NeuralNetworkTutorial

February 14, 2016

1 Code Description, Plots, Analysis and Conclusion

GA WU
Oregon State University

1.1 Activation Functions

The ReLU activation function can output both ReLU and its derivative, We compute ReLU first and if it is used in backpropagation, we then compute its derivative by simply make all none zero output of ReLU be one.

```
In [ ]: def ReLU(nets,derivative=False):
            output= np.maximum(nets,0)
            if derivative:
                derivate = output
                derivate[derivate != 0] = 1
                return derivate
            else:
                return output
```

The Sigmoid activation function can output both Sigmoid and its derivative, We compute Sigmoid first and if it is used in backpropagation, we then compute its derivative by simply product it $\sigma(1-\sigma)$. Note that we made hard boundary if the network output is too big that can cause Sigmoid function give 1 or 0 as result.

```
In [ ]: def Sigmoid(nets, derivative=False):
            output = np.clip( nets, -500, 500 )
            output= 1.0/(1+np.exp(-output))
            if derivative:
                derivate=np.multiply(output,1-output)
                return derivate
            else:
                return output
```

1.2 Normalization Function

The most critical preprocesing procedure in Neural network is to normalize data into standard normal distribution

We normalize training data without using mean and std, but we will store those two variable after doing normalization. When normalizing testing data, to guarentee it is normalized in same standard with training data, we need mean and std as parameters.

```
In [ ]: import numpy as np
        def normalize(features, mean = [], std = []):
```

```
        if mean == []:
            mean = np.mean(features, axis = 0)
            std = np.std(features, axis = 0)
        print std
        print std[:,None]
        new_feature = (features.T - mean[:,None]).T
        new_feature = (new_feature.T / std[:,None]).T
        new_feature[np.isnan(new_feature)]=0
        print new_feature
        return new_feature, mean, std
```

1.3 Loss Function

In this code, we provide two loss functions cross-entroy loss and mean squared error. Be aware, when using cross-entroy loss, you must use Sigmoid function as final output layer nonlinear function. (the code automatically include derivative of cross-entroy and final sigmoid together.)

```
In [ ]: import numpy as np
        import scipy as sp

        def CrossEntropyLoss(y,z,deriv=False):
            epsilon = 1e-15
            z=sp.maximum(z,epsilon)
            z=sp.minimum(z,1-epsilon)

            if deriv:
                return y-z #this include derivative of sigmoid function
            else:
                return -1.0/len(y)*np.sum((np.multiply(y,np.log(z))+\
                                           np.multiply((1-y),np.log(1-z))),axis=0)
```

```
In [ ]: def MeanSquaredError(y,z,deriv=False):
            if deriv:
                return (y-z)*y*(1-y) #this include derivative of sigmoid function
            else:
                return -1.0/len(y)*np.sum(np.multiply(y-z,y-z),axis=0)
```

1.4 Network Settings

We want to create a format that allows new instance of neural network to do initial setting using such format. The following one feasible format that is used in this code.

```
In [ ]: initial_setting = {
            "inputs_N"          : 3072, #Assign input dimensions
            "layers"            : [ (100, network.ReLU),(1, network.Sigmoid)], #Assign Layer details
            "weights_L"         : -0.1, #Random weight initialization boundary
            "weights_H"         : 0.1, #Random weight initialization boundary
            "save"              : False, #haven't implement this one yet
            "loss"              : network.CrossEntropyLoss, #Assign Loss Function to use
            "learning_C"        : [], #Track Learning Error Curve
            "testing_C"         : [] #Track Testing Error Curve
                        }
```

But initial setting may not enough for making modification of the network structure, we want it more flexible. Therefore, we also need another function to do modifcation on initial setting.

```
In [ ]: def replace_value_in_initial_setting(key_to_find, definition):
            for key in initial_setting.keys():
                if key == key_to_find:
                    initial_setting[key] = definition
```

