Assignment 1
Introduction to Deep Learning

September 2020

1 Introduction

The objective of this assignment is to develop and evaluate several algorithms for classifying images of handwritten digits and designing your own neural network from scratch for the XOR problem. 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: Data dimensionality, distance-based classifiers

The purpose of this task is to develop some intuitions about clouds of points in high-dimensional spaces. In particular, you are supposed to use dimensionality reduction techniques to visualize your data, develop a very simple algorithm for classifying hand-written digits and compare it to another distance-based classifier.

1. For each digit $d, d = 0, 1, ..., 9$, let us consider a cloud of points in 256 dimensional space, C_d, which consists of all training images (vectors) that represent d. 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: by calculating the distance from the vector that represents this image to each of the 10 centers, the closest center defines the label of the image. Next, calculate the distances between the centers of the 10 clouds, $dist_{ij} = dist(c_i, c_j)$, for $i, j = 0, 1, ...9$. 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?

2. Experiment with three dimensionality reduction algorithms: PCA, LLE, t-SNE and apply them to the MNIST data to generate a visualization of the different classes, preferably in 2D. You are free to use any library to do this, however *sklearn* contains all the necessary methods.

   Does the visualization agree with your intuitions and the between-class distance matrix $dist_{ij}$?

3. Implement the simplest *distance-based classifier* that is described in part **1**. 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.

4. A less naive distance-based approach is the KNN (K-Nearest-Neighbor) classifier (you can either implement it yourself or use the one from *sklearn* package). By using this method repeat the same procedure as in part **3**. Then, for both classifiers 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? Again, for calculating and visualising confusion matrices you may use the *sklearn* package. Describe your findings, compare performance of your classifiers on the train and test sets.

Task 2: 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 (inputs and bias) 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 task. As your algorithm is non-deterministic (results depend on how you initialize weights), repeat your experiments a few times to get a feeling 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 matrices: $Train$ and W, where $Train$ denotes the matrix of all input vectors (one per row), augmented with 1's (biases). 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 set consists of patterns that are linearly separable so the perceptron algorithm will converge).

How does the accuracy of this single layer multi-class perceptron compare to the distance based methods in task 1?

Task 3: 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 $xor\_net(x1, x2, weights)$ 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: $weights = weights - \eta * grdmse(weights)$,

   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 obtained by your network on the training set, 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. 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 have 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 the report.

You may experiment with alternative activation functions, e.g., hyperbolic tangent (tanh) or a linear rectifier, $relu(x) = max(0, x)$. How do they affect the training process of your network, how would you explain these differences?

Submission

Your submission should consist of a report in *pdf* format (at most 5 pages) and neatly organized code (either Jupyter notebook or *.py* scripts) that you've used in your experiments so that we could reproduce your results. You can find more information about report writing on Brightspace → Assignments section.

Submit your work via Brightspace, all the files should be contained in a single *.zip* (without the MNIST data files).

Deadline: 18th October, 23:30PM