thus
same shape as b
    """
```

```
    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T, X) + b)            # compute
activation
    cost = -1 / m * np.sum(Y * np.log(A) + (1 - Y) *
np.log(1 - A))          # compute cost

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = 1 / m * np.dot(X, (A - Y).T)
    db = 1 / m * np.sum(A - Y)

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)  # remove axes of length one
from a numpy array
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost


# Graded function: optimize

def optimize(w, b, X, Y, num_iterations, learning_rate,
print_cost = False):
    """
    This function optimizes w and b by running a gradient
descent algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px *
3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of
examples)
    Y -- true "label" vector (containing 0 if non-cat, 1
if cat), of shape (1, number of examples)
```

```
    num_iterations -- number of iterations of the
optimization loop
    learning_rate -- learning rate of the gradient descent
update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias
b
    grads -- dictionary containing the gradients of the
weights and bias with respect to the cost function
    costs -- list of all the costs computed during the
optimization, this will be used to plot the learning
curve.
    """

    costs = []

    for i in range(num_iterations):


        # Cost and gradient calculation
        grads, cost = propagate(w, b, X, Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule
        w = w - learning_rate * dw
        b = b - learning_rate * db
        #print('loop = ', i,'dw=',dw[0], 'db=',db, 'cost
=', cost )

        # Record the costs
        if i % 1 == 0:
            costs.append(cost)

        # Print the cost every 1 training examples
        if print_cost and i % 1 == 0:
```

```
            print ("Cost after iteration %i: %f" %(i,
cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs


# Graded function: predict

def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned
logistic regression parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px *
3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of
examples)

    Returns:
    Y_prediction -- a numpy array (vector) containing all
predictions (0/1) for the examples in X
    '''

    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    # Compute vector "A" predicting the probabilities of a
cat being present in the picture
    A = sigmoid(np.dot(w.T, X) + b)

    for i in range(A.shape[1]):
```

```
        # Convert probabilities A[0,i] to actual
predictions p[0,i]
        if A[0, i] <= 0.5:
            Y_prediction[0, i] = 0
        else:
            Y_prediction[0, i] = 1

    assert(Y_prediction.shape == (1, m))

    return Y_prediction


# GRADED FUNCTION: model

def model(X_train, Y_train, X_test, Y_test, num_iterations
= 100, learning_rate = 0.5, print_cost = False):
    """
    Builds the logistic regression model by calling the
function you've implemented previously

    Arguments:
    X_train -- training set represented by a numpy array
of shape (num_px * num_px * 3, m_train)
    Y_train -- training labels represented by a numpy
array (vector) of shape (1, m_train)
    X_test -- test set represented by a numpy array of
shape (num_px * num_px * 3, m_test)
    Y_test -- test labels represented by a numpy array
(vector) of shape (1, m_test)
    num_iterations -- hyperparameter representing the
number of iterations to optimize the parameters
    learning_rate -- hyperparameter representing the
learning rate used in the update rule of optimize()
    print_cost -- Set to true to print the cost every 100
iterations

    Returns:
    d -- dictionary containing information about the
model.
    """
```

```
    # initialize parameters with zeros
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent
    parameters, grads, costs = optimize(w, b, X_train,
Y_train, num_iterations, learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary
"parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples
    Y_prediction_train = predict(w, b, X_train)
    Y_prediction_test = predict(w, b, X_test)

    # Print train/test Errors
    print("train accuracy: {} %".format(100 -
np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
    print("test accuracy: {} %".format(100 -
np.mean(np.abs(Y_prediction_test - Y_test)) * 100))

    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d


if __name__ == '__main__':
    # Get the data set for training and testing
    X_train = load_image_data("apple_train")
    Y_train = np.array([0,1,0,1,0,1,1,1,1])
    X_test = load_image_data("apple_test")
    Y_test = np.array([1,1,0,0,0,1,1,1,1])

    # Get the model parameters
```

```
    print('X_train=',X_train.shape, '\n X_test=',
X_test.shape)
    apple_model = model(X_train, Y_train, X_test, Y_test,
num_iterations = 1000, learning_rate = 0.0001, print_cost
= True)
    costs_model = apple_model["costs"]
    num_iterations =  apple_model["num_iterations"]