1.5 Network Framework

Each neural network that generated by this code is one instance of network framework class. The class has several private functions that allows it initialization, updating weights, backpropagation, training and testing on data.

```
In [1]: class NetworkFrame:

            # Compute number of weights that needed by this network according to settings
            def __init__(self, settings):
                self.__dict__.update(settings)
                self.weights_N=(self.inputs_N+1)*self.layers[0][0]+\
                    sum( (self.layers[index][0]+1)*layer[0] for\
                            index,layer in enumerate(self.layers[1:]))
                self.weights_Matrics = self.ReshapeWeights(self.RandomInitialWeights())

            # Random initialize weights
            def RandomInitialWeights(self):
                return np.random.uniform(self.weights_L,self.weights_H,self.weights_N)

            # Reshape weights into matrix format for later computation convenience
            def ReshapeWeights(self, weights_flat):
                head = 0;
                tail = 0;
                weights_matrics = []

                previous_node_N = self.inputs_N + 1

                for current_node_N, node_T in self.layers:
                    head = tail
                    tail += previous_node_N * current_node_N
                    weights_matrics.append(weights_flat[head:tail].\
                                           reshape(previous_node_N,current_node_N))
                    previous_node_N = current_node_N + 1
                return weights_matrics

            # And other member functions. we will describe those later
```

The forward function in neural network is very simple. In two layers network. we can use single equation to represent it: $\mathbf{y} = g_2(\mathbf{w_2} g_1(\mathbf{w_1 x} + b_1) + b_2)$. In the specific setting, g_2 is sigmoid function and g_1 is ReLU function.

```
In [ ]:     def FeedForward(self, inputs, backpropagation = False):

                output = inputs

                if backpropagation:
                    outputs = [ output ]
                    derivates = []
```

```
        for index, weight_Matrix in enumerate(self.weights_Matrics):
            signal = weight_Matrix[0:1,:]+np.dot(output, weight_Matrix[1:,:])
            output = self.layers[index][1](signal)

            if backpropagation:
                outputs.append(output)
                derivates.append(self.layers[index][1](signal,True).T)

        # When doing training, we need all mediate outputs to update weights
        # and as well as derivates computed by activation functions
        if backpropagation:
            return outputs, derivates
        # When testing, we only need final output of the network
        else:
            return output
```

Backpropagation is also computationally simple. When updating weights, we actually doing multiplication on several derivatives: $\Delta w = \frac{\partial E}{\partial g}\frac{\partial g}{\partial net}\frac{\partial net}{\partial w}$, which is exactly $\Delta w_{ij} = \delta_j a_i$, where a_i representing previous layer outputs and δ is current layer jacobian value. In this code, we also consider Δw of previous step backpropagation and using momentum to make the learning smooth.

In [ ]:
```
    def BackPropagation(self, outputs, derivates, training_targets,\
                        prev_delta_w, momentum, learning_rate):
        curr_delta_w = []
        output = outputs[-1]
        delta = self.loss(training_targets, output, True).T

        for i in range( len(self.layers) )[::-1]:
            delta_w = learning_rate*np.dot(delta, utils.add_ones(outputs[i])).T
            if i != 0:
                delta = np.dot(self.weights_Matrics[i][1:,:], delta)*derivates[i-1]
            self.weights_Matrics[i]+=momentum*delta_w+(1-momentum)*prev_delta_w[i]
            curr_delta_w.append(delta_w)
        return curr_delta_w[::-1]
```

The training function is actually an warp of backpropagation and forward prediction functions. We also do mini batch training in this function, which cut out subsample of data with minibatch size that assigned through argument.

In [ ]:
```
    def Train(self, training_features, training_targets,\
              learning_rate = 0.05, minibatch_size = 10,\
              max_epoch = 1000, error_threshold = 1e-3,momentum = 0.7):
        epoch = 0 #epoch initialization
        error = 1 #Error initialization
        features = np.copy(training_features)
        targets = np.copy(training_targets)
        prev_delta_w = [0.001*x for x in self.weights_Matrics]

        #Two stop conditions Max epoch or Error threshold
        while error < error_threshold or epoch < max_epoch:
            epoch += 1
            perm = range(len(features))
            random.shuffle(perm)
            features = features[perm]
            targets = targets[perm]
```

```
                #Mini Batch Size is assigned when doing training but not NN initial setting!
                for i in range(len(features)/minibatch_size):
                    mini_features = features[i*minibatch_size:(i+1)*minibatch_size]
                    mini_targets = targets[i*minibatch_size:(i+1)*minibatch_size]
                    outputs, derivates = self.FeedForward(mini_features, True)
                    error = self.loss(mini_targets, outputs[-1], deriv=False)
                    prev_delta_w = self.BackPropagation(outputs, derivates,mini_targets,\
                                                        prev_delta_w, momentum, learning_rate)
                train_output = self.Test(features)
                train_error = self.Error(targets, train_output)
                print "* Epoch %d : Error %f." %(epoch,train_error)

            if epoch%max_epoch==0:
            # Show the current training status
                print "* Default maximum epoch reached!"
            print "* Trained for %d epochs." % epoch
```

