Report of Project-3

徐茂然 14300180099

April 30, 2017

Abstract

This report illustrates how a classification task on *mnist* dataset is implemented and optimized with artificial neural network (ANN). An example Matlab program is given at first. However, it does not perform well. Different modifications are done to this model and are included in the following sections respectively. Those sections with a * means it is not mentioned in the instruction paper.

Contents

0 Initial Framework

The initial program uses a one-layer, ten units neural network based on gradient descent learning. In each iteration it randomly pick one sample to do stochastic gradient descent. Running the *example_neuralNetwork.m*, I get a test error 0.492000, with 100,000 iterations.

1 Network Structure

Firstly I tried to change merely the number of $nHidden$ in *example_neuralNetwork.m*, that is to expand the number of hidden units in a one-layer neural network. The number of hidden units vs. generalization performance graph was U-shaped, as is shown in the table. For one-layer neural network, I would choose 180 as the most well-performing numbers.

The next question is how many layers I should choose. Maybe one layer has not enough

hidden units No.	Test error
10(original)	0.632000
100	0.260000
120	0.232000
180	0.203000
300	0.224000

Table 1: Test error v.s the number of hidden units in one-layer network

capacity to cope with such a complicated set of input features.I tried a lot of times to find a good choice, but I failed. Two or three-layers neural network gives poor result in every iteration. And the test error with final model has not fall below 0.60 with 100,000 times of iterations. This is because the complication of multilayer net and we have to wait for its progressing till we use some method to avoid overfitting. Actually, I will always try for an optimal network structure in the following steps, as more implements are added to this network. I will turn back to it later.

2 Weight Initialization*

Another important hyper-parameter is the initial value of weight matrices. Several recipes have been raised(Lecun *et al.*,1998a;Gloot and Bergio), based on the idea that units with more inputs should have smaller weights value. The above two recipes recommended sampling from Uniform(-a,a), where a $=\sqrt{(6/fan_{in}+fan_{out})}$ for *tanh* units. Also, as was provided in the original example program, a zero-mean Gaussian distribution can be take to initialize the weights. A little modification should be made with the variance. I think 0.01 or 0.001 will be better. Here are some trials I made to initialize w in *example_neuralNetwork.m* (Since regularization had not been done yet, I just used one-layer model to see if the initialization worked well.)

```
% One-layer network with 100 hidden units
w = randn(nParams,1)/1000;
Test error with final model = 0.114000
```

```
w = unifrnd(-0.05,0.05,nParams,1);
Test error with final model = 0.105000
```

Notice that test error on the same structure in section 1 is 0.26000. A little change in initialization can significantly improve the accuracy of the model. However, it depends on the structure of our network. So I still have to work on it when I changed the shape of hidden layers or the activation function.

3 Learning Rate and Momentum Strength

Observing the performance of section 1, I found that valid error hardly change after about 10,000 times of iterations, due to overfitting and too large a step size. As the parameter approaching the optimal solution, a small step size should be taken. Else, our parameter may never reach the bottom of the object function. A simple step to do is to decrease the learning rate over periods. I upgraded learning rate α_t every $maxIter/5$ times (here is 20,000 times) to be $\alpha_t/5$.

A momentum can be used to avoid stucking at saddle point, which is not even a local optima. The learning procedure may stuck around a fake 'local optima'. A momentum is used to keep part of the velocity in the last iteration and help it rush to saddle points. A common choice of momentum strength is $\beta_t = 0.9$.

Some key Matlab codes are listed here(included in $example\_neuralNetwork.m$):

```
% initial stepSize=1e-3, beta =0.9
    if iter == 1
        w = w - stepSize*g;
        weights(:,2) = w;
    else
        w = w - stepSize*g+beta*(w-weights(:,2));
        weights(:, 1) = weights(:, 2);
        weights(:, 2) = w;
       if iter > maxIter/5
            stepSize = 1e-2/2;
            beta = 0;
        end
        if iter > maxIter*2/5
            stepSize = 1e-3;

        end
        if iter > maxIter*3/5
            stepSize = 1e-3/2;

        end
        if iter > maxIter*4/5
            stepSize = 1e-4/2;

        end
end
```

A new model is run, setting initial weights as $w = unifrnd(-0.01, 0.01, nParams, 1)$ and network structure as$nHidden = [100, 50, 20]$. I ran the model before learning rate and momentum is changed for comparison. The result is listed below. It seems that the model was trapped somewhere and because the step size is to big, it was stuck in a bottle neck.

After the modification, the model escaped those saddle points and got to a fairly good result.