    # Plot learning curve (with costs)
    plt.plot(np.arange(num_iterations/1),costs_model)
    plt.ylabel('cost')
    plt.xlabel('iterations ')
    plt.title("Learning rate =" +
str(apple_model["learning_rate"]))
    plt.show()
```

2. 单层神经网络

2.1 带标签的训练集

带1个标签的m个样本，每个样本有n个特征：

$$[x_1^{(1)},x_2^{(1)};y^{(1)}],\cdots[x_1^{(i)},x_2^{(i)};y^{(i)}]\cdots[x_1^{(m)},x_2^{(m)};y^{(m)}]$$

训练样本集可表示为如下矩阵形式：

$$\boldsymbol{X}=(x_{ij})_{n_x\times m}=\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \end{bmatrix},$$

训练样本集的标签可表示为如下矩阵形式：

$$\boldsymbol{Y}=(y_{ij})_{1\times m}=\begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

2.2 输入层、隐藏层、输出层

确定输入层元素数量n_x：由训练集每个样本的特征决定，因此输入层元素数量为

$$n_x = 2$$

确定隐藏层元素数量n_h：可人为决定，例如取

$$n_h = 4$$

确定输出层元素数量n_y：由输入层样本的标签决定，标签由1个数字进行表示（0或1），因此输出层数量为

$$n_y = 1$$

2.3 正向传播

（1）正向传播流程

$$\left.\begin{array}{r} \boldsymbol{A}^{[0]} = \boldsymbol{X} \\ \boldsymbol{W}^{[1]} \\ \boldsymbol{B}^{[1]} \end{array}\right\} \Rightarrow \boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]}\boldsymbol{X} + \boldsymbol{B}^{[1]} \Rightarrow \left.\begin{array}{r} \boldsymbol{A}^{[1]} = g(\boldsymbol{Z}^{[1]}) \Rightarrow \boldsymbol{W}^{[2]} \\ \boldsymbol{B}^{[2]} \end{array}\right\} \Rightarrow$$
$$\boldsymbol{Z}^{[2]} = \boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]} + \boldsymbol{B}^{[2]} \Rightarrow \boldsymbol{A}^{[2]} = \sigma(\boldsymbol{Z}^{[2]}) \Rightarrow J = J(\boldsymbol{Y}, \boldsymbol{A}^{[2]})$$

（2）详细推导过程

初始化权重矩阵$\boldsymbol{W_1}$，该矩阵作用于输入层训练样本$\boldsymbol{X}$

$$\boldsymbol{W}^{[1]} = \left(w_{ij}^{[1]}\right)_{n_h \times n_x} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & \cdots & w_{1n_x}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & \cdots & w_{2n_x}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h1}^{[1]} & w_{n_h2}^{[1]} & \cdots & w_{n_hn_x}^{[1]} \end{bmatrix}$$

同时，初始化偏差矩阵$\boldsymbol{B}^{[1]}$，该向量作用于权重矩阵与训练样本矩阵的乘积$\boldsymbol{W}^{[1]}\boldsymbol{X}$

$$\boldsymbol{B}^{[1]} = \left(b_{ij}^{[1]}\right)_{n_h \times m} = \begin{bmatrix} b_{11}^{[1]} & b_{12}^{[1]} & \cdots & b_{1m}^{[1]} \\ b_{21}^{[1]} & b_{22}^{[1]} & \cdots & b_{2m}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n_h1}^{[1]} & b_{n_h2}^{[1]} & \cdots & b_{n_hm}^{[1]} \end{bmatrix}$$

根据矩阵的运算法则可得：

$$\boldsymbol{Z}^{[1]}=\left(z_{ij}^{[1]}\right)_{n_h\times m}=\boldsymbol{W}^{[1]}\boldsymbol{X}+\boldsymbol{B}^{[1]}=\begin{bmatrix} z_{11}^{[1]} & z_{12}^{[1]} & \cdots & z_{1m}^{[1]} \\ z_{21}^{[1]} & z_{22}^{[1]} & \cdots & z_{2m}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n_h1}^{[1]} & z_{n_h2}^{[1]} & \cdots & z_{n_hm}^{[1]} \end{bmatrix}$$

单一隐藏层的激活函数为：

