Eliska Kloberdanz

COM S 525: Semester Project

Iowa State University

Distributed Neural Network Training with MPI

Spring 2021

Introduction

In this project I implement a parallel neural network in C++ leveraging Eigen for linear algebra operations, and MPI for data parallelism to reduce training time via distributed learning by spreading the training data across multiple processors.

The neural network I implemented contains two dense layers, a non-linear ReLU activation function, a softmax activation function, and is optimized via categorical cross entropy loss and stochastic gradient descent. The training data is divided into partitions equal to the total number of MPI processes p - 1, which are used as worker nodes. The neural network model itself is replicated in each of these worker nodes. Therefore, each worker has a copy of the neural network and operates only on a subset of the training data. One of the processors in the cluster stores the global model parameters, and acts as a parameter server that synchronizes and processes updates to the model parameters using the gradients calculated by the worker nodes. The optimization algorithm used in this project to find the neural network parameters (i.e.: the weights and biases) is Stochastic Gradient Descent (SGD), which is the most commonly used algorithm for performing parameter updates in neural networks. The traditional definition of SGD is inherently serial, which is why I use a parallel version of SGD in this project. There are two parameter update paradigms in parallel SGD: synchronous and asynchronous. In case of synchronous parameter update (shown in *Algorithm 1*) the parameter server waits to receive computed gradients from all worker nodes to update the global model parameters, which are then broadcasted to all worker nodes to be utilized in the next training iteration. On the other hand, asynchronous updates are processed by the

parameter server immediately without waiting to receive gradients from all worker nodes for the current training iteration. In this project I implement the synchronous version of SGD.

Algorithm 1 S-SGD

procedure S-SGD(parameters, data, N)
  for *each worker* $i \in \{1, 2, ..., N\}$ **do**
    FeedForward(parameters, $\frac{data}{N}$)
    $\nabla g_i \leftarrow$ BackPropagation()
  end for
  Synchronous()
  Aggregate from all workers: $\nabla g \leftarrow \frac{1}{N} \sum_{i=1}^{N} \nabla g_i$
  for *each worker* $i \in \{1, 2, ..., N\}$ **do**
    UpdateModel()
  end for
end procedure

A serial implementation of the neural network is used as a benchmark to assess the speed up from distributed training achieved via data parallelism with MPI and also scalability.

Background

There are two primary approaches to speeding up neural network training via parallelization: (1) model parallelism, and (2) data parallelism. Model parallelism involves distributing the neural network across different processors and training various parts of the model simultaneously. On the other hand, data parallelism distributes the training data across different processors and computes updates to the neural network in parallel. While model parallelism makes it possible to train neural networks that are larger than a single processor can support, it usually requires tailoring the model architecture to the available hardware. In contrast, data parallelism is model agnostic and applicable to any neural network architecture – it is the simplest and most widely used technique for parallelizing neural network training. It is also possible to implement a neural network

that leverages both model and data parallelism. A neural network developed by Dean et al. (2012) called DistBelief enables model parallelism within a machine (via multithreading) and across machines (via message passing). In addition to supporting model parallelism, the DistBelief framework also supports data parallelism, where multiple replicas of a model are used to optimize a single objective.

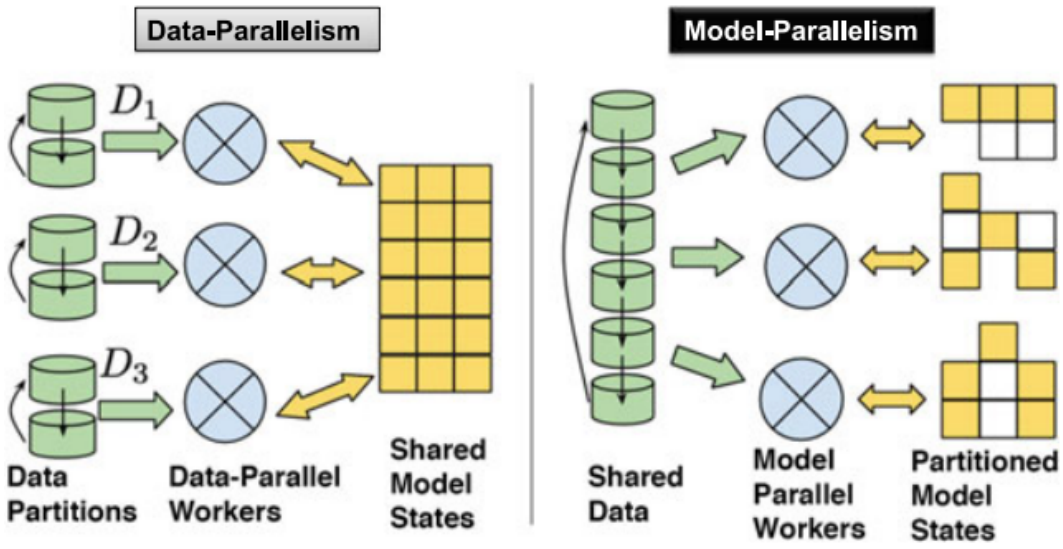


Figure 1: An illustration of data and model parallelism

Data Distribution

I decided to go with the standard data distribution that is used for distributed neural network training via data parallelism, which involves dividing the data into equal partitions. The number of partitions is equal to the total number of processes p - 1, which represent the workers. The model is replicated in each of these worker nodes; therefore, each worker operates on its own subset of the data. One of the p processes acts as a centralized parameter server, which holds the “global” neural network parameters and processes their updates. This data distribution is logical. Each worker has an identical copy of the neural network and performs the same computations as the other workers, which is why it makes sense to distribute the data evenly amongst the workers. The advantage of this distribution is that the workers should finish each training iteration at

about the same time. This minimizes waiting prior to each parameter update processed by the central parameter server (i.e.: the master process) in case of synchronous SGD. Another advantage is that even data distribution ensures that each worker computes parameter updates on a dataset that is large enough, which should produce reliable gradients and hence, reliable updates.

Serial Optimization

The first step to success in high performance computing is to produce an optimized serial version of the program, which can then be parallelized to allow for execution across multiple processors on a cluster.

Programming Language

The first design decision for developing optimized code for distributed neural network training was my choice of a programming language. A large number of scientific computing applications are written in Fortran, C, or C++, because these languages have the advantage of speed over more high level languages such as Python. I chose C++ over C and Fortran, because it offers more features and libraries such as Eigen.

Optimized Library Routines

Eigen is a free numerical library, which contains optimized implementations of various linear algebra routines. Training a neural network involves many linear algebra computations such as multiplying matrices, which is why I chose to use Eigen to ensure optimal performance. Another advantage of Eigen is that it is quite high level and easy to use.

