Project 1

赵心怡 19307110452

2022/4/7

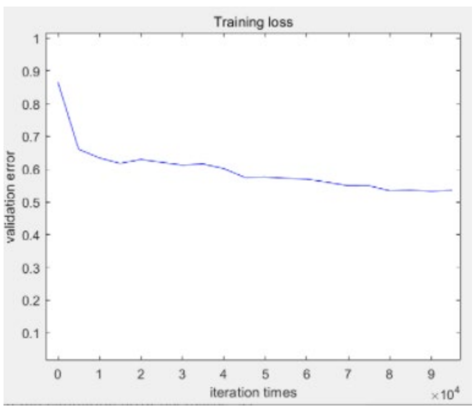
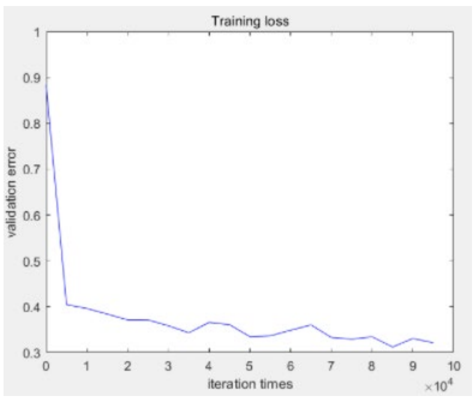
Task1： Change the network structure: the vector nHidden specifies the number of hidden units in each layer.

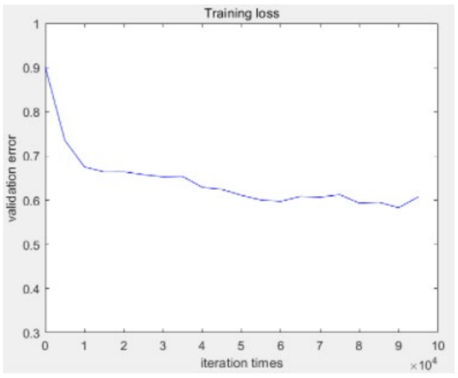
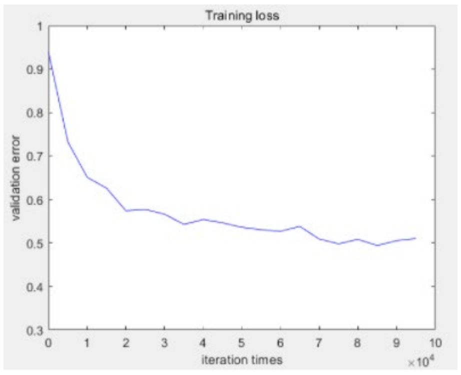
nHidden 表示了每个 layer 的 hidden unit。 ‘nHidden = [n]’表示了单层的神经网络， [n1,n2,n3]代表了三层的神经网络，以此类推。

nhidden 的长度代表了神经网络的深度，因此我们可以通过修改这个参数来改变网络的结构。

```
% Choose network structure
nHidden = [10];
% nHidden = [30];
% nHidden = [10,10];
% nHidden = [10,30,30];
```

n-hidden=[10]
 	
test-error=0.513000
	n-hidden=[30]
	
	test error = **0.313000**

n-hidden=[10,10] test error: 0.59300	n-hidden = [10,30,30] test error=0.48900
	

从结果上可以看出 n-hidden 单层的神经元增加会减少 test error，使模型更加准确，运算时间成比例增加。

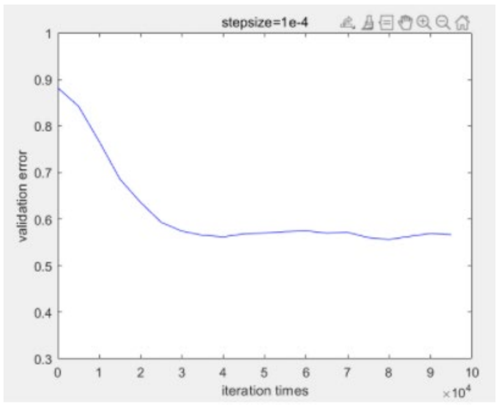
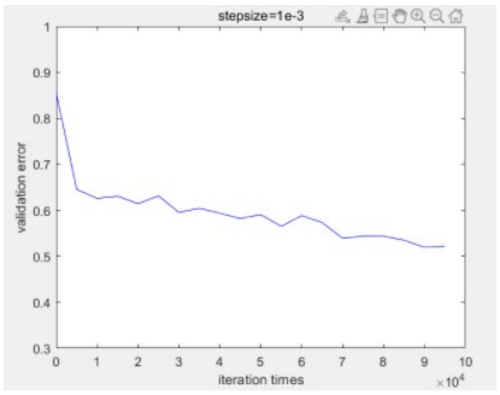
层数 layer 的增加不一定能让 test error 降低，同时它的运行速度减慢，因此当单层的神经元为 30 的时候模型表现最好。

Task2: Change the training procedure by modifying the sequence of step-sizes or using different step-sizes for different variables.

