CS545 Machine Learning - Assignment 4

Yu, Qiu

November 1, 2015

Contents

1 Part 1: Activation Functions

1.1 Formula Derivation

1. Recall that:

$$\sigma(x) = \frac{1}{1 + exp(-x)} \tag{1}$$

$$\tag{2}$$

So,

$$\begin{aligned} \sigma(x) + \sigma(-x) &= \frac{1}{1+e^{-x}} + \frac{1}{1+e^{x}} && (3) \\ &= \frac{(1+e^{x}) + (1+e^{-x})}{(1+e^{-x})(1+e^{x})} && (4) \\ &= \frac{2 + e^{x} + e^{-x}}{1 + e^{-x} + e^{x} + 1} && (5) \\ &= 1 && (6) \end{aligned}$$

Therefore,

$$\sigma(x) + \sigma(-x) = 1 \tag{7}$$
$$\sigma(x) = 1 - \sigma(-x) \tag{8}$$

2.

$$\begin{aligned}
2\sigma(2x) - 1 &= \frac{2}{1+e^{-2x}} - 1 && (9)\\
&= \frac{2-1-e^{-2x}}{1+e^{-2x}} && (10)\\
&= \frac{1-e^{-2x}}{1+e^{-2x}} && (11)\\
&= \frac{e^{-x}(e^{x}-e^{-x}}{e^{-x}(e^{x}+e^{-x})} && (12)\\
&= \frac{e^{x}-e-x}{e^{x}+e-x} && (13)\\
& && (14)
\end{aligned}$$

Recall that:

$$tanh(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{15}$$

So,

$$tanh(x) = 2\sigma(2x) - 1 \tag{16}$$

1.2 Equivalent Network

According to the textbook 7.3, for a single hidden layer, the formula to get the prediction for the output layer is:

$$h(x) = \theta\left(w_{01}^{(2)} + \sum_{j=1}^{m} w_{j1}^{(2)} \cdot tanh(\sum_{i=0}^{d} w_{ij}^{(1)} x_i)\right) \tag{17}$$

Applying formula (16), then we got:

$$h(x) = \theta\left(w_{01}^{(2)} + \sum_{j=1}^{m} w_{j1}^{(2)} \cdot tanh(\sum_{i=0}^{d} w_{ij}^{(1)} x_i)\right) \tag{18}$$
$$h(x) = \theta\left(w_{01}^{(2)} + \sum_{j=1}^{m} w_{j1}^{(2)} \cdot \left(2\sigma(2\sum_{i=0}^{d} w_{ij}^{(1)} x_i) - 1\right)\right) \tag{19}$$
$$h(x) = \theta\left(w_{01}^{(2)} + 2\sum_{j=1}^{m} w_{j1}^{(2)} \cdot \sigma(\sum_{i=0}^{d} 2w_{ij}^{(1)} x_i) - \sum_{j=1}^{m} w_{j1}^{(2)}\right) \tag{20}$$
$$h(x) = \theta\left(\left(w_{01}^{(2)} - \sum_{j=1}^{m} w_{j1}^{(2)}\right) + 2\sum_{j=1}^{m} w_{j1}^{(2)} \cdot \sigma(\sum_{i=0}^{d} 2w_{ij}^{(1)} x_i)\right) \tag{21}$$

Let:

$$v_{01}^{(2)} = w_{01}^{(2)} - \sum_{j=1}^{m} w_{j1}^{(2)}, \quad j = (1, 2, ..., m) \tag{22}$$

$$v_{j1}^{(2)} = 2w_{ji}^{(2)}, \quad j = (1, 2, ..., m) \tag{23}$$

$$v_{ij}^{(1)} = 2w_{ij}^{(1)}, \quad i = (1, 2, ..., d) \quad j = (1, 2, ..., m) \tag{24}$$

Then:

$$h(x) = \theta \left(v_{01}^{(2)} + \sum_{j=1}^{m} v_{j1}^{(2)} \cdot \sigma(\sum_{i=0}^{d} v_{ij}^{(1)} x_i) \right) \tag{25}$$

Therefore, if we replace old weight vector w_{ij} with the new weight vector v_{ij} from formula $(22) \sim (24)$, we can get the result which applied logistic sigmod as activation function.

