Neural Networks

Harshil Shah
Computer Science
Colorado State University

October 31, 2015

Contents

---

1 Activation Functions

1.1 Relation Between Activation Functions

Two basic activation functions of neural networks are the *tanh* and *logistic sigmoid* functions. The *tanh* will produce output in range of (-1, 1) but *logistic sigmoid* will produce output in range of (0, 1).

For a given feature(x), their mathematical equations can be defined as below.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

&

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Now,

by putting $x = -x$ in above sigmoid equation

$$\sigma(-x) = \frac{1}{1+e^{x}}$$
$$= 1 - 1 + \frac{1}{1+e^{x}}$$
$$= 1 - (1 - \frac{1}{1+e^{x}})$$
$$= 1 - (\frac{1+e^{x}-1}{1+e^{x}})$$
$$= 1 - (\frac{e^{x}}{1+e^{x}})$$
$$= 1 - (\frac{1}{1+e^{-x}})$$
$$= 1 - \sigma(x)$$

Therefore,

$$\boxed{\sigma(-x) = 1 - \sigma(x)}$$

Now let's derive relation between two functions: *tanh* & *logistic sigmoid*

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$
$$= \frac{e^{2x} - 1}{e^{2x} + 1}$$
$$= \frac{e^{2x} - 1}{e^{2x} + 1} + 1 - 1$$
$$= \frac{e^{2x} - 1 + e^{2x} + 1}{e^{2x} + 1} - 1$$
$$= \frac{2e^{2x}}{e^{2x} + 1} - 1$$
$$= 2\frac{1}{1+e^{-2x}} - 1$$
$$= 2\sigma(2x) - 1$$

Therefore,

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

1.2 Network With The Same Function Using Logistic Sigmoid

Graphical representation of neural network with any activation function may look like as below figure 1.

In below figure, network contains *tanh* as activation function.

$$\theta(s) = tanh(s)$$

Each activation function will take $w_i^T x($ (where w_i = incoming weight vector for i^{th} neuron) as input and pass the result to neurons of next layer of network.

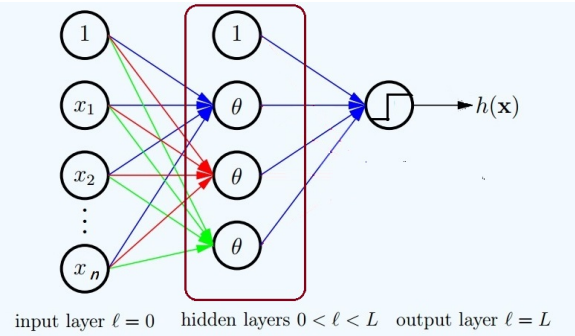

Figure 1: Graphical Representation Of Neural Network

In above figure 1, boxed section depicts implementation of activation function. Now, for getting same output by using *logistic sigmoid* as activation function instead of *tanh*, we can use formula derived in section 1.1 and implement it in boxed section.

$$\tanh(X) = 2\sigma(2X) - 1$$
$$Where : X = w^T x$$
$$\tanh(X) = \frac{e^X - e^{-X}}{e^X + e^{-X}}$$

So according to formula, instead of directly passing incoming signal from previous layer to tanh function, network needs to double the incoming signal and pass result to sigmoid function (i.e. similar to linear activation function). Then sigmoid function's output needs to be doubled again (i.e. again similar to linear activation function) and then result needs to be passed to neurons of next layer by subtracting 1.

Implementation of boxed section of figure 1 can be given by below shown figure 2.

