# Project 1 报告

By 19307110499 李一帆

在此次project中，我将会完成要求中的10个Questions，但我会按照自己优化的习惯，将优化的方法分为7大部分，不同的Question将会被分配到各个部分中，因此完成question的顺序和要求中并不一样，我会在标题中明确这一部分的优化属于哪个question，并尽可能地使每个优化的部分上下之间有逻辑。

每个Question对应的主代码命名为Q+问题号.m，如Question 1对应Q1.m，我会在每个Question报告的开头再次强调，因此无须担心在大量的文件中找不到对应的代码。

此外，为了更好地优化我的神经网络，我还做了一些问题清单上没有的优化，这些部分会用Additional item命名，和上面的question一样分布在优化的各个部分。你可以通过下面的目录来快速查找对应的部分。

最终，我最好的模型的测试集误差率可以降低到0.007！

目录

## 熟悉前馈神经网络结构

使用给出的MATLAB神经网络样例代码，基本熟悉神经网络的结构，为后续优化该神经网络打下基础。

当前的神经网络架构主要分为三部分：输入数据处理、向前传播、向后传播。

在输入数据处理部分，对训练集的X变量进行了标准化，使得输入的各个特征在一开始的权重是相近的，防止由于数据的本身偏态性导致方差更大的维度得到的‘关注’更多，保证网络更好的收敛。在对训练集X做过变换后，要对验证集、测试集做相同的变换。除此之外，为了满足多维**logistic regression**的输出要求，先对训练集的y进行二分标签化。

在向前传播部分，需要预定义隐藏层层数以及每一层的节点数、最大训练轮数等超参数，并使用随机初始化的方式对其初始化（不能使用相同的常数对其进行初始化，否则会因为对称性导致训练无效）。注意到在X输入层为X多加了一维1向量，这是为了在输入层加入bias项，后续会在每一层都加入bias项。在向前传播的过程中，目前每一层采用的激活函数都为tanh函数。

在向后传播部分，由于使用的梯度下降方法为随机梯度下降，因此，每次从n个样本中随机抽取一个样本加入神经网络进行训练。首先使用平方差损失计算每次训练结果的损失，后使用该损失计算每一层神经网络权重的梯度。使用预定义的stepsize更新权重参数。

训练完后，将得到的模型在测试集上测试，得到该模型的测试误差。

目前，初始的神经网络在训练轮数为100000次，stepsize为1e-3的情况下，测试集错误率为0.534000：

```
Test error with final model = 0.534000
```

## Question 1 Change the network structure

当前模型使用的隐藏层数为1，节点单元数为10,。尝试修改超参数，得到更好的结果。

代码详见**Q1.m**。以下测试使用的学习率和最大训练轮数保持为1e-3,100000.

### 修改隐藏层单元数

保持隐藏层数不变，修改节点单元数，观察网络变化。分别尝试节点单元数为16,32,64,128，训练结果（测试集误差）如下：

- 10个神经元（初始）：0.534000
- 16个神经元：0.421000
- 32个神经元：0.321000
- 64个神经元：0.250000
- 128个神经元：0.234000
- 256个神经元：0.186000

从结果可以看到，在一定范围内，测试集的错误率随神经元数量的增加而下降。根据通用近似定理（Kurt Hornik，1991#，这是合理的。当神经元数较少时，增加少量的神经元就可以测试集的准确率带来较大提升；但

当神经元数比较大时，增加神经元对测试集准确率的提升并不多。原因之一是单层的神经元对信息的挖掘能力有限；原因之二是当参数量过大时，模型容易发生过拟合。

且参数量越大，模型训练的所需的时间就越多，因此，选择数量适中的神经元数是必要的。

增加隐藏层数

尽管一个足够大的隐藏层足以近似大多数函数，但“足够大”的隐藏层很有可能效率非常低。从经验上来看，网络越深，对任务的归纳就越好。因此，我们尝试加深网络的层数：

- 两层隐藏层，神经元为（10,10）：0.540000
- 三层隐藏层，神经元为（10,10,10）：0.613000
- 两层隐藏层，神经元为（16,10）：0.50800
- 三层隐藏层，神经元为（16,10,10）：0.619000
- 两层隐藏层，神经元为（64,16）：0.477000

可以从结果看出，单纯加深隐藏层层数效果并不好。监测验证集误差提出猜测：可能是由于深层神经网络需要的训练轮数更多，而现在的网络模型架构较初级，导致模型参数的更新较为低效，后续对架构优化后会再次尝试增加隐藏层数。

深入了解，发现虽然前馈神经网络很早就被广泛使用，但两层神经网络（即一个隐藏层和一个输入层）仍是大部分人的选择（邱锡鹏，2020）。

## 修改代码架构

目前的代码多使用循环，导致运行效率较低，不能使用太多训练轮数。且当隐藏层增加时，只有输入层有bias项，导致模型的泛化能力较低。此外，目前网络使用的激活函数为Tanh函数，而激活函数有多种，不同的激活函数也会对模型的优化带来影响。因此，对模型代码进行优化，尽可能向量化运算过程，以及对多层神经网络添加bias项，此外，尝试不同的激活函数对网络的影响。

### **Additional item 1** 改变激活函数

激活函数最常用的就是Sigmoid型函数和ReLU型函数。

Sigmoid型函数主要有Logistic函数和Tanh函数。比起Logistic函数，Tanh函数具有零中心化的特点，而Logistic函数则会导致偏置偏移问题，所以一般使用在0-1分类问题的输出层。使用Tanh函数前面已经探索过，在此不再赘述。

但由于sigmoid型函数的饱和性，饱和区的导数接近0，再经过层层传递后衰减，容易出现梯度消失问题，导致网络训练困难，一种有效的方法是使用导数比较大的ReLU函数。

ReLU函数具有计算高效、生物学合理性等特点，也能够缓解梯度消失问题，尝试使用ReLU函数代替Tanh函数。代码详见**MLPclassficationLoss_ReLU.m**和MLPclassificationPredict_ReLU.m。

但实际运行后，却得到了验证集误差恒为0.901600的结果。调试发现在模型训练过程中，梯度全部变为NaN。很有可能是因为ReLU函数右侧没有饱和态，导致其输出过大。因此在下一节尝试调节参数的初始方差来改善这个问题。

### **Additional item 2** 基于方差缩放的参数初始化

当神经元的输入连接很多时，它的每个输入权重就应该小一点，以避免神经元的输出过大。

目前网络的初始化使用的是均值为0，方差为1的高斯分布初始化，这是不太合理的，因为初始的神经元经过ReLU激活后，大约只有一半输出为0，因此根据**He**初始化（He et al., 2015）方法，参数$\omega_i^{(l)}$的理想方差为

$$var(\omega_i^{(l)}) = \frac{2}{M_{l-1}}$$

其中$M_{l-1}$为第l-1层的神经元个数。

使用**He**初始化方法对初始高斯分布的权重进行缩放。依旧是使用最大训练轮数为100000次，学习率为1e-3的超参数组合，只使用一层隐藏层，当神经元数为64时，模型结果如下：

```
Training iteration = 90000, validation error = 0.035800
Training iteration = 95000, validation error = 0.036400
Test error with final model = 0.037000
```

当神经元数为128时，模型结果如下：

```
Training iteration = 90000, validation error = 0.027200
Training iteration = 95000, validation error = 0.029600
Test error with final model = 0.026000
```

对比使用**Tanh**函数作为激活函数时的模型测试结果：

- 64个神经元：0.250000
- 128个神经元：0.234000

可以看到模型的准确性大大提升！

以上两个additional items的代码详见**A12.m**。

## Question 3 Vectorization

在loss函数的计算中，使用较多的是低维的循环计算，这对训练速度很不友好，为了在可接受的训练时间内使用更多的参数量和实现更多的训练次数，我们对代码进行向量化。 代码详见**Q3.m**和**MLPclassificationPredict_vec.m**和**MLPclaasificationLoss_vec.m**。

需要向量化的文件有Loss计算函数和Predict计算函数。向量化主要有两个部分：

1. 对于多组数据输入时，一次计算完loss并用于后续梯度计算，而非一组一组数据计算，累加loss和梯度。
2. 对于梯度的反向传播，一次更新完所有label的维度，而非一个一个label更新。

未向量化前，使用单元数为64的单个隐藏层训练100000轮需要：

```
历时 39.503688 秒。
```

向量化后，仍然使用单元数为64的单个隐藏层训练100000轮，时间仅需：

```
历时 13.589161 秒。
```

程序运行速度大大提升！

程序运行提速后，好处主要有两方面，一是可以训练更加多的参数；而是可以增加训练轮数，使参数尽可能逼近最佳值。

不妨来验证一下上述想法。

增加训练参数量：

- 隐藏层单元数为256

```
Training iteration = 90000,  validation error = 0.025000
Training iteration = 95000,  validation error = 0.024800
Test error with final model = 0.024000
历时 50.683499 秒。
```

- 隐藏层单元数为512：

```
Training iteration = 90000,  validation error = 0.023800
Training iteration = 95000,  validation error = 0.025600
Test error with final model = 0.023000
历时 289.744307 秒。
```

可以看到，虽然参数量我们可以调至很大，训练时间也在可以接受的范围内，但是在这个简单的神经网络上，一味地增大训练参数量带来的提升效果已经微乎其微，应该从其他方面入手，寻找优化途径。

增加训练轮数：（以神经元数为64为例）

- 训练轮数为100000轮：

```
Training iteration = 90000,  validation error = 0.033400
Training iteration = 95000,  validation error = 0.036200
Test error with final model = 0.037000
历时 14.854921 秒。
```

- 训练轮数为200000轮:

```
Training iteration = 180000, validation error = 0.036800
Training iteration = 190000, validation error = 0.037600
Test error with final model = 0.032000
历时 27.091316 秒。
```

- 训练轮数为300000轮：

```
Training iteration = 270000, validation error = 0.036200
Training iteration = 285000, validation error = 0.038200
Test error with final model = 0.032000
历时 41.253428 秒。
```

可以看到，增加训练轮数的时间也是可以接受的，但是模型的准确率依旧是达到一定高度后就不在变化。

看一个更加极端的例子，将神经元数增加至128个，训练轮数增至200000轮，训练结果为：

```
Training iteration = 180000, validation error = 0.026200
Training iteration = 190000, validation error = 0.028400
Test error with final model = 0.029000
历时 46.569896 秒。
```

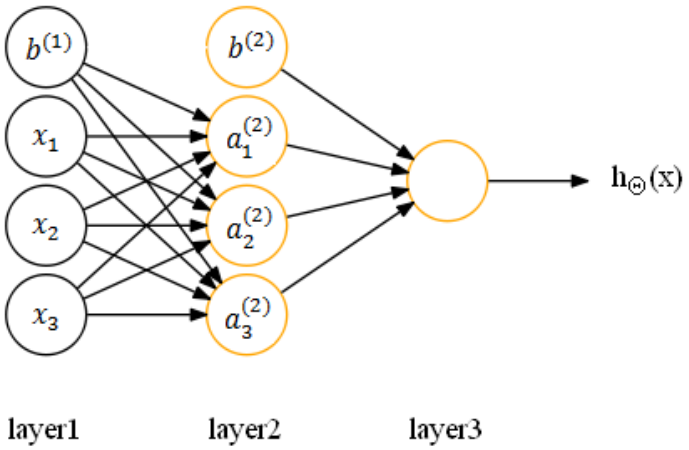
如果你有印象，在上一小节，我的128个神经元的网络在100000轮的训练下已经达到了0.026000的准确率，也就是说，增加了训练轮数后，模型效果不增反降！答案显而易见，模型发生了过拟合。因此一味地增加训练轮数也是走不通的。