$$\boldsymbol{A}^{[1]}=\left(a_{ij}^{[1]}\right)_{n_h\times m}=\begin{bmatrix} a_{11}^{[1]} & a_{12}^{[1]} & \cdots & a_{1m}^{[1]} \\ a_{21}^{[1]} & a_{22}^{[1]} & \cdots & a_{2m}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_h1}^{[1]} & a_{n_h2}^{[1]} & \cdots & a_{n_hm}^{[1]} \end{bmatrix}, \text{where} \quad a_{ij}^{[1]}=\tanh(z_{ij}^{[1]})$$

其中

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

初始化权重矩阵$\boldsymbol{W}^{[2]}$，该矩阵作用于隐藏层元素$\boldsymbol{A}^{[1]}$

$$\boldsymbol{W}^{[2]}=\begin{bmatrix} w_1^{[2]} & w_2^{[2]} & \cdots & w_{n_h}^{[2]} \end{bmatrix}_{n_y\times n_h}$$

同时，初始化偏差矩阵$\boldsymbol{B}^{[2]}$，该矩阵作用于权重矩阵与训练样本矩阵的乘积$\boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]}$

$$\boldsymbol{B}^{[2]}=\left(b_{ij}^{[2]}\right)_{n_y\times m}=\begin{bmatrix} b_1^{[2]} & b_2^{[2]} & \cdots & b_m^{[2]} \end{bmatrix}$$

同样，根据矩阵的运算法则可得：

$$\boldsymbol{Z}^{[2]}=\left(z_{ij}^{[2]}\right)_{n_y\times m}=\boldsymbol{W}^{[2]}\boldsymbol{A}^{[1]}+\boldsymbol{B}^{[2]}$$

输出层矩阵为：

$$\boldsymbol{A}^{[2]}=\left(a_{ij}^{[2]}\right)_{n_y\times m}=\begin{bmatrix} a_1^{[2]} & a_2^{[2]} & \cdots & a_m^{[2]} \end{bmatrix}, \text{where} \quad a_{ij}^{[2]}=\sigma(z_{ij}^{[2]})$$

其中

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

单个训练样本损失函数：

$$L^{(i)}(a^{(i)}, y^{(i)}) = -y^{(i)}\ln a^{(i)} - (1 - y^{(i)})\ln(1 - a^{(i)})$$

整个训练集代价函数：

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

矩阵形式的代价函数：

$$J = \frac{1}{m}\left[-\boldsymbol{Y}\ln(\boldsymbol{A}^{T[2]}) - (\boldsymbol{E}_1 - \boldsymbol{Y})\ln(\boldsymbol{E}_2 - \boldsymbol{A}^{T[2]})\right]$$

其中，$\boldsymbol{E}_1$为与$\boldsymbol{Y}$具有相同行列数的全1矩阵，$\boldsymbol{E}_2$为与$\boldsymbol{A}^T$具有相同行列数的全1矩阵。

2.6 反向传播

目前，大多数教程中并未对深度学习正向传播、反向传播的理论过程进行规范化的整理与表达。因此，本节根据矩阵微分相关的运算法则，参考《矩阵分析与应用（第2版）》—张贤达，对深度学习反向传播过程进行梳理，并阐释符合数学规范的表达方式。

（1）反向传播流程

$$\frac{\partial J}{\partial \boldsymbol{A}^{[2]}} \Rightarrow \frac{\partial \boldsymbol{A}^{[2]}}{\partial \boldsymbol{Z}^{[2]}} \Rightarrow \begin{cases} \frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{W}^{[2]}} \\ \frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{B}^{[2]}} \end{cases}$$

$$\Rightarrow \frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{A}^{[1]}} \Rightarrow \frac{\partial \boldsymbol{A}^{[1]}}{\partial \boldsymbol{Z}^{[1]}} \Rightarrow \begin{cases} \frac{\partial \boldsymbol{Z}^{[1]}}{\partial \boldsymbol{W}^{[1]}} \\ \frac{\partial \boldsymbol{Z}^{[1]}}{\partial \boldsymbol{B}^{[1]}} \end{cases}$$

（2）详细推导过程

计算代价函数J对$\boldsymbol{W}^{[2]}$的偏导数

$$\frac{\partial J}{\partial \boldsymbol{W}^{[2]}} = \frac{\partial J}{\partial \boldsymbol{A}^{[2]}}\frac{\partial \boldsymbol{A}^{[2]}}{\partial \boldsymbol{Z}^{[2]}}\frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{W}^{[2]}}$$

计算代价函数J对$\boldsymbol{B}^{[2]}$的偏导数