Stride 1 Access

One of the basic serial optimization techniques is to ensure that arrays are accessed with a stride of 1. In C++ arrays are stored in memory in a row-major order, which means that row elements of a matrix or a vector are consecutively stored in memory. This is different from Fortran, which stores arrays in a column-major order. In this project, any vectors or

matrices are accessed such that a stride of 1 is ensured. Please see below a conceptual example showing how to achieve a stride of 1 in C++.

```
 1 #define ROWS 100
 2 #define COLS 100
 3
 4 // stride 100
 5 void bad_stride() {
 6   int A[ROWS][COLS];
 7
 8   for (int j = 0; j < COLS; ++j) {
 9     for (int i = 0; i < ROWS; ++i) {
10       A[i][j] = i + j;
11     }
12   }
13 }
14
15 // stride 1
16 void good_stride() {
17   int A[ROWS][COLS];
18
19   for (int i = 0; i < ROWS; ++i) {
20     for (int j = 0; j < COLS; ++j) {
21       A[i][j] = i + j;
22     }
23   }
24 }
```

Figure 2: A C++ code snippet of a conceptual example of bad and good stride

Variable precision

All variables in my neural network C++ implementation are double precision floating points or integers. Double precision floating point format takes up 64 bits in memory, which means that double precision can represent a floating point number more accurately than, for example, single precision with only 32 bits. While double precision ensures accuracy of computations, as future work I could experiment with mixed precision computations that are often used in HPC deep learning software as a serial optimization technique.

Compiler options

Another optimization that I utilized is the -O3 intel compiler optimization flag, which is recommended for code that involves intensive floating point calculations. Neural network training requires many linear algebra floating point operations, which is why I chose to use the -O3 compiler flag as opposed to the more general use -O2 optimization flag.

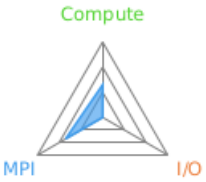
Load Balancing

Load balancing is a very important aspect of high performance computing, because it deals with how to distribute the work among multiple processors. Computing with multiple processors can help to speed up the runtime of a program, but it also adds communication overhead between the processors, which need to synchronize. Desirable load balancing ensures that all processors are busy simultaneously and that the amount of wait idle time is minimized.

I first ran a performance report utilizing the Allinea module to get a high level overview of how my parallel program utilizes the HPC cluster resources. Based on this report shown in Figure 1, it seems that most time is spent in MPI calls and running application code, where collective calls account for a vast majority of time spent in MPI calls and memory accesses account for most CPU computations. The thread usage is well optimized, but memory peak usage seems to be higher than the average usage.

allinea PERFORMANCE REPORTS

Command:	mpirun -np 64 ./net
Resources:	4 nodes (16 physical, 16 logical cores per node)
Memory:	126 GiB per node
Tasks:	64 processes
Machine:	hpc-class04
Start time:	Sun May 2 10:47:45 2021
Total time:	37 seconds
Full path:	/home/eklober



Summary: net is MPI-bound in this configuration

Compute	43.2%	Time spent running application code. High values are usually good. This is **low**; consider improving MPI or I/O performance first.
MPI	56.8%	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it.
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times.

This application run was MPI-bound. A breakdown of this time and advice for investigating further is in the MPI section below.

CPU

A breakdown of the 43.2% CPU time:

Scalar numeric ops	8.0%
Vector numeric ops	20.6%
Memory accesses	70.5%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 56.8% MPI time:

Time in collective calls	98.1%
Time in point-to-point calls	1.9%
Effective process collective rate	2.04 MB/s
Effective process point-to-point rate	208 MB/s

Most of the time is spent in collective calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Threads

A breakdown of how multiple threads were used:

Computation	100.0%
Synchronization	0.0%
Physical core utilization	99.8%
System load	101.7%

Thread usage appears to be well-optimized. Check the CPU breakdown for advice on further improving performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	75.3 MiB
Peak process memory usage	191 MiB
Peak node memory usage	4.0%

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how the 5.07 Wh was used:

CPU	100.0%
System	not supported %
Mean node power	not supported W
Peak node power	not supported W

The whole system energy has been calculated using the CPU energy usage.

No Allinea IPMI Energy Agent config file found in (null). Did you start the Allinea IPMI Energy Agent?

Figure 3: Allinea Performance Report

Another tool I used to assess the performance of my program in greater detail is MAP, a parallel profiler tool that identifies the parts code took the most time to execute. Figure 4 shows three metrics: main thread activity, CPU floating point operations, and memory usage. There are some small spikes in main thread activity and floating point unit usage; however, I believe that this is just the nature of neural network training - some parts of the algorithm are more computationally intensive than others. For example, the forward and backward pass performed on dense network layers involve matrix matrix multiplications, which are computationally intensive. This is evidenced by Figure 5, which shows that the majority of total CPU time is spent on the backward and forward pass of dense layers. Figure 5 also shows us that 12% of MPI time is spent in MPI_Barrier. This is not optimal; however, it is the nature of the synchronous SGD optimization algorithm used for computing updates for weights and biases of the neural network. Specifically, the synchronous SGD algorithm specifies that all worker nodes must compute all gradients for the network parameter updates before they can be averaged by the parameter server, which causes the increased wait time.

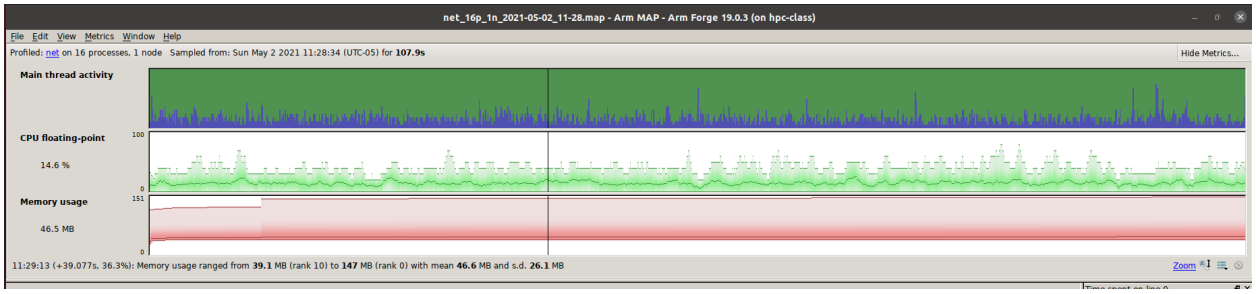


Figure 4: MAP Report: Main thread activity (top), CPU floating-point (middle), Memory usage (bottom)

