# SLML Project_3 Neural Network

Shun Zhang, student ID:15300180012

December 10, 2017

In this problem we will investigate handwritten digit classification. The inputs are 16 by 16 grayscale images of handwritten digits (0 to 9), and the goal is to predict the number of the given the image. We'll train a neural network with stochastic gradient descent, printing the validation set error as it goes. To handle the 10 classes, there are 10 output units (using a {-1,1} encoding of each of the ten labels) and the squared error is used during training. Our goal is to modify this training procedure architecture to optimize performance.

During this optimization, we will first go through the ten suggestions given by the professor **separately** and discuss which one or which ones are good to our goal. After that, we will do some **combination** work to find out something like whether $'A + B > A'$. Finally, we will get an optimized version of the original network, whose performance is the best we can find.

(More information about the supporting Matlab codes please refer to *readme.txt*)

## 1 Apply the ten methods separately

### 1.1 Network structure

First, we run $example\_neuralNetwork$, with one single hidden layer including 10 units, several times and find the initial average performance is about **52.4%**.

#### 1.1.1 single hidden layer

**Attention:**

- the codes give out the test error, while all the performance here refer to the accurate percentage, which is $\boldsymbol{1 - error}$.
- the method we use is *stochastic gradient descent* so one single result may contain some uncertainty, which is the reason why we should run it several times and take the *average performance*. What's more, through the following report, we'll do the same procedure with the notation of **A.performance**.

We then try different amount of units within one single hidden layer and find out the following results.

| **Number of units** | 5 | **10** | 20 | 50 | 100 | **200** | 300 | 500 |
|---|---|---|---|---|---|---|---|---|
| **A.performance** | 33.8% | **52.4%** | 60.7% | 70.8% | 76.6% | **81.1%** | 81.2% | 82.1% |

We may find that in the case of single hidden layer, the more units we use, the better performance we can achieve, while when the number of units is bigger than 200, the performance grows a little after a great increase in the number of units.

### 1.1.2 multiple hidden layers

To put it simply, we assume that the number of units in each hidden layer is the same, which is 10 units in this case, so we have the following results.

Table 1: performances of different number of hidden layers

| **Number of hidden layers** | 1 | 2 | 3 |
|---|---|---|---|
| **A.performance** | 53.1% | 44.6% | 38.3% |

Till here, we find that with the number of hidden layers grow, the performance ‘decreases’. But is that true really? Let’s have some more experiments with **more iterations**.

Table 2: performances of different number of hidden layers(more iterations)

| **Number of hidden layers** | 1 | 2 | 3 |
|---|---|---|---|
| **A.performance** | 77.2% | 72.0% | 67.5% |

Through Table 1 and 2, we can conclude that with more iterations, the performance is getting better and also, we can see that multiple hidden layer structure performs worse than that with single hidden layer **under the condition of same iterations**. But which one’s best performance is better? Let’s discuss it later.

**Remark:**

- Since we haven’t come to an exact conclusion of which structure performs best, we will **default the structure as the initial one when applying other methods separately, which is single hidden layer with 10 units**.

## 1.2 Momentum training procedure

Here, we try to change the training procedure by modifying the sequence of step-sizes or using different step-sizes for different variables, which uses the update

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

where $\alpha_t$ is the learning rate (step size) and $\beta_t$ is the momentum strength. (A common value of $\beta_t$ is a constant 0.9.)
Then we find the new performance much better.

| | **initial procedure** | **momentum procedure** |
|---|---|---|
| **A.performance** | 52.7% | 71.5% |

## 1.3 Vectorize evaluating the loss function

In this part, we try to find out the codes which can be re-written into matrix calculation. Simply, we should focus on those *for loops.*

We then open the *MLPclassificationLoss.m* and find there are many *for loops.* However, some of them are loops of *cells* in order to do jobs such as *feed forward* or *back propagation*, which can not be re-written in a matrix way.

Well, there are three exceptions

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

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

```
% Last Layer of Hidden Weights
clear backprop
for c = 1:nLabels
    backprop(c,:) = err(c)*(sech(ip{end}).^2.*outputWeights(:,c)');
    gHidden{end} = gHidden{end} + fp{end-1}'*backprop(c,:);
end
backprop = sum(backprop,1);
```

which can be re-written into the following forms

```
% Output Weights
gOutput = gOutput + fp{end}' * err;
```

```
% Input Weights
gInput = X(i,:)' * (sech(ip{end}).^2.*(err * outputWeights'));
```

```
% Last Layer of Hidden Weights
clear backprop
backprop = (err*outputWeights').*sech(ip{end}).^2;
gHidden{end} = gHidden{end} + fp{end-1}'*backprop;
```

Finally, we find that, compared to the initial case with single hidden layer of 10 units, the re-written version runs much more quickly.

| | initial version | vectorized version |
|---|---|---|
| 10units × 1 hidden layer | 15.60s | **6.85s** |
| 10units × 2 hidden layers | 36.21s | **25.82s** |

**Remark:**

- Since vectorization do not affect the model, we will apply it to all of the following experiments for the sake of shorter running time.

