Project3(Neural Network) Report

Pingxuan Huang

I. Test different Hints

(I) Changing the structure of Neural Network

Purpose:

To explore the impact of neural network structure (number of layers, number of neurons) on the training results of the model

Experiment process:

- By changing the nHidden (recording the number of neurons in each hidden layer) vector, the training effect of different neural network is tested.

Training Result (error)	Number of Hidden layers：1 Number of hidden units:10	Layers: 1 Units:50	Layers:1 Units:100	Layers:1 Units:500	Layers:1 Units:1000
validation	0.483800	0.278800	0.234800	0.224600	0.865400
test	0.475000	0.256000	0.228000	0.207000	0.889000
	Layers:2 Units:10	Layers:2 Units:50	Layers:2 Units:100	Layers:2 Units:200	Layers:2 Units:400
validation	0.596800	0.357000	0.302200	0.277200	0.242400
test	0.589000	0.362000	0.280000	0.286000	0.252000
	Layers:5 Units:10	Layers:5 Units:50	Layers:5 Units:100		
validation	0.693200	0.559600	0.550000		
test	0.670000	0.552000	0.567000		
	Layers:10 Units:10	Layers:10 Units:50	Layers:10 Units:100		
validation	0.809600	0.809600	0.836400		
test	0.809000	0.760000	0.813000		

Result and analysis:

- In all tests, the result of validation will quickly become good within 15,000 iterations, and then fluctuate around the final value. It could be see that the model can converge quickly and fluctuate within a small range (because the "random gradient descent" is used, the iterative process will have some volatility);
- In all tests, validation and Test have the same trend, and the error rates of the two are alike in value, so the network training results can be indirectly evaluated by the classification result of validation;
- There is no strict correlation between the increase of the number of hidden layer

neurons and the improvement of classification result. The overall trend is that under a certain number of hidden layers, the increase of the number of neurons will lead to the improvement of training result at a beginning stage. However, when the number of neurons is too large, the classification result of the test set will be worse, in other words, the over-fitting phenomenon will become obvious;

- The difference in the number of hidden layers has a significant impact on the classification results. When the number of hidden layers is 1\2, the classification effect is obviously better, but when the number of hidden layers reaches 5, 10, the classification effect is rapidly deteriorated. The phenomenon of over-fitting occurs rapidly;

Analysis and conclusion:

- Increasing the number of hidden layers increases the complexity of the model, so that the model can fit the training data better, but the disadvantage of this operation--over-fitting is very obvious;
- Increasing the number of neurons can also improve the fitting result. However, compared with increasing the number of hidden layers, the over-fitting phenomenon caused by increasing neurons should be much lessened. Put another way, the changes in the structure and complexity of the model should be small;
- Taking the characteristics of the number of neurons and the number of hidden layers and the experimental results into consideration, subsequent experiments will be carried out under the setting of a single hidden layer, 400 neurons.

(II) Weight update (experimental under the condition of 'single hidden layer', '400 neurons')

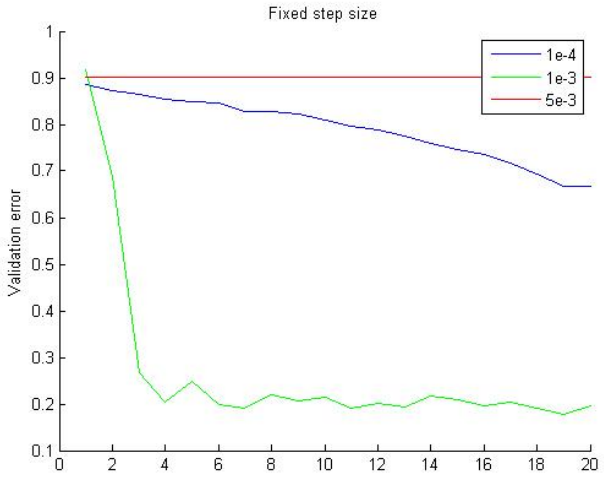
Purpose: to explore the impact of different weight update methods on the convergence process

experiment process:

- Change the weight update mode by changing the step size (non-uniform step size) and using the momentum (momentum);
- The introduction of the momentum makes the previous vector update value have an impact on this time, which is equivalent to adding a time accumulation effect. From the formula introduced by impulse:$\beta_t(w^t - w^{t-1})$, it can be seen that when w^t is the same as w^{t-1}, the impulse has an acceleration effect, vice versa;
- Experiment with different steps

a) Experiment with different steps size

Step size	1e-4	1e-3	5e-3
Test error	0.593000	0.193000	0.905000



Experimental phenomena and analysis:

- When the step size reduces, the convergence speed of the model is significantly slower (blue line), and it can be seen from the figure that the model does not converge within the number of iterations of the experiment. Although it can be expected that the final convergence effect of the model should be better after reducing the step size, the improvement of the training cost and the over-fitting phenomenon are negative effects to be considered;
- The model converges quickly after the step size increases, but the convergence result is very poor. This is in line with the theoretical expectation (the convergence speed is faster but the convergence effect may be worse)

b) Experiment by changing step size

i. Strategy 1: Since the number of iterations is 100,000, the Step size is halved (multiple by 0.5) per iteration of 20,000 times, and the final step size is (0.0325) of the initial step size(II_1.m).

- Add the following code to the iteration process：

```
if ((mod(iter-1,round(maxIter/5)) == 0 )&(iter ~= 1))
    stepSize = stepSize/2
end
```

ii. Strategy 2: Record the results of two consecutive 5000 iterations. If the result is not better (the error rate is not reduced), the step size is halved, otherwise the step size is unchanged (II_2.m)

