About MLP

I used Multilayer Perceptron (MLP), which can have multiple hidden layers in between except for the input and output layers. The simplest MLP contains only one hidden layer, i.e., a three-layer structure. In my project, two hidden layers are used, and the principle is roughly the same as that of a single hidden layer.

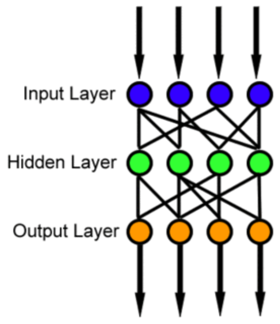

Figure 1 Structure of Multilayer Perceptron

From the above figure, the multilayer perceptron layers are fully connected to each other. (Meaning, any neuron in the upper layer is connected to all neurons in the lower layer). The bottom layer of the multi-layer perceptron is the input layer, the middle layer is the hidden layer, and finally the output layer.

Since the hidden layer and the input layer are fully connected, assuming that the input layer is represented by a vector X, the output of the hidden layer is f(W1X+b1), where W1 is the weight (also called the connection coefficient), b1 is the bias, and the function f can be the commonly used sigmoid function or tanh function.

The hidden layer to the output layer can be viewed as a multiclass logistic regression, or softmax regression, so the output of the output layer is softmax(W2X1+b2), and X1 denotes the output of the hidden layer, f(W1X+b1).

Therefore, all parameters of the MLP are the connection weights between each layer and the bias, including W1, b1, W2, b2. For a specific problem, solving the best parameters is an optimization problem, and the easiest way to solve the optimization problem is gradient descent (SGD): first initialize all parameters randomly, and then train it iteratively, continuously calculating the gradient and updating the parameters until it satisfies until a certain condition is met (e.g., when the error is small enough and the number of iterations is large enough).

Code to implement the MLP process.

(1) Import the necessary python modules

```
import os
import sys
import time
import numpy
import theano
```

```python
import theano.tensor as T
```

(2) Define the MLP model (HiddenLayer + LogisticRegression)
This section defines the basic "building blocks" of the MLP, namely the HiddenLayer and LogisticRegression, which have been mentioned above.

HiddenLayer
The implicit layer we need to define the connection coefficients W, bias b, input, output, the specific code as well as the interpretation is as follows.

```python
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,
                 activation=T.tanh):

        """
This is the class that defines the hidden layer. First, it is clear: the input of the hidden layer is the input, and the output is the number of neurons in the hidden layer. The input layer is fully connected to the hidden layer.

Suppose the input is an n_in dimensional vector (or n_in neurons), and the hidden layer has n_out neurons, then because it is fully connected, there are n_in*n_out neurons.

Therefore, the size of W is (n_in,n_out), n_in rows and n_out columns, and each column corresponds to the connection weight of each neuron in the hidden layer.

b is the bias, the hidden layer has n_out neurons, so b when n_out dimensional vector.
rng, the random number generator, numpy.random.RandomState, is used to initialize W.

input is used to train the model, not the input layer of MLP, the number of neurons in the input layer of MLP is n_in, and the size of input is (n_example,n_in), one sample per line, that is, each line as the input layer of MLP.

activation: activation function, defined here as the function tanh
        """

self.input = input # The input of class HiddenLayer is the input passed in.

"""
Code to be GPU compatible, then W, b must use dtype=theano.config.floatX, and defined as theano.shared
In addition, the initialization of W has a rule: if the tanh function is used, it is evenly between -sqrt(6./(n_in+n_hidden)) and sqrt(6./(n_in+n_hidden))
Draw values to initialize W. If the sigmoid function, multiply the above by another 4 times.
"""
```

```
# If W is not initialized, it is initialized according to the above method.
# The reason for adding this judgment is that sometimes we can initialize W with the trained
parameters

        if W is None:
            W_values = numpy.asarray(
                rng.uniform(
                    low=-numpy.sqrt(6. / (n_in + n_out)),
                    high=numpy.sqrt(6. / (n_in + n_out)),
                    size=(n_in, n_out)
                ),
                dtype=theano.config.floatX
            )
            if activation == theano.tensor.nnet.sigmoid:
                W_values *= 4
            W = theano.shared(value=W_values, name='W', borrow=True)

        if b is None:
            b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
            b = theano.shared(value=b_values, name='b', borrow=True)

# Initialize the W and b of class HiddenLayer with the W and b defined above
        self.W = W
        self.b = b

#The output of HidenLayer
        lin_output = T.dot(input, self.W) + self.b
        self.output = (
            lin_output if activation is None
            else activation(lin_output)
        )

# Parameters of the hidden layer
        self.params = [self.W, self.b]
```