2 Multi-layer perceptrons

For the following decision boundary, we assign 4 functions to each line: h1, h2, h3, h4.

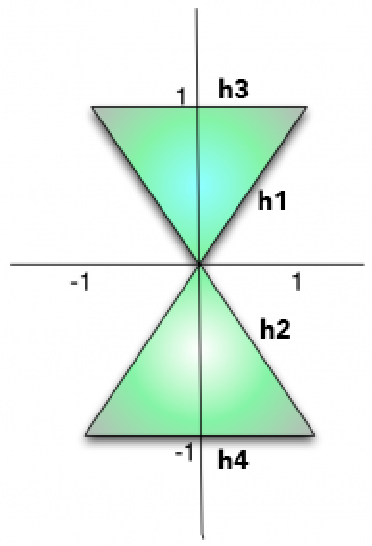


Figure 1: Decision Boundary

In order to achieve the specified area, we noticed that the area is alway at bottom of h3, on top of h4. For h1 and h2, the area is always at the same side of h1 and h2. More specifically, for any point (x, y) in the area, $sign(h1(x)) = sign(h2(x))$. So the multilayer perceptron implementation should be:

$$f(x) = h1 \cdot h2 \cdot \overline{h3} \cdot h4 + \overline{h1} \cdot \overline{h2} \cdot \overline{h3} \cdot h4 \tag{26}$$

3 Exploring neural networks for digit classification

3.1 Add Bias Term

First of all, the given implementation does not provide bias term correctly. In the given code:

```
def forward(self, x) :
   a = [x]
   for i in range(self.num_layers) :
         a.append(self.activation(np.dot(a[i], self.weights[i])))
   return a
```

It only applies bias term for the input layer correctly, but for the hidden layer, it computes the bias term w_0 based on the previous result, which is not right. So we have to modify it in order to get the correct bias term:

```
def forward(self, x):
   a = [x]
   for i in range(self.num_layers):
         ai = self.activation(np.dot(a[i], self.weights[i]))
         if (len(a) < self.num_layers):
               ai[-1] = 1
            a.append(ai)
   return a
```

3.2 Single hidden layer neural network

For single hidden layer neural network, we first modified test_digits method, so that it can get the accuracy from confusion matrix:

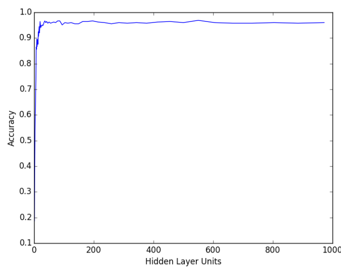
```
def test_digits_single_hidden_layer(hidden_layer_units):
      digits = load_digits()
      X = digits.data
      y = digits.target
      X /= X.max() # Normalize data

      nn = NeuralNetwork([64, hidden_layer_units, 10], 'logistic')
      X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                  test_size=0.25, random_state=0)
      labels_train = LabelBinarizer().fit_transform(y_train)
      labels_test = LabelBinarizer().fit_transform(y_test)
      nn.fit(X_train, labels_train, epochs=100)
      predictions = []
      for i in range(X_test.shape[0]):
            o = nn.predict(X_test[i])
            predictions.append(np.argmax(o))
      cm = confusion_matrix(y_test, predictions)
      correct_prediction = 0.0
      for i in range(cm.shape[0]):
            correct_prediction += cm[i][i]
      accuracy = correct_prediction / X_test.shape[0]
      return accuracy
```

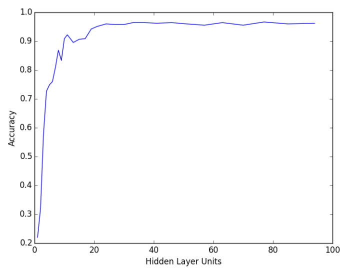
Then in order to plot the graph between accuracy and the number of hidden layer units, we have to use a loop to calculate accuracy under different number of hidden layer units. For speed purpose, we step 10% of current units number:

```
max_hidden_layer_units_1 = 1000
## single hidden layer
unit_1 = 1
```

```
units_1 = []
accuracy_1 = []
while unit_1 <= max_hidden_layer_units_1:
      units_1.append(unit_1)
      unit_1 += math.ceil(unit_1 / 10)
units_1 = units_1[1:]
for unit in units_1:
      accuracy_1.append(test_digits_single_hidden_layer(unit))

plt_units = np.asarray(units_1)
plt_accuracy = np.asarray(accuracy_1)
plt.plot(plt_units, plt_accuracy, "b")
plt.xlabel("Hidden Layer Units")
plt.ylabel("Accuracy")
plt.show()
```

