Report on optical recognition of handwritten digits coursework

Michael Bausano

Middlesex University London

Author Note

Middlesex University London Computer Science student with student number M00596333 and email address bausanomichal@gmail.com

Input transformation

I have read a few interesting articles about convolutional neural networks. The concept of the filters inspired me to transform the input slightly. I have mapped the original 64 input parameters over two filters that detect horizontal and vertical edges, resulting in input with the size of $64 + 2 * 64 = 192$. This has improved the success rate by approximately **0.3 % - 0.5 %** varying based on the algorithm.

How to run

Each algorithm runs by in a different thread. By default, all algorithms run. If you'd like to run only one, feel free to comment out any of the last three lines in the *main* method. Here is an example output on the testing data:

```
> NEAREST NEIGHBOUR
Correctly classified 2764 out of 2810 (98.36 %).
> NEURAL NETWORK
Correctly classified 2746 out of 2810 (97.72 %).
> ESTIMATOR
Correctly classified 2771 out of 2810 (98.61 %).
```

Nearest Neighbour

The nearest neighbour was an obvious choice for an algorithm to implement as it is very straightforward, fairly fast on the given data set and yields baseline outputs.

The idea behind this algorithm is to find the shortest Euclidian distance to the given input digit in the previously seen data set. It runs in *O(n)* as it compares the input to each training data set digit, where *n* is the number of the training digits (which means the space complexity is *O(n)* as well).

To find the distance from one member to another, you sum squared difference in each point:

$$\sum_{n=0}^{POINTS} (a_n - b_n)^2$$

Where *POINTS* is the length of the points that define each member. To be precise, to calculate the distance the result should be square rooted, however, we just need to compare that result, therefore this computation is redundant.

We always store the class and distance of the neighbour (member of the training data set) that was closest to the input digit. After we loop through all data, we return the closest class.

I have implemented a *k* parameter, which let's researcher define how many of the fittest neighbours get to vote on the result. From my cross-fold validation experiments, this parameter is best to be 1.

This algorithm achieved a **98.36 % success rate** on the provided testing data.

Multi-Layer Perceptron

The first algorithm I have implemented was a basic feedforward neural network with backpropagation algorithm that trains it.

Feeding forward the values through the network is trivial vector folding. We only have to make sure we map the net of each neuron through an activation function.

I have read an interesting paper on learning rate called "Cyclical Learning Rates for Training Neural Networks". I made the learning rate to cycle in linear intervals plus it descends linearly with each iteration.

The weights are initialized randomly for each neuron in interval $<-0.25; 0.25)$. From what I have understood from my research, the bias can be initialized to 0 as the activation potential is always 1, therefore it will get trained to desired value nonetheless.

To fight the fading gradient problem, I have implemented an iterative layer cloning. After each training iteration, the last hidden layer is cloned. This improved the success rate, however drastically raised training time. Since the success rate improvement was not very prominent, in the build I am submitting this feature is turned off.

I have used the sigmoid activation function. My attempts to introduce ReLU were not very successful, it always performed much worse. See link https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ for a more detailed explanation of the chain rule and the math behind the algorithm.

In order to pick the best performing hyperparameters, I have implemented easy cross-validation to my data set handling class. You can split the input set n into two parts with parameter k which splits the data into training data of size $n * (k - 1) / k$ and validation set of size

n / k. When picking the hyperparameters, I was using the validation set. After I have fine-tuned the algorithms, I have used the testing data to report the success rates for this report.

My multi-layered perceptron achieves around **96.30 % - 98.00 % success rate**.

Estimator

The last algorithm I have implemented was a combination of MLP and NN. I have split the digits into two groups and bootstrapped two MLP with 6 output neurons (5 digits neurons and 1 IDK neuron). Both instances of MLP and one instance of NN computed their estimations for each class. Results were added and the digit with the highest probability was picked.

The IDK neuron ("I Don't Know") means that the network does not recognise the digit. Therefore, it must be in the other group. Hence the IDK neuron of network *n* contributes to probabilities of all classes that are not in *n* set.

The digits were split into two groups of 5 based on confusion matrix analysis. I have generated dozens of confusion matrices for both MLP and NN. After seeing which digits are often misrecognized, I have created two following groups:

- 2, 4, 5, 7, 8
- 0, 1, 3, 6, 9

This approach yielded slightly better results than NN, around **98.50 % - 98.90 %**.