Programming Assignment 5: Neural Network

Part 1: Group member and contribution

Bin Zhang (5660329599): Report part2 and neural network with keras.
Yihang Chen (6338254416): Report part 3 and neural network with sklearn.
Hanzhi Zhang (4395561906): Report part 4.
All of us implemented Back Propagation algorithm for Feed Forward Neural Networks without libraries (except for reading ”pgm” file).

Part 2: Implementation of Neural Network:

Neural Network is a machine learning algorithm that looks complex but is easy to understand. It can be seen as a stacking of multiple perceptrons. Here we use the following notations for the sake of implementation convenience, for the layer l, we suppose it has m perceptrons, and for the layer $l-1$, we suppose it has k perceptrons, then:

$$x^{(l)}: ouput\ , shape: (m, 1)$$

$$x^{(0)}: special\ case\ for\ x^{(l)}, shape: (d, 1)$$

$$w^{(l)}: weights, shape: (k, m\ )$$

$$b^{(l)}: bias, shape: (m, 1\ )$$

$$S^{(l)}: linear\ forward, S^{(l)} = w^{(l)}.T \cdot x^{(l-1)} + b^{(l)}\ \ shape: (m, 1\ )$$

$$\delta^{(l)}: \partial e/\partial S\ for\ lth\ layer, shape: (m, 1\ )$$

In this assignment, we first use PIL to help us read images and then use nested list to store each sample data: each sample is stored as a list and each attribute of this sample is stored as an item in the list. The format is like [[..], [..], [..]]. And for the labels, we use another list to store the corresponding label for each image. When we build a neural network, given the sharing of parameters and variables, we decide to represent the model as a class. When we instantiate, we use an instance variable parameter to store $w^{(l)}$, $b^{(l)}$ and $\delta^{(l)}$. The variable parameters are a dictionary, the format is like:

```
{ "W1":shape (960, 100),
  "W2":shape (100, 1),
```

```
"b1":shape (100, 1),
"b2":shape (1, 1),
"Delta1": shape (100, 1),
"Delta2": shape (1, 1)}
```

So when we process our forward propagation, we can store the result into the parameters, and when process our backpropagation, we can retrieve the corresponding $w^{(l)}$, $b^{(l)}$ and $\delta^{(l)}$ from the dictionary.

However, there is a problem that we encounter, that is: when we test our code, we need almost 100 seconds to finish an epoch, which means we have to spend nearly 28 hours to get the updated parameters for prediction. To solve this problem, we use vectorization, which increases the processing speed from 28 hours to less than 1 minute.

The vectorization mainly focuses on three parts:

1. reformat the train set and test set: we give up the nested format and turn to nparray for help:

```
X shape: (sample_number, 960, 1)
Y shape : (sample_number,)
```

2. calculation of $\delta^{(l)}$

```
self.parameters['Delta'+str(layer_level)]=np.dot(W_l,Delta_l)*X_L
*(1-X_L)
```

3. update of $w^{(l)}$, $b^{(l)}$

```
self.parameters['W'+str(l)]=WL-
self.learning_rate*np.dot(X_last_l,Delta_l.T)
self.parameters['b'+str(l)]=BL-
self.learning_rate*self.parameters['Delta' + str(l)]
```

while in the former version, we use two for loops to realize them.

Another problem we meet is the accuracy problem, although we try to train our neural network for different epochs and different learning rate, the performance is not so good, the accuracy we get is between 75% to 80%. We try several times and there might be a possible reason:

We use the standard squared error as the final cost function, but in fact it' s not a convex function for sigmoid function and increasing epoch times cannot make sure the model gets to the global optimal. So to verify our assumption, we try the log loss as our error function and after 1000 epochs with learning rate 0.01, we get almost 90% accuracy on test data. So that could be a possible reason for our problem. And to further verify our assumption, we user the third party libraries to build network and check the accuracy on the test data.(Will be described in the part3).