Below is the graph we got:





As we can see, the relation between hidden layer units and accuracy is almost linear when units is relatively small, say less than 100, where underfitting happens. While when the hidden layer units is larger, the accuracy stays at a high level. For this dataset, overfitting is quite hard to happen.

3.3 Two hidden layers neural network

In order to add an additional layer to the neural network, we have to slightly modify the test_digits method again:

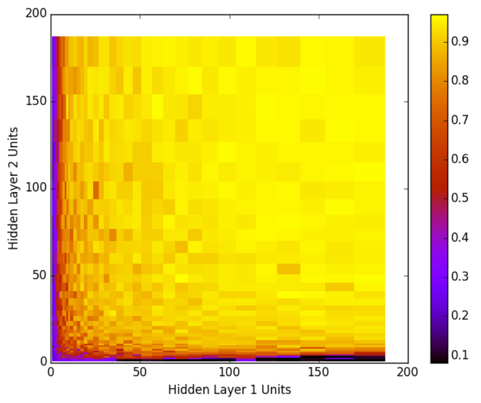
```
def test_digits_two_hidden_layers(hidden_layer1_units, hidden_layer2_units):
      digits = load_digits()
      X = digits.data
      y = digits.target
      X /= X.max() # Normalize data

      nn = NeuralNetwork([64, hidden_layer1_units, hidden_layer2_units, 10], 'logistic')
      X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                 test_size=0.25, random_state=0)
      labels_train = LabelBinarizer().fit_transform(y_train)
      labels_test = LabelBinarizer().fit_transform(y_test)
      nn.fit(X_train, labels_train, epochs=100)
      predictions = []
      for i in range(X_test.shape[0]):
            o = nn.predict(X_test[i])
            predictions.append(np.argmax(o))
```

```
        cm = confusion_matrix(y_test, predictions)
        correct_prediction = 0.0
        for i in range(cm.shape[0]):
              correct_prediction += cm[i][i]
        accuracy = correct_prediction / X_test.shape[0]
        print accuracy
        return accuracy
```

Then we use two loops to get the relationship between accuracy and number of both hidden layer units:

```
max_hidden_layer_units_2 = 200
unit_2 = 1
units_2 = []
accuracy_2 = []
while unit_2 <= max_hidden_layer_units_2:
      units_2.append(unit_2)
      unit_2 += math.ceil(unit_2 / 10)
units_2 = units_2[1:]
accuracy_mat_2 = np.zeros((len(units_2), len(units_2)))

for i in range(len(units_2)):
      for j in range(len(units_2)):
            accuracy_mat_2[i][j] = test_digits_two_hidden_layers(units_2[i], units_2[j])

plt_units_1 = np.asarray(units_2)
plt_units_2 = np.asarray(units_2)
plt.pcolor(plt_units_1, plt_units_2, accuracy_mat_2, cmap='gnuplot')
plt.xlabel("Hidden Layer 1 Units")
plt.ylabel("Hidden Layer 2 Units")
plt.colorbar()
plt.show()
```

Here is the figure we got:



From the figure above, we can see that when either one of the hidden layer has few units, the accuracy will be relatively low. When both hidden layer increase their units number, the accuracy increases dramatically. More specifically, when hidden layer units are under 50, the accuracy are normally below 80%, which is underfitting. And when both layer units are more than 50, the accuracy keeps at a high level, generally more than 90%.

