Building a Handwritten Digit Classifier using Neural Network and Optimization with Pruning and Genetic Algorithms

Xiangyou (Kevin) Zhang

Research School of Computer Science, Australian National University
U5992627@anu.edu.au

Abstract. Traditional feed-forward neural networks are off the center of the contemporary researches, but they are still instructional for beginners to learning neural networks because of their simplicity and lower difficulty of implementation. However, it is acknowledged that managing the appropriate scale of a traditional neural network is a hard problem. My report investigates a feed-forward neural network which aims to classify a set of handwritten digits. Further, some optimization methods which aim to find out the best scale of a neural network are implemented and discussed. The methods used in the report are network pruning on distinctiveness and genetic algorithms.

Keywords: Feed-forward, neural network, handwritten digit classification, distinctiveness, genetic algorithms

1 Introduction

In this paper, I devise a feed-forward neural network to recognize and classify handwritten digits. However, it is always hard to determine an appropriate size of a feed-forward neural network, as the internal hidden neurons are implicit to observe and track. Thus, I further investigate two prominent approaches for choosing the right size of a neural network. The one is pruning on distinctiveness of hidden units, which is done after a network was trained or in the training process. The other is incorporating genetic algorithms (GA) for training on small dataset before the actual training starts. GA, imitating biological genes and their behavior, can evolve and choose fittest offspring.

The training data set is collected from UCI machine learning repository [3]. The data set contains 1593 handwritten digits from around 80 persons by scanning, and are all labeled. The image information is cropped and resized to 16 by 16 black-white format, and flattened into an array of length 256. The labels are in the format which is an array where the length is the total number of all classes and the index of "1" represents which class the label means.

As I observe the data set, digits of the same class are usually cluster together, which is to say, for example, the first to 20th pieces of data are all digit "0". The sequence of the data from the same class may cause the network training to incline to a particular result, thus I use some methods to avoid the situation.

My report contains the methods I used for data preparation, network construction and algorithms details. Results are also compared and analyzed in the report. In addition, I introduce Buscema's result [2] on the same data set and compare to my model with the sized determined by the pruning and GA algorithms.

2 Methods

This section describes my process of designing and implementing my neural network. I set up a two-layer (one hidden layer) network using pytorch, read data, and train the model. Before the actual training, I manually alter some parameters to decide the proper values I should use later, such as number of hidden layer nodes, learning rate, epoch, etc. Then, I explore "distinctiveness" which compares hidden units' similarity, and the redundant similar units will be cut off to reduce the network size. Subsequently, I use GA to determine the network size.

2.1 Network construction

As the input data set provides black-white image data of 256 pixels, the dimension of input layer is naturally set to 256. Also, the goal is to classify the digit to "0" ~ "9", thus the output layer dimension should be 10. A node of a layer is fully connected to all other nodes of the previous/next layer.

Softmax is used on output layer, as softmax is suitable for a N-class problem. Softmax is defined as equation (1):

$$\text{softmax}(x) = \exp(x_i) \,/\, \sum_j \exp(x_j) \quad , (1)$$

which rescales elements in range (0, 1) and sum to 1.

In the back propagation, I use cross entropy as loss function, which is useful when training a classification problem with N classes. And the classic stochastic gradient descent is used as weight optimizer.

In terms of training parameters, I have pre-run several tests to find out appropriate learning rate, number of epochs and dataset structure before I actually started.

2.2 Data pre-process

I mentioned in previous section that the given data set has a very particular order which may not good for my training. Therefore, I wrote some functions to read the data from file, split the features and labels, and group each row of data by the label ("0" ~ "9"). Besides, items of data are **shuffled** in their groups.

