Kody Quintana
CS 473
Artificial Neural Networks
May 8, 2019

Final Project

0.1 Given: A system with 3 inputs and 1000 randomly generated instances, x_i's for each input to give a full input set

$$\boldsymbol{X} = [\boldsymbol{x_0},\ \boldsymbol{x_1},\ \boldsymbol{x_2}]$$

where:

$$\boldsymbol{x_1} = [\boldsymbol{x_0},\ \boldsymbol{x_1},\ \boldsymbol{x_3},\ \ldots\ \boldsymbol{x_{999}},\ \boldsymbol{1}]$$

$\boldsymbol{x_2}$ and $\boldsymbol{x_3}$ are similar. A hidden layer of four nodes:

$$\boldsymbol{h} = [\boldsymbol{h_0},\ \boldsymbol{h_1},\ \boldsymbol{h_2},\ \boldsymbol{h_3}]$$

and output labels:

$$\boldsymbol{Y_0} = [\boldsymbol{y_0},\ \boldsymbol{y_1},\ \ldots\ \boldsymbol{y_{999}}]$$
$$\boldsymbol{Y_1} = [\boldsymbol{y_0},\ \boldsymbol{y_1},\ \ldots\ \boldsymbol{y_{999}}]$$

Find:

1. The hyper-dimensional linear solution to the system utilizing a single hidden layer model with forward and backward propagation with the above parameters utilizing a sigmoid activation function.
2. Discuss your final weight matrices, $\boldsymbol{W_{hi}}$, and $\boldsymbol{W_{oh}}$.
3. Plot the MSE as a function of epochs.

Hint: for part b remember to normalize your labels between $\boldsymbol{0}$ and $\boldsymbol{1}$. In fact, it might be wise to one hot encode the label data to values of $\boldsymbol{0}$ or $\boldsymbol{1}$.

0.2 Extra Credit: (50 pts.) Solve the above problem with an additional hidden layer, $\boldsymbol{h_2}$ with three hidden nodes.

This implementation supports a neural network of any number of layers and any number of nodes per layer. Larger configurations are very slow however because it is single-threaded. It's layout is stored as a multi-linked list. This structure is probably less performant because of all the required pointer dereferencing, but it is the most simple way I could think of to implement generalized back-propagation.

The neural network consists of 3 major components totalling 637 lines.

1. Weight Matrix class - this class is a functor that stores all of the weights in contiguous memory. Each weight is accessed with 3 arguments: the layer, the current node within this layer, and node position from the previous layer.
2. Node Vector class - this class stores a node struct called NN_Node in contiguous memory for each node of the neural network. The range of nodes per layer is stored in a vector.
3. Neural Network class - this class contains the previous two classes as members. Most of the logic of the neural network is done by this class's methods.

Headers are shown in blue and source files are shown in grey.

WeightMat.hpp

```
#ifndef KWEIGHTMATRIX
#define KWEIGHTMATRIX
#include <vector>

using std::vector;

class WeightMatrix{
   private:
      const vector<int> layer_sizes;
      const int layers;
      const vector<int> L;
      vector<double> W;
   public:
      WeightMatrix( std::vector<int> sizes );
      double& operator()(int a, int b, int c);
      int index(int a, int b, int c);
      void print_all();
};
#endif
```

