Assignment_10

April 14, 2021

0.1 REMEMBER: even if a cell is locked / read only, you can always copy the contents to a new cell to modify and execute! Cells are locked here to provide you with a safe fallback, not to stop you from exploring!

0.2 Multilayer-Perceptron for MNIST

In this notebook, we will program a MLP with 1 hidden layer and apply it to the famous MNIST problem. MNIST is a freely available dataset, see e.g. http://yann.lecun.com/exdb/mnist/. It is so popular and often used that is included in the general ML frameworks like tensorflow, pytorch etc.

Here, we use the version provided by Kaggle https://www.kaggle.com/oddrationale/mnist-in-csv, as it gives the data in easily manageable CSV format.

The idea for this notebook follows the books "Make Your Own Neural Network" by T. Rashid and "Neural Networks from Scratch in Python" by Harrison Kinsley & Daniel Kukieła

0.2.1 The MNIST data

This data contains 28x28 pixel wide gray scale images of handwritten digits 0-9. There are 60k training samples, and 10k test samples. The data is arranged in rows, each row consists of 785 values: the first value is the label (0 to 9) and the remaining 784 values are the color pixel values (0 to 255). You are free to use the full set for training (set downsample =1.0 in the next cell), or stick with a reduced set. Execute the next cell to load the training and test data, select a random subset of both and plot a sample image.

Note that the label is stored as the first entry of the row, i.e. it is in sample[0], while the rest of the row are the 28x28 pixel values sample[1:]

[ ]:
```
import numpy as np
import array as arr
import matplotlib.pyplot as plt
from IPython import display
import random
%matplotlib inline

# Load the MNIST data sets, containing 60k training and 10k test data sets.
 ↪Each row contains a single
# 28x28 pixel grayscale image (0,255), with the first entry of the row being
 ↪the label
```

```
training_data_file = open("./mnist_dataset/mnist_train_60k.csv", 'r')
test_data_file = open("./mnist_dataset/mnist_test_10k.csv", 'r')

training_data_list = training_data_file.readlines()
test_data_list = test_data_file.readlines()

training_data_file.close()
test_data_file.close()

# select ratio of data to actually use in training. this of course influences␣
 ↪the training results and speed
# for downsample=0.1, we thus have 1k test data and 6k training data samples
downsample=0.1
test_data_list = random.
 ↪sample(test_data_list,int(len(test_data_list)*downsample))
training_data_list = random.
 ↪sample(training_data_list,int(len(training_data_list)*downsample))

# visualize the first [0] image from the training set and print its label
sample = training_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
```

[ ]:
```
# Helper functions, only change them if you have ruled out any other source of␣
 ↪problems
# Sigmoid activations can run into numerical problems for their limits of 0 and␣
 ↪1. Thus, one can avoid these
# by scaling inputs and outputs.

# Scale pixel values to 0-1
def scale_inputs(image_in):
    #return (np.asfarray(image_in)/255.0 * 0.99) + 0.01
    return np.asfarray(image_in)/255.0

# one-hot encode label vectors
def mod_one_hot(labels_in):
    #labels_out = np.zeros(neurons_per_layer[2]) + 0.01
    #labels_out[int(labels_in[0])] = 0.99

    labels_out = np.zeros(neurons_per_layer[2]) + 0.0
    labels_out[int(labels_in[0])] = 1.0
    return labels_out
```

0.2.2 The neural network

We will now build an MLP to recognize the digits in MNIST and later see how it does on your own handwriting. For this, we will build a simple MLP with one hidden layer. We will feed the network with one sample at a time, and update the weights after each sample.

Since each sample has 28x28=784 pixels, these will be the input neurons. As outputs, we have a vector of length 10, each entry corresponding to the likelihood of the digit as a label. In other words, the true labels from the training data set look for example like this: [0,0,0,0,0,0,0,0,1,0]. This would represent the label "8" (0-9). The network outputs y_hat will have the same shape, but will contain numbers between 0 and 1 - the network is approximate, and will never return a label with absolute certainty. The final output of the network will then be chosen as the digits with the highest likelihood, e.g. [0.1,0.2,0.3,0.1,0.2,0.1,0.4,0,0.8,0.1] would produce a network prediction of "8".

Thus, the input neurons are fixed at 784, the output neurons are fixed at 10, and we are free to choose the number of neurons in the hidden layer. As usual, there will be no activation applied to the input, but we will use a *sigmoid* on the hidden and output layers.

