Artificial Neural Network Final Report

Forecasting of Silver Spot Price

TEAM LXO: Zhang Yibin Duan Chenggu Guan Shuke

12/20/2019

1.Brief Introduction

This project will analyze several factors which will affect the silver price's fluctuating. Actually, there are a lot of factors that could have a great effect on the silver price. However, we would like to find some vital and stable factors to analyze it in a more concise way as well as to test if neural network could be a tool to deal with the silver price prediction. In this case, based on the mechanism of our C++ program, we distinguished our dependent variable into increase and decrease which are represented by 1 and 0 in dataset. Then we used several relevant factors to run our program to see if it can be approximately converged to a typical value. Thus, from the outcome, we can conclude if our program is successful and if our variables are sufficient. The sufficient model means we can use these factors to predict the direction of fluctuation of silver price correctly. The model uses the lagged phase of factors to predict the next phase silver's price. Therefore, if the accuracy is decent, then we can conclude this model is capable of forecasting.

In the very first place, there will be several variables included in the model and their symbols are as below:

Variables
CPI
USD Index
Gold Spot Price
Crude Oil Spot Price
Explained Variables
Silver Spot Price (Next Phase)

2.Data Processing

2.1 Variables

In the case, we sample the monthly silver spot price to test our program. Then, we pick up four factors in order to evaluate the silver spot price. Here we choose four variables for our program to train.

Firstly, one is the CPI. We assume CPI has an apparent effect on the silver spot price since silver purchasing is also a consumption. Thus, these two should be related. Then we use USD index as the second factors. All these commodities are priced by US dollar. Therefore, the currency condition is vital as well. Also, we use gold spot price and crude oil spot price as two factors to reflect the market condition. Although, Oil price is related to USD, we can still use it to measure since we use the lagged phase to forecast. The accuracy is the only standard we have to focus on.

2.2 Data Reprocessing

After the pick-up of our variables and collect the primal data, we should reprocess our data.

Firstly, we go over our explained variables. We could not plug in the increase rate or the pure price into the program since the precision cannot be reached. Thus, we classify the data into 2 categories in terms of the value of increase or not. If the price goes up, it will be 1, otherwise 0.

Interval	**Value**
Increase	y=1
Decrease	y=0

Then for the other four variables, we would standardize the data using the formula below:

$$\hat{x} = \frac{x - \bar{x}}{\sigma}$$

This process is important for a couple of reasons:

1. Facilitates initialization
2. Avoid numerical problems for updating gradient values
3. Conducive to the adjustment of the learning rate value
4. Speed up finding the optimal solution

Then we will go to the designation part of ANN process.

3.ANN Designation

3.1 Overview

The main idea of ANN is let data go through the neurons of each layer in the network, calculate the error, then adjust their weights according to the partial derivatives of each neurons. In order to complete these calculations above, we break down neural network into layers, and break down the layer into neurons, then combination all of them.

3.2 Neuron

For each neuron in neural network, take input in the form as (1xN) vector and multiplies with their own weights, then pass through sigmoid function. We can treat the first part calculation as one-dimension matrix multiply calculation. For all neurons in one layer, we can combination them as a matrix to perform multiply calculation with one vector. Therefore, we decide not to implement single neural calculation but combine them in a matrix format and implement in layers.

3.3 Layers

As we have discussed above, we let layer to be our minimum unit to perform neural network. At first, for each layer, we should randomly generate their weights for each neuron. Then we should implement forward process, backward process and update weight process for each layer.

For forward process, we simply calculate forward result through matrix multiply calculation and sigmoid activated function. Meanwhile we should also store the derivative result of sigmoid function to perform backward process.

The sigmoid function is: $f(z) = \frac{1}{1+e^{-x}}$

The derivative function of sigmoid is: $f'(z) = f(z)(1-f(z))$

The forward function is: $a^k = \sigma(w^k a^{k-1} + b^k)$

Where **a** means the value of kth layer, **b** means the bias in **k**th layer, **w** means weights and σ means sigmoid function.

In backward process, to implement chain rule, for each layer should take the next layer partial derivative result as input. According to the chain rule, we can calculate one middle derivative value by next layer partial derivative result multiplied with current layer derivative value of sigmoid. This value can be used to calculate delta of weight and delta result for the former layer. We should also store the delta result for the former layer.

$$\frac{\partial \Delta E}{\partial w_{ji}^{(k)}} = \frac{\partial E}{\partial v_j^{(k)}} \cdot \frac{\partial v_j^{(k)}}{\partial w_{ji}^{(k)}} = \frac{\partial E}{\partial v_j^{(k)}} \cdot \frac{\partial v_j^{(k)}}{\partial w_{ji}^{(k)}} (\sum_{s=1}^{n_{k-1}} w_{js}^{(k)} x_s^{(k-1)}) = -\delta_j^{(k)} x_i^{(k-1)}$$

In update weight process, we use the middle derivative value and current layer input to calculate delta of weight, then we update weights by learning rate which set by user. We can perform update weight process after we calculate the delta result for the former layer.