## 1.4 Regularization and early stopping

### 1.4.1 $\mathcal{L}_2$−regularization

Denote the weights as $\omega = (\omega_1, \omega_2, \cdots)$ and the objective function as $J(\omega)$ so we have

$$J_{\mathcal{L}_2}(\omega) = J(\omega) + \frac{\lambda}{2}\omega^T\omega$$

$$\frac{dJ_{\mathcal{L}_2}(\omega)}{d\omega} = \frac{dJ(\omega)}{d\omega} + \lambda\omega$$

where $\frac{dJ(\omega)}{d\omega}$ is the initial gradient and $\frac{dJ_{\mathcal{L}_2}(\omega)}{d\omega}$ is the new one. So what we should do is only to change the training procedure from

$$\omega^{t+1} = \omega^t - \eta \frac{dJ(\omega)}{d\omega^t}$$

to

$$\omega^{t+1} = \omega^t - \eta(\frac{dJ(\omega)}{d\omega^t} + \lambda\omega^t)$$

where $\eta$ is the learning rate.

Then we can have different results according to different penalty($\lambda$) in Table 3.

Table 3: different results according to different penalty($\lambda$)

| $\lambda$ | 0 | 0.001 | 0.01 | **0.1** | 1.0 | 10 |
|---|---|---|---|---|---|---|
| **A,performance** | 52.6% | 54.0% | 68.9% | **89.2%** | 76.3% | 9.5% |

From the results above we can see that the $\mathcal{L}_2-$regularization with penalty parameter $\lambda = 0.1$ performs the best, which is a lot better than our initial performance with no such regularization.

**Remarks:**

- it's quite sure that proper penalization does enhance the performance
- it's not so sure to say which $\lambda$ works best and we will discuss this when doing combination work

### 1.4.2 $\mathcal{L}_1-$regularization

Using the same notations as section 1.4.1 and we have

$$J_{\mathcal{L}_1}(\omega) = J(\omega) + \lambda||\omega||_1$$

$$\frac{dJ_{\mathcal{L}_1}(\omega)}{d\omega} = \frac{dJ(\omega)}{d\omega} + \lambda\text{sign}(\omega)$$

where $||\omega||_1 = \sum_{i=1}^{n} |\omega_i|$ and the function 'sign()' is

$$\text{sign}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

Then we can have corresponding results to different penalty($\lambda$).(see the top of next page)

| $\lambda$ | 0 | 0.001 | 0.01 | 0.1 |
|---|---|---|---|---|
| **A.performance** | 53.0% | 49.2% | 61.3% | 24.7% |

We find that even the best performance of $\mathcal{L}_1-$regularization is just a bit better than the initial one and of cause is much worse than that with $\mathcal{L}_2-$regularization. So here we only consider applying the $\mathcal{L}_2-$regularization.

### 1.4.3 Early stopping

Early stopping is a simple but efficient technique when applying deep neural network and you want to know whether the performance keeps growing. Whenever the performance on validation set stops growing, it stops the training procedure and test it with the last recorded parameters. And this can be realized by the following codes:

```
% Validation error
    if mod(iter-1,round(maxIter/20)) == 0
        yhat = MLPclassificationPredict(w/2,Xvalid,nHidden,nLabels);
        error_v = sum(yhat~=yvalid)/t;
        fprintf('Training iteration = %d, validation error = %f\n',iter-1,
            error_v);

        if iter < 2
            elast = error_v;
            wlast = w;
        elseif elast < error_v
            w = wlast;
            break;
        else
            elast = error_v;
            wlast = w;
        end
    end
```

However, note that the main method we use here is *stochastic gradient descent*, so it's quite reasonable to find that the performance on validation set decreases when the number of iterations between each time you examine the performance on validation set is small. Besides, there are other versions of early stopping, such as stop when the $(n+k)^{th}$ validation error is bigger than the $n^{th}$ one.

In a word, if we apply a large iteration number in training procedure then we will consider doing early stopping. Or, an alternative choice is to early stop after a certain portion of iterations, such as half of the iterations.