$$\frac{\partial J}{\partial \boldsymbol{B}^{[2]}} = \frac{\partial J}{\partial \boldsymbol{A}^{[2]}} \frac{\partial \boldsymbol{A}^{[2]}}{\partial \boldsymbol{Z}^{[2]}} \frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{B}^{[2]}}$$

首先，计算代价函数J的对$\boldsymbol{A}^{[2]}$的偏导数

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{A}^{[2]}} &= \left[\frac{\partial J}{\partial a_1^{[2]}}, \frac{\partial J}{\partial a_2^{[2]}}, \cdots, \frac{\partial J}{\partial a_m^{[2]}}\right] \\
&= \frac{1}{m}\left[-\frac{y_1}{a_1^{[2]}} - \frac{1-y_1}{1-a_1^{[2]}}, -\frac{y_2}{a_2^{[2]}} - \frac{1-y_2}{1-a_2^{[2]}}, \cdots, -\frac{y_m}{a_m^{[2]}} - \frac{1-y_m}{1-a_m^{[2]}},\right]
\end{aligned}$$

其次，计算$\boldsymbol{A}^{[2]}$对$\boldsymbol{Z}^{[2]}$的偏导数

$$\begin{aligned}
\frac{\partial \boldsymbol{A}^{[2]}}{\partial \boldsymbol{Z}^{[2]}} = \frac{\partial \mathrm{vec} \boldsymbol{A}^{[2]}}{\partial \mathrm{vec}(\boldsymbol{Z}^{[2]})^T} &= \begin{bmatrix} \frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} & \frac{\partial a_1^{[2]}}{\partial z_2^{[2]}} & \cdots & \frac{\partial a_1^{[2]}}{\partial z_m^{[2]}} \\ \frac{\partial a_2^{[2]}}{\partial z_1^{[2]}} & \frac{\partial a_2^{[2]}}{\partial z_2^{[2]}} & \cdots & \frac{\partial a_2^{[2]}}{\partial z_m^{[2]}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_m^{[2]}}{\partial z_1^{[2]}} & \frac{\partial a_m^{[2]}}{\partial z_2^{[2]}} & \cdots & \frac{\partial a_m^{[2]}}{\partial z_m^{[2]}} \end{bmatrix} \\
&= \begin{bmatrix} \sigma(z_1^{[1]})[1-\sigma(z_1^{[1]})] & 0 & \cdots & 0 \\ 0 & \sigma(z_2^{[1]})[1-\sigma(z_2^{[1]})] & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma(z_m^{[1]})[1-\sigma(z_m^{[1]})] \end{bmatrix}
\end{aligned}$$

其中，$\sigma(z) = \frac{1}{1+e^{-z}}$。

激活函数的偏导数项:

$$\frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{W}^{[2]}} = \frac{\partial \mathrm{vec}\boldsymbol{Z}^{[2]}}{\partial \mathrm{vec}(\boldsymbol{W}^{[2]})^T} = \begin{bmatrix} \frac{\partial z_1^{[2]}}{\partial w_1^{[2]}} & \frac{\partial z_1^{[2]}}{\partial w_2^{[2]}} & \cdots & \frac{\partial z_1^{[2]}}{\partial w_{n_h}^{[2]}} \\ \frac{\partial z_2^{[2]}}{\partial w_1^{[2]}} & \frac{\partial z_2^{[2]}}{\partial w_2^{[2]}} & \cdots & \frac{\partial z_2^{[2]}}{\partial w_{n_h}^{[2]}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_m^{[2]}}{\partial w_1^{[2]}} & \frac{\partial z_m^{[2]}}{\partial w_2^{[2]}} & \cdots & \frac{\partial z_m^{[2]}}{\partial w_{n_h}^{[2]}} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}^{[1]} & a_{21}^{[1]} & \cdots & a_{n_h 1}^{[1]} \\ a_{12}^{[1]} & a_{22}^{[1]} & \cdots & a_{n_h 2}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1m}^{[1]} & a_{2m}^{[1]} & \cdots & a_{n_h m}^{[1]} \end{bmatrix} = (\boldsymbol{A}^{[1]})^T$$

