Deep Learning Homework 1 Report – Steven Jonathan (0760825)

1. Regression

(a) The prediction of heating load is calculated using the following equation.

$$E(w) = \sum_{n=1}^{N} \left(t_n - y(x_n; w)\right)^2$$

Where $y(x_n; w)$ is the predicted output that we get after one iteration of feed forward and back propagation, while t_n is the target or the desired output of the system. The first step to obtain the output is by feed forwarding the data from the feature neurons through hidden layers all the way to output neurons, by using the following equation.

$$f\left(y(x_n; w)\right) = \omega^T x + bias$$

From the above equation we can see that the output from each neuron need to be activated by an activation function. Since this is linear regression, we can use identity activation function that is output is the same as input, in other words $f(x) = x$. Therefore, we are ready to implement the equations in the program.

```
for iteration in range(0, iterations): #number of epochs
      output_history = []
      for index in range(0, len(fdata)):
      #Forward Propagation
         for j in range(0, len(self.weights)):
             if j == 0:
                  w = self.weights[j]
                  wt = np.transpose(w[:-1]) #omit the bias weight, then
             transpose it
                  y = np.dot(wt, fdata[index])
                  z = y + w[-1] #add with bias weight
                  self.sum_result[j] = z
                  k = 0
             else:
                  w = self.weights[j]
                  wt = np.transpose(w[:-1])
                  y = np.dot(wt, self.sum_result[k])
                  z = y + w[-1]
                  self.sum_result[j] = z
                  k += 1 #pointer
      output_history.append(z) #intended solely for output result storage,
start from index == 0
```

From the above code snippet we can see that we are not including the bias weight in the transpose, because the bias is always 1. However, it does not mean that we don't use the bias weights in the system at all, the proof is `z = y + w[-1]` that we finally add the bias after the output which means the equation (2) is satisfied.

The system should learn by updating the weights for each iteration, which means there is a need of back propagation. Because we want to back propagate using this loss function, the derivative will be.

$$\frac{\partial E(w)}{\partial y} = y_n - t_n$$

Since it will be used throughout the whole iterations and the results will be a sequence of derivative products, we can simply remove the summation operator from the equation. Therefore, we can implement this derivative function in the program.

```
dError = output_history[index] - target[index]
```

However, the result of substraction is not stored inside the variable `dError` for our own convenience, because `dError` is not used as a list. The next step is to derivate the error of output to previous hidden layer with respect to the products of feed forward and corresponding weight by using chain rule.

$$\frac{\partial E(w)}{\partial h} = \frac{\partial E(w)}{\partial y}\frac{\partial y}{\partial h}$$

$$\frac{\partial E(w)}{\partial w_{ij}} = \frac{\partial E(w)}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial w_{ij}}$$

After the result of derivatives of error function with respect to the weight is obtained, we should update the weight so that the network can learn. Because of the error function, we use minimization algorithm to compute the updated weight.

$$w_{i,j\,new} = w_{i,j\,old} - \lambda \frac{\partial E(w)}{\partial w_{ij}}\frac{\partial E(w)}{\partial h}$$

Where λ is the learning rate, $\frac{\partial E(w)}{\partial w_{ij}}$ is the derivative of error function with respect to corresponding weights (that connects each neurons) and $\frac{\partial E(w)}{\partial h}$ is the derivative of error function with respect to the products of feed forward. Therefore, we are now ready to implement this into the program.

```
#Backward Propagation
    dError = output_history[index] - target[index] #derivative of loss
function SSE
    for j in range(0, len(self.hnodes) + 1):
        l = len(self.hnodes) - j #for reversal of for index
       if j == 0: #l == 3
           self.sum_result[l - 1] = np.append(self.sum_result[l - 1], 1)
           weight_nobias = self.weights[j - 1]
           dSum = np.dot(weight_nobias[:-1], dError)
           self.dElist[j] = dSum
           for k in range(0, (self.hnodes[j - 1]) + 1): #+1 for the bias
               updated_weight = self.weights[j - 1][k] -
np.dot(np.dot(learningrate, dError), self.sum_result[l - 1][k])
               self.weights[j - 1][k] = updated_weight #replace existing
weights with new weights
       else:
           if l == 0:
            #do weight update for weights between feature layer and first
hidden layer here
               inputs = np.append(fdata[index], 1)
               for m in range(0, (self.hnodes[l])):
                   for n in range(0, self.fnodes + 1): #+1 for the bias
                       updated_weight = self.weights[l][n, m] -
np.dot(np.dot(learningrate, self.dElist[j - 1][m]), inputs[n])
                       self.weights[l][n, m] = updated_weight
           else: #when j == 1, l == 2 and when j == 2, l == 1
               self.sum_result[l - 1] = np.append(self.sum_result[l - 1], 1)
               weight_nobias = self.weights[l]
               dSum = np.dot(weight_nobias[:-1], self.dElist[j - 1]) #when j
== 1 equals to self.weights[1], and so on
               self.dElist[j] = dSum
               for m in range(0, (self.hnodes[l])):
                   for n in range(0, self.hnodes[l - 1] + 1): #+1 for the
bias
                   updated_weight = self.weights[l][n, m] -
np.dot(np.dot(learningrate, self.dElist[j - 1][m]), self.sum_result[l -
1][n])
                   self.weights[l][n, m] = updated_weight
```

