Final project report

Lingfeng Zhou

Reema Dutta

Overview

An artificial neural network is built from scratch. This ANN can be built with arbitrary neurons and layers, with four kinds of activation functions, which are sigmoid, tanh, Relu and Softmax. MNIST dataset of handwritten digits is adopted to demonstrate the feasibleness of the ANN.

Procedures

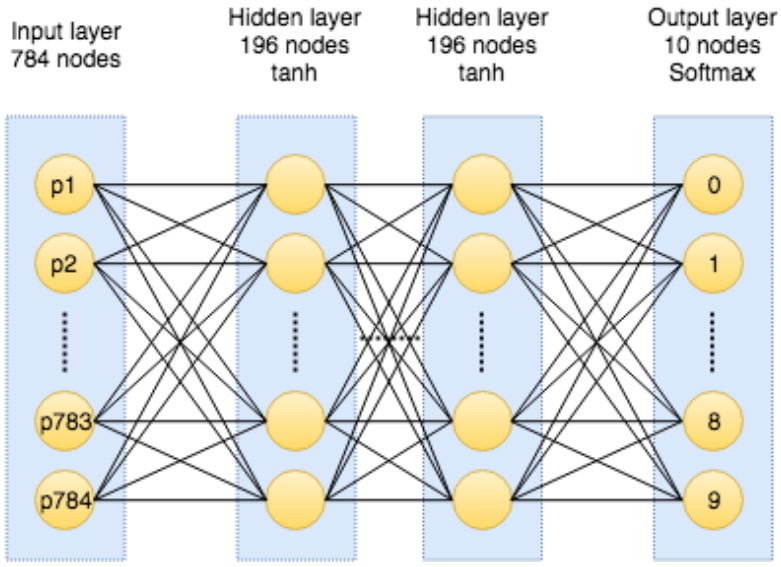
Input data

The input data to our neural network are numbers which converted in pixels. the perceptron will guess from the data set which number the pixels represent. The inputs are stored in a simple array. The output is also stored as an array of possibilities of 10 possible digits.

The full data set contains two csv files . We have a training set called train.csv and a testing set called validate.csv, where train.csv contains 40,000 pictures and validate.csv contains 2,000 pictures. When training the network we will use the corresponding numeric label to “teach” it. When testing, it will be used as an “exam” question for our computer to see whether it can guess correctly.

Neural Network

Firstly let’s begin with our “**Neuron Network**” class which consists of a input-layer, an output-layer and a collection of hidden-layers. The class has been initialised in two ways either as an input layer with inputs such as input nodes, total hidden layer, hidden nodes and output nodes weights and biases Or as hidden or output layer which receives as input the output of the preceding layer and the learning rate.



2 hidden-layer Neural network

We also give some weights which are randomly generated (between -1 and +1). The weights and the biases have been initialised randomly and also depending on which layer (input , hidden and output)they are given as input.

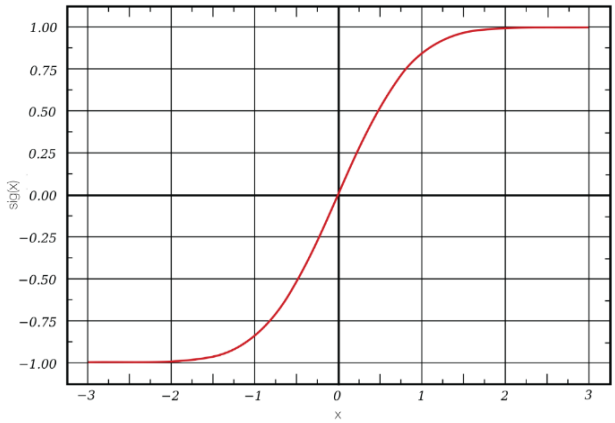
The learning rate has been initialised as 0.001 . it is neither too big so fine tuning would not be difficult nor too small so the process would not take a lot of time.

```
public class NeuralNetwork {

    private Random random = new Random();

    private Layer inputLayer;
    private HiddenLayers hiddenLayers;
    private Layer outputLayer;
    //learningRate
    private double rt=0.001;
    //Activation Function
    //sigmoid
    public static MathFunction actFunc=ActivationFunction.tanh;
    public static MathFunction deFunc = ActivationFunction.dTanh;
    private SimpleMatrix[] weights;
    private SimpleMatrix[] biases;
```

We have different kinds of activation function for hidden-layers such as sigmoid, tanh and relu. Among these we use the tanh. There are two reasons for that choice :

1.Having stronger gradients: since data is centred around 0, the derivatives are higher. To see this, calculate the derivative of the tanh function and notice that its range (output values) is [0,1]. The maxed derivative is great because we can better pass our error through the layers when we use back-propagation.