### 1.5 Use softmax layer as output layer

According to *MLPclassificationLoss.m*, we know that the original output layer is simply a linear one, which is

$$\widehat{y}_i = \sum_{j=1}^{m_{l-1}} W^o_{ij} z_j^{(l-1)}$$

$$z_j^{(l-1)} = \tanh\Bigg( \sum_{k=1}^{m_{l-2}} W_{jk}^{(l-1)} z_k^{(l-2)} \Bigg)$$

**Remarks:**

- $W^o_{ij}$ is the weight between the $j^{th}$ unit of the last hidden layer (the $(l-1)^{th}$ layer) and the $i^{th}$ unit of the output layer
- $W_{jk}^{(l-1)}$ is the weight between the $k^{th}$ unit of the $(l-2)^{th}$ layer and the $j^{th}$ unit of the $(l-1)^{th}$ layer
- $z_j^{(l-1)}$ is the activated value of the $j^{th}$ unit of the last hidden layer (the $(l-1)^{th}$ layer)
- $m_l$ is the number of the units of the $(l)^{th}$ layer
- here 'tanh' is our activation function
- the notations throughout this report will all be consistent to these ones

Instead of doing so, we 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. Here the softmax function is

$$p(\widehat{y}_i) = \frac{e^{\widehat{y}_i}}{\sum_{j=1}^{10} e^{\widehat{y}_j}}$$

and this can be interpreted as the activation function of the output layer.

After applying this, we can adapt the objective function from squared error to the negative log-likelihood of the true label under this loss:

$$J(W) = -\text{log} p(\widehat{y}_i)$$

where the true label is $i$.

Then we move on to update the gradients and because of the property of *back propagation* method, we only need to update the error term on the output layer.

- squared error version

$$\delta_i^o = \frac{\partial}{\partial \widehat{y_i}} J(W) = \frac{\partial}{\partial \widehat{y_i}} \sum_{j=1}^{10} (y_j - \widehat{y_j})^2 = 2(y_i - \widehat{y_i})$$

- softmax version

$$\delta_i^o = \frac{\partial}{\partial \widehat{y_i}} J(W) = \frac{\partial}{\partial \widehat{y_i}} - \log p(\widehat{y_k}) = -\frac{1}{p(\widehat{y_k})} p(\widehat{y_k})(\delta_{ik} - p(\widehat{y_i})) = -(\delta_{ik} - p(\widehat{y_i}))$$

where

$$\delta_{ik} = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

So here, we **re-encode the true label** using {0,1} instead of {-1,1} and the error term can be re-written into

$$\delta_i = -(y_i - p(\widehat{y_i}))$$

Then we find the comparative result is better.

| | squared error version | softmax version |
|---|---|---|
| A.performance | 52.4% | 64.5% |

## 1.6 Make one of the hidden units in each layer a constant

Note that the original hidden layer structure is full-connected. Recall that in linear regression, adding a bias can do a great job in performance. And here it's quite easy to do so by using a 'stupid' method: set the last unit of a hidden layer to 1 (or any constant) after the unit is activated. That is to say, all we should do is to make sure that one of the units in each layer is a constant when feeding forward to the next layer. Then we find the results as shown in Table 4.

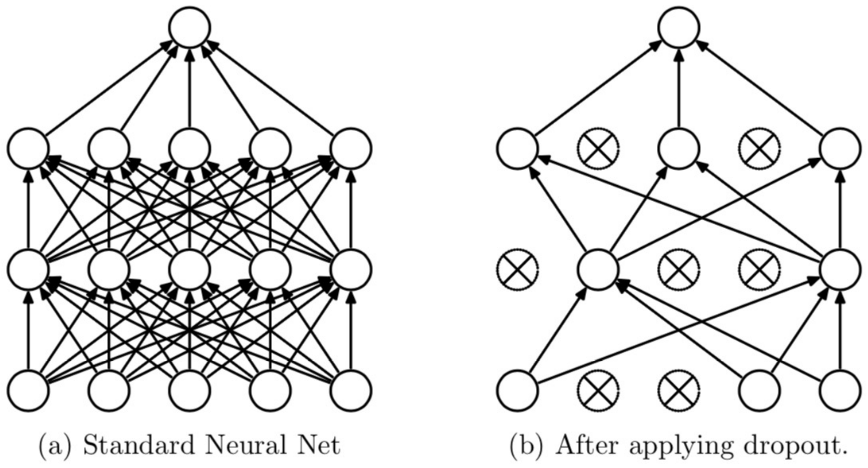
Table 4: performance with & without bias

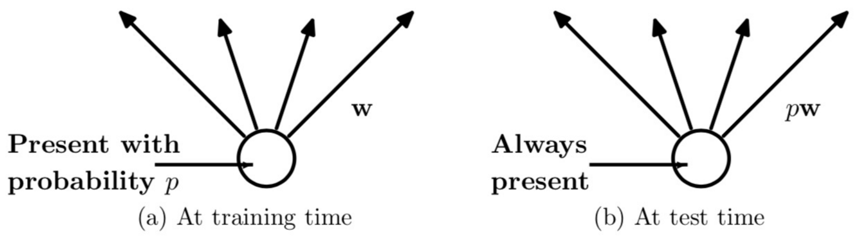
| | without bias | add a bias |
|---|---|---|
| A.performance | 52.4% | 56.5% |

The reason why the increase in performance is not significant may be that by just making one as a bias, the linear units is one fewer than the initial model. More details will be discussed later.

