

Report to Project-3

Yuanbo Han 15300180032

December 10, 2017

Contents

1	Implements	2
1.1	Determine $nHidden$	2
1.2	Step Size and Momentum	2
1.2.1	Step Size	2
1.2.2	Momentum	4
1.3	Optimize Program	4
1.4	L2-Regularization	5
1.5	Softmax	5
1.6	Add Bias	7
1.7	Dropout	7
1.8	Fine-tune by Least Squares	8
1.9	Image Transformation	8
1.10	CNN	8
2	My Model	9
A	MATLAB Codes	9
A.1	<i>example_neuralNetwork.m</i>	9
A.2	<i>plotnHidden.m</i>	11
A.3	<i>MLPclassificationLoss_mat.m</i>	12
A.4	<i>MLPclassificationPredict.m</i>	14
A.5	<i>MLP_L2.m</i>	15
A.6	<i>plotLambda.m</i>	18
A.7	<i>MLP_softmax.m</i>	19
A.8	<i>addBias6.m</i>	21
A.9	<i>MLP_addbias.m</i>	23
A.10	<i>MLP_addbias_predict.m</i>	25
A.11	<i>MLP_dropout.m</i>	26
A.12	<i>fine_tune8.m</i>	28
A.13	<i>MLP_finetune.m</i>	30
A.14	<i>creat_train.m</i>	32
A.15	<i>convol10.m</i>	34
A.16	<i>convolution_f.m</i>	36

A.17 <i>CNN_update.m</i>	36
A.18 <i>CNN_predict.m</i>	39
A.19 <i>MLP_softmax_L2.m</i>	40

1 Implements

1.1 Determine *nHidden*

When it comes to a neural network model, we always have to decide the network structure, i.e. the number of layers and of hidden units in each layer. Unfortunately, there are currently no theoretical or rigorous methods for determining them. But some empirical summaries and formulae do work well in practice. One experience is that, in MLP, we usually set only one hidden layer, as long as the number of features is not fairly large. Since here we have 256 features for each observed data, the network would contain no more than 2 hidden layers. And I will check it soon that only 1 hidden layer is enough.

Another useful formula is $\frac{2}{3} \times (\text{features} + \text{labels})$. This is often a good number of hidden units with single hidden layer for MLP. In this question, $\frac{2}{3} \times (256 + 10) \approx 177$.

During experiment, I adopt one hidden layer. Surprisingly, I find it workable to plot the average validation error of 10 runs, against the number of hidden units, for the network is not so complicated. I write the script *plotnHidden.m* to plot Figure 1. Note that the function *MLPclassificationLoss_mat* applied in the script is the optimized function in Section 1.3, which does the same thing as but is much faster than the original function *MLPclassificationLoss*.

From Figure 1, we shall take *nHidden* = [120]. The validation error is now around 24%.

```
Training iteration = 9500, validation error = 0.245400
Test error with final model = 0.227000
Elapsed time is 7.681454 seconds.
```

In the following sections, we **always** set *nHidden* = [120], because this is part of the basic parameters of a neural network model.

1.2 Step Size and Momentum

1.2.1 Step Size

A big step size at early stage could effectively accelerate the learning process, whereas a small one usually results in an accurate convergence. So we make the step size decreases as the training processes.

$$\text{stepSize} = \text{max_stepSize} - \frac{\text{iter}}{\text{maxIter}}(\text{max_stepSize} - \text{min_stepSize})$$

Implement in codes:

...

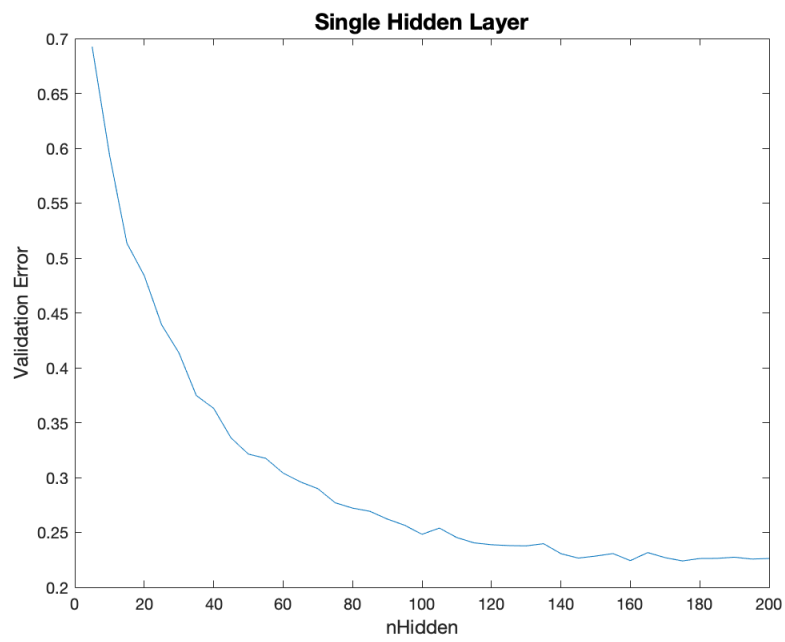


Figure 1: Validation error against n_{Hidden}

```

1  max_stepSize = 1e-3 * 5;
2  min_stepSize = 1e-4 * 5;
...
1  stepSize = max_stepSize - iter/maxIter * (max_stepSize - min_stepSize);
2  w = w - stepSize * g;
...

```

1.2.2 Momentum

Add a momentum via modifying a small part of codes when updating *Weights*:

```

...
1  stepSize = 1e-3 * 3;
2  momentumStrength = 0.9;
3  delta = 0;
...
1      [~,g] = funObj(w,i);
2      delta = stepSize * g - momentumStrength * delta;
3      w = w - delta;
...

```

By adding a momentum, we could set the initial *stepSize* bigger, which helps accelerating the learning process but not result in vibration meanwhile. To its credit, there is high chance that the potential vibration around target is avoided.

```

Training iteration = 9500, validation error = 0.269800
Test error with final model = 0.259000
Elapsed time is 7.603368 seconds.

```

As we see, with enough iteration, adding a momentum does not improve the final performance of our model. However, if we check the intermediate steps, we shall find that it usually takes only around 4000 iterations to reach 30% validation error, whereas the original model (used in Section 1.1) need to learn 6000 or so times.

In conclusion, adding a momentum increases the speed of converging, but usually cannot improve accuracy.

1.3 Optimize Program

The new function *MLPclassificationLoss_mat* written by me has approximately tripled the programming efficiency. Every thing has been tried best to operate by matrix. Not a detail is left for whether a single hidden layer or multiple hidden layers. Even when *nargout* == 1, or rather, when we only need to compute the squared-error, the program is accelerated. On the other hand, I

also made small fixes on function *MLPclassificationPredict* by defining variables before assignment.

Before optimization:

```
Training iteration = 9500, validation error = 0.262400
Test error with final model = 0.236000
Elapsed time is 16.313542 seconds.
```

After optimization:

```
Training iteration = 9500, validation error = 0.240600
Test error with final model = 0.254000
Elapsed time is 6.403858 seconds.
```

1.4 L2-Regularization

Regularization, especially L2-regularization, is a common proposal to avoid over-fitting in learning models. To be specific, $E = E_0 + \frac{\lambda}{2}||w||_2^2$, where E_0 is the original error. Don't forget there is a trick that the bias need not be regularized. I write function *MLP_L2* to add L2-regularization of *Weights* to the loss function.

Having completed the preparation, we are ready to select λ . *plotLambda.m* written by me plots how the average validation error of 10 runs changes as λ increases from 0.001 to 0.512 (See Figure 2).

Obviously, $\lambda \approx 0.03$ performs best. Let's set $\lambda = 0.03$ and iterate 100000 times. The validation error is about 5%. L2-regularization largely improves the model!

```
Training iteration = 95000, validation error = 0.056600
Test error with final model = 0.049000
Elapsed time is 42.722980 seconds.
```

1.5 Softmax

The softmax function is defined as

$$p(y_i) = \frac{\exp(z_i)}{\sum_{j=1}^J \exp(z_j)}$$

Use the cross entropy as the loss function instead of squared-error:

$$L = -\log p(y_t)$$

where t is the true label.

