backpropagation

May 23, 2021

[1]:
```
import numpy as np
```

[2]:
```
def define_structure_of_NN(features, labels):
    '''
    Defines the structure of neural network with input layer, hidden layer and␣
 ↪output layer

    :params
     features : number of features in the dataset
     labels   : corresponding labels of the features

    :return
     input_units : number of units in input layer
     hidden_units: number of units in hidden layers
     output_units: number of units in output layer
    '''
    input_units = features.shape[0]
    hidden_units = 2
    output_units = labels.shape[0]
    return (input_units, hidden_units, output_units)
```

[3]:
```
def parameter_initialization(input_units, hidden_units, output_units):
    '''
    Initializes the weight matrices and bias vectors

    :parma
     input_units : number of units in input layer
     hidden_units: number of units in hidden layers
     output_units: number of units in output layer

    :return
     parameters : parameters of the neural net
    '''
    W1 = np.ones([hidden_units, input_units])
    b1 = np.zeros((hidden_units, 1))
    W2 = np.ones([output_units, hidden_units])
    b2 = np.zeros((output_units, 1))
```

```python
    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
    print('Initial W11 :\n',W1[0][0])

    return parameters
```

[4]:
```python
def forward_propagation(X, parameters):
    '''
    given the set of input features (X), we need to compute the activation␣
 ↪function for each layer

    :params
     X          : input features
     parameters : weights and bias

    :returns
     A2
     cache
    '''
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    Z1 = np.dot(W1, X) + b1
    A1 = Z1
    Z2 = np.dot(W2, A1) + b2
    A2 = 1/(1 + np.exp(-Z2))
    cache = {"Z1": Z1,"A1": A1,"Z2": Z2,"A2": A2}
    return A2, cache
```

[5]:
```python
def cross_entropy_cost(A2, Y, parameters):
    '''
    compute the cross-entropy cost

    :params
     A2
     Y          : labels
     parameters : weights and bias

    :returns
     cost : cross entropy cost
    '''
    m = Y.shape[1]
    logprobs = np.multiply(np.log(A2), Y) + np.multiply((1-Y), np.log(1 - A2))

    cost = - np.sum(logprobs) / m
    cost = float(np.squeeze(cost))
```

```python
    return cost
```

[6]:
```python
def backward_propagation(parameters, cache, X, Y):
    '''
    calculate the gradient with respect to different parameters

    :params
     parameters : weights and bias
     cache      : cached parameters from forward propagation
     X          : features
     Y          : labels

    :returns
     grads : gradients of the parameters
    '''
    m = X.shape[1]

    W1 = parameters['W1']
    W2 = parameters['W2']
    A1 = cache['A1']
    A2 = cache['A2']

    dZ2 = A2-Y
    dW2 = (1/m) * np.dot(dZ2, A1.T)
    db2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
    dW1 = (1/m) * np.dot(dZ1, X.T)
    db1 = (1/m)*np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2,"db2": db2}

    return grads
```

[7]:
```python
def gradient_descent(parameters, grads, learning_rate = 0.001):
    '''
    update the parameters using the gradient descent rule

    :param
     parameters    : weights and bias
     grads         : gradients obtained from back propagation
     learning rate : rate of learning of the neural net

    :returns
     parameters : updated weights and bias
    '''
    W1 = parameters['W1']
```

```python
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    parameters = {"W1": W1, "b1": b1,"W2": W2,"b2": b2}

    return parameters
```

[8]:
```python
def neural_network_model(X, Y, hidden_unit, num_iterations = 1000):
    '''
    build a neural network model with a single hidden layer

    :params
     X              : features
     Y              : labels
     hidden_unit    : number of hidden units
     num_iterations : number of iterations

    :returns
     parameters : weights and bias after final iteration
    '''
    np.random.seed(3)
    input_unit = define_structure_of_NN(X, Y)[0]
    output_unit = define_structure_of_NN(X, Y)[2]

    parameters = parameter_initialization(input_unit, hidden_unit, output_unit)

    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    for i in range(num_iterations):
        A2, cache = forward_propagation(X, parameters)
        cost = cross_entropy_cost(A2, Y, parameters)
        grads = backward_propagation(parameters, cache, X, Y)
        parameters = gradient_descent(parameters, grads)
        if i % 5 == 0:
```

```python
            print(f'iteration: {i:>4d}, cross entropy cost: {cost:10.6f}')

    return parameters
```

[9]:
```python
X = np.array([[1],[0]])
Y = np.array([1])
Y = Y.reshape(1, Y.shape[0])

parameters = neural_network_model(X, Y, 2, num_iterations=1)
```

```
Initial W11 :
 1.0
iteration:    0, cross entropy cost:   0.126928
```

[10]:
```python
print("Value of W11 after backpropagation :",parameters["W1"][0][0])
```

```
Value of W11 after backpropagation : 1.0
```