In the training stage, I have two types of the data set:

a) **Sequential**: strictly ordered from "0" to "9" and looped over again:
   e.g.: **"0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 …"**
   using function: `get_sequential_data(dataset)`;

b) **Batch**: randomly ordered and grouped in a number of batches:
   e.g.: batch 1 "7, 4, 5, 2, 3…", batch 2 "0, 2, 5, 4, 1, 1…" …
   using function: `get_batched_data(dataset, batch_num)`

2.3 Validation and testing

I divide my data set which has been shuffled beforehand into two sub sets. The one is for training and the other is for testing. The two sets will have little similarity which can deduce whether the model classifies well on new data. I use validation which inputs training data to see how well the model has learnt, and test the model with unseen test data set. The most important criterion I check is the percentage of correct output produced by the model on training set and test set.

2.4 Network pruning

The problem that always bothers the implementation of traditional feed-forward networks is choosing appropriate network structures. The networks with too many redundant units may influence its training efficiency. The paper [1] introduces a range of methods of finding unnecessary hidden units, such as relevance, contribution, distinctiveness, etc. My implementation uses distinctiveness to find the units that is too similar to others. In the end of a run, the unit that has highest similarity to others will be removed, and its weights merge to one of other similar units. The pruning process will proceed until there are no more similar units. The model accuracy will be checked again to see if the procedure has negative effect on the model.

To calculate distinctiveness, I feed in 10 different patterns (10 different digits) and obtain the activations of the hidden layer. Activations form a vector which is offset to 0.5 to ensure the angular range is (0, 180) degrees. Then the activation vector is normalized and dot produced to itself transposed, which calculates the cosine similarity. If the value is greater than $\cos 15°$, it is seen as too similar.

2.5 Genetic Algorithms

There is also an approach to identifying the proper size of a network using genetic algorithms. Other than the pruning discussed earlier, GA is implemented before the actual training starts. GA imitates biological gene's behavior and usually contains a serial of procedures, such as initializing population, fitness evaluation, selection and reproduction. In the reproduction, a programmatic "chromosome" may crossover with others to generate new chromosome with good feature inherited and probably some good or bad mutations.

In order to find the proper number of hidden units, I was intended to use a **binary representation** for chromosome which has seven bits to represent a number ranging from 1 to 128 (as the input is 256-dim, I do not think much higher is necessary). Another important part of GA is the fitness function. I designed a pure heuristic fitness function for my particular case. The function takes test accuracy, stableness of result (the standard deviation) and overfitting into consideration. That is to say, the function thinks that if a model is of high accuracy, stable in training and low overfitting risk, it is considered as a good model. In the fitness function, each population will generate an NN model with the size it represents and train the model repetitively for several times looking for stableness (model will be re-initialized in repetition). This process is very costly.

The procedures I implemented for GA are:

a) **Initializing population**: Randomly initializing a list of chromosomes given a population size
b) Getting **fitness values** for each population in the list
c) **Selecting** higher fitness value population into a new list
d) **Reproducing** the populations in pair by sexual **crossover**: using uniform crossover
e) **Mutating** the population and **replace** its parent: random mutation on a mutation rate

3 Results and Discussion

3.1 Network pruning

First of all, I attempted to test the model with the hidden unit number 16, 32, 64, 128 and 15 degrees as threshold (other parameters: 8 batches 100 each, 1000 epochs, learning rate 0.1). I surprisingly found that every unit was distinct from others in all these cases. However, with more hidden units, the model is more likely to be reliable as the testing accuracy rate will be more stable.

In order to see more significant result, I had to increase the threshold to 24 degrees for experimental purpose. I used 128 hidden units, 24 degrees of threshold, and other parameters remained the same. The training data set was the first 800 (pre-shuffled) batched pieces of data, whereas the test data set was the rest. In a loop, similar units were gradually removed until there were no more similar units found. **Table 3.1** demonstrates the result.

The result reveals that 5 units were reduced, but the model correctness is degrading slightly with the pruning. I also tried several runs with the same condition, but the results were a bit unpredictable and randomized. Sometimes unit removal had drastic decrease on correctness, sometimes the correctness went down and then increased.

