Developing an Artificial Neural Network in C

MECHTRON2MP3 - Programming for Mechatronics

Ethan Otteson and Andrew De Rango

1 Introduction

The following report outlines the development of an artificial neural network (ANN) in C by Ethan Otteson and Andrew De Rango for a bonus assignment in the McMaster 2MP3 Programming for Mechatronics course. This C program will train an ANN based on a user-provided dataset including many examples of the independent variables and their effects on the dependent variables. More specifically, the choice of model that is employed in this program is the feedforward neural network (FNN), in which neuron activations are computed strictly uni-directionally. Some of these examples will be used to train the model through backpropagation; the rest will be used to validate the model's accuracy and ensure there is no overfit. After the model has been trained the user is asked if they would like to save the model to a separate file for future use. Additionally, this program has the flexibility to allow the user to choose the model's architecture, allowing for any number of inputs, outputs, hidden layers, and neurons in each hidden layer. The rest of this report will include how the program functions, the challenges we faced and overcame to develop the program, and our final results and testing.

2 Program Structure and Function

2.1 File Breakdown

The code can be broken down into four files:

- `Makefile`: Defines rules to be employed upon compilation.
- `main.c`: Defines the main function that calls functions from `mymodel.c`.
- `functions.c`: Defines all support functions for data handling, model training, and performance evaluation.
- `mymodel.h`: Contains function prototypes that are formally defined in `mymodel.c`.

2.2 Function Breakdown

`main` is where the user's execution command is extracted, and where all functions are called. It starts by ensuring that the user's execution command follows proper formatting, then assigns their inputs to variables to be passed into future function calls. It prints the network's architecture, then asks for the file name of the input data. Then, it calls `OrganizeData` and `Evaluation`.

`Sigmoid` returns a double of the sigmoid function when `x` is plugged in. It follows the formula $f(x) = \frac{1}{1+e^{-x}}$, where `x` is the only input.

`RandomDouble` returns a random double between the two parameters `min` and `max` via the `libsodium` library. Includes `libsodium` initialization failure warnings.

`ReadFile` reads the file specified by the user. It iterates through the file twice. It first counts the number of rows, then uses this value to dynamically allocate memory for `**data`, a 2D array that stores the data in the input file. It then reads through the file again and fills the array with the file data now that the array is declared. The function returns both this `**data` array and the number of rows in the file.

`OrganizeData` first shuffles every data point that is passed in via `**data`. It then computes how many training and validation datapoints there should be, based on the train/validation split determined by the user upon execution. It then assigns the proper number of datapoints to the training and validation datasets.

`InitializeArrays` simply iterates through the weights, biases, and activations arrays and allocates the necessary memory for each element within each array. For the weights and biases, it assigns a different random double between (`-initial_range`, `initial_range`) to each of them.

`CalculateMetrics` computes the metrics that are printed every hundredth epoch during the model's training so that the user can get a gauge on the ANN's performance throughout training. This includes a variation of MSE and accuracy. To do this, it creates alterative activation 3D arrays and runs `ForwardPass` for both the training and validation datasets to assign activations, and further, predictions for the model on each dataset. For more information, read the dynamic memory allocation section which delves deeper into the memory handling and logistics of this function.

`ForwardPass` assigns values for each training example to each neuron in each layer via the `***a` array. The three dimensions arise from `a[layer][neuron][training_example]`. The activations for the first layer are simply the inputs from the data file. The activations of each of the following layers must be computed sequentially via a linear combination of the previous layer's activation and the weights, bias, and activation function associated with the neuron in question, hence the function name "ForwardPass" and the type of model "feedforward neural network".

The `BackwardPass` function deals with the backpropagation algorithm responsible for optimizing the weights and biases selected by the program to best predict the label values. The weights and biases are updated in each epoch, a change dependent on their influences on the ultimate predictions, and by extension, the cost function. It starts by allocating memory in all three dimensions for `PL`, which stores the partial derivative of the cost function with respect to the activations of the neurons. This process is begun by taking the derivative of the cost function with respect to the activations of the neurons in the output layer, which is simply the derivative of the activation function. The activation function is the sigmoid function and was computed manually to be in the form seen in the double `for` loop. Then, the function enters a loop for each hidden layer and computes `PL` for each iteration. Since `PL` is a partial derivative, we can split it into the intermediate partial derivatives via the chain rule, and calculate it as the derivative of the cost function, computed manually, multiplied by the weighted sum of the gradients of the neurons in the following layer of the network. Hence the function name "BackwardPass" and the name of the algorithm "backpropogation". Using the gradients computed in the previous loop and the learning rate specified by the user, we update the weight of each synapse and bias of each neuron.

The `Evaluation` function calls `InitializeArrays` then loops through the epochs of training, calling `ForwardPass` and `BackwardPass` each iteration. If the iteration is a hundredth epoch, then `CalculateMetrics` is called to show the performance. Following training, `DownloadANN` is called.

`DownloadANN` is the function responsible for downloading or saving the network that was just trained. It first asks the user if they would like to save the network, then creates a file called Downloaded Data to their current directory if they say yes and the directory does not already exist. It creates the file `ANN_data.txt` in this folder then writes the model's architecture, performance metrics, and weights and biases into it.

3 Running the Program

3.1 Makefile Preamble

After downloading the program, the user must first edit the Makefile. In order to accommodate for the inclusion of the `sodium.h` library in the program, the Makefile must be altered depending on the machine's operating system. The user must ensure that the Makefile only includes the proper sections of the Makefile that correspond to the operating system that they're compiling the code on. This is further explained in the Makefile section of this report.

3.2 Execution

After this alteration, the user can compile the program using the `make` command. This will create one Unix executable file in the same directory titled `ANN`. `ANN` can be executed using the following execution format:

```
./ANN <epochs> <learning_rate> <train_split> <num_inputs> <num_neurons_layer2>
... <num_neurons_layerN> <num_outputs>
```

For example, to train an FNN using 100,000 epochs, 0.0001 learning rate, 10% of the data used for training, and an architecture of 2-64-32-18-2, then run the following command:

```
./ANN 100000 0.0001 0.1 2 64 32 18 2
```

3.3 Data File Input

The user will then be prompted to enter the name of the file in which the data is stored. Two preexisting data files in the proper format are included in the repository: `data.txt` and `data33.txt`. `data.txt` contains two labels and features, and `data33.txt` contains three of each. The data file must be in the same directory that the program is being compiled from in order for the program to find the data file. After the model is trained, the program will prompt the user if they would like to download the model. This will return a file called `ANN_data.txt` in a created folder. The program writes the model's architecture and training parameters to the file, as well as each neuron's bias and each neural connection's weight. An example of this can be seen in the `ANN_data.txt` file currently in the repo.

