magenta`github.com/mathhat`

Oblig 2 Nerual Network

INF4490

Joseph Knutson
josephkn joseph.knutson@fys.uio.no

November 23, 2017

Contents

1 Initialization

Before defining forward and backward movement, we need to define some datatypes. I use a matrix notation for my input, weights (1 and 2), hidden activation (f_{hidden} in the code , but a_{hidden} in the book) and the output spikes (f in the code, but h_k or a_k in the book).

Since I add extra indice to my structure (to efficiently implement bias in my forward movement), the __init__ function looks a little messy. I start by giving my input matrix an extra row of bias inputs. Then I do the same for the hidden potential so the second bias can be implemented.

The rest of the script defines the weight matrices as described in the book. Both carry an extra vector to compensate for bias operations.

```
def __init__(self, inputs, targets, nhidden):

  self.inputs2 = np.zeros((len(inputs),len(inputs[0])+1))
      for i in range(len(inputs)):
          for j in range(len(inputs[0])):
              self.inputs2[i,j] = inputs[i,j]
          self.inputs2[i,40] = -1    #extra row vector of bias

  self.y_h = np.zeros((len(inputs),nhidden+1))
  self.y_h[:][-1] -= 1  #extra row vector of bias

  self.y = np.zeros((len(inputs),len(targets[0]))) #output node
    spikes



  #Weights are also given an extra index for the bias input
  #scaling constant 1/sqrt(number of hidden nodes)

  self.weights1 = np.random.random((nhidden,len(inputs[0]) + 1))
    *1./np.sqrt(nhidden+1)
  #weights between input n hidden
  self.weights2 = np.random.random((len(targets[0]),nhidden+1))*1./
    np.sqrt(len(targets[0])+1)
  #weights between hidden and output

  #the following forloops randomly mirror the weights around 0 (so
    we get negative weights too)

  for j in range(nhidden):
      for i in range(len(inputs[0])-1):
          if (np.random.randint(2)):
              self.weights1[j,i] = self.weights1[j,i]*(-1)
  for i in range(nhidden-1):
      for j in range(len(targets[0])):
          if (np.random.randint(2)):
              self.weights2[j,i] = self.weights2[j,i]*(-1)
```

2 Forward Movement

As the formulas in the book describe, I first calculate a_{hidden} into the matrix f_h. I then use these values to calculate the output spikes in the f matrix. I've implemented the dot product and the g function seperately.

```
def g(self, h):                          #spike function
    return(1./np.exp(-self.beta*h))

def add_up(self, w, x):                   #vector dot product
    return (sum(w * x))

def forward(self, inputs):
  v = self.weights1
  w = self.weights2
  for n in range(len(inputs)):
      for i in range(len(self.y_h[n])-1):
          self.y_h[n,i] = self.g(self.add_up(np.append(inputs[n],
    [-1]),v[i,:]))
      for i in range(len(self.y[n])):
          self.y[n,i] = self.g(self.add_up(self.y_h[n], w[i,:]))
  return(self.y[n])
```

3 Backward Movement, Train and Recall

This part consists of calculating the deltas and updating the weights and is the very definition of an iteration. Pretty straight forward to implement once the matrices are of correct dimensions:

```
    def train(self, inputs, targets, iterations):
        w2 = self.weights2
        w1 = self.weights1

        for I in range(iterations):
            self.forward(inputs)

            y = self.y  #end node spikes
            y_h = self.y_h  #hidden layer spikes

            delta_k = (y-targets)*y*(1-y) #deltas
            delta_h = y_h*(1-y_h)*np.dot(delta_k,w2)

            #weights being updated
            w1 -= self.eta* np.dot(delta_h[:,:-1].T,self.inputs2)
            w2 -= self.eta*np.dot(delta_k.T, y_h)
```

The recall function is quite important because it is our "classifier". As we feed it a single input vector (my train and forward functions only accept input matrices, so I've edited the forward algorithm to only accept vectors below) the function returns the classification vector e.g. [0,0,0,0,1,0,0,0]. These vectors are what we will use to create confusion matrices and calculate the accuracy of our model.

```
    def recall(self, inputs):
        n=0
        v = self.weights1
        w = self.weights2
        #copy paste of most of the forward function
        for i in range(len(self.y_h[n])-1):
            self.y_h[n,i] = self.g(self.add_up(np.append(inputs, [-1]),v[i,:]))
        for i in range(len(self.y[n])):
            self.y[n,i] = self.g(self.add_up(self.y_h[n], w[i,:]))
  #This bit rounds the vector's -
  #largest element to 1 and the rest to 0.
  #This way we can create a proper conf. matrix
        swag = np.copy(self.y[n])
        swag[np.argmax(swag)] = 1
        return(np.round(swag))
```