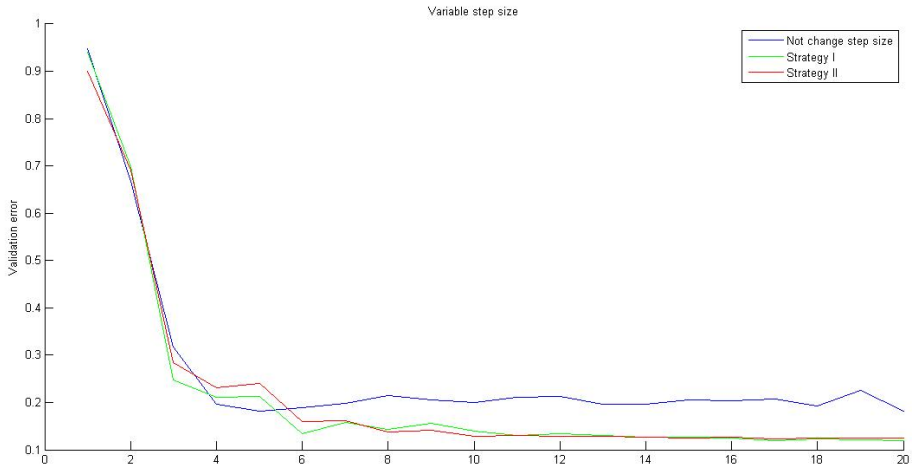
- Add the following code to the iteration process：

  Set pre_error as 1:

```
if mod(iter-1,round(maxIter/20)) == 0
    yhat = MLPclassificationPredict(w,Xvalid,nHidden,nLabels);
    current_error = sum(yhat~=yvalid)/t;
    if current_error >= pre_error
        stepSize = stepSize/2;
    else
```

```
            pre_error = current_error;
        end
    end
```

Strategy	1e-3	I	II
Test error	0.191000	0.114000	0.115000



Result and analysis:

- The result of using dynamic step size are obviously better;
- From the experimental results, the Strategy I and Strategy II are not much different. The convergence curves of the two are similar, and the final convergence effect is also similar.
- Considering that the parameter of strategy I requires manual pre-judgment of the model convergence situation, the robustness is not strong. On the contrary, the generalization ability of the strategy II is much stronger;

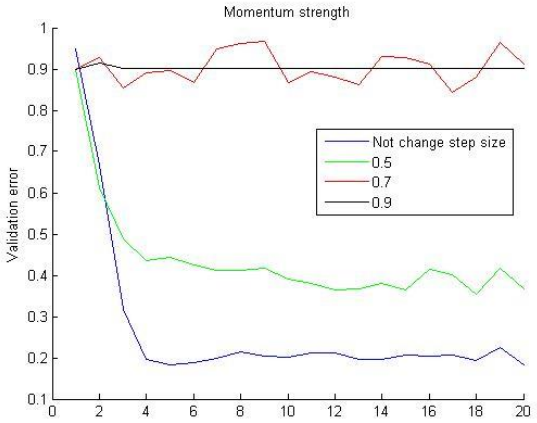
a) Add momentum for experiment

- Code to modify the weight update:

```
i = ceil(rand*n);
[f,g] = funObj(w,i);
change = - stepSize*g + belta*pre_change;    %belta:momentum strength
w = w + change;
pre_change = change;    % the previous changing value，default as [0,0,0……]
```

- Change momentum strength（β_t）for experiment:

β_t	0	0.5	0.7	0.9
Test error	0.191000	0.375000	0.853000	0.905000



Result and analysis:

- In general, the addition of momentum strength did not improve the training results, but the training results became worse, and the training results became worse as the momentum strength increased.

Analysis of results:

- The purpose of adding momentum strength is to solve the problems in the two training processes:

  1) When the step size is too small, the convergence speed is too slow, and the impulse can accelerate the training process at this time;

  2) When the step size is too large, the convergence is prone to fluctuations. At this time, the impulse can play a deceleration.
- However, from the experimental results, the introduction of impulse does not have the expected effect. This aspect may be related to the selected step size. (In the first part of this question, the step size has been tested, so the step size chosen is a relative More reasonable step size). For another thing, it may also be related to the uncertainty of the impulse itself. When the current change makes the result worse, it will also affect the subsequent changes, and this bad effect may even have a superposition effect.

(III) Vectorization

Experimental purpose: speed up training by using as many vector operations as possible (removing loops)

experiment process:

(1) Multiple hidden layers:

- Implement (MLPclassificationLoss_III.m)
- Change

```
for c = 1:nLabels
        gOutput(:,c) = gOutput(:,c) + err(c)*fp{end}';
end
```

Into:

```
gOutput = gOutput + repmat(fp{end}',[1,length(err)]) * diag(err);
```

- Change

```
for c = 1:nLabels
        backprop(c,:) = err(c)*(sech(ip{end}).^2.*outputWeights(:,c)');
        gHidden{end} = gHidden{end} + fp{end-1}'*backprop(c,:);
end
```

Into:

```
backprop = diag(err)*(outputWeights' * diag(sech(ip{end}).^2));
gHidden{end} = gHidden{end} + repmat(fp{end-1}',[1,size(backprop,1)]) * backprop;
```

- Testing with "double hidden layer, 100 neurons" neural network

	Origin	After changing
Time	141.9260s	104.1460s

(2) single hidden layer:

- Change

```
for c = 1:nLabels
      gInput = gInput + err(c)*X(i,:)'*(sech(ip{end}).^2.*outputWeights(:,c)');
end
```

Into:

```
gInput = gInput + repmat(X(i,:)',[1,size(outputWeights',1)]) * (diag(err)*(outputWeights' *
diag(sech(ip{end}).^2)));
```

- Test with (single hidden layer, 200 units) neural network:

	Origin	After changing
Time	200.4490s	96.3480s

Experimental results and analysis: It can be seen from the experimental results that the training time can be greatly shortened by rewriting the cycle into a matrix operation. This is because the number of processes is greatly reduced, so that the loop stack operation is avoided. On the other hand, because MATLAB has many optimizations for matrix operations, it can speed up the operation.。

(IV) Regularization

Purpose：

a) By adding l_2/l_1-regularization to the weights, the over-fitting phenomenon is reduced and the model generalization ability is increased;
b) Avoid the effects of invalid training and data volatility by using the early stopping strategy

Process：

1) Add regularization item（MLPclassificationLoss_IIII_1.m）

- Add l_2-regularization
- The original target function is$f(w \cdot b; x, y) = \left\| h_{w,b}(x) - y \right\|^2$， after adding the

regularization， the target function transformed into $f(w,b) = \sum_{i=1}^{n} \|h(x_i) - y_i\|^2 + \frac{\lambda}{2}\|W\|^2$； After deriving, we can see that Weights $\lambda * w_{ij}$ needs to be subtracted from w_{ij} when performing weights update.