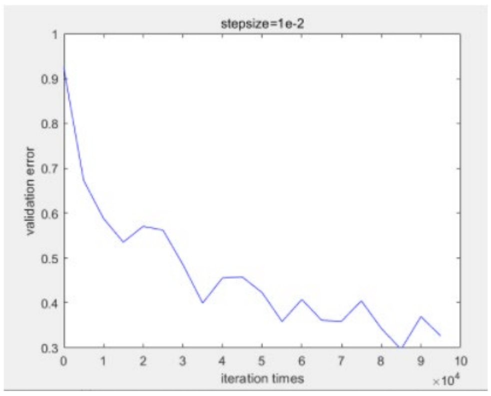
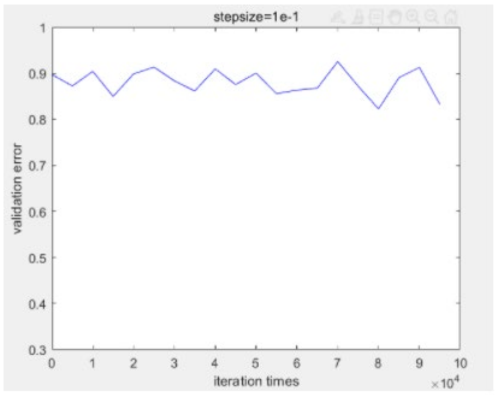
```
% Train with stochastic gradient
maxIter = 100000;
stepSize = 1e-4;
% stepSize = 1e-3;
% stepSize = 1e-2;
%stepSize = 1e-1;
```

step-size 的大小可以影响决定什么时候模型函数能达到局部收敛已经什么时候停止。

step-size = 1e-4
 	
test-error=0.542000
	step-size=1e-3
	
	test-error = 0.514000

step-size =1e-2
 	
test-error = **0.397000**
	step-size=1e-1
	
	test-error = 0.943000

从图中可以发现当 stepsize 设置过大会导致走的过快而不收敛，而步长过小会导致收敛速度很慢，由上结果可以看出这四个取值中的最优的梯度是 1e-2

Task3: You could vectorize evaluating the loss function (e.g., try to express as much as possible in terms of matrix operations), to allow you to do more training iterations in a reasonable amount of time.

我们用矩阵的形式来尽可能表示损失函数。主要修改的地方如下

1. 删了 outputweights，存在 hiddenweights 最后

```
% outputWeights = w(offset+1:offset+nHidden(end)*nLabels);
% outputWeights = reshape(outputWeights,nHidden(end),nLabels);
hiddenWeights{length(nHidden)} = w(offset+1:offset+nHidden(end)*nLabels);
hiddenWeights{length(nHidden)} = reshape(hiddenWeights{length(nHidden)},nHidden(end),nLabels);
% saves in hiddenweights in the last
```

2. 修改了 gInput 的表示，在这里不再需要讨论 nHidden 的情况。删除了 gOutput，存在 gHidden 的最后

```
if nargout > 1
    % no need to discuss nhidden
    err = 2*relativeErr;
    for h = length(nHidden):-1:1
        gHidden{h} = fp{h}'* err;% equals gOutput
        err = sech(ip{h}).^2 .* (err * hiddenWeights{h}');
    end
    gInput = X(i,:)' * err;
end
```

测试了在 nHidden=[30], step-size=1e-2 的情况下前后两种函数的耗时:

Original time	13.840808s
Matrix time	**6.725201s**

在 nHidden=[10, 10], step-size=1e-2 的情况下：

Original time	57.829088s
Matrix time	**6.653249**

从结果上可以看出，将函数转换成矩阵形式可以显著加快运算速度，当层数多的时候矩阵函数的优化效果更明显。

因为在 nhidden=[30]的时候省的时间是 2 倍，故将迭代次数改为 20 0000 次

Task4: Add l2 regularization (or l1-regularization) of the weights to your loss function. For neural networks this is called weight decay. An alternate form

of regularization that is sometimes used is early stopping, which is stopping training when the error on a validation set stops decreasing.