4 Confusion Matrices

Before writing the earlystopper, I need to implement the accuracy, which we know is the sum of diagonal elements in our confusion matrix divided by the sum of all elements. The confusion matrix shows us how well trained the neural network has become, and below I've printed a couple of them.

```
    def confusion(self, inputs, targets):
        self.conf = np.zeros((len(targets[0]),len(targets[0]))) # confusion matrix dim: 8x8
        non_spikes = 0
        for i in range(len(inputs)):
            recall = self.recall(inputs[i])      #
            if 1 in recall:
                self.conf[np.argmax(targets[i]),np.argmax(recall)] += 1
            else:
                non_spikes +=1
        accuracy = float(np.trace(self.conf))/(np.sum(self.conf)+non_spikes)
        print(accuracy)
        return (accuracy)
```

Training for about 100 iterations with 10 hidden nodes, we can get a pretty good accuracy in both the validation and test set:

```
net.train(train, train_targets, iterations=100)
net.confusion(test,test_targets)
net.confusion(valid,valid_targets)
```

This code prints:

```
 joseph@Lappy:~/Documents/Inf4490/2/code$ time python3 movements.py
[[ 14.   0.   0.   0.   0.   0.   0.   0.]
 [  0.  13.   0.   0.   0.   1.   0.   0.]
 [  0.   0.   9.   0.   0.   0.   0.   0.]
 [  0.   0.   0.  12.   0.   0.   0.   0.]
 [  0.   0.   0.   0.  21.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   7.   0.   0.]
 [  0.   0.   0.   0.   0.   3.  20.   0.]
 [  0.   0.   0.   0.   0.   0.   0.  11.]]
test accuracy: 0.963963963964
[[ 11.   0.   0.   0.   2.   0.   0.   2.]
 [  0.  14.   0.   0.   0.   0.   0.   0.]
 [  0.   0.  13.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   9.   0.   0.   0.   0.]
 [   2.    0.    0.    0.   12.    0.    0.    0.]
 [   1.    0.    0.    0.    0.   14.    0.    0.]
 [   0.    0.    0.    0.    0.    0.   10.    0.]
 [   0.    0.    1.    2.    0.    0.    0.   19.]]
validation accuracy: 0.910714285714

real    0m4.039s
user    0m4.104s
sys 0m0.596s
```

In the test set, we can see that class 6 and 7 are most frequently misclassified for each other. The validation test has however no single pair of classes that are missed interchangeably.

5 Earlystopping

The earlystopper puts our training function in a loop. This loop runs little intervals of 5-15 iterations. After each interval, the accuracy relative to the validation set is compared to the accuracy of the previous model (15 iterations in the past). Once the validation accuracy stops sinking (converges in my case around 90-96%) we stop training.

```python
def earlystopping(self, inputs, targets, valid, validtargets):
        maxiter = 5000
        checkpoint = 15 #each time we check if accuracy in validation set is failing
        self.train(inputs,targets,checkpoint) #initial training
        accuracy0 = self.confusion(valid,validtargets)  #initial accuracy

        local_optima_dodger = 0  #counter for worse than previous accuracies

        #for each checkpoint on our way to the max iteration treshold
        for i in range(2,int(maxiter/checkpoint)):
            self.train(inputs,targets,checkpoint)    #train 15 times more
            accuracy = self.confusion(valid,validtargets) #calcualte accuracy
            if accuracy <= accuracy0:                #if new accuracy sucks
                local_optima_dodger +=1              #count bad accuracy development
            else:
                accuracy0 = accuracy                 #else, we set new accuracy to best
            if local_optima_dodger > 15:
                break

        print("final accuracy = ", accuracy)
        print("number of iterations of max 5000: ", ((1+i)*checkpoint ))
        return(accuracy)
```

6 K folds

My K folds method samples k (30) random vectors of my input matrix and turns it into a validation set, while the rest is used of training. We then run the Earlystopper on the training set and validate it with the k validation targets. This sampling, training and validation is repeated 10 times, giving us 10 accuracies which we can calculate the mean and variance value of. This is not a traditional k-fold algorithm, as I'm using k random samplings instead of k fixed datasets, but the algorithm runs great nonetheless, which I hope can excuse my misunderstanding of the formula.