We then can evaluate the performance of the neural network by using the root-mean-square error.

$$E_{RMS}(w) = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(t_n - y(x_n; w)\right)^2}$$

Where *N* is the size of feature data, and the error is calculated by dividing the running sum by total amount of data and the result will be root squared. We can interpret the equation in our program:

```
SSE = np.sum([(target[index] - output_history[index]) ** 2 for index in range(0, len(fdata))])
RMSE = np.sqrt(SSE/len(fdata)) #fdata is feature data
```

From the above code snippet it is clear that `SSE` is loss function (SSE is sum of squared errors) and `RMSE` is the root mean square error.

(b) **(1)** The network architecture for the neural network is the same as provided in the homework, that is 16 – 15 – 10 – 10 – 1 with exception 16 being the number of features. Since we need to consider the data that contains glazing area distribution of 0. We initialize the neural network using class neural_network().

```
class neural_network():
      def __init__(self, fnodes, hnodes, onodes):
      . . .
NN = neural_network(16, [15, 10, 10], 1)
```

From above code snippet is clear that `fnodes = 16, hnodes[0] = 15, hnodes[1] = 10, hnodes[2] = 10` and `onodes = 1`. This mean that hidden layers used in this network are 3 layers for 15 neurons in 1st hidden layer, 10 neurons in 2nd hidden layer and 10 neurons in 3rd hidden layer. The output layer only has 1 neuron because we only need to find the heating load prediction. However, we must be careful considering the data has categorical feature for orientation and glazing area distribution. The data for orientation are 2 = north, 3 = east, 4 = south, 5 = west and for glazing area distribution are 0 = undefined, 1 = uniform, 2 = north, 3 = east, 4 = south and 5 = west. Therefore, it needs to be hot-encoded first before it can be used.

```
#one hot encoding
def one_hot_encoding(data, category):
      size = len(data)
      hot_encode = np.zeros((size, category))
      data -= np.min(data)
      for i in range(0, size):
            hot_encode[i, int(data[i])] = 1
      return hot_encode
```

From above code snippet we will get hot encoded result both for orientation and glazing area distribution. The hot encoded orientation are 2 = [0, 0, 0, 1], 3 = [0, 0, 1, 0], 4 = [0, 1, 0, 0] and 5 = [1, 0, 0, 0]. While hot encoded glazing area distribution are 0 = [0, 0, 0, 0, 0, 1], 1 = [0, 0, 0, 0, 1, 0], 2 = [0, 0, 0, 1, 0, 0], 3 = [0, 0, 1, 0, 0, 0], 4 = [0, 1, 0, 0, 0, 0] and 5 = [1, 0, 0, 0, 0, 0]. We still need to append these hot-encoded data so it can be further processed by using the following code.

```
feature_cell = [0, 1, 2, 3, 4, 6]
train_data = []
for i in range(0, len(new_dataset)):
      temp = [new_dataset[i, j] for j in feature_cell]
      [temp.append(orientation_encoding[i, j]) for j in range(0, 4)]
      [temp.append(glazing_dist_encoding[i, j]) for j in range(0, 6)]
      train_data.append(temp)
```

Since the result of hot-encoded data for both orientation and glazing area distribution are 10 features, combined with 6 other non-categorical features makes the total of 16 features for our network.