接下来，让我们尝试更多的优化方法吧。

## Question 6 Add bias at each layer

目前，在我们的网络中，只有第一层的输入有bias项，后面的隐藏层和输出层均无bias项。这对模型输出的整体调整并不有利。当某一层的输入分布中心不为0时，bias项有利于给予偏置补偿，提高神经元的拟合能力。代码详见**Q6.m**。

具体实现方法为：在每一层隐藏层输出后，加上一个值为常数的神经元，不失一般性，我将这个常数设为1. 具体结构可以参考下图：




$b^{(1)}, b^{(2)}$即为加入的常数神经元。但需要注意的是，加入该神经元之后，相应的梯度传播方程也要变化。因为篇幅原因，不在此做数学推导，但可以从直观上来理解这个过程：以$b^{(2)}$为例，它的加入只会影响**layer2**之后的传递，但在layer1之前，$b^{(2)}$并不起作用，因此，在向后传递梯度时，计算layer1的weight梯度时应将$b^{(2)}$的影响剔除。而计算layer2的weight时，则要考虑$b^{(2)}$带来的影响。具体实现可以参照代码**MLPclassificationPredict_bias.m**和**MLPclassificationLoss_bias.m**。

接下来，看看模型的效果：

- 10个神经元，不加bias项，训练200000轮：

```
Training iteration = 180000, validation error = 0.077600
Training iteration = 190000, validation error = 0.076200
Test error with final model = 0.081000
历时 17.810203 秒。
```

- 10个神经元，加bias项，训练200000轮：

```
Training iteration = 180000, validation error = 0.075400
Training iteration = 190000, validation error = 0.076200
Test error with final model = 0.071000
历时 19.349418 秒。
```

- 11个神经元，不加bias项，训练200000轮:

```
Training iteration = 180000, validation error = 0.075600
Training iteration = 190000, validation error = 0.074800
Test error with final model = 0.076000
历时 17.173640 秒。
```

可以看到，加入bias项之后还是对模型效果有少许的提升。但是这会不会是由于加了一个bias项，参数量增大一个带来的影响呢？因此，第三组设置为11个神经元的无bias项组起到了对照作用。可以看到的是，单纯增加一个参数量并不能给模型带来稳定的提升，因此，加入bias项对模型还是有一定增益作用的。

## 修改目标函数

目前，我的模型使用的分类器为多个logistic分类器，训练的目标函数为平方差损失。我们尝试在分类器以及目标函数这一方面对模型做出一些改变，并观察其对模型效果的影响。

## Question 5 Use softmax layer and the negative log-likelihood loss

代码详见**Q5.m**。

### 使用**softmax**分类器

分类问题常用的分类器有logistic分类器和softmax分类器。logistic分类器多用于二分类问题，而softmax回归是logistic回归的推广版本，可以用于多分类问题，二者均基于线性模型建立。当然，使用多个logistic回归也可以表示多分类问题，就像该问题背景下，有K个类别就建立K个logistic分类器。但是，多个logistic分类器和softmax分类器的效果还是有一定区别。多个logistic分类器淡化了类别和类别之间的排斥性，需要的分类的物体可以有多个类别，类别之间也可以有交叉。但softmax分类器则强调‘非此即彼’，类别与类别之间互斥。对于该题而言，选择softmax分类器更能达到识别数字的效果。

具体实现来看，softmax的计算公式为：

$$x_i = z_i W_{out} + b_{out}$$

$$p(y_i) = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}$$

$x_i, i = 1, \cdots, n$是输出层输出的10个类别的结果，通过上面的公式，可以将输出结果转化为概率，使之具有归一化的性质，实现代码见**softmax.m**。

修改了分类器之后，我们顺带修改目标函数后再对模型进行测试。

### 使用交叉熵损失函数

函数形式为：

$$L = -\sum_i y_i log(p(y_i))$$

从表达式上来看，该损失函数只考虑了正确类别的预测结果的损失，根据损失函数进行梯度推导：

$$\frac{\partial p(y_i)}{\partial x_i} = p(y_i)(1 - p(y_i))$$

$$\frac{\partial p(y_i)}{\partial x_j} = -p(y_i)p(y_j), \quad i \neq j$$

$$\frac{\partial L}{\partial x_k} = \frac{\partial p(y_{true})}{\partial x_k} \cdot \frac{\partial L}{\partial p(y_{true})}$$

$$= p(y_{true}) - 1, \quad x_k \text{ is true label}$$

$$= p(y_{true}), \quad x_k \text{ is not true label}$$

实现代码见**SoftmaxPredict.m**、SoftmaxLoss.m、Q5.m以及**linearIndOnehot.m**（这个函数是为了将标签转化为0-1向量，以配合softmax使用），来看看模型效果：（超参数组合为：10个神经元，学习率1e-3，最大训练轮数为100000轮）

修改分类器及损失函数之前：

```
Training iteration = 90000, validation error = 0.082400
Training iteration = 95000, validation error = 0.082800
Test error with final model = 0.086000
历时 9.689767 秒。
```

修改分类器及损失函数之后：

```
Training iteration = 90000, validation error = 0.067000
Training iteration = 95000, validation error = 0.065000
Test error with final model = 0.049000
历时 11.100112 秒。
```

可以看到这个改变对模型还是有显著提升的。修改分类器的影响在上面已经解释过，为什么修改损失函数会带来影响呢？事实上，在分类问题中，常用的损失函数是交叉熵函数，而非平方差损失函数，二分类交叉熵函数数学形式如下：

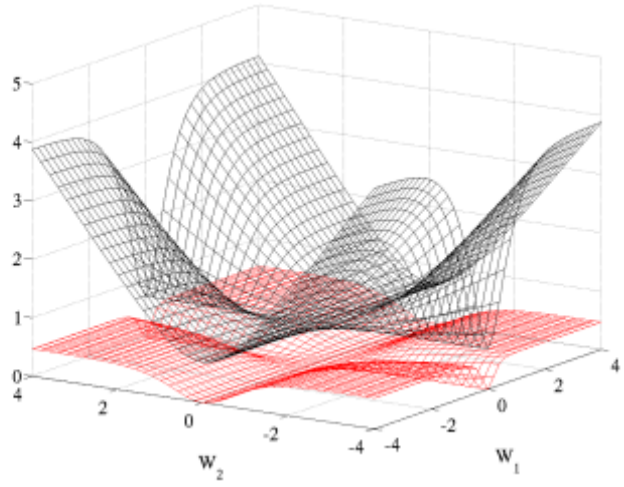
$$L = -(ylog(y) + (1 - y)log(1 - y))$$

多分类的交叉熵可以写为：

$$L(Y) = -\sum_{k=0}^{K} y_k log p_k$$

写成简化形式，只保留正确类别就是上面的形式。交叉熵损失函数比起平方差损失函数的优势在于：

- 由于sigmoid型函数的性质，导致其在饱和区（函数两端）的导数接近于零，而平方损失函数又在接近正确值的地方梯度很小，这导致在实际值无论在远离真实值时还是接近真实值时都更新非常慢，不利于函数更新。（下图上面的函数为sigmoid函数，下面为平方函数）




(Xavier et al., 2010)

- 平方损失函数作为最大似然理解，其假设为结果服从高斯分布，这是一个很强的连续变量假设，和实际并不相符。对于分类问题，伯努利分布更加贴合，其最大对数似然函数形式即为交叉熵损失函数。
- 交叉熵损失函数只和正确分类的预测结果比较相关，而平方损失的数学形式决定了错误分类的预测结果也会对损失值造成较大影响，只有当所有错误分类的预测概率都达到比较平均的时候，损失才会比较小，而这在模型中是非必要的。

### 隐藏惊喜

在完成前三大部分后，我的模型在训练集上的准确率已经可以达到100%!

```
Train error with final model = 0.000000
```

## 参数更新的优化

尽管前面从代码架构和目标函数两大方面对模型做出了改进，但如果参数更新比较初级的话，可能会出现模型参数在最优值附近振荡、过拟合等现象，一样会导致模型没有办法达到最佳效果。然而，我们并不能’看见‘参数最优值所在的位置，我们必须对参数更新过程进行优化，以达到更优的下降曲线。

