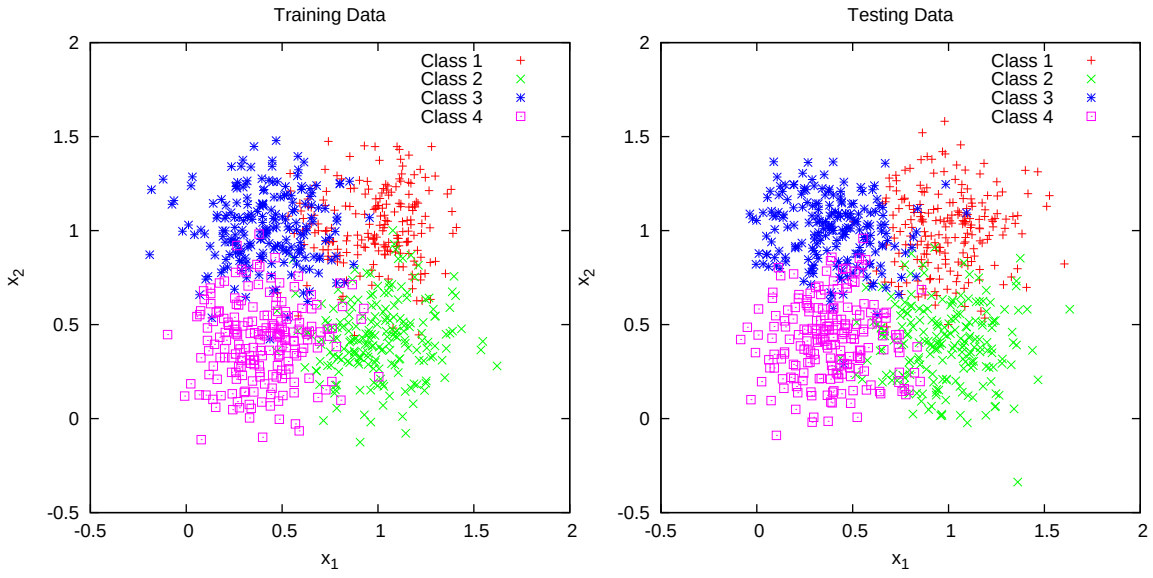
CS547 Homework 5

Bryan Topp

October 17, 2014

1 Scatter Plots

The training and testing data are plotted in Figure 1. The data is not linearly separable in either case, although they appear to imply roughly linear decision boundaries. It appears that each class has a Gaussian distribution of roughly the same variance centered at different means.



(a) Data set for training.

(b) Data set for testing.

Figure 1: Scatterplots of input data.

2 Back-propagation Implementation

The back-propagation algorithm was implemented in C++, to make use of containers, bounds checking, and some minimal OOP structure. The clarity of code was prioritized over speed, given the relatively small training data set and network size, and the warnings of the instructor.

Neurons consist of their incoming weights, an internal activation, an output, and a partial derivative and weight momentums for the back-propagation. They are grouped into Layers, which contain some number of Neurons and pointers to previous and next layers. The structure of the network is thus implicit by the linking of Layers; between layers, all Neurons are fully-connected. Initial weights are random and distributed uniformly between -1 and 1.

Bias neurons are handled separately during computation, and are never represented with a structure. The bias weight for each Neuron is stored and computed separately from the inter-Neuron weights.

During back-propagation, data modification is ordered to avoid disturbing weights before they are used to calculate partial derivatives. This gives a consistent update step for each presentation, though it likely does not make a noticable impact on the convergence if improperly implemented.

The stopping criterion is based on percentage change in training accuracy. If the error does not make a relative .00001 improvement in 100 epochs, the algorithm is considered converged. In comparison with a fixed run of 10000 epochs, this results in very small (less than 0.1%) changes to final accuracy. Typically, because an optional momentum is implemented in the weight changes, the algorithm will not get stuck for long at a given plateau.

3 Network Organizations

3.1 Single Hidden Layer

Four different networks were tested with a single hidden layer; one with 1 hidden node, one with 2, one with 10, and one with 50. All tests were done with an η rate of 0.01 and a momentum α of 0.1. Training presentations were shuffled in each epoch. Generalization plots with training and testing accuracy are given in Figure 2. Converged decision boundaries are shown in Figure 3.

