Two methods for handwritten digit classification

Xavier Servot *

https://github.com/Xavier0301/HandwrittenDigitClassification

We want to build a function $C\colon \mathbb{R}^{28\times 28} \mapsto \{0, \dots, 9\}$ that assigns the digit corresponding to an image of size 28×28 of a handwritten digit.

We are facing what we call a classification problem, and there are multiple algorithms that can can help us achieve an acceptable classification, i.e. a classification that assigns the correct digit to the fed image a good portion of the time.

In the two approaches discussed in this document, we are going to use the MNIST dataset of handwritten digits images and their corresponding digits.

1 The k-nearest neighbors approach

1.1 Intuition

The intuition behind this approach goes as follows:

- We load the dataset in memory. (parsing the dataset, converting pixel values...). We won't be going over this part here.
- For a given image im to classify, we
  - Compute the distances between the given image and every image in the dataset. We keep this data as an array of pairs (distance to im, index)

*EPFL

  - We sort the array by increasing distances. We will explain what we mean by "distance" between two images.
  - We keep the index of the first k images after sorting
  - We determine which digit l is the most frequent among those k. Remember that we know the labels corresponding to the images.

- We output l: $C(im) = l$.

1.2 Distances

So knowing this, we know that this approach is going to heavily rely on the concept of distance between two images. Here, a distance d has to be defined on the space of all possible images:

$$d: \begin{array}{c} \mathbb{R}^{28\times 28} \times \mathbb{R}^{28\times 28} \to \mathbb{R} \\ (im_1, im_2) \mapsto d(im_1, im_2) \end{array}$$

and is going of course to designate if two images are far away from each other. The two distances that will be presented are:

- The euclidean distance in $\mathbb{R}^n$:

$$d_{euclidean}((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = (\sum_{i=1}^{n} (y_i - x_i)^2)^{\frac{1}{2}}$$

  so for images $\mathbf{a} = (a_1, \ldots, a_{784}), \mathbf{b} = (b_1, \ldots, b_{784})$, we have

$$d_{euclidean}(\mathbf{a}, \mathbf{b}) = (\sum_{i=1}^{784} (b_i - a_i)^2)^{\frac{1}{2}}$$

  In this algorithm, we will only use this distance to compare it to other, thus we don't have to incur the cost of applying the square root and we will use the square euclidean distance

$$d_{squared}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{784} (b_i - a_i)^2$$

- The inverted similarity SI, using the same notation:

$$SI(\mathbf{a}, \mathbf{b}) = 1 - \frac{\sum_{i=1}^{784}(a_i - \overline{\mathbf{a}})(b_i - \overline{\mathbf{b}})}{((\sum_{i=1}^{784}(a_i - \overline{\mathbf{a}})^2)(\sum_{i=1}^{784}(b_i - \overline{\mathbf{b}})^2)))^{\frac{1}{2}}}$$

  which is often used in computer vision, and relies on the concept of normalized correlation. Here $\overline{\mathbf{a}}$ and $\overline{\mathbf{b}}$ designate the average value of each component of the vectors. More formally, for $\mathbf{x} = (x_i)_{i=1}^n \in \mathbb{R}^n$,

$$\overline{\mathbf{x}} := \frac{1}{n}\sum_{i=1}^{n} x_i$$

We will see that each distance has its advantages and disadvantages.

1.3 Sorting the array by increasing distance

After having computed all the distances (whatever distance was used), we end up with

- An array of distances between the images of the dataset and the image to classify. The element at index i of that array contains the distance between the i-th image of the dataset and the given image to classify.
- An array of corresponding labels, i.e. the i-th element is the digit corresponding to the i-th image of the dataset.

