CS 545 ASSIGNMENT 4

Paresh Bhambhani

November 1, 2015

Contents

---

1 Activation Functions

1.1 Relationship between tanh and Logistic Sigmoid

In this section we will be proving:

1.1.1 $\sigma(-x) = 1 - \sigma(x)$

Proof:

$$\begin{aligned} \text{we have: } \sigma(\mathbf{x}) &= \frac{1}{1+\exp(-\mathbf{x})} \\ \sigma(-\mathbf{x}) &= \frac{1}{1+\exp(\mathbf{x})} = \frac{\exp(-\mathbf{x})}{1+\exp(-\mathbf{x})} = \frac{1+\exp(-\mathbf{x})-1}{1+\exp(-\mathbf{x})} \\ &= 1 - \frac{1}{1+\exp(-\mathbf{x})} \\ &= 1 - \sigma(-\mathbf{x}) \end{aligned} \tag{1}$$

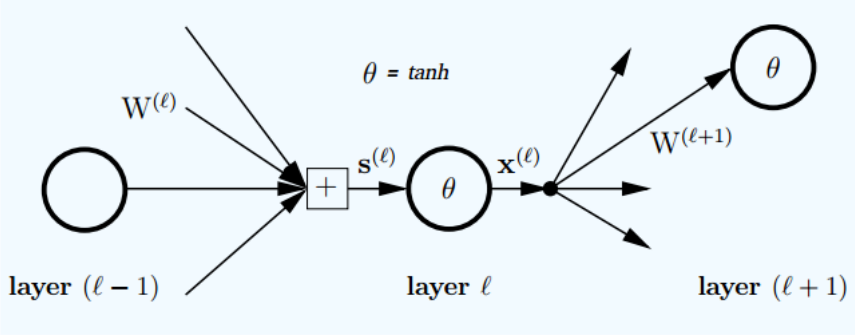
1.1.2 $\tanh(x) = 2\sigma(2x) - 1$

Proof:

$$\begin{aligned}
\text{we have: } \tanh(\mathbf{x}) &= \frac{\exp(\mathbf{x}) - \exp(-\mathbf{x})}{\exp(\mathbf{x}) + \exp(-\mathbf{x})} \\
&= \frac{(1 - \exp(-2\mathbf{x}))(\exp(\mathbf{x}))}{\exp(\mathbf{x}) + \exp(-\mathbf{x})} = \frac{1 - \exp(-2\mathbf{x})}{\frac{\exp(\mathbf{x}) + \exp(-\mathbf{x})}{\exp(\mathbf{x})}} \\
&= \frac{1 - \exp(-2\mathbf{x})}{1 + \frac{\exp(-\mathbf{x})}{\exp(\mathbf{x})}} = \frac{1 - \exp(-2\mathbf{x})}{1 + \exp(-2\mathbf{x})} \\
&= \frac{2 - 1 - \exp(-2\mathbf{x})}{1 + \exp(-2\mathbf{x})} = \frac{2}{1 + \exp(-2\mathbf{x})} - 1 \\
&= 2\sigma(2x) - 1
\end{aligned} \tag{2}$$

1.2 Designing Network with Sigmoid activation function to give same output as network with tanh activation function

We are given a network which has *tanh* as the activation function. The interconnection between the layers can be seen as shown below in Figure 1 from the lecture slide.



signals in	$\mathbf{s}^{(\ell)}$	$d^{(\ell)}$ dimensional input vector
outputs	$\mathbf{x}^{(\ell)}$	$d^{(\ell)} + 1$ dimensional output vector
weights in	$\mathrm{W}^{(\ell)}$	$(d^{(\ell-1)} + 1) \times d^{(\ell)}$ dimensional matrix
weights out	$\mathrm{W}^{(\ell+1)}$	$(d^{(\ell)} + 1) \times d^{(\ell+1)}$ dimensional matrix

Figure 1: Layer interconnection for *tanh* activation

For this given network the hypothesis $\mathbf{h}(\mathbf{x})$ is:

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{W^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \xrightarrow{W^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}^{(2)} \cdots \longrightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x})$$

In the section 1.1.2 we have proved that $\tanh(x) = 2\sigma(2x) - 1$. We use this to modify our network to have Sigmoid as activation function and still result in the same output as the previous network with the *tanh* as the activation function. The modified network is shown below in Figure 2

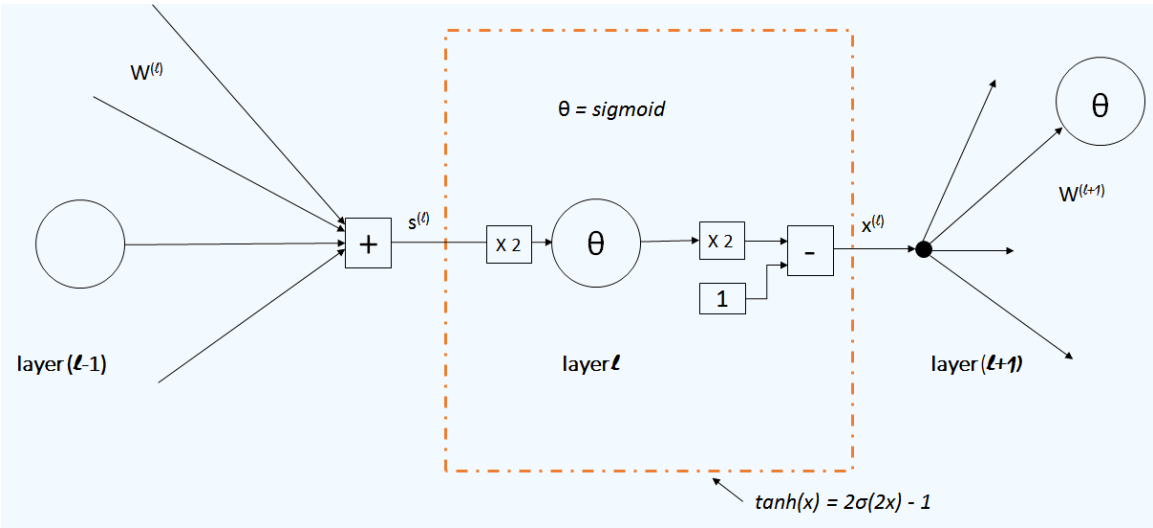


Figure 2: Layer interconnection for *Sigmoid* activation

For this new network the hypothesis $\mathbf{h}(\mathbf{x})$ will be:

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{2W^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}'^{(1)} \xrightarrow{W'^{(1)}} \mathbf{x}^{(1)} \xrightarrow{2W^{(2)}} \mathbf{s}^{(2)} \xrightarrow{\theta} \mathbf{x}'^{(2)} \cdots \longrightarrow \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}'^{(L)} \xrightarrow{W'^{(L)}} \mathbf{x}^{(L)} = h(\mathbf{x})$$

We can conclude from the Figure 2 and the new hypothesis above that the network is modified with the activation function as the *Sigmoid* function and to achieve the same output, the input weights to each layer are multiplied by 2 and an additional weight $W'^{(l)}$ is added at the output of activation function in each layer l which results in $\mathbf{x}^{(l)} = 2\mathbf{x}'^{(l)} - 1$.

2 Multi-layer perceptrons

We will be constructing a multi-layered perceptron that will result in the following decision boundary (Figure 3):