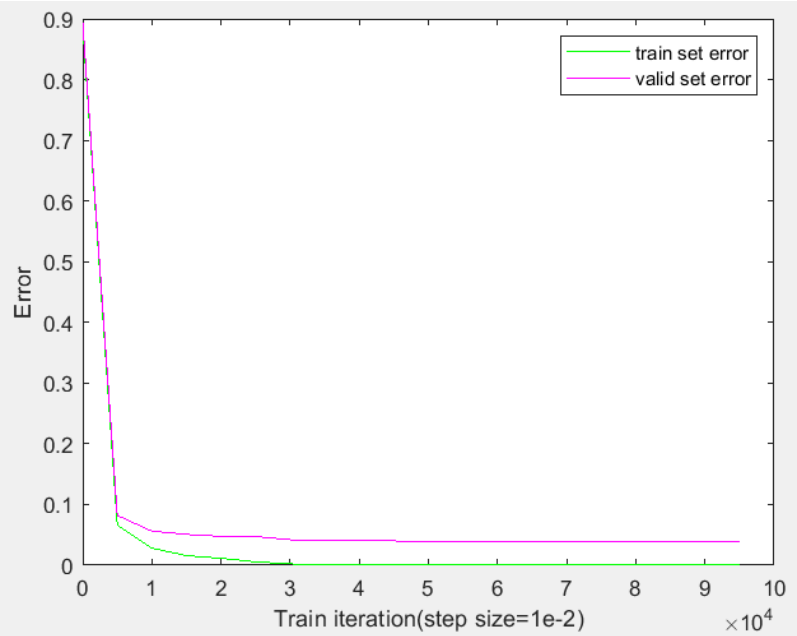
## Question 2 Change step sizes

代码详见**Q2.m**。

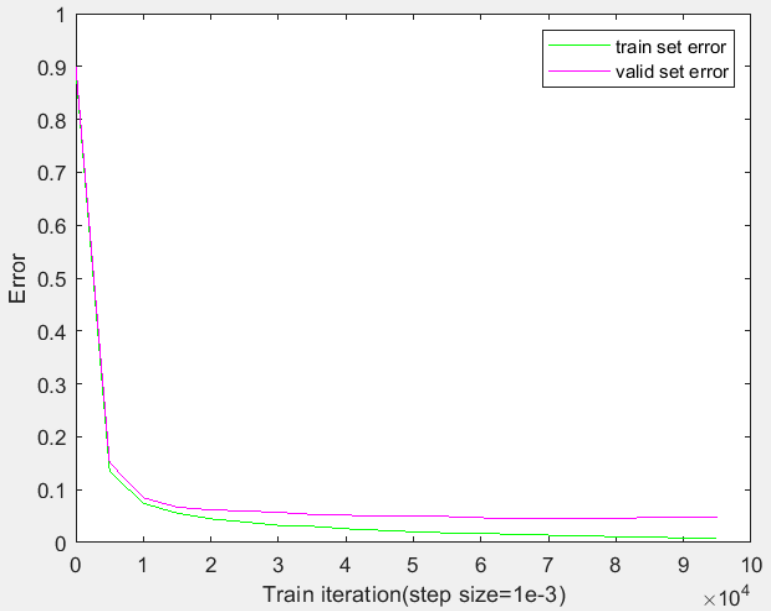
### 改变步长

目前为止，我们的更新步长或者说学习率恒为常数1e-3，先改变这个常数感受一下步长对模型训练的影响。（以下使用神经元数为64，训练轮数为100000轮）

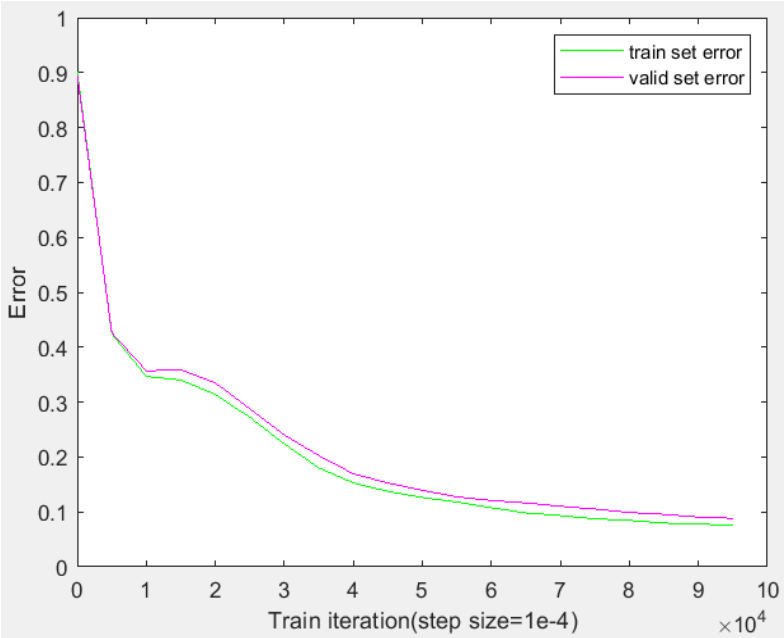
- 步长为1e-2：测试集准确率为0.037000




- 步长为1e-3：测试集准确率为0.039000




- 步长为1e-4：测试集准确率为0.076000




显而易见，学习率越大，模型训练越快，但也越容易过拟合，第一个图中虽然训练集的错误率已经和0重合，但验证集错误率依旧比较大。

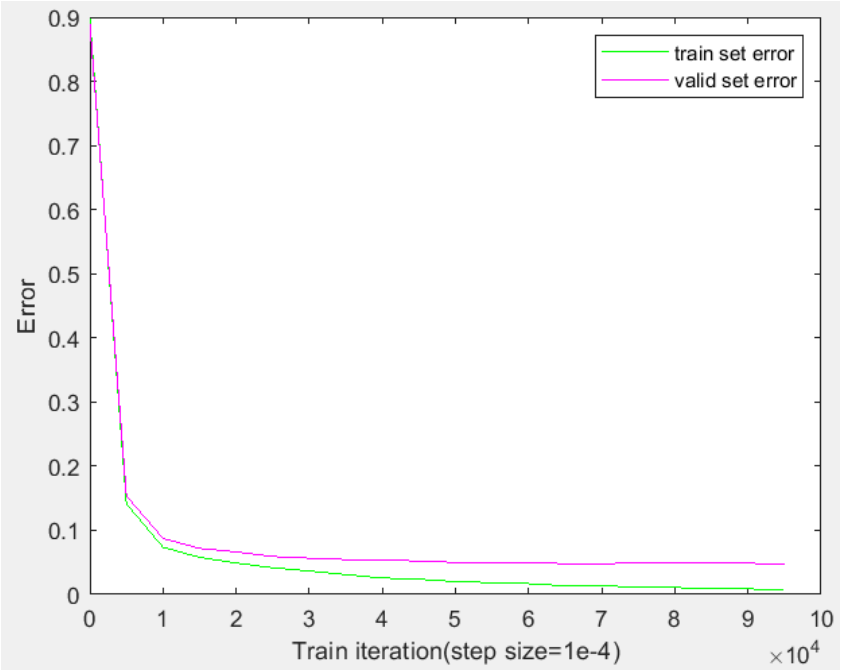
使用**Momentum**算法

目前为止，我们使用的梯度下降方法是随机梯度下降，这是一个振荡非常大的下降方法（后面会详细介绍），这些不必要的振荡减慢了参数逼近最优值的速度。因此，可以使用Momentum算法对冲振荡：

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

这相当于对原始梯度做了一个平滑，然后再用来做梯度下降。实验表明，相比于标准梯度下降算法，Momentum算法具有更快的收敛速度。Momentum通过对原始梯度做了一个平滑，正好将振荡的梯度抹平了，使得参数更新方向更多地沿着与最优值的直线方向进行，因此速度更快。

使用$\beta = 0.9$更新模型，选用上面训练效果不太好的1e-4步长，其他超参数保持不变，训练结果如下：测试集准确率为0.045000




可以看到模型训练错误率下降速度大大提升！误差下降的曲线逼近前面两个的步长的下降曲线，且训练集正确率有有很大提升（0.076000->0.045000）。但在1e-3的步长上再进行实验，会发现训练集误差仍为0.039000.可见，momentum算法主要还是加速参数下降速度，从而在一定训练轮数下尽快达到’最好效果‘，但对’最好效果‘的影响并不大。这可以帮助加快模型训练，也是不错的效果。

改变步长序列

从第一个步长的实验可以看到，当步长比较大时，参数下降（或者说逼近最优值）比较快；相反，参数比较小时，参数下降比较慢但较为稳健。一个很天然的想法是，我们希望在训练开始的时候，参数可能比较远离最优值时下降快一些，更新步长较大，但在训练后期，参数比较接近最优值时，更新步长小一点，以免产生较大震荡。因此，我将给步长乘上一个衰减系数，使得它一开始较大，但每下降一轮都会变小一些。

经过多次尝试后，我将这个衰减系数定位0.999，且每训练5000轮衰减一次。在1e-3的初始步长上测试：

```
Training iteration = 95000, validation error = 0.039400
Test error with final model = 0.034000
历时 20.267130 秒。
```

可以看到最终结果还是有一定提升的。但在其他的一些超参数上测试，并不是一定会得到较好的优化，因此，这个方法是否有用还要等到最后组合最佳模型的时候再观察。

## Question 4 Regularization

事实上，在很久之前，我的模型在训练集上的准确率就已经达到了100%，但是验证集和测试集的错误率居高不下，甚至在加了几个优化算法后，正确率不增反降，很显然，这是发生了过拟合。接下来，尝试正则化，这是非常有效的改善过拟合的方法。代码见**Q4.m**和**SoftmaxLoss_Reg.m**。

### $l_2$ regularization

通过在目标函数（损失函数）上加参数加权的2-范数达到正则化的目的。即loss function变为

$$L = \sum_i log(p(y_i) + \lambda||W||_2^2$$

对每一层的weight求梯度只需在原来的基础上加上$2\lambda W_i$即可。测试代码效果：以神经元为64，步长为1e-3，β为0.9作为超参数组合：

- λ为0：

```
Training iteration = 90000, validation error = 0.038800
Training iteration = 95000, validation error = 0.039000
Test error with final model = 0.038000
历时 18.002695 秒。
```

- λ为1e-3:

```
Training iteration = 90000, validation error = 0.037600
Training iteration = 95000, validation error = 0.036800
Test error with final model = 0.036000
历时 17.475340 秒。
```

- λ为1e-4：

```
Training iteration = 90000, validation error = 0.036000
Training iteration = 95000, validation error = 0.036600
Test error with final model = 0.033000
历时 17.123539 秒。
```

- λ为1e-5：

```
Training iteration = 90000, validation error = 0.038400
Training iteration = 95000, validation error = 0.037600
Test error with final model = 0.035000
历时 17.382639 秒。
```

从结果上来看，**λ**为**1e-4**的效果在几组实验中效果最好，作为之后的weight decay参数。

### Early stoppping

除了通过常见的结构化目标函数来降低过拟合的影响，常用的一种方法是early stopping。简单粗暴，在验证集的错误率不再下降的时候直接中断模型的训练，以此来预防模型的过拟合。但在实际情况下，无论是验证集还是训练集的错误率都不可能是平滑下降或下降后再上升的，而是会有一定的抖动，尤其是目前我们使用的梯度下降方法还为随机梯度下降。如果简单地当验证集错误率一开始上升就中断训练，训练时间也许会很短，但泛化误差可能会很大。为了在这二者中间选取一个可以接受的平衡，我参考了*Early Stopping -- but when?*这篇文章提出的**GL**标准（Prechelt，2012）。

记$E_{opt}(t)$是在$t$时刻取得的最小的验证集误差，定义泛化损失generalization loss：

$$GL(t) = 100 \cdot (\frac{E_{va}(t)}{E_{opt}(t)} - 1)$$

其中$E_{va}(t)$为迭代时刻$t$的验证集误差，$E_{opt}(t) = min_{t' \le t} E_{va}(t')$。当$GL(t)$大于阈值$\alpha$时，就认为其出现了过拟合，停止训练。参考论文中的参数，我们暂时将这个阈值设为**2**.

模型训练的超参数组合为64个神经元，步长1e-3，momentum的β设为0.9，最大训练轮数为10000.