Now for our next step, we have to sort these by increasing order of distance, while applying the same transformations to the array of indices. To keep it simpler, we consider the sorted array as an array that indicates which index goes where after sorting. For example:

distances: $5.5, 12.3, 1.2, 0.133, 6.0$
indices: $0, 1, 2, 3, 4$

sorted distances: $0.133, 1.2, 5.5, 6.0, 12.3$
corresponding indices: $3, 2, 0, 4, 1$

Which indicates that the first element of the sorted array is the third element of the unsorted array. Thus, we know that the label of the image

which distance is 0.133 is the 3-rd label of our array of labels. The following quicksort algorithm will do the trick:

```
Input: d = (d_1, ..., d_n) the array of distances, i = (i_1, ..., i_n) the
       array of indices, low, high the inclusive bounds where we want
       to sort
let (l, h) ⟵ (low, high);
let pivot ⟵ d_l;
while l <= h do
    if d_l < pivot then
        l ⟵ l + 1;
    else if d_h > pivot then
        h ⟵ h − 1;
    else
        swap(l, h, d, i);
        l ⟵ l + 1;
        h ⟵ h − 1;
end
if low < h then
    quicksort(d, i, low, h);
if high > l then
    quicksort(d, i, l, high);
```

Algorithm 1: quicksort

Where the method swap$(i, j, \mathbf{d}, \mathbf{i})$ simply assigns the values d_i to d_j and d_j to d_i and well as the value i_i to i_j and i_j to i_i.

1.4 Guessing the corresponding digit

Let's suppose we have a *sorted* array of indices $(j_1, \ldots, j_n)$. We fix $k \in \mathbb{N}$ and we only consider the values $(j_1, \ldots, j_k)$. That's were our array of corresponding labels comes in handy: let's say that $(b_1, \ldots, b_n)$ designates an array where the i-th element is the corresponding digit of the i-th element of the dataset. Thus, this array is unsorted.

The corresponding labels of the sorted array of indices is given by

$$(b_{j_1}, \ldots, b_{j_n})$$

Now, we consider the array $(b_{j_1}, \ldots, b_{j_k})$ and we pick the digit which is the

most frequent i.e. that appears the most in this array. The following algorithm will do the trick:

Input : $(c_1, \ldots, c_n)$ an array of integers
let $max \longleftarrow c_1$;
let $maxIndex \longleftarrow 1$;
for *i in 1...n* **do**
    if $c_i > max$ **then**
        $max \longleftarrow c_i$;
        $maxIndex \longleftarrow i$;
end
Output: $maxIndex$

Algorithm 2: Index of max

1.5 Results

Using

- a training dataset of a 1000 pairs of (image, label) for each digit possible
- $k = 7$
- a testing dataset of 10 000 images
- the euclidean distance

We have a predictive accuracy of 92.6% and it took 40.26 seconds to classify the 10k images on a 2015 MacBook Pro with a Intel Core i7 of the Intel Haswell generation. Now, using

- the inverted similarity distance

we achieved a better accuracy of 94.3% but it took 74.068 seconds to classify. This is explained by the much more time costly inverted similarity compared to the squared euclidean distance.

1.6 Improving the results using clustering algorithms

Using a reduced version of the MINST dataset (only 100k images instead of 500k), we only capable of classifying images at a speed of 0.04 seconds per image for the squared euclidean distance and 0.07 for the inverted similarity.

However, what the algorithm is essentially doing is figuring out from which images the given image is the closest and telling us what digit these images correspond to.

This means that if we could reduce the amount of images by picking a very good representation of each digit, the algorithm would be very fast because of such reduced dataset. However, picking only one good representation for each digit does not reveal to be practical, but we will use this intuition.

Instead of picking one digit, we are going to reduce to dataset to one that is a tenth the size and hope to not give up much predictive accuracy. The way to do it is using the k-means clustering algorithm. We will start our explanation with concepts applicable to all clustering algorithm.