线性组合的偏导数项

$$\frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{B}^{[2]}} = \frac{\partial \mathrm{vec}\boldsymbol{Z}^{[2]}}{\partial \mathrm{vec}(\boldsymbol{B}^{[2]})^T} = \begin{bmatrix} \frac{\partial z_1^{[2]}}{\partial b_1^{[2]}} & \frac{\partial z_1^{[2]}}{\partial b_2^{[2]}} & \cdots & \frac{\partial z_1^{[2]}}{\partial b_m^{[2]}} \\ \frac{\partial z_2^{[2]}}{\partial b_1^{[2]}} & \frac{\partial z_2^{[2]}}{\partial b_2^{[2]}} & \cdots & \frac{\partial z_2^{[2]}}{\partial b_m^{[2]}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_m^{[2]}}{\partial b_1^{[2]}} & \frac{\partial z_m^{[2]}}{\partial b_2^{[2]}} & \cdots & \frac{\partial z_m^{[2]}}{\partial b_m^{[2]}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \boldsymbol{I}_{m\times m}$$

其中，$\boldsymbol{A}_{:,i}^{[1]T}$为矩阵$\boldsymbol{A}^{[1]}$第$i$列的转置，是一个与$\boldsymbol{W}^{[2]}$形状、元素数量相同的行向量。

$$
\begin{aligned}
\frac{\partial \boldsymbol{Z}^{[2]}}{\partial \boldsymbol{A}^{[1]}} = \frac{\partial \mathrm{vec} \boldsymbol{Z}^{[2]}}{\partial \mathrm{vec}(\boldsymbol{A}^{[1]})^T} &= \begin{bmatrix}
\frac{\partial z_1^{[2]}}{\partial a_{11}^{[1]}} & \cdots & \frac{\partial z_1^{[2]}}{\partial a_{41}^{[1]}} & \cdots & \frac{\partial z_1^{[2]}}{\partial a_{1m}^{[1]}} & \cdots & \frac{\partial z_1^{[2]}}{\partial a_{4m}^{[1]}} \\
\frac{\partial z_2^{[2]}}{\partial a_{11}^{[1]}} & \cdots & \frac{\partial z_2^{[2]}}{\partial a_{41}^{[1]}} & \cdots & \frac{\partial z_2^{[2]}}{\partial a_{1m}^{[1]}} & \cdots & \frac{\partial z_2^{[2]}}{\partial a_{4m}^{[1]}} \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\frac{\partial z_m^{[2]}}{\partial a_{11}^{[1]}} & \cdots & \frac{\partial z_m^{[2]}}{\partial a_{41}^{[1]}} & \cdots & \frac{\partial z_m^{[2]}}{\partial a_{1m}^{[1]}} & \cdots & \frac{\partial z_m^{[2]}}{\partial a_{4m}^{[1]}}
\end{bmatrix}_{m \times 4m} \\
&= \boldsymbol{I}_{m \times m} \otimes \boldsymbol{W}_{1 \times 4} \\
&= \begin{bmatrix}
w_1^{[2]} & w_2^{[2]} & w_3^{[2]} & w_4^{[2]} & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & w_1^{[2]} & w_2^{[2]} & w_3^{[2]} & w_4^{[2]} & \cdots & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & w_1^{[2]} & w_2^{[2]} & w_3^{[2]} & w_4^{[2]}
\end{bmatrix}
\end{aligned}
$$

z2对a1继续求导

$$\frac{\partial \boldsymbol{A}^{[1]}}{\partial \boldsymbol{Z}^{[1]}}=\frac{\partial \mathrm{vec} \boldsymbol{A}^{[1]}}{\partial \mathrm{vec}(\boldsymbol{Z}^{[1]})^T}=\begin{bmatrix} \frac{\partial a_{11}^{[1]}}{\partial z_{11}^{[1]}} & \cdots & \frac{\partial a_{11}^{[1]}}{\partial z_{41}^{[1]}} & \cdots & \frac{\partial a_{11}^{[1]}}{\partial z_{1m}^{[1]}} & \cdots & \frac{\partial a_{11}^{[1]}}{\partial