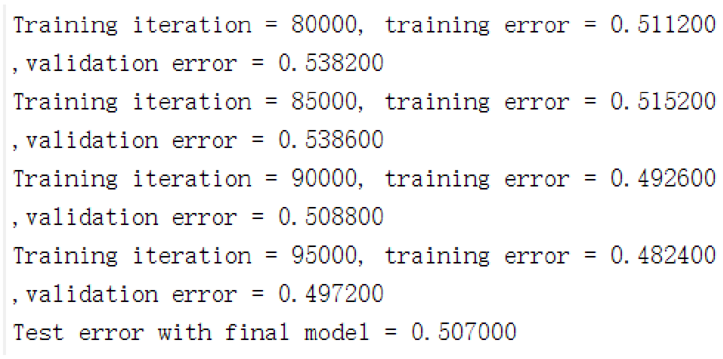
Neural Networks and Deep Learning Project 1

(Cheng Zhong 16307110259)

0.Baseline

First, I tested the performance of the neural network itself. As the number of iterations increased, both the training error and the validation error decreased. The figure is shown below. After training for 10^5 iterations, the best test error is 0.507.



```
Training iteration = 80000, training error = 0.511200
,validation error = 0.538200
Training iteration = 85000, training error = 0.515200
,validation error = 0.538600
Training iteration = 90000, training error = 0.492600
,validation error = 0.508800
Training iteration = 95000, training error = 0.482400
,validation error = 0.497200
Test error with final model = 0.507000
```

Figure 1: Screenshot of Baseline result

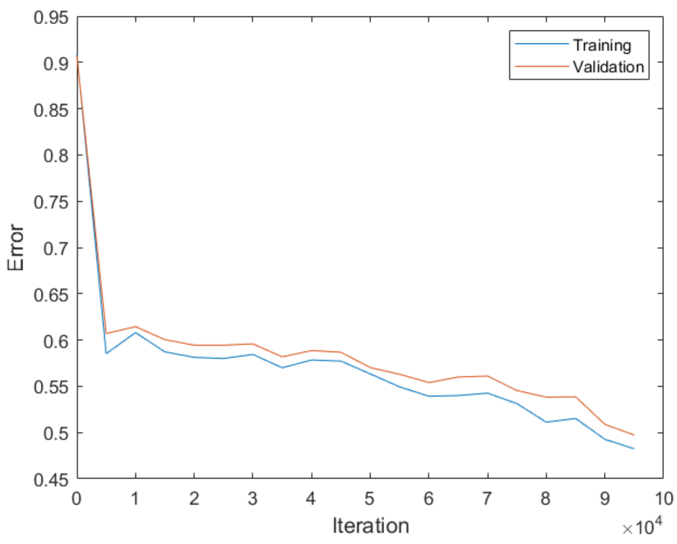


Figure 2: The training process

1. Change the Network structure

In this part, I change the number of the hidden units by change the hyper-parameter nHidden. The result is shown below.

In **Figure 3**, I change the number of hidden units from 1 to 200. From the figure, we can see that the Training error and Validation error both decreased when the number of

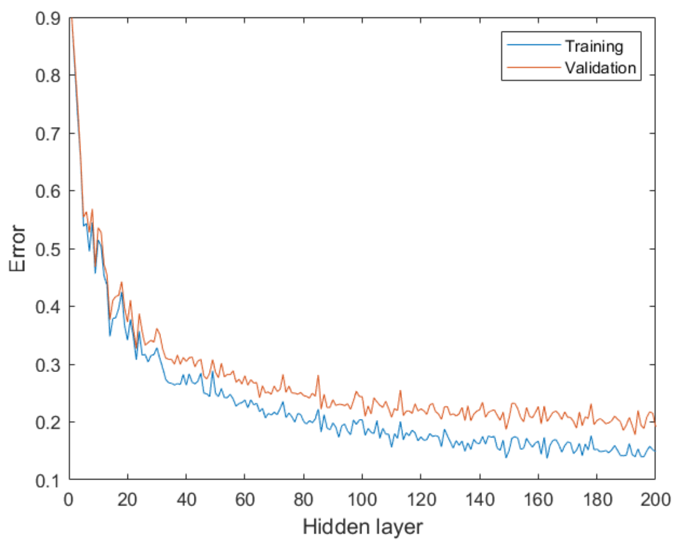


Figure 3: The line chart when changing the structure

hidden units increased after 10^5 iterations. When the number of hidden units increase to 193, we can find the smallest validation error 0.178, with the test error 0.196.

