Problem4

May 14, 2018

1. This notebook can build DNN with user-defined number of layers and neurons for MNIST classification.
2. The calculation process with Matrix Notation is included in Layer Class.

1 Load Data

```
In [1]: import copy
        import random
        import numpy as np
        import matplotlib.pyplot as plt
        from mnist import MNIST
        from math import exp,log,tanh,sqrt
        mndata = MNIST('./mnist/')
        mndata.gz = True
        images, labels = mndata.load_training()
        test_imgs, test_labels = mndata.load_testing()
```

2 Data Preprocess

Shuffle Data

```
In [2]: all_data=np.concatenate((np.array(images),np.array(labels).reshape(len(labels),1)),axis
        np.random.shuffle(all_data)
        images=all_data[:,:-1]
        labels=all_data[:,-1]

In [3]: images=np.array(images)
        transfered_images=np.zeros((len(images),784))
        input_images_feature=np.zeros((len(images),785))
        transfered_test_images=np.zeros((len(test_imgs),784))
        input_test_images_feature=np.zeros((len(test_imgs),785))

In [4]: # Put all values into [-1,1]
        for i in range(len(images)):
            transfered_images[i]=np.array(images[i])
            transfered_images[i]=transfered_images[i]/127.5 - 1
            input_images_feature[i]=np.insert(transfered_images[i],0,1)
```

```
for i in range(len(test_imgs)):
    transfered_test_images[i]=np.array(test_imgs[i])
    transfered_test_images[i]=transfered_test_images[i]/127.5 - 1
    input_test_images_feature[i]=np.insert(transfered_test_images[i],0,1)
```

This is equal to randomly select since the data has been shuffled before

```
In [5]: train_features=input_images_feature[:50000]
        train_labels=labels[:50000]
        valid_features=input_images_feature[50000:60000]
        valid_labels=labels[50000:60000]
        test_features=input_test_images_feature
        test_labels=test_labels
```

3 Minibatch

```
In [6]: BATCH = 256

        batch_train_features=[]
        batch_train_labels=[]
        for i in range(int(len(train_features)/BATCH)):
            batch_train_features.append(train_features[i*BATCH:i*BATCH+BATCH])
            batch_train_labels.append(train_labels[i*BATCH:i*BATCH+BATCH])
        batch_train_features.append(train_features[i*BATCH+BATCH:])
        batch_train_labels.append(train_labels[i*BATCH+BATCH:])
```

4 Layer Class

```
In [7]: class Layer(object):

            def __init__(self,num_next_layer_neuron):
                self.output_num=num_next_layer_neuron

            def configure(self,input_shape,reg_lam):
                self.lam = reg_lam
                self.w_shape=(input_shape[1],self.output_num)
                self.w=np.random.normal(0,1/sqrt(input_shape[0]),self.w_shape)
                self.delta_w=np.zeros(self.w_shape)

            def hidden_forward_prop(self,inputs,activate_index):
                self.x=copy.deepcopy(inputs)
                self.a=np.dot(self.x,self.w)
                self.activate_type=activate_index
                self.y=np.array(self.activate_func(self.a,activate_index))
                self.b=np.ones(len(inputs))
                self.output=np.c_[self.b,self.y]
                self.gradient=self.gradient_calc(self.output,activate_index)
```

```
        return self.output

    def hidden_back_prop(self,layer_index,next_layer_w,next_layer_delta,rate,alpha):
        if layer_index==1:
            self.delta=self.gradient*np.dot(next_layer_delta,next_layer_w.T)
        else:
            self.delta=self.gradient*np.dot(next_layer_delta[:,1:],next_layer_w.T)

        self.oldweight=copy.deepcopy(self.w)
        if self.delta_w.shape[1]>layer_neuron_num_list[len(layer_list)-1-layer_index]:
            self.old_delta_weight=copy.deepcopy(self.delta_w[:,1:])
        else:
            self.old_delta_weight=copy.deepcopy(self.delta_w)

        self.delta_w=rate*(np.dot(self.x.T,self.delta)[:,1:]/len(self.x))
        self.old_weight=copy.deepcopy(self.w)
        self.w+=alpha*self.old_delta_weight+self.delta_w

    def output_forward_prop(self,inputs,activate_index,label):
        self.x=copy.deepcopy(inputs)
        self.a=np.dot(self.x,self.w)
        self.label=label
        self.vector_label=np.zeros((len(inputs),self.output_num))
        for i in range(len(inputs)):
            self.vector_label[i][int(label[i])]=1
        self.activate_type=activate_index
        self.y=np.exp(self.a)/np.repeat(np.sum(np.exp(self.a),axis=1).reshape(self.a.sl
        return self.y

    def output_back_prop(self,rate,output_y,alpha):
        self.delta=self.vector_label-output_y
        self.old_weight=copy.deepcopy(self.w)
        self.old_delta_weight=copy.deepcopy(self.delta_w)
        self.delta_w=rate*(np.dot(self.x.T,self.delta)/len(self.x) - 2*self.lam*self.w
        self.w+=alpha*self.old_delta_weight+self.delta_w

    def predict(self):
        self.predicts=[0]*len(self.x)
        self.predicts=self.y.argsort()[:,-1]

    def accuracy(self):
        total_num=len(self.x)
        correct_num=sum([1 if self.predicts[i]==self.label[i] else 0 for i in range(tot
        return correct_num/total_num

    def activate_func(self,a,index):
        if index==0:
            return 1/(1+np.exp(-a))
```

```
        if index==1:
            return 1.7159*np.tanh(2*a/3)
        if index==2:
            zeros=np.zeros(a.shape)
            return np.maximum(zeros,a)

    def gradient_calc(self,output,index):
        if index==0:
            return np.multiply((1-output),output)
        if index==1:
            return 1.7159*(2/3)*(1-(np.tanh(output))**2)
        if index==2:
            zeros=np.zeros(output.shape)
            return np.greater(output,0).astype(int)

    def softmax_entropy(self):
        entropy=0
        entropy-=sum(np.log(np.sum(self.y*self.vector_label,axis=1)))/self.y.shape[1]
        return entropy/(len(self.x)) + np.sum(np.square(self.w)) * self.lam
```