z_{4m}^{[1]}} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial a_{41}^{[1]}}{\partial z_{11}^{[1]}} & \cdots & \frac{\partial a_{41}^{[1]}}{\partial z_{41}^{[1]}} & \cdots & \frac{\partial a_{41}^{[1]}}{\partial z_{1m}^{[1]}} & \cdots & \frac{\partial a_{41}^{[1]}}{\partial z_{4m}^{[1]}} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial a_{1m}^{[1]}}{\partial z_{11}^{[1]}} & \cdots & \frac{\partial a_{1m}^{[1]}}{\partial z_{41}^{[1]}} & \cdots & \frac{\partial a_{1m}^{[1]}}{\partial z_{1m}^{[1]}} & \cdots & \frac{\partial a_{1m}^{[1]}}{\partial z_{4m}^{[1]}} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{\partial a_{4m}^{[1]}}{\partial z_{11}^{[1]}} & \cdots & \frac{\partial a_{4m}^{[1]}}{\partial z_{41}^{[1]}} & \cdots & \frac{\partial a_{4m}^{[1]}}{\partial z_{1m}^{[1]}} & \cdots & \frac{\partial a_{4m}^{[1]}}{\partial z_{4m}^{[1]}} \end{bmatrix}_{4m\times 4m}$$

$$=\begin{bmatrix} 1-g^2(z_{11}) & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1-g^2(z_{41}) & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1-g^2(z_{1m}) & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1-g^2(z_{4m}) \end{bmatrix}_{4m\times 4m}$$

a1对z1继续求导

$$\frac{\partial \boldsymbol{Z}^{[1]}}{\partial \boldsymbol{W}^{[1]}} = \frac{\partial \mathrm{vec} \boldsymbol{Z}^{[1]}}{\partial \mathrm{vec}(\boldsymbol{W}^{[1]})^T} = \begin{bmatrix} \frac{\partial z_{11}^{[1]}}{\partial w_{11}^{[1]}} & \cdots & \frac{\partial z_{11}^{[1]}}{\partial w_{41}^{[1]}} & \frac{\partial z_{11}^{[1]}}{\partial w_{12}^{[1]}} & \cdots & \frac{\partial z_{11}^{[1]}}{\partial w_{42}^{[1]}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{41}^{[1]}}{\partial w_{11}^{[1]}} & \cdots & \frac{\partial z_{41}^{[1]}}{\partial w_{41}^{[1]}} & \frac{\partial z_{41}^{[1]}}{\partial w_{12}^{[1]}} & \cdots & \frac{\partial z_{41}^{[1]}}{\partial w_{42}^{[1]}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{1m}^{[1]}}{\partial w_{11}^{[1]}} & \cdots & \frac{\partial z_{1m}^{[1]}}{\partial w_{41}^{[1]}} & \frac{\partial z_{1m}^{[1]}}{\partial w_{12}^{[1]}} & \cdots & \frac{\partial z_{1m}^{[1]}}{\partial w_{42}^{[1]}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{4m}^{[1]}}{\partial w_{11}^{[1]}} & \cdots & \frac{\partial z_{4m}^{[1]}}{\partial w_{41}^{[1]}} & \frac{\partial z_{4m}^{[1]}}{\partial w_{12}^{[1]}} & \cdots & \frac{\partial z_{4m}^{[1]}}{\partial w_{42}^{[1]}} \end{bmatrix}_{4m \times 8}$$

$$= (\boldsymbol{X}^T)_{m \times 2} \otimes \boldsymbol{I}_{4 \times 4} = \begin{bmatrix} x_1^{(1)} & \cdots & 0 & x_2^{(1)} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & x_1^{(1)} & 0 & \cdots & x_2^{(1)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & \cdots & 0 & x_2^{(m)} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & x_1^{(m)} & 0 & \cdots & x_2^{(m)} \end{bmatrix}_{4m \times 8}$$