3.4 Dependencies

To run this program, the user must have the `libsodium` library installed. Documentation and installation instructions can be found here: https://libsodium.gitbook.io/doc/.

3.4.1 `libsodium` Installation on WSL

To install `libsodium` on Windows Subsystem for Linux (WSL), follow these instructions:

1. Open your WSL terminal.
2. Update the package list:

   ```
   sudo apt update
   ```

3. Install `libsodium`:

   ```
   sudo apt install libsodium-dev
   ```

3.4.2 `libsodium` Installation on macOS

To install `libsodium` on macOS, you can use Homebrew. If Homebrew is not installed, you can install it by following the instructions on the Homebrew website: https://brew.sh/.

1. Open your terminal.
2. Install `libsodium` using Homebrew:

   ```
   brew install libsodium
   ```

After completing these steps, `libsodium` should be installed on the user's system, and they may proceed to compile the program.

4 Dynamic Memory Allocation

4.1 Implementation

In order to allow the user to adjust the quantity of features and labels exist within the input data, as well as the quantity of hidden layers and the number of neurons within these layers, we must implement some sort of dynamic memory allocation. Additionally, we implemented dynamic memory allocation to handle the input data because the size of the data is not known upon the time of compilation. The 2D arrays for X and Y training and validation samples are all dynamically allocated memory then passed into the `OrganizeData` function to be assigned actual values and then into the `Evaluation` function for training and evaluating the model. In the `Evaluation` and `InitializeArrays` functions, the 3D array `***W`, which represents the weights of each connection for each neuron in each layer has its memory dynamically allocated in a triple `for` loop (because it's 3D). `**b` and `**a` follow similar processes, but are 2D because there exists only one value for every neuron within every layer. All of `***W`, `**b`, and `**a` need to use `malloc` because none of the sizes for any of their dimensions are known upon compile time; it is dependent on the input file `data.txt` or whatever the user enters. When the model is being trained, the program will print the performance of the model at every hundredth epoch. In order to do this, we must evaluate the model's performance metrics for both the training and validation datasets. This requires the dynamic memory allocation of two different 3D arrays for the neuron's activations. These arrays, `***a_train` and `***a_val`, are given values directly depending on the network's weights and biases, as well as the feature neurons given from `X_train` and `X_val`. It needs to be 3D because it stores the activation of every neuron within every layer for every example within the training or validation datasets, respectively. This assignation is done in `ForwardPass`. After being dynamically allocated memory, `output_neurons_train` and `output_neurons_val` hold a transposed version of just the last layer's activations, which represent the network's predictions for the train and validation datasets, respectively. This transposition needs to occur to align with the dimensions that actual labels are stored in `Y_train`. We also use dynamic memory allocation when storing the gradient vectors of the cost function with respect to neural activations in `BackwardPass`. This requires dynamic memory allocation in three dimensions because the partial derivative is computed for every neuron within every layer for each training example. `malloc` is required here because we do not know the size of any of these dimensions upon compilation; they are all dependent on the user's execution command and the data file's size.

4.2 Assignment Questions

The importance of this dynamic memory allocation can be illustrated with a simple example from an earlier version of the code. Figure 1 shows an earlier version of the code before the implementation of dynamic memory allocation. If the training split was 10% of the data or higher the program would experience a segmentation fault and crash. Using GDB to analyze the program, it is determined that the error is from the declaration of the `a3_squared_complement` variable. Taking a look at the exact dimensions of this two-dimensional array it is larger than the maximum memory allocated by stack and will lead to stack overflow. Interestingly, several lines above the line where this segmentation fault occurs there is another variable, `PL2`, which has the same dimensions and size as the `a3_squared_complement` variable. The reason that this does not cause a segmentation fault earlier is that the program attempts to access the `a3_squared_complement` variable before the `PL2` variable. This earlier version of the code which was previously mentioned did require some dynamic memory allocation. Dynamic memory allocation was required to allocate memory for the activation values needed to check the validation cost and accuracy. This was for the same reason that the program would run into a segmentation fault when the training split was greater than 10%, as when the training split is less than 10% that means that over 90% of the data is being used to validate the model. If 10% of the data would cause a segmentation fault as the activation values took up more space than available in the stack then more than 90% would certainly do the same which is why dynamic memory allocation was required even in the early iterations of the program.

4.3 Memory Leak

Memory leaks can be an issue when a program uses dynamic memory allocation. If the memory allocated is not freed before the program completes, the pointer that points to that address in memory will be lost and the computer will be unable to access the more to free it and will not automatically free the memory as it was dynamically allocated by the user. The only way to remove this data from memory is to restart the computer and wipe the random access memory, and if the computer is not restarted in a relatively long time then the memory will fill and the computer or program will crash as it no longer has the available required memory. For these reasons, it is very important that all allocated memory is freed.

After completing the program and implementing dynamic memory allocation for any variable that may need it, we began to train different models to find the most optimal architecture for this dataset. However, when training larger models after many iterations of the training the program would crash and the terminal would output "killed." Initially, we assumed this was just because we were training the model on an old laptop and coincidently a Windows update was happening in the background at the time of the crash and we assumed it must have just taken too many system resources away from the WSL virtual machine. However, after this problem persisted and seemed to consistently stop around the same number of training iterations we further investigated and found that the system memory was filling up just before the crash. At this point, we assumed it was a memory leak and used Intel VTune to find it. Using a simple ANN architecture and a training split of 1%, Figure 3 showed that the `BackwardPass` and `ReadFile` functions were not completely deallocating their memory. Figure 4 in the `BackwardPass` function shows which line was not properly being deallocated at the end, this was because the if statement which included the output layer was not included in the for loops which freed the memory, while Figure 5 shows the line of code which we completely forgot to deallocate. After fixing these deallocation mistakes Intel VTune was rerun and the results (Figure 6) show that there is no longer a difference between the amount of memory allocated and deallocated which also fixed the problem with the terminal outputting "killed."