5 Initialize Training

In [8]:
```
# Set Initial Parameter for neural network

################################################
# Set number of neurons for every layer       ##
# This also decide how the number of layers   ##
# For MNIST the last number must be 10.       ##
layer_neuron_num_list=[128,64,10]             ##
################################################

######################################
# Set update rate                   ##
output_layer_update_rate=0.00001    ##
hidden_layer_update_rate=0.01       ##
momentum_alpha=0.9                  ##
reg_lambda=0.1                      ##
######################################

###########################################################################
# Set activate function type: 0 for sigmoid, 1 for tanh and 2 for ReLU   ##
activate_type_index = 2                                                  ##
# Set nunber of data per mini-batch                                      ##
num_per_batch = BATCH                                                    ##
num_batch_per_epoch=int(len(train_features)/num_per_batch)               ##
###########################################################################
```

```
# Initial list for the layers and their output
layer_list=[]
valid_layer_list=[]
test_layer_list=[]
layer_output_list=[]
valid_layer_output_list=[]
test_layer_output_list=[]

# Save data to list
train_entropy_data=[]
train_accuracy_data=[]
valid_entropy_data=[]
valid_accuracy_data=[]
test_entropy_data=[]
test_accuracy_data=[]

# Initialize each layer
for i in range(len(layer_neuron_num_list)):
    layer_list.append(Layer(layer_neuron_num_list[i]))
    valid_layer_list.append(Layer(layer_neuron_num_list[i]))
    test_layer_list.append(Layer(layer_neuron_num_list[i]))
    if i==0:
        layer_list[i].configure((len(batch_train_features[0]),785),reg_lambda)
        valid_layer_list[i].configure((len(valid_features),785),reg_lambda)
        test_layer_list[i].configure((len(test_features),785),reg_lambda)
    else:
        layer_list[i].configure((len(batch_train_features[0]),layer_neuron_num_list[i-
        valid_layer_list[i].configure((len(batch_train_features[0]),layer_neuron_num_li
        test_layer_list[i].configure((len(batch_train_features[0]),layer_neuron_num_lis
```