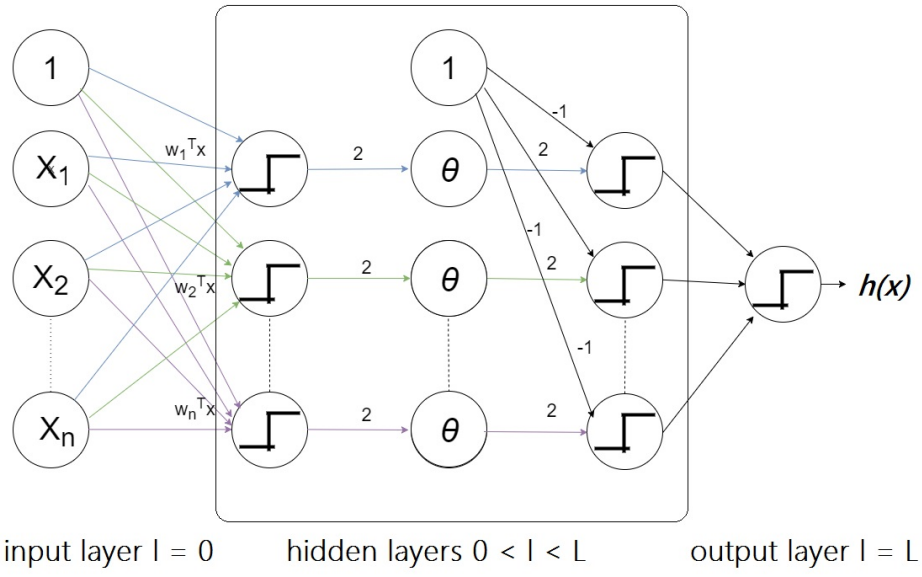

Figure 2: Same Network Using Sigmoid Function

For above shown network (figure 2), implemented functions can be described as below.

$$s_i^l = \text{In Coming Signal To Neuron } i \text{ In Layer } l$$

$$w_i^l = \text{In Coming Weight Vector For Neuron } i \text{ In Layer } l$$

$$X_i^l = \text{Input To Neuron } i \text{ In Layer } l$$

$$\implies s_i^l = w_i^{lT} * X_i^l$$

$$\implies s_i^{'l} = 2s_i^l \qquad \text{(Doubling Input Signal)}$$

$$\implies \theta(s_i^{'l}) = \sigma(s_i^{'l}) \qquad \text{(Applying Sigmoid Function)}$$

$$Where : \sigma(s_i^{'l}) = \frac{1}{1+e^{-s_i^{'l}}}$$

$$\implies s_i^{'(l+1)} = 2\sigma(s_i^{'l}) - 1 \qquad \text{(Generating Output For Next Layer)}$$

$$\implies X_i^{(l+1)} = s_i^{'(l+1)}$$

$$X_i^{l+1} = \text{Output Of Neuron } i \text{ In Layer } l$$

So hypothesis regarding similar network can look like as below.

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

$$\boxed{h(X) = 2\sigma(2W_L^T(.....(2\sigma(2W_1^T(2\sigma(2W_0^T X(0)) - 1)) - 1))) - 1}$$

By using above hypothesis implementation, it is observable that weight of input signal will be doubled and output of activation function will also be doubled to produce next layer's input vector.

2 Multi-layer Perceptrons

Given problem statement is clearly not linearly classified, but to classify the data according to shown in problem statement, a combination of multiple linear classifiers can be used.

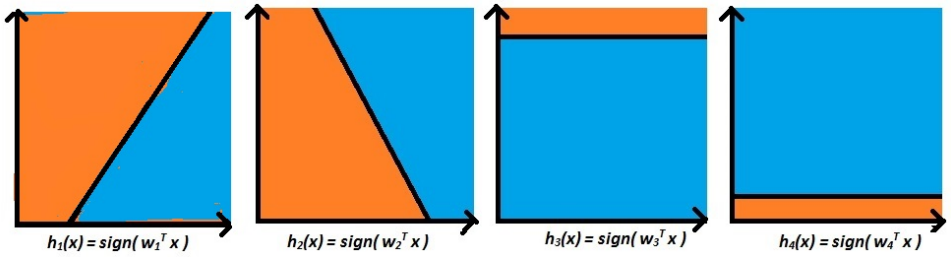

Figure 3: 4 Linear Classifiers

As shown in above figure 3, 4 different linear perceptron classifiers are shown. Having different weight vectors for each perceptron, function can be defined as below.

$$h_1(x) = sign(w_1^T x)$$

$$h_2(x) = sign(w_2^T x)$$

$$h_3(x) = sign(w_3^T x + b3)$$

$$h_4(x) = sign(w_4^T x + b4)$$

Here, only $h_3(x)$ & $h_4(x)$ will require bias terms as they are not passing through origin. As $h_1(x)$ & $h_2(x)$ should pass through origin, bias terms regarding both perceptron will be zero.