Training iteration	Validation error
60000	0.192800
65000	0.198800
70000	0.183200
75000	0.166600
80000	0.174200
85000	0.160400
90000	0.168600
95000	0.152600
Test error with final model =	0.165000

Table 2: Three-layers-network before setting learning rate and momentum strength

Training iteration	Validation error
0	0.875600
20000	0.215600
40000	0.070400
60000	0.061400
80000	0.060000
Test error with final model =	0.052000

Table 3: Three-layers-network before setting learning rate and momentum strength

4 Vectorization

A common method is to increases times of iterations. This requires high speed and avoiding using explicit for-loops whenever possible. I changed those for-loops on c in 1 to $nLabels$. Some key Matlab codes are listed here(included in $MLP\_2.m$):

```
        %for c = 1:nLabels  gOutput(:,c) = gOutput(:,c) + err(c)*fp{end
            }'; end
        [fa, fb] = size(fp{end});
            gOutput = gOutput + repmat(err,fb,1).*repmat(fp{end}',1,
                nLabels);

        if length(nHidden) > 1
                        clear backprop
%               for c = 1:nLabels
%                   backprop(c,:) = err(c)*(sech(ip{end}).^2.*
    outputWeights(:,c)');
%                   gHidden{end} = gHidden{end} + fp{end-1}'*backprop(c
    ,:);
%               end
%               backprop = sum(backprop,1);
            backprop = sum(repmat(err',1,length(ip{end}))).*(repmat(sech
                (ip{end}).^2,nLabels,1).*outputWeights'),1);

            for h = length(nHidden)-2:-1:1
                backprop = (backprop*hiddenWeights{h+1}').*sech(ip{h
                    +1}).^2;
                gHidden{h} = gHidden{h} + fp{h}'*backprop;
```

```
            end

            backprop = (backprop*hiddenWeights{1}').*sech(ip{1}).^2;
            gInput = gInput + X(i,:)'*backprop;
        else
            % Input Weights
            %for c = 1:nLabels
            %    gInput = gInput + err(c)*X(i,:)'*(sech(ip{end}).^2.*
                outputWeights(:,c)');
            %end
            gInput = gInput+ repmat(X(i,:)',1,nLabels)*(repmat(sech(ip{
                end}).^2,nLabels,1).*(repmat(err,nHidden(end),1).*
                outputWeights)');
        end
```

I used a [100,100,100] neural network and 50,000 iterations to test if vectorization really works. Adding tic-toc outside the iteration to calculate time cost, I found that the elapsed time is 120.522906 seconds. seconds for a *for* loop and 35.175772 seconds for a vectorized model. The speed is really higher. For our own convenience, we had better do vectorization in the first place.

5 Regularization

There are mainly two ways to keep the model from overfitting, weight decay and early stop. A common way is to put a l_2 regularization of the weights, which is called *weight decay* in neural network. A Regularization parameter is nominated with λ, also a hyper-parameter in this model. Since l_2 regularization of $w = (w_1, w_2, ..., w_n)$ is in the form of

$$\frac{\lambda}{2}\sum_{i=1}^{n} w_i^2,$$

the derivation of this part is λw. This step is implemented by the following codes, included in the end of MLP_2

```
% Put Gradient into vector,with a l2 regularization
if nargout > 1
    lambda = 0.001;
    g = zeros(size(w));
    g(1:nVars*nHidden(1)) = gInput(:)+lambda*inputWeights(:);
    offset = nVars*nHidden(1);
    for h = 2:length(nHidden)
        g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1}+
            lambda*hiddenWeights{h-1};
        offset = offset+nHidden(h-1)*nHidden(h);
    end
end
    g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:)+lambda*
        outputWeights(:);
```

After several times of trials, I chose $\lambda = 0.0001$ for the [100,50,25] model. I gained a final test error of 0.052300, which is not significantly better than the one I got in section 3. However, when I used λ for an uninitialized network, it worked efficiently.

Training iteration	validation error
20000	0.284000
30000	0.278400
40000	0.275800
Test error with final model =	0.254000

Table 4: Three-layers-network before initialization and regularization

Training iteration	validation error
10000	0.108000
20000	0.087400
30000	0.088000
40000	0.086000
Test error with final model =	0.082000

Table 5: Three-layers-network after regularization

6 Softmax Regression

In the modification above, we were using *tanh* as the activation function in every layer. The graphic of *tanh* is shown in the figure.