6 Start Training

```
In [9]: # Start training
        saved_valid_entropy=[0,0,0]
        tmp_train_entropy=[]
        tmp_train_accuracy=[]

        # Initial Forward Propagation
        for i in range(len(layer_list)):
            valid_layer_list[i].w=copy.deepcopy(layer_list[i].w)
            test_layer_list[i].w=copy.deepcopy(layer_list[i].w)
            if i==0:
                layer_output_list.append(layer_list[i].hidden_forward_prop(batch_train_features
                valid_layer_output_list.append(valid_layer_list[i].hidden_forward_prop(valid_fe
                test_layer_output_list.append(test_layer_list[i].hidden_forward_prop(test_featu
            elif i!=len(layer_list)-1:
                layer_output_list.append(layer_list[i].hidden_forward_prop(layer_output_list[i-
```

```
        valid_layer_output_list.append(valid_layer_list[i].hidden_forward_prop(valid_la
        test_layer_output_list.append(test_layer_list[i].hidden_forward_prop(test_laye
    elif i==len(layer_list)-1:
        layer_output_list.append(layer_list[i].output_forward_prop(layer_output_list[i
        valid_layer_output_list.append(valid_layer_list[i].output_forward_prop(valid_la
        test_layer_output_list.append(test_layer_list[i].output_forward_prop(test_laye

# Start Loop
count_epoch=0
print('iter\ttrain_entropy\t\tvalid_entropy\t\ttest_entropy\t\ttrain_acc\tvalid_acc\tte

for num in range(100000000):

    # Backward Propagation
    for i in range(len(layer_list)):
        if i==0:
            layer_list[len(layer_list)-1].output_back_prop(output_layer_update_rate,lay
        else:
            layer_list[len(layer_list)-i-1].hidden_back_prop(i,layer_list[len(layer_lis

    # Forward Propagation
    for i in range(len(layer_list)):
        valid_layer_list[i].w=copy.deepcopy(layer_list[i].w)
        test_layer_list[i].w=copy.deepcopy(layer_list[i].w)
        if i==0:
            layer_output_list[i]=layer_list[i].hidden_forward_prop(batch_train_features
        elif i!=len(layer_list)-1:
            layer_output_list[i]=layer_list[i].hidden_forward_prop(layer_output_list[i
        elif i==len(layer_list)-1:
            layer_output_list[i]=layer_list[i].output_forward_prop(layer_output_list[i
    layer_list[-1].predict()
    tmp_train_accuracy.append(layer_list[-1].accuracy())
    tmp_train_entropy.append(layer_list[-1].softmax_entropy())

    # One epoch finished
    if num%num_batch_per_epoch==0:
        count_epoch+=1
        train_accuracy=sum(tmp_train_accuracy)/len(tmp_train_accuracy)
        train_entropy=sum(tmp_train_entropy)/len(tmp_train_entropy)
        tmp_train_entropy=[]
        tmp_train_accuracy=[]

        # Forward Propagation for valid and test
        for i in range(len(layer_list)):
            valid_layer_list[i].w=copy.deepcopy(layer_list[i].w)
            test_layer_list[i].w=copy.deepcopy(layer_list[i].w)
            if i==0:
                valid_layer_output_list[i]=valid_layer_list[i].hidden_forward_prop(vali
```

```
            test_layer_output_list[i]=test_layer_list[i].hidden_forward_prop(test_i
        elif i!=len(layer_list)-1:
            valid_layer_output_list[i]=valid_layer_list[i].hidden_forward_prop(vali
            test_layer_output_list[i]=test_layer_list[i].hidden_forward_prop(test_l
        elif i==len(layer_list)-1:
            valid_layer_output_list[i]=valid_layer_list[i].output_forward_prop(vali
            test_layer_output_list[i]=test_layer_list[i].output_forward_prop(test_l

    valid_layer_list[len(valid_layer_list)-1].predict()
    valid_accuracy=valid_layer_list[len(valid_layer_list)-1].accuracy()
    valid_entropy=valid_layer_list[len(valid_layer_list)-1].softmax_entropy()
    test_layer_list[len(test_layer_list)-1].predict()
    test_accuracy=test_layer_list[len(test_layer_list)-1].accuracy()
    test_entropy=test_layer_list[len(test_layer_list)-1].softmax_entropy()

    # Save data to list for plotting
    train_entropy_data.append(train_entropy)
    train_accuracy_data.append(train_accuracy)
    valid_entropy_data.append(valid_entropy)
    valid_accuracy_data.append(valid_accuracy)
    test_entropy_data.append(test_entropy)
    test_accuracy_data.append(test_accuracy)
    saved_valid_entropy[num%3]=valid_entropy

    # Print Result
    print(str(count_epoch)+'\t'+str(train_entropy)+'\t'+str(valid_entropy)+'\t'+
          str(test_entropy)+'\t'+str(train_accuracy)+'\t'+
          str(valid_accuracy)+'\t'+str(test_accuracy))

    # Shuffle train data after one epoch
    all_train_data=np.concatenate((np.array(train_features),np.array(train_labels)
    np.random.shuffle(all_train_data)
    new_train_images=all_train_data[:,:-1]
    new_train_labels=all_train_data[:,-1]
    train_features=copy.deepcopy(np.array(new_train_images))
    train_labels=copy.deepcopy(np.array(new_train_labels))

    # Split mini-batch again
    batch_train_features=[]
    batch_train_labels=[]
    for i in range(int(len(train_features)/num_per_batch)):
        batch_train_features.append(train_features[i*num_per_batch:i*num_per_batch
        batch_train_labels.append(train_labels[i*num_per_batch:i*num_per_batch+num_
```