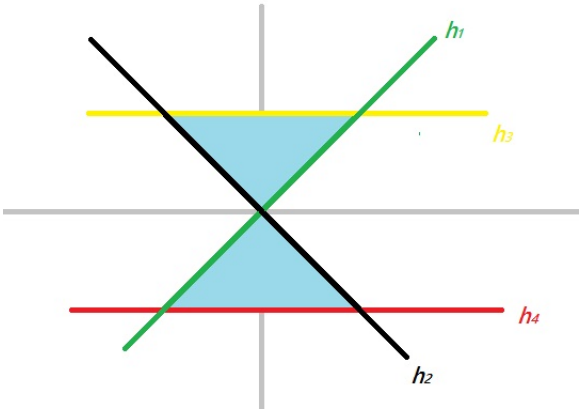

Figure 4: Classifying Using Combination Of Multiple Perceptron

Now, as shown in figure 4 using multiple linear classifiers data has been classified as needed. Equation regarding above classification may look like as below.

$$f = \bar{h_1} h_2 h_3 + h_1 \bar{h_2} h_4$$

A graphical representation of neural network can look like as shown below figure 5. Weights of each edge is taken according to function f.

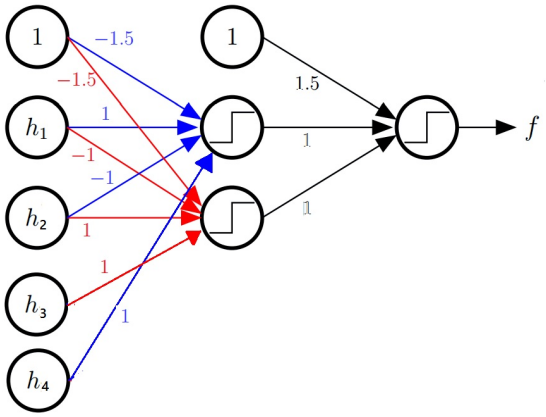

Figure 5: Neural Network

So in the nutshell, example neural network having two feature can be implemented as shown below figure 6 to classify problem statement.

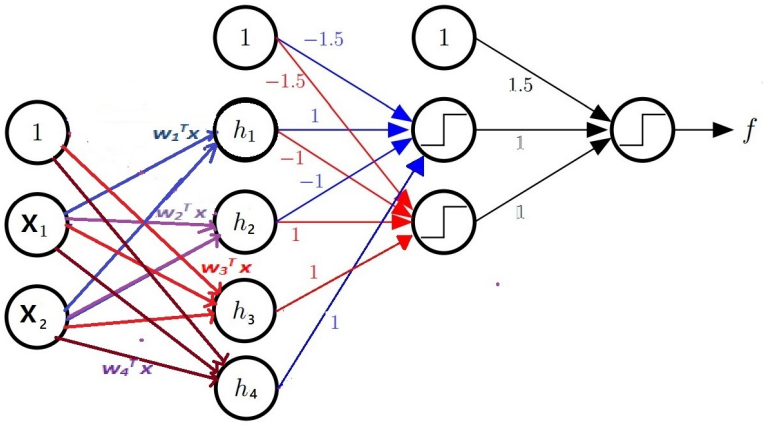

Figure 6: Neural Network Using Combination Of Multiple Linear Classifiers

3 Exploring Neural Networks For Digit Classification

Basically, for neural network, parameters - number of layers, number of hidden units, learning rate - are customized and must be chosen such that network gives higher accuracy without overfitting. Generally, It is impossible to determine the optimal number of hidden units without training different networks and estimating the generalization error(i.e. accuracy) of each. If fewer hidden units are taken, network will get high training error and high generalization error due to underfitting and low accuracy. If too many hidden units are taken, network may give low training error but still give high generalization error due to overfitting and high variance. Cross-validation can be used to determine optimal value of hidden layer units. Choosing proper units is required because the number of hidden units governs the expressive power of network.