WeightMat.cpp

```
#include "WeightMat.hpp"

#include <iostream>
```

```
#include <vector>
#include <algorithm>
#include <random>

using namespace std;

WeightMatrix::WeightMatrix( std::vector<int> sizes ) :

   layer_sizes( sizes ),

   layers( sizes.size() ),

   //L is used to translate the first arg of WeightMatrix(X,X,X)
   // to the appropriate offset because each set of matrices
   // is a different size
   L( [&]() -> std::vector<int> {
      std::vector<int> t(layers-1, 0);
      for (int i = 1; i < layers-1; ++i){
         int this_mat_size = layer_sizes[i-1]*layer_sizes[i];
         for (int j = i; j < layers-1; ++j){
            t[j] += this_mat_size;
         };
      };
      //cout << "Weight L offsets:\n";
      //for (auto i : t){ cout << "  " << i << endl;};
      return t; //Initializes const L to this t.
      }()

   ),

   //W is all weights stored in a single vector
   W( [&]() -> std::vector<double> {
      srand(time(NULL));
      int full_length = 0;
      for (int i = 1; i < layers; ++i){
         full_length += layer_sizes[i-1]*layer_sizes[i];
      };
      std::vector<double> rw;
      rw.reserve(full_length);
      //Fill with random between 0 and 1
      std::generate_n(std::back_inserter(rw), full_length,
         []() -> double{
            return rand() * (1.0/RAND_MAX);
         }
      );
      //for (auto i : rw ){ cout << i << endl; };
      return rw;
      }()
   )
   {
```

```
      //Constructor body
   }


double& WeightMatrix::operator()(int a, int b, int c){
   //a-1 because forward propigation starts at layer 1
   //and looks "back" to sum nodes
   //WeightMatrix(0,X,X) should never be called
   return W[ L[a-1] + layer_sizes[a-1]*b + c];
}


int WeightMatrix::index(int a, int b, int c){
   return L[a-1] + layer_sizes[a-1]*b + c;
}


void WeightMatrix::print_all(){
   for (auto i : W){cout << i << endl;};
}
```

NodeVec.hpp

```
#ifndef KNODEVECTOR
#define KNODEVECTOR

#include <iostream>
#include <vector>

using namespace std;

struct NN_Node{
   const int layer;
   const int l_node;
   vector<NN_Node*> next_paths;
   vector<NN_Node*> prev_paths;

   //Set to 1.0 incase this node gets used as a bias node
   //if it isnt, the value will get overwritten on first f prop.
   double value = 1.0;

   //Reserve space for vectors on construction, set layer and l_node
   //next_paths, and prev_paths are not filled yet
   //if you don't want full connectivity
   NN_Node(int this_layer, int this_node, int next_size, int prev_size);
};

class NodeVector{
```

```cpp
    typedef std::pair<int, int> Range;
    friend class NeuralNet;
    private:
       const vector<int> n_per_layer; //Count of nodes per layer, not including the bias nodes
       const int layers; //Number of layers
       const vector<int> n_per_layer_bias; //Count of nodes, inclusing the bias nodes
       const int full_size; //Total number of nodes
       const vector<Range> n_indices; //Ranges for each layer, not including the bias node
       const vector<Range> n_indices_bias; //Ranges for each layer, with the bias nodes
       const int possible_path_size;
       vector<NN_Node> Nodes; //Single vector of all nodes from all layers
       void set_connectivity(); //Hard-coded to full connectivity
    public:
       NodeVector(vector<int> layer_sizes);
};
#endif
```