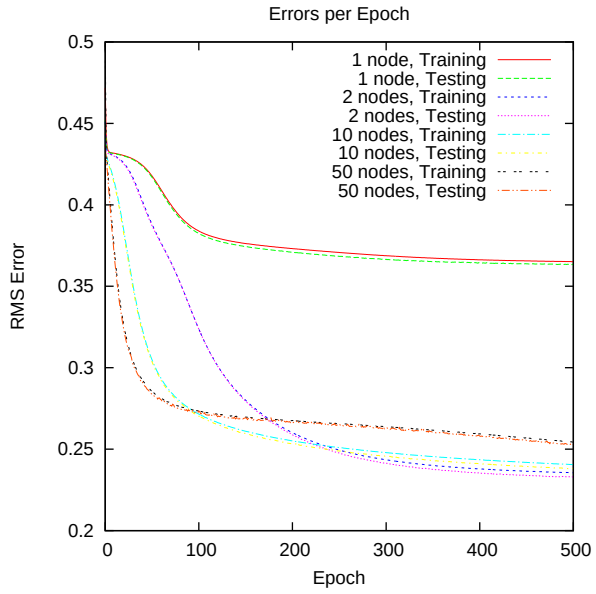


Figure 2: Generalization plots, one hidden layer.

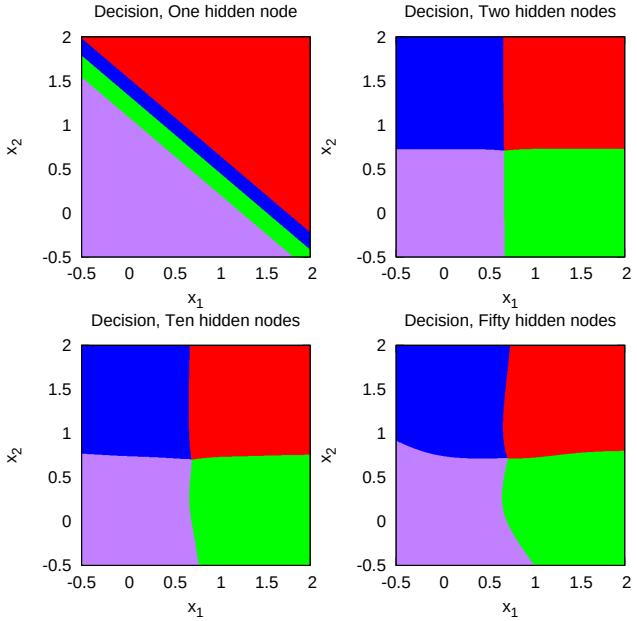


Figure 3: Decision boundaries, one hidden layer.

The data quite obviously needs two hidden nodes for a good representation; this is fitting, as the classes are separated by a set of two planes in two dimensions. Using only one hidden node reveals an essentially one-dimensional decision being made, along a metric of something like $x + y$.

Additional hidden nodes past two do not result in a better decision boundary. In fact, they simply make the algorithm take longer to converge, and result in overfitting the training data. In comparison, the network with two hidden nodes gives a remarkably good boundary with low error rather quickly. This indicates that choosing a well-reasoned network to represent the data is important for the performance and accuracy of the system.

3.2 Two Hidden Layers

Similar tests were run using networks with two hidden layers. Networks were made with configurations of $1 \rightarrow 2$, $2 \rightarrow 1$, $2 \rightarrow 2$, and $10 \rightarrow 50$ hidden nodes. The $O(n^2)$ scaling of the algorithm with the size of fully-connected layers becomes apparent; the 500 hidden weights took a sizable time to compute even on a modern system. Generalization plots with training and testing accuracy are given in Figure 4. Converged decision boundaries are shown in Figure 5.

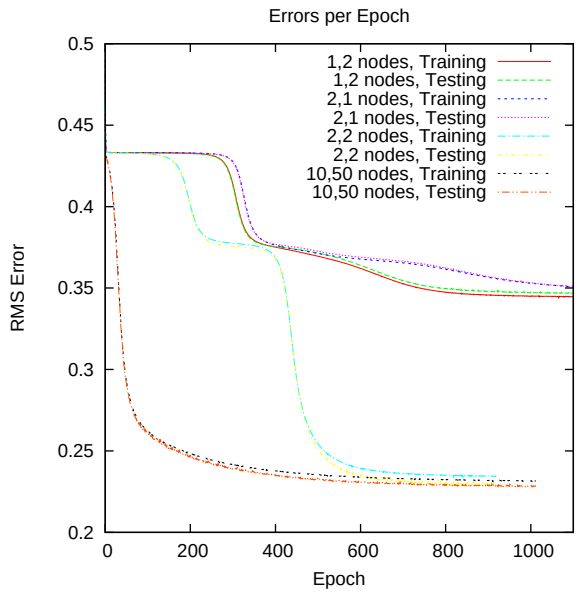


Figure 4: Generalization plots, two hidden layers.