Training set: $[x, \bar{y}] = [(x_1, x_2, \ldots, x_{n_0}), (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_{n_k})^T]$ where x is input and y is output.

Weight update:

$$w_{ji}^{(k)}(s+1) = w_{ji}^{(k)}(s) - \eta \frac{\partial \Delta E}{\partial w_{ji}^{(k)}}$$

$E = \frac{1}{2}\sum_{i=1}^{n_k}(\bar{y}_t - y)^T(\bar{y} - y)$ is the error in the matrix form.

3.4 Neural Network

For now, we have defined layer for our neural network, it's time to sum them up. For the first layer, the input is the original data, we should set the number of rows as our original data's dimension. For the last layer, it should output the same dimension as our target value. Therefore, we should add an output layer at the end of network we've have built before. The number of columns should be the same as target value's dimension.

In this case, we simply implement SGD algorithm to perform weight's update. We also set maximum tolerance and maximum loop number to control our training process.

4.ANN Implementation

4.1 Overview

According to the designation of the former section, we implement class for matrix, layers and neural network to perform functions as we've talked above. In addition, we should implement some manual process to perform training, testing and scoring.

4.2 Matrix

Matrix class has two main functions: generating weights and multiply calculation.

Initialization Weight：

```
// random initialize matrix
Matrix::Matrix(int r, int c): rows(r), columns(c){
    for (int i = 0; i < this->rows; ++i){
        vector<double> colValues;
        for (int j = 0; j < this->columns; ++j){
            colValues.push_back(rand() % 10000000 / double(10000000));
        }
        this->vals.push_back(colValues);
    }
}
```

Multiply Calculation:

```
// random initialize matrix
Matrix::Matrix(int r, int c): rows(r), columns(c){
    for (int i = 0; i < this->rows; ++i){
        vector<double> colValues;
        for (int j = 0; j < this->columns; ++j){
            colValues.push_back(rand() % 10000000 / double(10000000));
        }
        this->vals.push_back(colValues);
    }
}
```

In addition, we also implement “toString()” function to transform into format output.

```
// output format string
string Matrix::toString() const{
    ostringstream output;
    for (int i = 0; i < rows; ++i){
        for (int j = 0; j < columns; ++j){
            output << "Neural " << (j+1) << " : ";
            output << fixed << setprecision(4) << setw(10) << this->vals.at(i).at(j) << "\t";
        }
        output << endl;
    }
    output << endl;
    return output.str();
}
```

4.3 Layer

Layer class performs many important calculations. Firstly, we add one bias into our input data, the value of bias can be modified though weight updating:

```
// add bias 1 at the end of input data
void Layer::setInput(vector<double> v){
    v.push_back(1);
    this->input = v;
}
```

We initialize our layer matrix with number of previous layer output plus one and number of current layer neurons. Each column means one neuron for current input with

bias:

```
// initialize layer
Layer::Layer(int numPreOutput, int neuralNum){
    this->neuralNum = neuralNum;
    this->weight = new Matrix(numPreOutput + 1, neuralNum);
}
```

Then we define our activated function and derivative activated function:

```
// calculate activated function sigmoid
double Layer::activated(double val){
    return 1 / (1 + exp(-val));
}

// calculate derivative of activated function
double Layer::derivated(double val){
    return activated(val) * (1 - activated(val));
}
```

Use matrix multiply function and active function to perform forward process:

```
// one layer feed forward
vector<double> Layer::feedForward(){
    vector<double> res;
    res = this->weight->vectorMultiply(this->input);
    for (int i = 0; i < res.size(); ++i){
        res[i] = activated(res[i]);
    }
    return res;
}
```

Store derivative sigmoid result:

```
// calculate layer derivative of sigmoid
vector<double> Layer::derivatedVector(){
    vector<double> res;

    res = this->weight->vectorMultiply(this->input);
    for (int i = 0; i < res.size(); ++i){
        res[i] = derivated(res[i]);
    }
    return res;
}
```

Calculate middle derivative value with derivative sigmoid result and next layer delta:

```
// calculate layer's derivative before sigmoid to update weight
vector<double> Layer::middleDerivatedValue(vector<double> nextLayerDelta){
    vector<double> res;
    vector<double> currDerivated = derivatedVector();

    for (int i = 0; i < currDerivated.size(); ++i){
        res.push_back(currDerivated[i] * nextLayerDelta[i]);
    }
    return res;
}
```