### 1.7 Implement 'dropout'

When dealing with deep neural network, 'dropout' is a commonly used technique to fight against overfitting. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. (More details in [1]) And here are two impressive figures from [1].





Recall the notations in section 1.5, the feed-forward operation can be described as:

$$z_i^{(l)} = \tanh\Big(\sum_{k=1}^{m_{l-1}} W_{ik}^{(l)} z_k^{(l-1)}\Big)$$

where ‘tanh’ is the activation function.
With dropout, the new feed-forward operation becomes:

$$\begin{aligned}
r_j^{(l-1)} &\sim \text{Bernoulli}(p), \\
\tilde{z}_j^{(l-1)} &= r_j^{(l-1)} * z_j^{(l-1)}, \quad \forall 1 \leq j \leq m_{l-1} \\
z_i^{(l)} &= \tanh\left( \sum_{k=1}^{m_{l-1}} W_{ik}^{(l)} \tilde{z}_k^{(l-1)} \right)
\end{aligned}$$

And here we simply take $p$ as 0.5, which is the common choice.
Then the result turns out to be:

| | no dropout | dropout |
|---|---|---|
| A.performance | 52.4% | 47.5% |

The model with dropout performs worse than the initial one. This is quite reasonable in that the dropout technique is intended to solve the problem of overfitting, while the initial model with single hidden layer, including only 10 units, is far from overfitting, so the dropout even increases the problem of under-fitting. Considering this, we apply a more complicated model with 100 units in a single hidden layer with different number of iterations.

Table 5: performances with different number of iterations

| Number of iterations | $10^5$ | $10^6$ | $10^7$ |
|---|---|---|---|
| A.performance without dropout | 76.7% | 84.4% | 86.5% |
| A.performance with dropout | 76.1% | 87.7% | 92.1% |

From Table 4, we could see that as the number of iterations grows, the performance with dropout becomes increasingly better than that of the initial one, which is consistent to our analysis before.

## 1.8 Do ‘fine-tuning’ of the last layer

‘Fine-tuning’ is always applied to solve the problem that one wants to train a deep neural network with only limited training samples. In such situation, he then can adapt other’s well-trained network to his training samples by applying ‘fine-tuning’ technique.

Here we can fix the parameters of all the layers except the last one (the output layer), and solve for the parameters of the last layer exactly as a convex optimization problem. A particularly fast choice is the linear regression with the squared error (also called the Least Squares).

You can think of this as 'adding another layer after the original output layer', of which the parameters are determined by Least Squares instead of back-propagation.

Recall that in section 1.5, $\widehat{y}_i$ is the $i^{th}$ output. Here we denote the $j^{th}$ value of the 'new layer' as $y_j^*$ and denote the weight between $\widehat{y}_i$ and $y_j^*$ is $W_{ji}^*$.

So we have a linear regression problem, which is

$$y_j^* = \sum_{i=1}^{10} W_{ji}^* \widehat{y}_i$$

$$E_j = \frac{1}{2}(y_j^* - \widehat{y}_j)^2$$

Recall that if we have $n$ training samples, and we denote the original output of the $k^{th}$ training sample as $\widehat{\boldsymbol{y}}^{(k)}$. So we have $\boldsymbol{\Phi} = (\widehat{\boldsymbol{y}}^{(1)}, \widehat{\boldsymbol{y}}^{(2)}, \cdots, \widehat{\boldsymbol{y}}^{(n)})^T$

$$E(\boldsymbol{W}_i^*) = \frac{1}{2}(\boldsymbol{\Phi}\boldsymbol{W}_i^* - \boldsymbol{y}_i)^T(\boldsymbol{\Phi}\boldsymbol{W}_i^* - \boldsymbol{y}_i)$$

$$\frac{\partial}{\partial \boldsymbol{W}_i^*} E(\boldsymbol{W}_i^*) = \boldsymbol{\Phi}^T(\boldsymbol{\Phi}\boldsymbol{W}_i^* - \boldsymbol{\Phi}^T \boldsymbol{y}_i) = 0$$

