Neural Networks Report #7

Introduction

This homework is about the concepts studied in the class, associative memory and transformer model. Bidirectional associative memory is a type of recurrent neural network. A transformer is a deep learning model that adopts the mechanism of attention, weighing the influence of different parts of the input data. BAM problem is implementation of the theory and analysis of result while the Transformer problem is discussion of the details of the model and possible improvements.

Problem1

This is more computational problem so I will briefly explain by code steps in each part and then analyze the result of the process.

a) Code is show below

```
# helper functions

def gen_binary():
    x = random.rand()
    if x < 0.5:
        return -1
    else:
        return 1

def gen_data(size):
    X = []
    Y = []
    for i in range(size):
        x = []
        y = []
        for j in range(512):
            x.append(gen_binary())
            y.append(gen_binary())
            y.append(gen_binary())
        X.append(x)
        Y.append(y)
    return X, Y
```

```
# Problem 1 a
x1, y1  = gen_data(50)
x2, y2  = gen_data(100)
x3, y3  = gen_data(200)
```

First, the function that generates either 1 or -1 with 50% probability is given. It is created from a random uniform variable as it takes ½ prob to be in half region. Then the function is created that takes a size argument of the number of samples and returns the data matrix where each column is for each s value. They are used to create 3 data samples.

b) In this step we just follow the equation of the algorithm for the particular sample and following the formulas we make a function that transfers new values to x and y by finding the matrix w from their given values.

```
# Problem 1 b
def transfer(x, y):
    w = 0
    xr = []
    yr = []
    for i in range(len(x)):
        w += np.matmul(x[i], np.transpose(y[i]))
    for i in range(len(x)):
        yi = np.sign(np.matmul(np.transpose(w), x[i]))
        xi = np.sign(np.matmul(w, y[i]))
        xr.append(xi)
        yr.append(yi)
    # print("in fun")
    return xr, yr
```

Testing the function, we get the result that makes sense.

c) In this one we again just follow the algorithm and use some advantages of numpy library to make the life easier.

```
# helper function for problem c

def error_average(r, r_hat):
    count = []
    #print(len(r))
    #print(len(r_hat))
    #print(len(r[0]))
    #print(len(r_hat[0]))
    for i in range(len(r)):
        c = 0
        for j in range(len(r[0])):
            if (r[i])[j] != (r_hat[i])[j]:
                #print("here")
                c+=1
        count.append(c)
    return np.mean(count)

def BAM(x, y, num):
    total_count = []
    xi = x
    yi = y
    # print(len(xi))
    avg = []
    for i in range(num):
        #print(i)
        xi, yi = transfer(xi, yi)
        #print(xi)
        avg.append(error_average(yi, y))
    plot2 = plt.figure(1)
    plt.plot(np.arange(1, num+1), avg, linestyle='solid' )
    plt.show()
```

So, for each sample, we make 10 cycles of transferring mapped information from x to y and again from y to x, and store the errors. Then we compute the average of errors in each sample in each iteration and plot the result. The accuracy decreases with time and reaches 0.42 in the end. This can be explained as with more iteration memory quality decreases giving more and more inaccurate result, but again stops decreasing very quickly and its average is less then one error which is very small due to zero noise case.