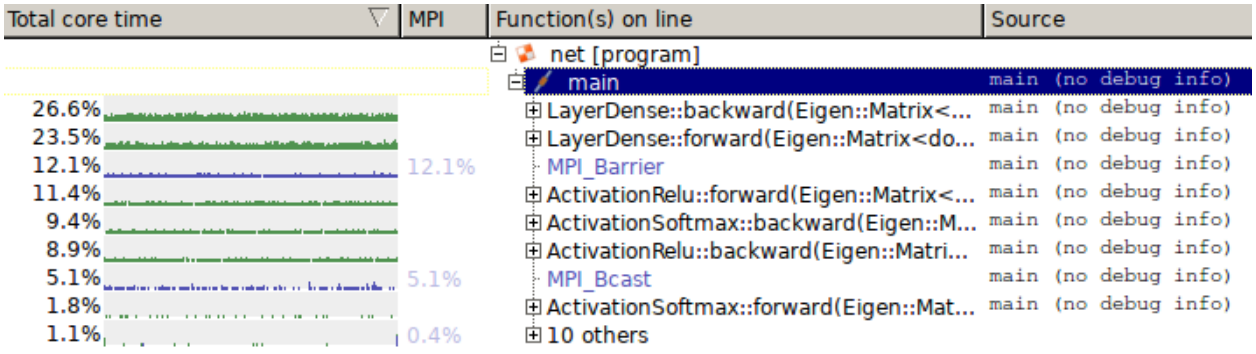


Figure 5: MAP Report: Main thread stacks

Performance/Scalability discussion and results

Data

The data used in this project consist of an X matrix and a y vector, which represent the features and labels respectively. The X matrix constrains two numerical features that are scaled to be in the range of -1 and 1 and form a spiral when plotted. The y vector represents labels, or categories associated with the X feature matrix. In this case there are 4 labels: 0, 1, 2, 3 and the goal is to train a neural network that can categorize input data into these four categories. Please see below a visual representation of the data in Figure 6, where the different spiral colors represent the different categories. It can be seen that this dataset and task are not trivial, because the data points are not linearly separable. Therefore, this dataset is a good use case for a neural network, which is a powerful universal function approximator. The training and testing data contain 40,000 and 4,000 examples respectively. I selected this data size for practical reasons. It allowed me to showcase the speed up in training time from utilizing MPI while being able to run the serial version of the code in a reasonable amount of time.

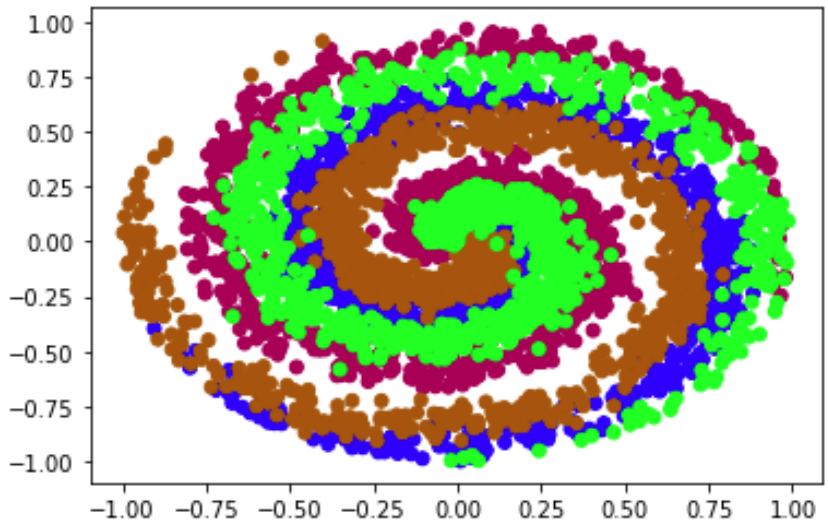


Figure 6: Spiral training data with 4 classes

Scalability

The most computationally expensive part of a neural network is the training. Neural network training involves passing inputs through the neural network during the forward pass and then calculating gradients during the backward pass, which are then used to update the network's weights and biases. This is why I focused on measuring the amount of time it takes to train a neural network as opposed to testing it. Table 1 shows the amount of time in seconds it takes to train a deep neural network using a varying number of processes while keeping the problem size constant. The first column shows the execution time of the serial code, which is much slower than any of the parallel versions. Merely upgrading from serial execution to a parallel one with 8 processes yields a significant speed up of more than a factor of 5 as shown in Table 2. As we double the number of processes the speed up if close to a factor of 2.

Table 1: Training execution time of neural network

Number of nodes and MPI processes	Nodes = 1, p=1	Nodes = 1, p=8	Nodes = 1, p=16	Nodes = 2, p=32	Nodes = 3, p=48	Nodes = 4, p=64
Execution time in seconds	791	146	94	46	27	19

Table 2: Scalability of parallel neural network training

Number of nodes and MPI processes	Nodes = 1, p=1	Nodes = 1, p=8	Nodes = 1, p=16	Nodes = 2, p=32	Nodes = 3, p=48	Nodes = 4, p=64
Speed up factor	N/A	5.4	1.6	2.0	1.7	1.4

Training and Testing Accuracy

The training and testing accuracy is quite high using any p number of processors. It is interesting that the highest accuracy is achieved with the serial training code. The reason for this could be the fact that in case of serial execution, the neural network updates are computed using gradients calculated on the entire training set, which mitigates overfitting and increases generalization performance that is manifested via high testing accuracy. The testing accuracy for different number of MPI processes is displayed in Table 3. Table 4 shows the training accuracy and loss across different training epochs for a varying number of MPI processes. Finally, Figure 7 and 8 graphically represent the training accuracy and loss as a function of epochs respectively.

Table 3: Testing accuracy of neural network

Number of nodes and MPI processes	Nodes = 1, p=1	Nodes = 1, p=8	Nodes = 1, p=16	Nodes = 2, p=32	Nodes = 3, p=48	Nodes = 4, p=64
Testing accuracy	81.60%	79.93%	70.90%	63.43%	78.20%	79.05%

