Performance analysis neural network

februari 2025

Student	Stan Merlijn
Email	stan.merlijn@student.hu.nl
Student number	1863967
Link github(classroom)	https://github.com/StanMerlijn/ML-sigmoid-neuron/tree/backpropagation
Testing framework	Catch2[1]

1 Abstract

In this essay, we discuss the performance improvements discovered during the development of a neural network. This essay summarizes the insights gained throughout the process. Although this essay does not contribute to the overall grading, it serves as a valuable reflection on the lessons learned during the assignment.

Contents

2 Performance improvements

2.1 Baseline Perforamance

Initially, the focus was on correctness rather than performance. While the implementation was accurate, it was inefficient. For example, training an XOR gate with 10,000 epochs using one hidden layer with 2 neurons and an output layer with a single neuron took approximately 190 ms. Remarkably, this is slower than the native Python implementation (despite Python's generally slower performance), which clearly indicated that our C++ implementation needed optimization. The first and simplest improvement was to compile in release mode. Since the project is built with CMake and the default build type is set to debug, switching to release mode significantly reduced the training time to roughly 25 ms. Further optimizations are discussed in the following chapters.

2.2 Returning Vectors by reference

Looking at the code, it was clear that the vectors were returned by value. This means that the vectors were copied every time they were returned. Meaning that for every copy new allocations were made. To test the amount of allocations made, the `new` operator was overloaded.

```
  static int s_Allocations = 0;

  void* operator new(size_t size) {
      s_Allocations++;
      return malloc(size);
}
```

By overloading the `new` operator, we can monitor the number of memory allocations. For example, by resetting `s_Allocations` to 0 before calling a function and printing its value afterwards, we determine that training the XOR network alone involved over 600 000 allocations.

A closer inspection of the code revealed that vectors were returned by value. Causing numerous unnecessary allocations. To address this, getter functions (and any functions returning vectors) were updated to return by reference. This modification greatly reduced the allocation count. For instance, the functions `getWeights` and `getNeurons` were changed from:

```
    std::vector<float> getWeights() const { return _weights; }

    std::vector<Neuron>& getNeurons() { return _neurons; }
```

to:

```
    const std::vector<float>& getWeights() const { return _weights; }

    const std::vector<Neuron>& getNeurons() { return _neurons; }
```

Fixing this issue reduced the number of allocations to approximately 40 000, and decreased the training time to 4 ms. Nevertheless, further optimizations are required, as the number of allocations remains too high. It is excpected that there are zero allocations made, because the vectors are preallocated.

2.3 Pre allocating memory

Preallocation of memory was used to reduce the number of allocations made. In this case the vectors for certain function we pre allocated. For example, the `FeedForward` function uses the `_currentLayerOutput` vector to store the output of the current layer. Now the inputs are only coppied to the `_currentLayerOutput` vector. Also, the `FeedForward` function now returns a reference to the `_currentLayerOutput` vector, eliminating unnecessary copies.

```
    const std::vector<float>& NeuronNetwork::feedForward(const std::vector<float>& inputs)
  {
      // Set the input vector and the current layer output
      _inputVec = inputs;
      _currentLayerOutput = inputs;

      // Feed forward through each layer in the network
      for (std::size_t i = 0; i < _layers.size(); i++) {
          _currentLayerOutput = _layers[i].feedForward(_currentLayerOutput);
      }
      return _currentLayerOutput;
  }
```