(2) The learning curve of this neural network is plotted from RMSE data for 1000 times of iteration.

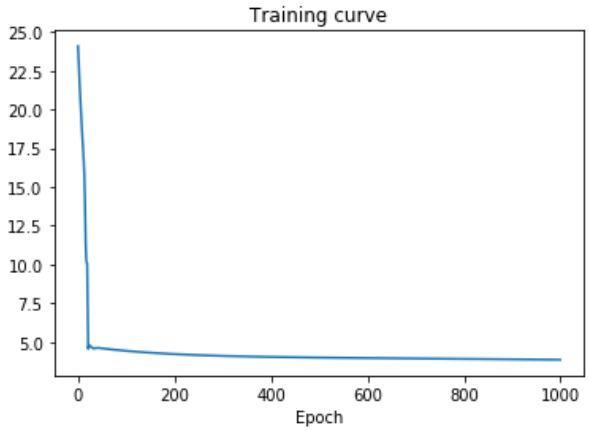

Fig. 1 Training curve

(3) The training RMSE is calculated inside the training process, we can see the calculation method in the following code snippet.

```
for iteration in range(0, iterations):
      . . .
      SSE = np.sum([(target[index] - output_history[index]) ** 2 for index in
range(0, len(fdata))])
      RMSE = np.sqrt(SSE/len(fdata))
      print('Training RMSE:', iteration, RMSE)
      RMSElist.append(RMSE)
print("\nTraining RMSE: ", RMSE)
```

The result of training RMSE by the time this report made is `Training RMSE: 3.850668348818966`.

```
Training RMSE: 990 3.8527837228303734
Training RMSE: 991 3.852547560146503
Training RMSE: 992 3.8523116775508806
Training RMSE: 993 3.8520760751395082
Training RMSE: 994 3.8518407529995615
Training RMSE: 995 3.851605711209509
Training RMSE: 996 3.8513709498392164
Training RMSE: 997 3.85113646895007
Training RMSE: 998 3.8509022685950742
Training RMSE: 999 3.850668348818966

Training RMSE:  3.850668348818966
```

Fig. 2 Training RMSE value

(4) The test RMSE is calculated inside the test process, we can see the calculation method in the following code snippet.

```
for index in range(0, len(test_set)):
      . . .
      testError += ((output_history[index] - target[index]) ** 2)
      test_rms = np.sqrt(testError/len(test_set))
      print("\nTest RMSE: ", test_rms)
```

The result of training RMSE by the time this report made is `Test RMSE:  0.29769857`.

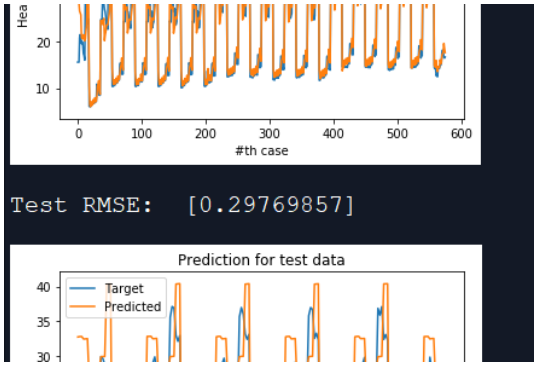

Fig. 3 Test RMSE

(5) The regression result for training data with labels is obtained by plotting the target superimposed with the predicted output in respect to amount of data in the training sets (75% of total data).

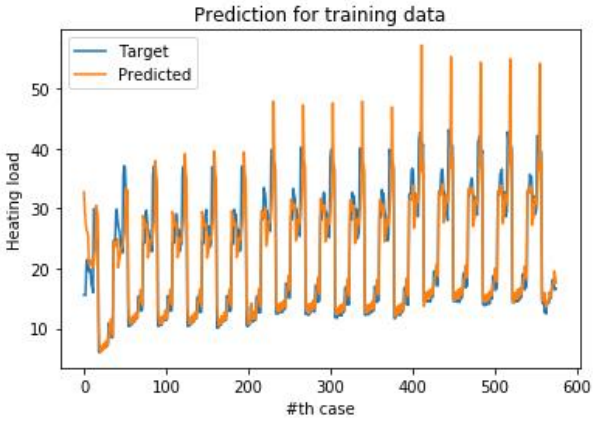

Fig. 4 Prediction for training data

(6) The regression result for test data with labels is obtained by plotting the target superimposed with the predicted output in respect to amount of data in the test sets (25% of total data).