当α为2时，

```
Training iteration = 0, validation error = 0.912800
It is time to stop train early, train iteration = 1500, and the validation error is 0.104000
Test error with final model = 0.099000
历时 1.097923 秒。
```

显然这个精度还远远不够，这可能是由于本身我的模型梯度震荡就比较厉害。经过多次调试，发现**α**值设为**50**是比较合适的：

```
It is time to stop train early, train iteration = 39700, and the validation error is 0.057600
Test error with final model = 0.058000
历时 11.066538 秒。
```

对比前面没有early stopping的模型训练效果，会发现我们大约牺牲了2%的准确率换来了节省接近1/3的时间，虽然在我的模型上，训练本来就跑的比较快，因此节省下来的这个时间看起来很不起眼，但这个比例如果放在比较大数据集或比较复杂的模型上，还是很有吸引力的。

## Question 7 Add dropout layer

Drop out的原理很简单，在每一层输出时，以一定概率丢弃掉一部分点，让其不参与接下来的训练，通过动态减少参数量的方法来预防过拟合。是很多网络都普遍使用的方法。实现也比较简单，但有一些需要注意到的点。

在每一个隐藏层（一般来说不在输入层做drop out，以保证输入信息的完整性）按照均匀分布随机地生成一些0到1之间的数，小于drop out概率的就认为其是被丢弃的节点。之后，需要对保留下来的节点进行归一化，即除以保留概率=1-丢弃概率，这一步是为了保证drop out前后的隐藏层输出期望保持不变，达到无偏的效果。在丢弃的过程中，要注意bias项并不参与drop out。

在前向传播丢弃节点后，反向传播也要做出对应的修正：

$$\hat{x} = dropoutMask \odot x$$

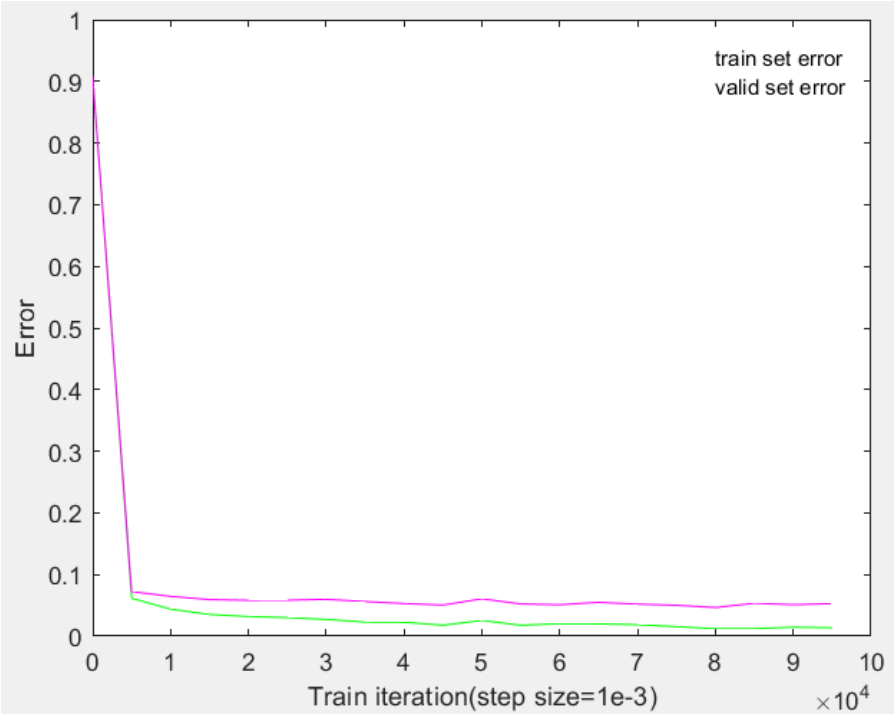
$$\frac{\partial l}{\partial x} = \frac{\partial \hat{x}}{\partial x} \cdot \frac{\partial l}{\partial \hat{x}}$$

$$= dropoutMask \odot \frac{\partial l}{\partial \hat{x}}$$

$$= dropoutMask \odot error$$

代码详见**Q7.m**和**SoftmaxLoss_Drop.m**。实现后，来看看模型训练的效果，超参数依旧是64个神经元，步长1e-3，momentum的β设为0.9，最大训练轮数为10000.对隐藏层的drop out概率暂时设置为问题清单中推荐的**0.5**。

先看模型训练时的训练集和验证集表现：




对比之前的训练表现，可以发现训练集的误差曲线和验证集的误差曲线靠近了一些，说明drop out方法对降低过拟合还是有一定效果的，但是再看测试集的误差率：

```
Test error with final model = 0.052000
历时 20.161530 秒。
```

发现模型的效果比之前差了一些，说明这个方法对我的模型的准确率是有一定的牺牲的。除此之外，在实验的过程中，我发现加入dropout的模型最终准确率很不稳定，且增加了训练轮数除去欠拟合的可能性后，这两个问题依旧存在。猜想是由于数据量比较小或者参数量比较小，导致dropout这个方法对模型的影响比较大，不太适用？后续会在更复杂的网络，比如CNN上进行实验。

## Question 8 Fine-tuning

在此之前，每一层的参数我们采用梯度下降的方法，让模型‘自己’寻找最佳参数。但根据一些优化知识，如果目标函数为平方差形式，那么最后一层的最优权重是完全可解的凸优化问题。具体地，推导如下：

记$fp_{end}$为输出层的输入，$W$为输出层的输入权重，$\lambda$为$l_2$正则化的权重，则优化问题可写成

$$\min_{W} ||fp_{end}W - Y||_2^2 + \lambda||W||_2^2$$

求导可得，$W$的最优值为

$$W = (fp'_{end}fp_{end} + \lambda)^{-1}fp'_{end}Y$$

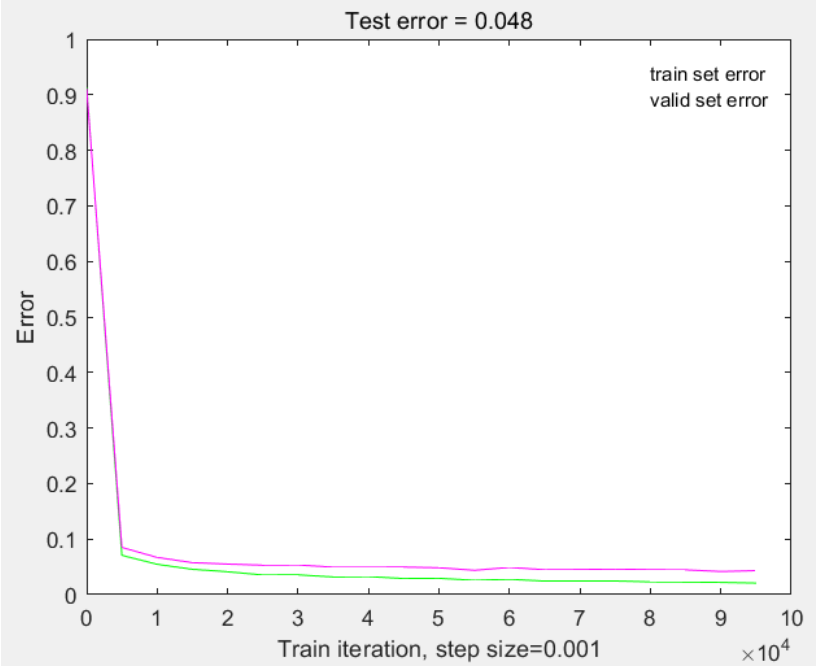
接下来，就是代码的实现。为了匹配该优化算法需要的平方损失函数，我回到A12小节的代码，简单来说，就是优化部分只有向量化和ReLU激活函数的时候，并在此基础上加上$l_2$正则化（主要是为了防止奇异矩阵求逆出错）和**Fine-tuning**优化。

在实现代码的时候，有一个小细节，需要多少轮微调一次输出层的权重呢？经过多次尝试后，发现以下规律：

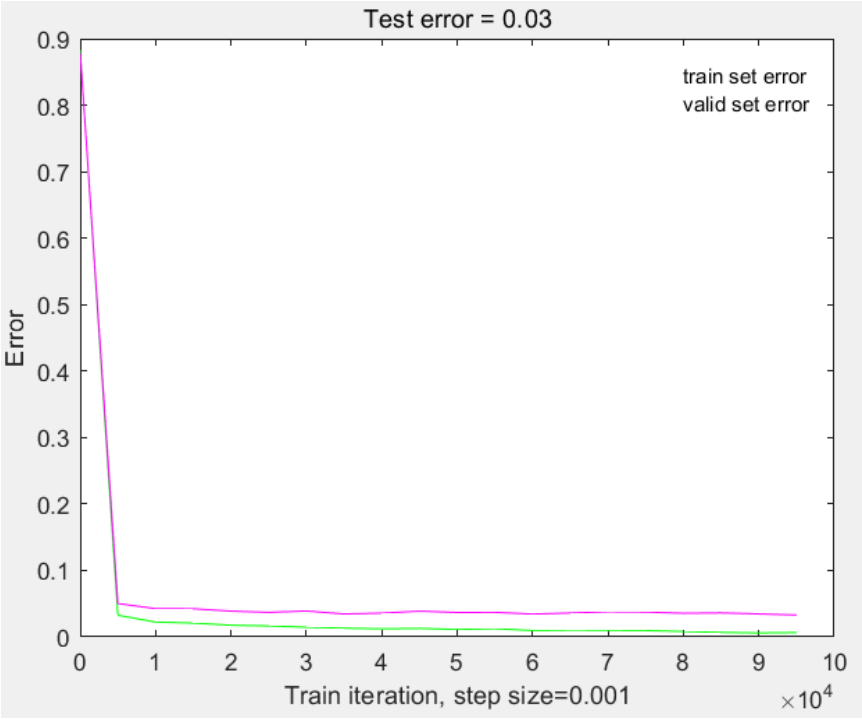
- 当微调的轮数周期较小时，比如最极端地，每轮调一次，最大的问题就是程序运行速度大大降低，但模型的准确率得到的增益并不与其增加的时间成正比。在实践中过程中，还发现当周期数特别小时，容易出现梯度爆炸现象。猜测原因可能是太早就对最后一层的权重进行的微调，使得整个模型下降的‘方向’错了，比较难以纠正回来。
- 当微调的轮数周期较大时，微调的效果又很难发挥出来，对模型的增益比较小。

因此，经过多次尝试，最终我将微调周期定为**500**轮一次。先不改变其他超参数（64个神经元，学习率为1e-3，λ为0.1），实现代码见Q8.m、FineTuning.m和MLPclassificationLoss_Tuning.m，看看微调对模型的增益效果：

- 不使用Fine-tuning:




- 使用Fine-tuning:




可以看到，对模型的增益效果还是很大的!

当然，优化其他超参数之后对模型的效果肯定是更好的。值得一提的是，在多次尝试调节超参数后，我调节到了一个截至目前最优的模型，超参数组合为（128个神经元，100000次训练轮数，步长为1e-3，λ为0.1），模型在测试集上的准确率为98.1%。

```
>> yhat = MLPclassificationPredict_vec(best_w, Xtest, nHidden, nLabels);
fprintf('Test error with final model = %f\n', sum(yhat~=ytest)/t2);
Test error with final model = 0.019000
```

将这个模型保存下来。

## Additional item 3 Mini-Batch gradient descent

到目前为止，我使用的梯度下降方法都是Stochastic gradient descent，这是一个速度极快的训练方法，但其也有不能被忽略的缺陷。总的来说，梯度下降方法分为三种：

- Batch Gradient Descent：每次梯度更新使用全部的样本
- Mini-Batch Gradient Descent:每次梯度更新使用b个样本，b>1，小于全部的样本数
- Stochastic Gradient Descent：每次梯度更新使用1个样本