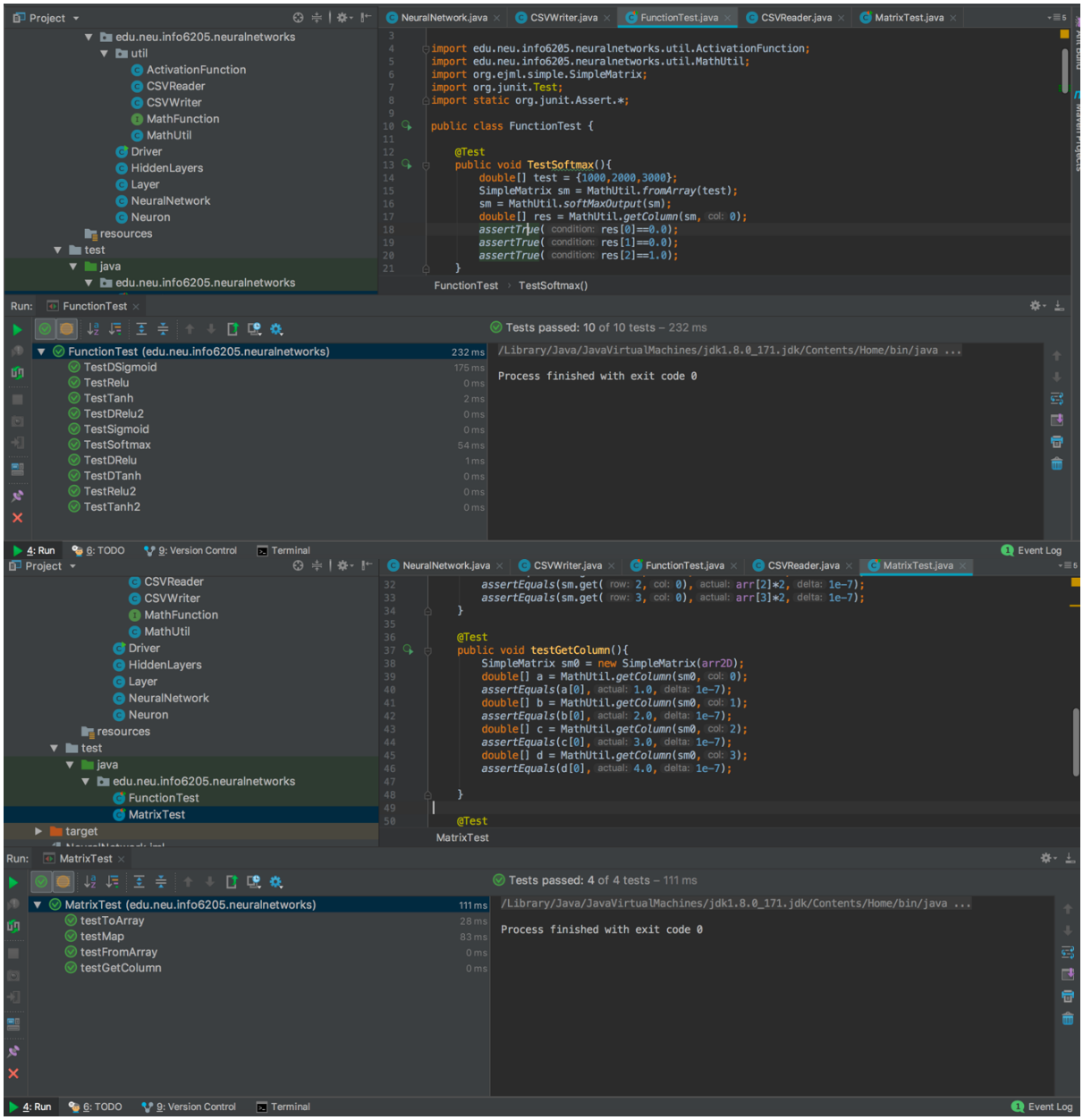
2.Avoiding the values of weights becoming so large as to overflow and result in NaN values with static learning rate.



tanh function

Unit test

A bunch of unit tests are designed to test basic matrix operations and activation function outputs.