Test function is an warp of forward function. This function will modify the probabilistic representation of output of neural network into discrete 2-classes 1 and 0.

```
In [ ]:     def Test(self, test_features):
                prediction = self.FeedForward(test_features, False)
                prediction[prediction>=0.5] = 1
                prediction[prediction != 1] = 0
                return prediction
```

We compute average error through Error function : $\frac{1}{N}\sum_{i=0}^{N}(|y_i - z_i|)$

```
In [ ]:     def Error(self, y, z):
                return np.sum(np.abs(y-z))/len(y)
```

1.6 Other Support Functions

Loading Data from cPickle file

```
In [ ]: import os

        def unpickle(file_path):
            try:
                import cPickle as pickle
            except:
                import pickle

            fo = open(file_path, 'rb')
            data = pickle.load(fo)
            fo.close()
            return data

        def load():
            datapath="data/cifar_2class_py2.p"
            script_dir = os.path.dirname(__file__)
            abs_data_path = os.path.join(script_dir, '..', '..',datapath)
            data=unpickle(abs_data_path)
            return data
```

Adding bias to each Layer Outputs

```
In [ ]: def add_ones(A):
            return np.hstack(( np.ones((A.shape[0],1)), A ))
```

1.7 Main Function

```
In [ ]: #Load Data
        dataset = data.load()
        #Create Network with initial setting
        NN = network.NetworkFrame(initial_setting)
        features_normalized,mean,std = data.normalize(features) #Normalize Training data
        test_normalized,_,_ = data.normalize(test_features,mean,std) #Normalize Testing data
        #Learning rate 10e-5, Mini Batch Size: 100, Maximum Epoch: 200, Objective Boundary: 0.001
        NN.Train(features_normalized, targets, 10e-5, 100, 200, 0.001)
        NN.Test(test_normalized)
```