Calculate current layer delta to pass to the previous layer

```
// calculate layer's derivative to pass to the former layer
vector<double> Layer::layerDelta(vector<double> nextLayerDelta){
    vector<double> middleDelta = middleDerivatedValue(nextLayerDelta);
    vector<double> res;
    for (int i = 0; i < weight->getNumRows(); ++i){
        double temp = 0.0;
        for (int j = 0; j < weight->getNumCols(); ++j){
            temp += weight->getValue(i, j) * middleDelta[j];
        }
        res.push_back(temp);
    }
    return res;
}
```

Use middle Delta, learning rate and current layer input to update weight:

```
// use middleDelta, learning rate and current layer input to update weight
void Layer::updateWeight(vector<double> nextLayerDelta, double lr){
    learningRate = lr;
    vector<double> middleDelta = middleDerivatedValue(nextLayerDelta);
    for (int i = 0; i < weight->getNumRows(); ++i){
        for (int j = 0; j < weight->getNumCols(); ++j){
            weight->setValue(i, j, weight->getValue(i, j) - learningRate * middleDelta[j] * input[i]);
        }
    }
}
```

4.4 Neural Network

We initialize Neural Network only one time and change parameters by setting functions, so we take no parameter as input of initialization function. We perform training only after setting parameters. The setting functions as following:

```
// set parameters functions
void setLayers(std::vector<int> n) {this->neuralsInLayers = n;}
void setLearningRate(double r) {this->learningRate = r;}
void setLossToleration(double l) {this->lossToleration = l;}
void setMaxLoops(int loops) {this->maxLoops = loops;}
```

After we setting those parameters, we can initialize our weights for layers. For right

now we only take input data as parameters, then we can build up all hidden layers:

```
void NeuralNetwork::initialWeight(std::vector<std::vector<double> > data){
    layers.clear();
    // according to the input size decide first hidden layer size
    int inputNeurals = data[0].size();
    vector<int>::iterator it = neuralsInLayers.begin();
    neuralsInLayers.insert(it, inputNeurals);

    // generate hidden layers
    for (int i = 0; i < neuralsInLayers.size() - 1; ++i){
        Layer *layer = new Layer(neuralsInLayers[i], neuralsInLayers[i+1]);
        layers.push_back(layer);
    }
}
```

For the forward function in Neural Network, we put one-dimension vector input and one-dimension target as parameters and go through one-time processing to get current result. Then we calculate loss value, and calculate loss derivative value of last layer:

```
void NeuralNetwork::feedForward(vector<double> currInput, vector<double> target){
    vector<double> temp = currInput;

    // feedforward
    for (int i = 0; i < layers.size(); ++i){
        layers[i]->setInput(temp);
        temp = layers[i]->feedForward();
    }
    L2Loss = 0.0;
    vector<double> tempLoss;

    // calculate error and loss derivative
    for (int i = 0; i < temp.size(); ++i){
        double lossDelta = target[i] - temp[i];
        tempLoss.push_back(-2 * lossDelta);
        L2Loss += lossDelta * lossDelta;
    }
    currLoss.clear();
    currLoss = tempLoss;
}
```

In backward function, we calculate each layer delta from last to beginning, and for each layer we update layer weight after we record current layer delta:

```
void NeuralNetwork::backward(){
    vector<double> nextLayerDelta;
    nextLayerDelta = currLoss;
    vector<double> currLayerDelta;
    // calculate delta layer and update weight
    for (int i = layers.size() - 1; i >= 0 ; --i){
        currLayerDelta = layers[i]->layerDelta(nextLayerDelta);
        layers[i]->updateWeight(nextLayerDelta, learningRate);
        nextLayerDelta = currLayerDelta;
    }
}
```

Then we can perform training function. We take data value and target value from dataset. We have not defined our output layer yet. Since we have target value, we can add output layer to our layers firstly. According to user's input for maximum loops and toleration, we can perform training. In addition, we output current loss result after some fixed training loops:

```
void NeuralNetwork::training(vector<vector<double> >data, vector<vector<double> > t){
    // initialize weight, set train label and target label
    this->initialWeight(data);
    int loop = 0;
    this->input = data;
    this->target = t;

    int numerLastLayerNeural = target[0].size();
    int numLastInput = neuralsInLayers[neuralsInLayers.size() - 1];

    // add output layer at last
    Layer *lastLayer = new Layer(numLastInput, numerLastLayerNeural);
    layers.push_back(lastLayer);

    // train and display error
    while (loop <= maxLoops){
        loop++;
        double oneLoopLoss = 0.0;
        for (int i = 0; i < input.size(); ++i){
            feedForward(input[i], target[i]);
            backward();
            oneLoopLoss += L2Loss;
        }
        oneLoopLoss = oneLoopLoss / input.size();
        if (loop % 100 == 0 || loop == 1){
            cout << "Current loop: " << loop << "=======> Loss: " << oneLoopLoss << endl;
        }
        if ((oneLoopLoss < 0 && oneLoopLoss > -lossToleration) || (oneLoopLoss > 0 && oneLoopLoss < lossToleration) ) {
            cout << "Current loop: " << loop << "=======> Loss: " << oneLoopLoss << endl;
            break;
        }
    }
}
```