NodeVec.cpp

```cpp
#include "NodeVec.hpp"

#include <iostream>
#include <vector>

using namespace std;

NN_Node::NN_Node(int this_layer, int this_node, int next_size, int prev_size) :

   layer(this_layer),

   l_node(this_node),

   next_paths( [=]()->vector<NN_Node*> {
         vector<NN_Node*> nvec;
         nvec.reserve(next_size);
         return nvec;
         }()
        ),

   prev_paths( [=]()->vector<NN_Node*> {
         vector<NN_Node*> pvec;
         pvec.reserve(prev_size);
         return pvec;
         }()
        )
{
   //Constructor body
}
```

```cpp


NodeVector::NodeVector(vector<int> layer_sizes) :

   n_per_layer( layer_sizes ),

   layers( n_per_layer.size() ),

   n_per_layer_bias( [&]()->vector<int>{
         vector<int> n_per_lb;
         n_per_lb.reserve(layers);
         for (int i = 0; i < layers-1; ++i){
            n_per_lb.emplace_back(n_per_layer[i]+1);
         }
         n_per_lb.emplace_back(n_per_layer.back());
         return n_per_lb;
         }()
      ),

   full_size( [&]()->int {
         int fsize = 0;
         for (auto i : layer_sizes){
            fsize += i;
         }
         //To account for the extra bias nodes:
         fsize += (layers - 1);
         return fsize;
         }()
      ),
   n_indices( [&]()->vector<Range> {
         vector<Range> n_ind;
         n_ind.reserve(layers);
         int start = 0;
         for (int layer = 0; layer < layers; ++layer){
            n_ind.emplace_back( make_pair(start, start + layer_sizes[layer]-0) );
            start += layer_sizes[layer]+1;
         }
         cout << "\nn_indices:\n";
         for (auto i : n_ind) { cout << i.first << "-" << i.second << endl;}
         return n_ind;
         }()
      ),

   n_indices_bias( [&]()->vector<Range> {
         vector<Range> n_ind_b;
         n_ind_b.reserve(layers);
         int start = 0;
         for (int layer = 0; layer < layers-1; ++layer){
            n_ind_b.emplace_back( make_pair(start, start + layer_sizes[layer]+1) );
            start += layer_sizes[layer]+1;
```

```cpp
         }
         //There is no bias node on last layer:
         n_ind_b.emplace_back( make_pair(start, start + layer_sizes[layers-1]) );
         cout << "\nn_indices_bias:\n";
         for (auto i : n_ind_b) { cout << i.first << "-" << i.second << endl;}
         return n_ind_b;
         }()
      ),

   possible_path_size( [&]()->int{
         int result = 1;
         for (long unsigned int i = 1; i < n_per_layer_bias.size(); ++i){
            result *= n_per_layer_bias[i];
         }
         return result;
         }()
      ),


   Nodes( [&]()->vector<NN_Node> {
         vector<NN_Node> node_vec;
         node_vec.reserve(full_size);
         for (int layer = 0; layer < layers; ++layer){
            int node_pos = 0;
            for (int n_count = n_indices_bias[layer].first; n_count < n_indices_bias[layer].second; ++n_count){
               node_vec.emplace_back(
                  NN_Node({
                     layer,
                     node_pos++,
                     (layer + 1 < layers) ? n_per_layer_bias[layer+1] : 0,
                     (layer - 1 >= 0) ? n_per_layer_bias[layer-1] : 0
                  })
               );
            }
         }
         return node_vec;
         }()
      )
{
   //Constructor body
   set_connectivity();
}


//This has unnessesary complexity but would be the general form if you didn't actually want full connectivity.
void NodeVector::set_connectivity(){
   for (int layer = 0; layer < layers; ++layer){
      for (int node = n_indices_bias[layer].first; node < n_indices_bias[layer].second; ++node){
```

```cpp
        if (layer > 0){
            for (int p_node = n_indices_bias[layer-1].first; p_node <
            ↪  n_indices_bias[layer-1].second; ++p_node){
                Nodes[node].prev_paths.push_back( &Nodes[p_node] );
            }
        }
        if (layer < layers-1){
            for (int n_node = n_indices_bias[layer+1].first; n_node <
            ↪  n_indices_bias[layer+1].second; ++n_node){
                Nodes[node].next_paths.push_back( &Nodes[n_node] );
            }
        }
    }
  }
}
```

NeuralNet.hpp

```cpp
#ifndef KNEURALNET
#define KNEURALNET

#include "NodeVec.hpp"
#include "WeightMat.hpp"

#include <vector>


using std::vector;

struct Gradient{
   double value;
   int current_layer;
   int current_node_pos;
   vector<NN_Node*>* next_paths;
   Gradient(int layer, int node, double start_value, vector<NN_Node*>* ready_next);
};

class NeuralNet{
   private:
      NodeVector NV;
      WeightMatrix W;

      static double activation(double sum);
      static double dx_activation(double activated_sum);
   public:
      NeuralNet(vector<int> layer_sizes, vector<double>& label_ref, vector<double>& input_ref,
       ↪  double learn_rate, int n_instances);

```

```
        void forward_prop();
        void backward_prop();

        void stage_inputs_and_labels(); //Set first layer node values to this instance of inputs
        void stage_labels();

        int instance_index = 0;
        vector<double>& labels;
        vector<double>& inputs;

        vector<double> error;
        vector<double> instance_label;

        double rmse;
        const double learning_rate;
        const int instance_size;


        void train();
};
#endif
```

