CS 61C Fall 2021

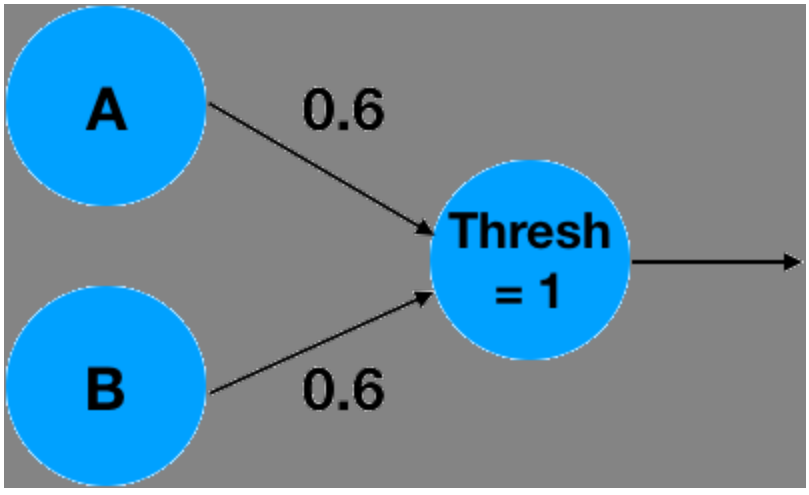
# Part B: File Operations and Classification

In this part, you will implement file operations to read pictures of handwritten digits. Then you will use your math functions from the previous part to determine what digit is in the picture.

If you are curious how the machine learning algorithm works, you can expand the Neural Networks section below. This is optional and not required to finish the project.

▼ Optional: Neural Networks

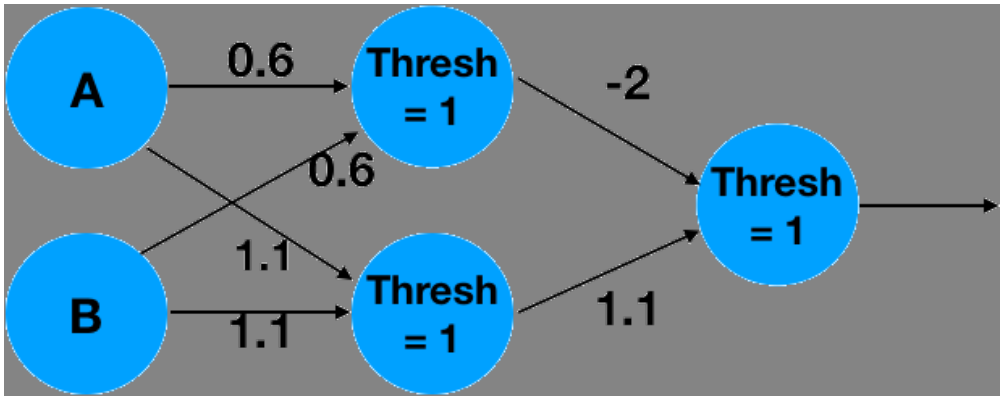
At a basic level, a neural networks tries to approximate a (non-linear) function that maps your input into a desired output. A basic neuron consists of a weighted linear combination of the input, followed by a non-linearity -- for example, a threshold. Consider the following neuron, which implements the logical `AND` operation:




It is easy to see that for A=0, B=0, the linear combination 0*0.6 + 0*0.6 = 0, which is less than the threshold of 1 and will result in a 0 output. With an input A=0, B=1 or A=1, B=0 the linear combination will results in 1*0.6* + *0*0.6 = 0.6, which is less than 1 and result in a 0 output. Similarly, A=1, B=1 will result in 1*0.6+1*0.6=1.2, which is greater than the threshold and will result in a 1 output! What is interesting is that the simple neuron operation can also be described

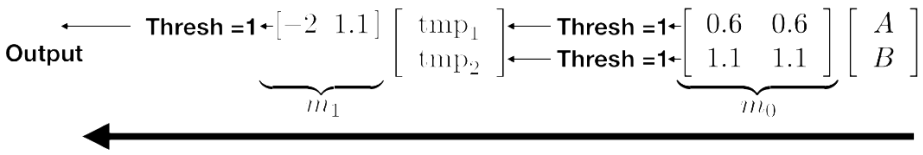
as an inner product between the vector [A,B]^T and the weights vector [0.6,0.6]^T followed by as thresholding, non-linear operation.