[ ]:
```
# Helper functions:
# Sigmoid Activation Function:
def sigmoid(z):
    # YOUR CODE HERE
    raise NotImplementedError()

# From the lecture, we know that a=sig(z), and dsig/dz=a(1-a). Implement this␣
↪last equation here
def sigmoidgrad(a):
    # YOUR CODE HERE
    raise NotImplementedError()
```

[ ]:
```
assert np.isclose(sigmoid(0), 0.5)
assert sigmoid(1) == 0.5*np.tanh(0.5)+0.5
```

[ ]:
```
assert sigmoidgrad(0) == 0
assert sigmoidgrad(1) == 0
assert np.isclose(sigmoidgrad((1+np.sqrt(5))/2), -1)
```

[ ]:
```

# the class for an MLP with 1 hidden layer
class MLP1H:
    # neuronsperlayer is a vector with 3 entries, alpha is the learning rate
    def __init__(self,neuronsperlayer,alpha):
        self.n=neuronsperlayer
        self.n[0]=neuronsperlayer[0]
        self.n[1]=neuronsperlayer[1]
        self.n[2]=neuronsperlayer[2]
        self.alpha=alpha
```

```
        # initialize activations
        self.a_1=np.zeros((self.n[0],1))
        self.a_2=np.zeros((self.n[1],1))
        self.a_3=np.zeros((self.n[2],1))

        # initialize outputs. It is not strictly necessary to make z self here, as it is locally used only
        self.z_1=np.zeros((self.n[0],1))
        self.z_2=np.zeros((self.n[1],1))
        self.z_3=np.zeros((self.n[2],1))

        # initialize the weight matrices between the input - hidden and hidden - ouput layers
        # and fill them with random values, either normally or uniformly distributed
        # YOUR CODE HERE
        raise NotImplementedError()

        self.activation=sigmoid
        return

    def sse_cost(self,pred,truth):
        # define the summed squared error cost function for a single sample here, including the 0.5
        # YOUR CODE HERE
        raise NotImplementedError()
        return cost

    def forwardpass(self,inputs):
        # We convert the inputs from (784,) to a  true vector (784,1)
        X = np.array(inputs, ndmin=2).T
        # Implement the forward pass
        # YOUR CODE HERE
        raise NotImplementedError()
        return Yhat

    def train(self, inputs, targets):
        # Making sure the targets have the right shape
        tY = np.array(targets, ndmin=2).T
        # Implement the training:
        # first, pass the current sample through the net to fill a,z, etc.
        # next, compute the layer erros \delta, remember the direction
        # then, compute the cost function gradients
        # for tips, see accompanying lecture and notes

        # YOUR CODE HERE
        raise NotImplementedError()
```

```
        # Update the weights
        self.w23  = self.w23-self.alpha * dCd2
        self.w12  = self.w12-self.alpha * dCd1

        cost = self.sse_cost(Y_hat,tY)
        return cost
```

[ ]:
```
# We set up a small ANN here to test the implementation

neurons_per_layer=np.array([2,3,4])
learningrate=0.15
testANN = MLP1H(neurons_per_layer,learningrate)

#testing the cost function
assert testANN.sse_cost(1, 1) == 0
assert testANN.sse_cost(8.123, 12.123) == 8
assert testANN.sse_cost(np.array([2, 3]),np.array([3, 2])) == 1


#testing the correct shapes of w12, w23

# now testing the forward pass


# NOTE: There are no explicit tests for the backpropagation part. If the error␣
 ↪drops in training, that
# typically is a good sign that it works as planned. With this settings as␣
 ↪provided, a performance_on_test
# (see "Performance" section below) should be >92%.
# When debugging the backprop, think about the shapes of the objects, the␣
 ↪possible matrix operations etc.
# As a last resort, work it out by hand - for a much smaller network and a␣
 ↪single layer
```

0.2.3 Generating the Network

[ ]:
```
# now initiate the ANN with the given parameters
neurons_per_layer=np.array([784,100,10])
learningrate=0.15

ANN = MLP1H(neurons_per_layer,learningrate)
```

0.2.4 Training the ANN

[ ]:
```
# Before training the network, let us first pass an input to the untrained␣
 ↪network and see what happens
# The prediction will likely be wrong, but we can test the forward pass in␣
 ↪principal this way
sample = training_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
inputs = scale_inputs(sample[1:])
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

[ ]:
```
# Training the network

nEpochs = 10

# Initializing the cost history plot
fig=plt.figure(figsize=(80, 40))
fig, ax = plt.subplots(figsize=(20, 10))
ax.size=(12,12)
plt.xlabel('Epochs',fontsize=24)
plt.ylabel('Costs C',fontsize=24)