high variance 也就是 overfitting 的时候，我们可以使用 regularization 来处理。

正则化函数可以表示为：

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})+\frac{\lambda}{2m}||w||_2^2$$

L2 正则损失部分是神经网络中每一层的权重矩阵 $W^{[l]}$的 Frobenius 范数的平方和，假设第 l 层的权重矩阵维度是 $W^{[l]}\in R^{n_l\times n_{l-1}}$，计算结果如图：

$$||w||_2^2=\sum_{l=1}^{L}||W^{[l]}||_F^2=\sum_{l=1}^{L}\sum_{i=1}^{n_l}\sum_{j=1}^{n_{l-1}}{W_{i,j}^{[l]}}^2$$

我们可以通过修改λ的值检验随着λ的变化测试误差能否变小。

修改部分代码：

```
f = f + lambda * norm(inputWeights,'fro');
for h = 2:length(nHidden)
    ip{h} = fp{h-1}*hiddenWeights{h-1};
    fp{h} = tanh(ip{h});
    f = f + lambda * norm(hiddenWeights{h-1},'fro');
end
yhat = fp{end}*hiddenWeights{end};
f = f + lambda * norm(hiddenWeights{end},'fro');

  gInput = X(i,:)' * err+2 * lambda * inputWeights;
```

不同 lambda 下的测试结果：

λ	Test error
1e-4	0.150000
1e-3	**0.082000**

1e-2	0.111000
1e-1	0.315000

从结果看出λ过小正则化效果不明显，λ过大模型可能比较简单，有欠拟合的风险。

在$\lambda = 1e - 3$的时候模型表现最好， 因此我们选择 1e-3.

Task5: Instead of using the squared error, use a softmax (multinomial logistic) layer at the end of the network so that the 10 outputs can be interpreted as probabilities of each class.

Softmax 函数是：

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

将损失函数对 y 求导

$$\frac{\partial loss}{\partial y_{truelabel}} = p(y_{truelabel}) - y_{truelabel}$$

修改的部分代码：

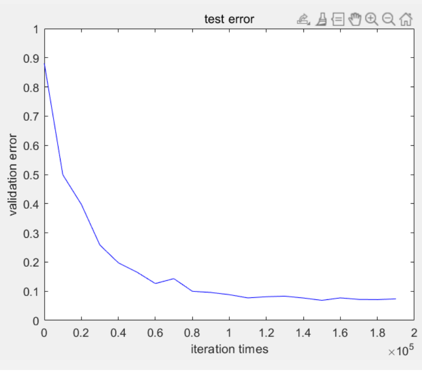
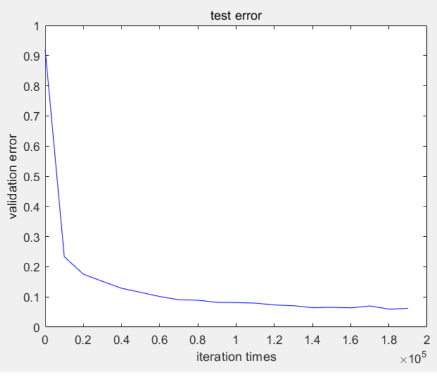
```
funObj = @(w,i)SoftmaxLoss(w,X(i,:),y(i,:),nHidden,nLabels,lambda);
```

```
        yi = fp{end}*hiddenWeights{end};
        yhat = exp(yi)./sum(exp(yi));
        f = f + lambda * norm(hiddenWeights{end},'fro');
        f = f + (- log(yhat(y(i))));

    if nargout > 1
%           err = 2*relativeErr;
          err = yhat;
          err(y(i)) = err(y(i)) - 1;
```

然后修改对应的 predict 中的函数

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

修改前后的结果比较：

	MLP	SOFTMAX
TIME	13.051451	12.370367
TEST ERROR	0.077000	**0.062000**
FIGURE		

结果中可以看出用 softmax 函数的结果 test error 更小，而且从图像上看下降速度更快。

Task6: Instead of just having a bias variable at the beginning, make one of the hidden units in each layer a constant, so that each layer has a bias.