5 Results

Once we were done developing the program we pivoted to trying to find the most optimal ANN architecture for this dataset. We began with the sensitivity analysis in the following three tables which were required for the assignment and continued with our own analysis methodology found in the appendix.

Table 1: Sensitivity Analysis I
Parameters of `epochs = 100000`, `train_split = 0.003`, `*num_neurons = [40,20]`

Learning Rate	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy
0.0005	0.435271	0.475252	65.52%	57.25%
0.001	0.194706	0.317943	86.90%	77.20%
0.005	0.019087	0.383050	97.93%	77.42%
0.01	0.004098	0.425162	99.31%	75.80%
0.1	0.047839	0.455859	95.86%	74.06%

Table 2: Sensitivity Analysis II
Parameters of `learning_rate = 0.005`, `train_split = 0.003`, `*num_neurons = [40,20]`

epochs	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy
100	0.478894	0.509057	58.62%	49.97%
1000	0.463212	0.489889	59.31%	54.52%
10000	0.438632	0.476145	58.62%	56.15%
100000	0.019087	0.383050	97.93%	77.42%

Table 3: Sensitivity Analysis III
Parameters of `epochs = 100000`, `learning_rate = 0.005`, `*num_neurons = [40,20]`

training_split	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy
0.0003	0.000009	0.995018	100.00%	49.98%
0.003	0.016038	0.430028	98.62%	74.59%
0.005	0.027844	0.370659	98.34%	78.91%
0.01	0.059593	0.296477	96.27%	82.04%
0.1	N/A	N/A	N/A	N/A
0.4	N/A	N/A	N/A	N/A
0.6	N/A	N/A	N/A	N/A
0.8	N/A	N/A	N/A	N/A

Looking at the results of these tables there are a few key takeaways. The learning rate must be balanced to ensure quick convergence to a solution without jumping over the optimal solution or causing overfitting. It is typically better to have a larger number of epochs, however, too many are redundant and will lead to insignificant improvement while taking much longer to train. With training split more is typically better as it results in less overfitting but this also has the effect of slowing convergence to an optimal solution as there are more variables to optimize.

The reason that most of Table 3 is blank is because the cost goes off to infinity and the accuracy to zero in those situations. After some research, we determined this to likely be because of the vanishing gradient problem. The basic idea is that during backpropagation the gradient which decides how much each of the variables should be changed vanishes and goes to zero. This problem becomes more common as the size of the ANN increases and there are more variables to work with. A common way to avoid this problem is to decrease the learning rate which is what is done in Table 4, which still tests the increasing training split percentages while making sure we achieve a result.

Table 4: Sensitivity Analysis IV
Parameters of `epochs = 100000`, `learning_rate = 0.001`, `*num_neurons = [40,20]`

learning_rate	training_split	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy
0.001	0.1	0.149489	0.176254	87.20%	87.51%
0.0001	0.4	0.157991	0.166107	88.36%	87.97%
0.0001	0.6	0.180729	0.178317	87.34%	87.42%
0.0001	0.8	0.477275	0.478851	57.12%	56.79%

After this set of testing we did much more, the most successful of which are included in the appendix. Table 5 shows the testing results for modifying the number of neurons in the hidden layers. With this testing suite, we discovered that an architecture of 2 inputs, 2 outputs, 10 and 20 neurons in the hidden layers to be most effective for this dataset. This structure achieved slightly better levels of accuracy than others while being quicker to train and having fewer signs of overfitting.

Once we decided on this architecture we moved on to the results in Table 6 which are random tests to try and obtain the maximum possible validation accuracy. The best of which uses epochs 100000, a learning rate of 0.0001, a training split of 40%, 10 neurons in the first hidden layer, and 20 in the second hidden layer. This achieved results which can be seen in Table 6 but most importantly a validation accuracy of 88.15% and training accuracy of 88.13%. This was the best result we were able to produce and shows zero signs of overfitting. Given the limited timeline for this project, we were only able to conduct so many tests and consider that many of them take well over an hour on our computers. We are sure that there are better results possible but we were not able to find them.

Finally, Intel VTune was used to assess which parts of the program were taking the longest time to execute. These results can be seen in Figure 7, which demonstrates that the function which takes the longest time is `BackwardPass` and hyperbolic tangent. The former is not surprising as it is a very complicated process. It was interesting that hyperbolic tangent was taking up so much of the processor time however when considering how many times it is called during ForwardPass this is more understandable. Future optimization of this program may see the substitution of the hyperbolic tangent function for one that is less computationally demanding. Additionally, in these results, it can be seen in the histogram that this program is not currently suitable for parallel computing. That may be an interesting addition to this project in the future which would require the replacement of the numerous if statements.

6 Model and Program Flexibility

The bonuses awarded in this assignment revolved around permitting the user to customize the input data that was handled and used to train the ANN, and the architecture of the ANN itself.

The first was allowing the user to customize the amount of input and output variables, otherwise known as features and labels. The implementation of custom feature and label quantities increases the flexibility and robustness of the program. When the user enters these numbers into the execution command, they are passed to the function `ReadFile` so that the program knows how many columns are in the file, which simplifies the reading process. The number of inputs and outputs helps the program know how much memory to allocate for the `X_train` and `X_val` arrays, and the `Y_train` and `Y_val` arrays, respectively. Aside from assigning testing and validation input and output arrays, these variables are also used to determine the number of weights that exist in the first hidden layer, because the number of weights in a layer is dependent on both the number of neurons within that layer and the number of neurons in the previous layer. For the first hidden layer, the number of neurons in the previous layer is simply the number of features. Similarly, the number of outputs is the number of neurons in the last layer of the network, which helps determine how many weights there should be for the last layer. This same logic, of using input and output neuron quantity to determine how many neurons are in the current or previous layers in `for` loops is used again in the `ForwardPass` function and the `BackwardPass` functions.