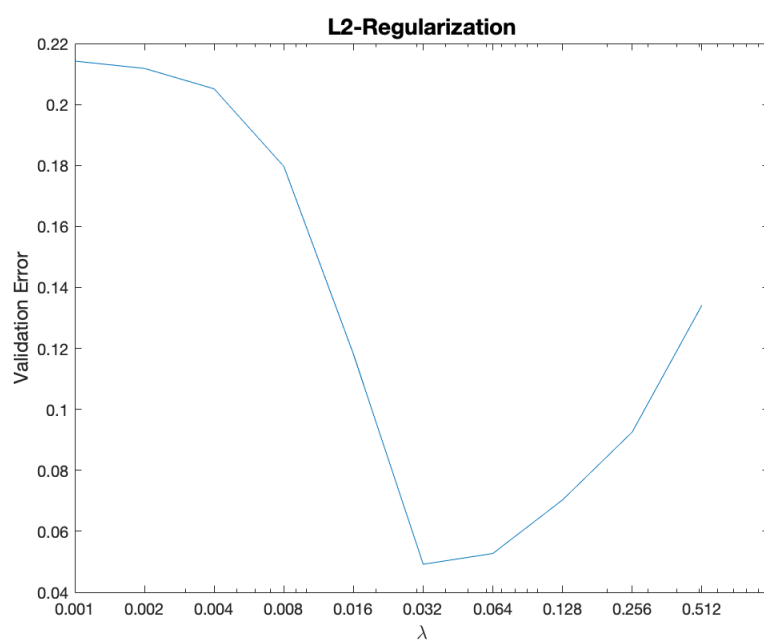


Figure 2: Validation error against λ

Now consider $\frac{\partial L}{\partial z_i}$. When $i = t$,

$$\begin{aligned}\frac{\partial L}{\partial z_t} &= \frac{\partial L}{\partial p(y_t)} \frac{\partial p(y_t)}{\partial z_t} \\ &= -\frac{1}{p(y_t)} (p(y_t) - p(y_t)^2) \\ &= p(y_t) - 1\end{aligned}$$

when $i \neq t$,

$$\begin{aligned}\frac{\partial L}{\partial z_i} &= \frac{\partial L}{\partial p(y_t)} \frac{\partial p(y_t)}{\partial z_i} \\ &= -\frac{1}{p(y_t)} (-p(y_t)p(y_i)) \\ &= p(y_i)\end{aligned}$$

Based on these, I write function *MLP_softmax* to compute the gradients of *Weights*. The validation error with 100000 iterations is about 20%.

1.6 Add Bias

As we can imagine, the value (before *tanh*) of a hidden unit might not be completely a linear form of its inputs. In most cases, there do exist a bias, so we shall add them up. I write function *MLP_addbias* to make one of the hidden units in each layer a constant, so that each layer has a bias. Note that with as we changed the network structure, the predicting function should also be modified. Use *MLP_addbias_predict* instead of *MLPclassificationPredict*, and the whole script is modified as *addBias6.m*.

Running it once, the result is below:

```
Training iteration = 9500, validation error = 0.246000
Test error with final model = 0.248000
Elapsed time is 6.076616 seconds.
```

1.7 Dropout

The dropout method is widely applied to complicated networks in case of overfitting. To carry it out, we multiply each hidden units an independent Bernoulli(1 - *p*)-distributed random variable, where *p* is exactly the probability of dropping out. The implementing function is *MLP_dropout*.

Setting *p* = 0.5, the results are below:

```
Training iteration = 9500, validation error = 0.233800
Test error with final model = 0.224000
Elapsed time is 6.450364 seconds.
```

As a result, the model has not been improved evidently. This is probably because our network is such an elementary topology that adopting dropout is needless.

1.8 Fine-tune by Least Squares

The error we are using is actually the *residualsumofsquares* (*RSS*) defined by $\|\hat{y} - y\|_2^2$. In Project-1, we have already discussed that for $y = XW$, when *RSS* reaches its minimal,

$$W = (X^T X)^\dagger X^T y \quad (*)$$

where $(X^T X)^\dagger$ is the Moore-Penrose pseudoinverse of $X^T X$. Therefore, for *outputWeights*, we could alternatively compute them by (*).

My function *MLP_finetune* first overwrite *outputWeights* by (*), and then computes the gradients as *MLPclassificationLoss_mat* does. It also returns the new *w* besides *f* and *g*.

Iterating 10000 times, the results turn out to be:

```
Training iteration = 9500, validation error = 0.841800
Test error with final model = 0.873000
Elapsed time is 23.957619 seconds.
```

An awfully poor performance! Why? Because what we are adopting is the best *outputWeights* for only one sample from training set. Such being the case, We are unlikely to assume they work well for any other observation. From my perspective, this method to determine *outputWeights* should only be implemented for the first training. In other words, Initialize *outputWeights* by (*). Then do stochastic gradient descend as usual, and **never** compute (*) **again**.

1.9 Image Transformation

One can write numbers at different positions, in various sizes, and slantly or straightly. Considering these situation, we could artificially create more training examples, by applying small transformations (translations, rotations, resizing, etc.) to the original images.

Script *creattrain.m* enlarges the training set 9 times. Each original image is respectively translated 1 pixel right, left, down, up, and rotated 5 degree clockwise, anti-clockwise, and resized 10% larger, smaller. Built-in functions *imtranslate*, *imrotate* and *imresize* are used.

The results for 10000 iterations are:

```
Training iteration = 9500, validation error = 0.277000
Test error with final model = 0.261000
Elapsed time is 4.983946 seconds.
```

1.10 CNN

In this section, we will add a 2D convolutional layer between input and hidden layer. *convol10.m* is my main script carrying this out. Three of my functions are involved in total.

convolution_f computes the full-connected-layer values.

CNN_update computes the gradients of weights, kernels and bias, and then updates these parameters by stepSize using gradient descent method.

CNN_predict classifies samples by the current CNN model.

Since I have written adequate comments step by step in the script and functions (See A.15), there is no need to analyze the detail again.

Running *convol10.m*, the results are:

```
>> convol10
Training iteration = 0, validation error = 0.922600
Elapsed time is 38.988072 seconds.
Training iteration = 10000, validation error = 0.231400
Elapsed time is 292.766576 seconds.
Training iteration = 20000, validation error = 0.205400
Elapsed time is 291.313784 seconds.
Training iteration = 30000, validation error = 0.206000
Elapsed time is 299.403606 seconds.
Training iteration = 40000, validation error = 0.182800
Elapsed time is 292.518354 seconds.
Training iteration = 50000, validation error = 0.191600
Elapsed time is 290.924409 seconds.
Training iteration = 60000, validation error = 0.189600
Elapsed time is 299.459586 seconds.
Training iteration = 70000, validation error = 0.180000
Elapsed time is 324.002883 seconds.
Training iteration = 80000, validation error = 0.178600
Elapsed time is 316.184651 seconds.
Training iteration = 90000, validation error = 0.170400
Elapsed time is 315.688232 seconds.
Training iteration = 100000, validation error = 0.169400
Test error with final model = 0.163000
Elapsed time is 283.742722 seconds.
```

The model has been improved a lot. However, the learning processes quite slow.

2 My Model

My final model uses single hidden layer with 120 units, and applies methods in Section 1.4 with $\lambda = 0.03$, Section 1.6 and Section 1.5. The implement function is *MLP_softmax_L2*.

The final test error is about 4%.

A MATLAB Codes

A.1 *example_neuralNetwork.m*

```

1 load digits.mat
2 [n,d] = size(X);
3 nLabels = max(y);
4 yExpanded = linearInd2Binary(y,nLabels);
5 t = size(Xvalid,1);
6 t2 = size(Xtest,1);
7
8 % Standardize columns and add bias
9 [X,mu,sigma] = standardizeCols(X);
10 X = [ones(n,1) X];
11 d = d + 1;
12
13 % Make sure to apply the same transformation to the validation/test data
14 Xvalid = standardizeCols(Xvalid,mu,sigma);
15 Xvalid = [ones(t,1) Xvalid];
16 Xtest = standardizeCols(Xtest,mu,sigma);
17 Xtest = [ones(t2,1) Xtest];
18
19 % Choose network structure
20 nHidden = [120];
21
22 % Count number of parameters and initialize weights 'w'
23 nParams = d*nHidden(1);
24 for h = 2:length(nHidden)
25     nParams = nParams+nHidden(h-1)*nHidden(h);
26 end
27 nParams = nParams+nHidden(end)*nLabels;
28 w = randn(nParams,1);
29
30 % Train with stochastic gradient
31 maxIter = 10000;
32 stepSize = 1e-3; %* 3;
33 %momentumStrength = 0.9;
34 %delta = 0;
35 %lambda = 0.03;
36 %p = 0.5;
37 funObj = @(w,i)MLP_softmax(w, X(i,:), yExpanded(i,:), ...
38     nHidden, nLabels);
39
40 tic
41 for iter = 1:maxIter
42     if mod(iter-1,round(maxIter/20)) == 0
43         yhat = MLPclassificationPredict(w,Xvalid,nHidden,nLabels);
44         fprintf('Training iteration = %d, validation error = %f\n', ...
45             iter-1, sum(yhat~=yvalid)/t);
46     end

```

```

47         i = ceil(rand*n);
48         [~,g] = funObj(w,i);
49         %delta = stepSize * g - momentumStrength * delta;
50         %w = w - delta;
51         w = w - stepSize * g;
52     end
53
54     % Evaluate test error
55     yhat = MLPclassificationPredict(w,Xtest,nHidden,nLabels);
56     fprintf('Test error with final model = %f\n',sum(yhat~=ytest)/t2);
57     toc