iter	train_entropy	valid_entropy	test_entropy	
1	0.7154980603645423	0.7172576631944949	0.717675219912451	0.132812
2	0.6556505728881975	0.5999836479093041	0.5985420656951873	0.53990
3	0.5717433468173614	0.553214781323728	0.5517557842403331	0.788461

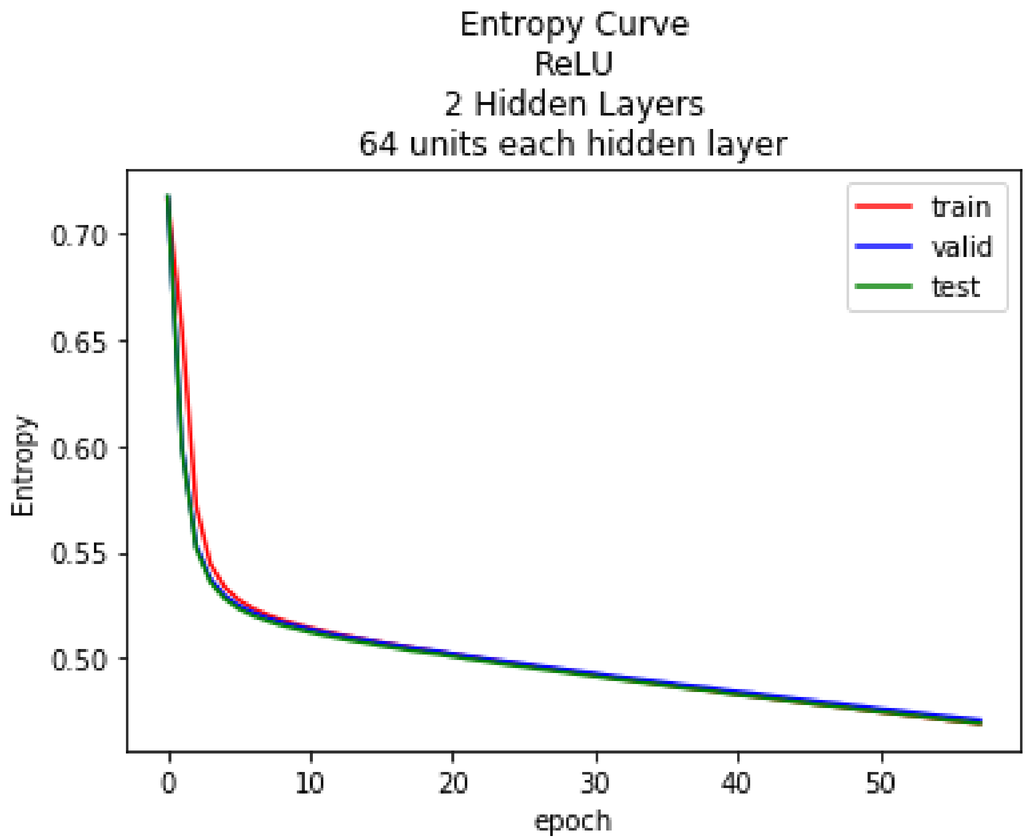
4	0.5444133176056103	0.5374353308218704	0.5357247306892783	0.846554
5	0.5333833916651912	0.5294035599339412	0.5279339803122501	0.867007
6	0.527323986556908	0.5247108295352664	0.5231940921728	0.878826121
7	0.523341085250825	0.5215270295809175	0.5199379521714237	0.886298
8	0.5203624103815641	0.51895929074165	0.5175220702602438	0.8923878
9	0.5179782736331043	0.516815930659365	0.5154145508853039	0.896113
10	0.5160189317753721	0.5152279245913436	0.5137714592678208	0.8991
11	0.5142268126005426	0.5136144840518221	0.5121518409705309	0.9022
12	0.5126303555354536	0.5120528964664969	0.5106742858044347	0.9056
13	0.5111600223553384	0.510673062325818	0.5093992082789386	0.90729
14	0.5098513527973403	0.5094885347711808	0.5080941519835734	0.9090
15	0.5085176411619676	0.5084632743923676	0.5070270532523083	0.9115
16	0.5073046925698975	0.5071070457858239	0.5057574199197051	0.9126
17	0.5061600778878759	0.5060753854880323	0.5047641647908098	0.9146
18	0.5050000164660945	0.5049442914048355	0.5035522574015312	0.9159
19	0.5038971258762027	0.5040199520175933	0.5026304736902907	0.9174
20	0.502811723214853	0.5032041998914394	0.5018549104415869	0.91800
21	0.5017241138140013	0.5019595704121764	0.5007374539761236	0.9194
22	0.5006850305492613	0.5010278091053164	0.49958169003611985	0.920
23	0.4996819999456934	0.5000322661551704	0.49873142536484383	0.922
24	0.49866249265843005	0.4989942445623796	0.49771953475999525	0.92
25	0.4976634068098296	0.49813631498185407	0.49670491862689425	0.92
26	0.4967117898982311	0.49727748424753	0.4958123540896485	0.924779
27	0.4957193659597147	0.4962610720019594	0.4948490122418546	0.9258
28	0.4947409707666461	0.4954348824632195	0.4940323367879214	0.9265
29	0.493823056581227	0.49447438436087326	0.4931469571256218	0.9274
30	0.49287831056043935	0.4936031688512348	0.4921620681091289	0.927
