High Performance Computing
Homework #5: Phase 1

Due: Wed Oct 20 2021 before 11:59 PM

Email-based help Cutoff: 5:00 PM on Tue, Oct 19 2021

! The runtime data and results in this report are meaningful only if your implementation is functionally correct and produce similar outputs as the reference run.

Name: Maciej Wozniak

Experimental Platform

The experiments documented in this report were conducted on the following platform:

Component	***Details***
CPU Model	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
CPU/Core Speed	2.40GHz
Operating system used	Unix
Interconnect type & speed (if applicable)	Not applicable
Was machine dedicated to task (`yes/no`)	Yes (via a `slurm` job)
Name and version of C++ compiler (if used)	
Name and version of Java compiler (if used)	None
Name and version of other non-standard software tools & components (if used)	

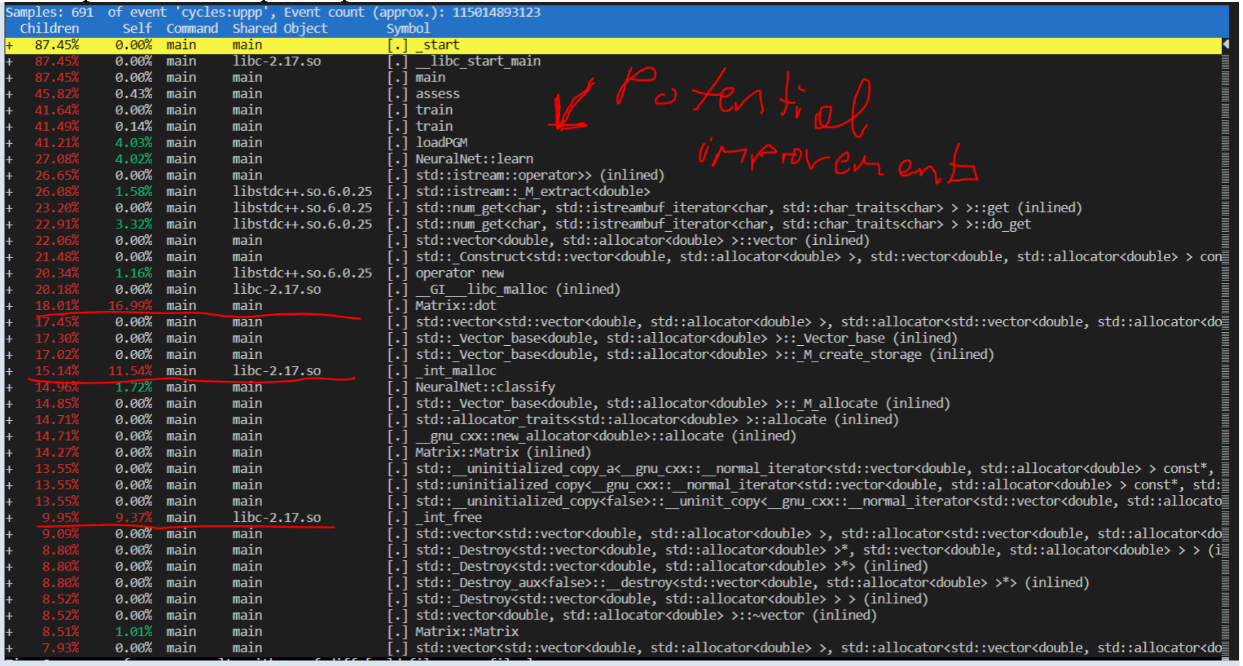
Runtime data for the reference performance

In the table below, record the reference runtime characteristics of the starter code:

Rep	**User time (sec)**	**Elapsed time (sec)**	**Peak memory (KB)**
1	31.49	34.13	3516
2	31.52	34.14	3516
3	31.12	34.73	3516
4	31.59	34.23	3516
5	31.63	34.30	3516