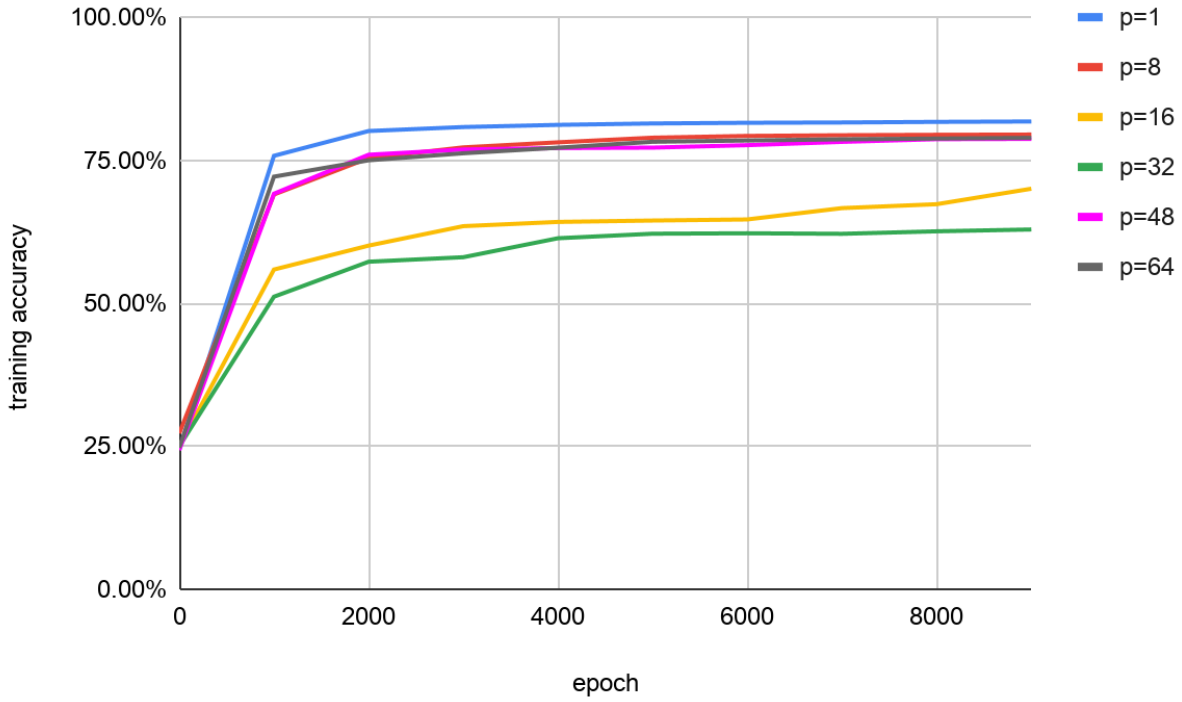


Figure 7: Training accuracy over epochs using p number of processors

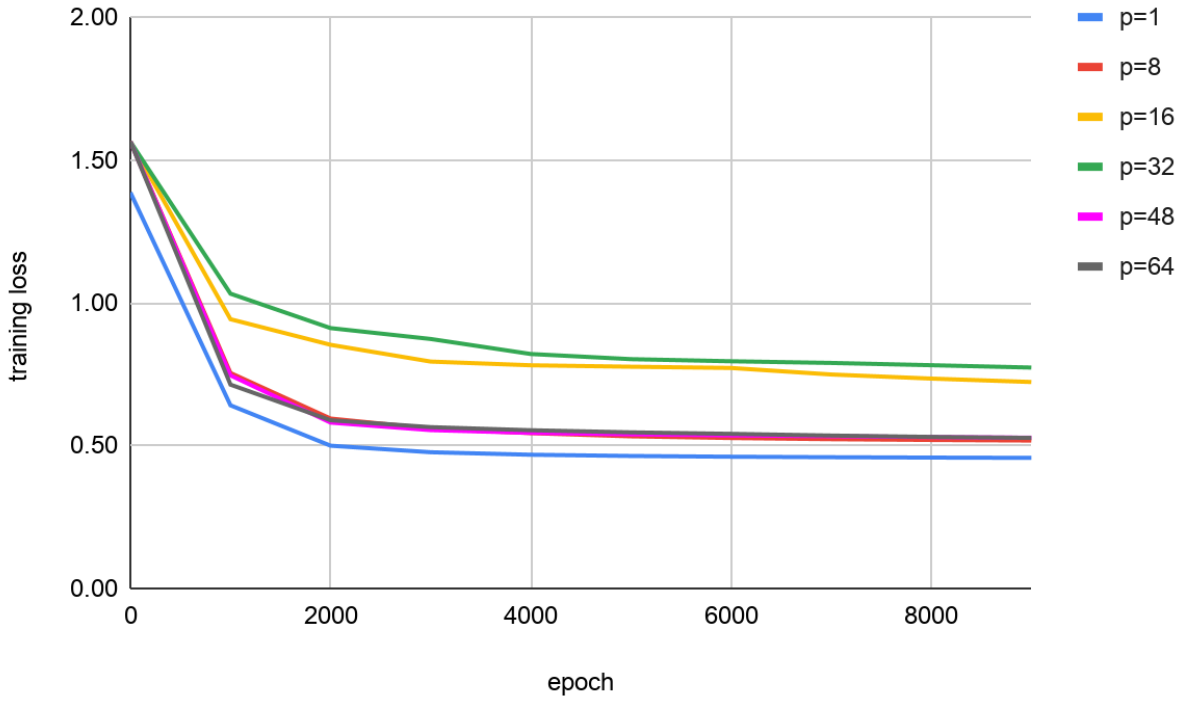


Figure 8: Training loss over epochs using p number of processors

Table 4: Training accuracy and loss over epochs using p number of processors

Serial	**Parallel: nodes = 1, p = 8**	**Parallel: nodes = 1, p = 16**	**Parallel: nodes = 2, p = 32**	**Parallel: nodes = 3, p = 48**	**Parallel: nodes = 4, p = 64**
epoch: 0	epoch: 0	epoch: 0	epoch: 0	epoch: 0	epoch: 0
train_accuracy: 0.244125	train_accuracy: 0.27265	train_accuracy: 0.25	train_accuracy: 0.25	train_accuracy: 0.243275	train_accuracy: 0.24735
loss: 1.3863	loss: 1.56419	loss: 1.56419	loss: 1.56419	loss: 1.56419	loss: 1.56419
epoch: 1000	epoch: 1000	epoch: 1000	epoch: 1000	epoch: 1000	epoch: 1000
train_accuracy: 0.757725	train_accuracy: 0.690125	train_accuracy: 0.558925	train_accuracy: 0.511425	train_accuracy: 0.6915	train_accuracy: 0.7213
loss: 0.641098	loss: 0.754222	loss: 0.942538	loss: 1.03193	loss: 0.745799	loss: 0.713488
epoch: 2000	epoch: 2000	epoch: 2000	epoch: 2000	epoch: 2000	epoch: 2000
train_accuracy: 0.801075	train_accuracy: 0.753725	train_accuracy: 0.60075	train_accuracy: 0.572575	train_accuracy: 0.759625	train_accuracy: 0.7499
loss: 0.499721	loss: 0.594768	loss: 0.852901	loss: 0.911382	loss: 0.580569	loss: 0.588854
epoch: 3000	epoch: 3000	epoch: 3000	epoch: 3000	epoch: 3000	epoch: 3000
train_accuracy: 0.808025	train_accuracy: 0.772475	train_accuracy: 0.63475	train_accuracy: 0.580475	train_accuracy: 0.76875	train_accuracy: 0.762175
loss: 0.476866	loss: 0.559589	loss: 0.794249	loss: 0.873338	loss: 0.554259	loss: 0.564701
epoch: 4000	epoch: 4000	epoch: 4000	epoch: 4000	epoch: 4000	epoch: 4000
train_accuracy: 0.811825	train_accuracy: 0.7813	train_accuracy: 0.642175	train_accuracy: 0.613625	train_accuracy: 0.771275	train_accuracy: 0.771875
loss: 0.4681	loss: 0.543767	loss: 0.781269	loss: 0.82074	loss: 0.544633	loss: 0.553889
epoch: 5000	epoch: 5000	epoch: 5000	epoch: 5000	epoch: 5000	epoch: 5000
train_accuracy: 0.8143	train_accuracy: 0.7894	train_accuracy: 0.6446	train_accuracy: 0.62155	train_accuracy: 0.772025	train_accuracy: 0.782275
loss: 0.463785	loss: 0.532602	loss: 0.776254	loss: 0.802749	loss: 0.539357	loss: 0.546489
epoch: 6000	epoch: 6000	epoch: 6000	epoch: 6000	epoch: 6000	epoch: 6000