```

A.2 *plotnHidden.m*

```

1  % Edited by Yuanbo Han, Dec. 7, 2017.
2
3  load digits.mat;
4  [n,d] = size(X);
5  nLabels = max(y);
6  yExpanded = linearInd2Binary(y,nLabels);
7  t = size(Xvalid,1);
8
9  % Standardize columns and add bias
10 [X,mu,sigma] = standardizeCols(X);
11 X = [ones(n,1) X];
12 d = d + 1;
13
14 % Apply the same transformation to the validation data
15 Xvalid = standardizeCols(Xvalid,mu,sigma);
16 Xvalid = [ones(t,1) Xvalid];
17
18 maxIter = 10000;
19 stepSize = 1e-3;
20 validError = zeros(1,40);
21 % Choose network structure
22 for nHidden = 5:5:200
23     tic
24     % Count number of parameters
25     nParams = d*nHidden(1);
26     for h = 2:length(nHidden)
27         nParams = nParams+nHidden(h-1)*nHidden(h);
28     end
29     nParams = nParams+nHidden(end)*nLabels;
30

```

```

31     for k = 1:10
32         % Initialize weights 'w'
33         w = randn(nParams,1);
34
35         % Train with stochastic gradient
36         funObj = @(w,i)MLPclassificationLoss_mat(w, X(i,:), ...
37             yExpanded(i,:), nHidden, nLabels);
38         for iter = 1:maxIter
39             i = ceil(rand*n);
40             [~,g] = funObj(w,i);
41             w = w - stepSize*g;
42         end
43
44         % Evaluate validation error
45         yhat = MLPclassificationPredict(w,Xvalid,nHidden,nLabels);
46         validError(nHidden/5) = validError(nHidden/5) + ...
47             1/10 * sum(yhat~=yvalid)/t;
48     end
49     fprintf('nHidden = %d\n', nHidden);
50     fprintf('Average validation error = %f\n', validError(nHidden/5));
51     toc
52 end
53
54 figure;
55 plot(5:5:200, validError);
56 xlabel('nHidden', 'FontSize', 12);
57 ylabel('Validation Error', 'FontSize', 12);
58 title('Single Hidden Layer', 'FontSize', 14);

```

A.3 MLPclassificationLoss_mat.m

```

1 function [f,g] = MLPclassificationLoss_mat(w,X,y,nHidden,nLabels)
2 % MLPCLASSIFICATIONLOSS_MAT does the same thing as MLPclassificationLoss,
3 % but computes as much by matrix as possible, which is very fast.
4 %
5 % Yuanbo Han, Dec. 5, 2017.
6
7 [nInstances, nVars] = size(X);
8 nHiddenLayers = length(nHidden);
9
10 % Form Weights
11 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
12 offset = nVars * nHidden(1);
13 hiddenWeights = cell(1, nHiddenLayers-1);
14 for h = 2:nHiddenLayers

```

```

15 hiddenWeights{h-1} = reshape(...
16 w(offset+1:offset+nHidden(h-1)*nHidden(h)),...
17 nHidden(h-1), nHidden(h));
18 offset = offset + nHidden(h-1) * nHidden(h);
19 end
20 outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
21 outputWeights = reshape(outputWeights, nHidden(end), nLabels);
22
23 ip = cell(1, nHiddenLayers);
24 fp = cell(1, nHiddenLayers);
25 if nargout > 1
26     % Form Gradient
27     gInput = zeros(size(inputWeights));
28     gHidden = cell(1, nHiddenLayers-1);
29     for h = 2:nHiddenLayers
30         gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
31     end
32     gOutput = zeros(size(outputWeights));
33
34     f = 0;
35
36     % Compute Output
37     for i = 1:nInstances
38         ip{1} = X(i,:) * inputWeights;
39         fp{1} = tanh(ip{1});
40         for h = 2:length(nHidden)
41             ip{h} = fp{h-1} * hiddenWeights{h-1};
42             fp{h} = tanh(ip{h});
43         end
44         yhat = fp{end} * outputWeights;
45
46         relativeErr = yhat - y(i,:);
47         f = f + sum(relativeErr.^2);
48
49         err = 2 * relativeErr;
50
51         % Output Weights
52         gOutput = gOutput + fp{end}' * err;
53
54         if nHiddenLayers > 1
55             % Last Layer of Hidden Weights
56             backprop = (err' * sech(ip{end}).^2) .* outputWeights';
57             backprop = sum(backprop,1);
58             gHidden{end} = gHidden{end} + fp{end-1}' * backprop;
59
60             % Other Hidden Layers

```

```

61 for h = nHiddenLayers-2:-1:1
62     backprop = (backprop * hiddenWeights{h+1}') .* ...
63     sech(ip{h+1}).^2;
64     gHidden{h} = gHidden{h} + fp{h}' * backprop;
65 end
66
67 % Input Weights
68 backprop = (backprop * hiddenWeights{1}') .* sech(ip{1}).^2;
69 gInput = gInput + X(i,:) * backprop;
70
71 else % nHiddenLayers == 1
72     % Input Weights
73     gInput = gInput + X(i,:) * ...
74     ( sech(ip{end}).^2 .* (outputWeights * err')' );
75 end
76 end
77
78 % Put Gradient into vector
79 g = zeros(size(w));
80 g(1:nVars*nHidden(1)) = gInput(:);
81 offset = nVars*nHidden(1);
82 for h = 2:nHiddenLayers
83     g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1};
84     offset = offset+nHidden(h-1)*nHidden(h);
85 end
86 g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
87
88
89 else % nargout <= 1
90     ip{1} = X * inputWeights;
91     fp{1} = tanh(ip{1});
92     for h = 2:nHiddenLayers
93         ip{h} = fp{h-1} * hiddenWeights{h-1};
94         fp{h} = tanh(ip{h});
95     end
96     yhat = fp{end} * outputWeights;
97
98     relativeErr = yhat - y;
99     f = sum(sum(relativeErr.^2));
100 end
101
102 end

```

A.4 MLPclassificationPredict.m

```

1 function [y] = MLPclassificationPredict(w,X,nHidden,nLabels)
2 % Modified by Yuanbo Han, Dec. 7, 2017.
3
4 [nInstances,nVars] = size(X);
5 nHiddenLayers = length(nHidden);
6
7 % Form Weights
8 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
9 offset = nVars*nHidden(1);
10 for h = 2:nHiddenLayers
11 hiddenWeights{h-1}=reshape(w(offset+1:...
12 offset+nHidden(h-1)*nHidden(h)), nHidden(h-1), nHidden(h));
13 offset = offset+nHidden(h-1)*nHidden(h);
14 end
15 outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
16 outputWeights = reshape(outputWeights,nHidden(end),nLabels);
17
18 ip = cell(1, nHiddenLayers);
19 fp = cell(1, nHiddenLayers);
20
21 y = zeros(nInstances, nLabels);
22 % Compute Output
23 for i = 1:nInstances
24 ip{1} = X(i,:)*inputWeights;
25 fp{1} = tanh(ip{1});
26 for h = 2:length(nHidden)
27 ip{h} = fp{h-1}*hiddenWeights{h-1};
28 fp{h} = tanh(ip{h});
29 end
30 y(i,:) = fp{end}*outputWeights;
31 end
32 [~, y] = max(y,[],2);
33 %y = binary2LinearInd(y);
34 end

```

A.5 MLP_L2.m

```

1 function [f,g] = MLP_L2(w,X,y,nHidden,nLabels,lambda)
2 % MLP_L2 adds L2-regularization of Weights to the loss function.
3 %
4 % Yuanbo Han, Dec. 5, 2017.
5
6 [nInstances, nVars] = size(X);
7 nHiddenLayers = length(nHidden);
8

```

```

9  % Form Weights
10 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
11 offset = nVars * nHidden(1);
12 hiddenWeights = cell(1, nHiddenLayers-1);
13 for h = 2:nHiddenLayers
14 hiddenWeights{h-1} = reshape(...
15 w(offset+1:offset+nHidden(h-1)*nHidden(h)),...
16 nHidden(h-1), nHidden(h));
17 offset = offset + nHidden(h-1) * nHidden(h);
18 end
19 outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
20 outputWeights = reshape(outputWeights, nHidden(end), nLabels);
21
22 ip = cell(1, nHiddenLayers);
23 fp = cell(1, nHiddenLayers);
24 if nargout > 1
25 % Form Gradient
26 gInput = zeros(size(inputWeights));
27 gHidden = cell(1, nHiddenLayers-1);
28 for h = 2:nHiddenLayers
29 gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
30 end
31 gOutput = zeros(size(outputWeights));
32
33 f = 0;
34
35 % Compute Output
36 for i = 1:nInstances
37 ip{1} = X(i,:) * inputWeights;
38 fp{1} = tanh(ip{1});
39 for h = 2:length(nHidden)
40 ip{h} = fp{h-1} * hiddenWeights{h-1};
41 fp{h} = tanh(ip{h});
42 end
43 yhat = fp{end} * outputWeights;
44
45 relativeErr = yhat - y(i,:);
46 f = f + sum(relativeErr.^2);
47
48 err = 2 * relativeErr;
49
50 % Output Weights
51 gOutput = gOutput + fp{end}' * err + lambda * outputWeights;
52
53 if nHiddenLayers > 1
54 % Last Layer of Hidden Weights