After we complete our training process, we should implement prediction process and give our result an accuracy score. To get accuracy score, we transform our result into [0, 1] by pass through a threshold value as 0.5:

```
std::vector<double> NeuralNetwork::predict(vector<double> currInput){
    vector<double> temp = currInput;
    for (int i = 0; i < layers.size(); ++i){
        layers[i]->setInput(temp);
        temp = layers[i]->feedForward();
    }
    // set 0.5 shreshold to get prediction label
    for (int i = 0; i < temp.size(); ++i){
        if (temp[i] > 0.5){
            temp[i] = 1;
        }else{
            temp[i] = 0;
        }
    }
    return temp;
}

void NeuralNetwork::score(vector<vector<double> > inputs, vector<vector<double> > yTrue){
    int numOfFail = 0;
    // sum up wrong prediction number
    for (int i = 0; i < inputs.size(); ++i){
        vector<double> yPred = predict(inputs[i]);
        if (yPred != yTrue[i]){
            numOfFail++;
            continue;
        }
    }
```

Finally, we transform our neural network weight result into format string to output or write into file:

```
// transform weight into string
string NeuralNetwork::toString() const{
    ostringstream outstring;
    for(int i = 0; i < layers.size(); ++i){
        if(i == (layers.size()-1)){
            outstring << "Output Layer: " << endl;
            outstring << layers[i]->weightToString();
        }else{
            outstring << "Hidden Layer " << (i+1) << " :" << endl;
            outstring << layers[i]->weightToString();
        }
    }
    return outstring.str();
}
```

4.5 Main Function

The main function has several functions: retrieve training and test data, asking for user to set parameters and write final results into file:

For the first part we define some parameters to get user's input and dataset, we also create a neural network class to implement all training and test work:

```
NeuralNetwork *nn = new NeuralNetwork();
double learningRate;
double loss;
int loops;
int numberOfLayers;
bool isFinished = false;

char trainName[] = "TrainData.csv";
char testName[] = "TestData.csv";

Data *train = new Data(trainName);
Data *test = new Data(testName);

// get normalized train data set
vector<vector<double> > trainData = train->getInput();
vector<vector<double> > trainTarget = train->getTarget();
// get normalized test data set
vector<vector<double> > testData = test->getInput();
vector<vector<double> > testTarget = test->getTarget();
```

Then we get user's input and perform training process. For each training, we output the accuracy result of training dataset. If user decide to finish his model, then the system will write his model into file. In the end, system will perform model on test dataset:

```
do{
    // parameters settings
    cout << "Welcome to Silver Prediction Training System\n";
    cout << "Please enter learning rate: ";
    cin >> learningRate;
    nn->setLearningRate(learningRate);
    cout << "Please enter loss toleration: ";
    cin >> loss;
    nn->setLossToleration(loss);
    cout << "Please enter max traninig loop: ";
    cin >> loops;
    nn->setMaxLoops(loops);
    cout << "Please enter the number of hidden layers in NeuralNetwork: ";
    cin >> numberOfLayers;
    vector<int> layerNeurals;
    for(int i = 0; i < numberOfLayers; ++i){
        cout << "Please enter the number of neurons in hidden layer " << (i+1) << " : ";
        int n;
        cin >> n;
        layerNeurals.push_back(n);
    }
    nn->setLayers(layerNeurals);

    // train training data set
    nn->training(trainData, trainTarget);
    cout << "Train data:" << endl;
    nn->score(trainData, trainTarget);

    // finish training, and output weights
    cout << "Finish Training? (y/n)";
    string finish;
    cin >> finish;
    if(finish == "y"){
        isFinished = true;
        cout << nn->toString();

        ofstream outfile;
        outfile.open("FinalWeight.txt");

        outfile << nn->toString() << endl;
        outfile.close();
    }
}while(!isFinished);

// output test results
cout << "Test data:" << endl;
nn->score(testData, testTarget);
```

4.6 Addition

We also define data class to read data and normalize our data.