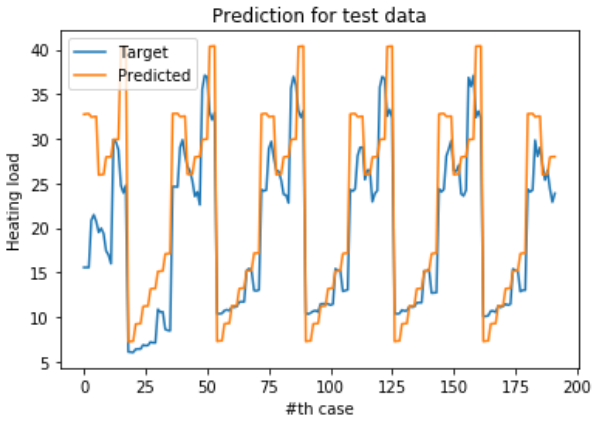

Fig. 5 Prediction for test data

(c) The feature selection can be done either automatically using an algorithm and manual way. This can be done by selecting certain sets of features and analysing the error result to see which sets of features that will affect the errors.

2. Classification

(a) The classification of the Ionosphere data is calculated using this equation.

$$E(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk}\, log y_k(x_n, w)$$

We are using softmax as the activation function for the output result of feed forward.

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a}$$

Softmax function take an N-dimensional arrays of real numbers and transforms it into an arrays of real number between (0, 1). The result of softmax is a "soft" version of maximum function, hence the name. It performs by breaks the whole 1 (its maximum value) with maximal elements get the largest portion of the value distribution, but other smaller elements get some of it as well. Therefore, it is unlike the maximum function which is getting the maximum value of 1.

This property of softmax function that outputs a probabily distribution is that makes it suitable for interpretation in classification tasks. Therefore, instead of using linear activation function like the first problem, we can implement the activation function of softmax by passing the results of feed forward into the softmax activation function.

```
def softmax(self, X):
      exps = np.exp(X)
      return (exps / np.sum(exps))

      . . .
for iteration in range(0, iterations): #number of epochs
      output_history = []
      for j in range(0, len(self.weights)):
            if j == 0:
            w = self.weights[j]
            wt = np.transpose(w[:-1]) #omit the bias weight, then transpose it
            y = np.dot(wt, fdata[index])
            z = y + w[-1] #add with bias weight
            z1 = self.softmax(z) #softmax activation function
            self.sum_result[j] = z1
            k = 0
      else:
            w = self.weights[j]
            wt = np.transpose(w[:-1])
            y = np.dot(wt, self.sum_result[k])
            z = y + w[-1]
            z1 = self.softmax(z) #softmax activation function
            self.sum_result[j] = z1
            k += 1 #pointer
      output_history.append(z) #intended solely for output result storage,
start from index == 0
```

To make this neural network learn we need to do the back propagation of the classification problem, the derivative of loss function with softmax is.

$$\frac{\partial E(w)}{\partial y_k} = y_k - t_n$$

Since it will be used throughout the whole iterations and the results will be a sequence of derivative products, we can simply remove the summation operator from the equation. Therefore, we can implement this derivative function in the program.

```
dError = output_history[index] - target[index]
```

The activation function of softmax is considered to be part of the chain rule. Because this is a non-linear network we can't drop the derivative of softmax out of the derivative chain. The next step is to derivate the error of output to previous hidden layer with respect to the products of feed forward and corresponding weight by using chain rule.

$$\frac{\partial E(w)}{\partial h} = \frac{\partial E(w)}{\partial \sigma^{y}(y)} \frac{\partial \sigma^{y}(y)}{\partial y} \frac{\partial y}{\partial h}$$

$$\frac{\partial E(w)}{\partial w_{ij}} = \frac{\partial E(w)}{\partial \sigma^{y}(y)} \frac{\partial \sigma^{y}(y)}{\partial y} \frac{\partial h}{\partial w_{ij}}$$