在每一个隐藏层中加一个单元作为偏移量。

偏移量参数的更新公式如下：

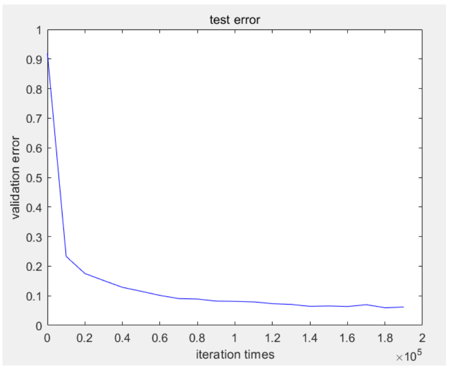
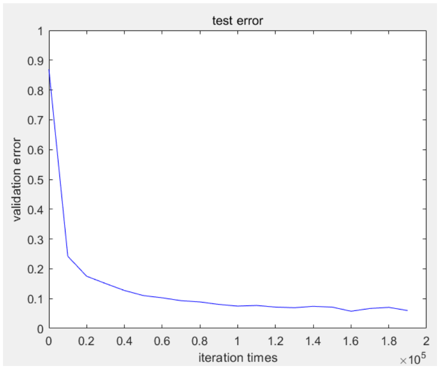
$$b_i^l = b_i^l - \alpha \frac{\partial loss(W,b)}{\partial b_i^l}$$

向隐藏层添加 bias 的部分代码：

```
for h = 2:length(nHidden)
  hiddenWeights{h-1} = reshape(w(offset+1:offset+(nHidden(h-1)+1)*nHidden(h)),nHidden(h-1)+1,nHidden(h));
  offset = offset+(nHidden(h-1)+1)*nHidden(h);
end
hiddenWeights{length(nHidden)} = w(offset+1:offset+(nHidden(end)+1)*nLabels);
hiddenWeights{length(nHidden)} = reshape(hiddenWeights{length(nHidden)},nHidden(end)+1,nLabels);
% saves in hiddenweights in the last
```

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

修改前后的结果比较

	WITHOUT BIAS	**BIAS**
TEST ERROR	0.062000	**0.060000**
FIGURE		

结果可以看出增加 bias 后 test error 的变小程度不大，经过多次测试几乎没有影响，我们猜测原因是我们的神经网络只有一层。
在后面的代码中依然保留 bias。

Task7: Implement “dropout", in which hidden units are dropped out with probability p during training. A common choice is p = 0.5.

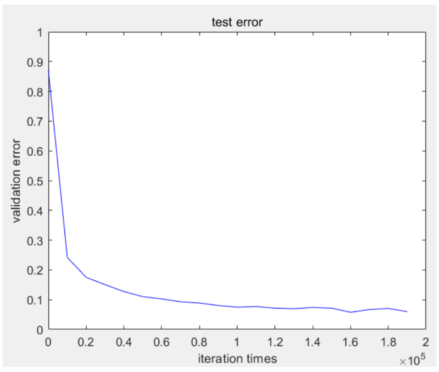
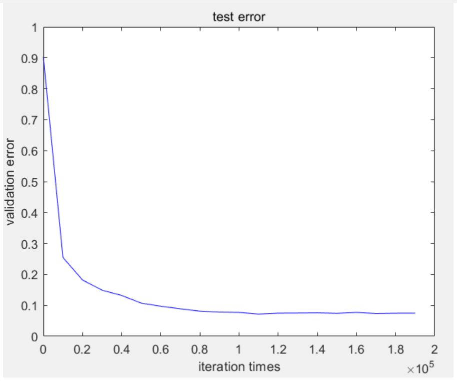
dropout 正则化可以减少过度拟合和提高深度神经网络的泛化能力。在训练期间，一些层输出被随机忽略或“*丢弃*”。 因为 dropout 下的层的输出是随机子采样的，所以在训练期间具有降低容量或细化网络的效果。当使用 dropout 时，可能需要更广泛的网络，例如更多节点。

我们在 softmax 算法的基础上增加了 dropoutMatrix

```
for h = 1:length(nHidden)
    dropoutMatrix{h} = rand(1, nHidden(h))<p;
end
% Compute Output
for i = 1:nInstances
    ip{1} = X(i,:)*inputWeights;
    fp{1} = [1,dropoutMatrix{1} .* tanh(ip{1})];
    f = f + lambda * norm(inputWeights,'fro');
    for h = 2:length(nHidden)
        ip{h} = fp{h-1}*hiddenWeights{h-1};
        fp{h} = [1,dropoutMatrix{h} .* tanh(ip{h})];
        f = f + lambda * norm(hiddenWeights{h-1},'fro');
    end
```

```
err = dropoutMatrix{h} .* sech(ip{h}).^2 .* (err * hiddenWeights{h}(2:nHidden(h)+1,:)');
```

修改前后的结果比较

	WITHOUT DROPOUT	**DROPOUT**
TEST ERROR	**0.060000**	0.065000
FIGURE		

从结果上看 dropout 可能会对提升 test error 但是变化不太明显。但 dropout 可以有效缓解过拟合问题。 为了在本实验中得到最好的测试误差，我们选择不添加 dropout。 关于 dropout 的代码可以在 task7 文件夹中

Task8: You can do ‘fine-tuning’ of the last layer.