Firstly, we read data from .csv file and store data into struct data structure, then put data into vector for each category. We also retrieve target value:

```cpp
// get file data
Data::Data(char filename[]){
    csvdata intp;
    csvtarget tgt;
    FILE* fp;
    fp = fopen(filename, "r");
    while (1) {
        fscanf(fp, "%lf,%lf,%lf,%lf,%lf", &tgt.target1, &intp.cpi, &intp.usd, &intp.xau, &intp.oil);
        // cout << intp.cpi << endl;
        incsv.push_back(intp);
        tcsv.push_back(tgt);
        if (feof(fp)) break;
    }
    fclose(fp);
}

// get normalize data
vector<vector<double> > Data::getInput(){
    vector<double> cpiList;
    vector<double> usdList;
    vector<double> xauList;
    vector<double> oilList;

    for (int i = 0; i < incsv.size(); ++i){
        cpiList.push_back(incsv[i].cpi);
        usdList.push_back(incsv[i].usd);
        xauList.push_back(incsv[i].xau);
        oilList.push_back(incsv[i].oil);
    }

    cpiList = normalizeData(cpiList);
    usdList = normalizeData(usdList);
    xauList = normalizeData(xauList);
    oilList = normalizeData(oilList);

    for (int i = 0; i < cpiList.size(); ++i){
        vector<double> singleInput;
        // cout << cpiList[i] << endl;
        singleInput.push_back(cpiList[i]);
        singleInput.push_back(usdList[i]);
        singleInput.push_back(xauList[i]);
        singleInput.push_back(oilList[i]);

        input.push_back(singleInput);
    }

    return input;
}
```

```cpp
// get target label
vector<vector<double> > Data::getTarget(){
    vector<double> t1;
    vector<double> t2;

    for (int i = 0; i < tcsv.size(); ++i){
        t1.push_back(tcsv[i].target1);
    }

    for (int i = 0; i < t1.size(); ++i){
        vector<double> singleTarget;
        singleTarget.push_back(t1[i]);
        target.push_back(singleTarget);
    }
    return target;
}
```

Use the formula discussed in **section 2**, we calculate average value and standard deviation for each category then normalize each one:

```
// normalize data
vector<double> Data::normalizeData(vector<double> d){
    double avg = dataAverage(d);
    double std = dataStd(d);
    vector<double> res;
    for (int i = 0; i < d.size(); ++i){
        res.push_back((d[i] - avg) / std);
    }
    return res;
}

// calculate average
double Data::dataAverage(vector<double> d){
    double sum = 0.0;
    int l = d.size();
    for (int i = 0; i < l; ++i){
        sum += d[i];
    }
    return sum / l;
}

// calculate standard deviation
double Data::dataStd(vector<double> d){
    int l = d.size();
    double avg = dataAverage(d);
    double std = 0.0;
    for (int i = 0; i < l; ++i){
        std += pow(d[i] - avg, 2);
    }
    std = sqrt(std / l);
    return std;
}
```

5. Testing

For the first test, we set both learning rate and loss tolerance to 0.1, and maximum loop to 10000. Then we try different hidden layers to perform training process:

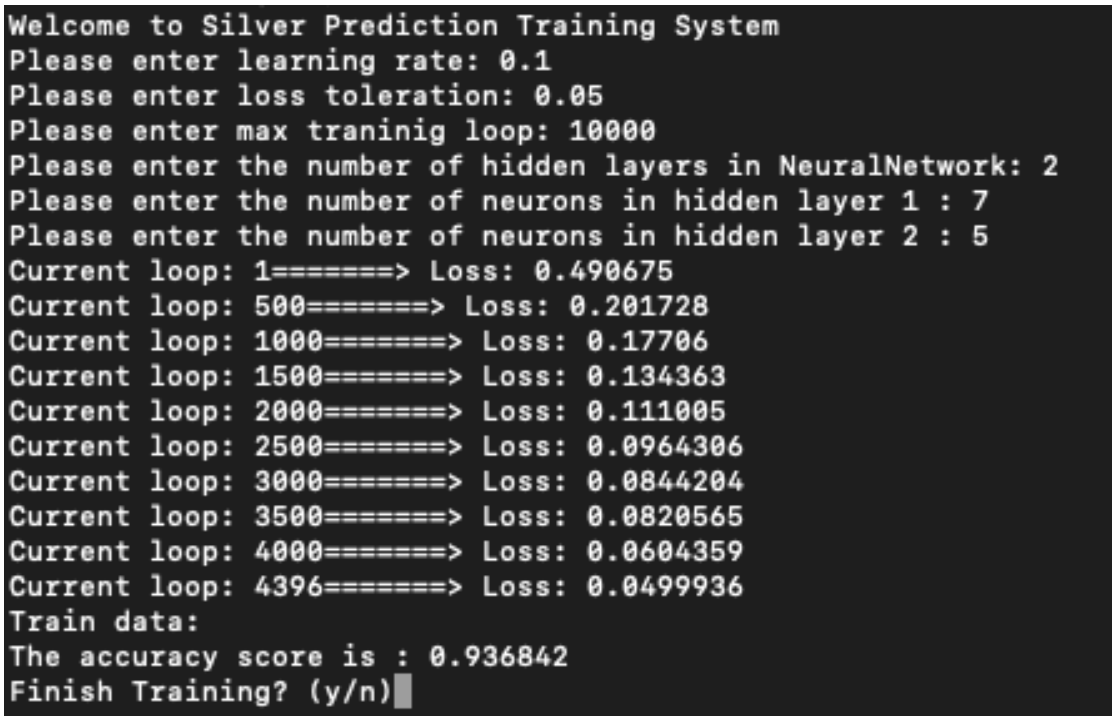
```
Welcome to Silver Prediction Training System
Please enter learning rate: 0.1
Please enter loss toleration: 0.1
Please enter max traninig loop: 10000
Please enter the number of hidden layers in NeuralNetwork: 1
Please enter the number of neurons in hidden layer 1 : 7
Current loop: 1=======> Loss: 0.32433
Current loop: 500=======> Loss: 0.197029
Current loop: 1000=======> Loss: 0.168569
Current loop: 1500=======> Loss: 0.134327
Current loop: 2000=======> Loss: 0.111573
Current loop: 2319=======> Loss: 0.099984
Train data:
The accuracy score is : 0.884211
Finish Training? (y/n)n
Welcome to Silver Prediction Training System
Please enter learning rate: 0.1
Please enter loss toleration: 0.1
Please enter max traninig loop: 10000
Please enter the number of hidden layers in NeuralNetwork: 2
Please enter the number of neurons in hidden layer 1 : 7
Please enter the number of neurons in hidden layer 2 : 5
Current loop: 1=======> Loss: 0.418998
Current loop: 500=======> Loss: 0.210971
Current loop: 1000=======> Loss: 0.183764
Current loop: 1500=======> Loss: 0.147266
Current loop: 2000=======> Loss: 0.115522
Current loop: 2310=======> Loss: 0.0999681
Train data:
The accuracy score is : 0.873684
Finish Training? (y/n)n
Welcome to Silver Prediction Training System
Please enter learning rate: 0.1
Please enter loss toleration: 0.1
Please enter max traninig loop: 10000
Please enter the number of hidden layers in NeuralNetwork: 3
Please enter the number of neurons in hidden layer 1 : 5
Please enter the number of neurons in hidden layer 2 : 7
Please enter the number of neurons in hidden layer 3 : 5
Current loop: 1=======> Loss: 0.377133
Current loop: 500=======> Loss: 0.224826
Current loop: 1000=======> Loss: 0.200895
Current loop: 1500=======> Loss: 0.189532
Current loop: 2000=======> Loss: 0.188021
Current loop: 2500=======> Loss: 0.175315
Current loop: 3000=======> Loss: 0.171731
Current loop: 3500=======> Loss: 0.17041
Current loop: 4000=======> Loss: 0.143941
Current loop: 4500=======> Loss: 0.158316
Current loop: 5000=======> Loss: 0.120543
Current loop: 5193=======> Loss: 0.0967925
Train data:
The accuracy score is : 0.873684
```

As we can see above, because the same loss tolerance, the simple model can be faster to converge. Then we set loss tolerance to 0.05:

```
Welcome to Silver Prediction Training System
Please enter learning rate: 0.1
Please enter loss toleration: 0.05
Please enter max traninig loop: 10000
Please enter the number of hidden layers in NeuralNetwork: 1
Please enter the number of neurons in hidden layer 1 : 7
Current loop: 1=======> Loss: 0.32433
Current loop: 500=======> Loss: 0.197029
Current loop: 1000=======> Loss: 0.168569
Current loop: 1500=======> Loss: 0.134327
Current loop: 2000=======> Loss: 0.111573
Current loop: 2500=======> Loss: 0.0944765
Current loop: 3000=======> Loss: 0.0820783
Current loop: 3500=======> Loss: 0.0710579
Current loop: 4000=======> Loss: 0.0631534
Current loop: 4500=======> Loss: 0.0577376
Current loop: 5000=======> Loss: 0.0541095
Current loop: 5500=======> Loss: 0.0511703
Current loop: 5706=======> Loss: 0.0499943
Train data:
The accuracy score is : 0.957895
Finish Training? (y/n)n
Welcome to Silver Prediction Training System
Please enter learning rate: 0.1
Please enter loss toleration: 0.05
Please enter max traninig loop: 10000
Please enter the number of hidden layers in NeuralNetwork: 2
Please enter the number of neurons in hidden layer 1 : 7
Please enter the number of neurons in hidden layer 2 : 5
Current loop: 1=======> Loss: 0.418998
Current loop: 500=======> Loss: 0.210971
Current loop: 1000=======> Loss: 0.183764
Current loop: 1500=======> Loss: 0.147266
Current loop: 2000=======> Loss: 0.115522
Current loop: 2500=======> Loss: 0.0931954
Current loop: 3000=======> Loss: 0.0599046
Current loop: 3449=======> Loss: 0.0499846
Train data:
The accuracy score is : 0.947368
```

```
Welcome to Silver Prediction Training System
Please enter learning rate: 0.1
Please enter loss toleration: 0.05
Please enter max traninig loop: 10000
Please enter the number of hidden layers in NeuralNetwork: 3
Please enter the number of neurons in hidden layer 1 : 5
Please enter the number of neurons in hidden layer 2 : 7
Please enter the number of neurons in hidden layer 3 : 5
Current loop: 1=======> Loss: 0.377133
Current loop: 500=======> Loss: 0.224826
Current loop: 1000=======> Loss: 0.200895
Current loop: 1500=======> Loss: 0.189532
Current loop: 2000=======> Loss: 0.188021
Current loop: 2500=======> Loss: 0.175315
Current loop: 3000=======> Loss: 0.171731
Current loop: 3500=======> Loss: 0.17041
Current loop: 4000=======> Loss: 0.143941
Current loop: 4500=======> Loss: 0.158316
Current loop: 5000=======> Loss: 0.120543
Current loop: 5500=======> Loss: 0.0825982
Current loop: 6000=======> Loss: 0.0803851
Current loop: 6500=======> Loss: 0.0792688
Current loop: 7000=======> Loss: 0.0733216
Current loop: 7500=======> Loss: 0.0720295
Current loop: 8000=======> Loss: 0.162117
Current loop: 8500=======> Loss: 0.138923
Current loop: 9000=======> Loss: 0.11191
Current loop: 9500=======> Loss: 0.106864
Current loop: 10000=======> Loss: 0.0967137
Train data:
The accuracy score is : 0.884211
```

