Neural Networks 2019: Assignment 1

Wojtek Kowalczyk
`wojtek@liacs.nl`

12 February 2019

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; select as a label the closest one. But first let us take a closer look at out data.

For each cloud C_d, $d = 0, 1, \ldots, 9$, calculate its center, c_i, and the radius, r_i. 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_i, n_i. 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 was 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 were classified as j. Which digits were 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 was 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 Bayes Rule classifier

Now approach the problem in the same way as in the `"a" or "b"?` example that was discussed during the course. To keep things simple, limit yourself to images of two digits only, for example, `5` and `7`. Which feature do you want to use to discriminate between the two classes? Implement your feature, apply it to the training data, discretize it and create corresponding histograms. Finally, calculate the terms $P(X|C)$ and $P(C)$ so you could apply the Bayes rule to find $P(C|X)$ over the training set. Next, apply your Bayes classifier to classify all cases from the test set. What accuracy have you achieved?

Task 4: Implement a multi-class perceptron algorithm

Implement (from scratch) a multi-class perceptron training algorithm and use it for training a single layer perceptron with 10 output 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.

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; then weights of your network can be stored in a matrix W of size 257x10. Then the output of the network on an input x is just a dot product of two vectors. To find the output node with the strongest activation use the `numpy argmax` function. You can get some extra hints by studying the blog `https://medium.com/@thomascountz/19-line-line-by-line-python-perceptron-b6f113b161f3` .

Task 5: Implement the Gradient Descent Algorithm

Study `http://neuralnetworksanddeeplearning.com/chap2.html` – the second chapter of the book of Michael Nielsen. You will notice that the plain implementation of the Gradient Descent algorithm might be very simple (although inefficient): using the equation (46) the gradient of a (differentiable) function can be implemented just in a few lines. Indeed, if we consider a (differentiable) function of, say, three variables, $f(x_1, x_2, x_3)$, we can approximate the gradient of f at x_1, x_2, x_3 by calculating values of:

$$(f(x_1 + \epsilon, x_2, x_3) - f(x_1, x_2, x_3))/\epsilon$$

$$(f(x_1, x_2 + \epsilon, x_3) - f(x_1, x_2, x_3))/\epsilon$$

$$(f(x_1, x_2, x_3 + \epsilon) - f(x_1, x_2, x_3))/\epsilon$$

where ϵ is a very small constant, e.g, $\epsilon = 10^{-3}$. Using this trick, implement an algorithm that can be used to train a simple network for the XOR-problem. 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. Vector $weights$ denotes 9 weights (tunable parameters): each non-input node has there incoming weights: one connected to the bias node that has a value 1, and two other connections that are leading from the input nodes to a hidden node or from the 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 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 value that is 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 component 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, $relu(x) = max(0, x)$. Describe your findings.

Optionally, modify your code so it could be applied to the MNIST data used Task 1. For example, try a network with 256 input nodes, 30 hidden nodes and 10 output nodes. How fast is your implementation? Can you obtain some interesting results?

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.

More details about the submission procedure and evaluation criteria will follow soon.

The deadline for this assignment is **Wednesday, 13th March, 23:59**.