Outline

NumPy Basics

Building basic functions with numpy

Sigmoid function, `np.exp()`

Exercise 2 - basic_sigmoid

Goal: Implement the sigmoid function using `math.exp()`

Sigmoid function: $σ(x) = \frac{1}{1+e^{(-x)}}$

Implementation with `math.exp()`: suitable for real number inputs, example implementation has been provided.

For example, testing with an input of `1`, the expected result is approximately `0.731`

```python
def basic_sigmoid(x):
    """
    Compute sigmoid of x.

    Arguments:
    x -- A scalar

    Return:
    s -- sigmoid(x)
    """
    s = 1 / (1 + np.exp(-np.array(x)))

    return s
```

Exercise 3 - sigmoid

Goal: Implement the sigmoid function using `numpy.exp()` to apply to real numbers, vectors and matrices.

Suitable for both real numbers, vectors and matrices

Check with vector `[1 2 3]`, the result is the array of corresponding sigmoid values. => “numpy” is flexible and efficient for numerical operations, especially networks and matrices.

```python
def sigmoid(x):
    """
    Compute the sigmoid of x

    Arguments:
    x -- A scalar or numpy array of any size

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

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

    return s
```

Sigmoid Gradient

Exercise 4 - Sigmoid_deravative

Optimizing the loss function using backpropagation requires calculating the gradient of the functions. The sigmoid function is one of the functions and its gradient calculation is important in adjusting the model's weights.

Objective: Implement the gradient calculation function of the sigmoid function using `numpy`

Gradient of sigmoid $σ'(x) =σ(x)(1-σ(x))$

Check with vector `[1 2 3]`, the result is the array of corresponding gradient values.

```
def sigmoid_derivative(x):
    """
    Compute the gradient (also called the slope or derivative) of the sigmoid function with respect to its input x.
    You can store the output of the sigmoid function into variables and then use it to calculate the gradient.

    Arguments:
    x -- A scalar or numpy array

    Return:
    ds -- Your computed gradient.
    """

    s = 1 / (1 + np.exp(-x))
    ds = s * (1 - s)

    return ds
```

Reshaping arrays

Exercise 5 - `image2vector()`

`image2vector()` converts a numpy3D array into a 1D vector

```
image.reshape((image.shape[0] * image.shape[1] * image.shape[2], 1))
```

converts the 3D array into a 1D vector of size `(length*height*depth,1)`

Using `reshape(-1, 1)` can help simplify the code, where `-1` automatically calculates the size needed to keep the same number of elements.

Example: Given a 3D array t_image of size `(3,3,2)`, the conversion will return a vector of size `(18,1)`

```
def image2vector(image):
    """
    Argument:
    image -- a numpy array of shape (length, height, depth)

    Returns:
    v -- a vector of shape (length*height*depth, 1)
    """

    length, height, depth = image.shape
    v = np.reshape(image, (length*height*depth, 1))
```

```
    return v
```

Normalizing rows

Exercise 6 - normalize_rows

Normalization leads to better performance because gradient descent converges faster after normalization. Normalize each row vector of matrix x into a unit vector (with length equal to 1).

Use `np.linalg.norm` to calculate the quadratic norm of each row.

$\frac{x}{x_{norm}}$ divides each element in x's row by that row's corresponding norm. This turns each row into a unit vector.

```
def normalize_rows(x):
    """
    Implement a function that normalizes each row of the matrix x (to have unit length).

    Argument:
    x -- A numpy matrix of shape (n, m)

    Returns:
    x -- The normalized (by row) numpy matrix. You are allowed to modify x.
    """

    x_norm = np.linalg.norm(x, ord=2, axis=1, keepdims=True)
    x = x / x_norm

    return x
```

Exercise 7 - softmax

The softmax function converts a vector of values into a probability distribution. Each value in the output vector is between 0 and 1, and the sum of all values is 1.

The `np.exp()` function is applied to each element of the matrix X, the result is the exponential value of the corresponding element in X

`np.sum()` sums each row.

$\frac{x_{exp}}{x_{sum}}$ uses numpy's broadcasting to divide each element of x_{exp} by the corresponding row total in x_{sum}.