在已经拥有训练好的模型的情况下利用微调可以降低训练成本。

确定除最后一层外的所有层的参数，并将最后一层的参数作为凸优化问题求解。

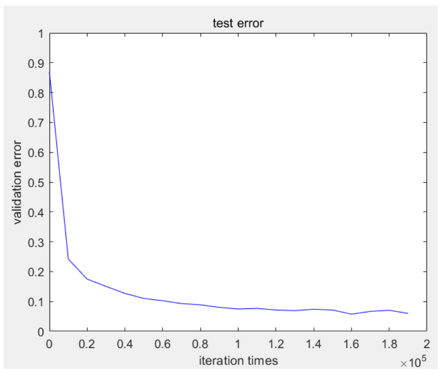
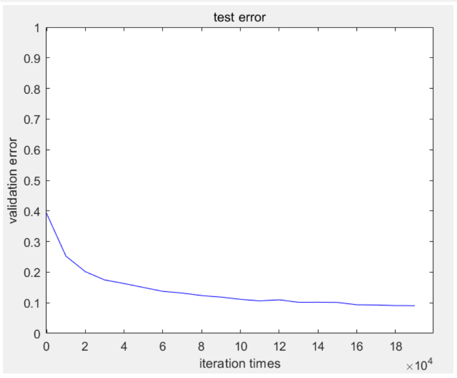
在 Sample 代码中已经存在线性回归模型 linearsquare basis，我们重新用他的显式解 X'*X\X'Y 来简化代码。 添加"微调"的代码如下：

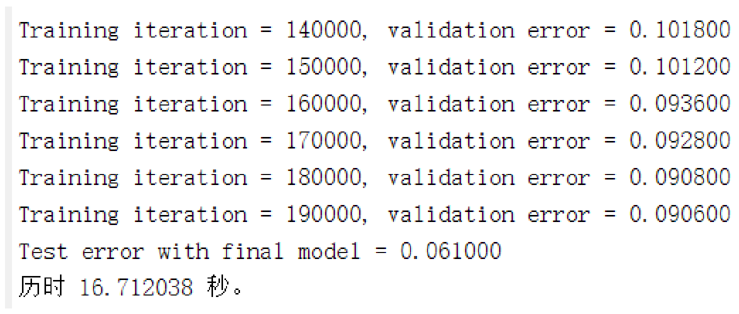
```
if mod(iter-1,round(maxIter/20)) == 0
    [~,j] = SoftmaxPredict(w,Xvalid,nHidden,nLabels);
    k = zeros(size(j,2),nLabels);
    yExpand = linearInd2Binary(yvalid,nLabels);
    for instance = 1:nLabels
        % least square
        weight = (j'*j)\j'*yExpand(:,instance);
        model.w = weight;
        model.predict = @predict;
        k(:,instance)=model.w;
    end
    yhat = j*k;
    [~,yhat] = max(yhat,[],2);
```

Predict 部分

```
y(i,:) = fp{end}*outputWeights;
j(i,:) = fp{end};
```

修改前后的结果比较

	WITHOUT FINETUNING	**FINETUNING**
TEST ERROR	**0.060000**	0.061000
FIGURE		



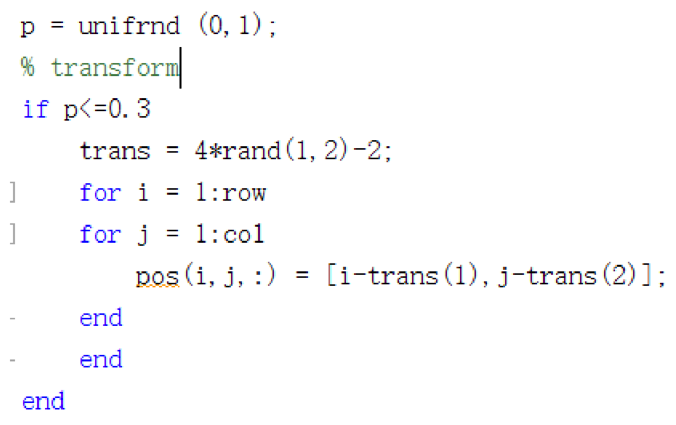
```
Training iteration = 140000, validation error = 0.101800
Training iteration = 150000, validation error = 0.101200
Training iteration = 160000, validation error = 0.093600
Training iteration = 170000, validation error = 0.092800
Training iteration = 180000, validation error = 0.090800
Training iteration = 190000, validation error = 0.090600
Test error with final model = 0.061000
历时 16.712038 秒。
```