The choice of learning rate depends upon behaviour of errors (misclassification). For the wide surface of error rate, higher learning rate can be chosen. But for narrow surface, lower learning rate seems to be more convincing. For given data problem, lower learning rate ($\eta = 0.2$) is taken.

Let's explore network for one and two hidden layer with dynamic hidden layer units.

3.1 Exploring Single Layer Network Using Logistic Sigmoid Activation Function

Here, number of hidden units are taken on logarithmic scale (1, 2, 4, ...). Nearly 15 observations were made and accuracy & error of network for each observation has been plot on graph (figure 7 & 8). Accuracy of network has been calculated by taking average number of correct classification and like-wise error of network is calculated by taking average number of misclassification. Implementation regarding this, has been shown in appendix section (A).

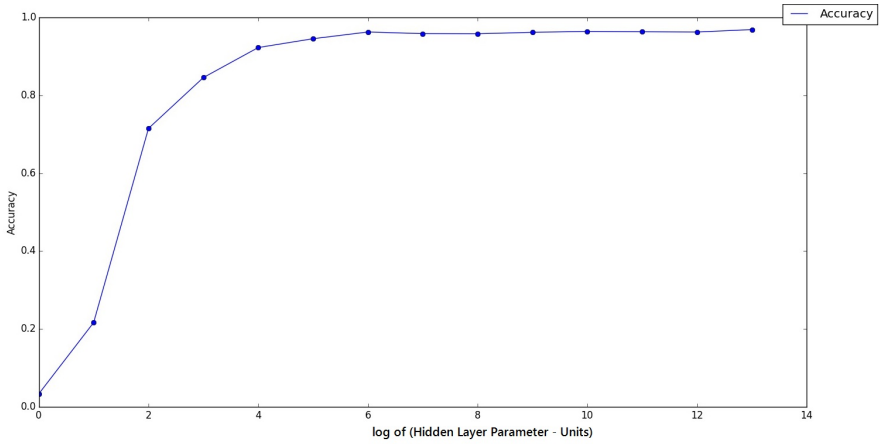

Figure 7: Accuracy V/S Number Of Hidden Layer Units (Number Of Layers : 1)

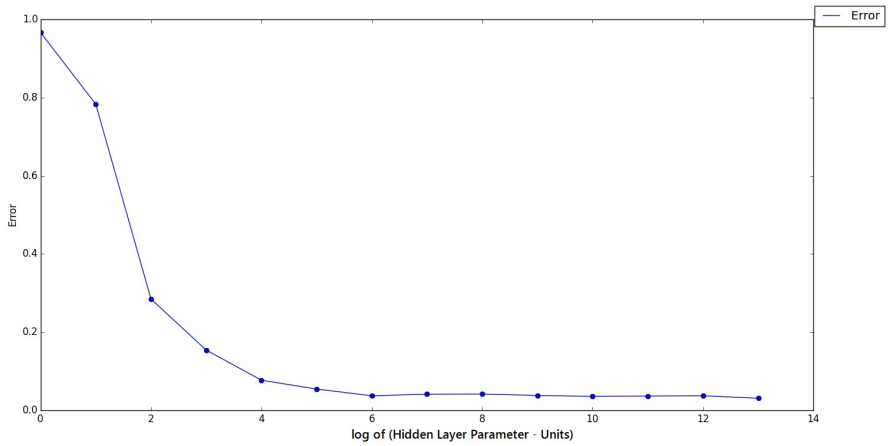

Figure 8: Error(Misclassification) V/S Number Of Hidden Layer Units (Number Of Layers : 1)

As mentioned earlier, it is observable from both graph that, for very low value of hidden units(i.e. 1, 2), network is giving high training error and high generalization error due to underfitting and low accuracy. After increasing number of units more than 2, network seems to be trying to get converged.

