Neural Networks for Handwritten Digit Classification

Hao Fu, School of Data Science
14307130013

Dec. 6th

0 Preface

0.1 Abstract

This report deals with a classification problem based on a handwritten digit database *mnist* provided by Lecun, et al. We work on a given neural network which does not deliver a satisfying accuracy (around 0.5) and optimize the performance step by step. All ten features assigned in the document are involved. We achieve 98.3% accuarcy at last.
This report is structured as follows. **Section 1** introduces vectorization to the training procedure of the network. **Section 2** focuses on some methods to improve final results, involving structure adjustment, efficient gradient descent, etc. **Section 3** analyzes how to prevent from overfitting. Common ways like regularization, dropout are mentioned. **Section 4** discusses some other approaches. In **Section 5** we turn to a convolutional neural network. At the end of each section, a succinct remark is addressed to summerize.

0.2 Raw Framework

First we briefly look through demo programs. The test script runs a basic training routine. The variable *nParams* adds up all weights needed betweens layers and is divided into corresponding parts when training. Stochastic gradient descent is deployed to update weights.
In the function *MLPclassificationLoss*, $ip\{i\}$ and $fp\{i\}$ are computed in a feedforward loop. Then it comes to back propagation that conveys partial derivatives to former layers. Function *MLPclassificationPredict* derives *yhat* by model constructed before so that accuracy can be assessed. The initial test error rate is 0.473[1].

[1]Data given in this report are rounded to three digits. Each error rate is according to 1-2 experimental results. Moreover, we use test error rate to evaluate models.

1 Vectorization: Efficient Training

To obtain more iterations within acceptable time, we consider replacing some naive loops with matrix-matrix operations, which reaches Level 3 BLAS and handles over 200 FLOPs per second. Table 1 shows a significant leap in efficiency.

Iter	5000	10000	15000	20000	25000
before	3.513	3.393	3.404	3.355	3.519
after	0.905	0.812	0.858	0.836	0.871

Table 1: Time consumed every 5000 iterations (unit: s)

2 Performance Improvement

2.1 Network Structure Modification

There are two decisions to make with regard to the network structure, namely the number of hidden layers and number of neurons in each layer. As for the former, Heaton has mentioned in [1]:

> Problems that require two hidden layers are rarely encountered. However, neural networks with two hidden layers can represent functions with any kind of shape.

And for determining neuron amounts, suggestions are generally given in an empirical manner. Too few neurons will result in underfitting, while too many may cause overfitting and unreasonable training time. Here we have some rule-of-thumb methods:

- ♣ Between the size of the input layer and the output layer.
- ♣ 2/3 the size of the input layer, plus the size of the output layer.
- ♣ Less than twice the size of the input layer.

We have tested different structures and results are displayed in Table 2.

nHidden	test	valid
[100]	0.276	0.222
*[150]	0.224	0.207
[200]	0.235	0.208
[100, 100]	0.300	0.321
*[100, 150]	0.283	0.285
[150, 100]	0.323	0.326

Table 2: Test/validation error rates after 10^5 iterations

Single-layer networks seemingly outperform multi-layer ones. Also, networks with more neurons in deeper layers (such as [100, 150]) are preferred. In following sections, [150] and [100, 150] are often used, so we may see (*) in Table 2 as baselines.

2.2 Weights Initialization

Starting values of the weights can have a great effect on the training process since too large weights lead to slow learning speed [2] and small weights produce rather small gradients. We pick two structures and generate initial weights from different distributions (See in Table 3 and Table 4).

distribution	test	valid
$\mathbb{N}(0, 0.1)$	0.041	0.046
$\mathbb{N}(0, 0.01)$	0.037	0.035
$\mathbb{N}(0, 1/16)$[1]	0.042	0.039
$\mathbb{U}(-0.25, 0.25)$	0.061	0.055
*$\mathbb{U}(-0.025, 0.025)$	0.030	0.036

[1] As suggested in [2], $\sigma_w = m^{-1/2}$ where m is the fan-in.