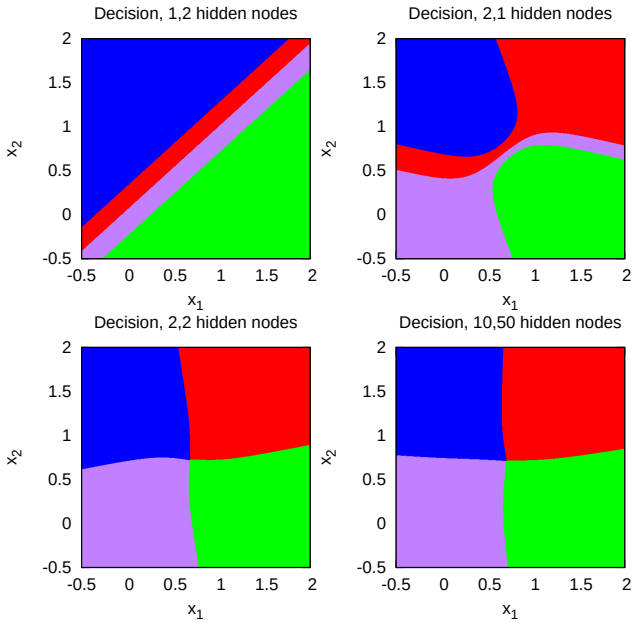


Figure 5: Decision boundaries, two hidden layers.

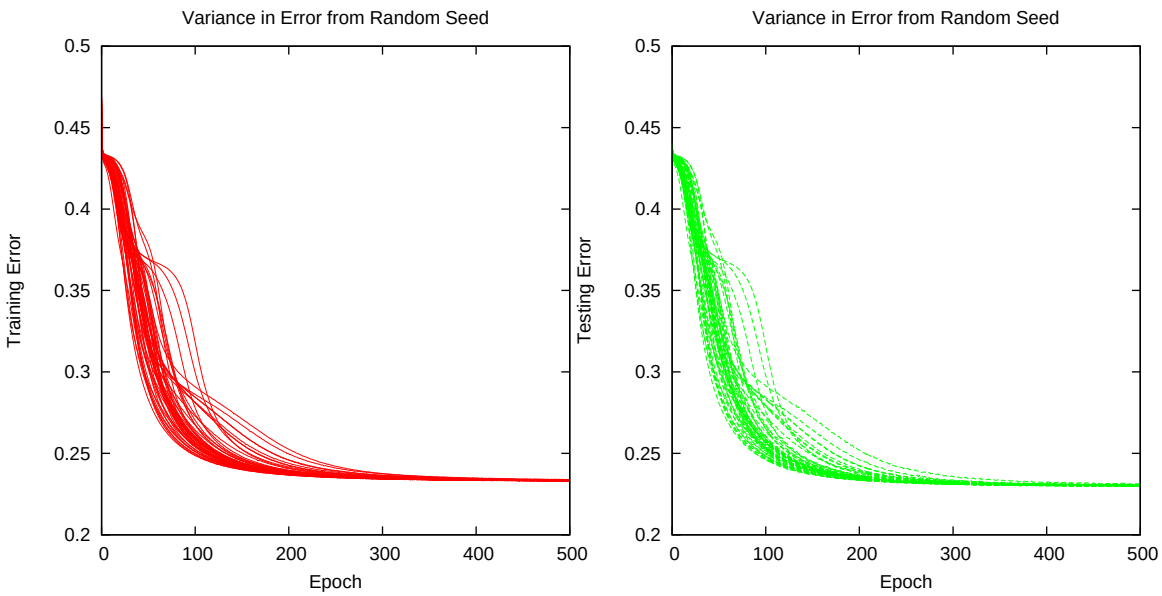
The addition of a second hidden layer makes the algorithm converge much more slowly. As seen previously, extra nodes beyond 2 per layer do not significantly help the accuracy of the algorithm. They do converge more quickly here; I suspect this is because the random initial weights happen to give - by chance - neurons already weighed near the local minima of their error surfaces.

The effect of a single-neuron layer is still apparent; in both such cases, the decision boundaries are very poor. The network with the single neuron in the first hidden layer gives the linear boundary seen in the trivial one-hidden-neuron network architecture. The network with the single neuron in the second hidden layer is more interesting. There is obviously some potential here for an accurate classifier, though it is quite roundabout to arrive there. Often, depending on the initial weight selection and presentation order, the algorithm simply does not converge.

I believe there exists a neural network, with a single-neuron hidden layer, that could classify this problem accurately. The single neuron would essentially encode its output as 4 different levels, triggering 4 differently-positioned nonlinearities in the next layer. Such a configuration would not likely be learned by this algorithm without significant refinement.

4 Effect of Random Seed

A series of 50 tests were run with a η rate of 0.02, a momentum α of 0.1, shuffled training data, and a single hidden layer of two nodes. The random seed (changing the starting weights and presentation order) was altered in each run, and the results plotted in Figure 6. The results are given separately for the training and testing errors for legibility. Each run, individually, shows a testing error that closely tracks the corresponding training error.