NeuralNet.cpp

```
#include "WeightMat.hpp"
#include "NodeVec.hpp"
#include "NeuralNet.hpp"

#include <vector>
#include <iostream>
#include <cmath>
#include <numeric>
#include <fstream>
#include <algorithm>

using std::vector;
using std::cout;
using std::endl;


NeuralNet::NeuralNet(vector<int> layer_sizes, vector<double>& label_ref, vector<double>&
 ↪ input_ref, double learn_rate, int n_instances) :
   NV(layer_sizes),
   W(NV.n_per_layer_bias),
   labels(label_ref),
   inputs(input_ref),
   error( vector<double>(layer_sizes.back(),0.0) ),
   instance_label( vector<double>(layer_sizes.back(),0.0) ),
```

```
    learning_rate(learn_rate),
    instance_size(n_instances)
{
    //Constructor body
    //cout << "nodes per layer with bias nodes:" << endl;
    //for (auto n : NV.n_per_layer_bias){
    //    cout << "  " << n << endl;
    //}
    for (auto const &node : NV.Nodes){
        cout << "\n";
        cout << "Node at layer: " << node.layer << ", pos: " << node.l_node;
        cout << ", has " << endl;
        for (auto const &nn : node.next_paths){
            cout << "     - Next node at layer: " << nn->layer << ", pos: " << nn->l_node;
            cout << ", through W: " << W.index(nn->layer, nn->l_node, node.l_node) << endl;
        }
        for (auto const &pn : node.prev_paths){
            cout << "    -  Prev node at layer: " << pn->layer << ", pos: " << pn->l_node;
            cout << ", from W: " << W.index(node.layer, node.l_node, pn->l_node) << endl;
        }
    }
}


Gradient::Gradient(int layer, int node, double start_value, vector<NN_Node*>* ready_next) :
    value(start_value), current_layer(layer), current_node_pos(node), next_paths(ready_next)
{
    //Constructor body
    //cout << "Gradient constructed with starting value of: " << value << endl;
}


void NeuralNet::stage_inputs_and_labels(){
    for (int node = 0; node < NV.n_indices[0].second; ++node){
        NV.Nodes[node].value = inputs[ NV.n_per_layer[0] * instance_index + node ];
    }
    for (int node = 0; node < NV.n_per_layer.back(); ++node){
        instance_label[node] = labels[ NV.n_per_layer.back() * instance_index + node ];
    }

    if ( (instance_index+1) * NV.n_per_layer[0] == inputs.size() ) { instance_index = 0; }
    else { ++instance_index; }
    double total = 0.0;
    for (auto i : error){ total += pow(i,2); }
    total /= error.size();
    rmse = sqrt(total);
}


double NeuralNet::activation(double sum){
```

```
    return 1.0 / (1.0 + exp(-sum));
}


double NeuralNet::dx_activation(double activated_sum){
    return activated_sum * (1.0 - activated_sum);
}


void NeuralNet::backward_prop(){
    for (int layer = 0; layer < NV.layers; ++layer){
        for (int node = NV.n_indices_bias[layer].first; node < NV.n_indices_bias[layer].second; ++node){
            for (auto const &nn : NV.Nodes[node].next_paths){
                //cout << "Starting back prop to node at layer: " << nn->layer << ", pos: " << nn->l_node;
                //cout << ", through W: " << W.index(nn->layer, nn->l_node, NV.Nodes[node].l_node) << endl;
                vector<Gradient> branch_storage;
                branch_storage.reserve(NV.possible_path_size);
                branch_storage.emplace_back(Gradient(
                            nn->layer,
                            nn->l_node,
                            (NV.Nodes[node].value * dx_activation(nn->value)),
                            &nn->next_paths)
                            );
                long unsigned int bs_index = 0;
                while (bs_index < branch_storage.size()){
                    if (branch_storage[bs_index].next_paths->size() > 1){
                        //When traversing towards the output, here the partial derivative has more than one branch.
                        //To account for each branch, the current derivative is copied for each possible next step
                        //Each copy is assigned one of the possible forward paths
                        //During this assignment the copy's gradient is updated
                        //with the partial from its current node to its assigned node.
                        //This movement has two parts multiplied with the chain rule
                        // 1) the weight between each node
                        // 2) the d/dx of the assigned node's activation function
                        for (long unsigned int n = 1; n < branch_storage[bs_index].next_paths->size(); ++n){
                            branch_storage.emplace_back(branch_storage[bs_index]);
                            int next_node_pos = (*branch_storage[bs_index].next_paths) [n]->l_node;
                            branch_storage.back().next_paths = &(*branch_storage[bs_index].next_paths) [n]->next_paths;
                            branch_storage.back().value *= (
                                W(
                                    (*branch_storage[bs_index].next_paths) [n]->layer,
                                    (*branch_storage[bs_index].next_paths) [n]->l_node,
                                    branch_storage[bs_index].current_node_pos
```

```
            ) * dx_activation( (*branch_storage[bs_index].next_paths)[n]->value )
        );
        branch_storage.back().current_node_pos = next_node_pos;
    }
    Gradient temp = branch_storage[bs_index];
    int next_node_pos = (*branch_storage[bs_index].next_paths) [0]->l_node;