So I choose 193 as the number of hidden units in later experience.

2. Change the training procedure

In this part, I change the training procedure by modifying the sequence of step-sizes or using different step-sizes for different variables. That momentum uses the update

$$\omega^{t+1} = \omega^t - \alpha_t \nabla f(\omega^t) + \beta_t(\omega^t - \omega^{t-1})$$

where α_t is the learning rate (step size) and β_t is the momentum strength. I set β_t as a constant 0.9 and change the learning rate to see the trends of validation error.

In experiencing I change the code in $example_neuralNetwork$**.m** by replacing the

```
      w = w - stepSize*g;
```

with

```
      w_temp = w; % Store the weight of the last iteration
      w = w - stepSize * g + beta * (w - w_pre); % Update the weight
      w_pre = w_temp;
```

From the **Figure 4** below, we can find that the best step size is $1e^{-}4$, with validation error 0.207 and test error 0.193. It seems that the change of training procedure doesn't have significant effect on the result.

However, Momentum method accelerates the training speed and avoids the improper initial weights which makes the network falls into a local minimum during the training process and cannot reach the global optimal state.

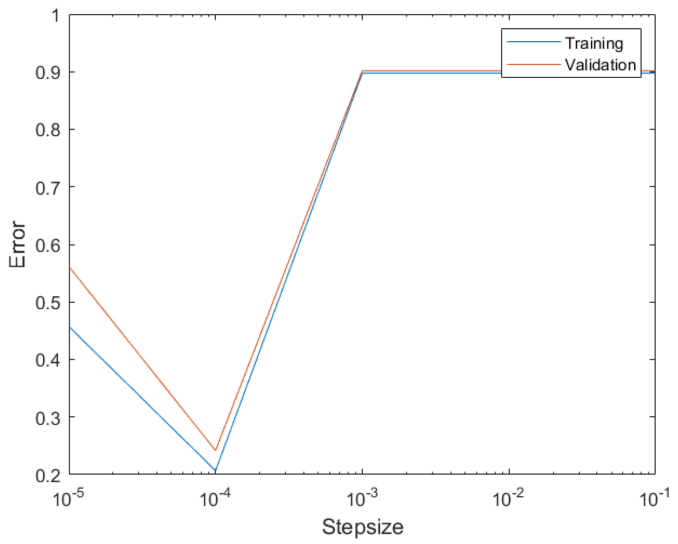


Figure 4: The line chart when changing the the training procedure

3. Vectorize evaluation

In this part, we vectorize evaluation the loss function to accelerate the training speed. From the **MLPclassificationPredict.m**, we can find the network structure of MLP. De-

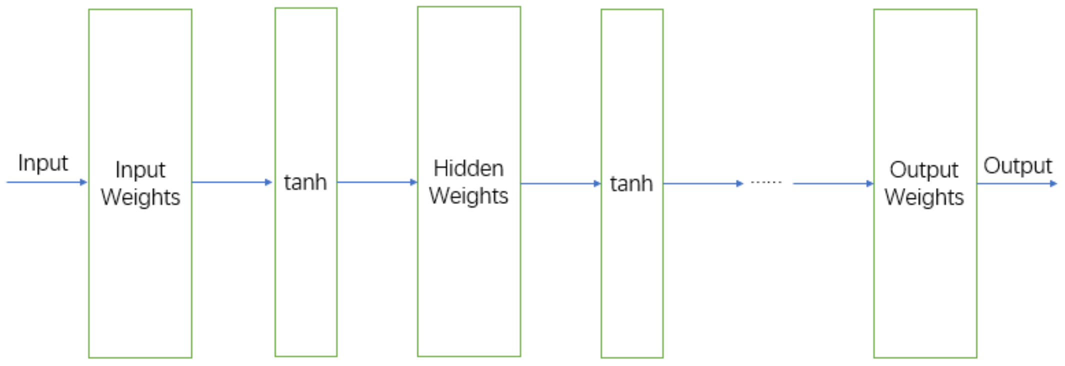


Figure 5: The network structure of MLP

note the weights of **Input Weights** is W_{in}, and the n weights of **Hidden layer** is $W_{1..n}$, **Output weights** is W_{out}. The output of each weights layer (include Input weights, Hidden layer and Output weights) and tanh layer is $IP_{1..n}$, $FP_{1..n}$, and the output $\hat{y}$ and the Loss $= \|y - \hat{y}\|_2^2$. So that we can get