More complex functions can not be described by a simple neuron alone. We can extend the system into a network of neurons, in order to approximate the complex functions. For example, the following 2 layer network approximates the logical function `XOR`:




The above is a 2 layer network. The network takes 2 inputs, computes 2 intermediate values, and finally computes a single final output.

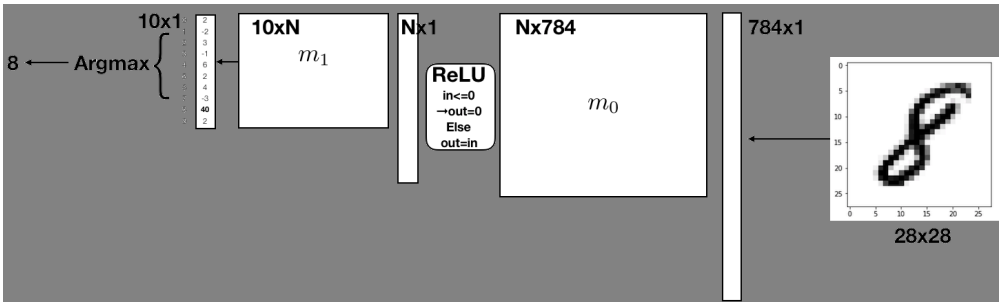
It can be written as matrix multiplications with matrices `m_0` and `m_1` with thresholding operations in between as shown below:




Convince yourself that this implements an `XOR` for the appropriate inputs!

You are probably wondering how the weights of the network were determined? This is beyond the scope of this project, and we would encourage you to take advanced classes in numerical linear algebra, signal processing, machine learning and optimization. We will only say that the weights can be trained by giving the network pairs of correct inputs and outputs and changing the weights such that the error between the outputs of the network and the correct outputs is minimized. Learning the weights is called: "Training". Using the weights on inputs is called "Inference". We will only perform inference, and you will be given weights that were pre-trained by your dedicated TA's.

In this project we will implement a similar, but slightly more complex network which is able to classify handwritten digits. As inputs, we will use the MNIST data set, which is a dataset of 60,000 28x28 images containing handwritten digits ranging from 0-9. We will treat these images as "flattened" input vectors of size 784 (`= 28 * 28`). In a similar way to the example before, we will perform matrix multiplications with pre-trained weight matrices `m_0` and `m_1`. Instead of thresholding we will use two different non-linearities: The `ReLU` and `ArgMax` functions. Details will be provided in descriptions of the individual tasks.




# Task 7: Read Matrix

## Conceptual Overview: Matrix Files

Recall from Task 5 that matrices are stored in memory as an integer array in row-major order.

Matrices are stored in files as a consecutive sequence of 4-byte integers. The first and second integers in the file indicate the number of rows and columns in the matrix, respectively. The rest of the integers store the elements in the matrix in row-major order.

To view matrix files, you can run `make read matrix_file.bin`. We have also provided a human-readable version of each of the matrix files at `matrix_file.txt`.

A video walkthrough of how to read and convert matrix files is available at this link.