train_accuracy: 0.815475	train_accuracy: 0.7925	train_accuracy: 0.6465	train_accuracy: 0.622325	train_accuracy: 0.77635	train_accuracy: 0.7845
loss: 0.461157	loss: 0.52662	loss: 0.771871	loss: 0.795361	loss: 0.535684	loss: 0.541215
epoch: 7000	epoch: 7000	epoch: 7000	epoch: 7000	epoch: 7000	epoch: 7000
train_accuracy: 0.816025	train_accuracy: 0.7937	train_accuracy: 0.6663	train_accuracy: 0.621375	train_accuracy: 0.7821	train_accuracy: 0.78655
loss: 0.459402	loss: 0.522551	loss: 0.749088	loss: 0.7895	loss: 0.532877	loss: 0.535241
epoch: 8000	epoch: 8000	epoch: 8000	epoch: 8000	epoch: 8000	epoch: 8000
train_accuracy: 0.817075	train_accuracy: 0.79445	train_accuracy: 0.67315	train_accuracy: 0.62575	train_accuracy: 0.7868	train_accuracy: 0.7879
loss: 0.458135	loss: 0.519791	loss: 0.734677	loss: 0.781537	loss: 0.529922	loss: 0.53052
epoch: 9000	epoch: 9000	epoch: 9000	epoch: 9000	epoch: 9000	epoch: 9000
train_accuracy: 0.81775	train_accuracy: 0.7949	train_accuracy: 0.7002	train_accuracy: 0.62895	train_accuracy: 0.787275	train_accuracy: 0.7891
loss: 0.45714	loss: 0.517804	loss: 0.722715	loss: 0.773301	loss: 0.52756	loss: 0.527091

Conclusions

In this project I implemented a distributed neural network utilizing C++, Eigen, and MPI. I demonstrate that training the model across multiple MPI processes yields a significant speed up while maintaining the quality of the resulting model measured using testing accuracy. I performed experiments and measured the execution time of training with a number of processes p = 1, 8, 16, 32, 48, and 64 while keeping the problem size constant to assess scalability. The difference in execution time between serial and parallel is very significant - I was able to decrease the training time from 791 seconds (serial) to only 19 seconds (parallel with 64 processes), a speed up of a factor of 42. Additionally, doubling the number of processors achieved a speed up by almost a factor of 2 each time. Neural network training is very computationally intensive and can take a long time especially when training on a very large dataset, which is why it is a great use case for data parallelization via MPI.

References

- Pacheco, P. (2011). An Introduction to Parallel Programming. 1-370.
- Dean, Jeffrey & Corrado, G.s & Monga, Rajat & Chen, Kai & Devin, Matthieu & Le, Quoc & Mao, Mark & Ranzato, Aurelio & Senior, Andrew & Tucker, Paul & Yang, Ke & Ng, Andrew. (2012). Large Scale Distributed Deep Networks. Advances in neural information processing systems.
- Xing, Eric & Ho, Qirong & Dai, Wei & Kim, Jin-Kyu & Wei, Jinliang & Lee, Seunghak & Zheng, Xun & Xie, Pengtao & Kumar, Abhimanu & Yu, Yaoliang. (2015). Petuum: A New Platform for Distributed Machine Learning on Big Data. IEEE Transactions on Big Data. 1. 1-1. 10.1109/TBDATA.2015.2472014.
- Shi, Shaohuai & Wang, TQiang & Chu, Xiaowen & Li, Bo. (2018). A DAG Model of Synchronous Stochastic Gradient Descent in Distributed Deep Learning. 425-432. 10.1109/PADSW.2018.8644932.
- Das, Dipankar & Avancha, Sasikanth & Mudigere, Dheevatsa & Vaidynathan, Karthikeyan & Sridharan, Srinivas & Kalamkar, Dhiraj & Kaul, Bharat & Dubey, Pradeep. (2016). Distributed Deep Learning Using Synchronous Stochastic Gradient Descent.
- https://eigen.tuxfamily.org/index.php?title=Main_Page
- https://en.cppreference.com/w/
- https://ai.googleblog.com/2019/03/measuring-limits-of-data-parallel.html
- https://medium.com/@coviamtech/distributed-training-in-deep-learning-models-fa0a1e75eb68
- http://seba1511.net/dist_blog/
- https://hpc-wiki.info/hpc/Load_Balancinghttps://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-map
- https://blogs.nvidia.com/blog/2019/11/15/whats-the-difference-between-single-double-multi-and-mixed-precision-computing/
- https://www.cs.ubc.ca/labs/lci/mlrg/slides/MLRG_Synchronous_Stochastic_Gradient.pdf

Programs

Link to my codebase on github for this project:

Serial code: https://github.com/ekloberdanz/Neural-Net-Implementation/tree/serial

Parallel code: https://github.com/ekloberdanz/Neural-Net-Implementation/tree/parallel