$$
\begin{aligned}
IP_1 &= X \times W_{in} \\
FP_i &= tanh(IP_i), i = 1..n \\
IP_{j+1} &= FP_j \times W_j, j = 1..n-1 \\
\hat{y} &= FP_n \times W_{out} \\
L &= \|y - \hat{y}\|_2^2
\end{aligned}
$$

So that,

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}t} &= \frac{\mathrm{d}L}{\mathrm{d}\hat{y}} \frac{\mathrm{d}\hat{y}}{\mathrm{d}W_{out}} \\
&= FP_h^T \times 2(\hat{y} - y)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}W_i} &= \frac{\mathrm{d}L}{\mathrm{d}\hat{y}} \frac{\mathrm{d}\hat{y}}{\mathrm{d}FP_n} \frac{\mathrm{d}FP_n}{\mathrm{d}IP_n} \frac{\mathrm{d}IP_n}{\mathrm{d}FP_{n-1}} \cdots \frac{\mathrm{d}IP_{i+1}}{\mathrm{d}W_i} \\
&= 2(\hat{y} - y) \times W_{out} \times (1 - IP_n^2) \times W_{n-1} \times ... \times (1 - IP_{i+1}^2) \times FP_i
\end{aligned}
$$

$$
\frac{\mathrm{d}L}{\mathrm{d}W_{in}} = \frac{\mathrm{d}L}{\mathrm{d}IP_1} \frac{\mathrm{d}IP_1}{\mathrm{d}W_{in}}
$$

We can change the code of **MLPclassificationLoss.m** as
(You can see more details in **MLPclassificationLoss_Task3.m**):

```
% Compute Output
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 % Forward
yhat = fp{end}*outputWeights;
relativeErr = yhat - y;
f = sum(relativeErr.^2);
if nargout > 1
    err = 2 * relativeErr;
    % Output Weights
    gOutput = fp{end}'* err;
    if length(nHidden) > 1
        % Last Layer of Hidden Weights
        clear backprop
        backprop = err * (sech(ip{end}).^2 .* outputWeights');
        gHidden{end} = fp{end-1}' * backprop;
        % backprop = sum(backprop,1);
        % Other Hidden Layers
        for h = length(nHidden)-2:-1:1
            backprop = (backprop*hiddenWeights{h+1}').*sech(ip{h+1}).^2;
            gHidden{h} = fp{h}'*backprop;
        end
        % Input Weights
        backprop = (backprop*hiddenWeights{1}').*sech(ip{1}).^2;
        gInput = X'*backprop;
    else
        % Input Weights
        gInput = X'*err*outputWeights'.*(sech(ip{end}).^2);
    end
end
```

By converting the loop into a matrix operation, we effectively reduce the time complexity of the gradient calculation, although this does not have a substantial impact on our results.

4. Weight Decay and Early Stopping

In this part, I add l_2 regularization of the weights to my loss function and stop training when the error on a validation set stops decreasing.
As for l_2 regularization, since the loss function turns to $L = E + \lambda \sum_j w_j^2$, the gradient of each layer is $\frac{\mathrm{d}L}{\mathrm{d}W} = \frac{\mathrm{d}E}{\mathrm{d}W} + 2\lambda W$. So for ease of calculation, we can change the way we update the weight of each layer into:

```
        w = w - stepSize * (g + lambda * w) + beta * (w - w_pre);
```

and search for the best λ for this model. The result is as followed. So when we choose λ = 0.01, we can get the best validation error 0.204 and test error 0.191.
As for Early Stopping, we can store the latest value of validation value and compare it

