CSCI964 Computational Intelligence: Lab #1

Mei Wangzhihui 2019124044

Task 1

1) Single-layer Neural Network is an Artificial Neural Network (ANN) with an input layer and a output layer.

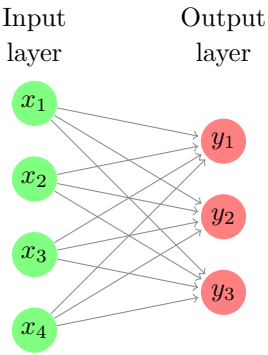

Figure 1: Single-layer Neural Network

2) Multi-layer Neural Network contains more than one layer of artificial neurons with several hidden layer.

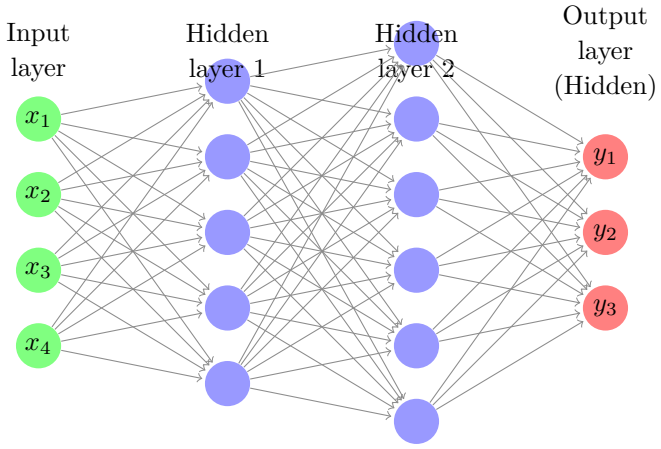

Figure 2: Multi-layer Neural Network

3) Shallow Neural Network contains less than 2 hidden layers. It fit functions with a lot parameters.

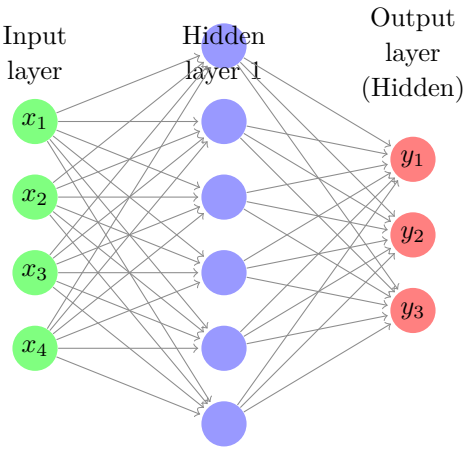

Figure 3: Shallow Neural Network

4) Deep Neural Network contains more than one hidden layers. It can fit functions better with less parameters than a shallow network.