LogisticRegression(softmax regression)

```
"""
Defining the classification layer, Softmax regression
In the deeplearning tutorial, it is straightforward to consider LogisticRegression as Softmax.
And the logistic regression we know as class two is the LogisticRegression when n_out=2

"""
#Parameter description.
```

```
#input, the size is (n_example,n_in), where n_example is the size of a batch.
#Because we use Minibatch SGD for training, so input is defined like this
#n_in, i.e. the output of the previous layer (hidden layer)
#n_out, the number of output categories

class LogisticRegression(object):
    def __init__(self, input, n_in, n_out):

#W size is n_in rows and n_out columns, and b is an n_out dimensional vector. That is, each output corresponds to a column of W and an element of b.
        self.W = theano.shared(
            value=numpy.zeros(
                (n_in, n_out),
                dtype=theano.config.floatX
            ),
            name='W',
            borrow=True
        )

        self.b = theano.shared(
            value=numpy.zeros(
                (n_out,),
                dtype=theano.config.floatX
            ),
            name='b',
            borrow=True
        )

#input is (n_example,n_in), W is (n_in,n_out), dot product to get (n_example,n_out), add bias b
#then as the input of T.nnet.softmax, we get p_y_given_x
#so each row of p_y_given_x represents the probability of each sample being estimated for each category
#hint: b is an n_out dimensional vector, summed with the (n_example,n_out) matrix, which internally actually copies n_example b first.
#then each row of the (n_example,n_out) matrix is added with b
self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

#argmax returns the maximum subscript, which happens to be the category. axis=1 means operate by row.
        self.y_pred = T.argmax(self.p_y_given_x, axis=1)

#params, Parameters of LogisticRegression
```

```
        self.params = [self.W, self.b]
```

Example: MLP with three layers.

```
class MLP(object):
    def __init__(self, rng, input, n_in, n_hidden, n_out):

        self.hiddenLayer = HiddenLayer(
            rng=rng,
            input=input,
            n_in=n_in,
            n_out=n_hidden,
            activation=T.tanh
        )

# Use the output of the hidden layer hiddenLayer as the input of the classification layer logRegressionLayer, thus connecting them

        self.logRegressionLayer = LogisticRegression(
            input=self.hiddenLayer.output,
            n_in=n_hidden,
            n_out=n_out
        )


# The above has defined the basic structure of the MLP, the following are the other parameters or functions of the MLP model

# Regularization terms: common L1, L2_sqr
        self.L1 = (
            abs(self.hiddenLayer.W).sum()
            + abs(self.logRegressionLayer.W).sum()
        )

        self.L2_sqr = (
            (self.hiddenLayer.W ** 2).sum()
            + (self.logRegressionLayer.W ** 2).sum()
        )


# Loss function Nll (also called cost function)
        self.negative_log_likelihood = (
            self.logRegressionLayer.negative_log_likelihood
        )
```

```

# inaccuracies
        self.errors = self.logRegressionLayer.errors

# Parameters of MLP
        self.params = self.hiddenLayer.params +
    self.logRegressionLayer.params
        # end-snippet-3
```

In MLP, besides the hidden and classification layers, loss functions, regularization terms are also defined, which are used in solving the optimization algorithm.
In my project, solver='lbfgs', because it works better than adam when dealing with small-scale data, and, after 200 iterations, it finally gets an accuracy of 0.98. However, due to the possible problem of sample imbalance, the sample imbalance may make the model 'lazy', which is like if there are 100 judgment questions and the answer to 90 questions in the period are True, then if we don't look at the questions, we can achieve 90% correct rate by choosing True for all 100 questions. This problem can be achieved by modifying the weights of different categories.

Reference

1. Multilayer perceptron - Wikipedia
2. Ruck, D.W., Rogers, S.K. and Kabrisky, M., 1990. Feature selection using a multilayer perceptron. *Journal of Neural Network Computing*, *2*(2), pp.40-48.
3. Raghu, S. and Sriraam, N., 2017. Optimal configuration of multilayer perceptron neural network classifier for recognition of intracranial epileptic seizures. Expert Systems With Applications, 89, pp.205-221.
4. Ramchoun, H., Idrissi, M.A.J., Ghanou, Y. and Ettaouil, M., 2016. Multilayer Perceptron: Architecture Optimization and Training. Int. J. Interact. Multim. Artif. Intell., 4(1), pp.26-30.
5. Pal, S.K. and Mitra, S., 1992. Multilayer perceptron, fuzzy sets, classifiaction.
6. Nassif, A.B., Ho, D. and Capretz, L.F., 2013. Towards an early software estimation using log-linear regression and a multilayer perceptron model. Journal of Systems and Software, 86(1), pp.144-160.
7. Taud, H. and Mas, J.F., 2018. Multilayer perceptron (MLP). In Geomatic Approaches for Modeling Land Change Scenarios (pp. 451-455). Springer, Cham.
8. Tam, K.Y., 1991. Neural network models and the prediction of bank bankruptcy. Omega, 19(5), pp.429-445.
9. Hill, T., Marquez, L., O'Connor, M. and Remus, W., 1994. Artificial neural network models for forecasting and decision making. International journal of forecasting, 10(1), pp.5-15.