For number of units greater than 16, network gives more than 90% of accuracy. For higher number of hidden units, number of weights will also be higher. For any activation function's implementation, increment in the number of weights makes it really easy for standard back-propagation to find a good local optimum. So use of large networks can reduce both training error and generalization error and give high accuracy.

Now according to nature of neural network, for higher values of hidden layer units, network should give low training error but still can give high generalization error due to overfitting. As above drawn graphs depict, for increasing number of hidden units, network still gives high accuracy and low out-of-sample error. It can be observed that, even for higher number of hidden units, network does not end up doing overfitting. One possible explanation regarding the observation is, testing dataset can be very easy that it seems to be giving static but high accuracy with many variance of weight vectors. So the weight vectors, generated by using large number of hidden units, works well for both training and testing datasets. Observations made during experiment are listed in below shown table (1)

Hidden Layer Units	Accuracy	Error (Misclassification)
1	0.033357482089672388	0.96664251791032763
2	0.21650394885527136	0.78349605114472864
4	0.7151061187157135	0.2848938812842865
8	0.84565697292732211	0.15434302707267789
16	0.92275496201624796	0.077245037983752041
32	0.94548536887162093	0.054514631128379065
64	0.96274667164364813	0.037253328356351867
128	0.95856939383943351	0.041430606160566485
256	0.95826213102346325	0.041737868976536752
512	0.96187363022828476	0.038126369771715241
1024	0.96425018229878767	0.035749817701212327
2048	0.96331694271911661	0.036683057280883391
4096	0.96262455970530492	0.037375440294695084
8192	0.96901291341508722	0.030987086584912782
16384	0.95683568204844782	0.043164317951552178

Table 1: Accuracy Observed In Accordance With Changing Hidden Layer Parameter(Units)

Another observation of accuracy was also made by incrementing 5 number of hidden units at a time instead of changing it on logarithmic scale. Number of hidden units taken for trainings were 1, 5, 10, 15, 20... Graphical representation of observation is shown in below figure 9. Behaviour of network seems to remain same as mentioned earlier in this section.

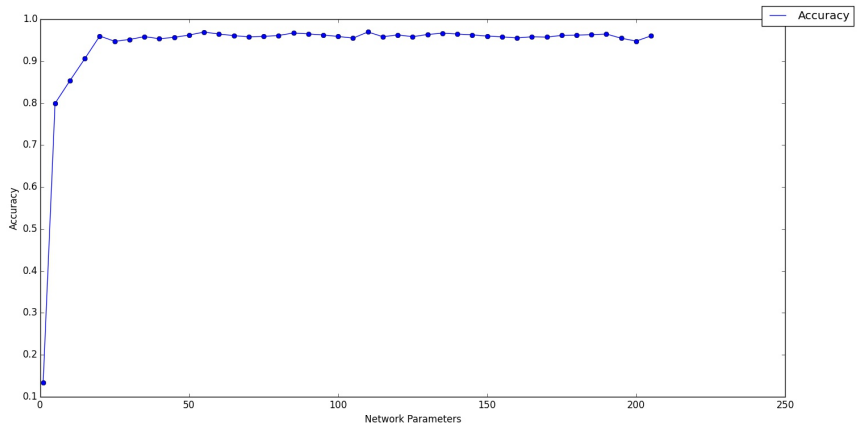

Figure 9: Accuracy V/S Number Of Hidden Layer Units (Number Of Layers : 1)

3.2 Exploring Double Layer Network Using Logistic Sigmoid Activation Function

Theoretically, a Neural Network with a single hidden layer can fit majority of the hypothesis functions. To explore network having two layers, number of hidden units are taken on logarithmic scale (1, 2, 4, 8...). Nearly 11 observations were made and as single layer network, accuracy & error of network for each observation has been plot on graph (figure 10 & 11) too. Accuracy & Error of network has been calculated the same way they were calculated for single layer neural network. Implementation regarding this, has been shown in appendix section A.