- Modified code（based on the change in question 3）：

```
gOutput = gOutput + repmat(fp{end}',[1,length(err)]) * diag(err);
gOutput = gOutput + lambda  * outputWeights;
```

```
gHidden{end} = gHidden{end} + repmat(fp{end-1}',[1,size(backprop,1)]) * backprop;
gHidden{end} = gHidden{end} + lambda * hiddenWeights{end};
```

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

```
gInput = gInput + lambda * inputWeights;
```

- Test based on (2 hidden layers, 100 units)

λ	0	0.1	0.5	1
Validation Error	0.277200	0.064800	0.136800	0.164400
Test error	0.286000	0.076000	0.145000	0.177000

Result：

- The addition of the regularization item greatly alleviates the over-fitting phenomenon and improves the generalization ability of the model

2) early stopping（IIII_2.m）

- Use the early stopping strategy to prevent invalid iterations after the model converges, and at the same time alleviate the deterioration due to data uncertainty fluctuations;
- Considering that the convergence process may have better results after a short stop or fluctuation, set the stop condition to: perform sampling every 1000 iterations, and stop training if the sampling results are not good for 3 consecutive times. And return the weights of the best time to sample the results.
- Add code:

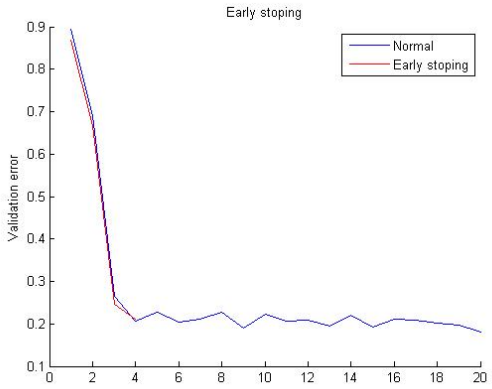
```
f0 = 1;
f1 = 1; % sign for unsampled
f2 = 1;
w_best = w; % record the best weights
```

```
if mod(iter-1,round(maxIter/100)) == 0    % sampling
        yhat = MLPclassificationPredict(w,Xvalid,nHidden,nLabels);
        error = sum(yhat~=yvalid)/t;
        if error < f0 % result is better
            f0 = error;
```

```
            w_best = w;
            f1 = 1; f2 = 1;
        elseif f1       % the first sampling
            f1 = 0;
        elseif f2
            f2 = 0; % the second sampling
        else
            w = w_best;
            break; % The results of three consecutive samplings have not improved
        end
    end
```

- Experiment with (single hidden layer,400 units)

	normal	Early stopping
Validation error	0.181400	0.212200
Test error	0.215000	0.190000



Result & analysis：

- As can be seen from the image, the model trained with the “Early stopping” strategy can stop training after converging to a relatively good result, thus saving the time of invalid training;
- Due to the volatility of the stochastic gradient, the initial model will fluctuate after converging to a relatively good result, which may cause the final result worse, and the “Early Stopping” method effectively avoids this volatility;
- It is worth bearing in mind that the use of “Early stopping” will conflict with the “validation step-size method”, which is utilized at question 2. Therefore, we need to tread off between them in specific task.

(V) Softmax

Purpose: Change the activation function to observe the impact on the training process and training results.

Process:

- Modify the MLPclassificationLoss.m into MLPclassificationLoss_V.m

- The new loss function derivates Z_k (the residual of the output layer) as：

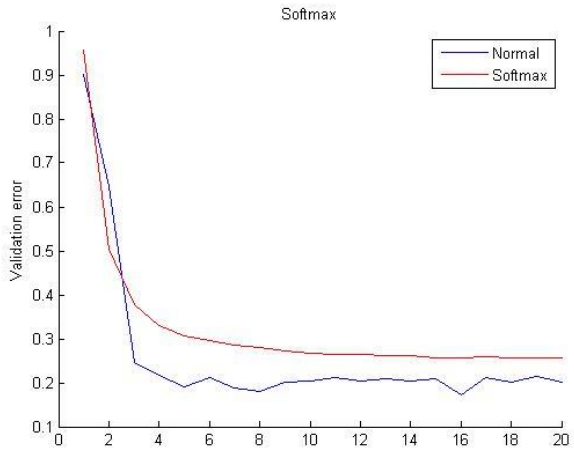
$$\begin{cases} \dfrac{\exp(z_k)}{\sum_{j=1}^{J}\exp(z_j)} & (k \neq i) \\ \dfrac{\exp(z_k)}{\sum_{j=1}^{J}\exp(z_j)} - 1 & (k = i) \end{cases}$$

- Modify the output layer:

```
yhat = fp{end}*outputWeights;
yhat = exp(yhat);
yhat = yhat/sum(yhat);
[v,yi] = max(y(i,:));    % the value of y(i,:) in term of Decimal
Erro = -log(yhat(yi));
f = f + Erro;



if nargout > 1
    err = yhat;    % the residual of output layer
    err(yi) = err(yi)-1;
```

- Experiment with (single layer, 400 units)

	normal	Softmax
Validation error	0.201200	0.255600
Test error	0.196000	0.240000



Result and analysis：

- It can be seen from the image that Softmax's convergence process is smoother than the traditional method, and there is no fluctuation. This may due to the use of a smoother logistic function for the definition of the objective function (error function);
- At the same time, Softmax still has a tendency to converge at the end of the image, but because the convergence speed drops too fast at the beginning, it can only converge at a slower rate in the later stage, which results a poor final result.

(VI) Add bias to the hidden layer

Purpose: Add a bias item in each hidden layer to observe the impact on the training process and training results.