(1) main.cpp

```
#include "NeuralNet.hpp"
#include <string>
#include <fstream>
#include <iostream>
#include <boost/range/irange.hpp>
#include <typeinfo>
#include <mpi.h>
#include <stdio.h>
#include <chrono>

int main() {
    // MPI initialization
    int rank, comm_sz;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    // temporary directory on cluster
    //char const* tmp = getenv("TMPDIR");
    //std::string TMPDIR(tmp);

    // all
    // Dataset
    Eigen::MatrixXd X_train;
    Eigen::MatrixXd X_train_subset;
    Eigen::VectorXi y_train;
    Eigen::VectorXi y_train_subset;
    Eigen::MatrixXd X_test;
    Eigen::VectorXi y_test;
    int data_total_size;
    int data_subset_size;

    // all
    // Parameters
    int NUM_CLASSES = 5;
    double start_learning_rate = 1.0;


    std::cout << "number of processes : " << comm_sz << std::endl;
```

```
// all
// Create for neural network objects
LayerDense dense_layer_1(2, 64);
ActivationRelu activation_relu;
LayerDense dense_layer_2(64, NUM_CLASSES);
ActivationSoftmax activation_softmax;
CrossEntropyLoss loss_categorical_crossentropy;
StochasticGradientDescent optimizer_SGD(1.0, 1e-3, 0.9);

// variables
double loss;
double train_accuracy;
double test_accuracy;
double pred;
int index_pred;

Eigen::MatrixXd weights_1_sum(dense_layer_1.weights.rows(), dense_layer_1.weights.cols());
Eigen::MatrixXd weights_2_sum(dense_layer_2.weights.rows(), dense_layer_2.weights.cols());
Eigen::VectorXd biases_1_sum(dense_layer_1.biases.rows(), dense_layer_1.biases.cols());
Eigen::VectorXd biases_2_sum(dense_layer_2.biases.rows(), dense_layer_2.biases.cols());

Eigen::MatrixXd weights_2_new(dense_layer_2.weights.rows(), dense_layer_2.weights.cols());
Eigen::VectorXd biases_1_new(dense_layer_1.biases.rows(), dense_layer_1.biases.cols());
Eigen::VectorXd biases_2_new(dense_layer_2.biases.rows(), dense_layer_2.biases.cols());

// Load training and testing data from the file, train data is pre-shuffled
X_train = load_matrix_data("/home/eklober/X_train_large.csv");
y_train = load_vector_data("/home/eklober/y_train_large.csv");
X_test = load_matrix_data("/home/eklober/X_test_large.csv");
y_test = load_vector_data("/home/eklober/y_test_large.csv");

std::cout << "X_train shape : " << X_train.rows() << "x" << X_train.cols() << std::endl;
std::cout << "X_train shape : " << X_train.rows() << "x" << X_train.cols() << std::endl;

data_total_size = X_train.rows(); // total number of train data points
data_subset_size = data_total_size/(comm_sz-1);

MPI_Barrier(MPI_COMM_WORLD); // wait for all workers to initialize neural net objects

// Train DNN
double time_start = MPI_Wtime();
int NUMBER_OF_EPOCHS = 10000;
for (int epoch : boost::irange(0,NUMBER_OF_EPOCHS)) {
    if (rank != 0) {
        X_train_subset = X_train.block((rank-1)*data_subset_size, 0, data_subset_size, X_train.cols());
        y_train_subset = y_train.segment((rank-1)*data_subset_size, data_subset_size);

       ////////////////////////////////////////forward pass////////////////////////////////////////////////////////////////////////////////
        dense_layer_1.forward(X_train_subset);
        activation_relu.forward(dense_layer_1.output);
        dense_layer_2.forward(activation_relu.output);
        activation_softmax.forward(dense_layer_2.output);

       ////////////////////////////////////////////backward pass/////////////////////////////////////////////////////////////////////////////
```

```
            loss_categorical_crossentropy.backward(activation_softmax.output, y_train_subset);
            activation_softmax.backward(loss_categorical_crossentropy.dinputs);
            dense_layer_2.backward(activation_softmax.dinputs);
            activation_relu.backward(dense_layer_2.dinputs);
            dense_layer_1.backward(activation_relu.dinputs);

        ////////////////////////////////optimizer - update weights and biases/////////////////////////////////////////////////////
            optimizer_SGD.pre_update_params(start_learning_rate);
            optimizer_SGD.update_params(dense_layer_1);
            optimizer_SGD.update_params(dense_layer_2);
            optimizer_SGD.post_update_params();

            // workers send  parameters to master, who uses them to compute a sum to compute average
            MPI_Send(dense_layer_1.weights.data(), dense_layer_1.weights.rows() *
dense_layer_1.weights.cols(), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(dense_layer_2.weights.data(), dense_layer_2.weights.rows() *
dense_layer_2.weights.cols(), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(dense_layer_1.biases.data(), dense_layer_1.biases.rows() * dense_layer_1.biases.cols(),
MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(dense_layer_2.biases.data(), dense_layer_2.biases.rows() * dense_layer_2.biases.cols(),
MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }

        MPI_Barrier(MPI_COMM_WORLD); // wait for all workers to compute weights and biases

        if (rank == 0) {
            weights_1_sum = Eigen::MatrixXd::Zero(dense_layer_1.weights.rows(), dense_layer_1.weights.cols());
            weights_2_sum = Eigen::MatrixXd::Zero(dense_layer_2.weights.rows(),
dense_layer_2.weights.cols());
            biases_1_sum = Eigen::VectorXd::Zero(dense_layer_1.biases.rows(), dense_layer_1.biases.cols());
            biases_2_sum = Eigen::VectorXd::Zero(dense_layer_2.biases.rows(), dense_layer_2.biases.cols());

            for (int p=1; p <= comm_sz-1; p++) {
                MPI_Recv(dense_layer_1.weights.data(), dense_layer_1.weights.rows() *
dense_layer_1.weights.cols(), MPI_DOUBLE, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(dense_layer_2.weights.data(), dense_layer_2.weights.rows() *
dense_layer_2.weights.cols(), MPI_DOUBLE, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(dense_layer_1.biases.data(), dense_layer_1.biases.rows() * dense_layer_1.biases.cols(),
MPI_DOUBLE, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(dense_layer_2.biases.data(), dense_layer_2.biases.rows() *
dense_layer_2.biases.cols(), MPI_DOUBLE, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                weights_1_sum = weights_1_sum + dense_layer_1.weights;
                weights_2_sum = weights_2_sum + dense_layer_2.weights;
                biases_1_sum = biases_1_sum + dense_layer_1.biases;
                biases_2_sum = biases_2_sum + dense_layer_2.biases;
            }

            // compute average
            dense_layer_1.weights = weights_1_sum/(comm_sz-1);
            dense_layer_2.weights = weights_2_sum/(comm_sz-1);
            dense_layer_1.biases = biases_1_sum/(comm_sz-1);
            dense_layer_2.biases = biases_2_sum/(comm_sz-1);

            // periodically calculate train accuracy and loss
```

```
        if (epoch % 1000 == 0) {
            dense_layer_1.forward(X_train);
            activation_relu.forward(dense_layer_1.output);
            dense_layer_2.forward(activation_relu.output);
            activation_softmax.forward(dense_layer_2.output);
            loss = loss_categorical_crossentropy.calculate(activation_softmax.output, y_train);
            Eigen::MatrixXd::Index maxRow, maxCol;
            Eigen::VectorXi predictions(activation_softmax.output.rows());
            Eigen::VectorXd pred_truth_comparison(activation_softmax.output.rows());

            for (int i=0; i < activation_softmax.output.rows(); i++) {
                pred = activation_softmax.output.row(i).maxCoeff(&maxRow, &maxCol);
                index_pred = maxCol;
                predictions(i) = index_pred;
                pred_truth_comparison(i) = predictions(i) == y_train(i);
            }
            train_accuracy = pred_truth_comparison.mean();
            std::cout << "epoch: " << epoch << std::endl;
            std::cout << "train_accuracy: " << train_accuracy << std::endl;
            std::cout << "loss: " << loss << std::endl;
        }
    }


    // broadcast new weights and biases to workers
    MPI_Bcast(dense_layer_1.weights.data(), dense_layer_1.weights.rows() * dense_layer_1.weights.cols(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(dense_layer_2.weights.data(), dense_layer_2.weights.rows() *
dense_layer_2.weights.cols(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(dense_layer_1.biases.data(), dense_layer_1.biases.rows() * dense_layer_1.biases.cols(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(dense_layer_2.biases.data(), dense_layer_2.biases.rows() * dense_layer_2.biases.cols(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }
  // Time training time
  double time_end = MPI_Wtime();
  double time_delta = time_end - time_start;
  /*compute max, min, and average timing statistics*/
  double time_execution;
  MPI_Reduce(&time_delta, &time_execution, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
  if (rank == 0) {
      std::cout << "\nTraining time in seconds: " << time_execution << std::endl;
  }

   // Test DNN
  if (rank == 0) {
      dense_layer_1.forward(X_test);
      activation_relu.forward(dense_layer_1.output);
      dense_layer_2.forward(activation_relu.output);
      activation_softmax.forward(dense_layer_2.output);
      // calculate loss
      loss = loss_categorical_crossentropy.calculate(activation_softmax.output, y_test);
      // get predictions and accuracy
      Eigen::MatrixXd::Index maxRow, maxCol;
      Eigen::VectorXi predictions(activation_softmax.output.rows());
```

```
        Eigen::VectorXd pred_truth_comparison(activation_softmax.output.rows());

        for (int i=0; i < activation_softmax.output.rows(); i++) {
                pred = activation_softmax.output.row(i).maxCoeff(&maxRow, &maxCol);
                index_pred = maxCol;
                predictions(i) = index_pred;
                pred_truth_comparison(i) = predictions(i) == y_test(i);
        }
        test_accuracy = pred_truth_comparison.mean();
        std::cout << "\ntest_accuracy: " << test_accuracy << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

(2) NeuralNet.cpp