```



```

55 backprop = err' * sech(ip{end}).^2 .* outputWeights';
56 gHidden{end} = gHidden{end} + fp{end-1}' * sum(backprop,1) ...
57 + lambda * hiddenWeights{end};
58
59 backprop = sum(backprop,1);
60 % Other Hidden Layers
61 for h = length(nHidden)-2:-1:1
62 backprop = (backprop * hiddenWeights{h+1}') .* ...
63 sech(ip{h+1}).^2;
64 gHidden{h} = gHidden{h} + fp{h}' * sum(backprop,1) + ...
65 lambda * hiddenWeights{h};
66 end
67
68 % Input Weights
69 backprop = (backprop * hiddenWeights{1}') .* sech(ip{1}).^2;
70 gInput = gInput + X(i,:) * backprop + lambda * inputWeights;
71 % The bias need not be included in regularization.
72 gInput(1,:) = gInput(1,:) - lambda * inputWeights(1,:);
73
74 else % nHiddenLayers == 1
75 % Input Weights
76 gInput = gInput + X(i,:) * ...
77 ( sech(ip{end}).^2 .* (outputWeights * err')' ) + ...
78 lambda * inputWeights;
79 % The bias need not be included in regularization.
80 gInput(1,:) = gInput(1,:) - lambda * inputWeights(1,:);
81 end
82 end
83
84 % Put Gradient into vector
85 g = zeros(size(w));
86 g(1:nVars*nHidden(1)) = gInput(:);
87 offset = nVars*nHidden(1);
88 for h = 2:nHiddenLayers
89 g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1};
90 offset = offset+nHidden(h-1)*nHidden(h);
91 end
92 g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
93
94
95 else % nargout <= 1
96 ip{1} = X * inputWeights;
97 fp{1} = tanh(ip{1});
98 for h = 2:nHiddenLayers
99 ip{h} = fp{h-1} * hiddenWeights{h-1};
100 fp{h} = tanh(ip{h});

```

```

101 end
102 yhat = fp{end} * outputWeights;
103
104 relativeErr = yhat - y;
105 f = sum(sum(relativeErr.^2));
106 end

```

A.6 *plotLambda.m*

```

1  % Edited by Yuanbo Han, Dec. 8, 2017.
2
3  load digits.mat
4  [n,d] = size(X);
5  nLabels = max(y);
6  yExpanded = linearInd2Binary(y,nLabels);
7  t = size(Xvalid,1);
8
9  % Standardize columns and add bias
10 [X,mu,sigma] = standardizeCols(X);
11 X = [ones(n,1) X];
12 d = d + 1;
13
14 % Apply the same transformation to the validation data
15 Xvalid = standardizeCols(Xvalid,mu,sigma);
16 Xvalid = [ones(t,1) Xvalid];
17
18 % Choose network structure
19 nHidden = [120];
20
21 % Count number of parameters
22 nParams = d*nHidden(1);
23 for h = 2:length(nHidden)
24     nParams = nParams+nHidden(h-1)*nHidden(h);
25 end
26 nParams = nParams+nHidden(end)*nLabels;
27
28 maxIter = 100000;
29 stepSize = 1e-3;
30 funObj = @(w,i,lambda)MLP_L2(w, X(i,:), yExpanded(i,:), nHidden, ...
31     nLabels, lambda);
32
33 lambda = zeros(1,10);
34 lambda(1) = 0.001;
35 for i = 2:10
36     lambda(i) = lambda(i-1) * 2;

```

```

37 end
38
39 validError = zeros(1,length(lambda));
40 for l = 1:length(lambda)
41 tic
42 for k = 1:10
43 % Initialize weights 'w'
44 w = randn(nParams,1);
45
46 % Train with stochastic gradient
47 for iter = 1:maxIter
48 i = ceil(rand*n);
49 [~,g] = funObj(w,i,lambda(l));
50 w = w - stepSize*g;
51 end
52
53 % Evaluate validation error
54 yhat = MLPclassificationPredict(w, Xvalid, nHidden, nLabels);
55 validError(l) = validError(l) + 1/10 * sum(yhat~=yvalid)/t;
56 end
57 fprintf('lambda = %.3f\n', lambda(l));
58 fprintf('Average validation error = %f\n', validError(l));
59 toc
60 end
61
62 figure;
63 semilogx(lambda, validError);
64 set(gca, 'XTick', lambda);
65 xlabel('\lambda', 'FontSize', 12);
66 ylabel('Validation Error', 'FontSize', 12);
67 title('L2-Regularization', 'FontSize', 14);

```

A.7 MLP_softmax.m

```

1 function [f,g] = MLP_softmax(w,X,y,nHidden,nLabels)
2 % MLP_SOFTMAX use a softmax (multinomial logistic) layer at the end of the
3 % network, and replace squared error with the negative log-likelihood of
4 % the true label under this loss.
5 %
6 % Yuanbo Han, Dec. 8, 2017.
7
8 [nInstances, nVars] = size(X);
9 nHiddenLayers = length(nHidden);
10
11 % Form Weights

```

```

12 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
13 offset = nVars * nHidden(1);
14 hiddenWeights = cell(1, nHiddenLayers-1);
15 for h = 2:nHiddenLayers
16 hiddenWeights{h-1} = reshape(...
17 w(offset+1:offset+nHidden(h-1)*nHidden(h)),...
18 nHidden(h-1), nHidden(h));
19 offset = offset + nHidden(h-1) * nHidden(h);
20 end
21 outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
22 outputWeights = reshape(outputWeights, nHidden(end), nLabels);
23
24 ip = cell(1, nHiddenLayers);
25 fp = cell(1, nHiddenLayers);
26 f = 0;
27 % Compute Output
28 for i = 1:nInstances
29 ip{1} = X(i,:) * inputWeights;
30 fp{1} = tanh(ip{1});
31 for h = 2:length(nHidden)
32 ip{h} = fp{h-1} * hiddenWeights{h-1};
33 fp{h} = tanh(ip{h});
34 end
35 yhat = fp{end} * outputWeights;
36 yhat = exp(yhat) / sum(exp(yhat));
37 yhat_true = (y(i,')==1) * yhat';
38
39 err = -log( yhat_true );
40 f = f + err;
41
42 if nargout > 1
43 % Form Gradient
44 gInput = zeros(size(inputWeights));
45 gHidden = cell(1, nHiddenLayers-1);
46 for h = 2:nHiddenLayers
47 gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
48 end
49 gOutput = zeros(size(outputWeights));
50
51 % Output Weights
52 gOutput = gOutput - fp{end}' * (1 - yhat_true) * (y(i,')==1);
53
54 % to be modified for nHiddenLayers > 1
55 if nHiddenLayers > 1
56 % Last Layer of Hidden Weights
57 backprop = err' * sech(ip{end}).^2 .* outputWeights';

```

```

58 gHidden{end} = gHidden{end} + fp{end-1}' * sum(backprop,1);
59
60 backprop = sum(backprop,1);
61 % Other Hidden Layers
62 for h = length(nHidden)-2:-1:1
63 backprop = (backprop * hiddenWeights{h+1}') .* ...
64 sech(ip{h+1}).^2;
65 gHidden{h} = gHidden{h} + fp{h}' * backprop;
66 end
67
68 % Input Weights
69 backprop = (backprop * hiddenWeights{1}') .* sech(ip{1}).^2;
70 gInput = gInput + X(i,:) * backprop;
71
72 else % nHiddenLayers == 1
73 % Input Weights
74 gInput = gInput - (1 - yhat_true) * X(i,:) * ...
75 ( sech(ip{end}).^2 .* outputWeights(:, y(i,:)=1) );
76 end
77
78 % Put Gradient into vector
79 g = zeros(size(w));
80 g(1:nVars*nHidden(1)) = gInput(:);
81 offset = nVars*nHidden(1);
82 for h = 2:nHiddenLayers
83 g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1};
84 offset = offset+nHidden(h-1)*nHidden(h);
85 end
86 g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
87
88 end
89 end
90 end

```

A.8 addBias6.m

```

1 % Edited by Yuanbo Han, Dec. 8, 2017.
2
3 load digits.mat
4 [n,d] = size(X);
5 nLabels = max(y);
6 yExpanded = linearInd2Binary(y,nLabels);
7 t = size(Xvalid,1);
8 t2 = size(Xtest,1);
9

```

```

10 % Standardize columns and add bias
11 [X,mu,sigma] = standardizeCols(X);
12 X = [ones(n,1) X];
13 d = d + 1;
14
15 % Apply the same transformation to the validation/test data
16 Xvalid = standardizeCols(Xvalid,mu,sigma);
17 Xvalid = [ones(t,1) Xvalid];
18 Xtest = standardizeCols(Xtest,mu,sigma);
19 Xtest = [ones(t2,1) Xtest];
20
21 % Choose network structure
22 nHidden = [120];
23
24 % Add a constant to each of the hidden layers
25 % Count number of parameters and initialize weights 'w'
26 nParams = d*nHidden(1);
27 for h = 2:length(nHidden)
28     nParams = nParams+(nHidden(h-1)+1)*nHidden(h);
29 end
30 nParams = nParams+(nHidden(end)+1)*nLabels;
31 w = randn(nParams,1);
32
33 % Train with stochastic gradient
34 maxIter = 10000;
35 stepSize = 1e-3; %* 3;
36 %momentumStrength = 0.9;
37 %delta = 0;
38 %lambda = 0.03;
39 funObj = @(w,i)MLP_addbias(w,X(i,:),yExpanded(i,:),nHidden,nLabels);
40
41 tic
42 for iter = 1:maxIter
43     if mod(iter-1,round(maxIter/20)) == 0
44         yhat = MLP_addbias_predict(w,Xvalid,nHidden,nLabels);
45         fprintf('Training iteration = %d, validation error = %f\n', ...
46             iter-1, sum(yhat~=yvalid)/t);
47     end
48
49     i = ceil(rand*n);
50     [~,g] = funObj(w,i);
51     % delta = stepSize * g - momentumStrength * delta;
52     % w = w - delta;
53     w = w - stepSize * g;
54 end
55

