1XC3 Final Project

Muhammad Muneeb Hassan

August 2023

Sensitivity Analysis Tables

Following I have included the sensitivity Analysis Tables and their descriptions for the model with different parameters.

Table 2

Table 2: epochs = 100000, train split = 0.003, **Learning Rate (LR) is variable.**

Test	Cost Train	Cost Validation	Train Accuracy [%]	Validation Accuracy [%]
LR = 0.0005	0.423329	0.463773	66.67	59.16
LR = 0.001	0.113295	0.358953	95.14	75.38
LR = 0.005	0.026180	0.465556	98.61	73.29
LR = 0.01	0.014394	0.394020	98.61	77.22

1.1.1 Description

In Table 2, we kept epochs and train_split constant at 100000 and 0.003 respectively. A moderate learning rate (e.g., LR = 0.001 to LR = 0.01) leads to better convergence and accuracy compared to LR values that are too small (slow convergence) or too large.

Table 3

Table 3: Learning rate = 0.005, train split = 0.003, **epochs is variable**

Test	Cost Train	Cost Validation	Train Accuracy [%]	Validation Accuracy [%]
epochs = 100	0.485698	0.504600	58.33	49.98
epochs = 1000	0.459125	0.487958	58.33	55.60
epochs = 10000	0.440553	0.476903	60.42	56.87
epochs = 100000	0.026180	0.465556	98.61	77.22

1.2.1 Description

In Table 3, increasing the number of epochs generally leads to better training and validation accuracy up to a certain point. However, it causes longer training times and over-fititng

Advantages and disadvantages of Table 2 and 3

Following is a detailed description of changing the respective parameters in table 2 and table 3.

Advantages of increasing Learning Rate

Faster convergence during early epochs, leading to quicker training. Higher Learning rate allows algorithm to take 'larger steps' in updating weights during each iteration. Higher **LR** values are beneficial when the model's weights are randomly initialized and are far from the optimal results. Larger **LR** helps the model to escape the shallow local minimas.

Disadvantages of increasing Learning Rate

Risk of overshooting resulting in divergence. Imagine this process as descending a hill to reach the lowest point (optimal weights) that represent minimal loss function. If learning rate is too large, you might take steps that are too big that you jump over the optimal point and end up on the other side of the hill. This can lead to loss function increasing rather than decreasing, causing training to become unstable.

Advantages of decreasing Learning Rate

Smoother convergence and better chance to find optimal weights. A lower **LR** leads to smaller weight updates in each iteration which results in a smoother convergence.

Disadvantages of increasing Learning Rate

Slower convergence during early epochs. With smaller **LR**, size of weight updates in each iteration is smaller and thus causes slower convergence.

Advantages of increasing epochs

Higher chance of reaching convergence. Increasing the number of epochs allows the algorithm more 'opportunities' (iterations) to converge to the weights of the optimal numbers.

Disadvantages of increasing epochs

Longer training time because the number of iterations increased. It took me 10 minutes to fill out table 2 because every sample took 100,000 epochs. However, table 3 took me no more than 2 minutes to fill out because number of epochs were less (100, 1000, 10000, 100000 respectively.)

Advantages of decreasing epochs

Faster training and reduced risk of over-fitting because total number of iterations reduced.

Disadvantages of decreasing epochs

May not allow algorithm to fully converge the weights to optimal values because the algorithm gets less number of 'opportunities' (iterations) to update weights.

Table 4

Table 4: epochs = 100,000, Learning rate = 0.005, **Train_split (TS) is variable**

Test	Cost Train	Cost Validation	Train Accuracy [%]	Validation Accuracy [%]
TS = 0.0003	0.000043	0.896585	100.00	53.17
TS = 0.003	0.024606	0.407476	98.61	76.53
TS = 0.005	0.030787	0.362535	97.92	79.2
TS = 0.01	0.051442	0.301591	96.47	81.56
TS = 0.1	Error			

Segmentation Fault

When Train_split = 0.1, we will get the segmentation fault (core dumped). I used gdb to find which line of code this is happening.