1.6.1 Clustering algorithms

With a clustering algorithm, we have a dataset $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and we want to partition the dataset into clusters $\mathcal{C}_1, \ldots, \mathcal{C}_k$ where $\sqcup_{i=1}^{k} \mathcal{C}_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$

For a cluster $\mathcal{C}$, we define the *centroid* to be the average of the points of the cluster

$$\boldsymbol{\mu}_{\mathcal{C}} = \frac{1}{|\mathcal{C}|} \sum_{\mathbf{a} \in \mathcal{C}} \mathbf{a}$$

We define the *meloid* to be the closest point to the centroid in the cluster

$$\mathbf{m}_{\mathcal{C}} = \arg\min_{\mathbf{a} \in \mathcal{C}} d(\mathbf{a}, \boldsymbol{\mu})$$

where d is a distance in the space in which the dataset takes in values, in general it is the euclidean distance in $\mathbb{R}^n$.

We define the *tightness* of the centroid to be the average distance of the values in the cluster to the centroid.

$$T_{\mathcal{C}} = \frac{1}{|\mathcal{C}|} \sum_{\mathbf{a} \in \mathcal{C}} d(\boldsymbol{\mu}, \mathbf{a})$$

The *global tightness* is the avergae tightness on the set of all centroids

$$T = \frac{1}{k} \sum_{i=1}^{k} T_{\mathcal{C}_i}$$

1.6.2 k-means clustering

Here, we have a dataset $\mathbf{x}_1, \ldots, \mathbf{x}_p \in \mathbb{R}^n$ that we want to separate into $k \in \mathbb{N}$ clusters $\mathcal{C}_1, \ldots, \mathcal{C}_k$. We will use Lloyd's algorithm, that aims to minimize global variance inside the clusters

$$\sum_{i=1}^{k} \frac{1}{|\mathcal{C}_k|} \sum_{\mathbf{a} \in \mathcal{C}_k} ||\boldsymbol{\mu}_{\mathcal{C}_k} - \mathbf{a}||^2$$

This concepts is tightly linked to the global tightness of the cluster: the norm in a vector space can be used for creating a distance

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||$$

thus by minimizing the global variance inside the clusters, we are also minimizing the global tightness of our clusters. The algorithm goes as follows

Intialize the centroids $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$ randomly with values in the dataset.;
Let $k_1, \ldots, k_p$ be elements of $\mathbb{R}$.;
for _ *in* $0...t$ **do**
  for i *in* $1...n$ **do**
    //(k_i) represents the centroid to which (x_i) is the closest;
    $k_i \longleftarrow \arg\min_{j=1}^{k} ||\boldsymbol{\mu}_j - x_i||$;
  end
  for i *in* $1...n$ **do**
    //We recompute the centroids;
    $\mathcal{C}_i \longleftarrow \{\mathbf{x}_j | k_j = i\}$;
    $\boldsymbol{\mu}_i \longleftarrow \frac{1}{|\mathcal{C}_k|} \sum_{\mathbf{a} \in \mathcal{C}_k} \mathbf{a}$;
  end
end

Now a few things to explain:

- The number of iterations t is empirically determined. Which brings us to our next point
- The algorithm insures that there is convergence of the centroids. Thus, we fix a constant number of iterations (to avoid doing infinitely many computations, you know).

This algorithm is linear in the amount of observations, meaning it is in $\mathcal{O}(p)$ in our case.

We are not going to show here that the algorithm actually minimizes the global variance inside the clusters.

1.6.3 Results using k-means clustering

Using

- $t = 20$ for k-means clustering
- the same setup as before for k-nearest neighbors

the results are

- using the inverted similarity
  - a predictive accuracy of 90.8%
  - 0.0064 seconds to classify each image in average
- using the square euclidean distance
  - a predictive accuracy of 89.9%
  - 0.0037 seconds to classify each image in average

As expected, classifying using a dataset tenth the size produces an algorithm that classifies with ten times the speed. Yet, the predictive accuracy are still high.

For comparison, the predictive accuracy using a random set of dataset the same size as the reduces dataset yields the following results

- using inverted similarity: roughly the same speed but a predictive accuracy of 88%.
- using squared euclidean distance: roughly the same speed but a predictive accuracy of 82.7%.