As for allowing the user to change the number of hidden layers, we permitted the user to relay this information to the program directly from the execution line, where they enter the quantity of neurons in each hidden layer between the number of features and labels. `*num_neurons` is the array that stores the number of neurons in each sequential hidden layer, and uses `malloc` because we don't know how many layers the user wants until they execute the program. `*num_neurons` is often used in the dynamic memory allocation of various pointers because their size depends on the number of neurons in a specific layer. For example, the size of all `***W`, `**b`, and `**a` depend on `*num_neurons` because they each have at least one value for each neuron in each layer. Instead of having a different variable for weight, bias, and activation for each layer, which is used in `ForwardPass`, `BackwardPass`, and `Evaluation`, we can have one variable for each that we dynamically allocate memory for, because it is not known how many layers or neurons the user will want to use when they compile the program.

7 Makefile

7.1 Cross-Platform Makefile Configuration

`sodium.h` is a library that was included in this project to generate random numbers. This had two use cases: to generate initial values for weights and biases for each neuron, and to help classify rows as training or validation. The inclusion of the `sodium.h` library can render some complications in developing the Makefile for this project, especially because one of us is operating on macOS and the other uses WSL. Thus, the solution that we decided on was to include one Makefile with two configurations; one for each of the environments. The principal reason that underpins the necessity for the two configurations lies in the differences in the ways that WSL and macOS build and link the program. To install libsodium on macOS, you must use Homebrew, which differs from the WSL installation process of libsodium. For this reason, we must define where libsodium is installed on the macOS configuration otherwise the program will not compile. Below is the Makfile with the WSL configuration active and the macOS configuration commented out.

```
######### WSL Makefile #########

CC = gcc
CFLAGS = -Wall -Wextra -g -O3
LDFLAGS = -lm -lsodium
TARGET = ANN
SRC = main.c mymodel.c
OBJ = $(SRC:.c=.o)

$(TARGET): $(OBJ)
$(CC) $(CFLAGS) -o $(TARGET) $(OBJ) $(LDFLAGS)
@rm -f $(OBJ)

%.o: %.c
$(CC) $(CFLAGS) -c $<

clean:
rm -f $(TARGET)

.PHONY: clean

######## MacOS Makefile #########
```

```
# CC = gcc
# CFLAGS = -Wall -Wextra -g -O3 -I/opt/homebrew/Cellar/libsodium/1.0.19/include
# LDFLAGS = -L/opt/homebrew/Cellar/libsodium/1.0.19/lib -lm -lsodium
# TARGET = ANN
# SRC = main.c mymodel.c
# OBJ = $(SRC:.c=.o)

# $(TARGET): $(OBJ)
#  $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) $(LDFLAGS)
#  @rm -f $(OBJ)

# %.o: %.c
#  $(CC) $(CFLAGS) -c $<

# clean:
#  rm -f $(TARGET)

# .PHONY: clean
```

7.2 Makefile Guide

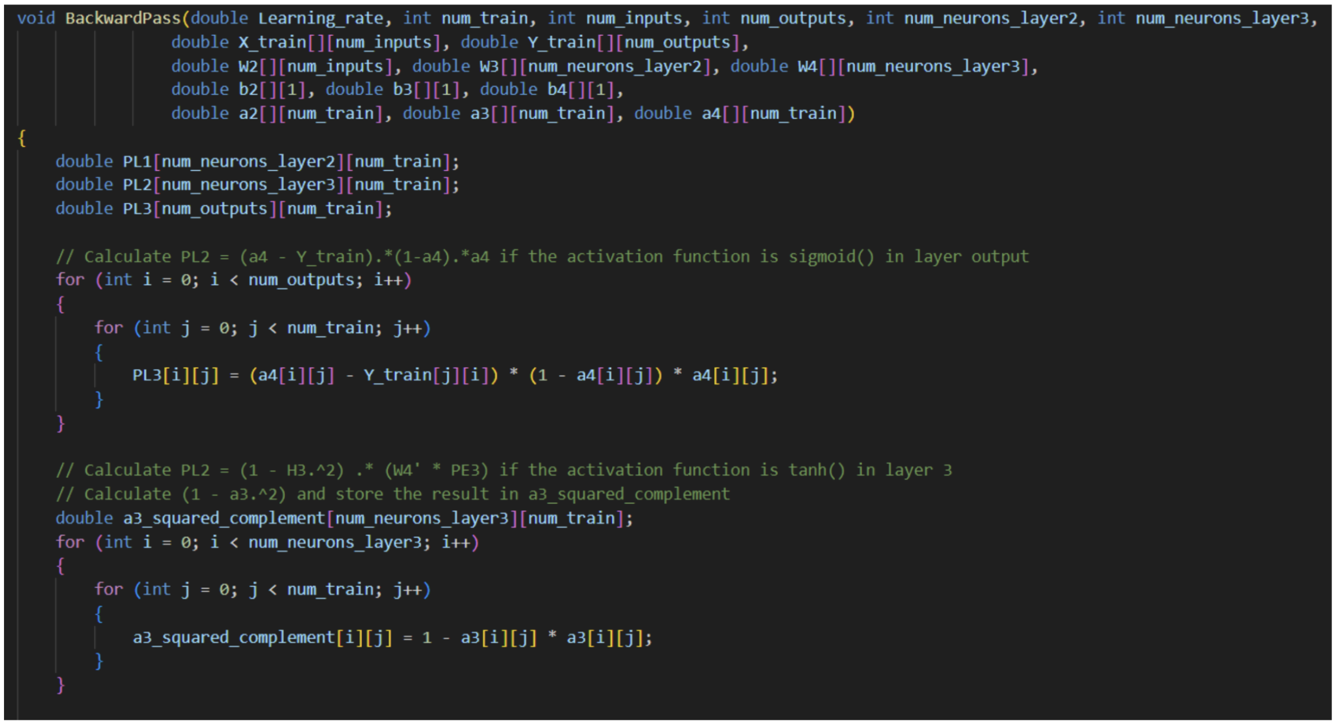
The purpose of this Makefile is to compile any changes made to any file within this project with a single command `make` in the terminal. It works as follows, line by line:

1. Sets the compiler to the GCC compiler.
2. A list of warning, tool, and optimization flags.
3. A list of library flags.
4. The target file which it should make executable.
5. The list of source code files on which the final executable depends.
6. A rule to convert the list of source code files into object files.
7. A rule on how to build the target executable from the object files.
8. The specific line that will be run in the terminal to compile the executable with the file dependencies and flags.

9. Removes all of the object files to only leave the executable.
10. Rule on how to make the object files from the source code.