Table 3: *nHidden* = [150], after 10^5 iterations

distribution	test	valid
$\mathbb{N}(0, 0.1)$	0.050	0.047
$\mathbb{N}(0, 0.01)$	0.058	0.053
*$\mathbb{N}(0, 1/16)$	0.037	0.040
$\mathbb{U}(-0.25, 0.25)$	0.059	0.063
$\mathbb{U}(-0.025, 0.025)$	0.045	0.048

Table 4: *nHidden* = [100, 150], after 10^5 iterations

Proper starting values optimize the performance a lot (e.g. from 0.224 to 0.030 in the first case) and the best choice (labeled by *) varies with respective structures.

Illustration: Since the improvement brought by weights initialization is quite notable, we stick with the original setting, i.e. weights from $\mathbb{N}(0, 1)$, so as to study **individual**[2] impacts of other factors (except for specific explanations).

[2]Modifications from different sections usually do not co-work.

2.3 Gradient Descent Optimization

2.3.1 Mini-batch Gradient Descent

We summarize three common types of gradient descent in Table 5. Vanilla gradient descent is so costly that Matlab fails to give answers (at least within fairish time).

type	expression	features
vanilla	$w = w - \eta \cdot \nabla_w J(w)$	redundant, global min
stochastic	$w = w - \eta \cdot \nabla_w J(w; x^{(i)}; y^{(i)})$	faster, fluctuation
mini-batch	$w = w - \eta \cdot \nabla_w J(w; x^{(i:i+n)}; y^{(i:i+n)})$	more stable[1]

[1] Compared with SGD.

Table 5: Comparison of three GD variants

We implement a mini-batch with $batchsize = 5$ and get a slightly better test error rate 0.189 ($nHidden = [150]$).

2.3.2 Momentum

SGD has trouble trapping at local optima. Momentum is introduced to help accelerate SGD in the relevant direction, or to temporally smooth out the stochastic gradient samples obtained [3]. Relevant codes are like:

```
beta = 0.9;
w0 = w;
w = w - stepSize * g + beta * (w - w0);
```

The test error rate is 0.203 ($nHidden = [150]$) and 0.301 ($nHidden = [100, 150]$).

2.3.3 Stepsize Strategies

Based on the curvature of the error surface, a small stepsize, or learning rate, can avoid divergence. On the other hand, a large learning rate ensures a reasonable speed. The algorithm below indicates an updating rule. The main idea is that we observe the error rate on the validation set every 200 iterations. If 10 continuous observations are unable to exceed an early epoch, then the learning rate is updated. Note that under this condition the final stepsize can be very close to 0. We can set a threshold and force an early stopping (See in **Section 3.2**).
The test error rate is 0.154 ($nHidden = [150]$) and 0.225 ($nHidden = [100, 150]$).

Algorithm 1 Update learning rates

$epoch = 1, count = 0, stepsize = 5e-3$
for every 200 iterations **do**
  Calculate valid error ve
  if $ve < epoch$ **then**
    $epoch = ve$
  else
    $count = count + 1$
  end if
  if count==10 **then**
    $stepsize = stepsize/2$
    $count = 0$
  end if
end for

2.4 Dataset expansion

Generating additional data using transformations may improve performance [4]. To do this, we first extract every single 16×16 character from the training set, and do affine transformations (shear) with function *imtransform*. Then we use *imresize* to adjust the size of dimension.

Algorithm 2 Affine transformation on original data (shear)

$m, t = linspace(0.05, 0.1, m)$; {$m$: expanding ratio}
for $i = 1$ to m **do**
  for every 16×16 character I in X **do**
    $tform = maketform('affine', [100; t(i)10; 001])$;
    $J = imtransform(I, tform)$;
  end for
  Expand X and y
end for

3 Avoid Overfitting

3.1 Weight Decay

A weight decay penalty term in the loss function is another common method for improving generalization. In contrast to early stopping, weight decay has the advantage of being well defined, but it is difficult to find a good parameter λ. A trick to get an approximate estimate is provided in [5].

	$nHidden = [150]$	$nHidden = [100, 150]$
$\lambda = 10^{-1}$	0.070	0.302[1]
$\lambda = 10^{-2}$	0.160	0.249
$\lambda = 10^{-3}$	0.201	0.314

[1] It is remarkable that the error rate on the validation set falls to a valley 0.105 after 30000 iterations but keeps rising afterwards.