31	0.49191167153402937	0.4927817311519361	0.49133397010230084	0.92
32	0.4909829534412215	0.49170859121245364	0.4903989948437832	0.929
33	0.4900593900082378	0.49082886185177377	0.4895859204971409	0.930
34	0.48919185829056283	0.49000438735750423	0.4887166650036418	0.93
35	0.48826192064931	0.4892658728198083	0.4879066134446592	0.932772
36	0.4873667898318975	0.48841608111918544	0.4870281267387874	0.932
37	0.4864847628113811	0.4874269802143834	0.48606320151996163	0.933
38	0.4855931723019565	0.4866908094458433	0.4852892902285951	0.9344
39	0.4847365940253561	0.48601593878728483	0.48463071212702286	0.93
40	0.4838323099157011	0.48498827804516537	0.4835193290577548	0.936
41	0.4830055976785999	0.48419056105740593	0.48269007799620084	0.93
42	0.4821431891776051	0.48347472161991123	0.4819798077486411	0.937
43	0.4813021417909321	0.4825122003667965	0.4811205130240096	0.9379
44	0.4804302652404236	0.48172933236824245	0.480285264906658	0.9388
45	0.4796077666407842	0.48086131701754486	0.4794950259190727	0.939
46	0.4787464188529167	0.4801278272947345	0.478643225330043	0.93954
47	0.4779334623849294	0.47922055358121357	0.47772773023620835	0.94
48	0.4771062398687963	0.47861393649402945	0.47704483571694817	0.94
49	0.47629418802366175	0.4777037131890451	0.476199736992455	0.9417
50	0.4754639845884814	0.4769713384742356	0.47536748847028076	0.942
51	0.47464735193489355	0.47601182179894785	0.47463427760129157	0.9

```
52        0.47382755969858914        0.47536109462365306        0.4738443056009891        0.943
53        0.4730385579439007        0.4745627277424426        0.4730403549769241        0.94375
54        0.4722493370883699        0.47381024229984464        0.47240013431649747        0.944
55        0.4714318540161252        0.4729549012325201        0.4714953704968348        0.94497
56        0.4706451317363141        0.47223238845236093        0.47065039897116917        0.945
57        0.4698480295652778        0.47150015160510084        0.4699597748742171        0.9458
58        0.46906173910801735        0.47078164873807116        0.4691857505202694        0.946
```