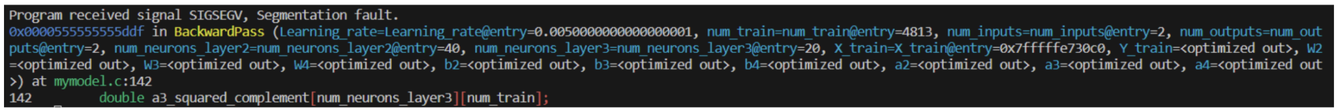
The last few lines allow for the `make clean` command to remove the executable file, and it also ensures that a file named `clean` doesn't already exist.

8 Appendix



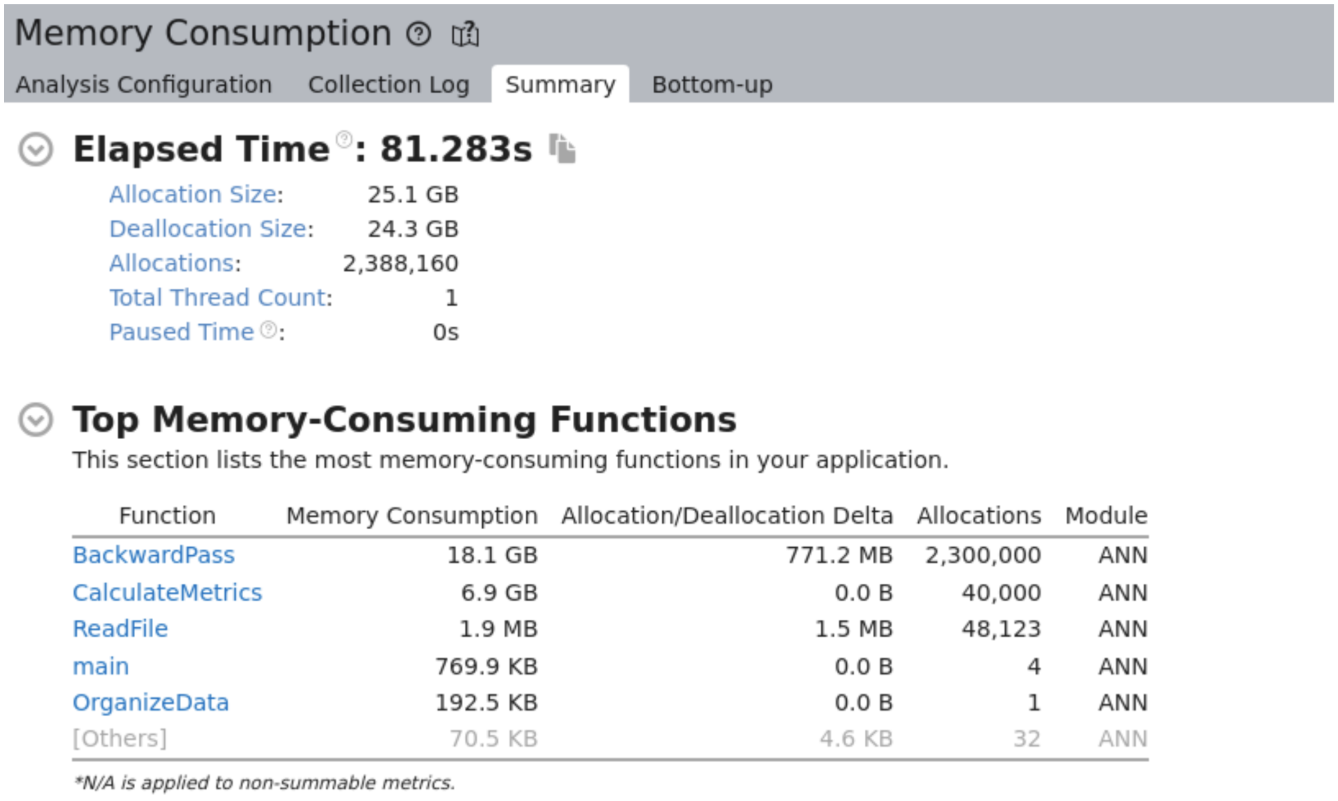
```c
void BackwardPass(double Learning_rate, int num_train, int num_inputs, int num_outputs, int num_neurons_layer2, int num_neurons_layer3,
                double X_train[][num_inputs], double Y_train[][num_outputs],
                double W2[][num_inputs], double W3[][num_neurons_layer2], double W4[][num_neurons_layer3],
                double b2[][1], double b3[][1], double b4[][1],
                double a2[][num_train], double a3[][num_train], double a4[][num_train])
{
    double PL1[num_neurons_layer2][num_train];
    double PL2[num_neurons_layer3][num_train];
    double PL3[num_outputs][num_train];

    // Calculate PL2 = (a4 - Y_train).*(1-a4).*a4 if the activation function is sigmoid() in layer output
    for (int i = 0; i < num_outputs; i++)
    {
        for (int j = 0; j < num_train; j++)
        {
            PL3[i][j] = (a4[i][j] - Y_train[j][i]) * (1 - a4[i][j]) * a4[i][j];
        }
    }

    // Calculate PL2 = (1 - H3.^2) .* (W4' * PE3) if the activation function is tanh() in layer 3
    // Calculate (1 - a3.^2) and store the result in a3_squared_complement
    double a3_squared_complement[num_neurons_layer3][num_train];
    for (int i = 0; i < num_neurons_layer3; i++)
    {
        for (int j = 0; j < num_train; j++)
        {
            a3_squared_complement[i][j] = 1 - a3[i][j] * a3[i][j];
        }
    }
```

Figure 1: Beginning of `BackwardPass` function for backpropagation.



```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555ddf in BackwardPass (Learning_rate=Learning_rate@entry=0.005000000000000001, num_train=num_train@entry=4813, num_inputs=num_inputs@entry=2, num_outputs=num_out
puts@entry=2, num_neurons_layer2=num_neurons_layer2@entry=40, num_neurons_layer3=num_neurons_layer3@entry=20, X_train=X_train@entry=0x7fffffe730c0, Y_train=<optimized out>, W2
=<optimized out>, W3=<optimized out>, W4=<optimized out>, b2=<optimized out>, b3=<optimized out>, b4=<optimized out>, a2=<optimized out>, a3=<optimized out>, a4=<optimized out
>) at mymodel.c:142
142         double a3_squared_complement[num_neurons_layer3][num_train];
```

Figure 2: Segmentation fault in the original program.