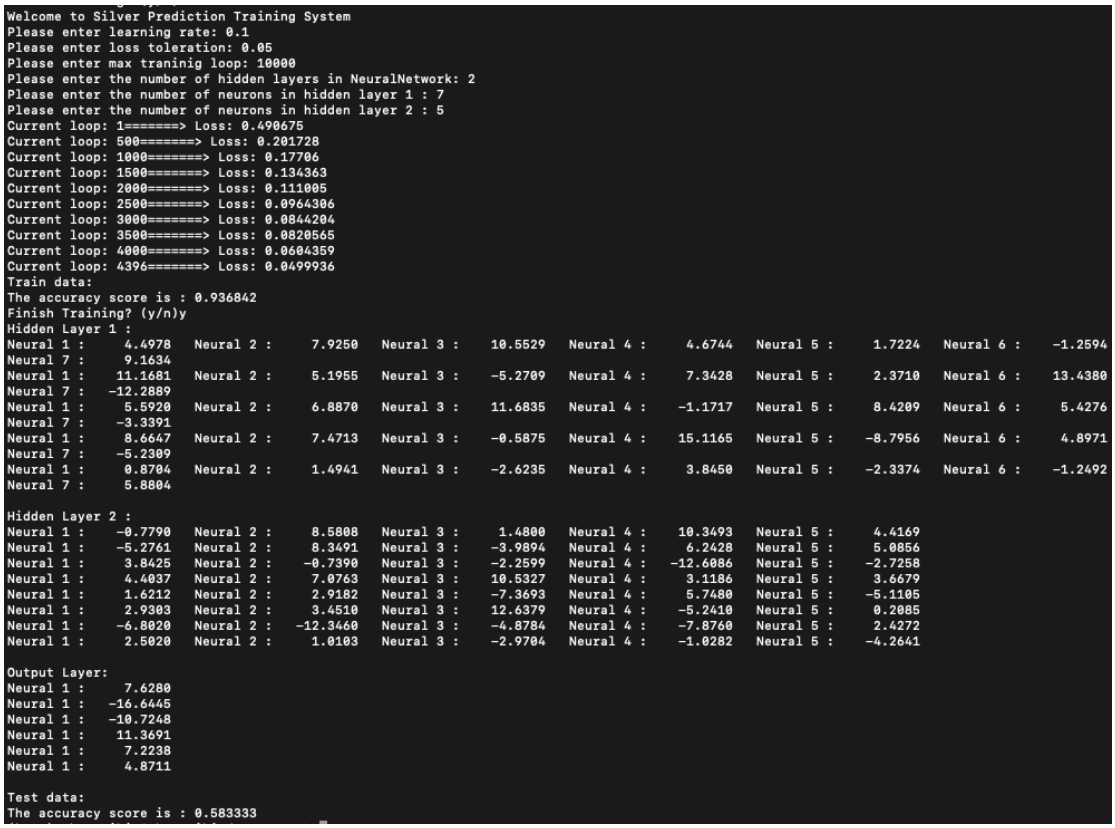
From the results above, we can observe that the most complex model can't pass over the loss tolerance but reverse learning for a while, it's should be a problem of large learning rate for this complicated model. For both 1 hidden layer and 2 hidden layer model, we get an average 0.95 accuracy on training dataset. The model with 2 hidden layers can be faster to converge, we try the model with 2 hidden layers again:



As we randomly initialize our model weights, the results will be a little different. We use this model to perform test data set.