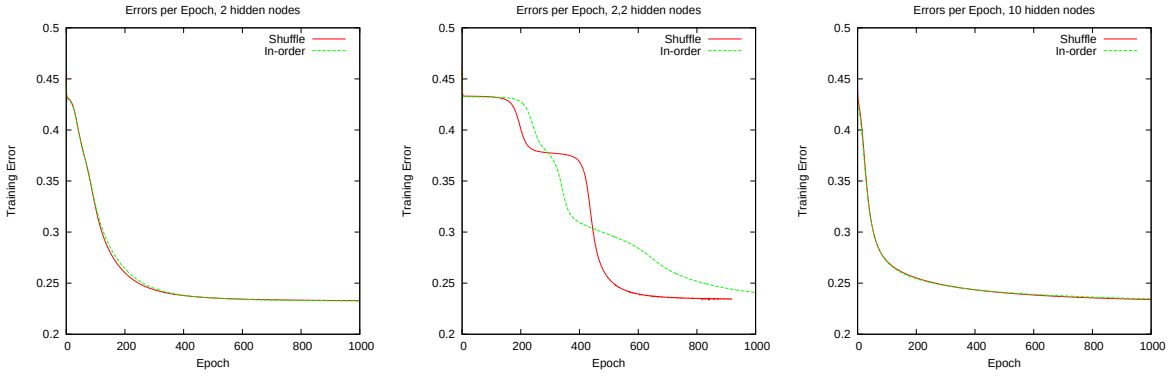
(a) Random variance in training error. **(b)** Random variance in testing error.

Figure 6: Effect of random seed on convergence.

As evident in the figure, the runs all converged to approximately the same final training and testing errors. There was wide variance in the amount of epochs needed to converge, however.

5 Effect of In-Order Presentation

A series of tests was run, similar to the previous section, to examine the effect of in-order versus randomized presentation of the training data. The training data, as given, is sorted by class; it was expected that this would negatively impact the convergence of the classifier. Presenting a large number of samples with the same desired output could strongly bias the output neurons rather than altering the inter-neuron weights. The results are given in Figure 7; multiple hidden layers are hurt more than a single hidden layer.



(a) 2 hidden nodes. **(b)** 2 and 2 hidden nodes. **(c)** 10 hidden nodes.

Figure 7: Effect of presentation order on convergence.

6 Data Summary

Table 1 gives a summary of the accuracy found in each test perfomed in this report. The results are sorted by order of increasing testing error.

Hidden Nodes	η	α	Shuffle	Epochs to Converge	Training Error	Testing Error
10	0.01	0.1	1	2680	0.230694	0.227683
50	0.01	0.1	1	1769	0.231006	0.227885
10-50	0.01	0.1	1	1012	0.231549	0.228182
2-2	0.01	0.1	1	919	0.234512	0.230004
2	0.01	0.1	1	1341	0.232634	0.230127
10	0.01	0.1	0	6785	0.228967	0.230466
2	0.01	0.1	0	1726	0.232212	0.232435
50	0.01	0.1	0	4101	0.227133	0.236424
2-2	0.01	0.1	0	4490	0.230817	0.241623
10-50	0.01	0.1	0	2936	0.225640	0.245263
2-1	0.1	0.99	1	3075	0.375658	0.300000
2-1	0.1	0.7	1	9999	0.323044	0.325329
2-1	0.05	0.7	1	9999	0.323633	0.326766
5-5-1	0.05	0.9	1	1219	0.327996	0.329801
2-1	0.05	0.1	1	2466	0.331859	0.333430
3-2-1	0.1	0.05	1	1109	0.330266	0.334716
2-1	0.1	0.1	1	1136	0.332679	0.334769
3-2-1	0.1	0.5	1	855	0.329852	0.335307
3-2-1	0.1	0.1	1	1109	0.329851	0.335379
2-1	0.05	0.75	1	935	0.333613	0.336810
5-5-1	0.1	0.9	1	651	0.332026	0.337919
3-2-1	0.05	0.9	1	602	0.335595	0.338369
3-2-1	0.5	0.5	1	356	0.336213	0.338544
3-2-1	0.5	0.05	1	434	0.333445	0.340999
1-4	0.05	0.1	1	639	0.346209	0.341047
1-2	0.01	0.1	1	1207	0.344247	0.346777
2-1	0.01	0.1	1	1730	0.347933	0.347521
1-10	0.05	0.1	1	471	0.345609	0.347714
1-2	0.05	0.1	1	328	0.345156	0.349598
1-2	0.1	0.1	1	328	0.346278	0.352342
1	0.01	0.1	1	1482	0.363023	0.361472
1	0.01	0.1	0	2264	0.362122	0.374055

Table 1: Summary of experiments performed.