After the result of derivatives of error function with respect to the weight is obtained, we should update the weight so that the network can learn. Because of the error function, we use minimization algorithm to compute the updated weight.

$$w_{i,j\,new} = w_{i,j\,old} - \lambda \frac{\partial E(w)}{\partial w_{ij}} \frac{\partial E(w)}{\partial h}$$

Where λ is the learning rate, $\frac{\partial E(w)}{\partial w_{ij}}$ is the derivative of error function with respect to corresponding weights (that connects each neurons) and $\frac{\partial E(w)}{\partial h}$ is the derivative of error function with respect to the products of feed forward.

(b) (1) The neural network structure is different from the first problem since there are more than 2 times of feature sizes and 2 outputs for the classification problem, it is provided by using 34 – 15 – 10 – 10 – 2 architecture. We initialize the neural network using class neural_network().

```
class neural_network():
      def __init__(self, fnodes, hnodes, onodes):
      . . .
      . . .
NN = neural_network(34, [15, 10, 10], 2)
```

From the above code snippet it is clear that `fnodes = 34, hnodes[0] = 15, hnodes[1] = 10, hnodes[2] = 10` and `onodes = 2`. This mean that hidden layers used in this network are 3 layers for 1 nodes/neurons in 1st hidden layer, 10 nodes in 2nd hidden layer and 10 nodes in 3rd hidden layer. However, we use 2 neurons in output layer because it is binary classification problem. Therefore, the output must be hot-encoded first before it can be used (the output is either 'g' for good or 'b' for bad).

```
#one hot encoding
def one_hot_encoding(data,category):
    h = []
    for i in range(0, len(data)):
        if data[i] == 'g':
            h.append(0) #0 for good output
        elif data[i] == 'b':
            h.append(1) #1 for bad output
        else:
            pass
    b = np.zeros((len(data), category))
    b[np.arange(len(data)), h] = 1
    return b
```

However, it is a different from the first problem since the target is the one that need to be hot-encoded. Therefore, it is obvious that we don't need to append the hot-encoded target into the feature sets.

```
output_encoding = one_hot_encoding(csvarray[:,34], 2)
. . .
```

```
NN = neural_network(34, [15, 10, 10], 2)
NN.training(new_dataset[0:int(training_set)], output_encoding, 0.00007, 390)
```

(2) The following learning curve of this neural network is plotted from Cross Entropy data for 390 times of iteration.

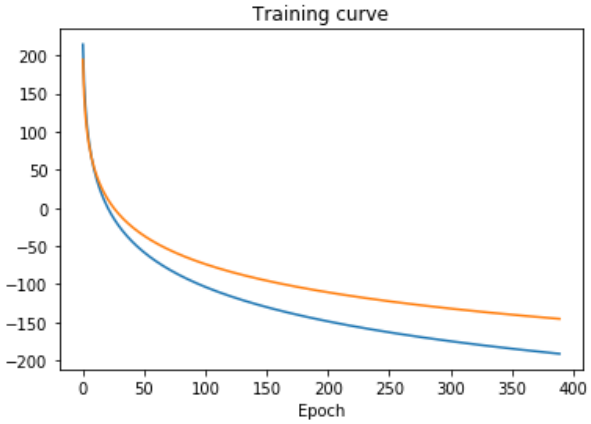

Fig. 6 Training curve

(3) The following are the result of training Cross Entropy error for each probability of both predicted outputs corresponding to the targets.

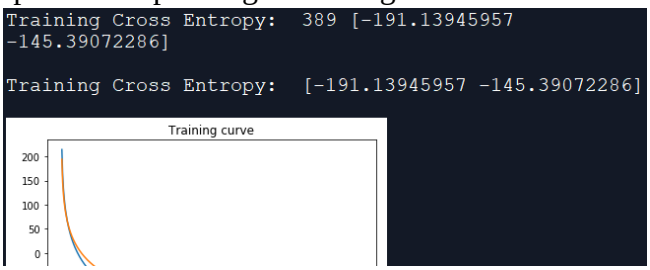

Fig. 7 Training Cross Entropy Error

(4)

(c)