```

```

56 % Evaluate test error
57 yhat = MLP_addbias_predict(w,Xtest,nHidden,nLabels);
58 fprintf('Test error with final model = %f\n',sum(yhat~=ytest)/t2);
59 toc

```

A.9 MLP_addbias.m

```

1 function [f,g] = MLP_addbias(w,X,y,nHidden,nLabels)
2 % MLP_ADDBIAS makes one of the hidden units in each layer a constant, so
3 % that each layer has a bias.
4 %
5 % Yuanbo Han, Dec. 8, 2017.
6
7 [nInstances, nVars] = size(X);
8 nHiddenLayers = length(nHidden);
9
10 % Form Weights
11 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
12 offset = nVars * nHidden(1);
13 hiddenWeights = cell(1, nHiddenLayers-1);
14 for h = 2:nHiddenLayers
15 hiddenWeights{h-1} = reshape(...
16 w(offset+1:offset+(nHidden(h-1)+1)*nHidden(h)),...
17 nHidden(h-1)+1, nHidden(h));
18 offset = offset + (nHidden(h-1) + 1) * nHidden(h);
19 end
20 outputWeights = w(offset+1:offset+(nHidden(end)+1)*nLabels);
21 outputWeights = reshape(outputWeights, nHidden(end)+1, nLabels);
22
23 ip = cell(1, nHiddenLayers);
24 fp = cell(1, nHiddenLayers);
25 if nargout > 1
26 % Form Gradient
27 gInput = zeros(size(inputWeights));
28 gHidden = cell(1, nHiddenLayers-1);
29 for h = 2:nHiddenLayers
30 gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
31 end
32 gOutput = zeros(size(outputWeights));
33
34 f = 0;
35
36 % Compute Output
37 for i = 1:nInstances
38 ip{1} = [1, X(i,:)*inputWeights];

```

```

39 fp{1} = tanh(ip{1});
40 for h = 2:length(nHidden)
41 ip{h} = [1, fp{h-1}*hiddenWeights{h-1}];
42 fp{h} = tanh(ip{h});
43 end
44 yhat = fp{end} * outputWeights;
45
46 relativeErr = yhat - y(i,:);
47 f = f + sum(relativeErr.^2);
48
49 err = 2 * relativeErr;
50
51 % Output Weights
52 gOutput = gOutput + fp{end}' * err;
53
54 % to be modified
55 if nHiddenLayers > 1
56 % Last Layer of Hidden Weights
57 backprop = err' * sech(ip{end}).^2 .* outputWeights';
58 gHidden{end} = gHidden{end} + fp{end-1}' * sum(backprop,1);
59
60 backprop = sum(backprop,1);
61 % Other Hidden Layers
62 for h = length(nHidden)-2:-1:1
63 backprop = (backprop * hiddenWeights{h+1}') .* ...
64 sech(ip{h+1}).^2;
65 gHidden{h} = gHidden{h} + fp{h}' * backprop;
66 end
67
68 % Input Weights
69 backprop = (backprop * hiddenWeights{1}') .* sech(ip{1}).^2;
70 gInput = gInput + X(i,:) * backprop;
71
72 else % nHiddenLayers == 1
73 % Input Weights
74 temp = sech(ip{end}).^2 .* (outputWeights * err')';
75 gInput = gInput + X(i,:) * temp(2:end);
76 end
77 end
78
79 % Put Gradient into vector
80 g = zeros(size(w));
81 g(1:nVars*nHidden(1)) = gInput(:);
82 offset = nVars*nHidden(1);
83 for h = 2:nHiddenLayers
84 g(offset+1:offset+(nHidden(h-1)+1)*nHidden(h)) = gHidden{h-1};

```



```

85 offset = offset+(nHidden(h-1)+1)*nHidden(h);
86 end
87 g(offset+1:offset+(nHidden(end)+1)*nLabels) = gOutput(:);
88
89
90 else % nargout <= 1
91 ip{1} = [ones(ninstances,1), X*inputWeights];
92 fp{1} = tanh(ip{1});
93 for h = 2:nHiddenLayers
94 ip{h} = [ones(ninstances,1), fp{h-1}*hiddenWeights{h-1}];
95 fp{h} = tanh(ip{h});
96 end
97 yhat = fp{end} * outputWeights;
98
99 relativeErr = yhat - y;
100 f = sum(sum(relativeErr.^2));
101 end
102
103 end

```

A.10 *MLP_addbias_predict.m*

```

1 function [y] = MLP_addbias_predict(w,X,nHidden,nLabels)
2 % Please pair it up with function MLP_addbias.
3 %
4 % Yuanbo Han, Dec. 8, 2017.
5
6 [nInstances,nVars] = size(X);
7 nHiddenLayers = length(nHidden);
8
9 % Form Weights
10 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
11 offset = nVars*nHidden(1);
12 for h = 2:nHiddenLayers
13 hiddenWeights{h-1} = reshape( w(offset+1:...
14 offset + (nHidden(h-1)+1)*nHidden(h) ), ...
15 nHidden(h-1)+1, nHidden(h));
16 offset = offset+(nHidden(h-1)+1)*nHidden(h);
17 end
18 outputWeights = w(offset+1:offset+(nHidden(end)+1)*nLabels);
19 outputWeights = reshape(outputWeights,nHidden(end)+1,nLabels);
20
21 ip = cell(1, nHiddenLayers);
22 fp = cell(1, nHiddenLayers);
23

```

```

24 y = zeros(nInstances, nLabels);
25 % Compute Output
26 for i = 1:nInstances
27 ip{1} = [1, X(i,:) * inputWeights];
28 fp{1} = tanh(ip{1});
29 for h = 2:length(nHidden)
30 ip{h} = [1, fp{h-1} * hiddenWeights{h-1}];
31 fp{h} = tanh(ip{h});
32 end
33 y(i,:) = fp{end} * outputWeights;
34 end
35 [~,y] = max(y,[],2);
36
37 end

```

A.11 MLP_dropout.m

```

1 function [f,g] = MLP_dropout(w,X,y,nHidden,nLabels,p)
2 % MLP_DROPOUT dropped hidden units out with probability p during training.
3 %
4 % Yuanbo Han, Dec. 8, 2017.
5
6 [nInstances, nVars] = size(X);
7 nHiddenLayers = length(nHidden);
8
9 % Form Weights
10 inputWeights = reshape(w(1:nVars*nHidden(1)), nVars, nHidden(1));
11 offset = nVars * nHidden(1);
12 hiddenWeights = cell(1, nHiddenLayers-1);
13 for h = 2:nHiddenLayers
14 hiddenWeights{h-1} = reshape(...
15 w(offset+1:offset+nHidden(h-1)*nHidden(h)), ...
16 nHidden(h-1), nHidden(h));
17 offset = offset + nHidden(h-1) * nHidden(h);
18 end
19 outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
20 outputWeights = reshape(outputWeights, nHidden(end), nLabels);
21
22 ip = cell(1, nHiddenLayers);
23 fp = cell(1, nHiddenLayers);
24 if nargout > 1
25 % Form Gradient
26 gInput = zeros(size(inputWeights));
27 gHidden = cell(1, nHiddenLayers-1);
28 for h = 2:nHiddenLayers

```

```

29 gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
30 end
31 gOutput = zeros(size(outputWeights));
32
33 dropout = cell(1, nHiddenLayers);
34
35 f = 0;
36 % Compute Output
37 for i = 1:nInstances
38 dropout{1} = (rand(1, nHidden(1)) > p);
39 ip{1} = (X(i,:) * inputWeights) .* dropout{1};
40 fp{1} = tanh(ip{1});
41 for h = 2:length(nHidden)
42 dropout{h} = (rand(1, nHidden(h)) > p);
43 ip{h} = fp{h-1} * hiddenWeights{h-1} .* dropout{h};
44 fp{h} = tanh(ip{h});
45 end
46 yhat = fp{end} * outputWeights;
47
48 relativeErr = yhat - y(i,:);
49 f = f + sum(relativeErr.^2);
50
51 err = 2 * relativeErr;
52
53 % Output Weights
54 gOutput = gOutput + fp{end}' * err;
55
56 if nHiddenLayers > 1
57 % Last Layer of Hidden Weights
58 backprop = ( err' * (sech(ip{end}).^2 .* dropout{end}) ) .* ...
59 outputWeights';
60 backprop = sum(backprop,1);
61 gHidden{end} = gHidden{end} + fp{end-1}' * backprop;
62
63 % Other Hidden Layers
64 for h = length(nHidden)-2:-1:1
65 backprop = (backprop * hiddenWeights{h+1}') .* ...
66 sech(ip{h+1}).^2 .* dropout{h+1};
67 gHidden{h} = gHidden{h} + fp{h}' * backprop;
68 end
69
70 % Input Weights
71 backprop = (backprop * hiddenWeights{1}') .* ...
72 sech(ip{1}).^2 .* dropout{1};
73 gInput = gInput + X(i,:) * backprop;
74

