NN-Numpy

March 30, 2023

1 Fully Connected Neural Networks in NumPy (70 points)

In this assignment, you will implement a fully connected neural net with one hidden layer using NumPy and evaluate it on 2 artificial datasets: *flower* and *spiral*.

1.1 Write Your Name Here:

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version showing the code and the output of all cells, and save it in the same folder that contains the notebook file.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Submit **both** your PDF and notebook on Canvas.
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

[3]:
```
import numpy as np
from scipy.sparse import coo_matrix
from numpy.random import randn, randint
from numpy.linalg import norm
import matplotlib.pyplot as plt

import utils

np.random.seed(1)
```

2.1 1. The Neural Network model (50 points)

In this part, you will write code to:

1. Run forward propagation to compute the softmax output of the network.
2. Compute the cost for the logistic regression model.

3. Compute the gradient of the cost.
4. Update the parameters using the gradient.
5. Compute the model predictions.

It is important to vectorize your code so that it runs quickly on larger datasets.

2.1.1 Forward Propagation (10 points)

You will need to write code for the `forward()` function, which computes and returns the softmax outputs in A3, using the forward propagation equations. Use ReLU on the hidden layer, and also use a separate bias vector for the softmax layer. The function also returns a cache with the A and Z values for the hidden and output layers. Make sure that you prevent overflow when computing the softmax probabilities, as shown on the slides and the LR assignment.

[1]:
```
def forward(X, params):
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']

    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Compute Z2, A2, Z3, A3 (the softmax output).




    ## ----------------------------------------------------------------

    cache = {'Z2': Z2,
             'A2': A2,
             'Z3': Z3,
             'A3': A3}

    return A3, cache
```

2.1.2 Compute the Cost (10 points)

Write the `cost()` function which should compute the average negative log-likelihood + the regularization term where `decay` is the hyper-parameter.

[ ]:
```
def cost(X, y, params, decay):
    W1 = params['W1']
    W2 = params['W2']

    m = X.shape[1]
    groundTruth = coo_matrix((np.ones(m, dtype = np.uint8),
                              (y, np.arange(m)))).toarray()
```

```
    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Compute cost of NN model.




    ## ---------------------------------------------------------------

    return cost
```

2.1.3 Compute the Gradient (20 points)

Write the `backward()` function to compute the gradient, using the backpropagation equations.

[ ]:
```
def backward(X, y, params, cache, decay):
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']

    A1 = X
    Z2 = cache['Z2']
    A2 = cache['A2']
    Z3 = cache['Z3']
    A3 = cache['A3']

    m = X.shape[1]
    groundTruth = coo_matrix((np.ones(m, dtype = np.uint8),
                              (y, np.arange(m)))).toarray()

    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Compute gradients dW2, db2, dW1, db1.




    ## ---------------------------------------------------------------

    dParams = {'dW1': dW1,
               'db1': db1,
               'dW2': dW2,
               'db2': db2}

    return dParams
```

2.1.4 Gradient Update (5 points)

Write the function `updateParams()` for performing the gradient update.

[ ]:
```
def updateParams(params, dParams, learning_rate):
    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Use gradients in dParams to update params in params.




    ## ----------------------------------------------------------------
```

2.1.5 Model Predictions (5 points)

Write the function `nnPredict()` for computing the labels predicted by the model on the test examples given as argument.

[1]:
```
def nnPredict(X, params):
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']

    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Compute label predictions of the NN model.

    pred = np.zeros(X.shape[1])


    ## ----------------------------------------------------------------

    return pred
```

2.1.6 Train the NN using gradient descent

The `nnTrain()` function uses all the above to train the NN.

[ ]:
```
def nnTrain(X, y, n_y, hyperparams):
    # Set hyper-parameters.
    decay = hyperparams[1]
    learning_rate = hyperparams[2]
    num_epochs = hyperparams[3]

    # Randomly initialize parameters
    params = initParams(n_x, n_h, n_y)
```

```
    # Gradient descent loop.
    for epoch in range(num_epochs):
        # Forward propagation.
        a3, cache = forward(X, params)

        # Backward propagation.
        dParams = backward(X, y, params, cache, decay)

        # Gradient update.
        updateParams(params, dParams, learning_rate)

        # Print cost every 1000 iterations.
        if epoch % 1000 == 0:
            print("Epoch %d: cost %f" % (epoch, cost(X, y, params, decay)))

    return params
```

2.1.7 Miscellaneous functions

[ ]:
```
def initParams(n_x, n_h, n_y):
    W1 = 0.01 * randn(n_h, n_x)
    b1 = np.zeros((n_h, 1))
    W2 = 0.01 * randn(n_y, n_h)
    b2 = np.zeros((n_y, 1))

    params = {'W1': W1,
              'b1': b1,
              'W2': W2,
              'b2': b2}

    return params


def ravelGrads(params):
    dW1 = params['dW1']
    db1 = params['db1']
    dW2 = params['dW2']
    db2 = params['db2']

    return np.hstack((dW1.ravel(), db1.ravel(), dW2.ravel(), db2.ravel()))


def relu(x):
    return np.maximum(0, x)
```