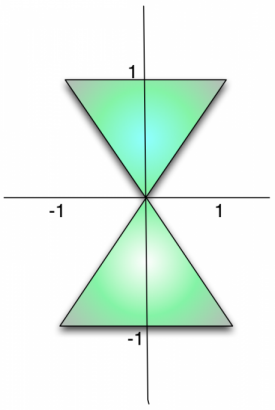
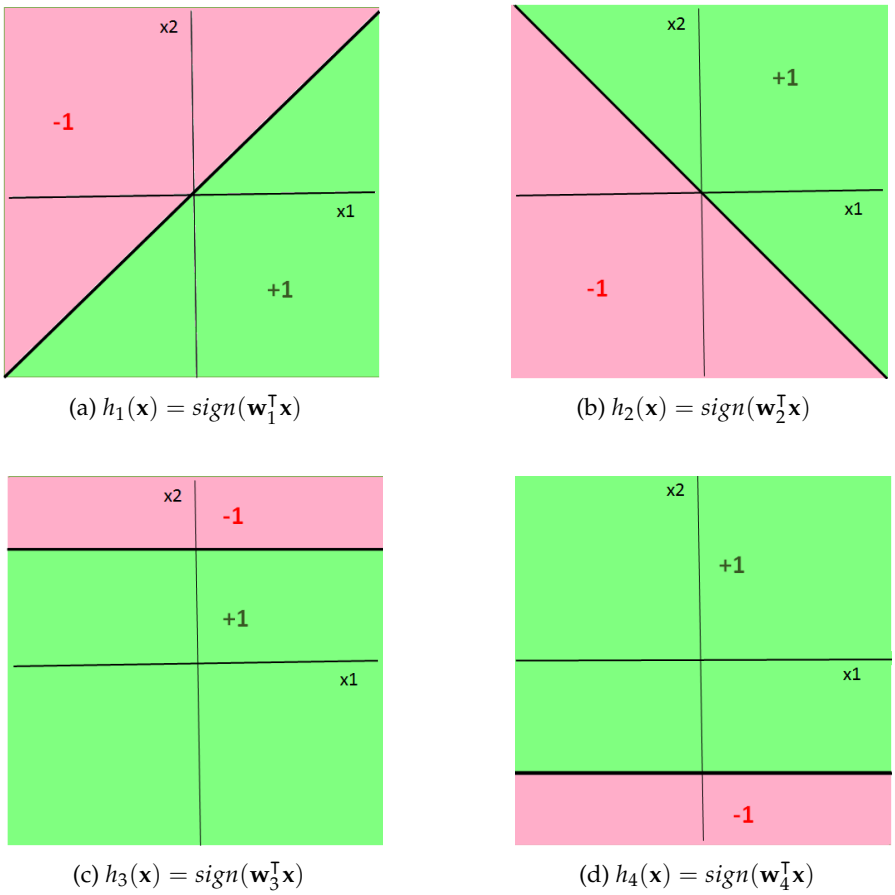


Figure 3: Decision Boundary

Such a decision boundary can be implemented using a combination of multiple linear classifiers as shown in Figure 4.



(a) $h_1(\mathbf{x}) = sign(\mathbf{w}_1^\intercal \mathbf{x})$ (b) $h_2(\mathbf{x}) = sign(\mathbf{w}_2^\intercal \mathbf{x})$

(c) $h_3(\mathbf{x}) = sign(\mathbf{w}_3^\intercal \mathbf{x})$ (d) $h_4(\mathbf{x}) = sign(\mathbf{w}_4^\intercal \mathbf{x})$

Figure 4: Individual Linear Classifiers

The required decision boundary can then be expressed as:

$$\mathbf{f} = \mathbf{h}_1 \overline{\mathbf{h}}_2 \mathbf{h}_4 + \overline{\mathbf{h}}_1 \mathbf{h}_2 \mathbf{h}_3 \tag{3}$$

This function results in the decision boundary as shown in Figure 5.

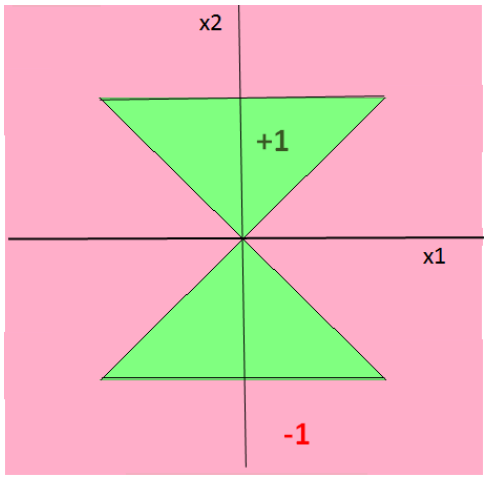


Figure 5: Decision Boundary from Multiple Linear Classifiers

3 Neural Networks for digit classification

3.1 Network accuracy as a function of the number of hidden units for a single-layer network

The network accuracy as a function of the no. of hidden units is shown in Figure 6.