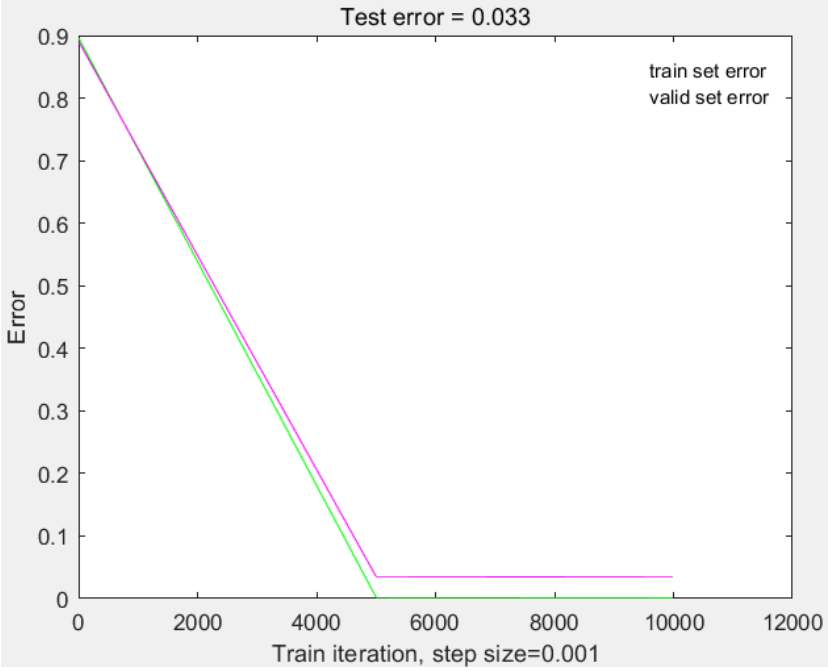
当每次更新权值使用的样本越多，梯度下降越快（更接近全局最优值），参数收敛也更快，但相应的，由于一次迭代的数据量大，训练过程也会很慢；样本少的情况下，每次迭代的速度很快，但收敛的方向随机性比较大，甚至有可能不收敛到最小值。为了平衡上面BGD和SGD的优缺点，我们选取折中的办法，选取小批量的数据进行梯度更新。

实现上也比较简单，主要分为两步：

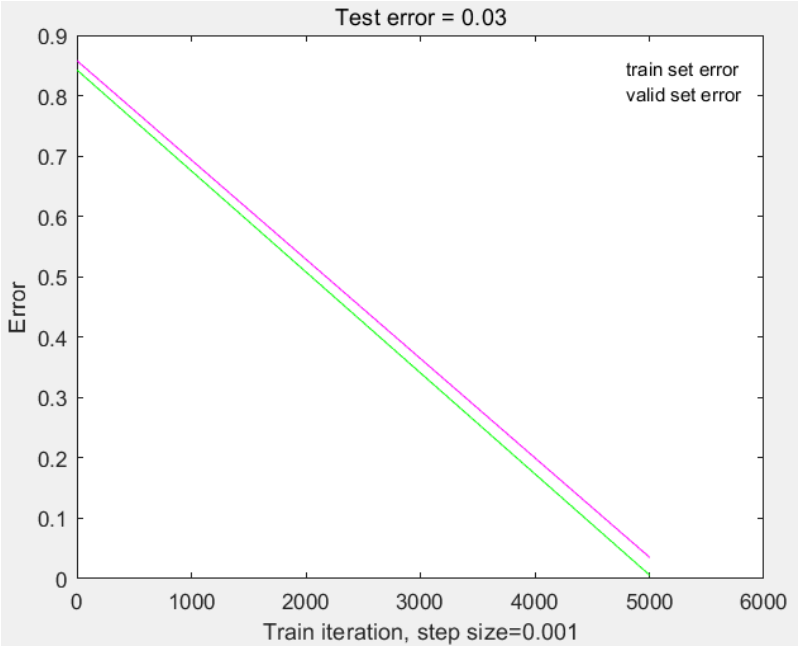
- 在训练之前，将数据进行批量划分，每批数据的大小为提前设置的超参数$K$。
- 训练时，每次梯度更新使用上面划分好的一批数据。在这时，之前做过的向量化就起了作用，如果没有做向量化，训练将十分耗时。

我使用上一题的训练框架进行测试，值得一提的是，由于batch方法的梯度下降较快，因此模型训练容易过拟合，必须使用**early stopping**方法来控制模型训练的轮数。除Batch size外，超参数组合为（最大训练轮数为100000，stepsize为1e-3，$l_2$正则化系数λ为0.1，early stopping的threshold α为20）。 代码见**A3.m**

- Batch size为32:




- Batch size为16:




多次实验下来，发现batch size对模型最终的准确率其实影响不大，但是模型的训练时间会得到大幅降低。

第二个有意思的点是，同样的early stopping的参数α在所有的batch size模型上的表现并不是一致的，总的来说，当batch size约大时，需要将α值相应地调小，比较极端的是，当batch size为1，也就是SGD时，$\alpha$可能要取到80才会有比较好的效果。这其实从侧面反应出，当**batch size**越大，模型的下降就越稳定，产生的波动越小。

## Additional item 4 Adam算法

前面我们使用了Momentum算法来对冲SGD带来的抖动。在Momentum之上，Adam算法更加优化了这一过程，通过调整模型的更新权重和偏差参数，来更好地更新参数。

在实现上，Adam主要分为两部分，记$g_t$为第t次向后传播的梯度：

1. 对梯度的一阶矩估计：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

2. 对梯度的二阶矩估计：

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

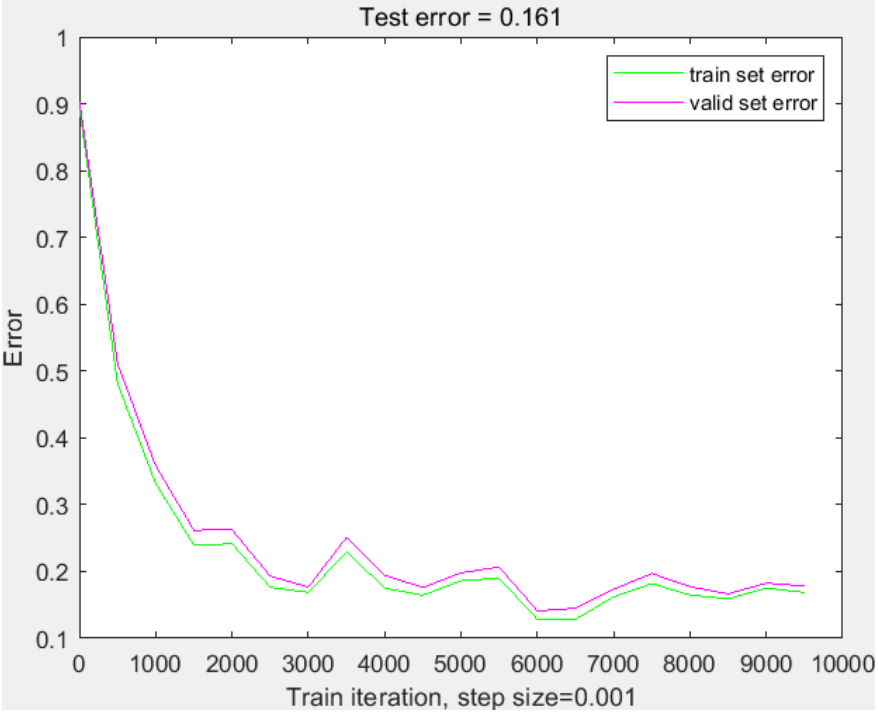
然后分别校正$m_t, v_t$，根据以下公式更新权重参数：

$$\hat{m}_t = m_t / (1-\beta_1^t)$$
$$\hat{v}_t = v_t / (1-\beta_2^t)$$
$$W_t = W_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

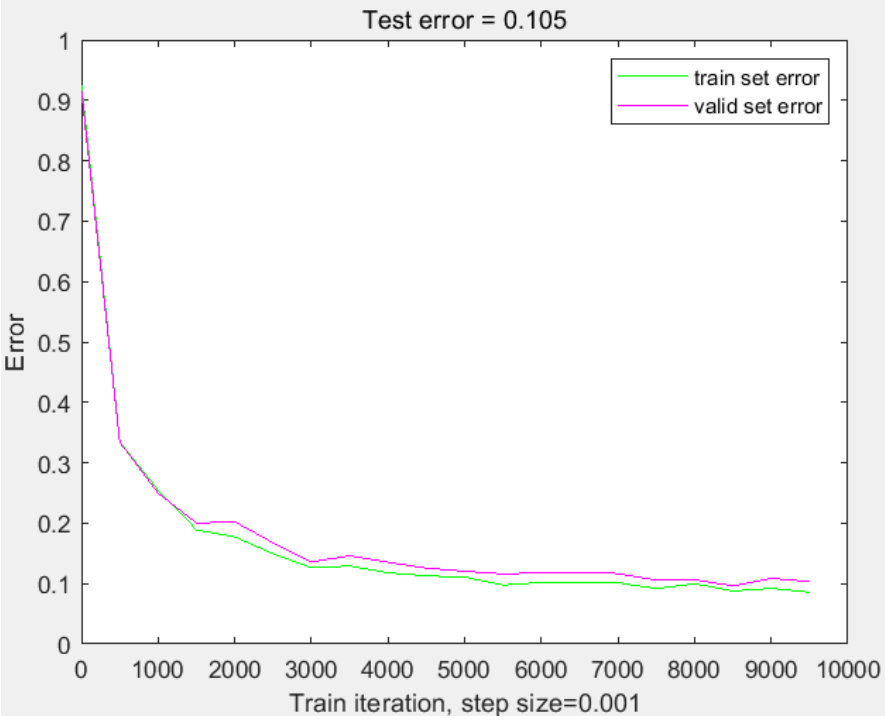
$\epsilon$是为了维持数值稳定性而添加的常数。实现代码见A4.m。

这里实验使用的超参数组合为最大训练轮数为10000，stepsize为1e-3，单层隐藏层神经元数为10，$l_2$正则化系数λ为0.1。Adam的相关参数使用epsilon=1e-8, beta_1=0.9, beta_2=0.999:

不使用Adam:




使用Adam算法：




可以看到模型训练变得非常平稳。

## Question 9 Data Augmentation

深度神经网络一般需要大量的训练数据才能获得比较好的效果。但是往往数据量，或者说有标注的数据量有限，因此可以通过数据增强的方法来增加数据量，可以提高模型鲁棒性，避免过拟合。数据增强分为两种：

- Offline augmentation：事先执行所有转换，会增强数据集的大小。适用于较小的数据集。
- Online augmentation：在送入训练之前，在小批量（mini-batch）上执行这些转换。更适用于较大的数据集。

这里为了降低后续模型每次的训练时间，我选取了前一种方法，提前对数据进行转换。

### Translation

将图像延X或Y方向移动，并使用**bilinear**插值方法填充多出来的边界，这样可以将图像对象的位置泛化，让神经网络看到所有角落。（见**transition.m**)

原始图像：　　向右和向下同时平移8个像素点：

### Rotation

旋转图像任意角度，让网络忽略数字角度这个特征。需要注意的是旋转后的图像尺寸会有一定改变，同样使用**bilinear**方法插值边界位置，并缩放至要求大小。 (见**rotation.m**)