7 Conclusion

The back-propagation learning algorithm is very good at fitting a neural network to a set of training data. It generalizes well and, with the right architecture, converges quickly and reliably. However, deciding on this "right architecture" can be challenging. Some knowledge of the data is required to organize the network. The most effective type of network will depend on the problem and data being analyzed. Though a non-optimal network will still converge on the training data and generalize well, it may slightly overfit, take a long time to converge (or get "stuck"), or simply fail to converge if it is a very poor match.

Code is attached. MPEG animations of decision boundaries across epochs are available.

```cpp
//backprop.cpp
//Implements back-propagation algorithm. 2 inputs, 4 outputs, variable hidden layers/nodes.
//Bryan Topp <betopp@cs.unm.edu>

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <vector>
#include <assert.h>
#include <string>
#include <sstream>
#include <algorithm> //shuffle

#define MAXEPOCHS 10000
#define DUMPFRAMEDIM 512
#define DUMPFRAMEDIM_s "512"

double eta;
int num_hidden_layers;
double momentum_scale;

double rand01()
{
    return (double)rand() / (double)RAND_MAX;
}

double sigmoid(double input)
{
    return 1.0 / (1.0 + exp(-input));
}
double sigmoidprime(double input)
{
    return sigmoid(input) * (1.0 - sigmoid(input));
}

//Turns a class number into a set of desired neural outputs.
const double *desired_output_for_class(int c)
{
    static const double c1_d[] = {1.0, 0.0, 0.0, 0.0};
    static const double c2_d[] = {0.0, 1.0, 0.0, 0.0};
    static const double c3_d[] = {0.0, 0.0, 1.0, 0.0};
    static const double c4_d[] = {0.0, 0.0, 0.0, 1.0};
    assert(c >= 1);
    assert(c <= 4);

    if(c == 1)
        return c1_d;
    if(c == 2)
        return c2_d;
    if(c == 3)
        return c3_d;
    if(c == 4)
        return c4_d;
}

class Neuron
{
public:
    std::vector<double> IncomingWeights;
    std::vector<double> IncomingWeightsMomentum;
    double BiasWeight;
    double v;
    double y; //the output of this neuron (forward-prop)
    double partial_deriv; //the effect on the error for exciting this neuron (back-prop)

    Neuron(unsigned int numweights);
};

class Layer
{
public:
    std::vector<Neuron> Neurons;
    Layer *PreviousLayer;
    Layer *NextLayer;

    Layer(unsigned int numneurons, unsigned int weights_per_neuron);
};


Neuron::Neuron(unsigned int numweights)
{
    for(unsigned int w = 0; w < numweights; w++)
    {
        IncomingWeights.push_back(rand01() - rand01());
        IncomingWeightsMomentum.push_back(0);
    }

    BiasWeight = 0.0;
}
```

```
Layer::Layer(unsigned int numneurons, unsigned int weights_per_neuron)
{
    for(unsigned int n = 0; n < numneurons; n++)
    {
        Neurons.push_back(Neuron(weights_per_neuron));
    }
}

class InputPoint
{
public:
    double x[2];
    int desired;

};

std::vector<InputPoint> TrainingData;
std::vector<InputPoint> TestingData;

void LoadData(const char *filename, std::vector<InputPoint> &points)
{
    FILE *inf = fopen(filename, "r");
    assert(inf);

    while(1)
    {
        int desired, index;
        double x1, x2;

        if(fscanf(inf, "%d %d %lf %lf", &desired, &index, &x1, &x2) != 4)
            break;

        InputPoint newpoint;

        newpoint.x[0] = x1;
        newpoint.x[1] = x2;
        newpoint.desired = desired;

        assert(newpoint.desired >= 1);
        assert(newpoint.desired <= 4);

        points.push_back(newpoint);
    }

    printf("Read %lu points from %s\n", points.size(), filename);

    fclose(inf);
}

//Feeds input layer and propagates through all subsequent layers.
void ForwardPropagate(Layer &InputLayer, float x1, float x2)
{
    //Set up the input nodes with the correct data.
    InputLayer.Neurons[0].y = InputLayer.Neurons[0].v = x1;
    InputLayer.Neurons[1].y = InputLayer.Neurons[1].v = x2;

    //Forward propagate through all layers
    Layer *TargetLayer = InputLayer.NextLayer;
    while(TargetLayer != NULL)
    {
        //Consider each neuron in the receiving layer
        for(unsigned int target = 0; target < TargetLayer->Neurons.size(); target++)
        {
            Neuron *t = &(TargetLayer->Neurons[target]);

            //Initialize activation with bias
            t->v = t->BiasWeight;

            //Accumulate incoming weights
            assert(t->IncomingWeights.size() == TargetLayer->PreviousLayer->Neurons.size());