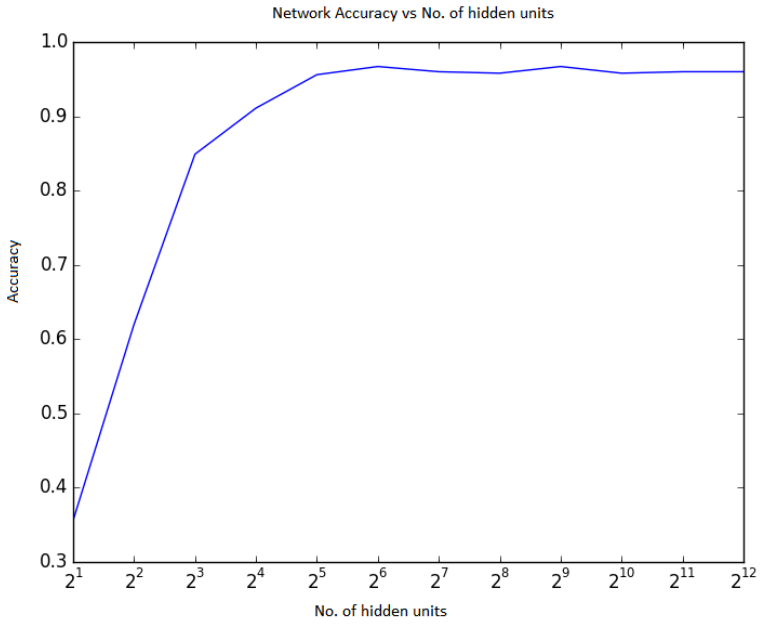
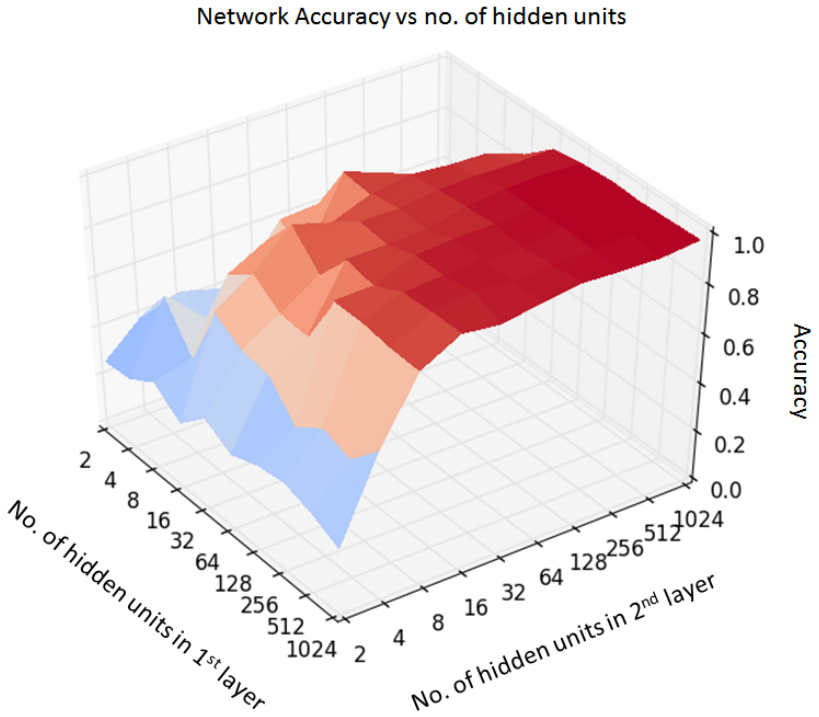