Process:

- Set the last neuron of each hidden layer as bias, modify （MLPclassificationLoss.m）into（MLPclassificationLoss_VI.m）
- First, rewrite the weights extraction part, and change the input weights of the last neuron of each hidden layer to 0 vector, so as to cut off the relation between the bias neurons and the previous layer of neurons (it will not affect the previous one during the retreat process)

```
inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
inputWeights(:,end) = zeros(nVars,1);
```

```
  hiddenWeights{h-1} = reshape(w(offset+1:offset+nHidden(h-1)*nHidden(h)),nHidden(h-1),nHidden(h));
  hiddenWeights{h-1}(:,end) = zeros(nHidden(h-1),1);
```

- Modify the forward derivation process so that the result of the last neuron in each hidden layer is α(default)

```
fp{1} = tanh(ip{1});
    fp{1}(end) = alpha;   % alpha is the bias for every hidden layers
```

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

- Change the update value of the output weights so that the original weights and bias related parts are set to 0

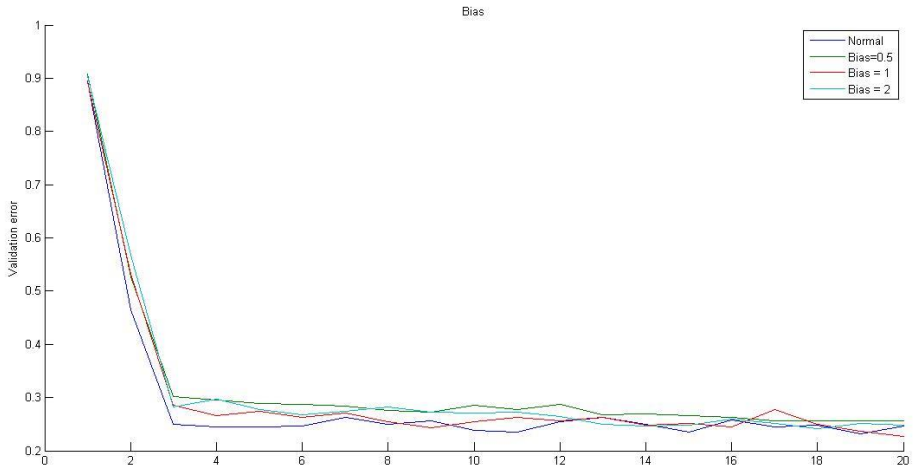
```
if nargout > 1
    g = -w;
    tem = gInput(:,1:end-1);
    g(1:nVars*(nHidden(1)-1)) = tem(:);
    offset = nVars*nHidden(1);
    for h = 2:length(nHidden)
        tem = gHidden{h-1}(:,1:end-1);
        g(offset+1:offset+nHidden(h-1)*(nHidden(h)-1)) = tem;
        offset = offset+nHidden(h-1)*nHidden(h);
    end
    g(offset+1:offset+nHidden(end)*nLabels) = gOutput(:);
end
```

- Rewrite (MLPclassificationPredict.m) prediction method (modify the forward derivation process so that the last neuron value of each hidden

layer is α) the modification of (MLPclassificationPredict_VI.m) is similar to the above

- Experiment with (single hidden layer,400 units)

	normal	Bias=0.5	Bias = 1	Bias = 2
Validation error	0.246400	0.256000	0.227600	0.247400
Test error	0.235000	0.241000	0.240000	0.225000



Result & analysis：

- From the end of the image and the final result, whether the bias is set in the hidden layer or not has little effect on the final result (the final convergence process);
- However, after adding the bias, the convergence effect in the early stage will be worse. It can be understood that the function of some neurons in the neural network declined (the ability to affect the neurons in the previous layer is lost);

(VII) Dropout

Purpose: test the influence of the dropout strategy

Experimental idea:

- here are two types of Dropout strategies: the first is to choose one of the hidden layers for Dropout, and the second is to make all hidden layers for Dropout;
- When using the Dropout strategy, you need to change the training and prediction functions at the same time.

Process:

i. Strategy 1: Select one of the hidden layers (choose the first one) for Dropout

- The key to the training process is to cut off the connection between the neurons of Dropout and all other neurons, that is, set all the weights of this neuron to 0 (including the front and back layers);

- Implementation method: randomly generate a row mask according to the probability P on the first hidden layer, diagonalize the mask and then multiply the weights of the layer by the right (eliminate the weights of the front layer and the dropout node), and multiply the left Layer weights (eliminate the weight of the layer's Dropout node and the next layer)
- The implementation code is as follows (MLPclassificationLoss_VII_1.m):

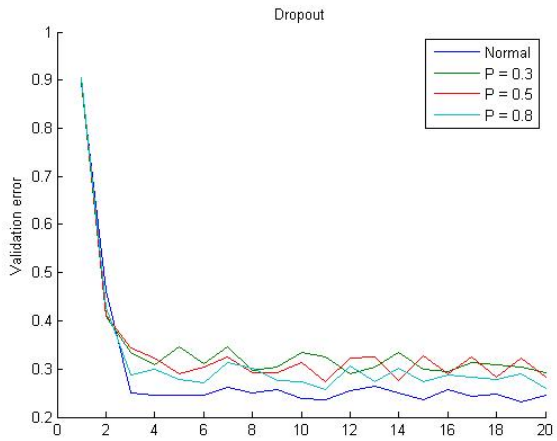
```
hide = diag(binornd(1,P,1,nHidden(1)));    % hidden mask, p is a default value
inputWeights = inputWeights * hide;    % Eliminate the weights between current and the previous layer
if(length(nHidden)>1)
    hiddenWeights{1} = hide * hiddenWeights{1}; %Eliminate the weights between current and the next layer
else % single hidden layer
    outputWeights = hide * outputWeights;
end
```

- In order to make the output of the Dropout layer in the prediction process have the same meaning and magnitude as the training process, it is necessary to rewrite the prediction function to the matching prediction function (MLPclassificationPredict_VII_1.m). The specific operation is to multiply the output result of the first hidden layer by P;
- Implementation code:

```
fp{1} = tanh(ip{1}) * P;    % P is a premise value
```

- Experiment with (2 hidden layers,200 units)

	normal	P=0.3	P = 0.5	P = 0.8
Validation error	0.246400	0.292600	0.284200	0.260600
Test error	0.235000	0.330000	0.321000	0.276000



Result and analysis:

- The purpose of Dropout is to prevent over-fitting. By hiding some certain nodes each time, it is equivalent to reducing the complexity of the model;

- From another perspective, Dropout is equivalent to generating a variety of models and then combining into one model;
- From the results, the effect is worse after adding Dropout, and the more nodes are hidden each time, the worse the final training result. It may for the reason that I haven chosen a neural network with good results before applying Dropout (the overfitting phenomenon is not obvious), and it may also for the reason that the Validation and Test have high similarity, and the operation to alleviate over-fitting couldn't achieve a good result.