Table 6: Test error rates over different λ, after 10^5 iterations

3.2 Early Stopping

Early stopping is a simple and widely used method to avoid overfitting. The idea is to stop training before the training set has been learned perfectly [5]. However, the criteria are sometimes ambiguous.
We apply the same stepsize strategy as that in **Section 2.3.3** and add the condition below to determine when to stop.

```
if stepSize < 1e-6
    fprintf('Early stopping called. Iter = %d\n', iter);
    break;
end
```

The test error rate is 0.159 ($nHidden = [150]$, stops at $iter = 46801$) and 0.233 ($nHidden = [100, 150]$, stops at $iter = 44801$). Though improvements hardly occur, we get close error rates with far less time.

3.3 Dropout Networks

The term "dropout" refers to temporarily removing some neurons from the network, as well as all incoming and outgoing connections. The choice of which to drop is random. We have a mask to do so:

```
p = 0.5;
dropoutmask = (rand(size(X(i, :))) < p)
```

It should be pointed out that at test time, the weights are scaled-down versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time [6].
We deploy dropout on the first layer with $p = 0.5$. The test error rate is 0.145 ($nHidden = [150]$). Considering that fewer neurons acturally work, we reconstruct the network by $nHidden = [200]$. The result goes better to 0.122. Similar improvement occurs on the multi-layer network. For $nHidden = [100, 150]$, the rate is 0.150.

4 Other Approaches

4.1 Softmax

It is often recommended to add a weight decay term to the softmax loss function, for it is not a strictly convex function. After that we have

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\{y^{(i)} = j\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right] + \frac{\lambda}{2}\sum_{i=1}^{k}\sum_{j=0}^{n}\theta_{ij}^2$$

where $1\{\cdot\}$ is the indicator function. In addition, we substitute the expression of *err* and use 0 instead of -1 in *linearInd2Binary*.
We can see the effect caused by weight decay which gives 0.154 test error rate against 0.209 ($nHidden = [150]$), and 0.238 against 0.370 ($nHidden = [100, 150]$). Both outperform squared error with weight decay (See in Table 6, $\lambda = 10^{-2}$).

4.2 A Bias for Each Layer

The bias neuron in one layer is connected to all the neurons in the next layer, but none in the previous layer and it always outputs 1. It allow us to move the threshold of an activation function.
The test error rate is 0.194 ($nHidden = [150]$) and 0.272 ($nHidden = [100, 150]$).

4.3 Fine Tuning

The motivation of fine tuning is to form a convex optimization problem on the last layer. We approach this by fixing other parameters and deploy linear regression to derive the closed form solution. After rounds of test, we have conditions and parameters in Table 7. The test error rate is 0.017.

condition	description
dataset	expanded by 7 times
initial weights	$\mathbb{U}(-0.025, 0.025)$
structure	$nHidden = [150]$
mini-batch	$batchsize = 10$
momentum	$\beta = 0.9$
stepsize	See in **Section 2.3.3**
early stopping	See in **Section 3.2**
loss function	softmax with weight decay, $\lambda = 10^{-3}$

Table 7: Settings for fine tuning

5 Convolutional Neural Networks

With regard to a convolution layer, feature maps of the previous layer are convolved with learnable kernels for weight sharing. Shortly after they are put through the activation function to form the output feature map. Convolution operations are done by *conv2* with the argument *'valid'*. Gradients for the kernel weights are computed using back propagation, except that the same weights are shared across many connections [7]. To cope with this, we implement with *rot180*.
We use six 5×5 kernels and a 2×2 subsampling layer. When 50 epochs are observed, the final test error rate reaches 0.019.

6 Conclusion

In this report we go through the ins and outs of neural networks. We discuss several methods to optimize performance, both assigned and self-studied. Efforts are mostly made on hyperparameter modification. We finally gain 98.3% accuracy at best, which is fairly good.

References

[1] Introduction to Neural Networks with Java, Heaton T. Jeff

[2] Efficient BackProp, Y. LeCun, L. Bottou, G. Orr, and K. Müller

[3] Practical Recommendations for Gradient-Based Training of Deep Architectures, Y. Bengio

[4] Effective Training of a Neural Network Character Classifier for Word Recognition, L. Yaeger, R. Lyon, B. Webb

[5] A Simple Trick for Estimating the Weight Decay Parameter, T. Rögnvaldsson

[6] Dropout: A Simple Way to Prevent Neural Networks from Overfitting, N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov

[7] Notes on Convolutional Neural Networks, J. Bouvrie