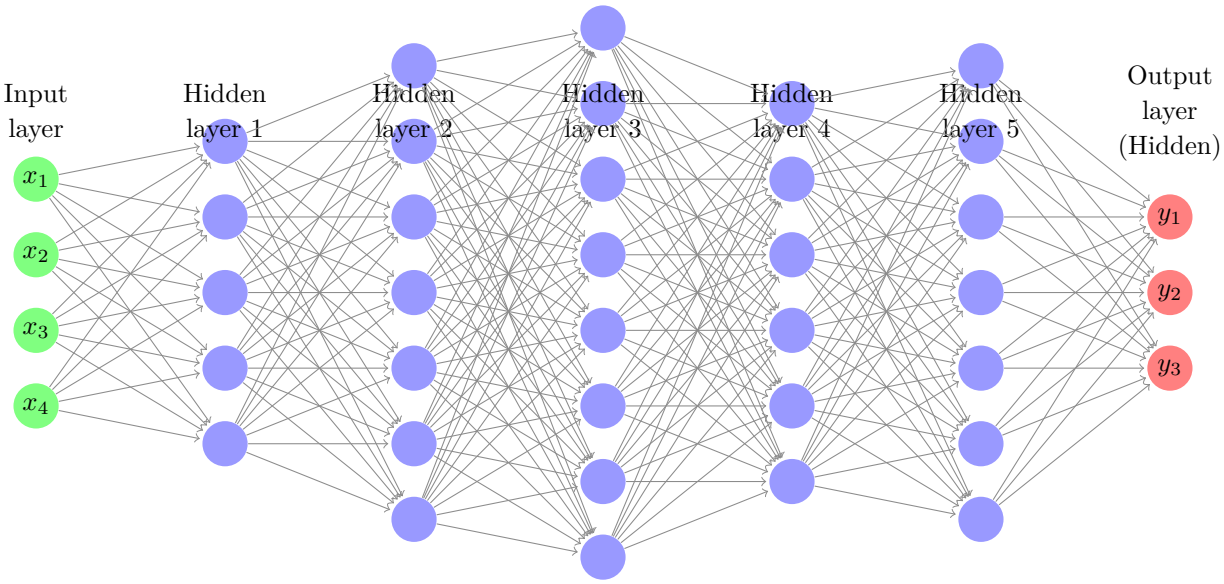

Figure 4: Deep Neural Network

Task 2

The implementation of matrix operation:

```
//matrix.h
class Matrix {
public:
   // constructors
   Matrix();
   Matrix(int m, int n, std::vector<std::vector<double>> matrix);
   Matrix(int m, int n); // zero matrix
   Matrix(std::vector<std::vector<double>> matrix); // zero matrix
   // applies a function over the entire matrix
   template<typename F> void apply(F f) {
      for (int i = 0; i < m; ++i) {
         for (int j = 0; j < n; ++j) {
            matrix[i][j] = f(matrix[i][j]);
         }
      }
   }

   // matrix operations
   Matrix operator+(const Matrix& other) const; // add
   Matrix operator-(const Matrix& other) const; // subtract
   Matrix operator*(const Matrix& other) const; // matrix multiplication

   Matrix unitMultiply(const Matrix& other) const;
   Matrix transpose();

   // stream operator
   friend std::ostream& operator<<(std::ostream& out, const Matrix& matrix);

   // randomizations
   void initNormal();
   void init(int m, int n, std::vector<std::vector<double>> &matrix);
   // misc operations
   double sum();
   void normalize();

   // accessors
   int getRows();
   int getCols();
   std::vector<std::vector<double>> getVector();

private:
   int m, n;
   std::vector<std::vector<double>> matrix;
};
```

Problem 2 continued on next page...

The neuralnet operator contains input layer, hidden layer and output layer.

```
//neuralnet.h
class NeuralNet {
public:
  // constructor
  NeuralNet(int input_size, int output_size, std::vector<int> hidden_sizes, Matrix inputs, Matrix
      outputs);
  NeuralNet(int input_size, int output_size, std::vector<int> hidden_sizes, std::vector<Matrix>
      &weights, std::vector<Matrix> &biases, Matrix inputs);
  // activation functions
  static double sigmoid(double n, bool deriv = false);
  static double relu(double n, bool deriv = false);
  static double htan(double n, bool deriv = false);

  template<typename F> static double activate(F f, double n, bool deriv = false) {
    return f(n, deriv);
  }

  template<typename F> static std::vector<double> activate(F f, std::vector<double> v, bool deriv
      = false) {
    std::vector<double> result = v;
    for (auto &n: result) {
      if (deriv) {
        n = f(n, true);
      }
      else {
        n = f(n, false);
      }
    }

    return result;
  }

  template<typename F> static Matrix activate(F f, Matrix m, bool deriv = false) {
    std::vector<std::vector<double>> vector = m.getVector();
    for (auto &v: vector) {
      if (deriv) {
        v = activate(f, v, true);
      }
      else {
        v = activate(f, v);
      }
    }

    Matrix result {m.getRows(), m.getCols(), vector};

    return result;
  }

  // loss function (average sum of squares)
  double loss();

  // propogation
```

```
    Matrix feedForward(Matrix input);
    void backProp(int batch_size = 0);
    void setInitialWeight(std::vector<Matrix> &weights){

}
    // ith weight matrix accessor
    Matrix getWeights(int i);

    // using the model
    void train(int epochs);
    Matrix predict(Matrix input);
    // stream operator
    friend std::ostream &operator<<(std::ostream &out, const NeuralNet &nn);

private:
    const int input_size;
    const int output_size;
    const std::vector<int> hidden_sizes;

    std::vector<Matrix> intermediates;

    Matrix inputs;
    std::vector<Matrix> weights;
    std::vector<Matrix> biases;
    Matrix outputs;
};
```

Problem 2 continued on next page...