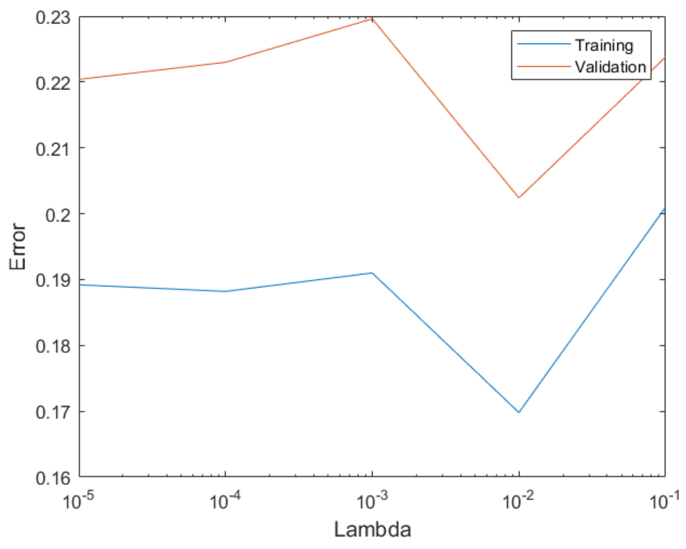


Figure 6: The Weight Decay of MLP

with the current value to decide whether we should stop the training. You can see more detail in my code **example_neuralNetwork_Task4.m**.

5. Add bias

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

First, I put a column filled with 1 in the right of the output matrix and a row at the bottom of each weight matrix so that when we do the matrix multiplication we can add a bias at each layer.

Note that:

1. When doing the back propagation, the last row of each gradient of "ip" is the gradient of bias.
2. When doing the regularization, we should not calculate the bias since it doesn't change the complexity of network.
3. I found that when using the momentum method in training, sometimes it may cause Nan in gradient, so I gave up this method and adjusted the hyper-parameters.

Since the code is too long to be listed in the report, you can see more details in **example_neuralNetwork_Task5.m**, **MLPclassificationLoss_Task5.m** and **MLPclassificationPredict_Task5.m**.

Finally, when Stepsize $= 10^{-3}$, regularzation parameter $\lambda = 10^{-1}$. We can get the optimal solution with validation error = 0.06 and test error = 0.05. The training procedure is shown below.

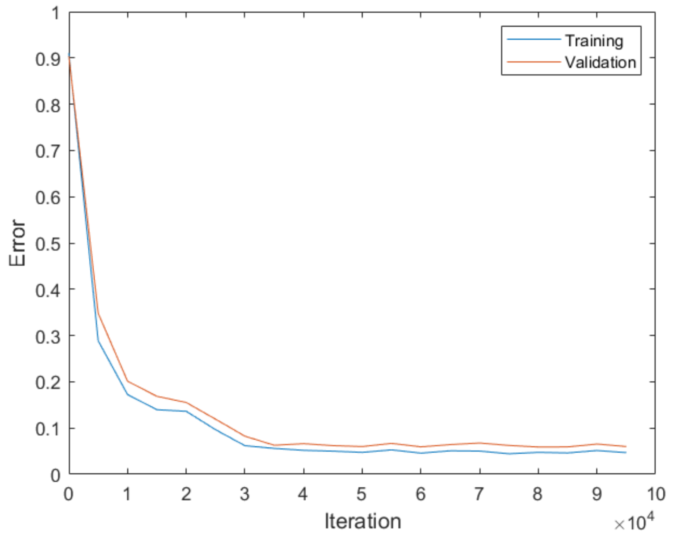


Figure 7: The result of adding bias in MLP

6. Softmax

In this part, I put a softmax layer at the end of the network and use the negative log-likelihood of the ture label to replace the squared error.
Since

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

$$L = -ln(p(y_i))$$

We need to change the way we calculate gradient.

$$\frac{\mathrm{d}L}{\mathrm{d}p(y_i)} = -\frac{1}{p(y_i)}$$

$$\frac{\mathrm{d}p(y_i)}{\mathrm{d}y_j} = \begin{cases} p(y_i)(1-p(y_i)) & i = j \\ -p(y_i)p(y_j) & i \neq j \end{cases}$$

$$\frac{\mathrm{d}L}{\mathrm{d}y_i} = \begin{cases} p(y_i) - 1 & i = j \\ p(y_j) & i \neq j \end{cases}$$

, and i is the true label.