## Your Task

Fill in the `read_matrix` function in `src/read_matrix.s`. This function should do the following:

1. Open the file with read permissions. The filepath is provided as an argument (`a0`).
2. Read the number of rows and columns from the file (recall: these are the first two integers in the file). Store these integers in memory at the provided pointers (`a1` for rows and `a2` for columns).
3. Allocate space on the heap to store the matrix. (Hint: Use the number of rows and columns from the previous step to determine how much space to allocate.)
4. Read the matrix from the file to the memory allocated in the previous step.
5. Close the file.
6. Return a pointer to the matrix in memory.

| `read_matrix`: Task 7. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `char *` | A pointer to the filename string. |
| | `a1` | `int *` | A pointer to an integer which will contain the number of rows. |
| | `a2` | `int *` | A pointer to an integer which will contain the number of columns. |
| **Return values** | `a0` | `int *` | A pointer to the matrix in memory. |

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

| **Return code** | **Exception** |
|---|---|
| 88 | `malloc` returns an error. |
| 89 | `fopen` returns an error. |
| 90 | `fclose` returns an error. |
| 91 | `fread` does not read the correct number of bytes. |

To implement this function, you will need to call some utility functions. A complete set of function definitions can be found in the appendix. The relevant function definitions for this task are provided below (expand the section to see them).

▼ Task 7: Relevant Function Definitions

1. Open the file with read permissions. The filepath is provided as an argument (`a0`).

| `fopen`: Open a file for reading or writing. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `str *` | A pointer to the filename string. |
| | `a2` | `int` | Permission bits. 0 for read-only, 1 for write-only. |
| **Return values** | `a0` | `int` | A file descriptor. This integer can be used in other file operation functions to refer to the opened file. If opening the file failed, this value is -1. |

2. Read the number of rows and columns from the file (recall: these are the first two integers in the file). Store these integers in memory at the provided pointers (`a1` for rows and `a2` for columns).

| `fread`: Read bytes from a file to a buffer in memory. Subsequent reads will read from later parts of the file. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `int` | The file descriptor of the file we want to read from, previously returned by `fopen`. |
| | `a2` | `int*` | A pointer to the buffer where the read bytes will be stored. The buffer should have been previously allocated with `malloc`. |
| | `a3` | `int` | The number of bytes to read from the file. |

| | | | |
|---|---|---|---|
| **Return values** | `a0` | `int` | The number of bytes actually read from the file. If this differs from the argument provided in `a3`, then we either hit the end of the file or there was an error. |

3. Allocate space on the heap to store the matrix. (Hint: Use the number of rows and columns from the previous step to determine how much space to allocate.)

| `malloc`: Allocates heap memory. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int` | The size of the memory that we want to allocate (in bytes). |
| **Return values** | `a0` | `void *` | A pointer to the allocated memory. If the allocation failed, this value is 0. |

4. Read the matrix from the file to the memory allocated in the previous step. (Use `fread` from above.)
5. Close the file.

| `fclose`: Close a file, saving any writes we have made to the file. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `int` | The file descriptor of the file we want to close, previously returned by `fopen`. |
| **Return values** | `a0` | `int` | 0 on success, and -1 otherwise. |

6. Return a pointer to the matrix in memory.

To test your function, run `make test-read_matrix`. Refer back to Task 1 for debugging instructions.

# Task 8: Write Matrix

Fill in the `write_matrix` function in `src/write_matrix.s`. This function should do the following:

1. Open the file with write permissions. The filepath is provided as an argument.
2. Write the number of rows and columns to the file. (Hint: The `fwrite` function expects a pointer to data in memory, so you should first store the data to memory, and then pass a pointer to the data to `fwrite`.)
3. Write the data to the file.
4. Close the file.

| `write_matrix`: Task 8. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `char *` | A pointer to the filename string. |
| | `a1` | `int *` | A pointer to the matrix in memory (stored as an integer array). |
| | `a2` | `int` | The number of rows in the matrix. |
| | `a3` | `int` | The number of columns in the matrix. |
| **Return values** | None | | |

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

| **Return code** | **Exception** |
|---|---|
| 89 | `fopen` returns an error. |
| 92 | `fwrite` does not write the correct number of bytes. |
| 90 | `fclose` returns an error. |

To implement this function, you will need to call some utility functions. A complete set of function definitions can be found in the appendix. The relevant function definitions for this task are provided below (expand the section to see them).