Training the network

Our train() method mainly consists of two parts feed forward and back propagation. The feed forward process is used to make predictions according to the weights and bias. While the back propagation process can be used to feedback errors to its inputs. Therefore, the network is fully connected together and it can go back-to-front through the layers, updating the weights and bias of each layer.

```
public void train(double[] inputArray, double[] targetArray) {

    // Feed forward
    SimpleMatrix input = MathUtil.fromArray(inputArray).divide(255);
//      SimpleMatrix input = MathUtil.fromArray(inputArray);
    SimpleMatrix layers[] = new SimpleMatrix[hiddenLayers.getSize()+2];
    layers[0] = input;

    //hiddenLayer output
    for (int i = 1; i < hiddenLayers.getSize()+1; i++) {
        layers[i] = calculateOutput(input,weights[i-1], biases[i-1]);
        input = layers[i];
    }
    //SoftMaxLayer output

    layers[hiddenLayers.getSize()+1] = calculateSoftMaxOutput(input, weights[hiddenLayers.getSize()], biases[hiddenLayers.getSize()]);

    //Back propagation
    SimpleMatrix target = MathUtil.fromArray(targetArray);



    for (int j = hiddenLayers.getSize()+1; j > 0; j--) {
        SimpleMatrix error = target.minus(layers[j]);
        SimpleMatrix gradient = (j==hiddenLayers.getSize()+1) ? calculateSoftMaxGradient(layers[j], error) : calculateGradient(layers[j], e
        biases[j-1] = biases[j-1].plus(gradient);

        SimpleMatrix delta = calculateDeltas(layers[j-1], gradient);
        weights[j-1] = weights[j-1].plus(delta);
        SimpleMatrix previousError = weights[j-1].transpose().mult(error);
        target = previousError.plus(layers[j-1]);
    }

}
```

Save network

The network will be saved through saveNN() method, generating two separated file weights.csv and bias.csv.

```
public static void saveNN(int inputNodes, int hiddenlayers, int hdNodes, int outputNodes, SimpleMatrix[] weights, SimpleMatrix[] bias){
    //save NN layer info
    StringBuilder builder = new StringBuilder();
    builder.append(inputNodes+",").append(hiddenlayers+",").append(hdNodes+",").append(outputNodes+",")
    .append("\n");

    //save weights
    for(SimpleMatrix sm:weights){
        double[][] wArray = MathUtil.toArray(sm);
        for(int i=0;i<wArray.length;i++){
            for(int j=0;j<wArray[0].length;j++){
                builder.append(wArray[i][j]+"");
                if(j < wArray[0].length - 1)//if this is not the last row element
                    builder.append(",");//then add comma (if you don't like commas you can use spaces)
            }
            builder.append("\n");
        }
    }
    bfWritter( name: "weights.csv", builder);
    //save bias
    StringBuilder builder2 = new StringBuilder();
    for(SimpleMatrix sm:bias){
        double[][] bArray = MathUtil.toArray(sm);
        for(int i=0;i<bArray.length;i++){
            for(int j=0;j<bArray[0].length;j++){
                builder2.append(bArray[i][j]+"");
                if(j < bArray[0].length - 1)//if this is not the last row element
                    builder2.append(",");//then add comma (if you don't like commas you can use spaces)
            }
            builder2.append("\n");
        }
    }
    bfWritter( name: "bias.csv", builder2);
```

Prediction

The predict action is actually the same as the feed forward part of the training process, but with adjusted weights and bias.

```
    public double[] predict(double[] inputArray) {
        SimpleMatrix output = MathUtil.fromArray(inputArray).divide(255);
//        SimpleMatrix output = MathUtil.fromArray(inputArray);
        for (int i = 0; i < hiddenLayers.getSize(); i++) {
            output = calculateOutput(output, weights[i], biases[i]);
        }
        output = calculateSoftMaxOutput(output, weights[hiddenLayers.getSize()], biases[hiddenLayers.getSize()]);
        return MathUtil.getColumn(output,  col: 0);

    }
```

Result and discussion

After 10 iterations of training, we got the following results shown in fig.1-6. As we can see in the results, using tanh function as activation function can improve 1% total precision compared with sigmoid function. And the total precision can reach as high as 98% when predicting the trained dataset. Even with untrained dataset(inputs are not included in the train dataset), our ANN can still get a precision of 94%. In addition, 5 ,8, and 9 are the most difficult numbers to recognize, and 0 is the most easily recognized number. Finally, compared with one hidden layer ANN, two-hidden-layer ANN can improve 5% total prediction precision.

Fig.1 prediction precision under different conditions

A	B	C	D	E	F	G	H	I	J	K	L
Result	target0	target1	target2	target3	target4	target5	target6	target7	target8	target9	
actual0	197	0	0	0	0	0	0	0	0	0	1
actual1	0	222	0	0	0	0	0	0	1	0	0.9955157
actual2	0	0	185	0	0	0	0	1	0	1	0.98930481
actual3	0	0	2	195	0	0	0	0	1	0	0.98484848
actual4	0	0	0	0	224	0	1	0	0	3	0.98245614
actual5	0	0	1	1	1	160	0	0	1	2	0.96385542
actual6	0	0	1	0	0	0	215	0	0	0	0.99537037
actual7	0	0	0	1	0	0	0	201	0	0	0.9950495
actual8	0	2	0	0	1	2	0	0	175	2	0.96153846
actual9	0	1	1	2	0	0	0	0	1	197	0.97524752
	1	0.98666667	0.97368421	0.9798995	0.99115044	0.98765432	0.99537037	0.9950495	0.97765363	0.96097561	0.9850075