ii. Strategy 2: apply the Dropout on every hidden layers

- apply the strategy 1 on every hidden layers
- Process:(MLPclassificationLoss_VII_2.m)

```
inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
hide = diag(binornd(1,P,1,nHidden(1)));    % Generate a mask for the first layer
inputWeights = inputWeights * hide;    % Eliminate the weights between current and previous layers
```

```
  hiddenWeights{h-1} = hide * hiddenWeights{h-1}; % eliminate the weights between current layer and the previous dropout layer
  hide = diag(binornd(1,P,1,nHidden(h-1)));
  hiddenWeights{h-1} = hiddenWeights{h-1} * hide ; % eliminate the weights between current dropout layer and the previous layer
```

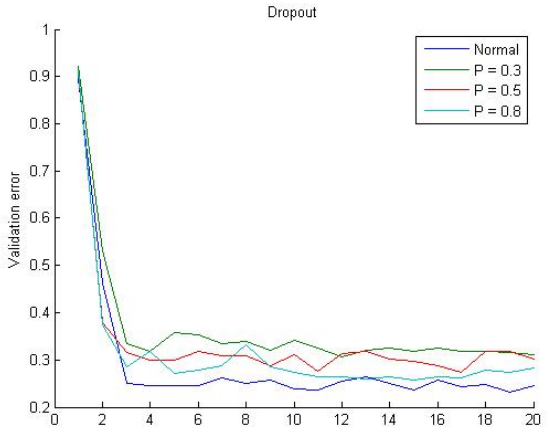
```
outputWeights = hide * outputWeights;
```

- Prediction process：

```
    ip{1} = X(i,:)*inputWeights;
    fp{1} = tanh(ip{1}) * P;
    for h = 2:length(nHidden)
        ip{h} = fp{h-1}*hiddenWeights{h-1};
        fp{h} = tanh(ip{h}) * P;
    end
```

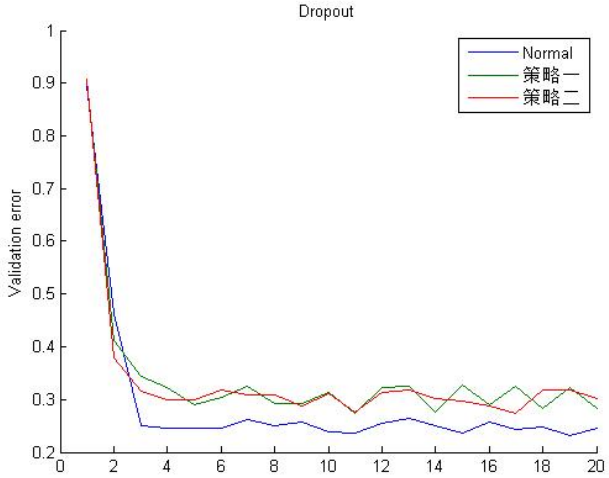
- Experiment with (2 hidden layer, 200 units)

	normal	P=0.3	P = 0.5	P = 0.8
Validation error	0.246400	0.311600	0.301600	0.284000
Test error	0.235000	0.296000	0.282000	0.263000



Result and analysis is similar to strategy 1；

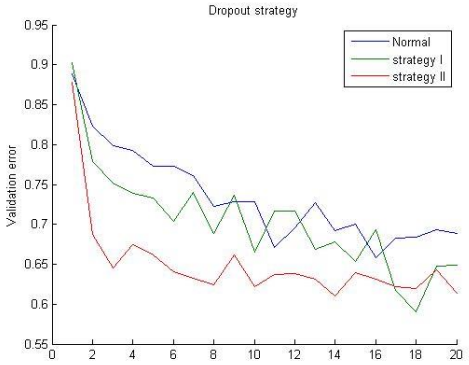
- Compare strategy 2 with strategy 1:



The convergence process and convergence result of the two strategies are very similar to each other, which shows the same trend.

- Experiment with (2 hidden layer，50 units)

	normal	Strategy I	Strategy II
Validation error	0.688600	0.649000	0.614200
Test error	0.657000	0.607000	0.563000



Result and analysis:

- The previous supposition was confirmed by the experiment conducted in the network with over-fitting. Previously, since the result was compared with that of a network with little over-fitting, after adding dropout, the result became worse. On the contrary, after applying Dropout to an over-fitting network, both the training process and training result are better;
- It is observed that Strategy 2 is better than Strategy 1. This is because Strategy 2 uses Dropout in each hidden layer, which further reduces the complexity of the model (the over-fitting phenomenon is further reduced)

(VIII) Fine-tuning

Purpose: Test the influence of Fine-tuning

Experiment idea:

- Optimize the parameters of the last layer by using the fine-tuning method each time after the weights are updated

```
w = w - stepSize*g;
w = fine_tuning(w,X(i,:),yExpanded(i,:),nHidden,nLabels); %Finetuinig
```

- Finetuning with the "gradient descent" (MSE) method:

$$\mathrm{E} = \frac{1}{2}\|\mathrm{XW} - \mathrm{Y}\|_2^2$$

$$\frac{\partial \mathrm{E}}{\partial \mathrm{W}} = (\mathrm{YW} - \mathrm{Y})^T \mathrm{X}$$

$$\mathrm{W}' = \mathrm{W} - \frac{\partial \mathrm{E}}{\partial \mathrm{W}} \cdot \alpha$$