Perf report data for the reference implementation

In the space below, copy-paste the `perf` profile data that you used to identify the aspect/method to reimplement to improve performance:



```
Samples: 691  of event 'cycles:uppp', Event count (approx.): 115014893123
  Children      Self  Command  Shared Object        Symbol
+   87.45%     0.00%  main     main                 [.] _start
+   87.45%     0.00%  main     libc-2.17.so         [.] __libc_start_main
+   87.45%     0.00%  main     main                 [.] main
+   45.82%     0.43%  main     main                 [.] assess
+   41.64%     0.00%  main     main                 [.] train
+   41.49%     0.14%  main     main                 [.] train
+   41.21%     4.03%  main     main                 [.] loadPGM
+   27.08%     4.02%  main     main                 [.] NeuralNet::learn
+   26.65%     0.00%  main     main                 [.] std::istream::operator>> (inlined)
+   26.08%     1.58%  main     libstdc++.so.6.0.25  [.] std::istream::_M_extract<double>
+   23.20%     0.00%  main     libstdc++.so.6.0.25  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::get (inlined)
+   22.91%     3.32%  main     libstdc++.so.6.0.25  [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::do_get
+   22.06%     0.00%  main     main                 [.] std::vector<double, std::allocator<double> >::vector (inlined)
+   21.48%     0.00%  main     main                 [.] std::_Construct<std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> > con
+   20.34%     1.16%  main     libstdc++.so.6.0.25  [.] operator new
+   20.18%     0.00%  main     libc-2.17.so         [.] __GI___libc_malloc (inlined)
+   18.01%    16.99%  main     main                 [.] Matrix::dot
+   17.45%     0.00%  main     main                 [.] std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<do
+   17.30%     0.00%  main     main                 [.] std::_Vector_base<double, std::allocator<double> >::_Vector_base (inlined)
+   17.02%     0.00%  main     main                 [.] std::_Vector_base<double, std::allocator<double> >::_M_create_storage (inlined)
+   15.14%    11.54%  main     libc-2.17.so         [.] _int_malloc
+   14.96%     1.72%  main     main                 [.] NeuralNet::classify
+   14.85%     0.00%  main     main                 [.] std::_Vector_base<double, std::allocator<double> >::_M_allocate (inlined)
+   14.71%     0.00%  main     main                 [.] std::allocator_traits<std::allocator<double> >::allocate (inlined)
+   14.71%     0.00%  main     main                 [.] __gnu_cxx::new_allocator<double>::allocate (inlined)
+   14.27%     0.00%  main     main                 [.] Matrix::Matrix (inlined)
+   13.55%     0.00%  main     main                 [.] std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<std::vector<double, std::allocator<double> > const*,
+   13.55%     0.00%  main     main                 [.] std::uninitialized_copy<__gnu_cxx::__normal_iterator<std::vector<double, std::allocator<double> > const*, std:
+   13.55%     0.00%  main     main                 [.] std::__uninitialized_copy<false>::__uninit_copy<__gnu_cxx::__normal_iterator<std::vector<double, std::allocato
+    9.95%     9.37%  main     libc-2.17.so         [.] _int_free
+    9.09%     0.00%  main     main                 [.] std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<do
+    8.80%     0.00%  main     main                 [.] std::_Destroy<std::vector<double, std::allocator<double> >*, std::vector<double, std::allocator<double> > > (i
+    8.80%     0.00%  main     main                 [.] std::_Destroy<std::vector<double, std::allocator<double> >*> (inlined)
+    8.80%     0.00%  main     main                 [.] std::_Destroy_aux<false>::__destroy<std::vector<double, std::allocator<double> >*> (inlined)
+    8.52%     0.00%  main     main                 [.] std::_Destroy<std::vector<double, std::allocator<double> > > (inlined)
+    8.52%     0.00%  main     main                 [.] std::vector<double, std::allocator<double> >::~vector (inlined)
+    8.51%     1.01%  main     main                 [.] Matrix::Matrix
+    7.93%     0.00%  main     main                 [.] std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<do
```

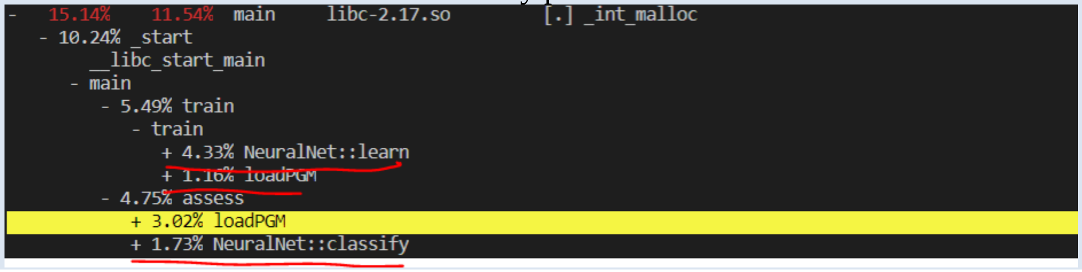
Description of performance improvement

Briefly describe the performance improvement you are implementing. Your description should document:

- Why you chose the specific aspect/feature to improve (obviously it should be supported by your `perf` data)
- What is the best-case improvement that you anticipate – for example, if you optimize a feature that takes 25% of runtime, then the best case would be a 25% reduction in runtime.
- Briefly describe what/how you plan to change the implementation