Elapsed Time: 81.283s

Allocation Size:	25.1 GB
Deallocation Size:	24.3 GB
Allocations:	2,388,160
Total Thread Count:	1
Paused Time:	0s

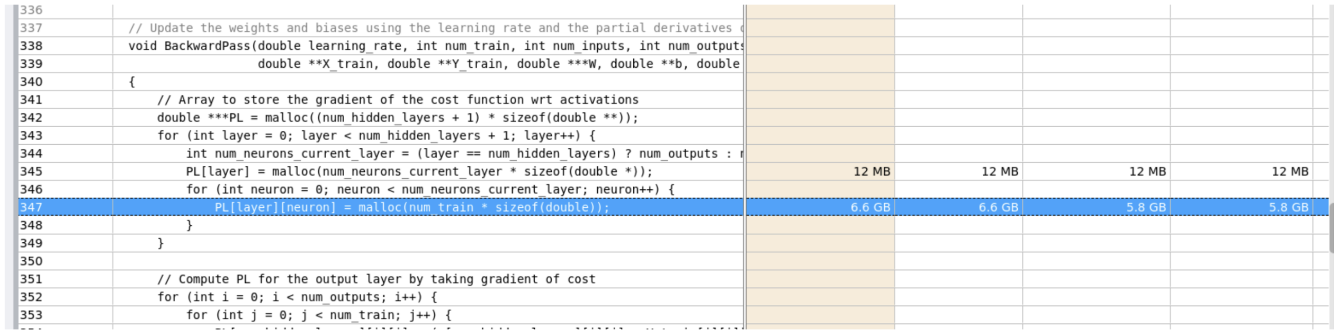
Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
BackwardPass	18.1 GB	771.2 MB	2,300,000	ANN
CalculateMetrics	6.9 GB	0.0 B	40,000	ANN
ReadFile	1.9 MB	1.5 MB	48,123	ANN
main	769.9 KB	0.0 B	4	ANN
OrganizeData	192.5 KB	0.0 B	1	ANN
[Others]	70.5 KB	4.6 KB	32	ANN

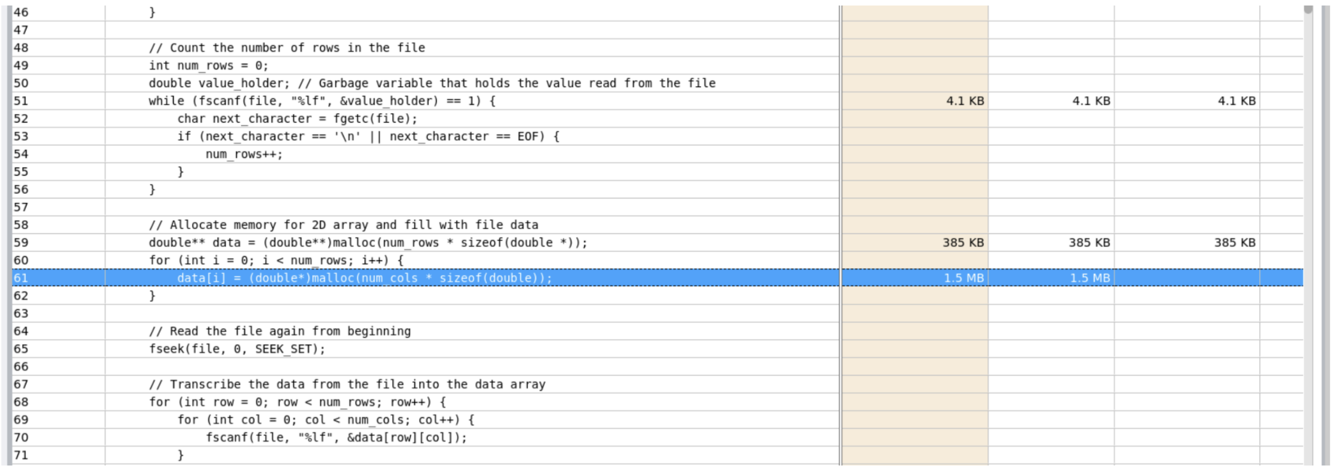
**N/A is applied to non-summable metrics.*

Figure 3: Memory consumption via VTune report, showing missing deallocations.