6. Results

We use the last model as our final model to perform on the test data set. We also output the model weights on the screen and write them into file:



We set layer one with 7 neurons and layer two with 5 neurons, and the output layer only has one neuron according to the target label dimension. The original data has four dimensions, so the weight number of the hidden layer one is five (plus bias). The following layers all take previous output as input to automatically generate matrix.

The accuracy on the test dataset is 58.3%.

7. Remarks and Possible Future Improvement

As the test the result, the performance of our model is acceptable. We use the very basic artificial neural network model to evaluate the silver price using several variables. Due to the features of financial market, there will not be a well-rounded model which precisely forecast how price will go. Therefore, here we merely provide a practical trial for analyzing the direction price will go. In the real market, even the best model at a time will not last long to keep being the leader for which a dynamic adjustment is vital in trading. From the result, optimistically, our model has a plausible outcome for the time period from 2010-2018 regarding to monthly data and the test set says our model, to some degree, can forecast the tendency of silver price. But we would never know if

our model will be sufficient in the near future. Thus, we still have to keep verifying if this model performs well.

In addition, we also have a couple of ideas that would improve the performance of the model:

7.1. Expand Dataset

Our dataset is sort of restricted. We use the monthly data from 2010 to 2018. In order to keep the accuracy of model, we sample the monthly data to reduce volatility. We assume that if we took the daily data of all variables, the accuracy may not be as good as the above. However, daily data could be more practical in use. So finally, we use the accuracy to trade off with practical use. Presumable improvement is to extent the time to pick up more data and use weekly data. These are the possible ways for us to improve our result.

7.2. Include more variables

Here we only use four variables to forecast the price of silver. There could be some variables which are more effectively reflecting the price of silver. The financial market is so complicated that we are always facing an asymmetric information issue. Therefore, our variables are possibly not the most efficient one. So, we can test more variables to test if it could improve the accuracy of our model.

7.3. Use RNNs in Replace of ANN

In a vanilla neural network, a fixed size input vector is transformed into a fixed size output vector. Such a network becomes “recurrent” when you repeatedly apply the transformations to a series of given input and produce a series of output vectors. There is no pre-set limitation to the size of the vector. And, in addition to generating the output which is a function of the input and hidden state, we update the hidden sate itself based on the input and use it in processing the next input.

For a time-series financial dataset, using RNNs could be a better choice than the basic

ANN because of the high volatility and unpredictable incident in the financial market. The past data may not occur again in future.

7.4. Relation Instead of Forecasting

In this model, the logic from which we build the model is to forecast instead of a simple relation evaluation. As we all know, a relation model is less constrained than a forecasting model. The simultaneous stage can easily conclude a relation between several variables while a lagged stage has limited information to the next stage dependent variable. A relation dataset would definitely improve the accuracy. Therefore, using simultaneous stage data for all variables may be an improvement in terms of model setting.

8. Reference

[1] Rohitash Chandra,Shelvin Chand. Evaluation of co-evolutionary neural network architectures for time series prediction with mobile application in finance[J]. Applied Soft Computing,2016,49.

[2] F. Lenartz, J.-M. Beckers, J. Chiggiato et al.. Super-ensemble techniques applied to wave forecast: performance and limitations[J]. Ocean Science (OS), 2010, 6(2).

[3] Nemesio Rodríguez-Fernández, Patricia de Rosnay, Clement Albergel et al.. SMOS Neural Network Soil Moisture Data Assimilation in a Land Surface Model and Atmospheric Impact[J]. Remote Sensing, 2019, 11(11).

[4] David Kozak, Scott Holladay, Gregory E. Fasshauer. Intraday Load Forecasts with Uncertainty[J]. Energies, 2019, 12(10).

[5] S. W. D. Turner, S. W. D. Turner, J. C. Bennett et al.. Complex relationship between seasonal streamflow forecast skill and value in reservoir operations[J]. Hydrology and Earth System Sciences, 2017, 21.

[6]SILVER: https://www.tradingview.com/symbols/XAGUSD/

[7]GOLD: https://www.tradingview.com/symbols/XAUUSD/

[8]CPI:https://www.bls.gov/regions/midatlantic/data/consumerpriceindexhistorical_us_table.htm

[9]US Dollar Index: https://www.tradingview.com/symbols/TVC-DXY/

[10]OIL: https://www.tradingview.com/symbols/USOIL