Listing 1: gdb environment

```
gdb) run
Starting program: /home/hassam85/COMPSCI1XC3/brian
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/
    libthread_db.so.1".
End of the file.

Breakpoint 1, BackwardPass (num_train=4812, X_train=0
    x7ffffe71150, Y_train=0x7ffffe5e490, W2=0x7ffffe842d0,
     W3=0x7ffffe84550, W4=0x7ffffe84190, b2=0x7ffffe84050,
     b3=0x7ffffe83fb0, b4=0x7ffffe83fa0, a2=0x7ffffb94010,
     a3=0x7ffffad8090, a4=0x7ffffac53d0) at mymodel.c:166
166         double a3_squared_complement[num_neurons_layer3
    ][num_train];
```

Reason for the error

The error was in the following command:

double a3_squared_complement[num_neurons_layer3][num_train];

This number of rows in this array is equal to **num_train.** I printed the value of **num_train** in the **gdb environment** and it came out to be **4812(These are the total number of samples we are going to train).** This is a huge number. the more training samples we pick, the more memory we need to allocate for saving weights and biases which can't be done on stack which is resulting in **Stack Overflow.** This is why we get the Segmentation fault.

Part C. (+5 bonus points)

I had to play around with all the parameters. I kept the **train_split to 0.01** which meant I will be using 1 percent of the data.

I started with a lower number of neurons and gradually increased them. At the end, I kept the original number of neurons which were **40 for Layer 2** and **20 for Layer 3.**

For the Learning Rate, I chose a **moderate value of 0.006.** I played around with different Learning Rates. Higher Learning Rates would cause overshooting, whereas lower Learning Rates were causing slower convergence.

Determining the value for epochs was the toughest part. I needed epochs big enough to allow values to converge, yet small enough to avoid over-fitting. I started with a lower epochs value of 100 and the gradually increased it. At the end, I used the **epochs value of 15000**

The maximum validation Accuracy I was able to achieve was 85.65.

I have included the table below with the above-mentioned parameters and the Accuracies I got.

Table for accuracies with above-mentioned parameters

train_split	epochs	LR	Train Accuracy [%]	Validation Accuracy [%]
0.01	15000	0.006	85.65	90.02

Part D. (+2 bonus points)

I went over both, the C and the python code and took some help from chat gpt to determine the conclusion of this question.

Although, Both C and Python have their own advantages, I feel, Python might be the better Programming Language primarily because it is much easier to use as compared to C.

Also, Python has a rich ecosystem in terms of the wide range of libraries for data manipulation and analysis. You used the Tensorflow library in your python code which after I did some research found that it is primarily used for building and training neural networks.

Use of Dynamic memory allocation

Below I have included the answer to why dynamic memory allocation was required. I answered the question in the mymodel.c file as comments and have include the code snippet below as well to make your life easier.

NOTE : The answer to this question was generated by Chat gpt. However, I understood the concept and the words used are mine.

Listing 2: A small part of mymodel.c

```
1
2
3  double cost_train = sum_squared_diff / num_train;
4
5          // QQQQQQQQQQQQQQQQQQQQQQQQQQQQ
6          /// Explain here why I needed to dynamically
               allocate the memory
7          /* Ans: Dynamic memory allocation is used to create
               arrays a2_val, a3_val, and a4_val. This approach
               is necessary because these arrays
8           have sizes determined at runtime, and their sizes
               are not known at compile time. If the size of
               array was large and we had not used
9           dynamic memory allocation, it could have resulted
               is Stack overflow. Dynamic memory allocation
               allocates memory from the heap,
10          which typically has a larger memory space available
                compared to the stack.*/
11         double(*a2_val)[num_val] = malloc(num_neurons_layer2
               * sizeof(double[num_val]));
12         double(*a3_val)[num_val] = malloc(num_neurons_layer3
               * sizeof(double[num_val]));
13         double(*a4_val)[num_val] = malloc(num_outputs *
               sizeof(double[num_val]));
14
15         ForwardPass(num_val, X_val, Y_val,
16                     W2, W3, W4,
17                     b2, b3, b4,
18                     a2_val, a3_val, a4_val);
19
20                     // Rest of the code
```

Makefile

I have included the Makefile in the submission. use the command '**make**' to build an executable file name '**ANN_model**.' After that, you can use the '**./ANN_model**' command to execute the object code. Use **make clean** command removes all compiled files and artifacts generated by the Makefile, leaving the directory in a clean state.

Acknowledgments

A couple of commands in this latex file were generated by Chat gpt. The commands for colors of sections and subsections along with the code box for C code were generated by chat gpt.