We can see that *tanh* gives the prediction value near 1 or -1. However, if we change the

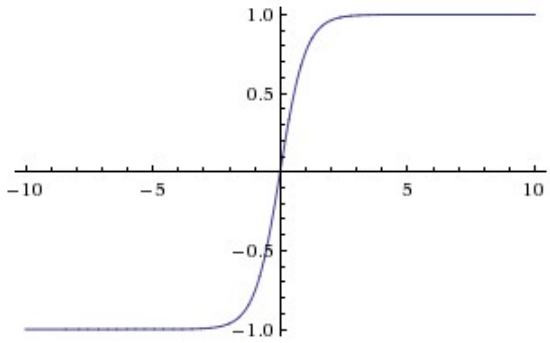

Figure 1: Functional graph of tanh(x)

activation function of the last layer to softmax, also known as multiple logistic, the ten labels can be interpreted as probability of each class. The softmax function is give as

$$p(y_i) = \frac{\exp(z_i)}{\sum_j^J \exp(z_j)}.$$

The graphic of softmax(logistic) function is shown in the figure.

I have to change the form of Output(y) in both $MLP\_2.m$ and $MLPclassificationPredict.m$.

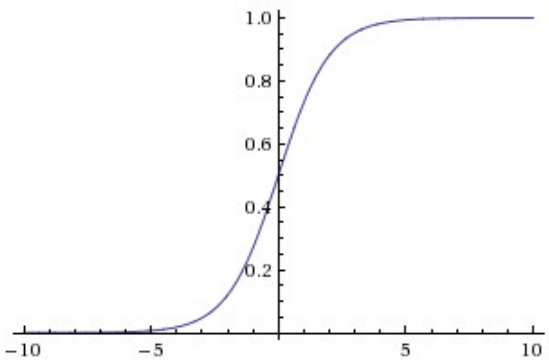

Figure 2: Functional graph of softmax

```
z = fp{end}*outputWeights;
y(i,:) = exp(z)/sum(exp(z));
```

I also have to change the binary setting to y from -1 to 0 in linearInd2Binary.m

```
y = zeros(n,nLabels)
```

The loss function has also changed. Instead of square error, cross entropy is used.

$$L = -\sum_{i=1}^{n}(t_i \log(p(y_i)))$$

$$\frac{\partial L}{\partial y_i} = \sum_{i=1}^{n}(p(y_i) - t_i)$$

So the final modification to the function is in $MLP\_2.m$, let $err = relativeErr$ instead of $2 * relativeErr$. Deviation in other layers are the same as before. I tried it on the model developed in section 5, and get the test error of 0.093000.

7 Bias

In the input layer, a 'ones' vector was appended to X to add a bias in the input weights. Then the first hidden layer can be computed by:

$$\mathbf{a}^{(1)} = \mathbf{w}\mathbf{x} + b.$$

Here b is called a bias. We can do the same thing to the other layers. That is to append a 'one' unit to each layer.

```
.....
for h = 2:length(nHidden)
        ip{h} = fp{h-1}*hiddenWeights{h-1};
 ip{h}(end) = 1;
.....
 end
```

Thus, if we set a 100-units layer, there are only 99 units in it for the last one is a bias unit which is not connected to any early layers.

Experiments were made on the new model. Changes on the result are not very obvious, so I ran the model several times. Adding a bias in each layer does do good to the model.

Bias in every layer(Y or N)	Test error
Y	0.048000
Y	0.046000
Y	0.045000
N	0.053000
N	0.054000
N	0.050000

Table 6: Test error with bias in all layers v.s. only 1^{st} layer

8 Dropout

Dropout is a different technique for regularization. In one of the earliest paper(Alex Krizhevsky *et al*. (2012).) that used dropout an explanation is given:

> This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.

In other words, dropout is to make sure that our network is robust enough that any lost neurons can not hurt it.
The algorithm of dropout can be illustrated as:

Dropout in neural network
1 In each hidden layer, drop neurons at possibility p=0.5.
2 Get both forward and backward propagation result by the dropout net.
3 Update appropriate weights and bias by the result above.
4 Repeat 1-3.
5 Halve the weights from the hidden neurons for prediction.

Note that step 5 requires an operation of halving the output weights. That is because the training step is completed when only half of the hidden layers takes part in it. When it comes to predict, twice as many hidden neurons will be active. It is necessary to compensate for that to times p to the weights.