            for(unsigned int exciter = 0; exciter < t->IncomingWeights.size(); exciter++)
            {
                t->v += t->IncomingWeights[exciter] * TargetLayer->PreviousLayer->Neurons[exciter].y;
            }

            //Calculate output
            t->y = sigmoid(t->v);
        }

        if(TargetLayer->NextLayer != NULL)
            assert(TargetLayer->NextLayer->PreviousLayer == TargetLayer);

        TargetLayer = TargetLayer->NextLayer;

    }

}
```

```
int main(int argc, const char **argv)
{
    if(argc < 8)
    {
        printf("usage: %s <eta> <momentum> <# hidden layers> <dump yuv? 0/1> <random seed> <shuffle training 0/1> <hidden node counts...>\n", argv[0]);
        exit(-1);
    }

    eta = atof(argv[1]);
    momentum_scale = atof(argv[2]);
    num_hidden_layers = atoi(argv[3]);

    int should_dump = atoi(argv[4]);
    int rseed = atoi(argv[5]);
    int shuffle_training = atoi(argv[6]);

    printf("Using learning rate (eta) of %lf.\n", eta);
    printf("Using momentum %lf.\n", momentum_scale);
    printf("Using %d hidden layers.\n", num_hidden_layers);

    FILE *dumpfile = NULL;
    if(should_dump)
    {
        printf("Dumping YUV4MPEG grid animation.\n");
        dumpfile = fopen("boundary.yuv4mpeg", "wb");
        assert(dumpfile);

        fprintf(dumpfile, "YUV4MPEG2 W"DUMPFRAMEDIM_s" H"DUMPFRAMEDIM_s" F30 Ip A1:1 C444\n");
    }
    else
    {
        printf("Not dumping grid animation data.\n");
    }


    printf("Using random seed %d.\n", rseed);
    srand(rseed);

    if(shuffle_training)
        printf("Shuffling training data.\n");
    else
        printf("Presenting training in-order.\n");


    printf("Hidden layer node configuration:\n");
    std::vector<int> hidden_layer_node_counts;
    for(int a = 7; a < argc; a++)
    {
        printf("\t%d\n", atoi(argv[a]));
        hidden_layer_node_counts.push_back(atoi(argv[a]));
    }
    assert(hidden_layer_node_counts.size() == num_hidden_layers);


    //Open file to dump generalization plots.
    std::ostringstream filename;
    filename << "gen/gen_eta" << eta << "_mom" << momentum_scale << "_rseed" << rseed << "_shuf" << shuffle_training << "_";
    for(unsigned int a = 0; a < hidden_layer_node_counts.size(); a++)
    {
        if(a != 0)
            filename << "-";

        filename  << hidden_layer_node_counts[a];
    }

    printf("Printing generalization plot to %s\n", filename.str().c_str());
    FILE *genf = fopen(filename.str().c_str(), "w");
    assert(genf);

    LoadData("TrainingData.txt", TrainingData);
    LoadData("TestingData.txt", TestingData);

    assert(num_hidden_layers > 0);
    assert(eta > 0.0);
    assert(momentum_scale >= 0.0);

    Layer InputLayer(2, 0);
    InputLayer.PreviousLayer = NULL;

    //Build hidden layers
    std::vector<Layer> HiddenLayers;
    for(unsigned int hl = 0; hl < num_hidden_layers; hl++)
    {
        if(hl == 0) //first hidden layer has 2 inputs per node, from the input layer
        {
            HiddenLayers.push_back(Layer(hidden_layer_node_counts[hl], 2));
            HiddenLayers[hl].PreviousLayer = &InputLayer;
            InputLayer.NextLayer = &(HiddenLayers[hl]);
        }
```

```
        else
        {
            HiddenLayers.push_back(Layer(hidden_layer_node_counts[hl], hidden_layer_node_counts[hl-1]));
            HiddenLayers[hl].PreviousLayer = &(HiddenLayers[hl-1]);
            HiddenLayers[hl-1].NextLayer = &(HiddenLayers[hl]);
        }

    }

    //Make pointers - AFTER the dynamic vector has resized itself as necessary.
    for(unsigned int hl = 0; hl < num_hidden_layers; hl++)
    {
        if(hl == 0)
        {
            HiddenLayers[hl].PreviousLayer = &InputLayer;
            InputLayer.NextLayer = &(HiddenLayers[hl]);
        }
        else
        {
            HiddenLayers[hl].PreviousLayer = &(HiddenLayers[hl-1]);
            HiddenLayers[hl-1].NextLayer = &(HiddenLayers[hl]);
        }

    }

    Layer OutputLayer(4, hidden_layer_node_counts[hidden_layer_node_counts.size()-1]);
    OutputLayer.PreviousLayer = &(HiddenLayers[HiddenLayers.size()-1]);
    HiddenLayers[HiddenLayers.size()-1].NextLayer = &OutputLayer;
    OutputLayer.NextLayer = NULL;


    assert(InputLayer.Neurons.size() == 2);
    assert(OutputLayer.Neurons.size() == 4);
    assert(HiddenLayers.size() == num_hidden_layers);
    assert(HiddenLayers[0].Neurons.size() == hidden_layer_node_counts[0]);


    double last_training_error = 1000.0;
    int epochs_since_significant_improvement = 0;


    int epoch;