Initialize the neuralnetwork.

```
\\neuralnet.cc
...
NeuralNet::NeuralNet(int input_size, int output_size, std::vector<int> hidden_sizes,
    std::vector<Matrix> &weights, std::vector<Matrix> &biases, Matrix inputs) :
    input_size{input_size}, output_size{output_size}, hidden_sizes{hidden_sizes}, inputs{inputs}
{
   int totalsize = weights.size();
   int m, n, bm, bn;
   for (unsigned int i = 0; i < totalsize; i++)
   {
      m = weights[i].getRows();
      n = weights[i].getCols();
      bm = biases[i].getRows();
      bn = biases[i].getCols();
      Matrix weight_layer{m, n, weights[i].getVector()}; //initialize the weight matrixes
      Matrix bias{bm, bn, biases[i].getVector()}; //initialize the bias vectors
      this->weights.push_back(weight_layer);
      this->biases.push_back(bias);
      intermediates.push_back(Matrix{0, 0}); //initialize the layers
   }
}


Matrix NeuralNet::feedForward(Matrix input)
{
  Matrix result = input;
  for (unsigned int i = 0; i < weights.size(); ++i)
  {
    result = result * weights[i] + biases[i];
    result = activate(sigmoid, result);
    intermediates[i] = result;
  }

  return result;
}
...
\\main.cc
Matrix input{1, 2, {{1, 2}}};
Matrix WeightIH1{2, 2, {{3, 2}, {1, 4}}};
Matrix WeightIH2{2, 2, {{3, 5}, {2, 1}}};
Matrix bias1{1, 2, {{1, 1}}};
Matrix bias2{1, 2, {{1, 1}}};
Matrix inputs{1, 2, {{1, 2}}};
vector<Matrix> weights{WeightIH1, WeightIH2};
vector<Matrix> biases{bias1, bias2};
vector<int> hidden{2, 2};
NeuralNet nn{2, 2, hidden, weights, biases, inputs};
nn.feedForward(input); \\ The forwardpropagate phase.
cout << nn;
```

The results:

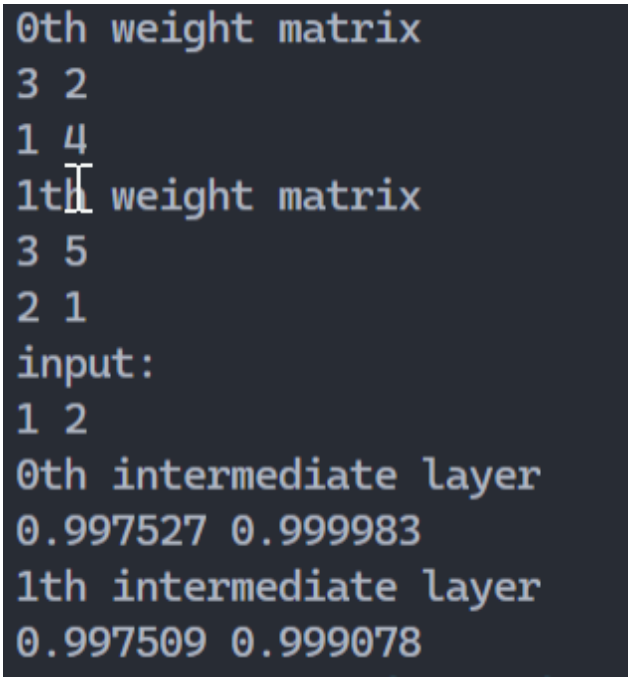

Figure 5: The results