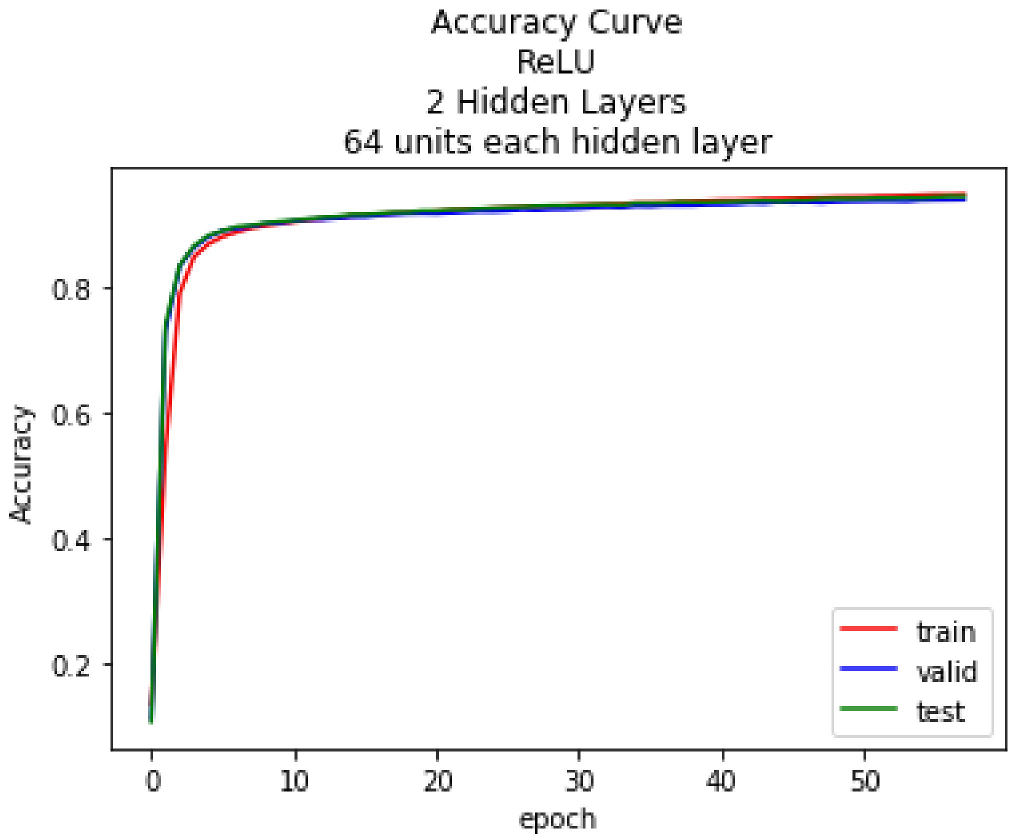
```
---------------------------------------------------------------------------

KeyboardInterrupt                         Traceback (most recent call last)

<ipython-input-9-1386e1e3c895> in <module>()
 92
 93         # Shuffle train data after one epoch
---> 94         all_train_data=np.concatenate((np.array(train_features),np.array(train_labe
 95         np.random.shuffle(all_train_data)
 96         new_train_images=all_train_data[:,:-1]


KeyboardInterrupt:
```

7 Plot Result

In [10]:
```python
plt.figure()
plt.title('Entropy Curve\nReLU\n'+str(len(layer_neuron_num_list)-1)+' Hidden Layers\n
plt.ylabel('Entropy')
plt.xlabel('epoch')
plt.plot(train_entropy_data,'red')
plt.plot(valid_entropy_data,'blue')
plt.plot(test_entropy_data,'green')
plt.legend(['train','valid','test'])
plt.savefig('HW3_entropy_'+str(len(layer_neuron_num_list)-1)+'_hidden_layers_'+str(lay
plt.show()
```




```
In [11]: plt.figure()
         plt.title('Accuracy Curve\nReLU\n'+str(len(layer_neuron_num_list)-1)+' Hidden Layers\n
         plt.ylabel('Accuracy')
         plt.xlabel('epoch')
         plt.plot(train_accuracy_data,'red')
         plt.plot(valid_accuracy_data,'blue')
         plt.plot(test_accuracy_data,'green')
         plt.legend(['train','valid','test'])
         plt.savefig('HW3_accuracy_'+str(len(layer_neuron_num_list)-1)+'_hidden_layers_'+str(la
         plt.show()
```




```
In [12]: sum([a-b for (a,b) in zip(train_entropy_data,valid_entropy_data)])/len(train_entropy_c
```

```
Out[12]: 0.0009075801914749066
```