2 The deep learning approach

2.1 Generalities on deep learning

Let's give a little bit of context about the concepts before giving a general intuition of the idea. We are not going to go into details about what makes a good model using deep learning, because I am afraid I won't be able to condense much of it.

2.1.1 Perceptron

A perceptron is a function of the form

$$\begin{aligned} \mathbb{R}^p &\to \mathbb{R} \\ \mathbf{x} &\mapsto f(\mathbf{w}^\top \mathbf{x} + b) \end{aligned}$$

where $\mathbf{w} \in \mathbb{R}^p$ is referred to as the vector of weights associated with each component of $\mathbf{x}$, and where $\mathbf{b} \in \mathbb{R}$ is called the bias. Intuitively, for

$$\mathbf{w} = (w_1, \ldots, w_p) \qquad\qquad \mathbf{x} = (x_1, \ldots, x_p)$$

the weight w_i can be seen as the amount of important we give x_i in the computation of the function.

f is the activation function, which can be chosen depending on which problem we are facing. For example, in our example, we could consider a perceptron measuring how confident it is that the given 28 by 28 image is a 6. In this case, we better choose an activation function that maps $\mathbb{R}$ to $[0; 1]$ while conserving the order in $\mathbb{R}$. Then, if the function outputs a 0.05, the perceptron tells us the image is probably not a 6, and if it outputs a 0.95, it's probably a 6.

Intuitively, this means that having a perceptron with a good predictive accuracy is a matter of tweaking the set of weights till we obtain satisfying outputs.

2.1.2 Loss function

How exactly do we mean by "obtaining satisfying outputs"? How do we measure how satisfying our outputs are? Well, in our example, we know for image of training set the corresponding digits. For an image of a 6, we would really like to have an output value 1.

So we can define how bad the actual input is by comparing it to 1 with a function called a loss function, like the following:

$$L(f(\mathbf{w}^\top \mathbf{x} + b), 1) = |f(\mathbf{w}^\top \mathbf{x} + b) - 1|$$

So for an image that is not a six, and for a bad perceptron, we would have a big value for the loss function considering that the actual output varies from the expected output by a lot.

2.1.3 Layer

Going deeper in our example: what if we wanted to have 10 perceptrons, each one telling us how confident it is that the image is a $0, 1, \ldots, 9$? That would suggest aligning the set of 784 inputs, called a layer, and a set of 10 perceptrons, another layer, and doing the computations we described earlier.

Now it might be a good idea to consider layers as a cohesive structure and not just a set of perceptrons that incidentally are stacked together. Why? Well instead of considering the activation function as a function of the form $\mathbb{R}^p \to \mathbb{R}$ we could consider it as a function $\mathbb{R}^p \to \mathbb{R}^q$. Each component of the output would be the value of the perceptron at the component's index.

So for example, the following activation function, called the softmax:

$$[\boldsymbol{f}(x_1, \ldots, x_p)]_i = \frac{e^{\mathbf{w}_k^\top \mathbf{x} + \mathbf{b}}}{\sum_{i=1}^{p} e^{\mathbf{w}_k^\top \mathbf{x} + \mathbf{b}}}$$

is only possible to define considering the layers as a cohesive structure. In a way, each perceptron has a value of $e^{\mathbf{w}_k{}^\top \mathbf{x} + \mathbf{b}}$ only normalized over the set of inputs. This way, we keep a perceptron's value from exploding and we have an output space $[0; 1]$. An more straightforward implementation of a multi-dimensional activation function would be

$$\frac{\mathbf{w}_k^\top \mathbf{x} + \mathbf{b}}{\sum_{i=1}^{p} \mathbf{w}_k^\top \mathbf{x} + \mathbf{b}}$$

Now, in our example we need to redefine the loss function to something that applies to a vector rather than a single value. So for an image with expected output vector $\mathbf{y} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)$ i.e. a six, and actual output vector of perceptrons $\boldsymbol{f}(\mathbf{x})$, a loss function could be

$$L(\mathbf{y}, \boldsymbol{f}(\mathbf{x})) = \frac{1}{10} \sum_{i=1}^{10} (y_i - [\boldsymbol{f}(\mathbf{x})]_i)^2$$

which is called the mean squared error.