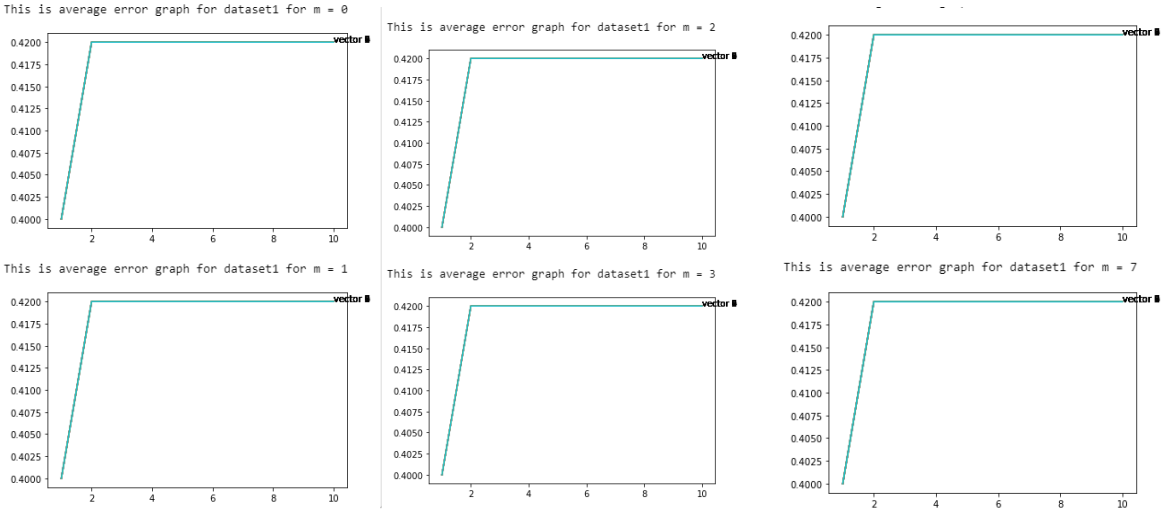
d) The code of this problem is similar to the function created previously, I only added parameter m that changes the values of x in all samples at random indices. The code is shown below

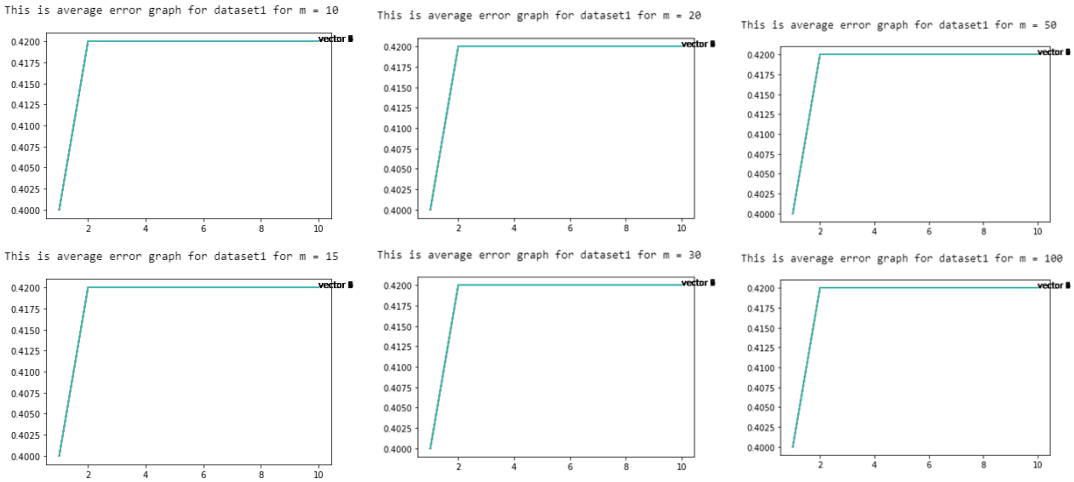
```
def poison(x, m):
    xi = []
    arr=random.randint( len(x[0]), size=(m))
    for s in range(len(x)):
        xx = []
        for i in range(len(x[0])):
            #print(x[s])
            num = x[s][i]
            #print(x[s][i])
            if i in arr:
                #print(num)
                num = np.negative(num)
                xx.append(num)
            else:
                xx.append(num)
        xi.append(xx)
    #print(np.shape(xi))
    return xi

def noisyBAM(x, y, num):
    total_count = []
    xi = x
    yi = y
    avg = []
    w = np.ones((512, 1024))
    for i in range(num):
        xi, yi= transfer(xi, yi)
        avg.append(error_average(yi, y))
    return avg
```

Then using the for loop we generate 10 different vectors and see the behavior.

```
for v in [0, 1, 2, 3]:
    print("This is average error graph for dataset1 for m =", v)
    for i in range(10):
        #print(x1)
        xi = poison(x1,v)
        avg0 = noisyBAM(xi, y1, 10)
        ix = np.arange(1, 11)
        plt.plot(ix, avg0, linestyle='solid')
        plt.text(ix[-1], avg0[-1], 'vector {i}'.format(i=i))
    plt.show()
```