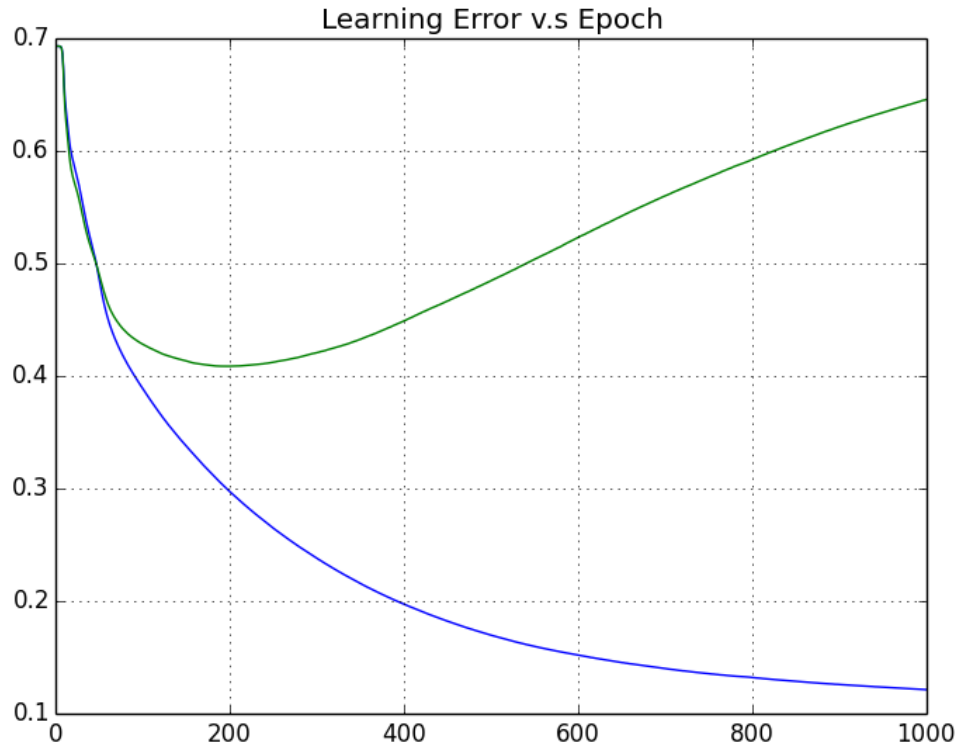
1.8 Plots

1.9 Cross-Entropy Objective vs Epoch Plot

Blue Line is training data curve, green line is testing data curve.(please ignore the title..)

```
In [3]: from IPython.display import Image
        Image(filename='objective.png')
```

Out[3]:

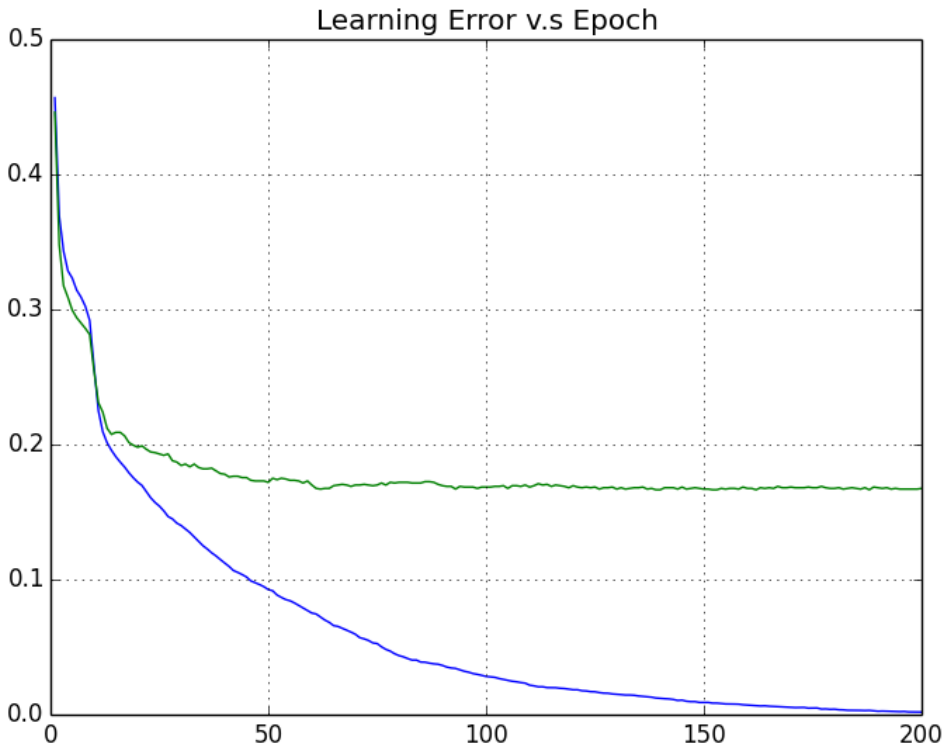


1.10 Error vs Epoch

Blue Line is training data curve, green line is testing data curve.(please ignore the title..)

```
In [4]: from IPython.display import Image
        Image(filename='error.png')
```

Out[4]:

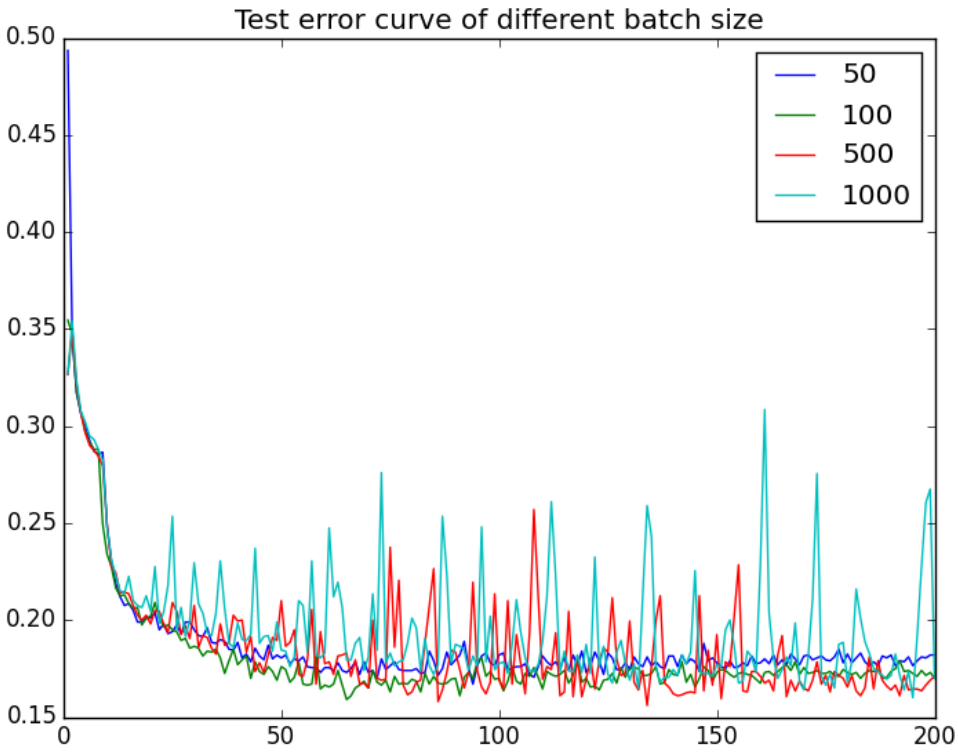


1.11 Test Error vs Mini Batch Size

Though size of mini batch doesn't effect the final result too much, the larger mini batch will cause the curve shaking.

```
In [1]: from IPython.display import Image
        Image(filename='batchsize.png')
```

Out[1]:

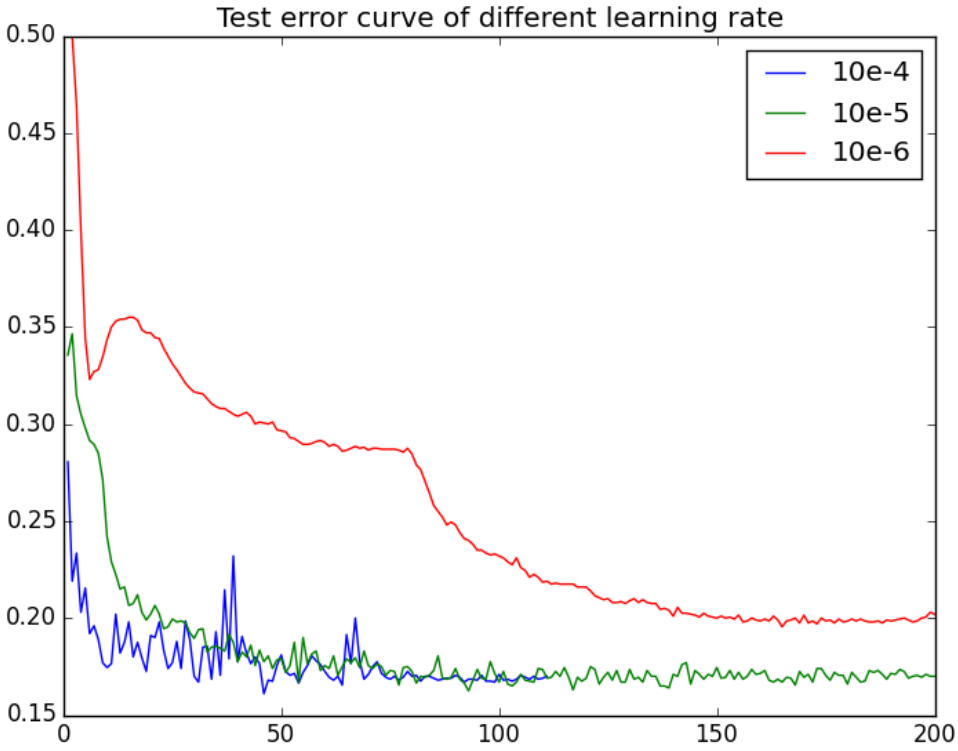


1.12 Test Error vs Learning Rate

Larger learning rate can cause the curve shake, whereas smaller learning rate cause error decrease slow. The following plot shows such property. We then selecte 10e-5 as our learning rate.

```
In [2]: from IPython.display import Image
        Image(filename='learningrate.png')
```