I will focus on improving the parts which has the highest self-percentage. We can see that these are **matrix.dot,**

Some functions in neuralnetwork classify part



```
-   15.14%    11.54%  main     libc-2.17.so       [.] _int_malloc
   - 10.24% _start
        __libc_start_main
      - main
         - 5.49% train
            - train
               + 4.33% NeuralNet::learn
               + 1.16% loadPGM
         - 4.75% assess
            + 3.02% loadPGM
            + 1.73% NeuralNet::classify
```

I will focus on improving these functions. It is hard to say what exactly will be improvements but we can expect 20-30% improvement if we decrease value of all 3 functions from 15-20% to 5%. Also if we check the whole **perf** report we can see that a lot of time is spent on allocation.

Runtime statistics from performance improvement

Use the supplied SLURM script to collect runtime statistics for your enhanced implementation.

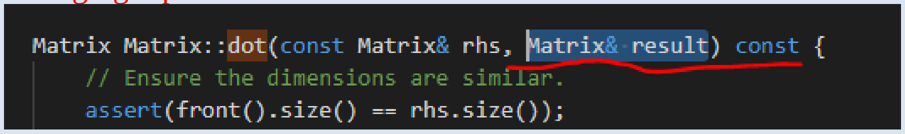
Rep	User time (sec)	Elapsed time (sec)	Peak memory (KB)
1	35.77	36.59	3516
2	34.26	34.9	3516
3	33.62	34.26	3516
4	34.78	35.43	3516
5	35.69	36.41	3516

Comparative runtime analysis

Compare the runtimes (*i.e.*, before and after your changes) by fill-in the Runtime Comparison Template and copy-paste the full sheet in the space below:

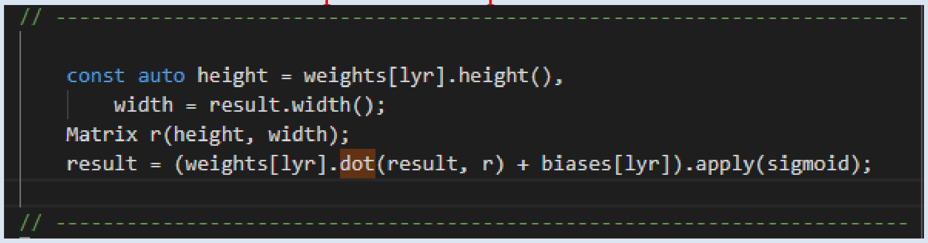
user time			elapsed time		
Replicate#	**original code**	**edited code**	**Replicate#**	**original code**	**edited code**
1	31.49	35.77	1	34.13	36.59
2	31.52	34.26	2	34.14	34.9
3	31.12	33.62	3	34.73	34.26
4	31.59	34.78	4	34.23	35.43
5	31.63	35.69	5	34.3	36.41
Average:	31.47	34.824	**Average:**	34.306	35.518
SD:	0.2033469941	0.9239209923	**SD:**	0.2470425065	0.9895807193
95% CI Range:	0.2524886417	1.147199433	**95% CI Range:**	0.3067437863	1.228726752
Stats:	**31.47 ± 0.25**	**34.824 ± 1.15**	**Stats:**	**34.306 ± 0.31**	**35.518 ± 1.23**
T-Test (H_0: μ1=μ2)	0.0009252383105		**T-Test (H_0: μ1=μ2)**	0.05014005963	

1. In **Matrix::dot** we can remove placeholder called sum and change the value directly in the matrix cell.
   This unfortunately **increased** time by 2sec…
2. Correcting types (**does auto slow down the process?)**
3. Another improvement we can try is suing PGO – profile guided optimization.
   Even though it reduced *self-percentage* for some of the methods (no method has % higher than 10) it increased self % for other parts of the program and consequently increased the time to 35sec, also not good
4. Methods inlining. We are going to put the train function into **main** improve the performance this way.
   Didn't help much and also increased time
5. Changing the matrix::dot by:
   a) Changing input as a matrixB and resutlPlaceholder



```
Matrix Matrix::dot(const Matrix& rhs, Matrix& result) const {
    // Ensure the dimensions are similar.
    assert(front().size() == rhs.size());
```