从上面的结果可以看出，虽然添加了“微调”，但测试误差并没有显著降低，甚至在验证中发现会比没有微调的结果要高一些。然而，通过“微调”可以在实验开始时获得更好的验证误差。考虑到测试误差没有明显变化，本实验没有使用“微调”。微调的代码保存在 task8 文件夹中。

Task9: You can artificially create more training examples, by applying small transformations (translations, rotations, resizing, etc.) to the original images.

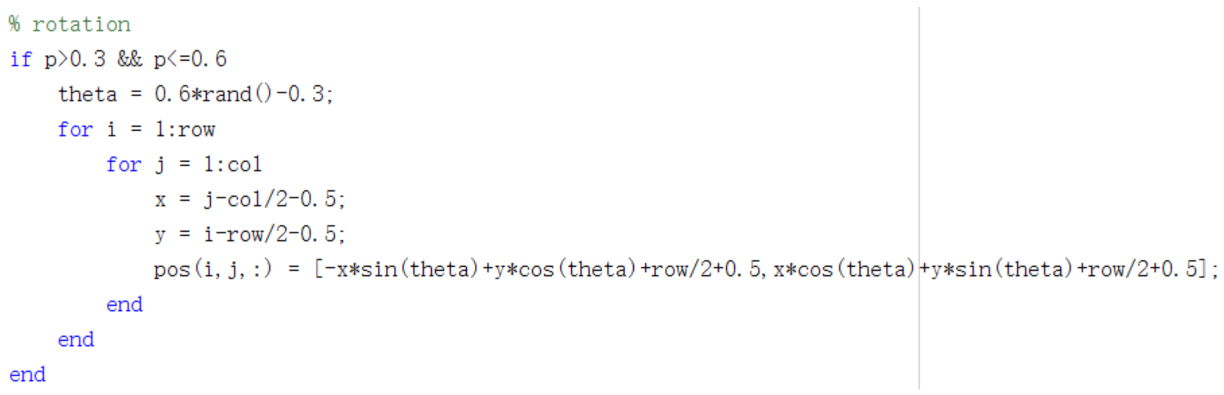
在 Matlab 中平移、旋转、翻转图像矩阵的操作都可以通过函数来完成。 具体代码如下:

Transform:



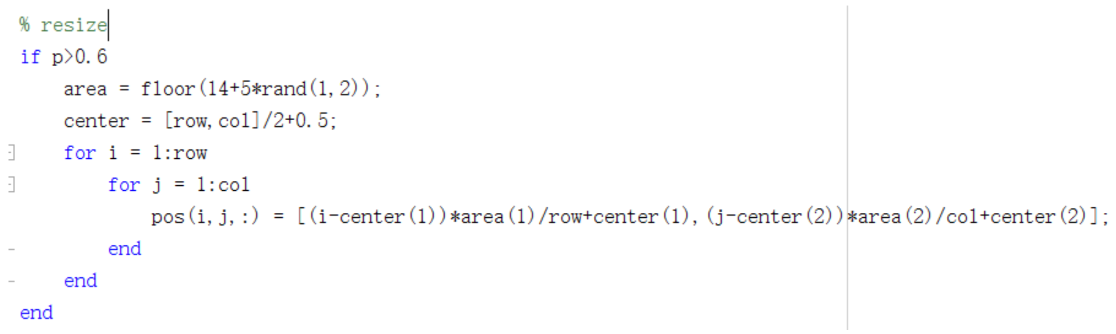
```
p = unifrnd (0,1);
% transform
if p<=0.3
    trans = 4*rand(1,2)-2;
    for i = 1:row
    for j = 1:col
        pos(i,j,:) = [i-trans(1),j-trans(2)];
    end
    end
end
```