    for(epoch = 0; epoch < MAXEPOCHS; epoch++)
    {
        if(shuffle_training)
            std::random_shuffle ( TrainingData.begin(), TrainingData.end() );


        double accumulated_squared_error = 0.0;

        //Present each training data point and train the network.
        int presentation;
        for(presentation = 0; presentation < TrainingData.size(); presentation++)
        {
            //Do forward-propagation
            ForwardPropagate(InputLayer, TrainingData[presentation].x[0], TrainingData[presentation].x[1]);

            //Ready desired outputs based on training data class.
            const double *desired_outputs = desired_output_for_class(TrainingData[presentation].desired);

            //Calculate errors
            double output_errors[4];
            for(unsigned int o = 0; o < 4; o++)
            {
                output_errors[o] = desired_outputs[o] - OutputLayer.Neurons[o].y;
                accumulated_squared_error += (output_errors[o] * output_errors[o]) / 4.0;
            }

            //Calculate partial derivatives for output, update weights
            for(unsigned int o = 0; o < 4; o++)
            {
                OutputLayer.Neurons[o].partial_deriv = output_errors[o] * sigmoidprime(OutputLayer.Neurons[o].v);
            }

            //Backward propagate
            Layer *TargetLayer = &OutputLayer;
            //For each layer, we'll adjust weights and find derivatives for the previous layer
            while(TargetLayer->PreviousLayer != NULL)
            {
                //Zero all the partial derivatives on the previous layer
                for(unsigned int exciter = 0; exciter < TargetLayer->PreviousLayer->Neurons.size(); exciter++)
                {
                    TargetLayer->PreviousLayer->Neurons[exciter].partial_deriv = 0.0;
                }

                //Loop through all neurons in the target layer
                //For each one, loop through all its weights (i.e., all the previous layer neurons).
                //Adjust weights based on excitation and current partial.
                //Accumulate new partials for the "previous" layer.
```

```
            for(unsigned int target = 0; target < TargetLayer->Neurons.size(); target++)
            {
                Neuron *t = &(TargetLayer->Neurons[target]);

                assert(t->IncomingWeights.size() == TargetLayer->PreviousLayer->Neurons.size());
                for(unsigned int exciter = 0; exciter < t->IncomingWeights.size(); exciter++)
                {
                    Neuron *e = &(TargetLayer->PreviousLayer->Neurons[exciter]);

                    //calculate this target's contribution to the exciter's partial
                    e->partial_deriv += sigmoidprime(e->v) * t->IncomingWeights[exciter] * t->partial_deriv;

                    //adjust the weight based on how strongly it was active, and which way it needs to go
                    t->IncomingWeightsMomentum[exciter] *= momentum_scale;
                    t->IncomingWeightsMomentum[exciter] += e->y * t->partial_deriv * eta;
                    t->IncomingWeights[exciter] += t->IncomingWeightsMomentum[exciter];


                }

                //Do the bias weight for this target as well.
                t->BiasWeight += t->partial_deriv * eta;
            }


            TargetLayer = TargetLayer->PreviousLayer;
        }

    }

    //Reset error measure
    double training_mse = sqrt(accumulated_squared_error / (double)presentation);
    accumulated_squared_error = 0;

    //Testing
    for(presentation = 0; presentation < TestingData.size(); presentation++)
    {
        //Forward propagation on test data
        ForwardPropagate(InputLayer, TestingData[presentation].x[0], TestingData[presentation].x[1]);

        //Ready desired outputs based on training data class.
        const double *desired_outputs = desired_output_for_class(TestingData[presentation].desired);

        //Calculate errors
        double output_errors[4];
        for(unsigned int o = 0; o < 4; o++)
        {
            output_errors[o] = desired_outputs[o] - OutputLayer.Neurons[o].y;
            accumulated_squared_error += (output_errors[o] * output_errors[o]) / 4.0;
        }
    }
    double testing_mse = sqrt(accumulated_squared_error / (double)presentation);

    printf("Epoch %d trained, found %lf rms training, %lf rms testing.\n", epoch, training_mse, testing_mse);
    fprintf(genf, "%d %lf %lf\n", epoch, training_mse, testing_mse);

    //Dump grid if requested