$$\boldsymbol{W}_i^* = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T \boldsymbol{y}_i$$

so we have

$$\boldsymbol{W}^* = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T \boldsymbol{Y}$$

where $\boldsymbol{W}^* = (\boldsymbol{W}_1^*, \boldsymbol{W}_2^*, \cdots, \boldsymbol{W}_{10}^*)$ and $\boldsymbol{W}_i^* = (\boldsymbol{W}_{i1}^*, \boldsymbol{W}_{i2}^*, \cdots, \boldsymbol{W}_{i10}^*)^T$ ; $\boldsymbol{Y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_{10})$ and $\boldsymbol{y}_i$ is the column vector of all the $i^{th}$ labels of the $n$ training samples.

So we compare the results between fine-tuning after training and the original one without fine-tuning as follows.

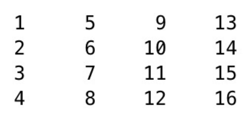
| (10 units) | original training | fine-tuning after training |
|---|---|---|
| performance | 54.0% | 56.3% |
| (100 units) | original training | fine-tuning after training |
| performance | 79.0% | 80.5% |

From this we can see that fine-tuning of Least Squares enhances the performance a little.

## 1.9 Create more training samples

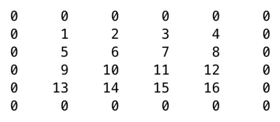
### 1.9.1 Create by applying some small transformations

- applying small translations Suppose we have a $16 \times 16$ matrix of training sample, we apply some small translations by doing the following steps: (take $4 \times 4$ matrix as an



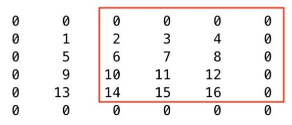
```
1   5   9   13
2   6   10  14
3   7   11  15
4   8   12  16
```

the original matrix



```
0   0   0   0   0   0
0   1   2   3   4   0
0   5   6   7   8   0
0   9   10  11  12  0
0   13  14  15  16  0
0   0   0   0   0   0
```

padding zeros



```
0   0   0   0   0   0
0   1   2   3   4   0
0   5   6   7   8   0
0   9   10  11  12  0
0   13  14  15  16  0
0   0   0   0   0   0
```

choose a 'new' matrix

example)
After the three steps above we can get a translated matrix different from the original one. Here is an example (move right 2, down 1):

Note that through this method we can get whatever translations you want by padding enough zeros and making different choices.

- applying small rotations

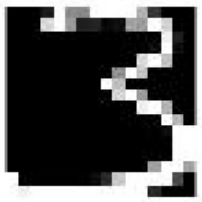
  Again, suppose we have a 16 × 16 matrix of training sample, we apply some small rotations by mainly use the build-in function *imrotate*:

```
% X will be rotated slightly by random degrees from 0 to 5
% also random with clockwise or anti-clockwise

theta = rand(1)*10 - 5;     % get a random degree
X = imrotate(X, theta);     % rotate X with degree theta
ss = size(X);
X = imresize(X,16/ss(1));   % resize the rotated matrix into size
    (16,16)
```

Here is an example(see next page):

Also note that we can do whatever rotation you want by setting different $\theta$.


the original figure

5 degree clockwise

30 degree clockwise

Finally, we choose to randomly translate by step 0 to 2 for all four directions and also randomly rotate by degree 0 to 5 clockwise or anti-clockwise. We then create 100000 more training samples, totally 105000 training samples, which are saved in *digits_more.mat*.

**Remarks:**

- recall that we also use the *stochastic gradient descent* method
- by doing so, we can create a training set which is much much much bigger than the given one because with only the translation we can get one that is 36 times larger and with the rotation, we can get countless samples (because the real number is countless)

### 1.9.2 Find more training samples

Here, we find another database of hand-written digits classification called 'semeion', including 1593 labeled samples of hand-written digits. What's more, the 'semeion' samples are also $16 \times 16$ pixels with the same {-1,1} encoding of the 10 labels. So, wow! Why not just add these samples to our training procedure?

### 1.9.3 Training with compact samples(Antagonism neural network)

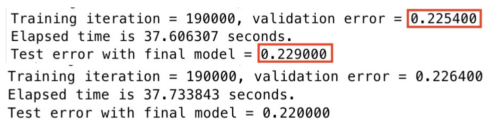
Well, this is a little bit different from just training with more samples we created. This method is to train an original sample and we get its $\widehat{y}$ then we use $\widehat{y}$ as the true target to train a sample which is generated with a slight transformation from the original sample. This method 'teaches' the network to be more robust when dealing with 'similar' samples. This can be done by the following codes:

```
% stpchastic gradient descent
    i = ceil(rand*n);
    [~,g,yhat] = funObj(w,X(i,:),yExpanded(i,:));
    w = w - stepSize*g;
    [~,g1,~] = funObj(w,compactsample(X(i,:)),yhat);
    w = w - stepSize2*g1;
```

And the performance on $100 units \times$ 1 hidden layer is better.