    temp.next_paths = &(*branch_storage[bs_index].next_paths) [0]->next_paths;
    temp.value *= (
        W(
            (*branch_storage[bs_index].next_paths) [0]->layer,
            (*branch_storage[bs_index].next_paths) [0]->l_node,
            branch_storage[bs_index].current_node_pos
        ) * dx_activation( (*branch_storage[bs_index].next_paths)[0]->value )

    );
    temp.current_node_pos = next_node_pos;
    branch_storage[bs_index] = temp;
}
else if (branch_storage[bs_index].next_paths->size() == 1){
    //Similar to above, this case handles when there is only one possible path moving forward
    Gradient temp = branch_storage[bs_index];
    int next_node_pos = (*branch_storage[bs_index].next_paths) [0]->l_node;
    temp.next_paths = &(*branch_storage[bs_index].next_paths)[0]->next_paths;
    temp.value *= (
        W(
            (*branch_storage[bs_index].next_paths)[0]->layer,
            (*branch_storage[bs_index].next_paths)[0]->l_node,
            branch_storage[bs_index].current_node_pos
        ) * dx_activation( (*branch_storage[bs_index].next_paths)[0]->value )
    );
    temp.current_node_pos = next_node_pos;
    branch_storage[bs_index] = temp;
}
else{
    //Finally this branch has reached an output node and there is no more
    //nodes to traverse, here this branch's derivative is updated with
    //the last d/dx which is the partial of this output node with respect to
    //the total Error
    //Note: at this point only this single branch has completed its traversal,
    //now the branch_storage index will be incremented and the next branch will
    //continue traversing until an output node is reached
    //Once all branches have finished they will be summed to create the gradient

    //No need to multply by -1 here, instead just add the gradient, instead of subtracting
    branch_storage[bs_index].value *= error[branch_storage[bs_index].current_node_pos];

    ////If you want to only adjust with the first output node.
```

```
                  //if (branch_storage[bs_index].current_node_pos != 0){
                  //   branch_storage[bs_index].value = 0.0;
                  //}

                  ++bs_index;
               }
            }
            //End while
            double branch_sum = std::accumulate(begin(branch_storage), end(branch_storage), 0.0,
                  [](double incoming, const Gradient& grad) {return grad.value + incoming; }
                  );
            //cout << "Branch sum: " << std::scientific << branch_sum << endl;
            //cout << "  Updating W:" << W.index(nn->layer, nn->l_node, NV.Nodes[node].l_node) << endl;
            //cout << "    from: " << W(nn->layer, nn->l_node, NV.Nodes[node].l_node) << endl;
            W(nn->layer, nn->l_node, NV.Nodes[node].l_node) += (learning_rate * branch_sum);
            //cout << "      to: " << W(nn->layer, nn->l_node, NV.Nodes[node].l_node) << endl;

         }
      }
   }
}


void NeuralNet::forward_prop(){
   for (int layer = 1; layer < NV.layers; ++layer){
      //Update all nodes except bias nodes which will always be 1.0
      for (int node = NV.n_indices[layer].first; node < NV.n_indices[layer].second; ++node){
         //cout << NV.Nodes[node].value << " -> ";
         NV.Nodes[node].value = 0.0;
         for (const auto p_node : NV.Nodes[node].prev_paths){
            NV.Nodes[node].value +=
               p_node->value * W(NV.Nodes[node].layer, NV.Nodes[node].l_node, p_node->l_node);
         }
         //cout << "A(" << NV.Nodes[node].value << ")";
         NV.Nodes[node].value = activation(NV.Nodes[node].value);
         //cout << std::fixed << " -> " << NV.Nodes[node].value << endl;
      }
      //cout << "\n";
   }
   for (int node = NV.n_indices.back().first; node < NV.n_indices.back().second; ++node){
      //cout << "Output: " << NV.Nodes[node].l_node << ", value: " << NV.Nodes[node].value << endl;
      error[ NV.Nodes[node].l_node ] = (/*Label*/instance_label[NV.Nodes[node].l_node]-/*Prediction*/NV.Nodes[node].value);
      //cout << "Error: " << error[ NV.Nodes[node].l_node ] << endl;
   }
   //for (const auto &node : NV.Nodes){ cout << node.value << endl;}
}
```

```
void NeuralNet::train(){
   cout << "\n";

   vector<double> rmse_dat;
   rmse_dat.reserve(2000000);

   stage_inputs_and_labels();
   forward_prop();
   backward_prop();

   stage_inputs_and_labels();
   forward_prop();
   backward_prop();
   rmse_dat.emplace_back(rmse);
   rmse_dat.emplace_back(rmse);

   int iteration = 1;
   //while (rmse > 0.002){
   while (iteration < 800000){
      stage_inputs_and_labels();
      forward_prop();
      backward_prop();
      cout << "RMSE: " << rmse << " Iteration: " << iteration++ << "\r";
      rmse_dat.emplace_back(rmse);

   }
   cout << endl;
   W.print_all();

   ofstream rmse_log;
   rmse_log.open ("./ex4_rmse.dat");
   for (auto r : rmse_dat) { rmse_log << r << "\n"; }

   //ofstream rmse_log2;
   //rmse_log2.open ("./rmse2_5.dat");
   //for (int i = 0; i < rmse_dat.size(); i += instance_size){
   //   std::sort (rmse_dat.begin()+i, rmse_dat.begin()+i+instance_size);
   //}
   //for (int i = 0; i < 10000; ++i){
   //   rmse_log << rmse_dat[i] << "\n";
   //}
   //for (int i = rmse_dat.size() - (10 * 1000); i < rmse_dat.size(); ++i){
   //   rmse_log2 << rmse_dat[i] << "\n";
   //}
   rmse_log.close();
}
```