We only need to change the way we calculate the predict error. By

```
    pyhat = exp(yhat) / sum(exp(yhat));
    True = ( y == 1 );
    err = pyhat - True;
```

we can get the validation error 0.09 and test error 0.08. The training procedure is shown below, which is little difference from the previous result.

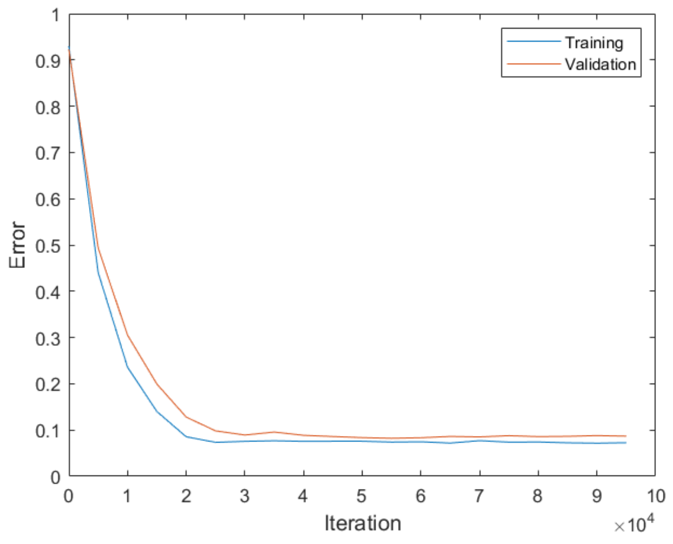


Figure 8: The result of using softmax in MLP

7. Drop out

In this part, in order to prevent the occurrence of over fitting, I delete some nodes during propagation, so that I can reduce the complexity of the network and let it not become over fitting.
There are two parts of drop out method.

1. During propagation, I let the neurons stop working with probability p, in other words, let its value become 0 with probability p.

2. While updating the value of weights, I multiply the weight of the neurons remained after the dropout by 1 / p, which called the penalty term.

```
% Drop out
p = 0.5;
delete = (rand(size(w)) > p);
w(delete) = 0;
% Calculate the gradient
......
% rescale
gradientW = gradientW * 1/(1-p);
gradientW(delete) = 0;
```

After adding Drop out method in network, we get the validation error = 0.13 and test error = 0.12. Although we solved the problem of overfitting, the accuracy of the model also decreased accordingly.

8 Fine-tuning

In this part, when fix the parameters of all the layers, we can regard the parameters of the last layer as a convex optimization problem.
Denote the output of last hidden layer as F_h, and the weights of output layer is W_{out}. The problem can be expressed as:

$$\arg\min_{W_{out}} = \|F_h W_{out} - Y\|_2^2 + \frac{\lambda}{2}\|W\|_2^2$$

and the closed form solution for W_{out} is:

$$W_{out} = (2F_h^T F_h + \lambda)^{-1} 2F_h^T y$$

So, we can store the output of final hidden layer each time we calculate gradient, and solve this convex optimization problem to get the value of output weight.

```
% When need to calculate validation error and test error
% fine tuning
[¬, ¬, fpn] = funObj(w,b,1:5000);
gOutput = (2* (fpn'*fpn) + lambda * eye(size(fpn,2))) \ ...
    (2*fpn'*yExpanded);
w(end - nHidden(end)*nLabels+1:  end) = reshape(gOutput(1:end-1,:), ...
    nHidden(end)*nLabels,1);
b(end-nLabels+1:end) = gOutput(end,:)';
```

By doing this, the best results have validation error 0.04 and test error 0.03, which is a tangible improvement to the model.