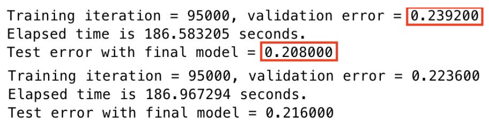
| | initial version ($2\times10^5$ iterations) | compact version ($10^5$ iterations) |
|---|---|---|
| **A.performance** | 77.5% | 78.9% |

If you see the performance on validation set and the testing set:



```
Training iteration = 190000, validation error = 0.225400
Elapsed time is 37.606307 seconds.
Test error with final model = 0.229000
Training iteration = 190000, validation error = 0.226400
Elapsed time is 37.733843 seconds.
Test error with final model = 0.220000
```

Figure 1: initial version



```
Training iteration = 95000, validation error = 0.239200
Elapsed time is 186.583205 seconds.
Test error with final model = 0.208000
Training iteration = 95000, validation error = 0.223600
Elapsed time is 186.967294 seconds.
Test error with final model = 0.216000
```

Figure 2: compact version

So you can see that the compact version's generalization is better and thus more robust. The only disadvantage is that the slight transformation do take some time(as you can see the initial version needs 37s to take 200000 interations while the compact version needs 186s to take just 100000 iterations).

### 1.10 Replace the first layer of the network with a 2D convolutional layer

Recall that the initial first layer, which is the input layer, is just a full-connected layer with the first hidden layer. Here we interpret 'replace' as that we replace the initial input directly from the training set by the *output* of the 2D convolutional layer, whose input is directly from the training set.

A common choice of the size of the filters is $5 \times 5$, by which the output of the 2D convolutional layer is the matrix of $12 \times 12$.

The main procedure goes like this:

**step** (1) randomly generate the initial kernel of size $5 \times 5$ from (0,1) interval, which we denoted as $\boldsymbol{K}$

**step** (2) randomly draw an input of $16 \times 16$ matrix from the training set, which we denote as $\boldsymbol{In}$

**step** (3) do convolution (in Matlab) by

$$\boldsymbol{In} * \boldsymbol{K} = conv2(\boldsymbol{In}, \boldsymbol{K}_{t180}, \text{'valid'})$$

where $\boldsymbol{In} * \boldsymbol{K}$ is the output matrix of the 2D convolutional layer and we simply denote it as $\boldsymbol{P}$ and here, $\boldsymbol{K}_{t180}$ is a rotation of $\boldsymbol{K}$ by 180 degrees because of the function $conv2$

**step** (4) take the activated version of the output matrix $\boldsymbol{P}$, with the same activation function *tanh*, as the input to the first hidden layer and then do the feed-forward process. To explain this more explicitly, we can think it as follows:

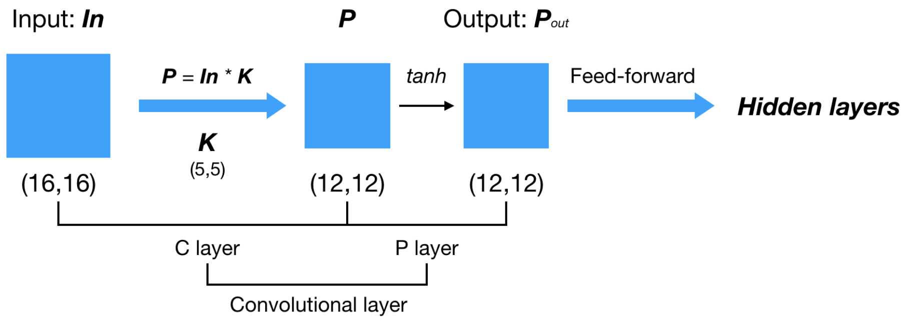


Figure 3: an explicit procedure of a convolutional layer

So, from Figure 3, you can think of a convolutional layer as 'two' layers.