For example: `[9 2 5 0 0][7 5 0 0 0]`

```
def softmax(x):
    """Calculates the softmax for each row of the input x.
```

```
    Your code should work for a row vector and also for matrices of shape
(m,n).

    Argument:
    x -- A numpy matrix of shape (m,n)

    Returns:
    s -- A numpy matrix equal to the softmax of x, of shape (m,n)
    """

    x_exp = np.exp(x)
    x_sum = np.sum(x_exp, axis=1, keepdims=True)
    s = x_exp / x_sum

    return s
```

Vectorization

Implement the L1 and L2 loss functions

Exercise 8 - L1

$L_1(\hat{y}, y) = \sum_{i=0}^{m-1}|y^{(i)} - \hat{y}^{(i)}|$

np.abs() calculates the absolute difference between each pair of predicted and actual values

Sum the absolute differences to obtain the L1 loss value

For example: `y = [0.9 0.2 0.1 0.4 0.9]`, `y = ]1 0 0 1 1]`, result L_1 = 1.1

=> Used to evaluate model performance and adjust model parameters in machine learning problems.

```
def L1(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)

    Returns:
    loss -- the value of the L1 loss function defined above
    """

    loss = np.sum(np.abs(y-yhat))

    return loss
```

Exercise 9 - L2

L2 calculates the sum of the squares of the difference between the model's prediction and the actual value.

$L_2(\hat{y},y) = \sum_{i=0}^{m-1}(y^{(i)} - \hat{y}^{(i)})^2$

For example: `y = [0.9 0.2 0.1 0.4 0.9]`, `y = [1 0 0 1 1]`, result L_2 = 0.43

```
def L2(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)

    Returns:
    loss -- the value of the L2 loss function defined above
    """

    loss = np.sum(np.square(y-yhat))

    return loss
```

Logistic Regression

Algorithms

Helper functions

Exercise 3 - sigmoid

Implement this function that return the about of formula sigmoid function $\sigma(z) = \frac{1}{1 + e^{-z}}$

```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.

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

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

    return s
```

Initializing parameters

Exercise 4 - initialize_with_zeros

Implement parameter initialization in the cell below. You have to initialize w as a vector of zeros. If you don't know what numpy function to use, look up `np.zeros()` in the Numpy library's documentation.

```
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) of type float
    """

    w = np.zeros((dim, 1))
    b = 0.0
    return w, b
```

Forward and Backward propagation

Exercise 5 - propagate

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, ..., a^{(m-1)}, a^{(m)})$
- You calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(a^{(i)})+(1-y^{(i)})\log(1-a^{(i)}))$

```
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-cat, 1 if cat) of size
(1, number of examples)

    Return:
    grads -- dictionary containing the gradients of the weights and bias
            (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)
    cost -- negative log-likelihood cost for logistic regression
```

```
    Tips:
    - Write your code step by step for the propagation. np.log(), np.dot()
    """

    m = X.shape[1]

    A = 1 / (1 + np.exp(-np.dot(w.T, X) - b))
    cost = np.sum(-(Y * np.log(A) + (1 - Y) * np.log(1 - A))) / m

    dw = np.dot(X, (A - Y).T) / m
    db = np.sum(A - Y) / m

    cost = np.squeeze(np.array(cost))


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

    return grads, cost
```

Optimization

Exercise 6 - optimize

Write down the optimization function. The goal is to learn w and b by minimizing the cost function J. For a parameter θ, the update rule is $ \theta = \theta - \alpha \text{ } d\theta$, where α is the learning rate.

```
def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009,
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.
```

```
    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters.
Use propagate().
        2) Update the parameters using gradient descent rule for w and b.
    """

    w = copy.deepcopy(w)
    b = copy.deepcopy(b)

    costs = []

    for i in range(num_iterations):
        grads, cost = propagate(w, b, X, Y)

        dw = grads["dw"]
        db = grads["db"]

        w = w - learning_rate * dw
        b = b - learning_rate * db

        if i % 100 == 0:
            costs.append(cost)

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

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

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

    return params, grads, costs
```

Exercise 7 - predict

The previous function will output the learned w and b. We are able to use w and b to predict the labels for a dataset X. Implement the `predict()` function. There are two steps to computing predictions:

1. Calculate $\hat{Y} = A = \sigma(w^T X + b)$

2. Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector `Y_prediction`. If you wish, you can use an `if`/`else` statement in a `for` loop (though there is also a way to vectorize this).

```
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)

    A = 1 / (1 + np.exp(-np.dot(w.T, X) - b))

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

        if A[0, i] > 0.5 :
            Y_prediction[0,i] = 1
        else:
            Y_prediction[0,i] = 0

    return Y_prediction
```

Model

Merge all functions into a model

Exercise 8 - model

Implement the model function. Use the following notation:

- `Y_prediction_test` for your predictions on the test set
- `Y_prediction_train` for your predictions on the train set
- parameters, grads, costs for the outputs of `optimize()`

```
def model(X_train, Y_train, X_test, Y_test, num_iterations=2000,
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.
    """
    w, b = initialize_with_zeros(X_train.shape[0])
    params, grads, costs = optimize(w, b, X_train, Y_train,
num_iterations, learning_rate, print_cost)
    w = params["w"]
    b = params["b"]
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    if print_cost:
        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
```