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))


def tanh(x):
    e = np.exp(2 * x - 1)
    return (e - 1) / (e + 1)
```

2.2 2. Gradient Checking of the NN Model (10 points)

In this part you will write code to compute the gradient numerically, and then compare the numerical gradient with the analytical gradient. They should be very close **when using an activation function that is differentiable everywhere**. For this step to work, temporarily replace the $ramp(z) = max(0, z)$ activation function of ReLU neurons on the hidden layer with the sigmoid $\sigma(z)$ activation function to make them logistic neurons (make sure both `forward()` and `backward()` functions are updated accordingly).

The reason gradient checking does not work when using ReLU neurons is that the activation $max(0, z)$ is not differentiable at 0, and when parameters are very close to 0 in `backward()` we calculate the derivative of $max(0, z)$ as 0 if z is negative and as 1 if z is positive, whereas the numerical gradient procedure will output approximately $\frac{\epsilon - 0}{2\epsilon} = 0.5$, which is far from either 0 and 1. For gradient checking to work, we need to use an activation function that is differentiable everywhere, such as sigmoid.

2.2.1 Numerical Gradient (10 points)

Implement the *computeNumericalGradient()* function. You can reuse code from the LR assignment.

[3]:
```
def computeNumericalGradient(J, theta):
    """ Compute numgrad = computeNumericalGradient(J, params)

    params: a dictionary of parameters
    J: a function that outputs a real-number and the gradient.
    Calling y = J(params) will return the function value at params.
    """

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


    # Initialize numgrad with zeros.
    numd_W1 = np.zeros(W1.shape)
    numd_b1 = np.zeros(b1.shape)
    numd_W2 = np.zeros(W2.shape)
    numd_b2 = np.zeros(b2.shape)
```

```
    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Implement numerical gradient checking, and return the result in numgrad.
    epsilon = 1e-4


    ## ---------------------------------------------------------------

    num_grad = {'dW1': numd_W1, 'db1': numd_b1, 'dW2': numd_W2, 'db2': numd_b2}

    return num_grad
```

2.2.2 Gradient Checking

Check that the numerically computed gradients are sufficiently close to the gradient computed analytically.

[ ]:
```
# For debugging purposes, you may wish to reduce the size of the input data
# in order to speed up gradient checking.
# Here, we create a synthetic dataset using random data for testing.
X, y, n_y = randn(8, 100), randint(0, 2, 100, dtype = np.uint8), 2

n_x = X.shape[0]
m = X.shape[1]

n_h = 5

decay = 1e-3

# Randomly initialize parameters
params = initParams(n_x, n_h, n_y)

a3, cache = forward(X, params)
dParams = backward(X, y, params, cache, decay)

dNumParams = computeNumericalGradient(lambda p: cost(X, y, p, decay), params)

rdParams = ravelGrads(dParams)
rdnParams = ravelGrads(dNumParams)

# Use this to visually compare the gradients side by side.
print(rdnParams.shape)
print(rdParams.shape)
print(np.stack((rdnParams, rdParams)).T)

# Compare numerically computed gradients with those computed analytically.
# The difference should be small.
```

```
# In our implementation, these values are usually less than 1e-7.
diff = norm(rdnParams - rdParams) / norm(rdnParams + rdParams)
print(diff)
sys.exit(0)
```

2.3 3. Experimental Evaluations (5 points)

In this part, you will train and evaluate the NN model on 2 artificial datasets: *flower* and *spiral*.

2.3.1 Evaluate the NN model on the *flower* dataset.

[ ]:
```
# Load flower data.
X, y, n_y = utils.load_flower_dataset()

# Train a NN with one hidden layer of size 20, for 20,000 epochs,
# with a learning rate of 0.05, and an L2 decay of 0.
hyperparams = 20, 0, 0.05, 20000

params = nnTrain(X, y, n_y, hyperparams)

pred = nnPredict(X, params)

# Accuracy is the proportion of correctly classified images
# Our Flower accuracy: 86.50%
acc = # YOUR CODE HERE
print('Accuracy: %0.3f%%.' % (acc * 100))

# Plot the decision boundary.
utils.plot_decision_boundary(lambda x: nnPredict(x, params), X, y)
plt.title("Neural Network")
plt.savefig('flower-boundary.jpg')
plt.show()
```

2.3.2 Evaluate the LR model on the *spiral* dataset.

[ ]:
```
# Load spiral data.
X, y, n_y = utils.load_spiral_dataset()

# Train a NN with one hidden layer of size 100, for 10,000 epochs,
# with a learning rate of 1, and an L2 decay of 0.001.
hyperparams = 100, 0.001, 1, 10000

params = nnTrain(X, y, n_y, hyperparams)

pred = nnPredict(X, params)

# Accuracy is the proportion of correctly classified images
```

```
# Our Spiral accuracy: 98.67%
acc =  # YOUR CODE HERE
print('Accuracy: %0.3f%%.' % (acc * 100))

# Plot the decision boundary.
utils.plot_decision_boundary(lambda x: nnPredict(x, params), X, y)
plt.title("Neural Network")
plt.savefig('spiral-boundary.jpg')
plt.show()
```

2.4 4. Bonus (20 points)

- Use the *sklearn.neural_network.MLPClassifier* class from sklearn to run the 2 experimental evaluations above.

http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

- Anything extra goes here.

[ ]:

2.5 5. Analysis (5 points)

Include here a nicely formatted report of the results, comparisons. Include explanations and any insights you can derive from the algorithm behavoir and the results. This section is important, so make sure you address it appropriately.