DLNN 2020: Assignment 1

Wojtek Kowalczyk
wojtek@liacs.nl

18 February 2020

Introduction

The objective of this assignment is to develop and evaluate several algorithms for classifying images of handwritten digits. You will work with a simplified version of the famous MNIST data set: a collection of 2707 digits represented by vectors of 256 numbers that represent 16x16 images. The data is split into a training set (1707 images) and a test set (1000 images). These data sets are stored in 4 files: `train_in.csv`, `train_out.csv`, `test_in.csv`, `test_out.csv`, where `_in` and `_out` refer to the input records (images) and the corresponding digits (class labels), respectively. These files are stored in `data.zip`.

You may find more information about the original problem of handwritten digit recognition, more data sets, and an overview of accuracies of best classifiers (it is about 99.6%!) at `http://yann.lecun.com/exdb/mnist/`.

Task 1: Analyze distances between images

The purpose of this task is to develop some intuitions about clouds of points in highly dimensional spaces. In particular, you are supposed to develop a very simple algorithm for classifying hand-written digits.

Let us start with developing a simple distance-based classifier. For each digit d, $d = 0, 1, \ldots, 9$, let us consider a cloud of points in 256 dimensional space, C_d, which consists of all *training* images (vectors) that represent d. Then for each cloud C_d we can calculate its center, c_d, which is just a 256-dimensional vector of means over all coordinates of vectors that belong to C_d. Once we have these centers, we can easily classify new images: to classify an image, calculate the distance from the vector that represents this image to each of the 10 centers; the closest center defines the label of the image. But first let us take a closer look at our data.

For each cloud C_d, $d = 0, 1, \ldots, 9$, calculate its center, c_d, and the radius, r_d. The radius of C_d is defined as the biggest distance between the center of C_d and points from C_d. Additionally, find the number of points that belong to C_d, n_d. Clearly, at this stage you are supposed to work with the training set only.

Next, calculate the distances between the centers of the 10 clouds, $dist_{ij} = dist(c_i, c_j)$, for $i, j = 0, 1, \ldots 9$. Given all these distances, try to say something about the expected accuracy of your classifier. What pairs of digits seem to be most difficult to separate?

Task 2: Implement and evaluate the simplest classifier

Implement the simplest distance-based classifier that is described above. Apply your classifier to all points from the training set and calculate the percentage of correctly classified digits. Do the same with the test set, using the centers that were calculated from the training set. In both cases, generate a *confusion matrix* which should provide a deeper insight into classes that are difficult to separate. A confusion matrix is here a 10-by-10 matrix (c_{ij}), where c_{ij} contains the percentage (or count) of digits i that are classified as j. Which digits are most difficult to classify correctly? For calculating and visualising confusion matrices you may use the `sklearn` package. Describe your findings. Compare performance of your classifier on the train and test sets. How do the results compare to the observations you've made in Step 1? How would you explain it?

So far we assumed that the distance between two vectors is measured with help of the Euclidean distance. However, this is not the only choice. Rerun your code using alternative distance measures that are implemented in `sklearn.metrics.pairwise.pairwise_distances`. Which distance measure provides best results (on the test set)?

Task 3: Implement a multi-class perceptron algorithm

Implement (from scratch) a multi-class perceptron training algorithm (from slide 36, second lecture) and use it for training a single layer perceptron with 10 nodes (one per digit), each node having 256+1 inputs and 1 output. Train your network on the train set and evaluate on both the train and the test set, in the same way as you did in the previous steps. As your algorithm is non-deterministic (results depend on how you initialize weights), repeat your experiments a few times to get a feelinig of the reliability of your accuracy estimates.

Try to make your code efficient. In particular, try to limit the number of loops, using matrix multiplication whenever possible. For example, append to your train and test data a column of ones that will represent the bias. The weights of your network can be stored in a matrix W of size 257x10. Then the output of the network on all inputs is just a dot product of two matricies: $Train$ and W, where $Train$ denotes the matrix of all input vectors (one per row), augumented with 1's. To find the output node with the strongest activation use the `numpy argmax` function. An efficient implementation of your algorithm shouldn't take more than a few seconds to converge on the training set (yes, the training sets consists of patterns that are linearly separable so the perceptron algorithm will converge).

Finally, notice that the perceptron algorithm that is presented in the Géron's textbook on page 287 (Equation 10-3) is slightly different than the algorithm discussed during the lecture (slide 36). Accordingly, slightly modify your code and rerun your experiments (assume η to be 1). Compare the results (accuracy, convergence speed, runtime).

Task 4: Linear Separability

What do you think: is the set of all images of the digit 1 (from the training set) linearly separable from the set of all images of the digit 7? More generally, are images of the digit i linearly separable from the images of the digit j, for i, j in $\{0, 1, \ldots, 9\}$, such that $i \neq j$? To answer these questions study the Cover's Theorem, think about the answers, and then implement the simple perceptron algorithm and apply it to all pairs of sets of images from classes i and j. (Hint: don't run your algorithm for more than a few hundred iterations).

And can you linearly separate the set of all images of i (for i in $\{0, 1, \ldots, 9\}$ from all remaining images? (This time you might need to use a few thousands iterations.)

Task 5: Implement the XOR network and the Gradient Descent Algorithm

This is probably the last time in your life that you are asked to implement a neural network from scratch – therefore, have fun! Proceed as follows:

1. Implement the function $xor\_net(x_1, x_2, weights)$ that simulates a network with two inputs, two hidden nodes and one output node. The vector $weights$ denotes 9 weights (tunable parameters): each non-input node has three incoming weights: one connected to the bias node that has value 1, and two other connections that are leading from the input nodes to a hidden node or from the two hidden nodes to the output node. Assume that all non-input nodes use the sigmoid activation function.

2. Implement the error function, $mse(weights)$, which returns the mean squared error made by your network on 4 possible input vectors $(0,0), (0,1), (1,0), (1,1)$ and the corresponding targets: $0, 1, 1, 0$.

3. Implement the gradient of the $mse(weights)$ function, $grdmse(weights)$. Note that the vector of values that are returned by $grdmse(weights)$ should have the same length as the input vector $weights$: it should be the vector of partial derivatives of the mse function over each element of the $weights$ vector.

4. Finally, implement the gradient descent algorithm:

   (a) Initialize $weights$ to some random values,

   (b) Iterate: $weights = weights - \eta \cdot grdmse(weights)$,
   where η is a small positive constant (called "step size" or "learning rate").

Use your program to train the network on the XOR data. During training, monitor two values: the MSE made by your network on the training set (it should be dropping), and the number of misclassified inputs. (The network returns a value between 0 and 1; we may agree that values bigger than 0.5 are interpreted as "1", otherwise as "0".)

Run your program several times using various initialization strategies and values of the learning rate. You may experiment with alternative activation functions, e.g., hyperbolic tangent, $tanh$, or a linear rectifier, $relu(x) = max(0, x)$.

Additionally, try the "lazy approach": just keep generating random weights of the network, testing if it computes the XOR function, and stop as soon as you've found such weights. To get an idea how many sets of weights should be tried before finding a good one repeat your experiment several times. Describe your work and findings in a report.

Organization

Your submission should consist of a report in the pdf format (at most 10 pages) and neatly organized code that you've used in your experiments (but no data) so we (or others) could reproduce your results. Submit your work via the Blackboard - details of the submission procedure will be announced soon.

The deadline for this assignment is **Sunday, 15th March, 23:59**.