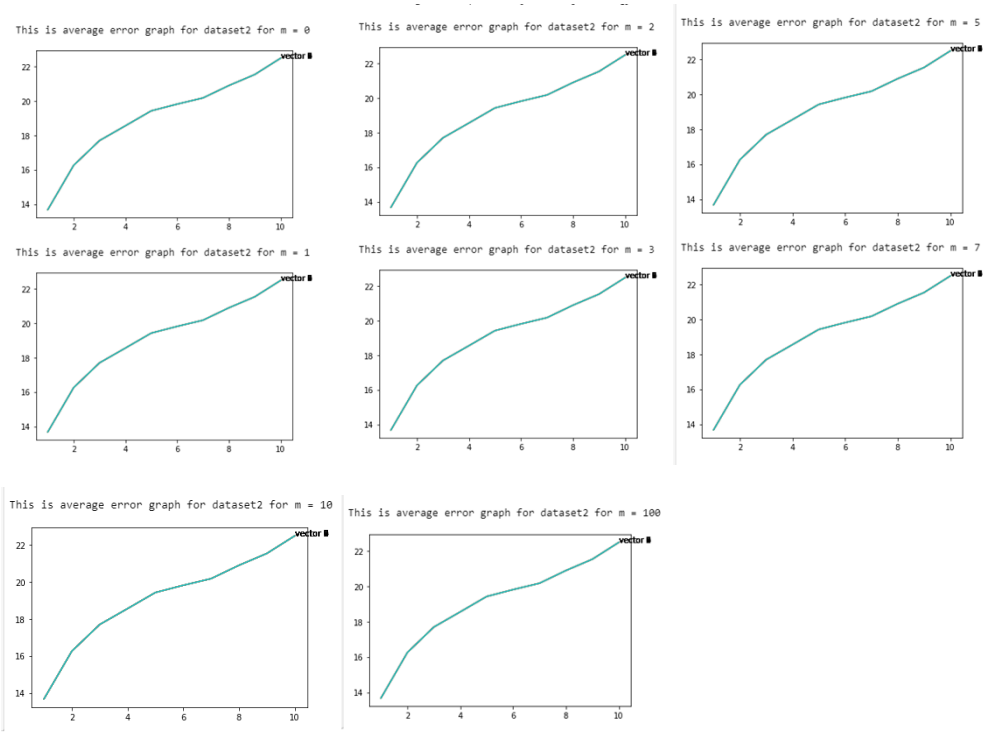
The result is very surprising as all accuracies are the same and independent in the number of mismatches. This gives a high suspicion that the code is incorrect even if it was checked huge number of times. So even thinking that something is incorrect in my code I will continue to discus with the hope that this should be as it is.

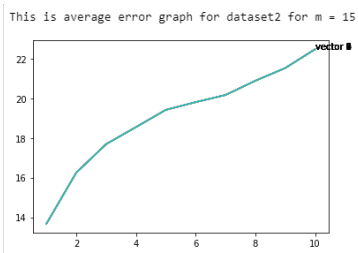


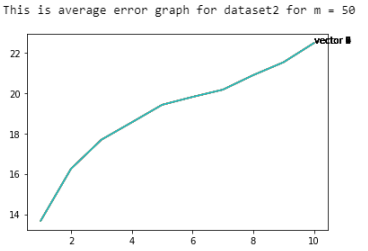


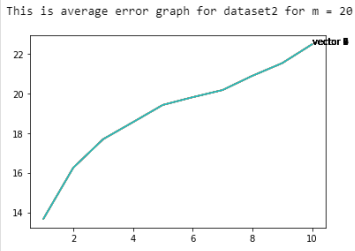
The reason may be that m is still vary small and w is very flexible for x and y as it makes a good mapping from x to y which can be proven easily mathematically. Thus, this behavior should take place.

e) The same problem happens in other datasets as they are the same independent of the number of errors in the x and their place.