逆时针旋转45度：

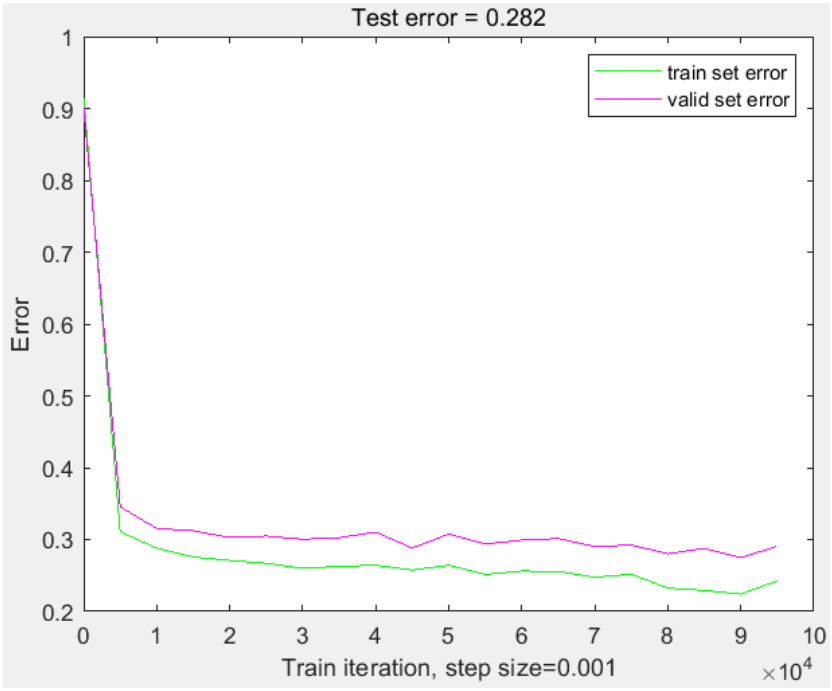
### Resize

和上面rotation时用到的缩放不一样，在这个变换中，可以将图像任意放大放小scale倍，然后使用固定大小（16*16）的窗口截取新图像。在图像缩放的过程中，依旧使用**bilinear**插值方法。但在scale小于1时，为了使图像的大小变大一些，我使用了pad方法在图像四周加入**0**，使用0的原因是数据集的背景是黑色的。(见**resize.m**)

上图放大1.5倍：　　缩小0.5倍：

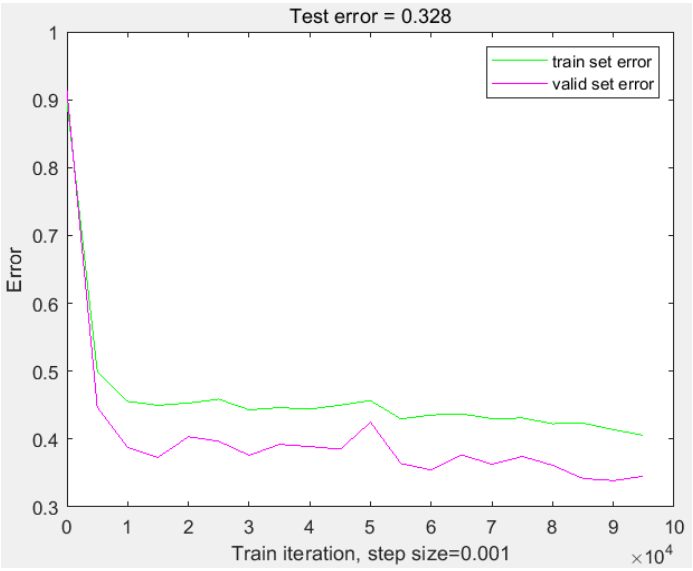
完成以上图像变换功能后，将数据送入模型进行训练。这里使用的神经网络框架为初始不加任何优化方法的网络，仅将神经元修改为64个。（见代码**Q9.m**）

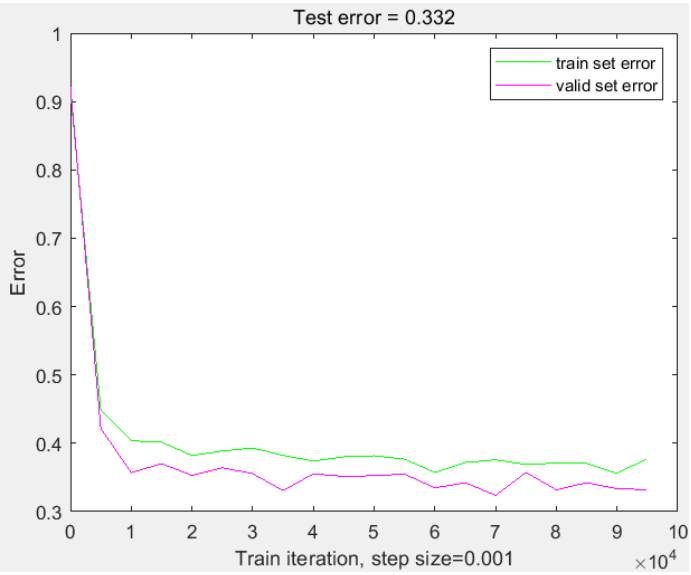
原始64个hidden units模型训练过程：



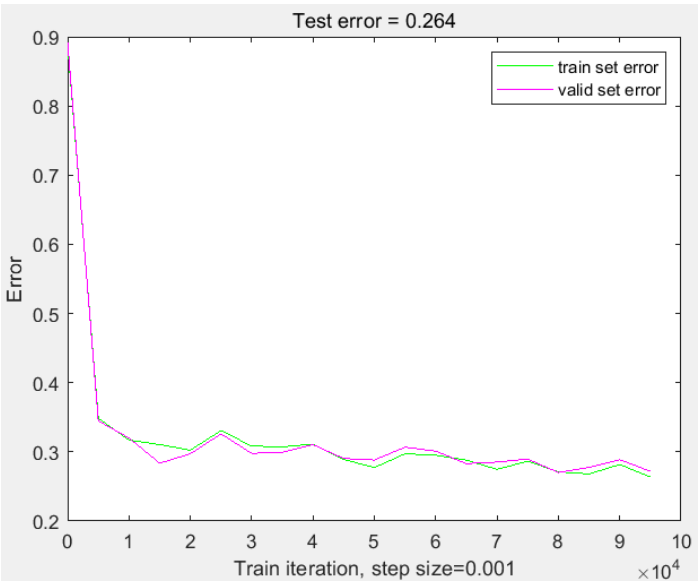

可以看到模型的训练集的准确率远高于验证集和训练集，发生了较严重的过拟合。

1. 数据增强一倍，使用translation：（移动范围为0-3个像素点）　（移动范围为0-2个像素点）



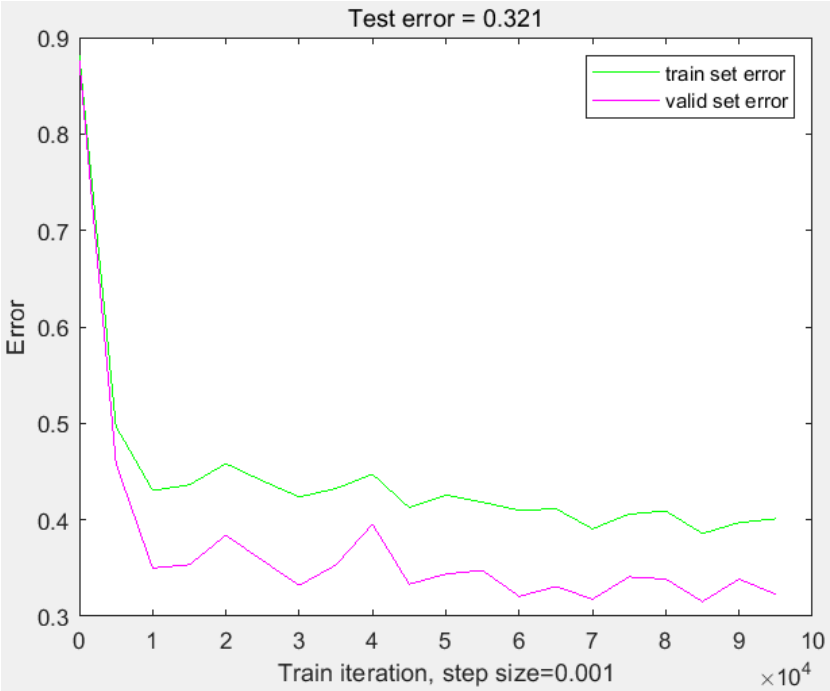

（移动范围为0-1个像素点）



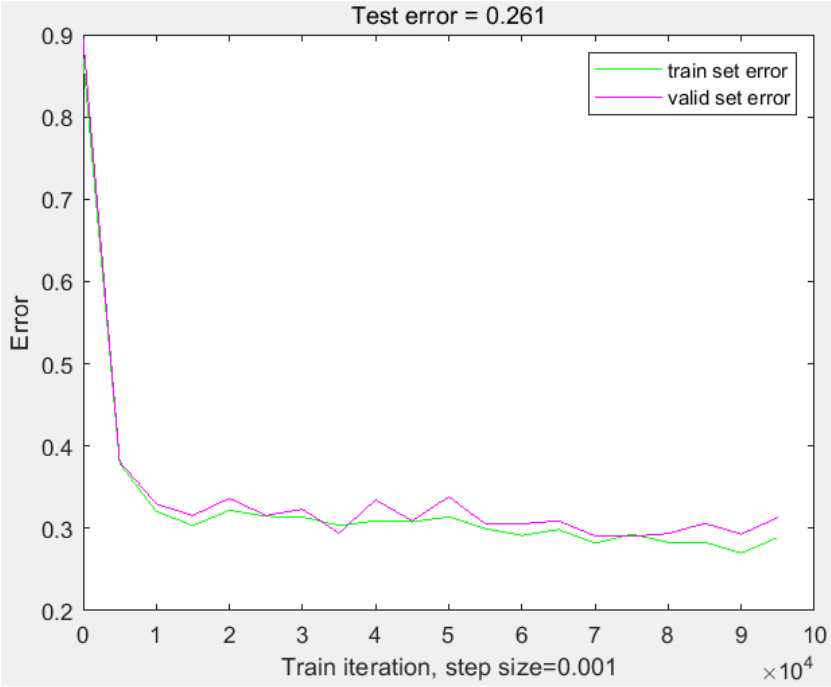

可以看到第三种选择（0-1个像素点）效果是最好的，可能是由于本身图像尺寸就比较小，如果移动的范围比较大，反而会丢失一些信息，造成模型训练效果变差。可以看到最好的超参数下，训练集和验证集的准确率变得相差不大，过拟合得到缓解。