Here is the concrete modification in MLP_2

```
% p = 0.5
......
ip{1} = X(i,:)*inputWeights;
    r{1} = ((rand(size(ip{1})))>1-p);
    ip{1}(end)=1;
    fp{1} = tanh(ip{1});
    fp{1} = r{1}.*fp{1};
    for h = 2:length(nHidden)
        ip{h} = fp{h-1}*hiddenWeights{h-1};
        r{h} = ((rand(size(ip{h})))>1-p);
        ip{h}(end) = 1;
        fp{h} = tanh(ip{h});
        fp{h} = r{h}.*fp{h};
    end
```

To test if it works, I used a [100,50,20] network without initialization and regularization. Before dropout, the model has converged after 15,000 times of iterations and test error with final model = 0.288000. After dropout, test error decreased to 0.234000. Actually, the effect on simpler network or regularized model is not obvious. And it even caused unstablility in the iterations.

9 Fine-tuning

After the training iterations a relatively optimal choice of weights and bias in each layer appears. Fine-tuning can be done based on the result of the network. For the last layer, instead of softmax regression, we can take a step like linear regression to get the 10 labels. Since linear regression loss function takes the form of square error, it has a closed-form solution. Then we changed the top layer as a convex optimal problem that is easy to solve. The steps can be described as follows:

Fine-tuning with linear regression
1 The last hidden layer gives n neurons (fp{end}). Restore them in matrix A.
2 Linear regression problem: $B = Wa$,
where B stands for numbers of instances $\times$numbers of output labels.
3 Solve $W = (A^T A)^{-1} A^T * B$;. This is the output weights for prediction.

The script *fine_tuning.m* implements those steps.

```
%fh stores fp{end} for every instance after training iterations
[yhat,fh] = MLPclassificationPredict(w,Xtest,nHidden,nLabels);
yExpanded1 = linearInd2Binary(ytest,nLabels);
wei = zeros(size(fh,2),nLabels);
    for j = 1:nLabels
        model = leastSquares(fh,yExpanded1(:,j));
        wei(:,j) = model.w;
    end
        yhat1 = fh*wei;
        [v,yhat1] = max(yhat1,[],2);
fprintf('Test error with final model = %f\n',sum(yhat1~=ytest)/t2);
```

Running *fine_tuning.m*, I found that the accuracy was raised greatly. Because it contains a linear regression based on a well trained network, the error can be around 11% without learning. And the error has been lowered to 0.021000. There is something to notice that the final valid error was always one percent higher than the final test error.

10 Expanding Training Samples

Bigger training samples can naturally lead to higher accuracy. Though we only have 5000 images in dataset, we can make some transformation on it to expand the size of dataset. Some possible transformations are rotation, translation, and resizing. I studied the shape of the images by reshape the digit matrix and print it out.

We can tell from the images above the numbers 8,3 and 5. The size of the written number

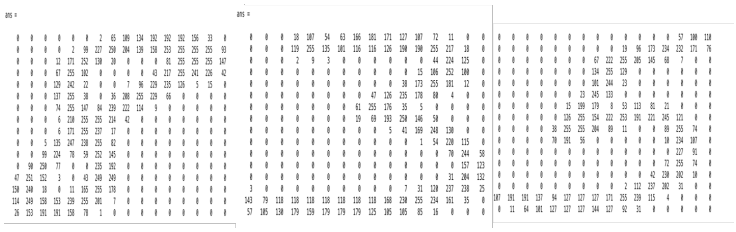

Figure 3: Shape of 16×16 hand written images

are as big as the 16×16 square. So translation is hard to make, otherwise the shape of numbers will be ruined. Also, direct rotation is not suitable here.

I tried to resize the images and rotate them by a small degree by using *makeform* and *imrisize* in Matlab.

```
rt = 5; %rotating times
n = size(X, 1); d = 16;
X0 = reshape(X', d, d, n);
y0 = y;
t = linspace(0.05, 0.2, rt);
for i = 1: rt
    X1 = zeros(d, d, n);
    for j = 1: n
        I = X0(:, :, j);
        tform = maketform('affine', [1 0 0; t(i) 1 0; 0 0 1]);
        J = imtransform(I,tform);
        X1(:, :, j) = imresize(J, [16, 16]);
    end
    X1 = reshape(X1, 256, n)';
    X = [X;X1];
    y = [y;y];
end
```

Thus the image has been rotated slightly.

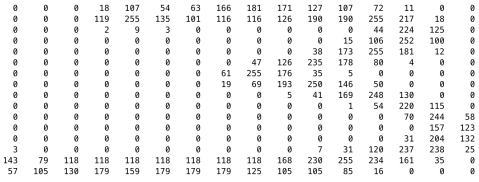

Figure 4: Unrotated image

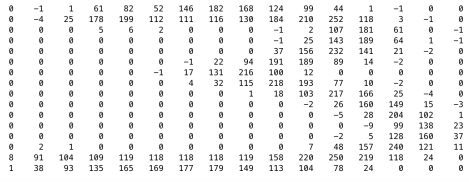

Figure 5: Rotated image

Another simple way is to add a random noise on each pixel.

```
for i =1:6
    X1 = X;
    X1 = X1 + randn(size(X))*10;
    X = [X;X1];
    y = [y;y];
end
```

As for the result, see the table.