```
#include "NeuralNet.hpp"
#include<fstream>
#include<iostream>
#include <eigen3/Eigen/Dense>

// LayerDense functions definitions
void LayerDense::forward(const Eigen::MatrixXd &inputs) {
    this->inputs = inputs;
    output = (inputs * weights).rowwise() + biases.transpose();
}

void LayerDense::backward(const Eigen::MatrixXd &dvalues) {
    dweights = inputs.transpose() * dvalues;
    dbiases = dvalues.colwise().sum();
    dinputs = dvalues * weights.transpose();
}

// ActivationRelu functions definitions
void ActivationRelu::forward(const Eigen::MatrixXd &inputs) {
    this->inputs = inputs;
    output = (inputs.array() < 0).select(0, inputs);
}

void ActivationRelu::backward(const Eigen::MatrixXd &dvalues) {
    dinputs = (inputs.array() <= 0).select(0, dvalues);
}

// ActivationSoftmax functions definitions
void ActivationSoftmax::forward(const Eigen::MatrixXd &inputs) {
    this->inputs = inputs;
    Eigen::MatrixXd exp_values;
    Eigen::VectorXd max_values;
    Eigen::MatrixXd probabilities;
    max_values = inputs.rowwise().maxCoeff(); // max
    exp_values = (inputs.colwise() - max_values).array().exp(); // unnormalized probabilities
    Eigen::VectorXd sum_exp = exp_values.rowwise().sum();
```

```
    output = (exp_values.array().colwise() / sum_exp.array()); // normalized probabilities
}

void ActivationSoftmax::backward(const Eigen::MatrixXd &dvalues) {
    Eigen::MatrixXd jacobian_matrix;
    Eigen::VectorXd single_output;
    Eigen::VectorXd single_dvalues;
    dinputs = Eigen::MatrixXd:: Zero(dvalues.rows(),dvalues.cols());
    Eigen::MatrixXd single_output_one_hot_encoded;
    int labels = dvalues.cols();
    for (int i = 0; i < dvalues.rows(); i++) {
        single_output = output.row(i).transpose();
        single_dvalues = dvalues.row(i);
        single_output_one_hot_encoded = single_output.asDiagonal();
        Eigen::MatrixXd dot_product = single_output * single_output.transpose();
        jacobian_matrix = single_output_one_hot_encoded - dot_product;
        Eigen::VectorXd gradient = jacobian_matrix * single_dvalues;
        dinputs.row(i) = gradient;
    }
}

// CrossEntropyLoss functions definitions
Eigen::VectorXd CrossEntropyLoss::forward(const Eigen::MatrixXd &y_pred, const Eigen::VectorXi &y_true)
{
    int samples = y_true.rows();
    int r;
    int index;
    double conf;
    Eigen::MatrixXd y_pred_clipped;
    Eigen::VectorXd correct_confidences(samples);

    y_pred_clipped = (y_pred.array() < 1e-5).select(1e-5, y_pred);
    y_pred_clipped = (y_pred_clipped.array() > 1 - 1e-5).select(1 - 1e-5, y_pred_clipped);

    for (r=0; r < samples; r++) {
        index = y_true(r);
        conf = y_pred_clipped(r, index);
        correct_confidences(r) = conf;
    }
    Eigen::VectorXd negative_log_likelihoods = - (correct_confidences.array().log());
    return negative_log_likelihoods;
}

void CrossEntropyLoss::backward(const Eigen::MatrixXd &dvalues, const Eigen::VectorXi &y_true) {
    int samples = dvalues.rows();
    int labels = dvalues.cols();
    int index;
    Eigen::MatrixXd y_true_one_hot_encoded;
    y_true_one_hot_encoded = Eigen::MatrixXd::Zero(samples, labels);
    for (int r=0; r < samples; r++) {
        index = y_true(r);
        y_true_one_hot_encoded(r, index) = 1;
    }
    // Calculate gradient
    dinputs = - (y_true_one_hot_encoded.array() / dvalues.array());
```

```
    // Normalize gradient
    dinputs = dinputs * (1/double(samples));
}

double CrossEntropyLoss::calculate(const Eigen::MatrixXd &output, const Eigen::VectorXi &y) {
    Eigen::VectorXd sample_losses = this->forward(output, y);
    double data_loss = sample_losses.mean();
    return data_loss;
}


// SGD functions declaration
void StochasticGradientDescent::pre_update_params(double start_learning_rate) {
    if (decay != 0.0) {
        learning_rate = start_learning_rate * (1.0 / (1.0 + (decay * iterations)));
    }
}

void StochasticGradientDescent::update_params(LayerDense &layer) {
    Eigen::MatrixXd weight_updates;
    Eigen::VectorXd bias_updates;
    if (momentum != 0.0) {
        weight_updates = (momentum * layer.weight_momentums) - (learning_rate * layer.dweights);
        layer.weight_momentums = weight_updates;
        bias_updates = (momentum * layer.bias_momentums).array() - (learning_rate * layer.dbiases).array();
        layer.bias_momentums = bias_updates;
    } else {
        weight_updates = -learning_rate * layer.dweights;
        bias_updates = -learning_rate * layer.dbiases;
    }
    layer.weights = layer.weights + weight_updates;
    layer.biases = layer.biases + bias_updates;
}

void StochasticGradientDescent::post_update_params() {
    iterations += 1;
}

Eigen::MatrixXd load_matrix_data(std::string fileToOpen) {
    // REFERENCE:
https://aleksandarhaber.com/eigen-matrix-library-c-tutorial-saving-and-loading-data-in-from-a-csv-file/
    std::vector<double> matrixEntries;
    std::ifstream matrixDataFile(fileToOpen);
    std::string matrixRowString;
    std::string matrixEntry;
    int matrixRowNumber = 0;
    while (getline(matrixDataFile, matrixRowString))
    {
        std::stringstream matrixRowStringStream(matrixRowString); //convert matrixRowString that is a string to
a stream variable.
        while (getline(matrixRowStringStream, matrixEntry, ',')) // here we read pieces of the stream
matrixRowStringStream until every comma, and store the resulting character into the matrixEntry
        {
            matrixEntries.push_back(stod(matrixEntry));
        }
```