```

```

75 else % nHiddenLayers == 1
76 % Input Weights
77 gInput = gInput + X(i,:) ' * ...
78 ( sech(ip{end}).^2 .* dropout{end} .* ...
79 (outputWeights * err') ' );
80 end
81 end
82
83 % Put Gradient into vector
84 g = zeros(size(w));
85 g(1:nVars*nHidden(1)) = gInput(:);
86 offset = nVars*nHidden(1);
87 for h = 2:nHiddenLayers
88 g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1};
89 offset = offset+nHidden(h-1)*nHidden(h);
90 end
91 g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
92
93
94 else % nargout <= 1
95 ip{1} = X * inputWeights;
96 fp{1} = tanh(ip{1});
97 for h = 2:nHiddenLayers
98 ip{h} = fp{h-1} * hiddenWeights{h-1};
99 fp{h} = tanh(ip{h});
100 end
101 yhat = fp{end} * outputWeights;
102
103 relativeErr = yhat - y;
104 f = sum(sum(relativeErr.^2));
105 end
106
107 end

```

A.12 *fine_tune8.m*

```

1 % Edited by Yuanbo Han, Dec. 9, 2017.
2
3 load digits.mat
4 [n,d] = size(X);
5 nLabels = max(y);
6 yExpanded = linearInd2Binary(y,nLabels);
7 t = size(Xvalid,1);
8 t2 = size(Xtest,1);
9

```

```

10 % Standardize columns and add bias
11 [X,mu,sigma] = standardizeCols(X);
12 X = [ones(n,1) X];
13 d = d + 1;
14
15 % Apply the same transformation to the validation/test data
16 Xvalid = standardizeCols(Xvalid,mu,sigma);
17 Xvalid = [ones(t,1) Xvalid];
18 Xtest = standardizeCols(Xtest,mu,sigma);
19 Xtest = [ones(t2,1) Xtest];
20
21 % Choose network structure
22 nHidden = [120];
23
24 % Count number of parameters and initialize weights 'w'
25 nParams = d*nHidden(1);
26 for h = 2:length(nHidden)
27     nParams = nParams+nHidden(h-1)*nHidden(h);
28 end
29 nParams = nParams+nHidden(end)*nLabels;
30 w = randn(nParams,1);
31
32 % Train with stochastic gradient
33 maxIter = 10000;
34 stepSize = 1e-3; %* 3;
35 %momentumStrength = 0.9;
36 %delta = 0;
37 %lambda = 0.03;
38 funObj = @(w,i)MLP_finetune(w,X(i,:),yExpanded(i,:),nHidden,nLabels);
39
40 tic
41 for iter = 1:maxIter
42     if mod(iter-1,round(maxIter/20)) == 0
43         yhat = MLPclassificationPredict(w,Xvalid,nHidden,nLabels);
44         fprintf('Training iteration = %d, validation error = %f\n', ...
45             iter-1, sum(yhat~=yvalid)/t);
46     end
47
48     i = ceil(rand*n);
49     [f,g,w] = funObj(w,i);
50     %reshape(w(nParams-nHidden(end)*nLabels+1:nParams),nHidden(end),nLabels)
51     % delta = stepSize * g - momentumStrength * delta;
52     % w = w - delta;
53     w = w - stepSize * g;
54 end
55

```

```

56 % Evaluate test error
57 yhat = MLPclassificationPredict(w,Xtest,nHidden,nLabels);
58 fprintf('Test error with final model = %f\n',sum(yhat~=ytest)/t2);
59 toc

```

A.13 MLP_finetune.m

```

1 function [f,g,w] = MLP_finetune(w,X,y,nHidden,nLabels)
2 % MLP_FINETUNE first overwrites outputWeights by least square method, and
3 % then computes the error and gradients.
4 %
5 % Yuanbo Han, Dec. 9, 2017.
6
7 [nInstances, nVars] = size(X);
8 nHiddenLayers = length(nHidden);
9
10 % Form Weights
11 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
12 offset = nVars * nHidden(1);
13 hiddenWeights = cell(1, nHiddenLayers-1);
14 for h = 2:nHiddenLayers
15 hiddenWeights{h-1} = reshape(...
16 w(offset+1:offset+nHidden(h-1)*nHidden(h)),...
17 nHidden(h-1), nHidden(h));
18 offset = offset + nHidden(h-1) * nHidden(h);
19 end
20
21 ip = cell(1, nHiddenLayers);
22 fp = cell(1, nHiddenLayers);
23
24 f = 0;
25 % Compute Output
26 for i = 1:nInstances
27 ip{1} = X(i,:) * inputWeights;
28 fp{1} = tanh(ip{1});
29 for h = 2:length(nHidden)
30 ip{h} = fp{h-1} * hiddenWeights{h-1};
31 fp{h} = tanh(ip{h});
32 end
33
34 % Compute outputWeights
35 outputWeights = pinv(fp{end}' * fp{end}) * fp{end}' * y(i,:);
36 w(offset+1:offset+nHidden(end)*nLabels) = outputWeights(:);
37
38 yhat = fp{end} * outputWeights;

```

```

39
40 relativeErr = yhat - y(i,:);
41 f = f + sum(relativeErr.^2);
42
43 err = 2 * relativeErr;
44
45 % Form Gradient
46 gInput = zeros(size(inputWeights));
47 gHidden = cell(1, nHiddenLayers-1);
48 for h = 2:nHiddenLayers
49 gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
50 end
51 gOutput = zeros(size(outputWeights));
52
53 % Output Weights
54 gOutput = gOutput + fp{end}' * err;
55
56 if nHiddenLayers > 1
57 % Last Layer of Hidden Weights
58 backprop = (err' * sech(ip{end}).^2) .* outputWeights';
59 backprop = sum(backprop,1);
60 gHidden{end} = gHidden{end} + fp{end-1}' * backprop;
61
62 % Other Hidden Layers
63 for h = length(nHidden)-2:-1:1
64 backprop = (backprop * hiddenWeights{h+1}') .* ...
65 sech(ip{h+1}).^2;
66 gHidden{h} = gHidden{h} + fp{h}' * backprop;
67 end
68
69 % Input Weights
70 backprop = (backprop * hiddenWeights{1}') .* sech(ip{1}).^2;
71 gInput = gInput + X(i,:) * backprop;
72
73 else % nHiddenLayers == 1
74 % Input Weights
75 gInput = gInput + X(i,:) * ...
76 ( sech(ip{end}).^2 .* (outputWeights * err') );
77 end
78 end
79
80 % Put Gradient into vector
81 g = zeros(size(w));
82 g(1:nVars*nHidden(1)) = gInput(:);
83 offset = nVars*nHidden(1);
84 for h = 2:nHiddenLayers

```

```

85 g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1};
86 offset = offset+nHidden(h-1)*nHidden(h);
87 end
88 g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
89
90 end

```

A.14 *creat_train.m*

```

1  % Edited by Yuanbo Han, Dec. 9, 2017.
2
3  load digits.mat
4
5  % Artificially creat more training examples.
6  tic
7  [n,d] = size(X);
8  Xright = zeros([n,d]);
9  Xleft = zeros([n,d]);
10 Xdown = zeros([n,d]);
11 Xup = zeros([n,d]);
12 Xclock = zeros([n,d]);
13 Xanticlock = zeros([n,d]);
14 Xbig = zeros([n,d]);
15 Xsmall = zeros([n,d]);
16 for i = 1:size(X,1)
17   Xfig = reshape(X(i,:),16,16);
18   % Translations
19   temp = imtranslate(Xfig, [0,1]);
20   Xright(i,:) = temp(:);
21   temp = imtranslate(Xfig, [0,-1]);
22   Xleft(i,:) = temp(:);
23   temp = imtranslate(Xfig, [1,0]);
24   Xdown(i,:) = temp(:);
25   temp = imtranslate(Xfig, [-1,0]);
26   Xup(i,:) = temp(:);
27   % Rotations
28   temp = imrotate(Xfig, 5, 'crop');
29   Xclock(i,:) = temp(:);
30   temp = imrotate(Xfig, -5, 'crop');
31   Xanticlock(i,:) = temp(:);
32   % Resizing
33   temp = imresize(Xfig, 1.1, 'OutputSize', [16,16]);
34   Xbig(i,:) = temp(:);
35   temp = imresize(Xfig, 0.9, 'OutputSize', [16,16]);
36   Xsmall(i,:) = temp(:);