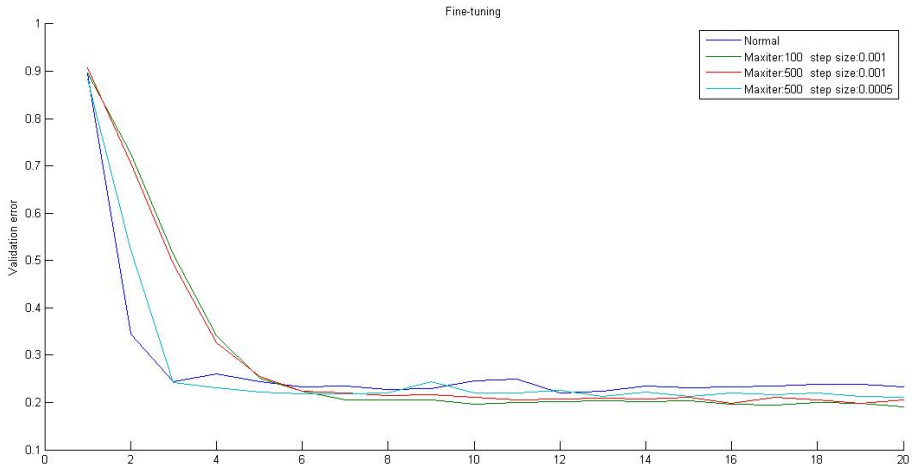
- Fine-tuning code（Gradient descent with variable step size）（fine_tuning.m）

```
function [ W ] = fine_tuning( w,X,y,nHidden,nLabels )



[nInstances,nVars] = size(X);
% Form Weights
inputWeights = reshape(w(1:nVars*nHidden(1)),nVars,nHidden(1));
offset = nVars*nHidden(1);
for h = 2:length(nHidden)
    hiddenWeights{h-1} = reshape(w(offset+1:offset+nHidden(h-1)*nHidden(h)),nHidden(h-
1),nHidden(h));
    offset = offset+nHidden(h-1)*nHidden(h);
end
outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
outputWeights = reshape(outputWeights,nHidden(end),nLabels);



ip{1} = X*inputWeights;
fp{1} = tanh(ip{1});
for h = 2:length(nHidden)
```

```
        ip{h} = fp{h-1}*hiddenWeights{h-1};
        fp{h} = tanh(ip{h});
end



XX = fp{end}; % the last input
YY = y;
Maxiter = 500;    % Gradient descent iterations
step_size = 0.01; % step size


% Gradient descent
for ite = 1 : Maxiter
        step_size = step_size / ite; % change step size
        delta = XX' * (XX * outputWeights - YY);
        outputWeights = outputWeights - step_size * delta; % gradient descent
end



% Form new weights
g = w;
offset = nVars*nHidden(1);
for h = 2:length(nHidden)
        offset = offset+nHidden(h-1)*nHidden(h);
end
g(offset+1:offset+nHidden(end)*nLabels) = outputWeights(:);
W = g;
end
```

- Experiment with (1 hidden layer，100 units)

	normal	Maxiter:100 Step_size:0.001	Maxiter:500 Step_size:0.001	Maxiter:500 Step_size:0.0005
Validation error	0.231800	0.190000	0.204600	0.210800
Test error	0.228000	0.194000	0.216000	0.203000



Result and analysis:

- The purpose of fine-tuning is to re-optimize the results with some optimization methods after the training process complete;
- The final convergence result after introducing fine-tuning is better than the original result;
- However, from the whole process, at the beginning stage, the convergence speed of the original method is obviously better than that after adding fine-tuning; and from the step-size perspective, when the step-size is small, the convergence effect is better at the beginning stage, so it can be inferred that adding fine-tuning at the beginning of training may cause fluctuations in the original convergence process;
- Because the gradient descent method has somehow volatility and uncertainty (probably get the local optimum), the relationship between the training result of fine-tuning and the number of iterations or step size is not that the more the iterations/ the less the step size, the better the result. In other words, increasing the number of invalid iterations has no effect on the training results and decreasing the step size may hinder the model from converging to a better result in the end.

(IX) Artificial data

Purpose: Improve the generalization ability by adding manual data

Process:

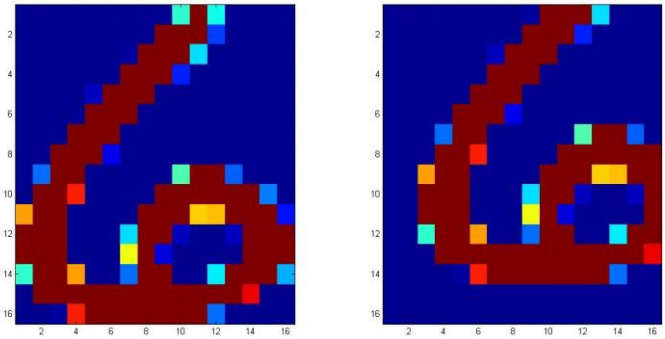
- Finish (IX_artificial_example.m) to randomly extract data from the original training set and apply (panning, rotating, scaling) to the data. Then add the generated data to the original training data.

```
clear all
load digits.mat
[n,d] = size(X);



Nsample = 400; % A total of 1200 samples added
```

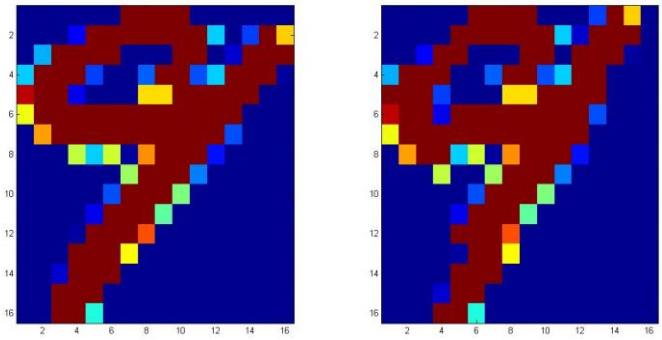
```
X_self = X;
y_self = y;


Tem = randperm(n);
tem = Tem(1:Nsample);
sample_x = X(tem,:);% Randomly sample 400 X samples
sample_y = y(tem);


y_self = [y_self;sample_y];    % add training labels


% rotation
for i = 1:Nsample
    A = reshape(sample_x(i,:),[16,16]);
    dushu = -10 + 20*rand(1);    % degree of rotation is a random number between -8 and 8
    B = imrotate(A,dushu,'crop');
    X_self = [X_self;reshape(B,[1,256])]; % add training data
end


tem = Tem(Nsample+1:2 * Nsample);
sample_x = X(tem,:);
sample_y = y(tem);


y_self = [y_self;sample_y];


% scale
for i = 1:Nsample
    A = reshape(sample_x(i,:),[16,16]);
    rate = 0.85 + 0.3 * rand(1);     % scale between 0.8 to 1.2
    b = imresize(A,rate,'bilinear');
    B = zeros(16,16);
    if rate<1     % make the scaled image the same size as the original
        k = size(b,1);
        fill = ceil((16-k)/2);
        B(fill+1:(fill+k),fill+1:(fill+k)) = b;
    elseif rate >1
        qu = ceil((rate-1)*8);
        B = b(qu:(qu+15),qu:(qu+15));
    end
```

```
        X_self = [X_self;reshape(B,[1,256])];
end


tem = Tem(2*Nsample+1:3 * Nsample);
sample_x = X(tem,:);
sample_y = y(tem);


y_self = [y_self;sample_y];


% transformation
for i = 1:Nsample
        A = reshape(sample_x(i,:),[16,16]);
        heng = round(-2 + 4 * rand(1));      % move the image left and right within 2 grids
        shu = round(-2 + 4 * rand(1)); % move the image up and down within 2 grids
        B = zeros(16,16);
        if heng <= 0 % move left
                if shu <= 0 % move down
                        B((1-shu):16,1:(16+heng)) = A(1:(16+shu),(1-heng):16);
                else % move up
                        B(1:(16-shu),1:(16+heng)) = A((1+shu):16,(1-heng:16));
                end
        else % move right
                if shu <= 0 % move down
                        B((1-shu):16,(1+heng):16) = A(1:(16+shu),1:(16-heng));
                else % move up
                        B(1:(16-shu),(1+heng):16) = A((1+shu):16,1:(16-heng));
                end
        end
        X_self = [X_self;reshape(B,[1,256])];
end


save mydata X_self y_self    % store new sampling
```