Rotation



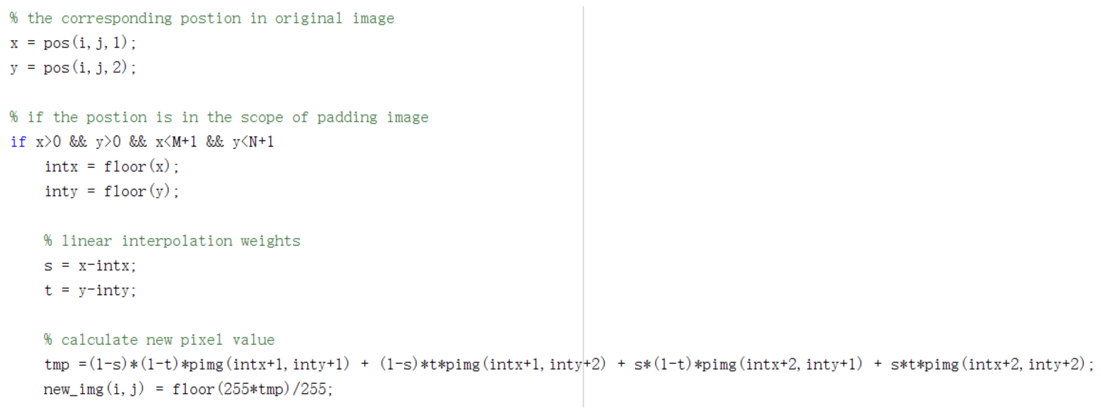
```
% rotation
if p>0.3 && p<=0.6
    theta = 0.6*rand()-0.3;
    for i = 1:row
        for j = 1:col
            x = j-col/2-0.5;
            y = i-row/2-0.5;
            pos(i,j,:) = [-x*sin(theta)+y*cos(theta)+row/2+0.5,x*cos(theta)+y*sin(theta)+row/2+0.5];
        end
    end
end
```

Resize



```
% resize
if p>0.6
    area = floor(14+5*rand(1,2));
    center = [row,col]/2+0.5;
    for i = 1:row
        for j = 1:col
            pos(i,j,:) = [(i-center(1))*area(1)/row+center(1),(j-center(2))*area(2)/col+center(2)];
        end
    end
end
```

我们将处理后的点的坐标保存在 pos 矩阵中然后用 bilinear_interpolation 插值到新图像中



```
% the corresponding postion in original image
x = pos(i,j,1);
y = pos(i,j,2);

% if the postion is in the scope of padding image
if x>0 && y>0 && x<M+1 && y<N+1
    intx = floor(x);
    inty = floor(y);

    % linear interpolation weights
    s = x-intx;
    t = y-inty;

    % calculate new pixel value
    tmp =(1-s)*(1-t)*pimg(intx+1,inty+1) + (1-s)*t*pimg(intx+1,inty+2) + s*(1-t)*pimg(intx+2,inty+1) + s*t*pimg(intx+2,inty+2);
    new_img(i,j) = floor(255*tmp)/255;
```

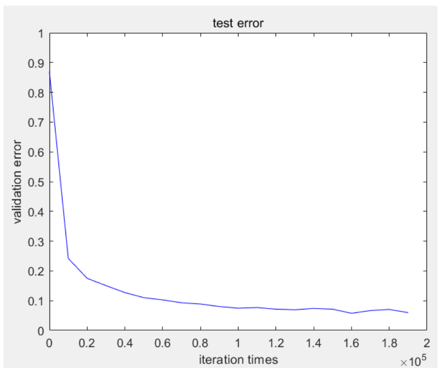
处理后的数据保存在 transformed_digits.mat 文件中

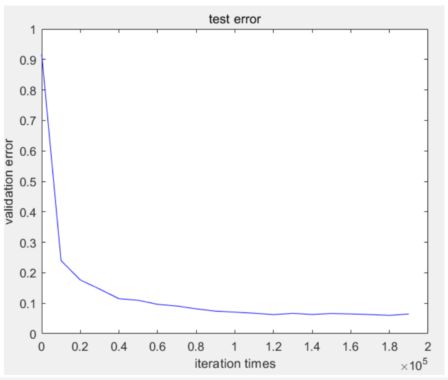
我们用随机转换处理后的训练和验证数据对模型进行训练

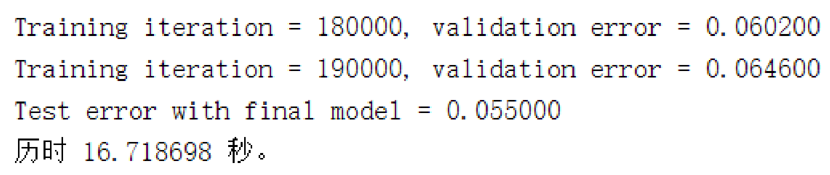
修改前后的结果比较：

	WITHOUT TRANSFORM	**TRANSFORM**
TEST ERROR	0.060000	0.055000

FIGURE







```
Training iteration = 180000, validation error = 0.060200
Training iteration = 190000, validation error = 0.064600
Test error with final model = 0.055000
历时 16.718698 秒。
```