    if(should_dump)
    {
        fprintf(dumpfile, "FRAME\n");

        unsigned char y[DUMPFRAMEDIM][DUMPFRAMEDIM];
        char cb[DUMPFRAMEDIM][DUMPFRAMEDIM];
        char cr[DUMPFRAMEDIM][DUMPFRAMEDIM];

        for(int yp = 0; yp < DUMPFRAMEDIM; yp++)
        {
            for(int xp = 0; xp < DUMPFRAMEDIM; xp++)
            {
                double gridx = (((double)xp / (double)DUMPFRAMEDIM) * 2.5) - 0.5;
                double gridy = (((double)yp / (double)DUMPFRAMEDIM) * 2.5) - 0.5;
                ForwardPropagate(InputLayer, gridx, gridy);

                unsigned char ly = 200;
                char lcb;
                char lcr;

                if(OutputLayer.Neurons[0].y >= OutputLayer.Neurons[1].y &&
                    OutputLayer.Neurons[0].y >= OutputLayer.Neurons[2].y &&
                    OutputLayer.Neurons[0].y >= OutputLayer.Neurons[3].y)
                {
                    lcb = 255;
                    lcr = 255;
                }
                else if(OutputLayer.Neurons[1].y >= OutputLayer.Neurons[0].y &&
                    OutputLayer.Neurons[1].y >= OutputLayer.Neurons[2].y &&
                    OutputLayer.Neurons[1].y >= OutputLayer.Neurons[3].y)
                {
                    lcb = 0;
```

```
                        lcr = 255;
                    }
                    else if(OutputLayer.Neurons[2].y >= OutputLayer.Neurons[1].y &&
                        OutputLayer.Neurons[2].y >= OutputLayer.Neurons[0].y &&
                        OutputLayer.Neurons[2].y >= OutputLayer.Neurons[3].y)
                    {
                        lcb = 255;
                        lcr = 0;
                    }
                    else
                    {
                        lcb = 0;
                        lcr = 0;
                    }

                    y[yp][xp] = ly;
                    cb[yp][xp] = lcb;
                    cr[yp][xp] = lcr;
                }
            }

            fwrite(y, 1, DUMPFRAMEDIM*DUMPFRAMEDIM, dumpfile);
            fwrite(cb, 1, DUMPFRAMEDIM*DUMPFRAMEDIM, dumpfile);
            fwrite(cr, 1, DUMPFRAMEDIM*DUMPFRAMEDIM, dumpfile);
        }

        //Abort if we haven't moved significantly
        if(training_mse < last_training_error * 0.9999)
        {
            epochs_since_significant_improvement = 0;
            last_training_error = training_mse;
        }
        else
            epochs_since_significant_improvement++;

        if(epochs_since_significant_improvement > 100)
            break;
    }
    fclose(genf);
    if(dumpfile)
        fclose(dumpfile);

    //print a grid of the final decision boundary
    filename.seekp(0);
    filename << "bound/bound_eta"<< eta << "_mom"<< momentum_scale << "_rseed"<< rseed << "_shuf"<< shuffle_training <<"_";
    for(unsigned int a = 0; a < hidden_layer_node_counts.size(); a++)
    {
        if(a != 0)
            filename << "-";
        filename  << hidden_layer_node_counts[a];
    }

    printf("Printing final decision plot to %s\n", filename.str().c_str());
    FILE *boundf = fopen(filename.str().c_str(), "w");
    assert(boundf);
    for(int yp = 0; yp < DUMPFRAMEDIM; yp++)
    {
        for(int xp = 0; xp < DUMPFRAMEDIM; xp++)
        {
            double gridx = (((double)xp / (double)DUMPFRAMEDIM) * 2.5) - 0.5;
            double gridy = (((double)yp / (double)DUMPFRAMEDIM) * 2.5) - 0.5;
            ForwardPropagate(InputLayer, gridx, gridy);

            if(OutputLayer.Neurons[0].y >= OutputLayer.Neurons[1].y &&
                OutputLayer.Neurons[0].y >= OutputLayer.Neurons[2].y &&
                OutputLayer.Neurons[0].y >= OutputLayer.Neurons[3].y)
            {
                fprintf(boundf, "%lf %lf %d\n", gridx, gridy, 1);
            }
            else if(OutputLayer.Neurons[1].y >= OutputLayer.Neurons[0].y &&
                OutputLayer.Neurons[1].y >= OutputLayer.Neurons[2].y &&
                OutputLayer.Neurons[1].y >= OutputLayer.Neurons[3].y)
            {
                fprintf(boundf, "%lf %lf %d\n", gridx, gridy, 2);
            }
            else if(OutputLayer.Neurons[2].y >= OutputLayer.Neurons[1].y &&
                OutputLayer.Neurons[2].y >= OutputLayer.Neurons[0].y &&
                OutputLayer.Neurons[2].y >= OutputLayer.Neurons[3].y)
            {
                fprintf(boundf, "%lf %lf %d\n", gridx, gridy, 3);
            }
            else
            {
                fprintf(boundf, "%lf %lf %d\n", gridx, gridy, 4);
            }
        }
    }
    fclose(boundf);
}
```