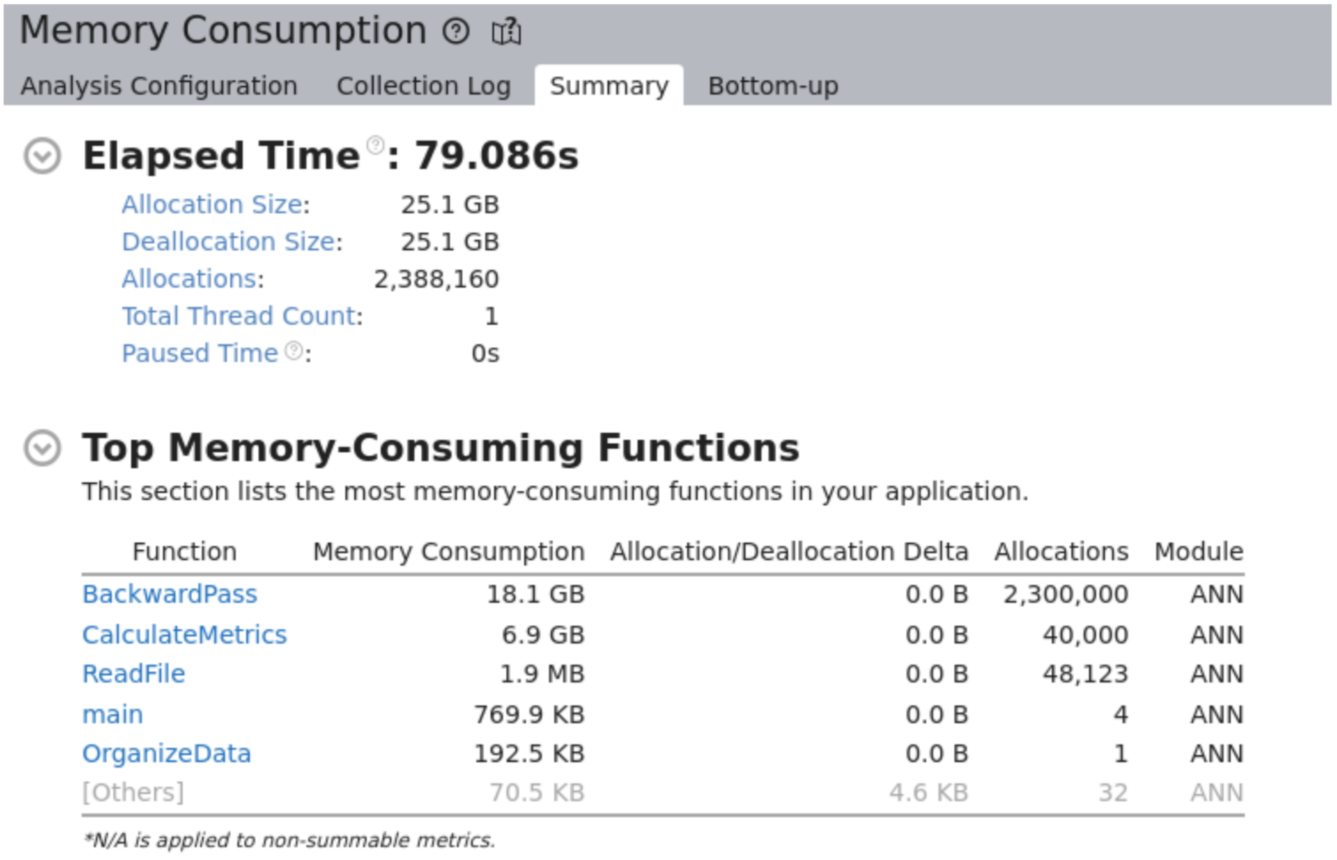
```

// Update the weights and biases using the learning rate and the partial derivatives
void BackwardPass(double learning_rate, int num_train, int num_inputs, int num_output
                  double **X_train, double **Y_train, double ***W, double **b, double
{
    // Array to store the gradient of the cost function wrt activations
    double ***PL = malloc((num_hidden_layers + 1) * sizeof(double **));
    for (int layer = 0; layer < num_hidden_layers + 1; layer++) {
        int num_neurons_current_layer = (layer == num_hidden_layers) ? num_outputs :
        PL[layer] = malloc(num_neurons_current_layer * sizeof(double *));            12 MB     12 MB     12 MB     12 MB
        for (int neuron = 0; neuron < num_neurons_current_layer; neuron++) {
            PL[layer][neuron] = malloc(num_train * sizeof(double));                  6.6 GB    6.6 GB    5.8 GB    5.8 GB
        }
    }

    // Compute PL for the output layer by taking gradient of cost
    for (int i = 0; i < num_outputs; i++) {
        for (int j = 0; j < num_train; j++) {
```

Figure 4: Part of the VTune memory consumption report showing which lines had memory allocated and not deallocated.



```
    }

    // Count the number of rows in the file
    int num_rows = 0;
    double value_holder; // Garbage variable that holds the value read from the file
    while (fscanf(file, "%lf", &value_holder) == 1) {                                    4.1 KB        4.1 KB        4.1 KB
        char next_character = fgetc(file);
        if (next_character == '\n' || next_character == EOF) {
            num_rows++;
        }
    }

    // Allocate memory for 2D array and fill with file data
    double** data = (double**)malloc(num_rows * sizeof(double *));                       385 KB        385 KB        385 KB
    for (int i = 0; i < num_rows; i++) {
        data[i] = (double*)malloc(num_cols * sizeof(double));                            1.5 MB        1.5 MB
    }

    // Read the file again from beginning
    fseek(file, 0, SEEK_SET);

    // Transcribe the data from the file into the data array
    for (int row = 0; row < num_rows; row++) {
        for (int col = 0; col < num_cols; col++) {
            fscanf(file, "%lf", &data[row][col]);
        }
```

Figure 5: Part of the VTune memory consumption report showing which lines had memory allocated and not deallocated.



Elapsed Time: 79.086s

Allocation Size:	25.1 GB
Deallocation Size:	25.1 GB
Allocations:	2,388,160
Total Thread Count:	1
Paused Time:	0s

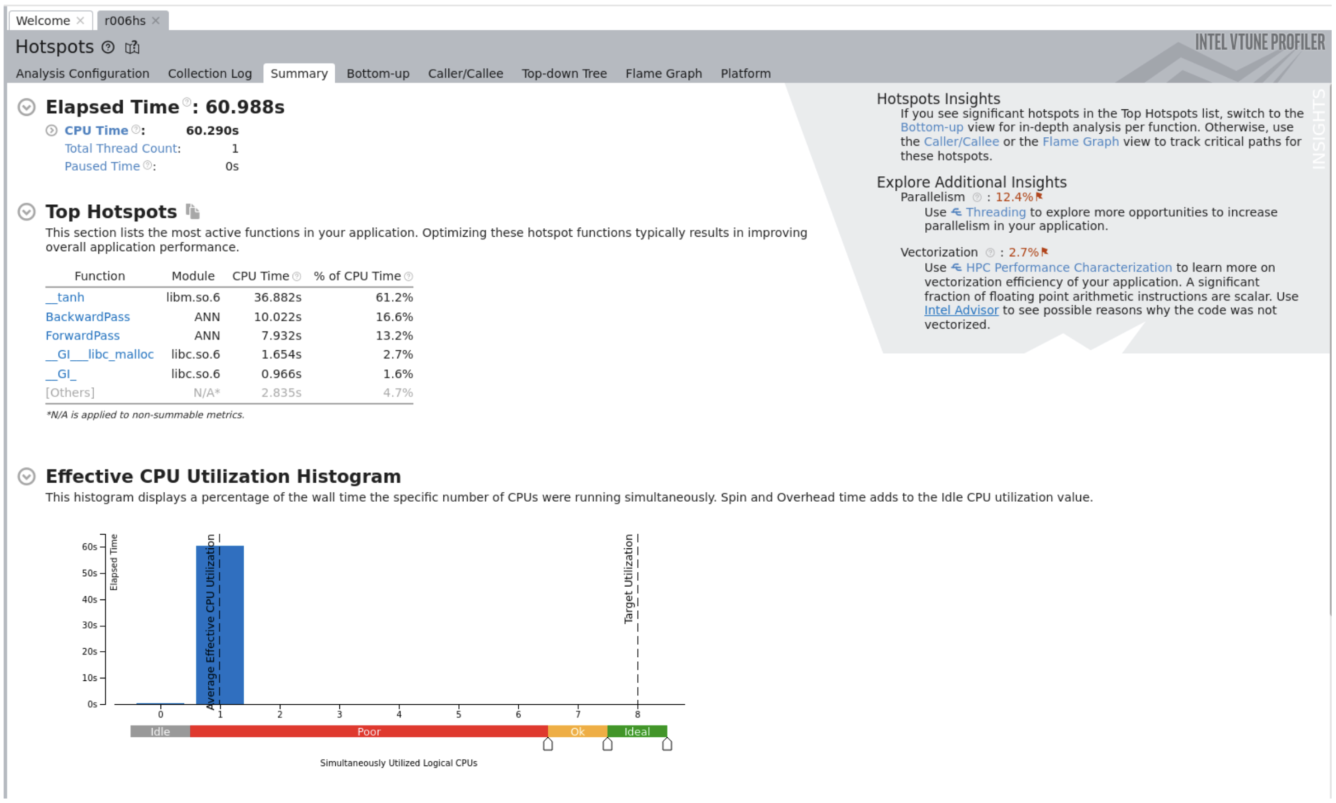
Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
BackwardPass	18.1 GB	0.0 B	2,300,000	ANN
CalculateMetrics	6.9 GB	0.0 B	40,000	ANN
ReadFile	1.9 MB	0.0 B	48,123	ANN
main	769.9 KB	0.0 B	4	ANN
OrganizeData	192.5 KB	0.0 B	1	ANN
[Others]	70.5 KB	4.6 KB	32	ANN