2. 数据增强一倍，使用rotation

（旋转范围为-45度-45度）



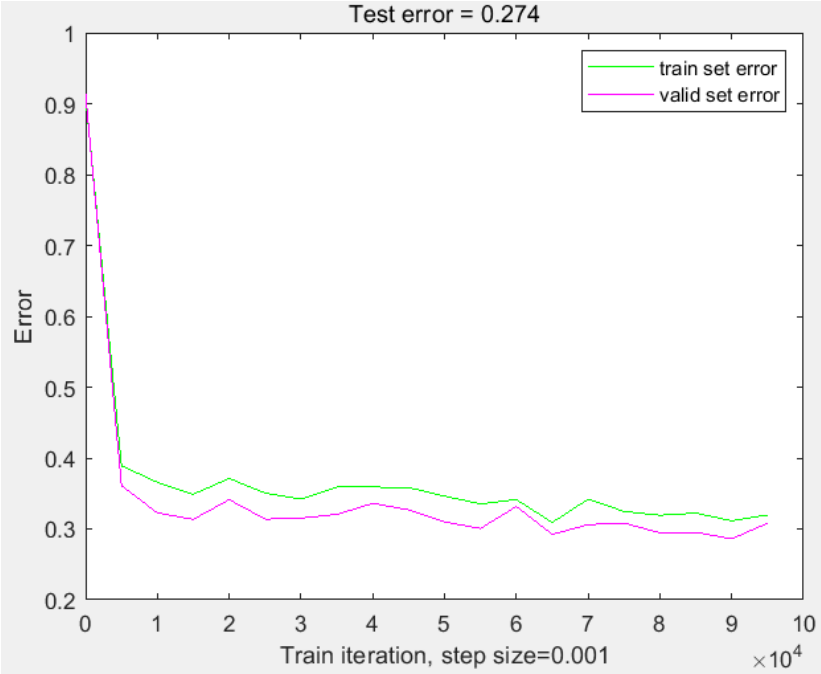

继续尝试了30度，15度和10度的摆动范围之后，效果最好的是**15**度（旋转角度为-15度到15度）



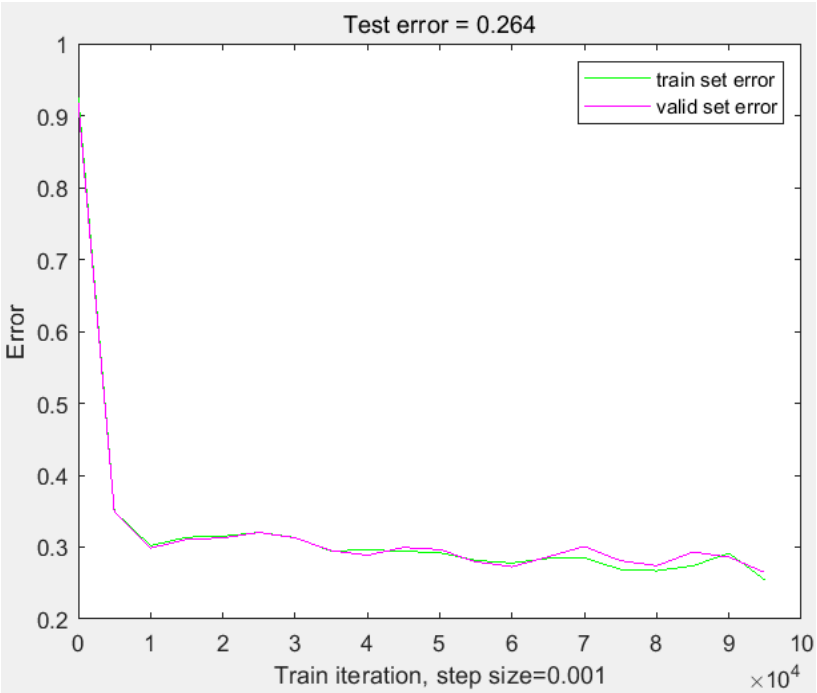

同样，过拟合问题得到较好的缓解。之所以比较大的旋转角度在该模型训练上不起作用，很可能是由于该数据集的图像都比较‘正’，旋转太多角度反而会给网络带来新的要提取的特征，且这些特征在其他数据集上并没有起作用，因此效果不太好。

3. 使用resize增强数据一倍

数据缩放倍数范围为0.6-1.2倍：



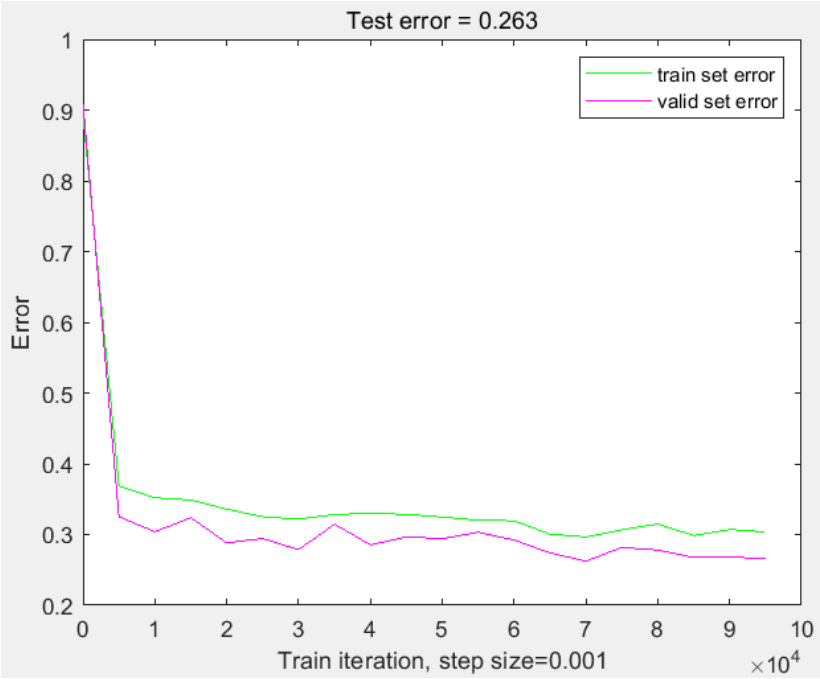

数据缩放倍数范围为0.8-1.1倍：




可以看到这个方法在此超参数下效果也非常不错！resize范围不能过大和上面原因一样，图像尺寸本身较小，再进行裁剪可能会带来信息丢失。

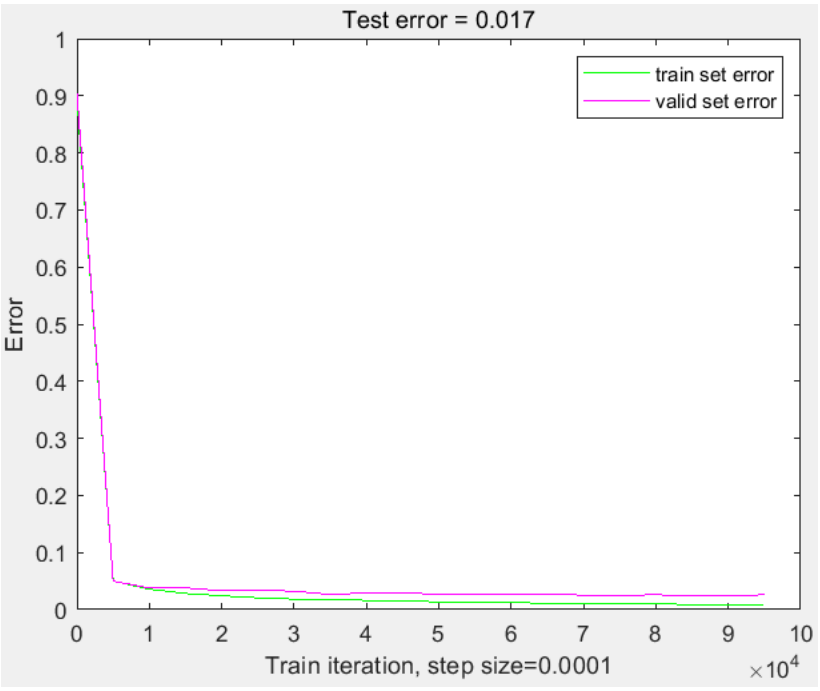
最后，我将三种数据增强方法一起使用，每个方法都使用上面挑选出来的最好的超参数，综合下来，对数据集共扩充了三倍，增强后的数据保存于**digits_argumentation.m**。 （见代码**data_argumentation.m**)

在模型上测试三种数据增强方法一起使用的效果，见代码**Q9.m**：




到目前为止，在所有优化算法实验效果中，该方法是缓解过拟合最好的方法。

Tip: 我将生成的增强后的数据集在先前最好的模型上（即Q8的模型框架）测试效果，在神经元为256个，stepsize为1e-4的超参数下，更新了我最好的模型：




可以看到模型几乎没有过拟合，梯度下降也非常平稳，没有太大波动，表现非常好。

## 使用卷积网络

## Q10 2D convolutional layer

到目前为止，使用的一直是前馈神经网络框架。而在图像处理上，卷积神经网络则使用的更加广泛。比起前馈神经网络，CNN的优点主要在于两点：

- 对于尺寸较大的图像，如果使用前馈神经网络往往需要非常大的隐藏层参数量，对于训练来说，是很困难的。而CNN则可以大幅度降低参数量。
- 卷积使用的是局部感知野，比起全局的像素联系，更关注局部的、比较紧密的像素联系，这和生物学上的视觉系统接收信息的结构也更相近。

我将第一层之前的隐藏层替换为卷积层，并在卷积层之后又接了一层隐藏层进行信息整合。

网络训练时，向前传播只需和给定的卷积核做卷积操作即可，假定第$l$层为卷积层，卷积核为$W^l$，运算即为：

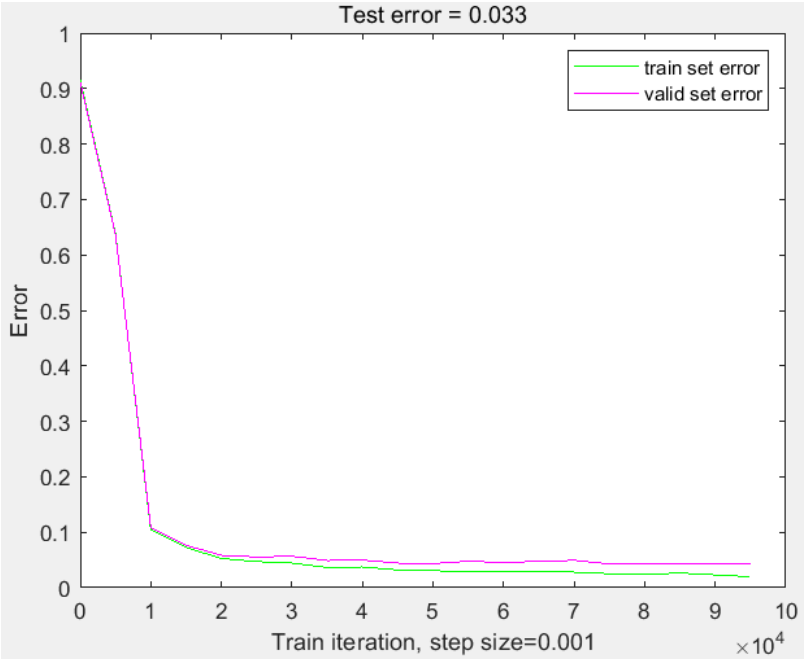
$$z^l = a^l \otimes W^l$$

相应的向后传播推导公式为：

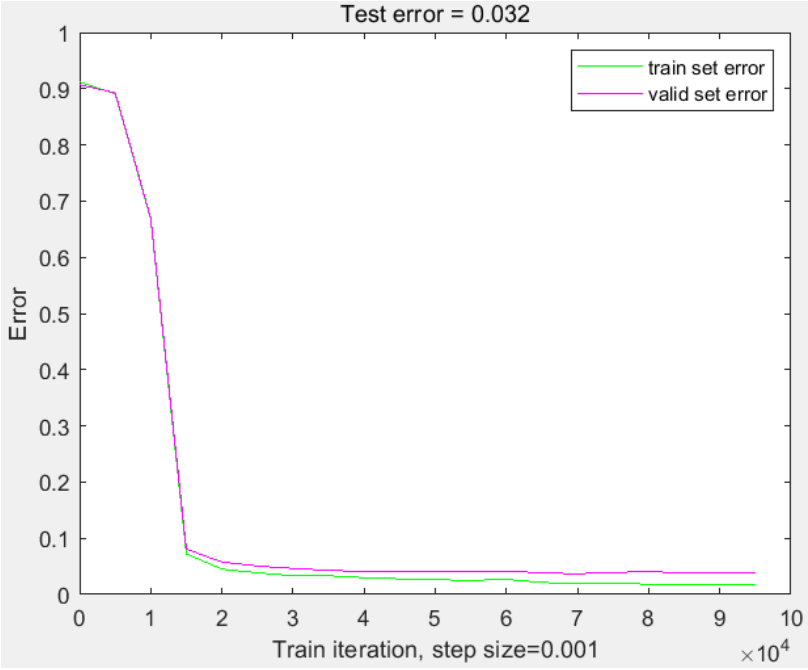
$$\begin{aligned}\frac{\partial l}{\partial W^l} &= \frac{\partial z^l}{\partial W^l} \cdot \frac{\partial J}{\partial z^l} \\ &= \frac{\partial z^l}{\partial W^l} \cdot \delta^l \\ &= flip(a^l) \otimes \delta^l\end{aligned}$$