```



```

37 end
38
39 X = [X; Xright; Xleft; Xdown; Xup; Xclock; Xanticlock; Xbig; Xsmall];
40 y = repmat(y,9,1);
41 toc
42
43 [n,d] = size(X);
44 nLabels = max(y);
45 yExpanded = linearInd2Binary(y,nLabels);
46 t = size(Xvalid,1);
47 t2 = size(Xtest,1);
48
49 % Standardize columns and add bias
50 [X,mu,sigma] = standardizeCols(X);
51 X = [ones(n,1) X];
52 d = d + 1;
53
54 % Apply the same transformation to the validation/test data
55 Xvalid = standardizeCols(Xvalid,mu,sigma);
56 Xvalid = [ones(t,1) Xvalid];
57 Xtest = standardizeCols(Xtest,mu,sigma);
58 Xtest = [ones(t2,1) Xtest];
59
60 % Choose network structure
61 nHidden = [120];
62
63 % Count number of parameters and initialize weights 'w'
64 nParams = d*nHidden(1);
65 for h = 2:length(nHidden)
66     nParams = nParams+nHidden(h-1)*nHidden(h);
67 end
68 nParams = nParams+nHidden(end)*nLabels;
69 w = randn(nParams,1);
70
71 % Train with stochastic gradient
72 maxIter = 10000;
73 stepSize = 1e-3;% * 3;
74 %momentumStrength = 0.9;
75 %delta = 0;
76 %lambda = 0.03;
77 funObj = @(w,i)MLPclassificationLoss_mat(w,X(i,:), ...
78     yExpanded(i,:), nHidden, nLabels);
79
80 tic
81 for iter = 1:maxIter
82     if mod(iter-1,round(maxIter/20)) == 0

```

```

83 yhat = MLPclassificationPredict(w,Xvalid,nHidden,nLabels);
84 fprintf('Training iteration = %d, validation error = %f\n', ...
85 iter-1,sum(yhat~=yvalid)/t);
86 end
87
88 i = ceil(rand*n);
89 [~,g] = funObj(w,i);
90 %     delta = stepSize * g - momentumStrength * delta;
91 %     w = w - delta;
92 w = w - stepSize * g;
93 end
94
95 % Evaluate test error
96 yhat = MLPclassificationPredict(w,Xtest,nHidden,nLabels);
97 fprintf('Test error with final model = %f\n',sum(yhat~=ytest)/t2);
98 toc

```

A.15 convol10.m

```

1 % Edited by Yuanbo Han, Dec. 9, 2017.
2 % Reference: http://blog.csdn.net/u010540396/article/details/52895074
3
4 load digits.mat
5 n = size(X,1);
6 nLabels = max(y);
7 yExpanded = linearInd2Binary(y,nLabels);
8 t = size(Xvalid,1);
9 t2 = size(Xtest,1);
10
11 % Standardize columns and reshape X to be an array of n pixels.
12 [X,mu,sigma] = standardizeCols(X);
13 X = reshape(X',16,16,n);
14
15 % Apply the same transformation to the validation/test data.
16 Xvalid = standardizeCols(Xvalid,mu,sigma);
17 Xvalid = reshape(Xvalid',16,16,t);
18 Xtest = standardizeCols(Xtest,mu,sigma);
19 Xtest = reshape(Xtest',16,16,t2);
20
21 % The number of neurons
22 nConv = 20;
23 nHidden = 200;
24
25 % Initialize bias.
26 bias_c = randn(1, nConv);

```

```

27 bias_f = randn(1, nHidden);
28 % Initialize convolution kernels.
29 kernel_c = randn(5,5,nConv);
30 kernel_f = randn(12,12,nHidden);
31 % Initialize weights for the full-connecting layer.
32 weight_f = randn(nConv, nHidden);
33 weight_output = randn(nHidden, nLabels);
34
35 % Train with stochastic gradient.
36 maxIter = 100000;
37 stepSize = 1e-3;
38 tic;
39 for iter = 1:maxIter
40     if mod(iter-1, round(maxIter/10)) == 0
41         yhat = CNN_predict(Xvalid, kernel_c, kernel_f, weight_f, ...
42             weight_output, bias_c, bias_f);
43         fprintf('Training iteration = %d, validation error = %f\n', ...
44             iter-1, sum(yhat~=yvalid)/t);
45         toc;
46         tic;
47     end
48
49     i = ceil(rand*n);
50     train_data = X(:, :, i);
51
52     % Convolution layer
53     state_c = zeros(12,12,nConv);
54     for k = 1:nConv
55         state_c(:, :, k) = conv2(train_data, rot90(kernel_c(:, :, k), 2), 'valid');
56         % apply tanh
57         state_c(:, :, k) = tanh(state_c(:, :, k) + bias_c(1, k));
58     end
59
60     % Full-connected layer
61     [state_f_pre, state_f_temp] = convolution_f(state_c, kernel_f, weight_f);
62     % apply tanh
63     state_f = zeros(1, nHidden);
64     for h = 1:nHidden
65         state_f(1, h) = tanh(state_f_pre(:, :, h) + bias_f(1, h));
66     end
67
68     % Output layer (Softmax)
69     output = zeros(1, nLabels);
70     for h = 1:nLabels
71         output(1, h) = exp( state_f*weight_output(:, h) ) / ...
72         sum( exp(state_f*weight_output) );

```

```

73 end
74
75 % Update weights, kernels and bias.
76 [kernel_c, kernel_f, weight_f, weight_output, bias_c, bias_f] = ...
77 CNN_update(stepSize, y(i), train_data, state_c, state_f, ...
78 state_f_temp, output, kernel_c, kernel_f, weight_f, ...
79 weight_output, bias_c, bias_f);
80 end
81
82 yhat = CNN_predict(Xtest, kernel_c, kernel_f, weight_f, weight_output, ...
83 bias_c, bias_f);
84 fprintf('Test error with final model = %f\n', sum(yhat~=ytest)/t2);
85 toc;

```

A.16 convolution_f.m

```

1 function [state_f, state_f_temp] = convolution_f(state_c, kernel_f, weight_f)
2 % CONVOLUTION_f computes the full-connected-layer values.
3 %
4 % Yuanbo Han, Dec. 9, 2017.
5
6 [nConv, nHidden] = size(weight_f);
7 [c_row, c_col, ~] = size(state_c);
8 f_row = size(state_c,1) - size(kernel_f,1) + 1;
9 f_col = size(state_c,2) - size(kernel_f,2) + 1;
10
11 state_f = zeros(f_row, f_col, nHidden);
12 state_f_temp = zeros(c_row, c_col, nHidden);
13 for n = 1:nHidden
14     count = 0;
15     for m = 1:nConv
16         count = count + state_c(:, :, m) * weight_f(m, n);
17     end
18     state_f_temp(:, :, n) = count;
19     state_f(:, :, n) = conv2(state_f_temp(:, :, n), ...
20     rot90(kernel_f(:, :, n), 2), 'valid');
21 end
22
23 end

```

A.17 CNN_update.m

```

1 function [kernel_c, kernel_f, weight_f, weight_output, bias_c, bias_f] = ...
2 CNN_update(stepSize, classify, train_data, state_c, state_f, ...

```

```

3 state_f_temp, output, kernel_c, kernel_f, weight_f, weight_output, ...
4 bias_c, bias_f)
5 % CNN_UPDATE computes the gradients of weights, kernels and bias, and then
6 % updates these parameters by stepSize using gradient descent method.
7 %
8 % Yuanbo Han, Dec. 9, 2017.
9 % Reference: http://blog.csdn.net/u010540396/article/details/52895074
10
11 % Compute the number of neurons and some sizes of matrices.
12 nHidden = size(state_f,2);
13 nLabels = size(output,2);
14 [c_row, c_col, nConv] = size(state_c);
15 [kernel_c_row, kernel_c_col] = size(kernel_c(:, :, 1));
16 [kernel_f_row, kernel_f_col] = size(kernel_f(:, :, 1));
17
18 % The temp values will record the updated values of parameters, in order
19 % not to overwrite the original values.
20 kernel_c_temp = kernel_c;
21 kernel_f_temp = kernel_f;
22 weight_f_temp = weight_f;
23 weight_output_temp = weight_output;
24
25 % Compute error.
26 label = zeros(1, nLabels);
27 label(1, classify) = 1;
28 delta_layer_output = output - label;
29
30 % Update weight_output.
31 delta_weight_output = zeros(nHidden, nLabels);
32 for n = 1:nLabels
33     delta_weight_output(:,n) = delta_layer_output(1,n) * state_f';
34 end
35 weight_output_temp = weight_output_temp - stepSize * delta_weight_output;
36
37 % Update full-connected-layer parameters (kernel_f, bias_f, weights_f).
38 delta_bias_f = zeros(1, nHidden);
39 delta_kernel_f = zeros(kernel_f_row, kernel_f_col, nHidden);
40 delta_layer_f = zeros(1, nHidden);
41 for n = 1:nHidden
42     count = 0;
43     for m = 1:nLabels
44         count = count + delta_layer_output(1,m) * weight_output(n,m);
45     end
46 % update bias_f
47 delta_layer_f(1,n) = count * (1 - tanh(state_f(1,n)).^2);
48 delta_bias_f(1,n) = delta_layer_f(1,n);

