Report for Lab 5

Task 1 – MPI version

- We followed the given instructions to build and run the code.
- All the work is done in module `nnetwork.cxx`.
- We distributed the computation of the updated weight vectors to allow the processes to work on their own data. In particular, we distributed chunks of rows to different processes.
- Each call to `dot` function will carry out a matrix multiplication and the result is stored in output vectors, therefore, we split and distribute the generation of each output vector into different MPI processes based on the the rows. Each MPI will process will get `total_row_num/process_num` rows to deal with.
- As mentioned, each process will compute the same functions on a different chunk of data.
- The root process is collecting and displaying the training progress every 100 iterations.

```
MPI_Gather(dW3.data(), ...);
MPI_Gather(dW2.data(), ...);
MPI_Gather(dW1.data(), ...);
 if (mpirank == 0) {
    W1 = W1 - lr * dW1;
    W2 = W2 - lr * dW2;
    W3 = W3 - lr * dW3;
}
    ...
if ((mpirank == 0) && (i+1) % 100 == 0){
    ...
};
```

- We used the given commands to figure out the number of total processes as well as the rank of the executing process.
- We are mainly using `MPI` collectives in the implementation.
  - For the data needed only by the root process, we used `MPI_Gather` to gather all the data partitions from all processes into root process.
  - For the data needed by all the processes, we used `MPI_Allgather` to make all processes get the same copy of the data.
  - For the data needed reduced at root process but needed by all the processes, we used `MPI_Bcast` to distribute the data to each processes.
- We used the strategy based on the collectives given above. First, the initial weights as well as the current batch index are broadcast to all processes. Second, each process does the computation in the training loop on a separate batch of data. Finally, the results are gathered in the root process.

- The table below shows the speedup for each number of processes.

	1 process	2 processes	4 processes
Total time	66.616 s	38.100 s	22.751 s
Speedup	1.0000	1.7485	2.9280

Table 1: Speedup for different number of processes

Task 2 – Parallel GEMM per process

We made an hybrid implementation of `MPI` and `OpenMP`. For `OpenMP`, we parallize the most outer loop of the `GEMM`. We set `OMP_NUM_THREADS`=2 and start 2 MPI processes by `mpirun -bind-to none -np 2 ./nnetwork_mpi`, so that we still fully utilize the available CPU resource without over utilization.

The result shows that the setup with 2 MPI processes + 2 threads per process (total: 21.616 s) slightly outperforms the setup with 4 MPI processes + 1 thread per process (total: 22.751 s). This is simply because whenever shared memory space is possible, it should be better than alternative distributed memory space due to smaller overhead.