Result: after 1000 epochs, the accuracy of our neural network is

```
0.7951807228915663
```

Part 3: Software Familiarization

We choose sklearn python library for the implementation of Back Propagation algorithm for the Feed Forward Neural Network. And we set the network parameters as following:

Network structure: input layer → hidden layer 100 → output perceptron
Perceptron function: sigmoid function
Training epoch: 1000
Learning rate: 0.1

The difference with the assignment is that the library initializes the weight using the initialization method recommended by Glorot et al. The initialization weight part is shown below:

```
    def _init_coef(self, fan_in, fan_out):
        # Use the initialization method recommended by
        # Glorot et al.
        factor = 6.
        if self.activation == 'logistic':
            factor = 2.
        init_bound = np.sqrt(factor / (fan_in + fan_out))

        # Generate weights and bias:
        coef_init = self._random_state.uniform(-init_bound, init_bound,
                                               (fan_in, fan_out))
        intercept_init = self._random_state.uniform(-init_bound, init_bound,
                                                    fan_out)
        return coef_init, intercept_init
```

So based on the structure of our network, the weight was initialized between [-0.25, 0.25]

For default, the parameter of early_stopping is set to False. If set true, the network will set aside 10% of the training data as validation, and will stop training when the score is not improving at least of n_iter_no_change(default = 10) turns. So the network can stop before getting the max iteration limit.

What is worth mention is that the error function used in the library is log-loss, which is different with the error function taught in the class. And it was set in the source code and for now we failed to find a API to change that function.

To solve the function compassion problem, we use another library: keras. From its API, we customize our learning rate, epochs, error function, layer, etc. as the professor requires. And after training, we get the result on the test data.

```
  1/184 [..............................] - ETA: 0s - loss: 3.4182e-04 - accuracy: 1.0000
 58/184 [========>.....................] - ETA: 0s - loss: 0.1864 - accuracy: 0.7586
120/184 [==================>...........] - ETA: 0s - loss: 0.1791 - accuracy: 0.7750
179/184 [============================>.] - ETA: 0s - loss: 0.1834 - accuracy: 0.7598
184/184 [==============================] - 0s 851us/step - loss: 0.1876 - accuracy: 0.7554
Epoch 1000/1000

  1/184 [..............................] - ETA: 0s - loss: 5.3653e-05 - accuracy: 1.0000
 62/184 [=========>....................] - ETA: 0s - loss: 0.1899 - accuracy: 0.7419
126/184 [===================>..........] - ETA: 0s - loss: 0.1762 - accuracy: 0.7778
184/184 [==============================] - 0s 832us/step - loss: 0.1736 - accuracy: 0.7880

32/83 [==========>...................] - ETA: 0s
83/83 [==============================] - 0s 245us/step
test_acc: 0.7951807379722595
```

Part 4: Applications

Neural Network can be applied on the image process. In this assignment, we use NN to predict whether there is a ”down” gestures in the image. With the same method, we can also apply it on the face detection. The face of people can be labeled by NN. We can even use this algorithm to recognize the face expression of people (smile or not smile) in the image. Moreover, NN can also be applied on Super Resolution Imaging which is the process of recovering a high-resolution image from a low-resolution one. The most interesting application I find is art style change by NN. NN transfer an image into an artist style which is similar to The Starry Night.

Figure 1 Original Image

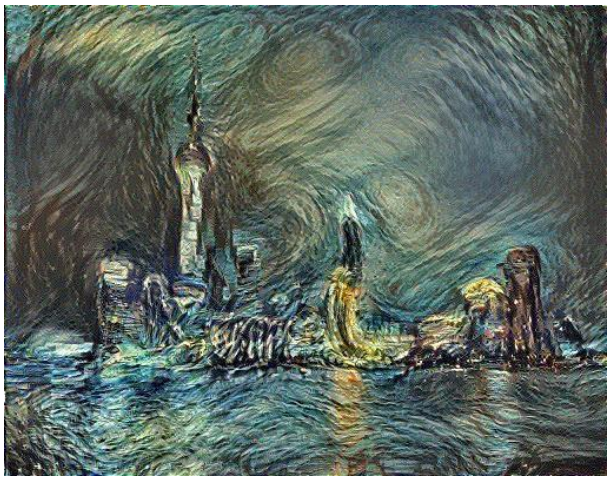

Figure 2 Image with The Starry Night Style Processed by NN