Expanding Size (n times of the original one)	Test Error
No expanding	0.020200
Rotate 3	0.013600
Rotate 6	0.012800
Noise 3	0.020000
Noise 6	0.017800

Rotating after resizing outperformed Gauss noise. It did proved learning efficiency.

11 Mini-batch gradient descent*

For the sake of saving computing time, stochastic gradient descent is used. That is to pick only one sample in the dataset at one time to update the weights.

```
i = [ceil(rand*n)];
[f,g] = funObj(w,i);
```

Notice that if we change the size of i, we can get a mini-batch of samples for gradient descent.
To some extend, stochastic gradient descent saved a lot of time and space. But an appropriate some of samples for gradient descent would do great in accuracy. Full batch would be better but there are not enough RAM for it. Here I chose a batch of [1,5,10] for experiment, and the progress is observable.

Batch Size	Iteration Times	Validation Error
1	10000	0.052200
	20000	0.048800
	30000	0.043200
	40000	0.043200
	Test error with final model = 0.019000	
5	10000	0.029800
	20000	0.027600
	30000	0.027200
	40000	0.027000
	Test error with final model = 0.015000	
10	10000	0.028200
	20000	0.024400
	30000	0.025600
	40000	0.025400
	Test error with final model = 0.013000	

Table 7: Validation error varying with batch size

12 Convolutional Layer

A convolutional layer is added before the full-connected layers. Then the structure of the network has changed to:

Convolutional network structure
Input layer: reshape X from [5000,256] to [16,16,5000]
Convolutional layer: Use several 5×5 kernels convolute
with 16×16 features matrices $\rightarrow$ 12×12 outputs
Pooling layer: Mean pooling. Convolute 12×12 input matrices
with a 2×2 pooling matrix (all the values are $\frac{1}{4}$)
$\rightarrow$Delete the overlapping weights' output$\rightarrow$ 6×6 outputs
Full-connected layer: reshape 6×6 outputs to a vector
and use them as the input vector into a traditional neural network
One or two layers of hidden units
Output

With the help of $DeepLearnToolbox-master$, I ran the convolutional layer with $test\_CNN$. After 100 epochs, I got a test error of 0.023000.

13 Final Result

Here comes the final result. Modifications are made from the former methods presented in the sections above. With hundreds and thousands of trials, I found the relations of them and picked the most efficient ones, tuning the hyper-parameters. That is :

Modification Method	Hyper-parameter	Value
Network Structure	Layers	1
	Hidden Units	180
Weight Initialization	Initial Weights Value(w)	randn(nParams,1)/10
Learning Rate	Initial Stepsize	5e-3
Momentum Strength	beta	0.9
Early Stop	maxIter	50,000
Regularization	lambda	0.01
Dropout	Dropout Possibility(p)	0.8
Expanding	Sample size	5000*7(3*rotate +3*noise)
Batch Gradient Descent	Batch Size	10

Table 8: Optimal hyper-parameter choice

With those hyper-parameters and fun the script $fine\_tuning.m$, the final test error is stably around 0.8%.

14 Reflection

This project is not an easy one. Over ten hyper-parameters have to be adjusted and several modification has to be done to the whole model. The most important thing is to identify what those methods are for and what the relations are between them. I think the most significant modifications are network structure, weight initialization, expanding training data and expanding batch-size.

The difficulty is in adjusting one hyper-parameter while others are changing. For example, as layers and hidden units increase, the range of initial value of weights should be smaller. Dropout and regularization may perform not so good when the network is simple or when it does not face problem of overfitting.

Through these tasks I have used over ten methods to minimize error and their pros and cons. Hundreds of mistakes have been made. I really learnt a lot. Great gratitude to our teacher who designed this project and who would be seeing this paper.

15 Reference

[1] Bengio, Yoshua. Practical Recommendations for Gradient-Based Training of Deep Architectures. Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012:437-478.

[2] Lecun Y, Bottou L, Orr G B, et al. Efficient BackProp[M]// Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 1998:9-50.

[3] http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial

[4] Tobergte, David R., and S. Curtis. "Improving neural networks with dropout." (2015).