**step** (5) also do the back-propagation until the P layer, then we have the error term of this layer, which we denote as $\boldsymbol{\delta}^P$

**step** (6) then we update the kernel $\boldsymbol{K}$ by gradient descent and according to the property of *sharing weights*, the gradient can be calculated by another convolution (in Matlab)

$$\boldsymbol{g} = conv2(\boldsymbol{In},\ \text{reshape}(\boldsymbol{\delta}^P(2:\text{end}), 12, 12)_{t180},\ \text{'valid'})$$

where $\boldsymbol{\delta}^P$ is a vector of length 145 (with one bias), so reshape$(\boldsymbol{\delta}^P(2:\text{end}), 12, 12)$ is a matrix of size (12,12) and thus $\boldsymbol{g}$ is a matrix of size (5,5), which is just the gradient matrix of kernel $\boldsymbol{K}$.

**step** (7) the updated kernel should be

$$\boldsymbol{K} = \boldsymbol{K} - \eta_{conv} \times \boldsymbol{g}$$

where $\eta_{conv}$ is the learning rate of the kernel.

Repeat step (2) to (7), then we get the procedure of training a network with a 2D convolutional layer by stochastic gradient descent.

And the convolutional version performs better(see the top of next page):

| | initial version | convolutional version |
|---|---|---|
| **A.performance** | 53.9% | 65.7% |

Table 6: performances with 10 units & $10^5$ iterations

| **Different combinations** | **A.performance** |
|---|---|
| original | 52.5% |
| momentum | 77.9% |
| $\mathcal{L}_2-$ regularization($\lambda = 0.1$) | 88.9% |
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) | 90.9% |

## 2 Combination work

First, let's take a look at Table 6. We can see that, with the same structure (single hidden layer with 10 units) and the same iterations, the best performance of the combination of momentum and $\mathcal{L}_2-$ regularization is better.

Through Table 7, you can see that with the combination of softmax and bias, our performance grows higher.

Table 7: performances with 10 units & $10^5$ iterations

| **Different combinations** | **A.performance** |
|---|---|
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) | 90.9% |
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax | 91.6% |
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias | 91.7% |

One thing should be noticed is that the best $\lambda$ for regularization changes when we combine regularization with momentum. It can be partly explained from the momentum training procedure

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

Note that when $\omega^t$ is bigger than $\omega^{t-1}$, then $\omega^{t+1}$ is bigger than that of ordinary gradient descent procedure, which makes $\omega^t$ 'easier' to be regularized. Thus, the required $\lambda$ is smaller.

However, as we have discussed in section 1.6, 'adding' a bias by making one of the units a bias requires a more complicated network to prove itself. So is 'dropout'.

Considering this, we apply our network on more complicated network with 100 units in the single hidden layer.

Table 8: performances with 100 units & $10^5$ iterations

| **Different combinations** | **A.performance** |
|---|---|
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax | 95.0% |
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias | 95.3% |
| momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias + dropout | 95.1% |

From Table 8, what we are sure is that momentum + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax do optimize the performance.

Then let's move on to 200 units (single layer) and $10^6$ iterations! And this time, we will also change the learning rate (step-size).

Table 9: performances with **200** units & $\mathbf{10^6}$ iterations

| **Different combinations** | **performance** |
|---|---|
| momentum(learning rate = 5e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) | **96.8**% |
| momentum(learning rate = 1e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) | **97.1**% |
| momentum(learning rate = 5e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax | 95.6% |
| momentum(learning rate = 1e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax | 95.5% |
| momentum(learning rate = 5e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias | 95.0% |
| momentum(learning rate = 1e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias | 95.1% |
| momentum(learning rate = 5e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias + dropout | 94.9% |
| momentum(learning rate = 1e-4) + $\mathcal{L}_2-$ regularization($\lambda = 0.005$) + softmax + bias + dropout | 94.7% |

Till here, we find that the simple combination of 'momentum + $\mathcal{L}_2-$ regularization' performs the best over others. So we come to our conclusion in the next section.

**Remarks:**

- when training with those samples we create in section 1.8.1 or more samples from *semeion* database, the results are no better than that of the original training samples
- because of the independent learning rate of the kernel and its regularization parameter of the 2D convolutional layer mentioned in section 1.10, it takes quite a time to get to an optimized model, so here we just combine other methods
- the results after fine-tuning are also no better than that of the original training procedure without it on the condition of 200 units in single hidden layer and $10^6$ iterations.

## 3 Final optimization

Well, you must have noticed that we haven't talked about our optimized *hidden layer structure*. However, after many experiments with different hidden layer structure, we find that single hidden layer with 200 units is good 'enough' (restricted by my knowledge), which is also consistent with our analysis in section 1.1.1.

But before we get to our final optimized model, we want to introduce the other two commonly used techniques for optimization, ***step-size decay*** and ***minibatch***. The former one is to apply a decreasing step-size (learning rate) while training to get more close to the optimization point and the latter one is to send a mini batch of training samples into the gradient descent procedure then take the mean of the batch's gradients as the one to update the weights, which is more stable in decreasing the objective function than stochastic gradient descent. However, unfortunately, the results turn out to be, again, no better (under the same condition as other combinations in Table 9).