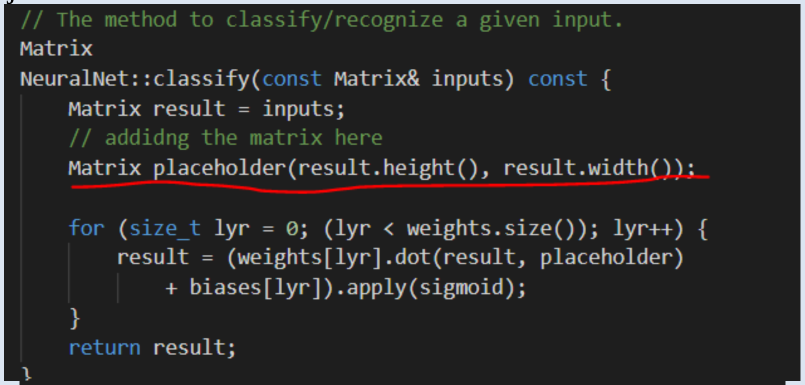
   b) Each matrix multiplication part of the code changed into



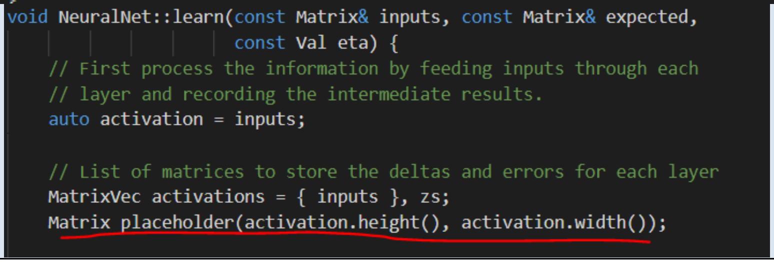
```
// ---------------------------------------------------------------------

    const auto height = weights[lyr].height(),
        width = result.width();
    Matrix r(height, width);
    result = (weights[lyr].dot(result, r) + biases[lyr]).apply(sigmoid);

// ---------------------------------------------------------------------
```

It did not improve performance much, the average is around 33 seconds because we are doing exactly the same thing but outside the loop

6. Copying just twice in NeuralNetworks::learn and NeuralNetworks::Classify

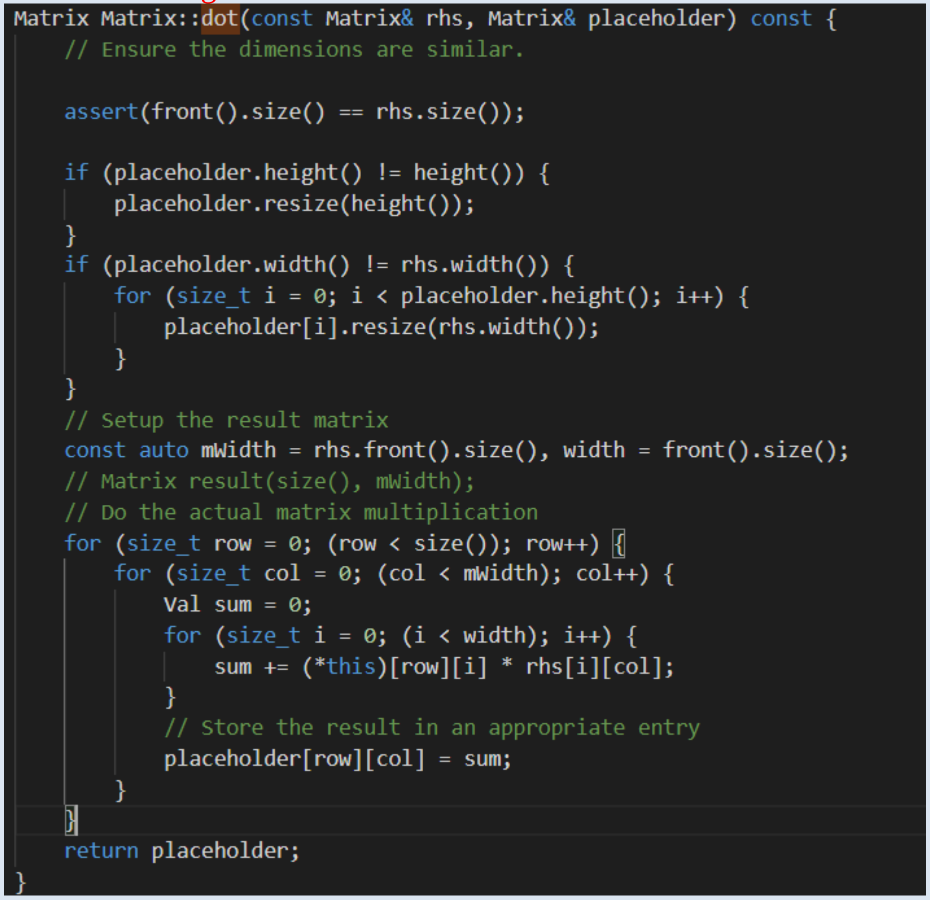


```
// The method to classify/recognize a given input.
Matrix
NeuralNet::classify(const Matrix& inputs) const {
    Matrix result = inputs;
    // addidng the matrix here
    Matrix placeholder(result.height(), result.width());

    for (size_t lyr = 0; (lyr < weights.size()); lyr++) {
        result = (weights[lyr].dot(result, placeholder)
            + biases[lyr]).apply(sigmoid);
    }
    return result;
}
```



```
void NeuralNet::learn(const Matrix& inputs, const Matrix& expected,
                      const Val eta) {
    // First process the information by feeding inputs through each
    // layer and recording the intermediate results.
    auto activation = inputs;

    // List of matrices to store the deltas and errors for each layer
    MatrixVec activations = { inputs }, zs;
    Matrix placeholder(activation.height(), activation.width());
```