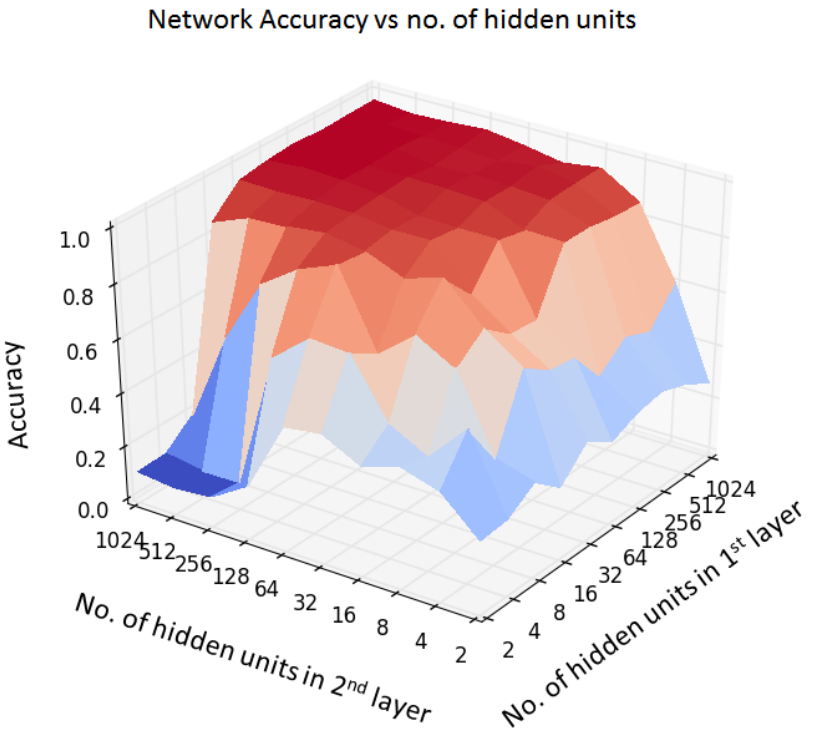
Figure 6: Network Accuracy vs No. of hidden units for a single layer network

As is visible from the Figure 6, the network under-fits when a very small no. of neurons are used. As the no. of hidden units is increased, the accuracy improves. There does not seem to be any over-fitting since the accuracy is high even for large number of hidden units (4096) with a maximum of 0.967.

3.2 Network accuracy as a function of the number of hidden units for a two-layer network



(a) Point of view -1



(b) Point of View - 2

Figure 7: Network Accuracy vs Number of Hidden units in layer 1 vs Number of Hidden units in layer 2

It can be seen from Figure 7a and Figure 7b that the network under-fits when the number of hidden units is small in either of the two layers. It is also observed that the accuracy is quite low when the number of hidden units is small in the first layer but very large in the second and vice versa. The accuracy increases when the number of hidden unit increases in both the layers with a maximum of 0.969. It is observed that there does not seem to be an over-fit even for large number of hidden units.
It is also observed that there is no significant difference in using two layer network over a single layer network for this dataset.

3.3 Adding weight decay regularization to the neural network

For Neural networks we update the weights as:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \bigtriangledown \mathbf{E}_{in}(\mathbf{w}(t)) \tag{4}$$

We need to find how the weights are updated after weight decay regularization is introduced.
In general, after addition of weight decay the augmented in-sample-error is given as:

$$\mathbf{E}_{aug}(\mathbf{w}) = \mathbf{E}_{in}(\mathbf{w}) + \lambda \mathbf{w}^{\intercal}\mathbf{w} \tag{5}$$

In our case the augmented in-sample-error will become:

$$\mathbf{E}_{aug}(\mathbf{w}) = \mathbf{E}_{in}(\mathbf{w}) + \frac{\lambda}{N}\sum_{l}\sum_{i}\sum_{j}(\mathbf{w}_{ij}^{(l)})^2 \tag{6}$$