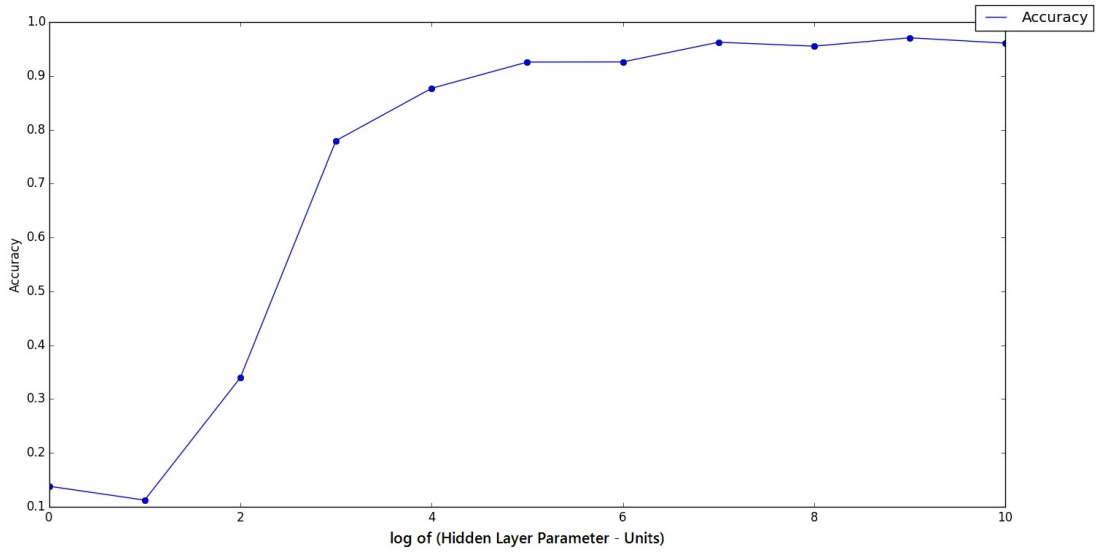

Figure 10: Accuracy V/S Number Of Hidden Layer Units (Number Of Layers : 2)

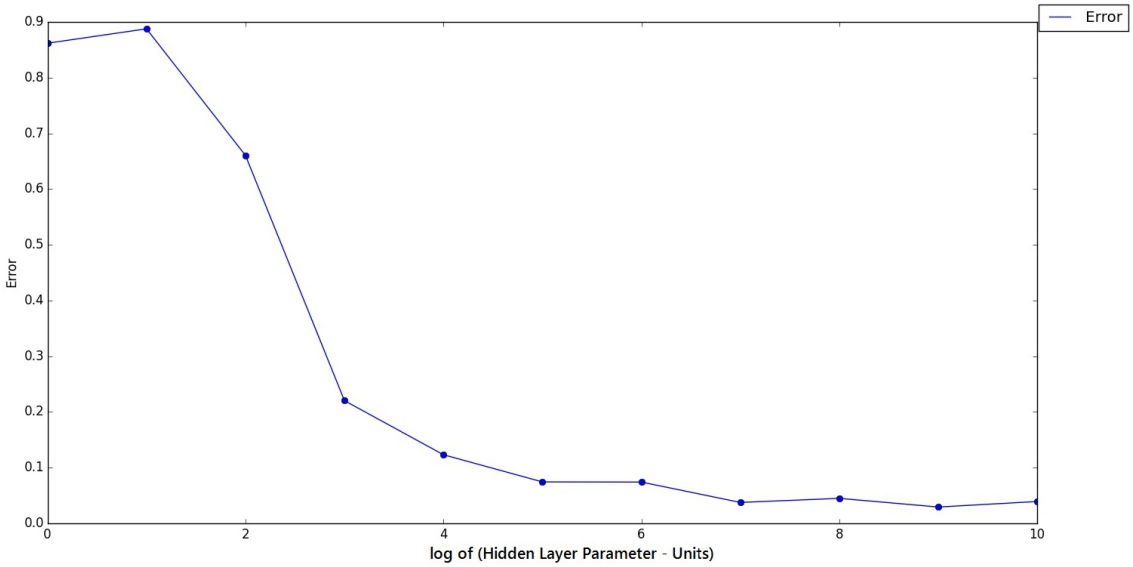

Figure 11: Error(Misclassification) V/S Number Of Hidden Layer Units (Number Of Layers : 2)

As observed from both above graphs, behaviour of network still remains same as it did with having single layer.