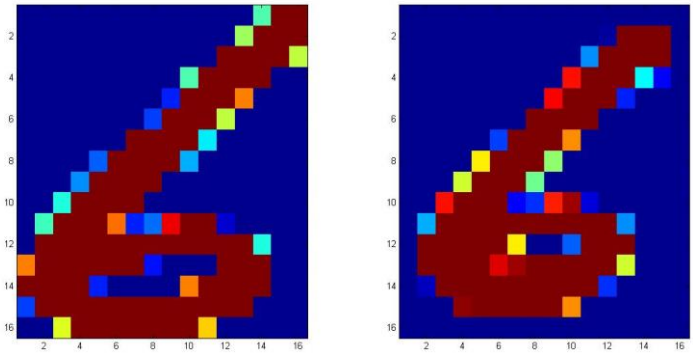
Translation example:



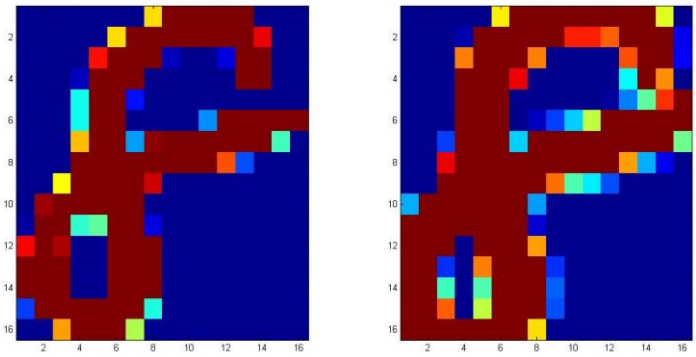
Rotation example:



Shrinkage example:



Amplification example：



- Modify the test file (example_neuralNetwork.m) to utilize the data in mydata.mat to train on a single hidden layer/100 neurons' network. Use the validation set and test set of digits.mat for testing:

	normal	mydata
Validation error	0.234800	0.245400
Test error	0.228000	0.246000

- Experiment with (5 hidden layer，50 units)

	normal	mydata
Validation error	0.688600	0.653600
Test error	0.657000	0.644000

Result and analysis:

- The purpose of adding data through transformation is to improve the generalization ability of the model and reduce the over-fitting phenomenon;
- Data as well as the noisy will be introduced into dataset; what's worse, due to the differences between transformation methods, the added samples may blur key features, so the influence of adding manual data is not definitely positive;
- The experimental results show that because the similarity between the validation (Test data) and the training data is evident, when the overfitting phenomenon of original model is not obvious, adding the manual data will make the training result worse;
- When the original model is over-fitting (the second group of experiments), however, the use of manual data can indeed improve the generalization ability of the model.

(X) Convolution layer

Purpose: Observe the effect of adding a convolutional layer on the first layer;

Process:

Due to the change of the input characteristics, all the three supporting files need to be rewritten (NNT_con.m/test file), (MLPclassificationLoss_con.m/training function), (MLPclassificationPredict_con.m/prediction function);

Only the most critical rewrites are explained here：

1. test file:

Remove the bias of the input layer (in order to reconstruct the image)：
[X,mu,sigma] = standardizeCols(X); % X = [ones(n,1) X]; % remove the bias of input layer
Change the weights dimension：
D = (sqrt(d)+1-lkernal)^2; % the amount of features after convolution nParams = D*nHidden(1);
nParams = nParams+nHidden(end)*nLabels + lkernal^2; % add kernel function

2. training function:

Convolution：

Kernal = w(1:lkernal^2); % reconstruct kernel
w = w(1+lkernal^2:end);
Kernal = reshape(Kernal,[lkernal,lkernal]);
T = [];
 	
for i = 1:nInstances
  A = reshape(X(i,:),[sqrt(nv),sqrt(nv)]); % reconstruct image
  B = conv2(A,Kernal,'valid'); % convolution
  c = reshape(B,[1,numel(B)]); % reconstruct features
  T = [T;c];
end
Input = X; % store input value for the update of convolutional weights
X = T;
[nInstances,nVars] = size(X);
gKernal = zeros(lkernal,lkernal); % the update of kernel weights