Fig.2 Tanh, 2 hidden layers, test data included in training data

Result	target0	target1	target2	target3	target4	target5	target6	target7	target8	target9	
actual0	196	0	2	2	0	1	0	0	1	0	0.97029703
actual1	0	220	0	0	0	1	0	1	2	0	0.98214286
actual2	0	0	179	1	0	0	0	4	0	1	0.96756757
actual3	0	0	2	192	0	3	0	0	2	1	0.96
actual4	0	1	0	0	220	0	0	1	1	4	0.969163
actual5	0	1	2	2	0	152	0	0	1	0	0.96202532
actual6	0	0	0	0	2	1	216	0	2	0	0.97737557
actual7	0	1	1	1	0	0	0	193	0	2	0.97474747
actual8	1	2	1	0	1	4	0	0	170	0	0.94972067
actual9	0	0	3	1	3	0	0	3	0	197	0.95169082
	0.99492386	0.97777778	0.94210526	0.96482412	0.97345133	0.9382716	1	0.95544554	0.94972067	0.96097561	0.96701649

Fig.3 Sigmoid, 2 hidden layers, test data included in training data

A	B	C	D	E	F	G	H	I	J	K	L
Result	target0	target1	target2	target3	target4	target5	target6	target7	target8	target9	
actual0	189	0	2	1	1	3	3	0	1	1	0.94029851
actual1	0	216	2	0	0	3	0	0	1	0	0.97297297
actual2	1	0	169	5	1	1	1	4	1	3	0.90860215
actual3	0	1	3	174	0	3	1	1	4	2	0.92063492
actual4	0	2	2	0	210	1	3	2	2	5	0.92511013
actual5	2	1	0	7	3	143	2	1	1	1	0.88819876
actual6	3	1	3	1	4	1	204	1	1	0	0.93150685
actual7	0	0	1	5	0	0	0	191	1	4	0.94554455
actual8	2	3	3	2	2	6	2	0	166	3	0.87830688
actual9	0	1	5	4	5	1	0	2	1	186	0.90731707
	0.95939086	0.96	0.88947368	0.87437186	0.92920354	0.88271605	0.94444444	0.94554455	0.9273743	0.90731707	0.92353823

Fig.4 Tanh, 1 hidden layer, test data included in training data

A	B	C	D	E	F	G	H	I	J	K	L
Result	target0	target1	target2	target3	target4	target5	target6	target7	target8	target9	
actual0	192	0	2	1	2	1	1	0	0	1	0.96
actual1	0	217	0	2	0	0	0	3	3	0	0.96444444
actual2	0	0	178	3	0	0	1	3	2	1	0.94680851
actual3	0	2	3	181	0	3	0	1	4	2	0.92346939
actual4	0	0	1	0	217	0	2	2	3	8	0.93133047
actual5	1	1	0	4	0	148	2	1	4	3	0.90243902
actual6	2	1	1	2	2	3	208	0	2	0	0.94117647
actual7	0	0	0	3	1	2	0	188	0	5	0.94472362
actual8	2	3	1	1	2	3	1	0	160	2	0.91428571
actual9	0	1	4	2	2	2	1	4	1	183	0.915
	0.97461929	0.96444444	0.93684211	0.90954774	0.96017699	0.91358025	0.96296296	0.93069307	0.89385475	0.89268293	0.93553223

Fig.5 Sigmoid, 1 hidden layer, test data included in training data

Result	target0	target1	target2	target3	target4	target5	target6	target7	target8	target9	
actual0	193	0	1	1	2	4	1	0	0	1	0.95073892
actual1	0	217	0	0	0	1	0	0	3	0	0.98190045
actual2	0	0	175	2	2	4	1	3	1	1	0.92592593
actual3	0	2	3	183	0	2	0	0	3	3	0.93367347
actual4	0	0	1	0	216	0	3	2	3	7	0.93103448
actual5	2	0	2	8	0	142	0	0	2	2	0.89873418
actual6	2	1	1	1	1	2	210	0	3	0	0.95022624
actual7	0	1	0	3	1	0	0	194	0	2	0.96517413
actual8	0	3	4	1	1	5	1	1	163	1	0.90555556
actual9	0	1	3	0	3	2	0	2	1	188	0.94
	0.97969543	0.96444444	0.92105263	0.91959799	0.95575221	0.87654321	0.97222222	0.96039604	0.91061453	0.91707317	0.94002999

Fig.6 Tanh, 2 hidden layers, test data excluded from training data

Conclusion

According to the results above, our ANN has been proved to be an effective neural network, the prediction precision can reach as high as 98% with MNIST dataset, where the train data size is about 40,000 and the size of test data is 2,000.