Therefore,

$$\frac{\delta \mathbf{E}_{aug}(\mathbf{w})}{\delta \mathbf{W}^{(l)}} = \frac{\delta \mathbf{E}_{in}(\mathbf{w})}{\delta \mathbf{W}^{(l)}} + \frac{2\lambda}{N}\mathbf{w}^{(l)} \tag{7}$$

We can now calculate the weight vector update which is,

$$\begin{aligned} \mathbf{w}(t+1) &= \mathbf{w}(t) - \eta \bigtriangledown \mathbf{E}_{in}(\mathbf{w}(t)) - 2\eta\frac{\lambda}{N}\mathbf{w}(t) \\ &= \mathbf{w}(t)(1 - 2\eta\frac{\lambda}{N}\mathbf{w}(t)) - \eta \bigtriangledown \mathbf{E}_{in}(\mathbf{w}(t)) \end{aligned} \tag{8}$$

To incorporate this change, in the "def fit" section of the code where the weight is updated, a modification can be made as:

```
#Updating weight vector with weight decay
for i in range(len(self.weights)):
  layer = np.atleast_2d(a[i])
  delta = np.atleast_2d(deltas[i])
  self.weights[i] = self.weights[i]*(1 - ((2*lambda)/(X.shape[0])))
         + learning_rate * layer.T.dot(delta)
```

Here λ is the regularization parameter and the best λ can be found out using Grid search method and then used in this code.
As updated by Prof. Asa, the data does not demonstrate over-fitting.

3.4 Solving regression problems with Neural Networks

To solve regression problems using Neural networks we replace only the output transformation with an Identity function ($\mathbf{x} = \mathbf{x}$) for the output to be a real number. A Sigmoid or tanh function squishes the output to be between 0 and 1 but since we are doing regression we need output to linearly map the input.
To use Neural networks for solving regression, we need to modify only the activation function of the output layer while maintaining the sigmoid activation function for the rest of the layers. Hence we make changes in the sections of the code where activation functions are being used. It can be modified as shown in the Appendix.

Appendices

The code presented here shows just the changes that are needed in the original code for implementing the regression problem solving. (*This is not the complete code*)

```
def tanh(x):
   return np.tanh(x)

def tanh_deriv(x):
   return 1.0 - np.tanh(x)**2

def logistic(x):
   return 1/(1 + np.exp(-x))

def logistic_derivative(x):
   return logistic(x)*(1-logistic(x))

def linear(x):
   return x

def linear_derivative(x):
   return 1

class NeuralNetwork:
   def __init__(self, layers, activation) :
      """
      layers: A list containing the number of units in each layer.
            Should contain at least two values
      activation: The activation function to be used. Can be
            "logistic" or "tanh"
      """
      if activation == 'logistic':
         self.activation1 = logistic
         self.activation1_deriv = logistic_derivative
         self.activation2 = logistic
         self.activation2_deriv = logistic_derivative
      elif activation == 'tanh':
         self.activation1 = tanh
         self.activation1_deriv = tanh_deriv
         self.activation2 = tanh
         self.activation2_deriv = tanh_deriv
      elif activation == 'regression':
         self.activation1 = logistic
         self.activation1_deriv = logistic_derivative
         self.activation2 = linear
         self.activation2_deriv = linear_derivative
      self.num_layers = len(layers) - 1
      self.weights = [ np.random.randn(layers[i - 1] + 1, layers[i] + 1)/10 \
      for i in range(1, len(layers) - 1) ]
      self.weights.append(np.random.randn(layers[-2] + 1, layers[-1])/10)

   def forward(self, x) :
      """
      compute the activation of each layer in the network
      """
      a = [x]
      for i in range((self.num_layers)-1) :
```

```
            a.append(self.activation1(np.dot((a[i]), self.weights[i])))
            j=i
        a.append(self.activation2(np.dot(a[j+1], self.weights[j+1])))
        return a

    def backward(self, y, a) :
        """
        compute the deltas for example i
        """
        deltas = [(y - a[-1]) * self.activation2_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.activation1_deriv(a[l]))
        deltas.reverse()
        return deltas
```