Here we allocate all the necessary vectors in the constructor of the `NeuronNetwork` class. All the output, input and temporary vectors are preallocated. In conjunction with the previous optimization, the number of allocations to train was reduced to zero. The training time was reduced to ±2.4 ms.

```
  NeuronNetwork::NeuronNetwork(std::vector<int> layerSizes)
{
    // Reserve the input vector and the current targets
    _inputVec.resize(layerSizes.front());
    _currentTargets.resize(layerSizes.back());

    // Reserve because there is no default constructor for NeuronLayer
    _layers.reserve(layerSizes.size());

    // Reserve the temp output buffer and the current layer output
    _tempOutputBuffer.resize(*std::max_element(layerSizes.begin(), layerSizes.end()));
    _currentLayerOutput.resize(*std::max_element(layerSizes.begin(), layerSizes.end()));

    // Create the layers
    for (std::size_t i = 1; i < layerSizes.size(); i++) {
        // If its the first layer then the input size is the first element in the layerSizes
        if (i == 1) {
            _layers.emplace_back(layerSizes[i], layerSizes.front());
        } else {
            _layers.emplace_back(layerSizes[i], layerSizes[i-1]);
        }
    }
}
```

2.4 Malloc and free

The malloc and free functions are used to dynamicly allocate and deallocate memory of size `size`. The malloc function asks the operating system if it can allocate a block of memory of size `size` bytes. Returning a pointer to the beginning of the block.

Depending on the type of operating system there are two possible algorithms that can be implemented in order to locate a block of memory in the heap to allocate[2]:

- **First fit:** The needed amount of memory is allocated in the first block that is large enough.
- **Best fit:** All of the heap is searched for the smallest block that is large enough.

Using the `Best fit` algorithm you can reduce the amount of memory fragmentation on the heap. HHeap fragmentation is a problem that occurs when the heap is filled with small blocks of memory that are not contiguous. Resulting in fragments scattered throughout the heap. It's an inefficient way of using memory since not all memory is used effectively. A fragmanted heap is slower because it takes longer to find the best hole to allocate memory.

Reserving memory for the vectors is a good way to reduce the amount of memory fragmentation.

3 Eigen library

One possible optimization is to use the Eigen library. This library is a C++ template library for linear algebra. It provides optimized matrix and vector operations. The library is written in C++ and supports all standard matrix and vector operations. It utilizes SIMD instructions instead of SISD instructions. SIMD instructions are Single Instruction, Multiple Data instructions.

3.1 SISD instructions

SISD or Single Instruction, Single Data instructions are instructions that operate on a single data element at a time, per clock cycle. This means that the CPU can only perform one operation on one data element at a time. So for example if you want to perform a dot product of two vectors, you would have to loop over the vectors and multiply the elements of the vectors and add them together. This can be improved by using SIMD instructions. Where you can perform multiple operations on multiple data elements at the same time.

3.2 SIMD instructions

SIMD or Single Instruction, Multiple Data instructions are instructions that operate on multiple data elements at the same time, per clock cycle. This means that the CPU can perform multiple operations on multiple data elements at the same time. This can be done by using vector registers. SIMD can be used cross platform, but the implementation is different for each platform. For example, on x86 you can use the AVX or SSE instructions. The intel has the instruction set

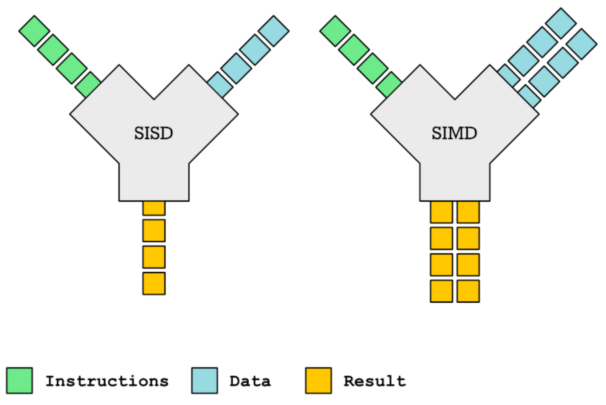

Figure 1: SISD instructions[3]

SIMD instructions can be found, to one degree or another, on most CPUs. Some examples of SIMD instructions are: Intel's MMX and iwMMXt, SSE, SSE2, SSE3 SSSE3 and SSE4.x, AMD's 3DNow!, ARC's ARC Video subsystem, SPARC's VIS and VIS2, Sun's MAJC, ARM's Neon technology, MIPS' MDMX (MaDMaX) and MIPS-3D. The IBM, Sony, Toshiba co-developed Cell Processor's SPU's instruction set is heavily SIMD based. Philips, now NXP, developed several SIMD processors named Xetal. The Xetal has 320 16-bit processor elements especially designed for vision tasks.[4]