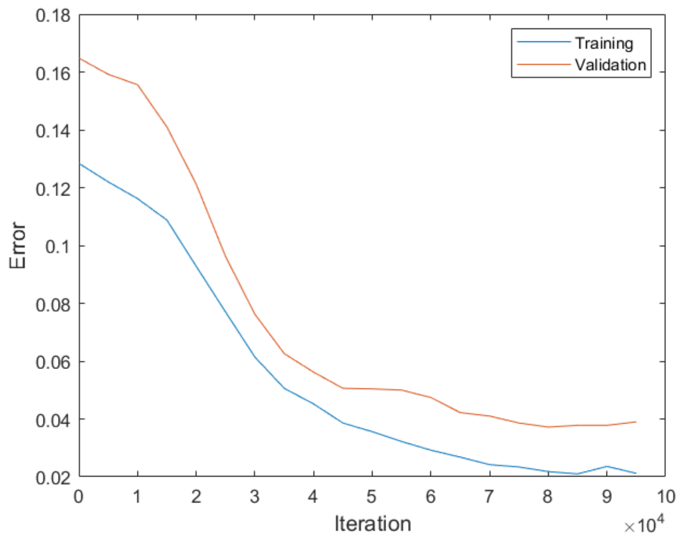
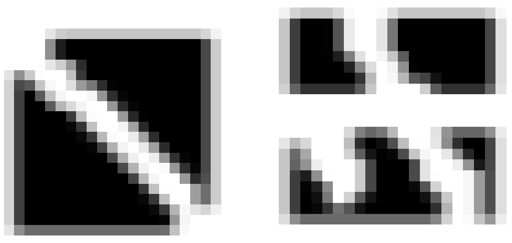


Figure 9: The result of using fine tuning in MLP

9 Transformation

In this part, I artificially create more training examples, by applying small transformations (translations, rotations, resizing, etc.) to the original images. Because resizing and rotation may cause some abnormal data (such as confusing 7 with 1, 8 and 3). So, I only apply rotation method. In the experiment, every input picture has a probability 0.5 to be flipped 180 degrees. The rotated pictures are shown below.

```
    % Transformation
    for i = 1:size(X,1)
    if rand(1) > 0.5
        X(label_count,:) = reshape(flipud(reshape(X(i,:),16,16)),256,1);
        y(label_count,:) = y(i);
        label_count = label_count + 1;
    end
end
```



(a) Number 1 (b) Number 4

Figure 10: Number 1 and 4 after flipud

However, the result of the experiment is not ideal, it may be because the network treats the picture as a sequence rather than a two-dimensional picture, and the information of the sequence itself is disrupted when the picture is rotated, so the accuracy decreased.

10 2D convolutional layer

In this part, I reshape the USPS images back to their original 16 by 16 format, and Replace the first layer of the network with a 2D convolutional layer.

10.1 Network structure

The structure of CNN network is shown below.

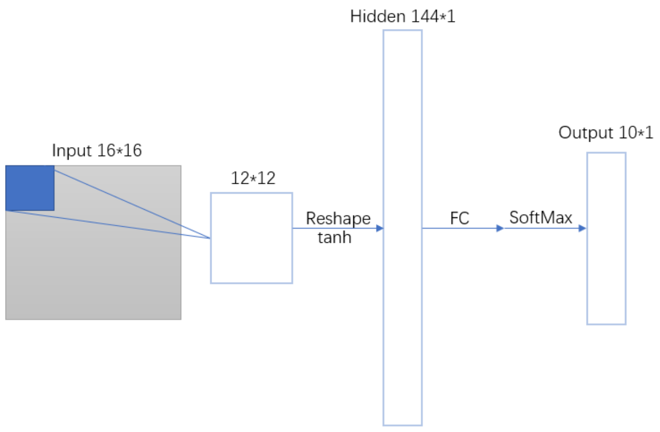


Figure 11: CNN network structure

Thought the convolution by a 5*5 kernel, the picture will be transfer into a 12*12 matrix. After go through a tanh activation function, stretched into a vector by a fully connection layer and a softmax layer,we can get the probability whether the picture belongs to a certain category.
So the parameter we need to optimize in this CNN network is a 5*5 kernel, a 144*10 weights of output layer and a bias of 10*1 vector.

10.2 Backward propagation

Since we already have the gradient of $\frac{\mathrm{d}L}{\mathrm{d}IP_1}$ while vectorize evaluation, so if we can get the gradient of $\frac{\mathrm{d}IP_1}{\mathrm{d}W_{CNN}}$, we can calculate the gradient of kernel.