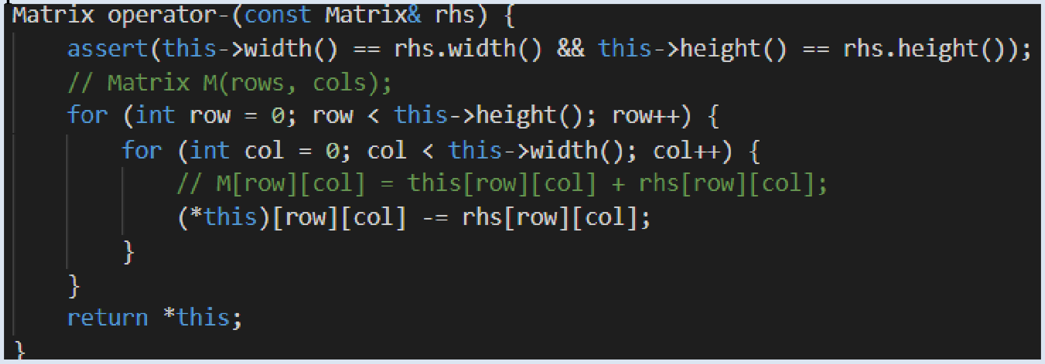
And overwriting matix::dot method



```
Matrix Matrix::dot(const Matrix& rhs, Matrix& placeholder) const {
    // Ensure the dimensions are similar.

    assert(front().size() == rhs.size());

    if (placeholder.height() != height()) {
        placeholder.resize(height());
    }
    if (placeholder.width() != rhs.width()) {
        for (size_t i = 0; i < placeholder.height(); i++) {
            placeholder[i].resize(rhs.width());
        }
    }
    // Setup the result matrix
    const auto mWidth = rhs.front().size(), width = front().size();
    // Matrix result(size(), mWidth);
    // Do the actual matrix multiplication
    for (size_t row = 0; (row < size()); row++) {
        for (size_t col = 0; (col < mWidth); col++) {
            Val sum = 0;
            for (size_t i = 0; (i < width); i++) {
                sum += (*this)[row][i] * rhs[i][col];
            }
            // Store the result in an appropriate entry
            placeholder[row][col] = sum;
        }
    }
    return placeholder;
}
```

c) That make the algorithm to run 38s on average on in user time and **up to 2 minutes in elapsed time**

7. I also edited adding, subtracting and multiplication methods so they do not allocate additional space (as apply was doing). That did not really improve the code or performance



```
Matrix operator-(const Matrix& rhs) {
    assert(this->width() == rhs.width() && this->height() == rhs.height());
    // Matrix M(rows, cols);
    for (int row = 0; row < this->height(); row++) {
        for (int col = 0; col < this->width(); col++) {
            // M[row][col] = this[row][col] + rhs[row][col];
            (*this)[row][col] -= rhs[row][col];
        }
    }
    return *this;
}
```

Inferences & Discussions

Now, using the data from the runtime statistics discuss (at least 5-to-6 sentences) the change in runtime characteristics (both time and memory) due to your changes. Compare and contrast key aspects/changes to the implementation. Include any additional inferences as to why one version performs better than the other.

Even though I used various approached to improve the methods, none of them seems to be working. Using predefined matrix as a results template, slowed down the whole process significantly, probably by constant need of resizing it. Honestly, we could have expected it a bit... Redefining the addition and subtraction functions did not improved much but decreased gap between the user and elapsed time. Surprisingly, it did not decreased memory allocation as we expected (since we do not copy a matrix).