Table 1: Benchmark Results for Merge Sort Iterative

Instruction set	Cache size	Data type
MMX	64 bits	Integer
SSE	128 bits	Single precision Floats
SSE2	128 bits	Double precision Floats
SSE3	128 bits	Integer
SSSE3	128 bits	Integer
SSE4.1	128 bits	Integer

3.3 Using Eigen

Using SIMD for the neuron network can theoretically speed up the training process. For in example in the activation of a neuron the dot product is computed. Where `b` is the bias, `w` is the weights and `x` is the input vector. The dot product is computed by multiplying the weights and inputs and adding them together.

$$d(x, b) = b + \sum_{i=0}^{n} w_i \times x_i \tag{1}$$

Where this function is implemented as follows:

```cpp
 float Neuron::activate(const std::vector<float> &inputs)
{
    // Calculate the weighted sum of the inputs
    _lastInput = inputs;
    float weightedSum = _bias;

    // Dot product of the weights and inputs
    for (std::size_t i = 0; i < _weights.size(); i++)
    {
        weightedSum += _weights[i] * inputs[i];
    }

    // Return the result of the sigmoid function
    _lastOutput = sigmoid(weightedSum);
    return _lastOutput;
}
```

In the current implementation, the dot product is computed by explicitly looping over the weights and inputs. This approach can be optimized by leveraging SIMD instructions. For instance, the dot product can be calculated more efficiently using the `Eigen` library. Alternatively, some compilers—such as `gcc`—are capable of generating SIMD instructions automatically.

```cpp
 float Neuron::activate(const Eigen::VectorXf &inputs)
{
    // Calculate the weighted sum of the inputs
    e_lastInput = inputs;
    e_weights = Eigen::Map<Eigen::VectorXf>(_weights.data(), _weights.size());

    // Calculate the weighted sum of the inputs
    float weightedSum = _bias + e_weights.dot(inputs);

    // Return the result of the sigmoid function
    _lastOutput = sigmoid(weightedSum);
    return _lastOutput;
}
```

Here we implemted the dot product using the `Eigen` library. The `Eigen::VectorXf` is a vector of floats. The `Eigen::Map` function is used to map the weights vector to an `Eigen::VectorXf` vector. Even though we map the weights vector to an `Eigen::VectorXf` vector, the performance increase we get from utilizing the SIMD instructions is noticable. Up to 4 times faster than the previous implementation, this makes sence because the dot product is computed for every neuron in the network with n weights and inputs, for every training pass times the number of epochs. Hence that is why even tho we copy the weights vector to an `Eigen::VectorXf` vector, the performance increase is still noticable.

4 Multi threading

Another possible optimization is to use multi threading. Multi threading is the ability of a CPU to execute multiple processes or threads concurrently. This can be done by using multiple cores or multiple CPUs. The CPU can switch between threads or processes to give the illusion that they are running concurrently. This can be done by using the `std::thread` Library in C++.

References

[1] Catchorg, "GitHub - catchorg/Catch2: A modern, C++-native, test framework for unit-tests, TDD and BDD - using C++14, C++17 and later (C++11 support is in v2.x branch, and C++03 on the Catch1.x branch)." [Online]. Available: https://github.com/catchorg/Catch2

[2] "CS 221." [Online]. Available: https://www.cs.uah.edu/~rcoleman/Common/C_Reference/MemoryAlloc.html

[3] I. Bogosavljević, "Crash course introduction to parallelism: SIMD Parallelism," 3 2022. [Online]. Available: https://johnnysswlab.com/crash-course-introduction-to-parallelism-simd-parallelism/

[4] W. contributors, "Single instruction, multiple data," 2 2025. [Online]. Available: https://en.wikipedia.org/wiki/Single_instruction,_multiple_data