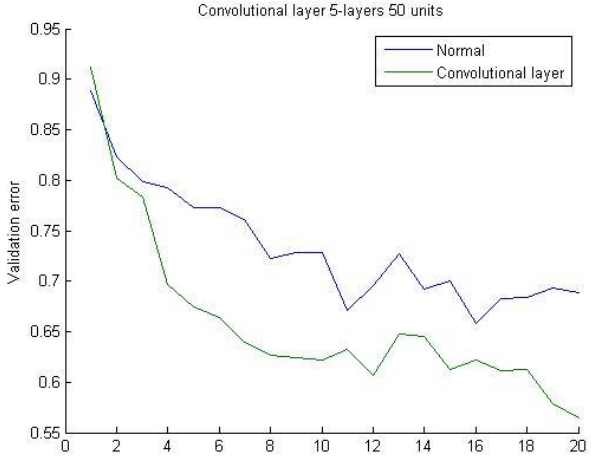
The residual of the convolution layer in multiple hidden layers: (the calculation method is the same as the BP algorithm of the neural network)
Backprop = (backprop*inputWeights').*sech(X(i,:)).^2
The residual of the convolution layer in a single hidden layer
backprop = sum((diag(err)*(outputWeights' * diag(sech(ip{end}).^2)))); % Feature layer Maps residual Backprop = (backprop*inputWeights').*sech(X(i,:)).^2; % convolutional residual image
Convolution kernel weight update: The 'residual' and "input" of the convolution layer are similar to the BP algorithm, but here we need to reconstruct the residual and input of the convolution layer into an image, and then use Reverse Convolution Method for Calculating Gradient
A = reshape(Input(i,:),[sqrt(nv),sqrt(nv)]); % reconstruct the input image t = sqrt(length(Backprop));

```
delta = reshape(Backprop,[t,t]);rot90(conv2(A,rot90(delta,2),'valid'),2); % reconstruct convolutional residual image
gKernal = rot90(conv2(A,rot90(delta,2),'valid'),2); % calculate the gradient
```

3. Predictive function: The input image needs to be convoluted first, and the process is similar to the "training file".

- Test on several neural networks with different structures

	1hiddens,50units	5hiddens,50units	1hiddens,400units
Normal			
Validation error	0.278800	0.688600	0.201200
Test error	0.256000	0.657000	0.196000
Convolutional layer			
Validation error	0.245600	0.564800	0.241600
Test error	0.236000	0.543000	0.259000



Result and analysis：

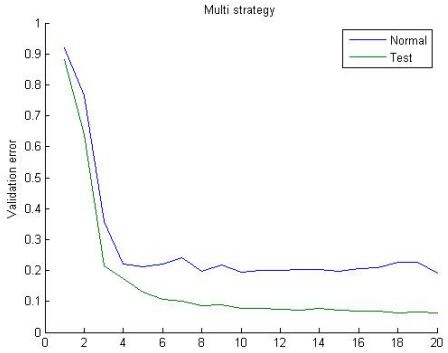
- The convolutional neural network is used to break the full connection between neurons by the introduction of the convolution method, which greatly reduces the complexity and the over-fitting of the model;
- The sharing of weights can greatly reduce the training amount, and reduce the complexity of the model. What's more, it could also reduce the over-fitting phenomenon;
- Compared whit the 5*50 data, we can clearly see the relief of the over-fitting phenomenon. It can also be seen from the image that the result is much better after adding the convolution layer, both in terms of convergence speed and final result.;
- However, from the opposite perspective, the introduction of convolution will lead to a reduction in the number of features, at the same time lead to a reduction in the complexity of the model. Therefore, when the original training result is good, the use of convolutional layer cannot significantly improve the model prediction ability. And sometimes the training model will be worse, as can be seen from the other two sets of experimental results.

II. Comprehensive test

This section will use several methods that perform better in the first part for combined experiments.

(I) Experiment with "Dynamic step size(strategy II)", "vectorization", "L2_Regularization" (1 hidden layer，500 units)

	Normal	Multi-strategy
Validation error	0.190400	0.062800
Test error	0.209000	0.058000

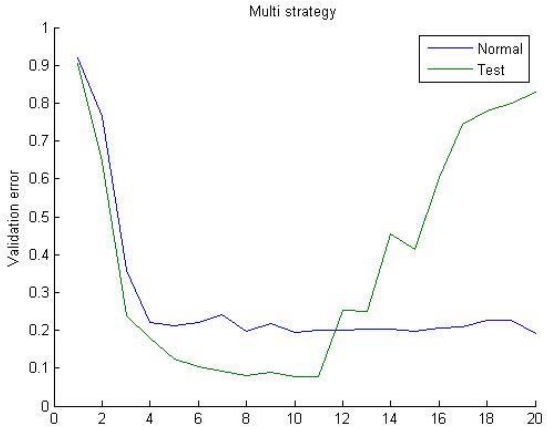


Result and analysis:

- The results of the combined strategy are significantly improved, and the combined result has several strategic advantages: the penalty term prevents over-fitting; the variable step size reduces the volatility and stabilizes the convergence process;
- It can be seen from the figure that the training process continues to converge. It is the advantage of the transform step size method. It can be predicted that if the number of iterations increases, the result can be better;

(II) Experiment with "Dynamic step size(strategy II)", "vectorization", "L2_Regularization", "Fine-tuning" (1 hidden layer，500 units)(Considering that "Fine-tuning" will cause certain fluctuations in the early stage of training, and affect the training rate, it is stipulated that "Fine-tuning" is enabled when the iteration is more than half)

	Normal	Multi-strategy
Validation error	0.190400	0.827600
Test error	0.209000	0.855000



Result and analysis:

- The result plumped after the “Fine-tuning” is enabled;
- The introduction of ‘Fine-tuning’ may make the result worse. What’s worse, when the result of original model is good, the use of “Fine-tuning” is more likely to cause the model to deteriorate;
- Because the variable step size strategy is used, the step size will be reduced after the “Fine-tuning” results deteriorate, which will result in the failure to return to the better result(even possible to go worse)

III. Conclusion

This experiment has tried various neural network optimization strategies. Through experiments and comparisons, we can see that the purpose of different strategies are different. For example, the use of Dropout and convolution is to prevent “over-fitting”, and the effect of “variable step size” is to enable the convergence result gets closer to the best result. Therefore, different strategies need to be utilized under actual situations.

The use of combination strategies is two-sided. The purpose of different optimization strategies is likely to be conflicting. Even optimization strategies with the same purpose may affect each other. The first combination strategies of this experiment work together to improve the final result, while the second combination strategy shows that when the intrinsic characteristics of the two strategies are contradictory, the training results will deteriorate.