Interestingly, network leaves underfitting zone quiet later than it did when it used only single hidden layer. Furthermore for most of the cases, for same number of units, observed accuracy is also lower than single layer network accuracy . Some fluctuations in accuracy, at initial point, are also observed. Possible explanation regarding such observations can be, with the increase of the number of hidden layer, effectiveness of backpropagation decreases. Network may perform well on the training data set but it may fail to generalize testing dataset it has not seen yet and it ends up giving bizarre performance on testing dataset.

Observations made during experiment are listed in below table 2.

Hidden Units	Accuracy	Error (Misclassification)
1	0.13821705426356587	0.8617829457364341
2	0.11241466587753343	0.88758533412246654
4	0.33962706329771164	0.66037293670228836
8	0.77973520749793357	0.22026479250206643
16	0.87682398991481936	0.12317601008518064
32	0.92559352658364025	0.074406473416359753
64	0.92587913521514031	0.074120864784859686
128	0.9624274555969311	0.037572544403068897
256	0.95512687488442061	0.044873125115579393
512	0.97067156114925002	0.029328438850749983
1024	0.96075807442927841	0.039241925570721592

Table 2: Accuracy Observed In Accordance With Changing Hidden Layer Parameter(Units)

Above mentioned behaviour of double layer network is not general. It can also be possible that for different dataset, two layer network performs better than single layer network. But for digit dataset, single layer network gives better performance than double layer network.

3.3 Impact Of Weight Decay Regularization

As described in earlier sections, dataset is not overfitting using neural networks. If it did, use of adding decay to weight to get regularization may help to get more generalization.

The learning rate(η) is a parameter that determines weights vector till now.

$$w_i = w_i - \eta \frac{\partial E}{\partial w_i}$$

where: $\frac{\partial E}{\partial w_i}$ = Partial Differentiation Of Error With Respect To Weight

Now for effectively caging the number of free parameters in model for avoiding over-fitting, it is possible to regularize the cost function. By introducing new regularization parameter λ which determines the trade off the original cost E with the large weights penalization, augmented error can look like as below,

$$E_{augment}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

By applying gradient descent to this new error function, w can be rewritten as follows:[1]

$$w_i = w_i - \eta \frac{\partial E}{\partial w_i} - \eta \lambda w_i.$$

Decay in the weight is proportion to its size. Implementation regarding weight decay is shown in appendix B.

3.4 Use Of Linear Activation Function

Best schema to solve regression problem is to use hidden layer of sigmoid function followed by a linear layer. Implementation regarding above schema is shown in appendix C. Only few minor changes and addition of two linear method was required to produce above schema. In appendix C only change in the code is shown.

Use of any activation function(except linear activation function) on input vector will transform input range to some other range. For example, as mentioned in earlier sections, *logistic sigmoid* will transform input in range of (0, 1) & *tanh* will transform input in range of (-1, 1). Output variable in regression problems needs to be predicted continuously and without having any bound. If we choose logistic sigmoid as activation function in the output layer too, then output will be caged in the range of 0 to 1 only. Because of such behaviour, classification problems can be solved but not

regression problems. So for hidden layer's computation any activation function can be applied but for output layer, linear activation function should be used. Furthermore, linear activation will not cage the data to specific range and because of that, network can produce outcome to desire range too.