The k-folds algorithm is found below and is implemented in the movements.py file instead of the mlp.py file due to the algorithm mainly being about sampling the movement and target data (which is found in movements.py).

```python
[...]
import random
random.seed(1)
def kfold(movements,target): #the input and targets are for day 1-3

    k = 30              #30 out of 447 vectors are for validation
    percentage = float(k)/len(movements) #that's less than 10%
    print("percentage of set in validation: ",percentage)
    k_times = 10          #this is how many times we bother to create
                          #- a new random test set out of the data
    accuracy = np.zeros(k_times)
    for times in range(k_times):
        k_valid = random.sample(list(range(len(movements))),k)

        k_train = list(range(len(movements)))
        i = 0
        while len(k_train) > len(movements)-k:
            if k_train[i] in k_valid:
                k_train.pop(i)
            else:
                i+=1

        k_targets = []
        k_valid_targets = []

        for i in range(len(k_train)):
            s = k_train[i]
            k_train[i] = movements[s][0:40]
            k_targets.append(target[s])
        for i in range(len(k_valid)):
            s = k_valid[i]
            k_valid[i] = movements[s][0:40]
            k_valid_targets.append(target[s])
        net = mlp.mlp(np.asarray(k_train), np.asarray(k_targets),
    hidden)
        accuracy[times] = net.earlystopping(np.asarray(k_train), np
    .asarray(k_targets), np.asarray(k_valid), np.asarray(
    k_valid_targets))
    print(np.argmax(accuracy))
    print("mean of the accuracies = ", np.mean(accuracy))
    print("variance of the accuracies = ", np.sqrt(np.mean(accuracy
    **2)-np.mean(accuracy)**2))
kfold(movements,target) #calling it
```

The accuracies were horrible until I decreased the amount of hidden nodes from 10 to 6. Probably a good idea to comment what number of nodes worked out best (see next section).

```
mean of the accuracies =  0.943333333333
variance of the accuracies =  0.0422952584682
```

7 Results and Hidden Nodes

Experimenting with the hidden nodes number, I found that 10 gave me very nice accuracies (6 for k-folds however).

To provide some results regarding the ideal number of hidden nodes, I've run my earlystopper on the data with a number of hidden nodes ranging from 6 to 12. Here is some output using the training set to train and test and validation set to calculate the accuracy:

```
N_Hidden_nodes = 6
final validation accuracy =  0.911
number of iterations of max 5000: 345 #<-iterations until convgence
test-set accuracy: 0.928
```

```
N_Hidden_nodes = 7
final validation accuracy =  0.92
number of iterations of max 5000: 360
0.964
```

```
N_hidden_nodes = 8
final validation accuracy =  0.910714285714
number of iterations of max 5000:  300
test accuracy = 0.955
```

```
N_hidden_nodes = 9
final validation accuracy =  0.9375
number of iterations of max 5000:  375
test accuracy = 0.955
```

```
N_hidden_nodes = 9
final validation accuracy =  0.9375
number of iterations of max 5000:  375
test accuracy = 0.955
```

```
N_hidden_nodes = 10
final validation accuracy =  0.919642857143
number of iterations of max 5000:  315
test accuracy = 0.964
```

```
N_hidden_nodes = 11
final validation accuracy =  0.9375
number of iterations of max 5000:  315
test accuracy = 0.955
```

At 11 nodes we see a big test and validation accuracy. Perhaps this is the best model I've found?

```
N_hidden_nodes = 12
final validation accuracy =  0.589285714286
number of iterations of max 5000:  345
0.603603603604
```

At 12 nodes we can see a huuge drop in accuracy, this is probably because my convergence treshold is too easily activated.

More nodes = less accuracy and more computation time from here on out.

7.1 Best K-Fold Model

Out of our 10 K-fold models, the fourth model scored 100% accuracy on the validation (the k input vectors that are independent from the training set) and test set (the set used for testing in our other algorithms):

```
4th K-fold model:
N_hidden_nodes = 6
final validation accuracy (k input vectors) =  1.0
Test set error (25% of the data)  = 1.0
number of iterations of max 5000:  330

[[ 14.   0.   0.   0.   0.   0.   0.   0.]
 [  0.  14.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   9.   0.   0.   0.   0.   0.]
 [  0.   0.   0.  12.   0.   0.   0.   0.]
 [  0.   0.   0.   0.  21.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   7.   0.   0.]
 [  0.   0.   0.   0.   0.   0.  23.   0.]
 [  0.   0.   0.   0.   0.   0.   0.  11.]]
```

In order to extract the best model, some rearrangement of my code had to take place. I've changed it back so that you can run my k-folds without being stuck on my fourth model.

The K-fold models might be overfitting our data due to ¿ 90% of the data is being used to train them. Our validation accuracy is something to be happy with, but alot of the training data is the same data as the "test-set", which means that scoring a 100% here isn't equally impressive. To find out if we have overfit, we need more data.