As a result, the network pruning with distinctiveness may not very suitable for reducing the size of my model for handwritten digit classification. The empirical threshold 15 degrees failed to find similarity, but higher threshold might deteriorate the correctness.

Table 3.1 Reducing hidden units from 128

LOOP	SIMILAR UNITS FOUND	DISTINCTIVENESS (COSINE SIMILARITY)	TEST CORRECTNESS BEFORE UNIT REMOVAL	TEST CORRECTNESS AFTER UNIT REMOVAL	UNITS REMAINED
NO.1	(7, 15)	0.978	0.913	0.908	127
NO.2	(30, 124)	0.952	0.913	0.904	126
NO.3	(71, 95)	0.934	0.913	0.903	125
NO.4	(69, 92)	0.928	0.913	0.900	124
NO.5	(14, 17)	0.922	0.913	0.898	123

3.2 Genetic Algorithms

As there are too many costly trainings involved in the fitness function, for the sake of saving time, I used first 200 pieces of a sequential data set for training and 200 ~ 400 for test. The sequential data was shuffled in pre-processing grouping stage, so it is unlikely that similar samples are clustered. In addition, I used sequential data rather than batch data and reduced the epochs to 500.

Here are GA parameters I chose:

- DNA size: 7 (7 bits representing 1 ~ 128)
- Population size: 10
- Crossover rate: 0.5
- Mutation rate: 0.002
- Number of generations: 20 (the program will run 20 iterations and stop)