▼ Task 8: Relevant Function Definitions

1. Open the file with write permissions. The filepath is provided as an argument.

| `fopen`: Open a file for reading or writing. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `str *` | A pointer to the filename string. |
| | `a2` | `int` | Permission bits. 0 for read-only, 1 for write-only. |
| **Return values** | `a0` | `int` | A file descriptor. This integer can be used in other file operation functions to refer to the opened file. If opening the file failed, this value is -1. |

2. Write the number of rows and columns to the file. (Hint: The `fwrite` function expects a pointer to data in memory, so you should first store the data to memory, and then pass a pointer to the data to `fwrite`.)
3. Write the data to the file.

| `fwrite`: Write bytes from a buffer in memory to a file. Subsequent writes append to the end of the existing file. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `int` | The file descriptor of the file we want to read from, previously returned by `fopen`. |
| | `a2` | `void *` | A pointer to a buffer containing what we want to write to the file. |
| | `a3` | `int` | The number of elements to write to the file. |
| | `a4` | `int` | The size of each element. In total, `a3` × `a4` bytes are written. |
| **Return values** | `a0` | `int` | The number of items actually written to the file. If this differs from the number of items specified (`a3`), then we either hit the end of the file or there was an error. |

4. Close the file.

| `fclose`: Close a file, saving any writes we have made to the file. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `int` | The file descriptor of the file we want to close, previously returned by `fopen`. |
| **Return values** | `a0` | `int` | 0 on success, and -1 otherwise. |

To test your function, run `make test-write_matrix`. Refer back to Task 1 for debugging instructions.

# Task 9: Classify

Fill in the `classify` function in `src/classify.s`. This function should do the following:

1. Read three matrices `m0`, `m1`, and `input` from files. Their filepaths are provided as arguments.
2. Compute `h = matmul(m0, input)`. You will probably need to `malloc` space to fit `h`.
3. Compute `h = relu(h)`. Recall that `relu` is performed in-place.
4. Compute `o = matmul(m1, h)` and write the resulting matrix to the `output` file. The `output` filepath is provided as an argument.
5. Compute and return `argmax(o)`. If the print argument is set to 0, then also print out `argmax(o)` and a newline character.
6. Free any data you allocated with `malloc` in this function.

| `classify`: Task 9. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int` | `argc` (the number of arguments provided) |
| | `a1` | `char **` | `argv`, a pointer to an array of argument strings (`char *`) |
| | `a1[1]` = `*(a1 + 4)` | `char *` | A pointer to the filepath string of the first matrix file `m0`. |

| | | | |
|---|---|---|---|
| | `a1[2] = *(a1 + 8)` | `char *` | A pointer to the filepath string of the second matrix file `m1`. |
| | `a1[3] = *(a1 + 12)` | `char *` | A pointer to the filepath string of the input matrix file `input`. |
| | `a1[4] = *(a1 + 16)` | `char *` | A pointer to the filepath string of the output file. |
| | `a2` | `int` | If set to 0, print out the classification. Otherwise, do not print anything. |
| **Return values** | `a0` | `int` | The classification (see above). |

If the input is malformed in the following ways, put the appropriate return code into `a1` and run `call exit2` to quit the program.

| **Return code** | **Exception** |
|---|---|
| 88 | `malloc` returns an error. |
| 72 | There are an incorrect number of command line arguments. |

To implement this function, you will need to call some utility functions. A complete set of function definitions can be found in the appendix. The relevant function definitions for this task are provided below (expand the section to see them).

▼ Task 9: Relevant Function Definitions

1. Read three matrices `m0`, `m1`, and `input` from files. Their filepaths are provided as arguments.

| `read_matrix`: Task 7. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `char *` | A pointer to the filename string. |

| | | | |
|---|---|---|---|
| | `a1` | `int *` | A pointer to an integer which will contain the number of rows. |
| | `a2` | `int *` | A pointer to an integer which will contain the number of columns. |
| **Return values** | `a0` | `int *` | A pointer to the matrix in memory. |