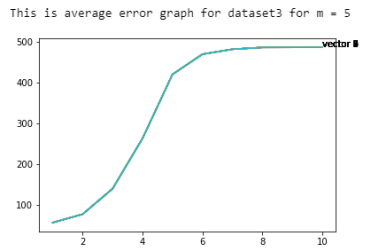


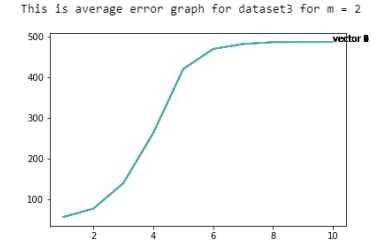


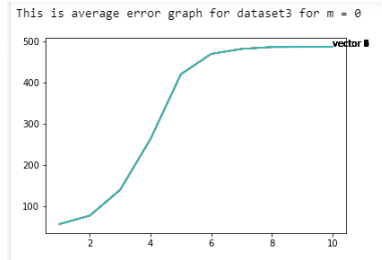


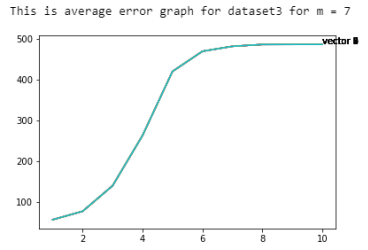


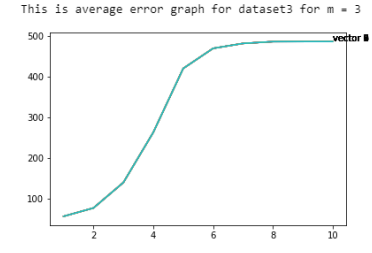
For dataset3 we have

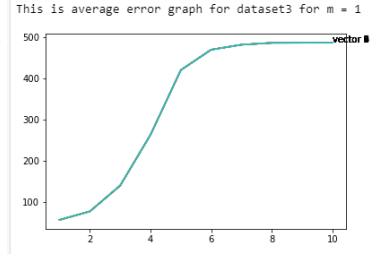


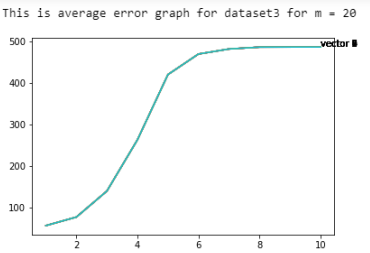


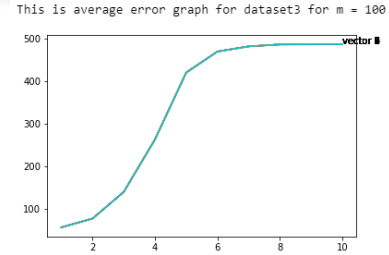


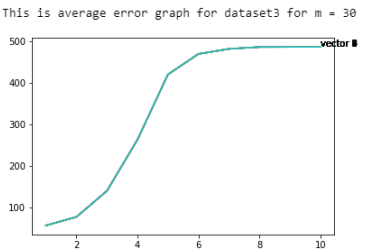


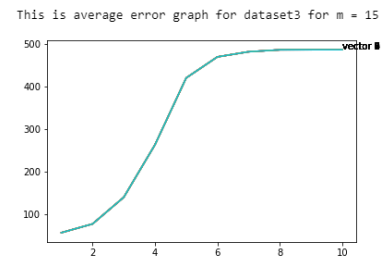


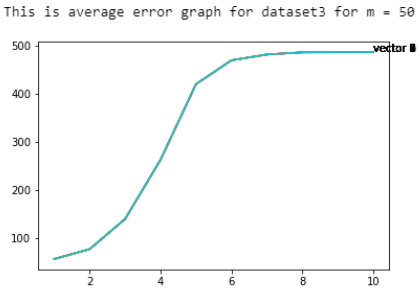












Each dataset has different shape, so for a latger one, accuracy decreases by a large amount after some time which already gives a sence of some behavior. Interestingly that the shape resembles a sigmoid function which in turn was created from the behavior of brain. After some number of iterations some information is forgotten which gives that large number of errors and it reaches some saturation after whcich it is constant. The fact that all data is mapped to the same shape says that algorithm does not care about noise and remebers information with same efficiency under any condition. So this small noise is just learnt as a real data and fits the same accuracy, just the weight value is changed from the one without noise.