Here is the result:

```
1/20 Generations
   calculating fitness for pop: [18, 112, 65, 71, 66, 41, 14, 103, 9, 122, 113, 70, 23, 53, 77, 34, 85, 89, 79, 87] ...
   Most fitted DNA: [1 0 0 0 1 1 0] (dec: 71)

2/20 Generations
   calculating fitness for pop: [79, 74, 65, 103, 70, 93, 66, 85, 74, 49, 21, 5, 41, 70, 71, 85, 89, 18, 77, 85] ...
   Most fitted DNA: [1 0 0 0 1 1 0] (dec: 71)

3/20 Generations
   calculating fitness for pop: [85, 41, 79, 85, 89, 41, 49, 71, 74, 66, 82, 89, 74, 71, 74, 49, 95, 115, 73, 41] ...
   Most fitted DNA: [1 1 1 0 0 1 0] (dec: 115)

4/20 Generations
   calculating fitness for pop: [73, 95, 87, 85, 74, 79, 81, 74, 74, 115, 69, 82, 73, 74, 82, 41, 89, 93, 82, 41] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

5/20 Generations
   calculating fitness for pop: [93, 89, 73, 42, 73, 85, 77, 121, 82, 89, 41, 82, 82, 74, 74, 93, 81, 73, 81, 90] ...
   Most fitted DNA: [1 0 1 1 0 0 1] (dec: 90)

6/20 Generations
   calculating fitness for pop: [90, 89, 81, 90, 77, 81, 82, 82, 73, 73, 81, 73, 90, 82, 82, 77, 41, 73, 90, 82] ...
   Most fitted DNA: [1 0 1 1 0 0 1] (dec: 90)

7/20 Generations
```

```
   calculating fitness for pop: [73, 90, 73, 73, 73, 90, 82, 82, 89, 90, 73, 65, 89, 81, 73, 89, 90, 82, 77, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

8/20 Generations
   calculating fitness for pop: [82, 82, 73, 82, 73, 89, 90, 90, 90, 73, 90, 73, 82, 90, 90, 82, 73, 81, 90, 73] ...
   Most fitted DNA: [1 0 1 0 0 0 1] (dec: 82)

9/20 Generations
   calculating fitness for pop: [73, 73, 82, 90, 73, 82, 89, 90, 89, 73, 81, 73, 73, 90, 90, 90, 73, 74, 73, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

10/20 Generations
   calculating fitness for pop: [89, 73, 73, 89, 73, 89, 73, 73, 73, 73, 73, 90, 73, 81, 74, 73, 73, 90, 73, 90] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

11/20 Generations
   calculating fitness for pop: [89, 73, 73, 74, 89, 73, 73, 73, 73, 90, 90, 81, 73, 73, 73, 77, 73, 73, 73, 73] ...
   Most fitted DNA: [1 0 1 0 0 0 0] (dec: 81)

12/20 Generations
   calculating fitness for pop: [73, 73, 73, 73, 73, 73, 90, 77, 73, 73, 73, 73, 90, 73, 73, 73, 73, 77, 73, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

13/20 Generations
   calculating fitness for pop: [73, 90, 73, 73, 74, 73, 73, 73, 74, 89, 73, 73, 73, 90, 73, 73, 90, 73, 73, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

14/20 Generations
   calculating fitness for pop: [73, 73, 73, 74, 90, 90, 73, 73, 89, 74, 73, 90, 73, 73, 73, 73, 73, 73, 89, 73] ...
   Most fitted DNA: [1 0 1 1 0 0 0] (dec: 89)

15/20 Generations
   calculating fitness for pop: [90, 73, 73, 73, 73, 89, 89, 73, 73, 73, 89, 73, 73, 73, 74, 73, 73, 73, 74, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

16/20 Generations
   calculating fitness for pop: [74, 73, 74, 73, 73, 73, 73, 74, 73, 73, 73, 73, 74, 74, 73, 89, 73, 73, 73, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

17/20 Generations
   calculating fitness for pop: [73, 73, 89, 74, 89, 74, 73, 73, 89, 74, 73, 73, 73, 89, 74, 73, 73, 73, 73, 74] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

18/20 Generations
   calculating fitness for pop: [73, 73, 89, 74, 74, 73, 73, 66, 73, 73, 74, 74, 73, 73, 73, 74, 74, 73, 73, 74] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

19/20 Generations
   calculating fitness for pop: [73, 73, 89, 73, 89, 74, 73, 73, 74, 73, 74, 74, 74, 73, 74, 73, 66, 74, 74, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 0] (dec: 73)

20/20 Generations
   calculating fitness for pop: [73, 73, 89, 73, 74, 74, 73, 73, 73, 73, 73, 73, 74, 89, 73, 66, 73, 73, 74, 73] ...
   Most fitted DNA: [1 0 0 1 0 0 1] (dec: 74)

Final fitted DNA: [1 0 0 1 0 0 1] (dec: 74)
```

The program took about 15 minutes to run. Finally, the result converged at around 73 and claimed that 74 is the best size for the model on the handwritten digit classification problem. However, the result of model remains highly random as when I looked into the intermediate outputs (which too long to show in the report), there was a significant variance over the fitness values of models with the same size 73. I presume that it is because the neural network training process involves too much randomness which is difficult to control, for example the initial weights are set randomly. Also, I suspect that the number 73 was chosen mainly by chance rather than the domain of problem itself, because the binary representation of chromosomes does not seem to directly reflect the features of the model, thus the crossover and the mutation are not as natural as a biological chromosome's.

3.3 Final classification result and Comparison to Buscema'

In the last stage, I took the best number of hidden units, which was 73 found by the GA and ran the training and test over the entire data set. The training set was 800 pieces of data in 8 batches, while the test set contained the rest 793 pieces of data. Number of epochs and learning rate were set to 500 and 0.01 respectively. The training completed quickly in several second. The test correctness for each digit are demonstrated separately in **Table 3.2**, as well as the overall value.

Table 3.2 Result of model with 73 hidden units classifying ten digits

NUMBERS	ZERO	ONE	TWO	THREE	FOUR	FIVE	SIX	SEVEN	EIGHT	NINE	OVERALL
CORRECTNESS	0.963	0.939	0.861	0.911	0.914	0.911	0.963	0.897	0.947	0.923	0.923

Additionally, the pruning process was implemented on the model as well, cut back off two redundant units from the hidden layer and maintained the final accuracy at 0.923. The threshold was set to 24 degrees. The result is in **Table 3.3**.