Use of linear neurons in a hidden layer do not add any positive effect to the neural network as all they only do is a linear projection. Linear neurons in a hidden layer may actually have a negative impact on the networks performance because they add additional weights that have to be trained. So instead of adding linear neurons to all layers, schema of using sigmoid function to all hidden layers and linear function in last layer only does make positive impact in solving regression problems. [2]

References

[1] Weight Decay Reference
http://stats.stackexchange.com/questions/29130/difference-between-neural-net-weight-decay-and-learning-rat

[2] Linear Function Reference
http://www.researchgate.net/post/In_a_feed-forward_neural_network_NN_which_type_of_neurons_linear_or_log-sigmo

Appendices

A Implementation Of One/Two Layer Neural Network

```
def test_digits() :
   digits = load_digits()

   X = digits.data
   y = digits.target
   X /= X.max()
   network_values = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
   print 'Network Paras', network_values

   X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
   print 'X Training Size: ', X_train.shape
   print 'Y Training Size: ', y_train.shape
   print 'X Testing Size: ', X_test.shape
   print 'Y Testing Size: ', y_test.shape

   labels_train = LabelBinarizer().fit_transform(y_train)
   labels_test = LabelBinarizer().fit_transform(y_test)

   accuracyMean = []
   errorMAD = []
   for index in (network_values) :
      print 'Network Parameter ', index
      nn = NeuralNetwork([64, index, 10],'logistic')
      # Use Below nn Instead Of Upper One For Two Layer Neural Network
      # nn = NeuralNetwork([64, index, index, 10],'logistic')
      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))

      print 'Confusion Matrix'
      print confusion_matrix(y_test,predictions)
      print 'Report'
      print classification_report(y_test,predictions)
      print 'Average'
      print np.mean(precision_score(y_test,predictions, average=None))
      print 'Accuracy'
      print accuracy_score(y_test,predictions)
      accuracyMean.append(np.mean(precision_score(y_test,predictions, average=None)))
      errors = (y_test - predictions)**2
      print 'Calculated Error', 1 - np.mean(precision_score(y_test,predictions, average=None))
      errorMAD.append(1 - np.mean(precision_score(y_test,predictions, average=None)))

   print 'Network', network_values
   print 'Accuracy', accuracyMean
   print 'Error MAD', errorMAD
   plot_graph(np.log2(network_values), accuracyMean, 1)
   plot_graph(np.log2(network_values), errorMAD, 2)
```

B Implementation Of Weight Decay

```
def fit(self, X, y, learning_rate=0.2, epochs=50):
   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)
   lambdaValue = 0.001

   for k in range(epochs):
      if k%10==0 : print "** ", k, " epochs **"
      I = np.random.permutation(X.shape[0])
      for i in I :
         a = self.forward(X[i])
         deltas = self.backward(y[i], a)
         # 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])
            self.weights[i] += learning_rate * layer.T.dot(delta)

            self.weights[i] -= lambdaValue * learning_rate * self.weights[i]
```

C Implementation Of Linear Activation Function In Last Layer

```
#Addition Of Method Linear And Linear Deriv
def linear(x) :
   return x

def linear_deriv(x) :
   return np.ones(x.shape)
# Change In Methods — forward,, backward, predict
def forward(self, x) :
   a = [x]
# Iterating Till Second Last Layer
   for i in range(self.num_layers - 1):
# print 'In Forward, Layer: ', i
      a.append(self.activation(np.dot(a[i], self.weights[i])))

# Adding Linear To Last Layer
   a.append(linear(np.dot(a[self.num_layers - 1], self.weights[self.num_layers - 1])))

# print 'Activation: ', a
   return a

def backward(self, y, a) :
# For Last Layer
   deltas = [(y - a[-1]) * linear_deriv(a[-1])]

   for l in range(len(a) - 2, 0, -1):
      deltas.append(deltas[-1].dot(self.weights[l].T)*self.activation_deriv(a[l]))

   deltas.reverse()
   return deltas

 def predict(self, x):
      x = np.asarray(x)
      temp = np.ones(x.shape[0]+1)
      temp[0:-1] = x
      a = temp
# print 'Self Weight', len(self.weights)
      #Iterate Till Second To Last Layer
      for l in range(0, len(self.weights) - 1):
         a = self.activation(np.dot(a, self.weights[l]))

      a = linear(np.dot(a, self.weights[len(self.weights) - 1]))
      return a
```