从结果上看验证误差的变化不大但是测试误差有明显的下降。这意味着该模型可以更好地防止过拟合。我们选取处理后的 data 来训练模型。

Task10: Replace the first layer of the network with a 2D convolutional layer. You will need to reshape the USPS images back to their original 16 by 16 format. The Matlab conv2 function implements 2D convolutions. Filters of size 5 by 5 are a common choice.

卷积层损失函数的偏导可以用下式表示，我们设 kernel_size=5, batch_size=3

$$\frac{\partial l}{\partial W^{\{(l,p,d)\}}} = \delta^{(l,p)} \otimes X^{l-1,d}$$

同时我们采用了 transformed_data 来训练模型。nHidden 设为[100],

lambda=1e-3, stepsize=1e-2.

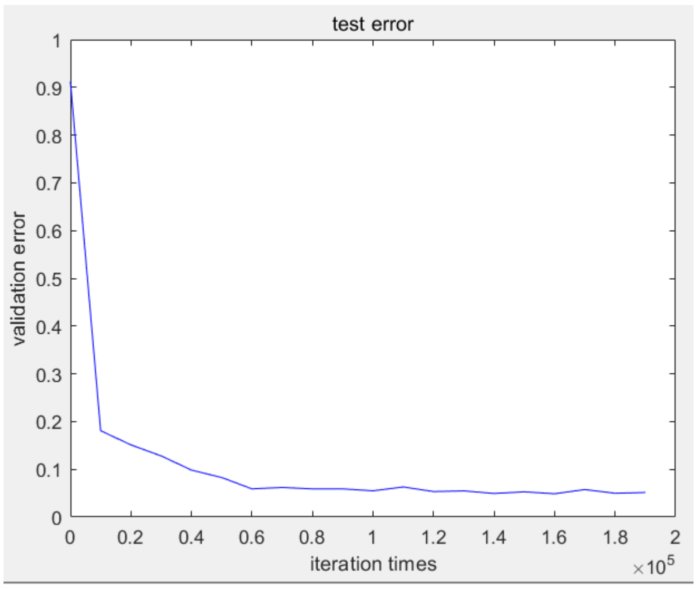
部分代码如下：

```
gConv = zeros(size(convWeights));
f = 0;
if nargout > 1
    gInput = zeros(size(inputWeights));
    gOutput = zeros(size(outputWeights));
end

% Compute Output
for i = 1:nInstances
    convInput = reshape(X(i,1:256),16,16);
    convOutput = conv2(convInput,convWeights,'valid');
    Z = reshape(convOutput,1,144);
    ip = Z * inputWeights;
    fp = tanh(ip);
    z = fp * outputWeights;
    yhat = exp(z) ./ sum(exp(z));


if nargout > 1
    err = yhat - (y(i,:) == 1);
    gOutput = gOutput + fp' * err + 2*lambda * outputWeights;
    backprop = err * (repmat(sech(ip),nLabels,1).^2.*outputWeights');
    gInput = gInput + Z'* backprop + 2*lambda*inputWeights;
    bias=[];
    bias= inputWeights;
    [~,col]=size(bias);
    bias(:,col)=0;
    gInput=gInput+lambda*bias;
    backprop = backprop * inputWeights';
    reverseX = reshape(X(i,end:-1:1),16,16);
    backprop = reshape(backprop,12,12);
    gConv = gConv + conv2(reverseX, backprop, 'valid') + 2*lambda*convWeights;
end
```

卷积神经网络的结果:

TEST error:　0.043000

```
Training iteration = 170000, validation error = 0.055800
Training iteration = 180000, validation error = 0.052400
Training iteration = 190000, validation error = 0.045600
Test error with final model = 0.043000
历时 159.533195 秒。
```



从结果看出卷积神经网络运算速度会慢一些，但通过卷积神经网络可以达到更低的测试误差。

总结：

通过我们的优化，最终的测试误差可以降低到 0.043000.