*N/A is applied to non-summable metrics.

Figure 6: Final VTune memory consumption report after fixing previously mentioned memory leaks.



Elapsed Time: 60.988s

CPU Time: 60.290s
Total Thread Count: 1
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
__tanh	libm.so.6	36.882s	61.2%
BackwardPass	ANN	10.022s	16.6%
ForwardPass	ANN	7.932s	13.2%
__GI___libc_malloc	libc.so.6	1.654s	2.7%
__GI_	libc.so.6	0.966s	1.6%
[Others]	N/A*	2.835s	4.7%

*N/A is applied to non-summable metrics.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.

Explore Additional Insights
Parallelism : 12.4%
Use Threading to explore more opportunities to increase parallelism in your application.

Vectorization : 2.7%
Use HPC Performance Characterization to learn more on vectorization efficiency of your application. A significant fraction of floating point arithmetic instructions are scalar. Use Intel Advisor to see possible reasons why the code was not vectorized.

Figure 7: Final VTune hotspots report.

Table 5: Sensitivity Analysis V: Different ANN Architectures
Parameters of `epochs = 100000`, `learning_rate= 0.005`, `train_split = 0.01`

Neurons Hidden Layer 1	Neurons Hidden Layer 1	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy
2	20	0.407421	0.465637	62.03%	57.93%
3	20	0.292101	0.364358	77.80%	72.37%
4	20	0.321393	0.344703	79.25%	77.16%
5	20	0.151973	0.268352	91.29%	81.88%
10	20	0.053372	0.300373	97.10%	81.93%
15	20	0.064048	0.3441	95.44%	80.28%
20	20	0.100649	0.282457	93.57%	82.70%
25	20	0.052072	0.311167	96.68%	81.70%
30	20	0.049486	0.312182	96.89%	82.12%
35	20	0.063069	0.272322	95.44%	83.56%
45	20	0.055373	0.311556	96.47%	81.37%
50	20	0.06193	0.267009	96.27%	84.14%
5	0	0.394584	0.400441	63.69%	63.45%
10	0	0.289959	0.335988	76.35%	72.75%
35	0	0.153907	0.237319	90.66%	84.70%
5	2	0.375306	0.386314	62.66%	62.88%
10	2	0.330036	0.363275	71.37%	68.76%
35	2	0.36471	0.390533	64.94%	63.28%
5	3	0.362012	0.385948	67.84%	66.26%
10	3	0.333068	0.369153	69.09%	66.79%
35	3	0.277782	0.341809	76.35%	72.24%
5	4	0.330507	0.362992	70.95%	69.37%
10	4	0.187891	0.289654	84.23%	78.13%
35	4	0.15889	0.281107	88.80%	81.43%
5	5	0.261181	0.329339	77.39%	72.29%
10	5	0.132656	0.267731	92.32%	82.73%
5	10	0.182285	0.280221	88.38%	81.42%
10	10	0.090766	0.261444	93.78%	83.25%
35	10	0.081028	0.269816	95.02%	83.56%
5	15	0.138072	0.246435	89.63%	83.17%
10	15	0.117379	0.286034	93.16%	82.79%

Table 6: Final Testing to Find the Optimal Solution

Epochs	Learning Rate	Train Split	Neurons HL1	Neurons HL2	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy
94500	0.005	0.1	5	4	0.36265	0.378516	74.19%	73.04%
94500	0.001	0.1	5	5	0.244735	0.253547	83.05%	82.21%
90000	0.001	0.1	5	10	0.206053	0.213597	86.22%	85.82%
90000	0.001	0.1	5	20	0.23933	0.252313	82.44%	81.50%
90000	0.001	0.1	10	20	0.146739	0.182867	89.69%	87.01%
50000	0.001	0.1	10	0	0.298077	0.402267	67.98%	67.14%
50000	0.002	0.1	10	20	0.478375	0.47885	56.99%	57.25%
50000	0.001	0.1	12	20	0.149538	0.179154	89.09%	87.06%
100000	0.0001	0.4	10	20	0.16096	0.161401	88.13%	88.15%
100000	0.01	0.003	10	20	0.036897	0.507647	96.55%	70.94%
161800	0.0001	0.5	5	5	0.251531	0.252387	82.36%	82.21%
100000	0.000001	0.8	10	20	0.476256	0.476976	57.36%	57.12%
57300	0.0001	0.4	10	20	0.161106	0.161515	88.01%	88.13%
100000	0.0001	0.4	20	10	0.16056	0.164782	88.14%	87.77%
50000	0.0001	0.5	10	20	0.162568	0.164397	88.14%	88.08%