具体实现见代码**Q10.m**、**CNN_Loss.m**和**CNN_Predict.m**。

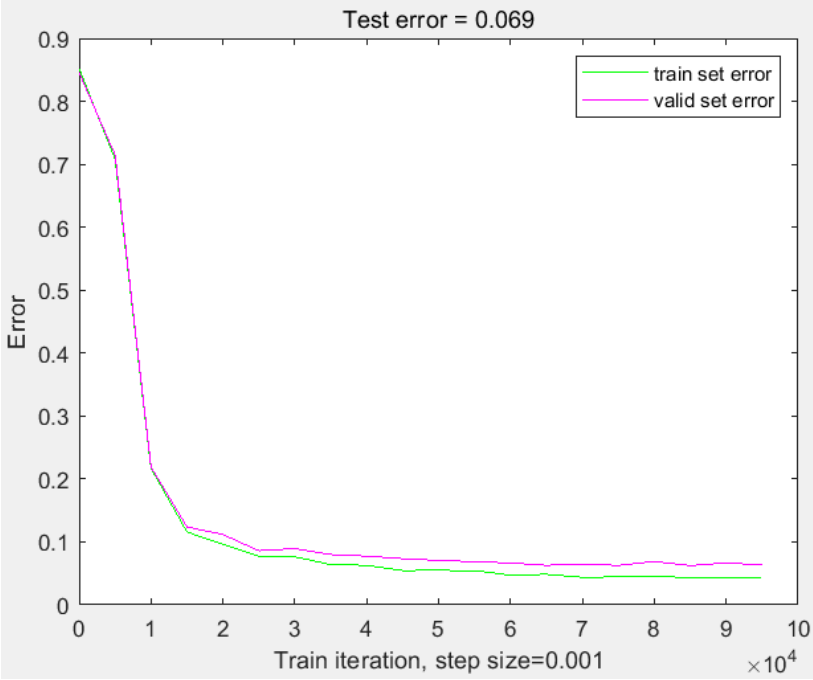
先使用Question list中推荐的kernel size为5，stride=1，padding=0。后接单个隐藏层，神经元为64个：




再尝试改变卷积核的大小，看其对网络训练的影响。改变卷积核为3*3：




两个模型区别不大。使用大一点的尺寸，卷积核为7*7：




可以看到当卷积核为7*7的时候，网络的效果变的差了一些，猜想是一是由于卷积核的尺寸变大后，感受野变大，但是由于这次使用的数据集中的图像的尺寸比较小，感受野增大后反而不能体现卷积核关注图像局部变化的优势；二是由于卷积核变大，参数量也变大，模型还未能训练至最优，也可以看到上图中模型的误差还是又在不断下降（但是可以看到上面的训练过程已经出现了过拟合，因此这里不再选择增加训练轮数）。

总结来说，在我目前的模型中，卷积核尺寸影响不大。对比之前没有加卷积层的网络，也并没有得到显著的提高。猜想有以下几个原因：

1. 该数字图像分类问题比较简单，前馈神经网络已经能够很好的解决该问题，CNN的优势并没有太显现出来。

2. 卷积网络有很多其对应的优化算法，比如加入池化层，选择多通道的卷积层来增加提取的特征数量等等。而这里都没有实现，因此卷积网络并没有发挥出它对应的优势。进一步的优化可以考虑使用更多卷积网络优化算法来加强。

## 寻找最佳网络

在经过多种优化算法的取舍和多次调试超参数，最终，我的最佳模型使用的优化算法及相应的超参数设置如下（见代码**best.m**）：

使用两层全连接的隐藏层，神经元数量分别为512、128。

使用Q9生成的三倍的数据增强，具体数据的生成方法见Q9.

使用**Mini-batch gradient descent**，batch size为**32**.

每层的激活函数使用**ReLU**函数，对应地，使用**He**初始化方法初始化权重。

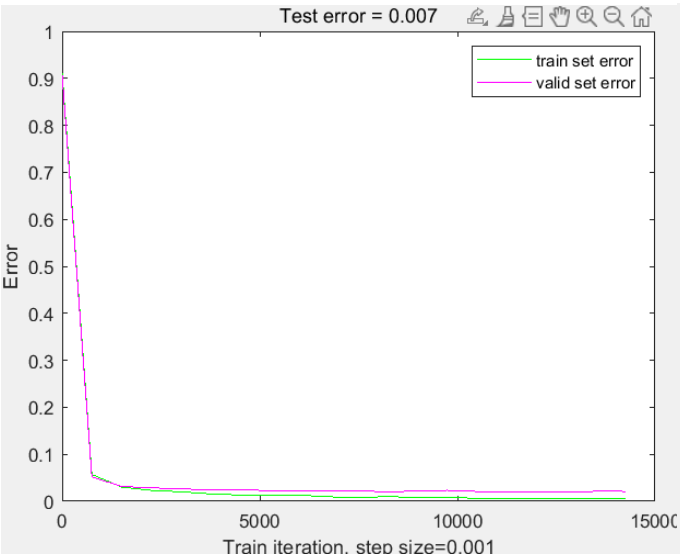
使用**Fine tuning**，每**500**轮训练微调一次最后一层的权重。

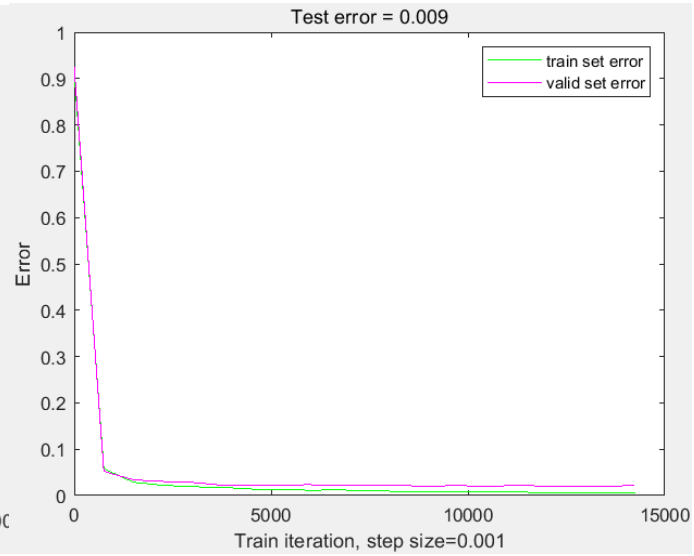
使用**Adam**算法更新参数，对应的超参数为$\epsilon$**=1e-8,** $\beta_1$**=0.9,** $\beta_2$**=0.999**

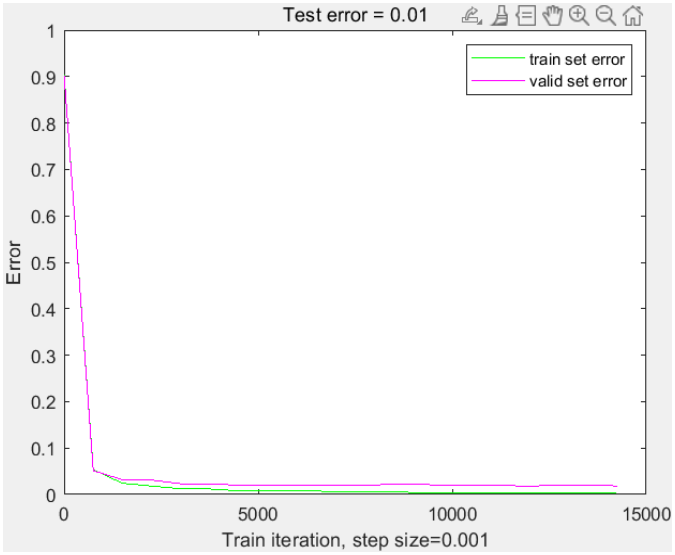
使用$l_2$正则化方法，罚项系数为**2**.

其余超参数：最大训练轮数为**15000**轮，stepSize为**1e-3**，损失函数为平方差函数.

为了防止模型的训练结果的随机性影响判断，在以上参数下训练多次检测其准确率，随机截取了以下三次训练结果：






总的来说，模型的效果还是比较稳定的，多次训练下来基本上测试集误差率能保持在0.011以下，最好的一次训练达到了**0.007**的误差率，我将相应权重参数保存为**best_weight_07.mat**文件，可以在我提交的任意一个工作文件夹下通过以下命令调用：

```
clear;
clc;
addpath(genpath('../'));
load digits.mat;
load best_weight_07.mat;
load mu.mat;
load sigma.mat;
t = size(Xtest, 1);
Xtest = standardizeCols(Xtest,mu,sigma);
Xtest = [ones(t,1) Xtest];
yhat = MLPclassificationPredict_vec(w,Xtest,[512,128],10);
fprintf('Test error with final model = %f\n',sum(yhat~=ytest)/t);
```

```
Test error with final model = 0.007000
```

回看模型的训练过程，会发现模型的错误率在非常稳定的下降，也几乎没有过拟合现象的出现，这主要得益于优化算法的选择和超参数的调配。在以上模型框架下，固定其他超参数，对模型准确率起重要的因素的就是训练轮数。虽然到后期，模型的错误率下降的很慢，但确实在依旧稳定地下降。事实上，在训练轮数约为10000轮时，我的模型已经可以稳定的达到0.011的测试集错误率，但如果想要突破0.01的测试集错误率，则还需增大训练轮数。且可以预见的是，如果继续增大训练轮数，模型会达到更好的测试集错误率，但是到训练后期，训练时间的增长已经和带来的测试集错误率的下降程度不成正比。如果想要尽可能高的测试集准确率，那么大可以继续增大训练轮数；但如果对准确率要求没有那么高，且希望训练时间越短越好，那么在我的模型下，5000到10000轮的训练足矣。

## 参考文献

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". *Neural Networks*. **4** (2): 251–257.

邱锡鹏，**神经网络与深度学习**，机械工业出版社，https://nndl.github.io/, 2020.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, 1026-1034.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *AISTATS*.

Prechelt, L. (2012). Early Stopping - But When? *Neural Networks: Tricks of the Trade*.