Table 3.3 Reducing hidden units from 73

LOOP	SIMILAR UNITS FOUND	DISTINCTIVENESS (COSINE SIMILARITY)	TEST CORRECTNESS BEFORE UNIT REMOVAL	TEST CORRECTNESS AFTER UNIT REMOVAL	UNITS REMAINED
NO.1	(14, 58)	0.945	0.923	0.922	72
NO.2	(34, 70)	0.924	0.923	0.923	71

Buscema [2] used a technique called independent judge with MetaNet on the classification problem with the same handwritten digit data set. The method can change the credibility of a classification problem. **Table 3.5** is the results. It compares four different ANN with MetaNet and proves that MetaNet outweighs any of them. Nevertheless, my ANN with back propagation works even better than their solution. It might be caused by different parameters set or training approaches. Their BP ANNs were provided with 2 layers of Hidden Units of 40 units each.

Table 3.4 Results obtained during the Testing phase of the 4 ANNs and of the MetaNet systems.

Numbers	LP	BP	LVQ	SQ	METANET
Zero	90.00%	96.25%	96.25%	92.50%	96.25%
One	85.19%	88.89%	95.06%	86.42%	95.06%
Two	92.50%	88.75%	91.25%	86.25%	95.00%
Three	87.34%	88.61%	89.87%	83.54%	91.14%
Four	74.07%	82.72%	91.36%	83.95%	88.89%
Five	93.67%	89.87%	96.20%	92.41%	97.47%
Six	88.89%	95.06%	95.06%	96.30%	95.06%
Seven	86.08%	79.75%	91.14%	83.54%	91.14%
Eight	93.51%	83.12%	88.31%	85.71%	93.51%
Nine	79.75%	87.34%	78.48%	87.34%	87.34%
Total Right %	87.06%	88.07%	91.33%	87.81%	93.09%

Apparently, my result generally is a little bit better than Buscema's. However, it is noticeable that both models have the same problem that they perform better on "0" and "6" but less well on "2" and "7". This may be caused by the data presentation itself, as "7" sometime are written pretty similarly to "1", but "0" and "6" are usually very distinct. This may enforce us to think about doing some preprocesses, such as encoding or CNN.

4 Conclusion

Although my model had achieved significant success in handwritten digit classification, the exploration to optimizing the scale of networks was not noticeably effective. Network pruning works faster but hardly prunes hidden units that are necessary to rule out in my case, whereas GA seems more reliable as it had found a proper number, but its time consumption is surprisingly high. Besides, designing the fitness function and the chromosome presentation is crucial to the performance of GA, but there is still space of improvement on what I have done. To wrap up, training and managing a feed-forward neural network remains difficult, as there are too many uncertainties and the results always change unexpectedly. I can understand why people are abandoning traditional neural networks and turning to deep learning.

In the future, I may look into CNN for this sort of problems, since the inputs of the data set are actually images. Right now, the model can only handle very small images in a particular format. I hope to extend it to be able to deal with more arbitrary images. GA is also an interesting field that I can have deeper investigation. I may need to design a better representation for a network chromosome which stands for more actual features and treat of the network. And the fitness function can be more informative and instructive.

Reference

1. Gedeon and D. Harris. " Network reduction techniques.".
2. Buscema, Massimo. "Metanet*: The theory of independent judges." Substance use & misuse 33.2 (1998): 439-461.
3. Tactile Srl, Brescia. "Semeion Handwritten Digit Data Set." UCI Machine Learning Repository: Semeion Handwritten Digit Data Set, 1994, archive.ics.uci.edu/ml/datasets/Semeion Handwritten Digit.