main.cpp

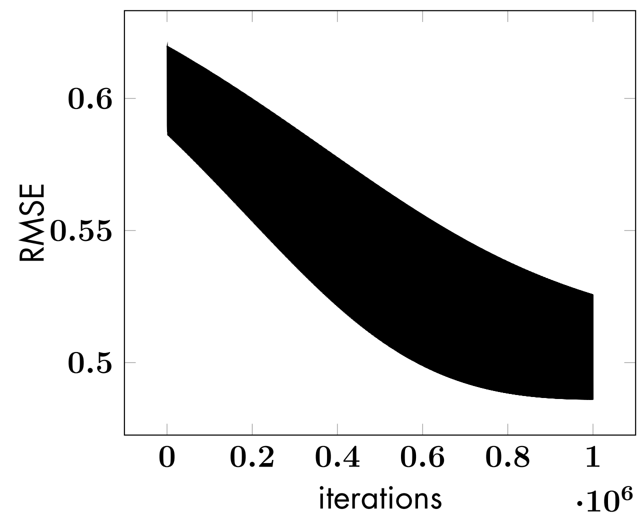
```
#include "WeightMat.hpp"
#include "NodeVec.hpp"
#include "NeuralNet.hpp"

#include <iostream>
#include <algorithm>
#include <random>
#include <iomanip>

int main( int argc, char** argv ){

   cout << std::fixed << std::setprecision(10);

   const int INSTANCE_SIZE = 1000;
   const int INPUT_SIZE = 3;
   const int LABEL_SIZE = 2;

   //Generate INPUTS
   vector<double> INPUTS;
   srand(time(NULL));
   std::generate_n(std::back_inserter(INPUTS), INSTANCE_SIZE * INPUT_SIZE,
      [=]() -> double{
         return rand() * (1.0/RAND_MAX);
      }
   );

   ////Print INPUTS
   //for (long unsigned int i = 0; i < INPUTS.size() - (INPUT_SIZE - 1); i+=INPUT_SIZE){
   //   for (int n = 0; n < INPUT_SIZE; ++n){
   //      cout << INPUTS[i + n] << " ";
   //   }
   //   cout << endl;
   //}

   //Generate LABELS
   vector<double> LABELS(INSTANCE_SIZE * LABEL_SIZE, 0.0);
   std::random_device rd; // obtain a random number from hardware
   std::mt19937 eng(rd()); // seed the generator
   std::uniform_int_distribution<> label_distr(0, LABEL_SIZE - 1); // define the range
   for (long unsigned int i = 0; i < LABELS.size() - (LABEL_SIZE - 1); i+=LABEL_SIZE){
      LABELS[i + label_distr(eng)] = 1.0;
   }

   ////Print LABELS
   //for (long unsigned int i = 0; i < LABELS.size() - (LABEL_SIZE - 1); i+=LABEL_SIZE){
   //   for (int n = 0; n < LABEL_SIZE; ++n){
   //      cout << LABELS[i + n] << " ";
   //   }
```