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}w_{uv}} &= \frac{\mathrm{d}L}{\mathrm{d}IP}\frac{\mathrm{d}IP}{\mathrm{d}w_{uv}} \\ &= \sum_{i=1}^{M-U+1}\sum_{j=1}^{N-V+1} x_{i+u-1,j+v-1}\frac{\mathrm{d}L}{\mathrm{d}IP_{ij}} \\ &= rot180(X)\otimes\frac{\mathrm{d}L}{\mathrm{d}IP}\end{aligned}$$

and rot180 means rotate 180 degrees, $\otimes$ means convolution operation.

10.3 Result

Through the method mentioned above, we can build a CNN network with validation error 0.18 and test error 0.17. When I adjusted the size of the filter and other hyper-parameters, the performance of the network did not improve significantly. It may be because the shallower convolution network cannot learn the semantic information in the picture, and the receptive field is smaller, so the accuracy is not good. You can find more details in my code.

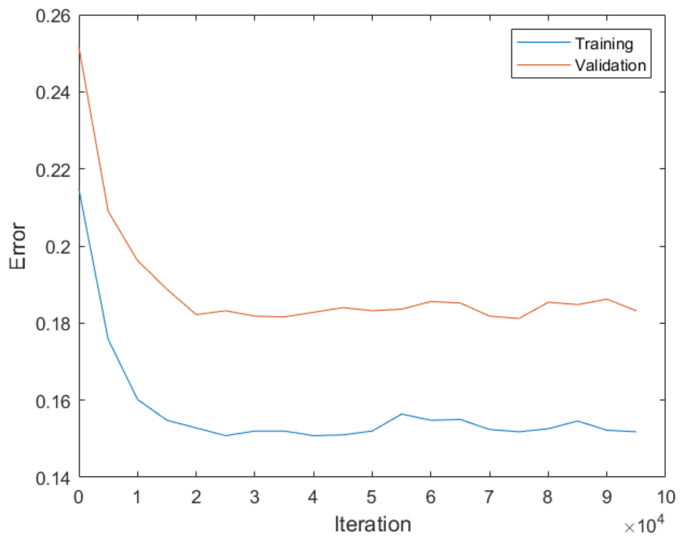


Figure 12: The result of CNN

11 Conclusion

Through this project, I try to incorporate different features into my neural network to improve performance. Here are all the results:

Table 1: Result

Features	Validation Error	Test Error
Baseline	0.18	0.20
Momentum	0.21	0.19
Vectorize		
Weight Decay	0.20	0.19
Bias	0.06	0.05
SoftMax	0.09	0.08
Dropout	0.13	0.12
Fine-tuning	0.04	0.03
Transformation	≥ 0.5	≥ 0.5
CNN	0.18	0.17

and the best network structure and hyper-parameter is:

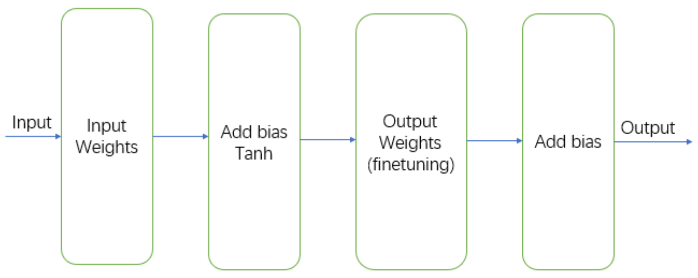


Figure 13: Best practice

Table 2: Best Hyper-parameters

Depth	Neurons	Stepsize	Regularization	Iteration	Optimizer	Loss
1	193	10^{-3}	$0.1(L_2$ Regular)	10^5	BGD	Square Loss

And it can reach the validation error 0.04 and test error 0.03, which is an ideal accuracy rate.

12 Further Improvement

For further experiments, the following ideas may help us to improve the accuracy of model.

1. We can try more optimizer, e.g. Nesterov, Adam
2. We can use a deeper convolutional network to extract the semantic information of the picture, which may help while prediction.
3. For multi-classification problems, it may be more appropriate to use cross-entropy as a loss function.
4. We can increase the resolution of the image through linear interpolation, so that the difference between the images in different categories can become much greater.