**So, here is our final optimized model**:

| **Components** | **Parameters** |
|---|---|
| hidden structure | 200 units $\times$ 1 layer |
| momentum training procedure | $\alpha_t$ = 1e-4, $\beta_t = 0.9$ |
| $\mathcal{L}_2-$ regularization | $\lambda = 0.005$ |
| **Iterations** | $10^6$ |

And here are several performances given by our final model.

| | **1** | **2** | **3** |
|---|---|---|---|
| **Performance** | 97.3% | 97.8% | 97.4% |

Table 10: several performances of final optimized model

From Table 10, we could see that our final optimized model's performance is stable, to a great extend. Meanwhile, the elapsed time is acceptable (within 400s) on my PC. (More detailed running procedure please refer to the Appendix)

## 4 Main references:

[1] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In *Journal of Machine Learning Research 15 (2014) 1929-1958*
[2] http://bubuko.com/infodetail-2365924.html

## 5 Appendix:

Running details of Table 10:

```
>> nn_final
 Training iteration = 0, validation error = 0.917000
 Training iteration = 50000, validation error = 0.200200
 Training iteration = 100000, validation error = 0.173400
 Training iteration = 150000, validation error = 0.161800
 Training iteration = 200000, validation error = 0.140200
 Training iteration = 250000, validation error = 0.118800
 Training iteration = 300000, validation error = 0.095600
 Training iteration = 350000, validation error = 0.073200
 Training iteration = 400000, validation error = 0.061000
 Training iteration = 450000, validation error = 0.052600
 Training iteration = 500000, validation error = 0.045400
 Training iteration = 550000, validation error = 0.040600
 Training iteration = 600000, validation error = 0.034800
 Training iteration = 650000, validation error = 0.032600
 Training iteration = 700000, validation error = 0.030600
 Training iteration = 750000, validation error = 0.030200
 Training iteration = 800000, validation error = 0.030200
 Training iteration = 850000, validation error = 0.027400
 Training iteration = 900000, validation error = 0.027600
 Training iteration = 950000, validation error = 0.026800
 Elapsed time is 362.168296 seconds.
 Test error with final model = 0.027000
>> nn_final
 Training iteration = 0, validation error = 0.929400
 Training iteration = 50000, validation error = 0.183000
 Training iteration = 100000, validation error = 0.174200
 Training iteration = 150000, validation error = 0.159000
 Training iteration = 200000, validation error = 0.148600
 Training iteration = 250000, validation error = 0.124600
 Training iteration = 300000, validation error = 0.092200
 Training iteration = 350000, validation error = 0.071800
 Training iteration = 400000, validation error = 0.059800
 Training iteration = 450000, validation error = 0.050200
 Training iteration = 500000, validation error = 0.040600
 Training iteration = 550000, validation error = 0.036200
 Training iteration = 600000, validation error = 0.032400
 Training iteration = 650000, validation error = 0.029800
 Training iteration = 700000, validation error = 0.027200
 Training iteration = 750000, validation error = 0.026600
 Training iteration = 800000, validation error = 0.026200
 Training iteration = 850000, validation error = 0.025000
 Training iteration = 900000, validation error = 0.023400
 Training iteration = 950000, validation error = 0.024800
 Elapsed time is 352.610436 seconds.
 Test error with final model = 0.022000
>> nn_final
 Training iteration = 0, validation error = 0.896400
 Training iteration = 50000, validation error = 0.209000
 Training iteration = 100000, validation error = 0.178000
 Training iteration = 150000, validation error = 0.166200
 Training iteration = 200000, validation error = 0.145600
 Training iteration = 250000, validation error = 0.125800
 Training iteration = 300000, validation error = 0.112800
 Training iteration = 350000, validation error = 0.081600
 Training iteration = 400000, validation error = 0.068400
 Training iteration = 450000, validation error = 0.057600
 Training iteration = 500000, validation error = 0.045200
 Training iteration = 550000, validation error = 0.044000
 Training iteration = 600000, validation error = 0.037000
 Training iteration = 650000, validation error = 0.032600
 Training iteration = 700000, validation error = 0.033200
 Training iteration = 750000, validation error = 0.031000
 Training iteration = 800000, validation error = 0.030200
 Training iteration = 850000, validation error = 0.027200
 Training iteration = 900000, validation error = 0.028000
 Training iteration = 950000, validation error = 0.027000
 Elapsed time is 368.346478 seconds.
 Test error with final model = 0.026000
```