# loop over epochs, remember, we train and update with single samples
for i in range(1,nEpochs+1):
    lcost=0.
    localcost=0.
    # loop over all training samples
    for record in training_data_list:
        data_list = record.split(',')
        # scale and encode inputs / labels
        inputs = scale_inputs(data_list [1:])
        labels= mod_one_hot(data_list [0:])
        # train and compute cost for each sample
        localcost=ANN.train(inputs,labels)
        lcost=lcost+localcost

    # Compute the training and test costs after each epoch
    Cost_Training = 0.
    Cost_Test     = 0.
    for record1 in training_data_list:
        # evaluate current model on the full training set and compute cost
        data_list  = record1.split(',')
        inputs = scale_inputs(data_list[1:])
        labels= mod_one_hot(data_list[0:])
```

```
        labels= np.array(labels,ndmin=2)
        y_hat=ANN.forwardpass(inputs)
        Cost_Training = Cost_Training + 0.5*np.sum(np.square(y_hat-labels.T))

    for record2 in test_data_list:
        # evaluate current model on the full test set and compute cost
        data_list= record2.split(',')
        inputs = scale_inputs(data_list[1:])
        labels= mod_one_hot(data_list[0:])
        labels= np.array(labels,ndmin=2)
        y_hat=ANN.forwardpass(inputs)
        Cost_Test = Cost_Test + 0.5*np.dot(y_hat.T-labels,(y_hat.T-labels).T)

    # normalize errors
    Cost_Training = Cost_Training / len(training_data_list)
    lcost = lcost / len(training_data_list)
    Cost_Test     = Cost_Test     / len(test_data_list)

    ax.scatter(i,Cost_Training,color='g',label='Training Data (epoch)')
    ax.scatter(i,lcost,color='k',label='Training Data (sample)')
    ax.scatter(i,Cost_Test,color='r',label='Test Data (epoch)')

    if i==1:
        ax.legend()
    display.display(plt.gcf())
    display.clear_output(wait=True)
print("training complete!")
print("Cost on test data (epoch): ",Cost_Test)
print("Cost on training data (epoch): ",Cost_Training)
print("Cost on training data (sample): ",lcost)
```

0.2.5 Querying the trained network

[ ]:
```
# let us now see if the training brought progress
sample = test_data_list[0].split(',')
image_array = np.asfarray(sample[1:]).reshape((28,28))
plt.imshow(image_array, cmap='Greys',interpolation='None')
print ("True label = ",sample[0])
inputs = scale_inputs(sample[1:])
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```

0.2.6 Performance

[ ]:
```
# evaluate the network perfomance by computing the ratio of correct predictions␣
 ↪on the full test set
test_pred=np.zeros(len(test_data_list))
performance_on_test = 0
j=0
for record in test_data_list:
    data_list= record.split(',')
    inputs = scale_inputs(data_list[1:])
    labels= mod_one_hot(data_list[0:])
    correct_label = np.argmax(labels)
    y_hat=ANN.forwardpass(inputs)
    pred_label = np.argmax(y_hat)
    test_pred[j]=np.argmax(y_hat)
    j+=1
    if (pred_label == correct_label):
        performance_on_test+=1
print ("Performance = ", (performance_on_test / len(test_data_list))*100,"%")
```

[ ]:
```
# Here are random samples from the test data, and the network prediction

from random import randint

fig, ax = plt.subplots(10,10,figsize=(10,10))
ax = ax.flatten()

for i in range(100):
    r = random.randint(0, len(test_data_list))
    sample= test_data_list[r].split(',')
    image= np.asfarray(sample[1:]).reshape((28,28))
    ax[i].imshow(image, cmap='gray_r')
    ax[i].set_title(int(test_pred[r]), fontsize=15)
    ax[i].axis('off')

plt.tight_layout()
plt.show()
```

0.2.7 Your own handwriting

Check the networks prediction on your own handwriting.Either write on a piece of paper and take a picture, or use e.g. gimp to directly create a sample image. Save as png, and store in the folder indicated below. On linux, you can convert e.g. a jpg with 'convert Name.jpg Name.png'

[ ]:
```
from skimage.transform import resize
import matplotlib.image as mpimg
from skimage.color import rgb2gray
```

```
# place own image in the folder, adjust name
own_image=mpimg.imread("./mnist_dataset/own/drei.png")
plt.imshow(own_image)
own_image = rgb2gray(own_image)
own_image = resize(own_image, (28,28))

own_image = 255 - own_image.reshape(784)

inputs = scale_inputs(own_image )
outputs = ANN.forwardpass(inputs)
print("Predicted label = ",np.argmax(outputs))
```