Out[2]:

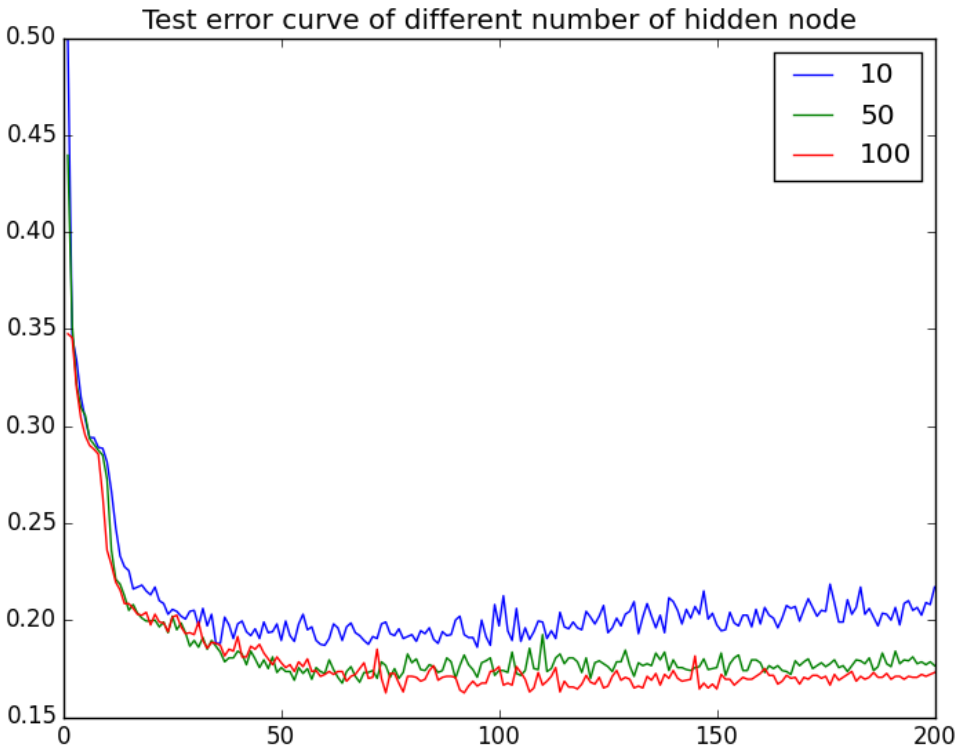


1.13 Test Error vs Number of Hidden Nodes

If number of the hidden nodes is not enough, the complexity of model is limited. The network cannot optimize the objective due to the model complexity limitation. However, too large number of hidden nodes doesn't help to further improve the performance because it is already enough to represent target function. In the following plot, we show that 100 hidden nodes is enough for the task.

```
In [3]: from IPython.display import Image
        Image(filename='numberofhidden.png')
```

```
Out[3]:
```



1.14 Discussion

The best testing error this single neural network can achieve is 0.17 while the training is not overfitting yet. We can see the the plot of Cross entropy loss, testing objective reach it optimal at 200 epoches and then getting worse when training continues.

The best learning rate is 0.0001 in this task. Larger learning rate can cause overshotting, whileas smaller rate will cause training longer.

Mini batch size will seriously impact learning time, because of doing many matrix computation separately. The relatively reliable mini batch size we used here is 100. Since the learning rate can also affected by the mini batch size, we specifically select 100, which is best when learning rate is 0.0001.

We believe 100 hidden nodes is the optimal number, which is enough for the task and also computationally feasible than larger number