```
        matrixRowNumber++;
    }
    //std::shuffle(std::begin(matrixEntries), std::end(matrixEntries), std::default_random_engine());
    return Eigen::Map<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic,
Eigen::RowMajor>>(matrixEntries.data(), matrixRowNumber, matrixEntries.size() / matrixRowNumber);

}

Eigen::VectorXi load_vector_data(std::string fileToOpen) {
    // REFERENCE:
https://aleksandarhaber.com/eigen-matrix-library-c-tutorial-saving-and-loading-data-in-from-a-csv-file/

    // the inspiration for creating this function was drawn from here (I did NOT copy and paste the code)
    // https://stackoverflow.com/questions/34247057/how-to-read-csv-file-and-assign-to-eigen-matrix
    // the input is the file: "fileToOpen.csv":
    // a,b,c
    // d,e,f
    // This function converts input file data into the Eigen matrix format
    // the matrix entries are stored in this variable row-wise. For example if we have the matrix:
    // M=[a b c
    //    d e f]
    // the entries are stored as matrixEntries=[a,b,c,d,e,f], that is the variable "matrixEntries" is a row vector
    // later on, this vector is mapped into the Eigen matrix format
    std::vector<int> matrixEntries;
    // in this object we store the data from the matrix
    std::ifstream matrixDataFile(fileToOpen);
    // this variable is used to store the row of the matrix that contains commas
    std::string matrixRowString;
    // this variable is used to store the matrix entry;
    std::string matrixEntry;
    // this variable is used to track the number of rows
    int matrixRowNumber = 0;
    while (getline(matrixDataFile, matrixRowString)) // here we read a row by row of matrixDataFile and store
every line into the string variable matrixRowString
    {
        std::stringstream matrixRowStringStream(matrixRowString); //convert matrixRowString that is a string to
a stream variable.
        while (getline(matrixRowStringStream, matrixEntry, ',')) // here we read pieces of the stream
matrixRowStringStream until every comma, and store the resulting character into the matrixEntry
        {
            matrixEntries.push_back(stod(matrixEntry));   //here we convert the string to double and fill in the
row vector storing all the matrix entries
        }
        matrixRowNumber++; //update the column numbers
    }
    //std::shuffle(std::begin(matrixEntries), std::end(matrixEntries), std::default_random_engine());
    // here we convet the vector variable into the matrix and return the resulting object,
    // note that matrixEntries.data() is the pointer to the first memory location at which the entries of the
vector matrixEntries are stored;
    return Eigen::Map<Eigen::Matrix<int, Eigen::Dynamic, Eigen::Dynamic,
Eigen::RowMajor>>(matrixEntries.data(), matrixRowNumber, matrixEntries.size() / matrixRowNumber);

}
```

(3) NeuralNet.hpp

```
#ifndef NEURALNET_HPP
#define NEURALNET_HPP


#include <eigen3/Eigen/Eigen>
#include <iostream>
#include <vector>
#include <string>
#include <boost/range/combine.hpp>
#include <fstream>

class LayerDense {
    public:
        int n_inputs; // number of inputs
        int n_neurons; // number of neurons

        Eigen::MatrixXd inputs; // inputs
        Eigen::MatrixXd weights;
        Eigen::MatrixXd weight_momentums;
        Eigen::VectorXd biases;
        Eigen::VectorXd bias_momentums;
        Eigen::MatrixXd output;

        Eigen::MatrixXd dweights; // derivative wrt weights
        Eigen::VectorXd dbiases; // derivative wrt biases
        Eigen::MatrixXd dinputs; // derivative wrt inputs

        // constructor
        LayerDense(int n_inputs, int n_neurons) {
            // srand(42);
            this->weights = Eigen::MatrixXd::Random(n_inputs,n_neurons) * 0.01; // initialize weights
            this->weights = weights;
            this->biases = Eigen::VectorXd::Zero(n_neurons); // initialize biases
            this->n_inputs = n_inputs;
            this->n_neurons = n_neurons;
            this->weight_momentums = Eigen::MatrixXd::Zero(n_inputs,n_neurons); // weight momentum
            this->bias_momentums = Eigen::VectorXd::Zero(n_neurons); // bias momentum
        }

        // Member functions declaration
        void forward(const Eigen::MatrixXd &inputs);
        void backward(const Eigen::MatrixXd &dvalues);
};

class ActivationRelu {
    public:
        Eigen::MatrixXd inputs; // inputs
        Eigen::MatrixXd dinputs; // derivative wrt inputs
        Eigen::MatrixXd output;

        // Member functions declaration
        void forward(const Eigen::MatrixXd &inputs);
        void backward(const Eigen::MatrixXd &dvalues);
```

```
};


class ActivationSoftmax {
    public:
        Eigen::MatrixXd inputs; // inputs
        Eigen::MatrixXd dinputs; // derivative wrt inputs
        Eigen::MatrixXd output;

        // Member functions declaration
        void forward(const Eigen::MatrixXd &inputs);
        void backward(const Eigen::MatrixXd &dvalues);
};

class CrossEntropyLoss {
    public:
        Eigen::MatrixXd dinputs;

        // Member functions declaration
        Eigen::VectorXd forward(const Eigen::MatrixXd &y_pred, const Eigen::VectorXi &y_true);
        double calculate(const Eigen::MatrixXd &output, const Eigen::VectorXi &y);
        void backward(const Eigen::MatrixXd &dvalues, const Eigen::VectorXi &y_true);
};

class StochasticGradientDescent {
    public:
        double learning_rate; // learning rate
        double decay; // decay
        double momentum; // momentum
        double iterations; // initialize number of iterations to 0

        // constructor
        StochasticGradientDescent(double learning_rate, double decay, double momentum) {
            this->learning_rate = learning_rate;
            this->decay = decay;
            this->momentum = momentum;
            this->iterations = 0.0;
        }

        // Member functions declaration
        void pre_update_params(double start_learning_rate);
        void update_params(LayerDense &layer);
        void post_update_params();
};

Eigen::MatrixXd load_matrix_data(std::string fileToOpen);
Eigen::VectorXi load_vector_data(std::string fileToOpen);

#endif // NEURALNET_HPP
```