CSC7343

HW1

Due Feb. 25th (23:59 central time). Include all your code in one .py file and upload it in moodle on or before the due time.

Your task in this homework is to implement a neural network by completing the following code for the class MLP. The neuron network should have several hidden layers, whose structure will be specified by the "config" python dictionary passed to the constructor of the class. The output layer of the network consists of one single neuron. It gives the probability that a data point belongs to class 1. All neurons in the hidden layers use **relu** as activation function while the single output neuron uses **sigmoid**.

Let $\{x^{(i)}, t^{(i)}\}$ be a set (or a mini batch) of N training examples and their labels and let $\{o^{(i)}\}$ be the output of the network when $x^{(i)}$ is the input. The training of the network should use the mean squared error:

$$E = \sum_i \left(o^{(i)} - t^{(i)}\right)^2 / N.$$

A skeleton of the class is given in the following, with the description/specification of the member functions in italic. (You may want to define more classes such as a layer class.) **You should implement these functions yourself, not by using some existing library/package. You can use the "numpy" package for matrix/vector computation.**

```
class MLP(object):
  def __init__(self, config):
```

__config__: a python dictionary providing the configuration of the neural network. It contains two entries:

> *input_dim: int, the dimension of the input*
>
> *layers: list of int, the number of neurons in each hidden layer. E,g., [10, 20] means the first hidden layer has 10 neurons and the second has 20.*

```
  def __call__(self, data):
```

compute the output of the neural network.

__data__: numpy ndarray of size (n x m) (i.e., a matrix) with dtype float32. Each row is a data point (total n data points) and each column is a feature (m features).

__return__: numpy ndarray of size (n,), the output from the neuron in the output layer for the n data points.

```
  def compute_gradients(self, data, label):
```

Compute the gradients of the parameters for the total loss

__data__: numpy ndarray of size (n x m) (i.e., a matrix) with dtype float32. Each row is a data point (total n data points) and each column is a feature (m features).

***label**: numpy ndarray of size (n,) (i.e., a vector) with dtype float32. The labels of the n data points.*

***return**: list of tuples where each tuple contains the partial derivatives (two numpy ndarrays) for the kernel and the bias of a layer. For example, suppose the network has two hidden layers, the list would look like: [(dedk_1, dedb_1), (dedk_2, dedb_2), (dedk_o, dedb_o)] where dedk_o is the partial derivative for the kernel of the output neuron and dedb_o is the derivative for the neuron's bias. (Note the number of tuples in the returned list is: # of hidden layers + 1.)*

```
def get_params(self):
```

***return**: the parameters of the neural network as a list of tuples [(kernel_1, bias_1), (kernel_2, bias_1), ...]. Note the parameters for the output neuron should be the last tuple in the list. To make it uniform, return the bias of the output neuron as a vector of single entry.*

```
def set_params(self, ps):
```

Set the parameters of the network

***ps**: a list of tuples in the form of [(kernel_1, bias_1), (kernel_2, bias_2), ...] (last tuple for the output neuron, the bias will be given as a vector of single entry). Calling the function sets the parameters of the network to the values given in ps.*

Once the class is implemented, we should be able to use it to create a neural network, train and make predictions using code similar to the following:

Create network:

```
nn = MLP({'input_dim':100, 'layers':[10, 20]})
```

Train:

```
for epoch in range(n_iter):
    loss = 0
    for b in make_batches():
    # make_batches return the indices of a mini batch
         batch_X = data[b]
         batch_y = label[b]
         loss += numpy.mean(numpy.square(nn(batch_X) - batch_y))
         gs = nn.compute_gradients(batch_X, batch_y)
         ps = nn.get_params()
         u = [(p[0]-lr*g[0], p[1]-lr*g[1]) for p, g in zip(ps, gs)]
         # lr is the learning rate
         nn.set_params(u)
    print(loss/len(label))
```

Make predictions:

```
p = (nn(data)>0.5).astype('float32')
```