3.4 Weight Decay

According to the chapter 7.4.1 on textbook, in order to add a weight decay term, we need to modify the error total as below:

$$E_{aug}(\mathbf{w}) = E_{in}(\mathbf{w}) + \frac{\lambda}{N}\sum_{l,i,j}(w_{ij}^{(l)})^2 \tag{27}$$

$$\frac{\partial E_{aug}(\mathbf{w})}{\partial \mathbf{W}^{(l)}} = \frac{E_{in}(\mathbf{w})}{\partial \mathbf{W}^{(l)}} + \frac{2\lambda}{N}\mathbf{W}^{(l)} \tag{28}$$

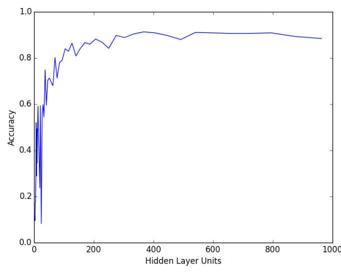
So we need to modify given code at fit method:

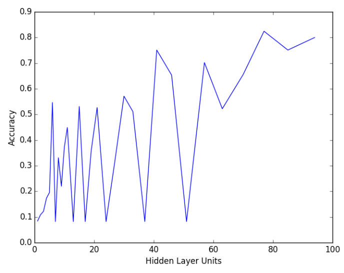
```
def fit(self, X, y, learning_rate=0.2, epochs=50, _lambda=0.0, linear=False):
      X = np.asarray(X)
      temp = np.ones((X.shape[0], X.shape[1] + 1))
      temp[:, 0:-1] = X # adding the bias unit to the input layer
      X = temp
      y = np.asarray(y)

      for k in range(epochs):
            I = np.random.permutation(X.shape[0])
            for i in I:
                  a = self.forward(X[i], linear)
                  deltas = self.backward(y[i], a, linear)
                  # update the weights using the activations and deltas:
                  for i in range(len(self.weights)):
                        layer = np.atleast_2d(a[i])
                        delta = np.atleast_2d(deltas[i])
                        N = len(a[-1])
                        self.weights[i] += learning_rate * (layer.T.dot(delta) + _lambda / N * self.wei
```

We modified the last line, where the weight vector got updated:

```
self.weights[i] += learning_rate * (layer.T.dot(delta) + _lambda / N * self.weights[i])
```

And we got the following figures:





We found that the accuracy performed worse than that without weight decay term. That is because the weight decay method is used for avoid overfitting situation. However, the given dataset is very well built, and it hardly has overfitting problem. We increased the hidden layer units number to 10000 but there is still no overfitting. Therefore, if we apply weight decay term, it will have an underfitting problem for this specific dataset.

3.5 Linear Activation Function

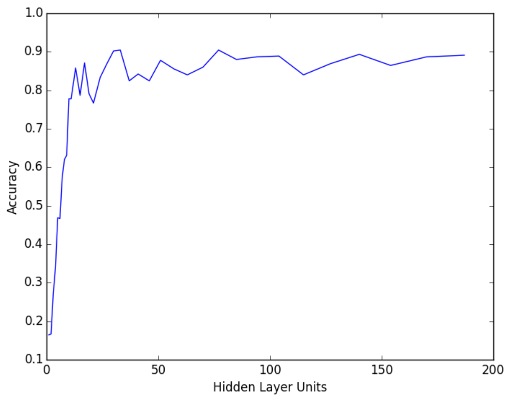
Normally, in a nerual network we offen use sigmod function as activation function, while in output layer, we often use a linear function. This is because that the sigmod function will only output values between 0 to 1. However, in the output layer, we have to compare the outputs to label y. A linear function can scale the output to arbitrary value, which is better for the comparison purpose.

We need to modify the code accordingly in order to apply linear activation function at output layer: If the layer is the last layer, we will apply linear function: $f = x$.

```
def forward(self, x, linear=False):
      a = [x]
      for i in range(self.num_layers):
            ai = self.activation(np.dot(a[i], self.weights[i]))
            if i == self.num_layers - 1 and linear == True:
                  ai = np.dot(a[i], self.weights[i])
            if (len(a) < self.num_layers):
                  ai[-1] = 1
            a.append(ai)
      return a
```

```
def backward(self, y, a, linear=False):
      if linear:
            deltas = [(y - a[-1])] * 1
      else:
            deltas = [(y - a[-1]) * self.activation_deriv(a[-1])]
      for l in range(len(a) - 2, 0, -1): # we need to begin at the second to last layer
            deltas.append(deltas[-1].dot(self.weights[l].T) * self.activation_deriv(a[l]))
      deltas.reverse()
      return deltas
```

And we got the following graph:


Accuracy
Hidden Layer Units