2. Compute `h = matmul(m0, input)`. You will probably need to `malloc` space to fit `h`.

| `malloc`: Allocates heap memory. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int` | The size of the memory that we want to allocate (in bytes). |
| **Return values** | `a0` | `void *` | A pointer to the allocated memory. If the allocation failed, this value is 0. |

| `matmul`: Task 5. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int *` | A pointer to the start of the first matrix A (stored as an integer array in row-major order). |
| | `a1` | `int` | The number of rows (height) of the first matrix A. |
| | `a2` | `int` | The number of columns (width) of the first matrix A. |
| | `a3` | `int *` | A pointer to the start of the second matrix B (stored as an integer array in row-major order). |
| | `a4` | `int` | The number of rows (height) of the second matrix B. |
| | `a5` | `int` | The number of columns (width) of the second matrix B. |

| | | | |
|---|---|---|---|
| | `a6` | `int *` | A pointer to the start of an integer array where the result C should be stored. You can assume this memory has been allocated (but is uninitialized) and has enough space to store C. |
| **Return values** | None | | |

3. Compute `h = relu(h)`. Recall that `relu` is performed in-place.

| `relu`: Task 2. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int *` | A pointer to the start of the integer array. |
| | `a1` | `int` | The number of integers in the array. You can assume that this argument matches the actual length of the integer array. |
| **Return values** | None | | |

4. Compute `o = matmul(m1, h)` and write the resulting matrix to the `output` file. The `output` filepath is provided as an argument.

| `write_matrix`: Task 8. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `char *` | A pointer to the filename string. |
| | `a1` | `int *` | A pointer to the matrix in memory (stored as an integer array). |
| | `a2` | `int` | The number of rows in the matrix. |
| | `a3` | `int` | The number of columns in the matrix. |
| **Return values** | None | | |

5. Compute and return `argmax(o)`. If the print argument is set to 0, then also print out `argmax(o)` and a newline character.

| `argmax`: Task 3. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int *` | A pointer to the start of the integer array. |
| | `a1` | `int` | The number of integers in the array. You can assume that this argument matches the actual length of the integer array. |
| **Return values** | `a0` | `int` | The index of the largest element. If the largest element appears multiple times, return the smallest index. |

| `print_int`: Prints an integer. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `int` | The integer to print. |
| **Return values** | None | | |

| `print_char`: Prints a character. | | | |
|---|---|---|---|
| **Arguments** | `a1` | `char` | The character to print. |
| **Return values** | None | | |

6. Free any data you allocated with `malloc` in this function.

| `free`: Frees heap memory. | | | |
|---|---|---|---|
| **Arguments** | `a0` | `int` | A pointer to the allocated memory to be freed. |
| **Return values** | None | | |

# Testing and Debugging

To test your function, run `make test-classify` and `make test-main`.

Debugging instructions are available in both video form and written form (expand the section below).

▼ Text Version

TODO: Video transcript.

# Submission and Grading

Submit your code to the Project 2B assignment on Gradescope.

To ensure the autograder runs correctly, do not add any `.import` statements to the starter code. Also, make sure there are no `ecall` instructions in your code.

Congratulations on finishing Project 2! Just for fun, you can now use your code to classify your own handwritten digits. This is optional and not required to finish the project. If you're interested, expand the section below.

▼ Optional: Classifying Your Own Digits

First, open up any basic drawing program like Microsoft Paint.

Next, resize the image to 28x28 pixels, draw your digit, and save it as a `.bmp` file in the directory `inputs/mnist/student_inputs/`.

Inside that directory, we've provided `bmp_to_bin.py` to turn this `.bmp` file into a `.bin` file for the neural net, as well as an `example.bmp` file. To convert it, run the following from inside the `inputs/mnist/student_inputs` directory:

```
$ python bmp_to_bin.py example

This will read in the example.bmp file, and create an example.bi
```

```
$ java -jar tools/venus.jar src/main.s -ms -1 -it inputs/mnist/bin/m0.
```

```
You can convert and run your own .bmp files in the same way.


You should be able to achieve a reasonable accuracy with your ow
```

Dark Mode