z1对w1求导

$$\frac{\partial \boldsymbol{Z}^{[1]}}{\partial \boldsymbol{B}^{[1]}}=\frac{\partial \operatorname{vec} \boldsymbol{Z}^{[1]}}{\partial \operatorname{vec}(\boldsymbol{B}^{[1]})^T}=\begin{bmatrix}
\frac{\partial z_{11}^{[1]}}{\partial b_{11}^{[1]}} & \cdots & \frac{\partial z_{11}^{[1]}}{\partial b_{41}^{[1]}} & \cdots & \frac{\partial z_{11}^{[1]}}{\partial b_{1m}^{[1]}} & \cdots & \frac{\partial z_{11}^{[1]}}{\partial b_{4m}^{[1]}} \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\frac{\partial z_{41}^{[1]}}{\partial b_{11}^{[1]}} & \cdots & \frac{\partial z_{41}^{[1]}}{\partial b_{41}^{[1]}} & \cdots & \frac{\partial z_{41}^{[1]}}{\partial b_{1m}^{[1]}} & \cdots & \frac{\partial z_{41}^{[1]}}{\partial b_{4m}^{[1]}} \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\frac{\partial z_{1m}^{[1]}}{\partial b_{11}^{[1]}} & \cdots & \frac{\partial z_{1m}^{[1]}}{\partial b_{41}^{[1]}} & \cdots & \frac{\partial z_{1m}^{[1]}}{\partial b_{1m}^{[1]}} & \cdots & \frac{\partial z_{1m}^{[1]}}{\partial b_{4m}^{[1]}} \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\frac{\partial z_{4m}^{[1]}}{\partial b_{11}^{[1]}} & \cdots & \frac{\partial z_{4m}^{[1]}}{\partial b_{41}^{[1]}} & \cdots & \frac{\partial z_{4m}^{[1]}}{\partial b_{1m}^{[1]}} & \cdots & \frac{\partial z_{4m}^{[1]}}{\partial b_{4m}^{[1]}}
\end{bmatrix}_{4m\times 4m}$$

$$=\boldsymbol{I}_{m\times m}\otimes \boldsymbol{I}_{4\times 4}=\begin{bmatrix}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 1 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 1 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{bmatrix}_{4m\times 4m}$$

z1对b1求导。

$$\frac{\partial}{\partial b^{[2](i)}} z^{[2](i)}(w^{[2]}, b^{[2](i)})=1$$

2.5 激活函数的导数

（1）sigmoid函数

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

sigmoid激活函数的导数:

$$\frac{\mathrm{d}}{\mathrm{d}z} g(z)=\frac{1}{1+e^{-z}}(1-\frac{1}{1+e^{-z}})=g(z)[1-g(z)]$$

（2）tanh函数

$$g(z)=\tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

tanh激活函数的导数：

$$\frac{\mathrm{d}}{\mathrm{d}z}g(z)=1-\left(\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}\right)^{2}=1-g^{2}(z)$$

（3）ReLU（Rectified Linear Unit）函数

$$g(z)=\max(0,z)$$

ReLU激活函数的导数：

$$\frac{\mathrm{d}}{\mathrm{d}z}g(z)=\begin{cases}0 & \text{if} \quad z<0\\ 1 & \text{if} \quad z>1\\ \text{undefined} & \text{if} \quad z=0\end{cases}$$

注意：当实际程序中出现z=0时，可赋值ReLU函数的导数值为0或1

（4）Leaky ReLU函数

$$g(z)=\max(0.01z,z)$$

Leaky ReLU激活函数的导数

$$\frac{\mathrm{d}}{\mathrm{d}z}g(z)=\begin{cases}0.01 & \text{if} \quad z<0\\ 1 & \text{if} \quad z>1\\ \text{undefined} & \text{if} \quad z=0\end{cases}$$

注意：当实际程序中出现z=0时，可赋值Leaky ReLU函数的导数值为0.01或1。

3.深层神经网络