Problem2

a) The major component of Transformer is head attention that computes not only the meaning of the input word (for example) but its importance according to its relative position in the whole sentence. So, it answers a question on what part of input it should concentrate. For this, it takes 3 inputs which are query, key and value which are abstract vectors that concentrate on different parts of the input and take different features from it. Then they produce one attention vector using the equation

$$\text{Attention}(Q,K,V)=\text{softmax}(QK^T/\sqrt{N_{que}})V$$

Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention multiple times in parallel. The independent attention outputs are simply concatenated and linearly transformed into the expected dimensions. According to the paper, “multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.”.

$$\text{MultiHead}(Q,K,V)=[\text{head}_1;\ldots;\text{head}_h]W^O \text{where head}_i$$

$$=\text{Attention}(QW_i^Q,KW_i^K,VW_i^V)$$

where W_i^Q, W_i^K, W_i^V, and W^O are parameter matrices to be learned.

In simple words, using many heads we can project multidimensional data into smaller different dimensions and try to measure similarities and differences between them. We don’t specify any constrains and let the network learn different similarities measures as it starts from different initial values; then this measures combine and give more sophisticated result. Thus, the network will learn the data from different point of view using parallelism concept and give more objective result than would be done using the single-head. Parallelism allows do it at the same time so there is no time disadvantage. As there are multiple heads the dimensions of vectors and matrices will be than reduced; if there are h

parallel attention layers, or heads, then each of these we use dk = dv = dmodel/h . Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

However, in this problem the embedding dimensions of the single and multi head attention are the same which will require much higher computational cost for the multi-head case. Additionally, more parameters such as weight matrices for each vector need to be optimized in training process. As a result, multi-head technique gives a big advantage by making a model more objective and robust and at the same time single head needs much computational cost which is also key thing in the large-scale ML systems.

b) In the paper explanation, author adds a scaling factor √(1/Nque), motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning. Adding more details to the last sentence the softmax function is an S shaped function ranging from 0 to 1 at is min and max inputs. As Nque = Nkey, if the input vector is large, we get large Q and K vectors, as they have nonnegative values their dot product will have larger value which than will be mapped by softmax to the values which are all very close to unity. In this region of graph, the slope of softmax is very steady and close to zero which means that the learning process that depends of the gradient of this result will be very slow and thus inefficient. The simple solution is to distribute mapping by softmax into a larder range making small input size not change much while displace larger inputs farther from the 1 attention value. That's why we divide the measure of similarity between Q and K by its square root of size. Square root is chosen to make the value just smaller than it should be to make gradient value significant, but not deviate from its true value much.

c) Going back to the attention function, we can show it with original Q and K input as

$$\text{Attention}(Q,K,V)=\text{softmax}(Q*Wq*(Wk^T)*K/\sqrt{Nque})V$$

$$=\text{softmax}(Q*Wqk*K/\sqrt{Nque})V$$

So, instead of mapping input for key and query to get weighed vectors we can generate the ready matrix and multiply the operation in the function itself which will reduce number of computational steps and so the learning time. However, both weight matrices are high dimensional which are NmodelxNque where Nque = (Nmodel/Nheads), so this multiplication can be computationally expensive which gives a burden to simplify the calculation. This can be solved by setting Wq = Wk which can be reasonable as original data anyway is mapped into 1 head and there are already multiple heads to find different similarity measure for the same input; thus, this move will not give the similarity disadvantage. Then, multiplication of the matrix by the same transpose matrix will cost less as it will give symmetrical result, so it halves the computation cost. In summary, although making matrix replacement will reduce number of steps it is not very efficient due to the large computational requirement, nonetheless making some smart approximation will only slightly reduce the overall model accuracy while will simplify the problem and make this case possible.

Reference

[1] https://papers.nips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[2] https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html