```

```

49 % update kernel_f
50 delta_kernel_f(:, :, n) = delta_layer_f(1, n) * state_f_temp(:, :, n);
51 end
52 bias_f = bias_f - stepSize * delta_bias_f;
53 kernel_f_temp = kernel_f_temp - stepSize * delta_kernel_f;
54
55 % update weight_f
56 delta_layer_f_temp = zeros(kernel_f_row, kernel_f_col, nHidden);
57 for n = 1:nHidden
58 delta_layer_f_temp(:, :, n) = delta_layer_f(1, n) * kernel_f(:, :, n);
59 end
60 delta_weight_f = zeros(nConv, nHidden);
61 for n = 1:nConv
62 for m = 1:nHidden
63 delta_weight_f(n, m) = sum(sum( delta_layer_f_temp(:, :, m) .* ...
64 state_c(:, :, n) ));
65 end
66 end
67 weight_f_temp = weight_f_temp - stepSize * delta_weight_f;
68
69 % Update convolution-layer parameters (i.e. kernel_c, bias_c).
70 % update bias_c
71 delta_layer_c = zeros(c_row, c_col, nConv);
72 delta_bias_c = zeros(1, nConv);
73 for n = 1:nConv
74 count = 0;
75 for m = 1:nHidden
76 count = count + delta_layer_f_temp(:, :, m) * weight_f(n, m);
77 end
78 delta_layer_c(:, :, n) = count .* ( sech(state_c(:, :, n)).^2 );
79 delta_bias_c(1, n) = sum(sum(delta_layer_c(:, :, n)));
80 end
81 bias_c = bias_c - stepSize * delta_bias_c;
82
83 % update kernel_c
84 delta_kernel_c_temp = zeros(kernel_c_row, kernel_c_col, nConv);
85 for n = 1:nConv
86 delta_kernel_c_temp(:, :, n) = rot90( conv2(train_data, ...
87 rot90(delta_layer_c(:, :, n), 2), 'valid'), 2);
88 end
89 kernel_c_temp = kernel_c_temp - stepSize * delta_kernel_c_temp;
90
91 % Final overwriting
92 kernel_c = kernel_c_temp;
93 kernel_f = kernel_f_temp;
94 weight_f = weight_f_temp;

```

```

95 weight_output = weight_output_temp;
96
97 end

```

A.18 *CNN_predict.m*

```

1 function [yhat] = CNN_predict(X, kernel_c, kernel_f, weight_f, ...
2 weight_output, bias_c, bias_f)
3 % CNN_predict classifies X by CNN model.
4 %
5 % Yuanbo Han, Dec. 9, 2017.
6
7 nInstances = size(X, 3);
8 yhat = zeros(nInstances, 1);
9 nConv = size(kernel_c, 3);
10 [nHidden, nLabels] = size(weight_output);
11 c_row = size(X,1) - size(kernel_c,1) + 1;
12 c_col = size(X,2) - size(kernel_c,2) + 1;
13
14 for i = 1:nInstances
15     train_data = X(:, :, i);
16
17     % Convolution layer
18     state_c = zeros(c_row, c_col, nConv);
19     for k = 1:nConv
20         state_c(:, :, k) = conv2(train_data, ...
21             rot90(kernel_c(:, :, k), 2), 'valid');
22         % apply tanh
23         state_c(:, :, k) = tanh(state_c(:, :, k) + bias_c(1, k));
24     end
25
26     % Full-connected layer
27     [state_f_pre, ~] = convolution_f(state_c, kernel_f, weight_f);
28     % apply tanh
29     state_f = zeros(1, nHidden);
30     for h = 1:nHidden
31         state_f(1, h) = tanh(state_f_pre(:, :, h) + bias_f(1, h));
32     end
33
34     % Output layer (Softmax)
35     output = zeros(1, nLabels);
36     for h = 1:nLabels
37         output(1, h) = exp( state_f * weight_output(:, h) ) / ...
38             sum( exp(state_f * weight_output) );
39     end

```

```

40 [~, yhat(i)] = max(output);
41 end
42
43 end

```

A.19 *MLP_softmax_L2.m*

```

1 function [f,g] = MLP_softmax_L2(w,X,y,nHidden,nLabels,lambda)
2 % MLP_SOFTMAX_L2 is my final model function.
3 %
4 % Yuanbo Han, Dec. 8, 2017.
5
6 [nInstances, nVars] = size(X);
7 nHiddenLayers = length(nHidden);
8
9 % Form Weights
10 inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
11 offset = nVars * nHidden(1);
12 hiddenWeights = cell(1, nHiddenLayers-1);
13 for h = 2:nHiddenLayers
14 hiddenWeights{h-1} = reshape(...
15 w(offset+1:offset+nHidden(h-1)*nHidden(h)),...
16 nHidden(h-1), nHidden(h));
17 offset = offset + nHidden(h-1) * nHidden(h);
18 end
19 outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
20 outputWeights = reshape(outputWeights, nHidden(end), nLabels);
21
22 ip = cell(1, nHiddenLayers);
23 fp = cell(1, nHiddenLayers);
24 if nargout > 1
25 % Form Gradient
26 gInput = zeros(size(inputWeights));
27 gHidden = cell(1, nHiddenLayers-1);
28 for h = 2:nHiddenLayers
29 gHidden{h-1} = zeros(size(hiddenWeights{h-1}));
30 end
31 gOutput = zeros(size(outputWeights));
32
33 f = 0;
34
35 % Compute Output
36 for i = 1:nInstances
37 ip{1} = X(i,:) * inputWeights;
38 fp{1} = tanh(ip{1});

```



```

39 for h = 2:length(nHidden)
40   ip{h} = fp{h-1} * hiddenWeights{h-1};
41   fp{h} = tanh(ip{h});
42 end
43 yhat = fp{end} * outputWeights;
44 yhat = exp(yhat) / sum(exp(yhat));
45 yhat_true = (y(i,:)==1) * yhat';
46
47 err = -log( yhat_true );
48 f = f + err;
49
50 % Output Weights
51 gOutput = gOutput - fp{end}' * (1 - yhat_true) * (y(i,:)==1) + ...
52 lambda * outputWeights;
53
54 % The bias need not be included in regularization.
55 gOutput(1,:) = gOutput(1,:) - lambda * outputWeights(1,:);
56
57 if nHiddenLayers > 1
58   % Last Layer of Hidden Weights
59   backprop = err' * sech(ip{end}).^2 .* outputWeights';
60   tempW = hiddenWeights{end};
61   tempG = gHidden{end} + fp{end-1}' * sum(backprop,1) + ...
62   lambda * tempW;
63
64   % The bias need not be included in regularization.
65   tempG(1,:) = tempG(1,:) - lambda * tempW(1,:);
66   gHidden{end} = tempG;
67
68   backprop = sum(backprop,1);
69   % Other Hidden Layers
70   for h = length(nHidden)-2:-1:1
71     backprop = (backprop * hiddenWeights{h+1}') .* ...
72     sech(ip{h+1}).^2;
73     tempW = hiddenWeights{h};
74     tempG = gHidden{h} + fp{h}' * backprop + ...
75     lambda * tempW;
76     % The bias need not be included in regularization.
77     tempG(1,:) = tempG(1,:) - lambda * tempW(1,:);
78     gHidden{h} = tempG;
79   end
80
81   % Input Weights
82   gInput = gInput - (1 - yhat_true) * X(i,:) * ...
83   ( sech(ip{end}).^2 .* outputWeights(:, y(i,:)==1)' ) + ...
84   lambda * inputWeights;

```

```

85
86 % The bias need not be included in regularization.
87 gInput(1,:) = gInput(1,:) - lambda * inputWeights(1,:);
88
89 else % nHiddenLayers == 1
90 % Input Weights
91 gInput = gInput - (1 - yhat_true) * X(i,:) ' * ...
92 ( sech(ip{end}).^2 .* outputWeights(:, y(i,:)=1)' ) + ...
93 lambda * inputWeights;
94 % The bias need not be included in regularization.
95 gInput(1,:) = gInput(1,:) - lambda * inputWeights(1,:);
96 end
97 end
98
99 % Put Gradient into vector
100 g = zeros(size(w));
101 g(1:nVars*nHidden(1)) = gInput(:);
102 offset = nVars*nHidden(1);
103 for h = 2:nHiddenLayers
104 g(offset+1:offset+nHidden(h-1)*nHidden(h)) = gHidden{h-1};
105 offset = offset+nHidden(h-1)*nHidden(h);
106 end
107 g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
108
109
110 else % nargout <= 1
111 ip{1} = X * inputWeights;
112 fp{1} = tanh(ip{1});
113 for h = 2:nHiddenLayers
114 ip{h} = fp{h-1} * hiddenWeights{h-1};
115 fp{h} = tanh(ip{h});
116 end
117 yhat = fp{end} * outputWeights;
118
119 relativeErr = yhat - y;
120 f = sum(sum(relativeErr.^2));
121 end
122 end

```