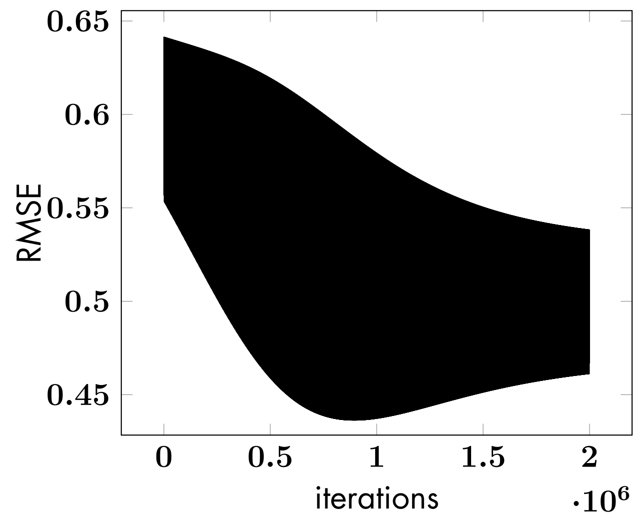
```
    //   cout << endl;
    //}

    NeuralNet project( {INPUT_SIZE, 4, 3, LABEL_SIZE}, LABELS, INPUTS, 0.00001, INSTANCE_SIZE );
    project.train();

    return(0);
}
```

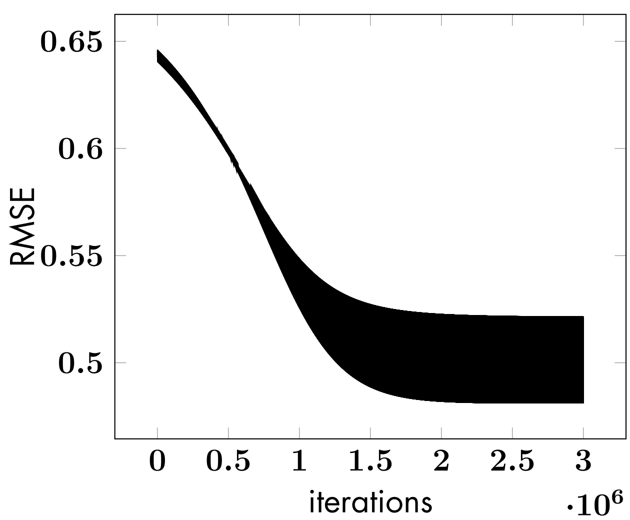
Results

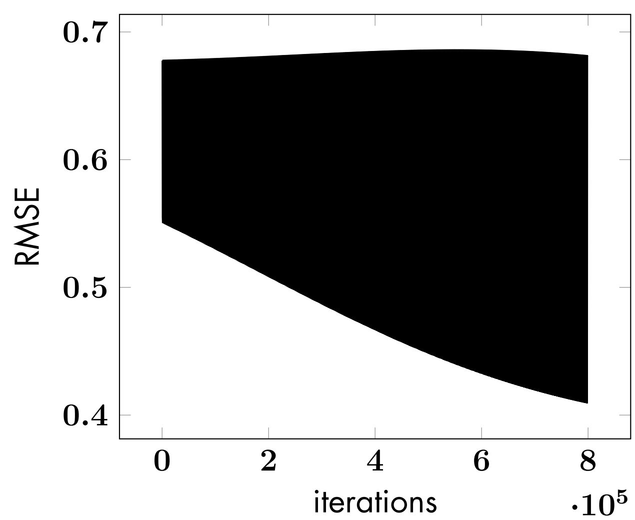
As the neural net trains the RMSE of the error function oscillates between the best case and the worst case error.

Here are some examples plotted with downsampled data from a neural network of three inputs, a hidden layer with four nodes, a second hidden layer with three nodes, and an output layer of two outputs.



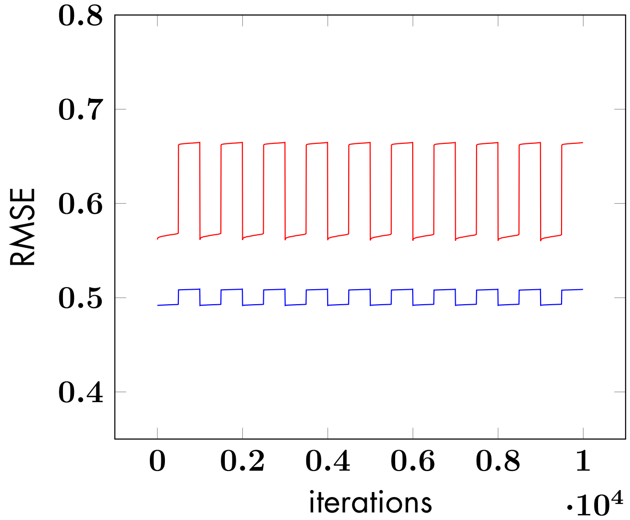




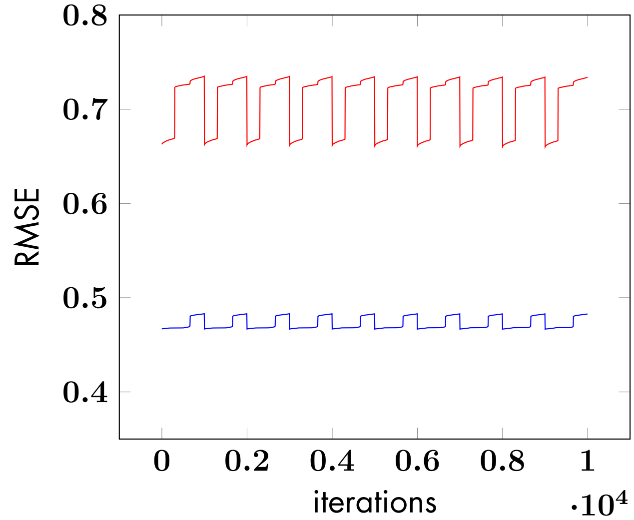


Here are the first 10,000 and last 10,000 out of 2,000,000 iterations. The errors of each instance (of size 1000) have been sorted to show the first and last 10 epochs.

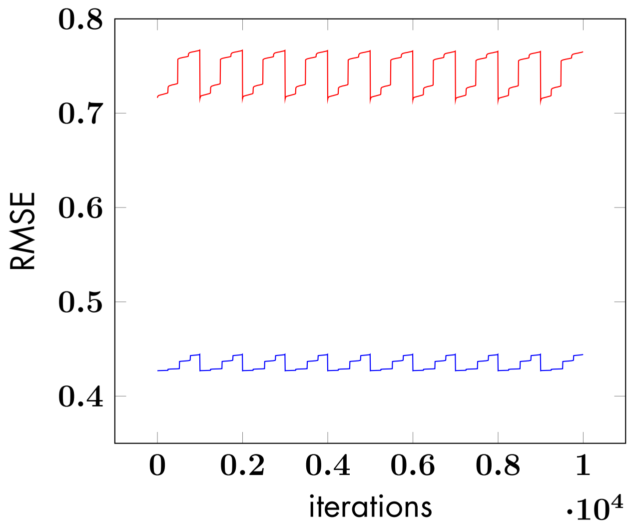
2 random outputs



3 random outputs



4 random outputs



5 random outputs