2.1.4 Training

As we saw, each weight tells us how much each component of the input vector carries more importance. And having a good predictive accuracy is just a matter of modifying the weights. But how exactly would we go on to do that?

Well, first of all we will consider the simplest form a training i.e. incremental training, which consists of feeding an image and its label to our training algorithm, which will modify the weights accordingly.

The method used to modify the weights is one of the things that I don't see myself condensing. It is an algorithm used in convex optimization, which has to do with finding the minimum of a convex function. The algorithm for modifying the weights of perceptron goes as follows:

Initialize the set of weights $w_1, \ldots, w_p$ randomly in the output space of the activation function.
for $(\mathbf{x}, \mathbf{y})$ *in the dataset of (feature, label)* **do**
  for i *in* $1...p$ **do**
    $w_j \longleftarrow w_j - \eta \frac{\partial L(\boldsymbol{f}(\mathbf{x}), \mathbf{y})}{\partial w_j}$
  end
end

We could also iterate multiple times over the dataset given enough computational time. You may have noticed we variable η which is called the learning rate. Intuitively, it represents how strongly we want to shift the value of each weight for each new given observation. Shifting too strongly means a possibility of overshooting the perfect value for the weight. Shifting too weakly means having a possibly slow convergence towards the perfect value for w_j.

Again, the technicalities of this are important to eventually understand, as is regularization, batch learning or defining a computational graph yet we won't cover any of them here.

2.1.5 Multiple layers structure

Now instead of having an input layer and an output layer, we also consider an array of layers in between, called hidden layers. We can easily conceptualise this with our previous definition, by considering each pairs of consecutive layers to be input layers and output layers in our previous conceptualisation.

The value of each perceptrons is going to be computed using the vector of values of the perceptrons of the previous layer.

We can thus define different activation functions for each layer, and a loss function for the output layer the same way we used to do. Yet, we need to modify the training algorithm to account for the multiple layer structure. We won't go into details but the algorithm is basically the same as the one for the no-hidden layer structure. Except we will compute the partial derivatives for weights of previous layers before computing the partial derivatives for a given layer; the reason is that the partial derivative of the given layer is dependant on the values of the partial derivative of the previous layers. We can compute the relation using the chain rule, knowing the formula for each perceptron.

2.2 Our implementation

2.2.1 Specifications

We will be using a 2-hidden layer structure of 40 perceptrons each. Our loss function is the cross entropy and we will use the stochastic gradient descent as our algorithm to shift the weights at each training step. We will train the model 100 over our training data.

- Input layer: 784 identity preceptrons
- Hidden-Layer 1: 256 perceptrons, activation function is ReLu
- Output layer: 10 perceptrons, activation function is softmax

2.2.2 Cross Entropy loss

In our example, we will use a version of the cross entropy, which if you are familiar with information theory is the entropy of a probability distribution p coded with another one q:

$$H(p,q) = \sum_x p(x) \log(\frac{1}{q(x)})$$

it applies perfectly in our example as we will use an output layer of 10 perceptrons, and each of those is going to give us a confidence of the image being the digit associated with the perceptron. This confidence will be between 0

and 1, thus it defines a probability distribution over the set of possible digits. The actual probability distribution we want to enforce will of course have a spike at the correct value, where the probability will be 1 and 0 for the rest.

So image we have the values $\mathbf{y} = (y_0 \ldots, y_9)$ for the perceptrons and the value $\mathbf{e}_k$, where k is the digit, and $\mathbf{e}_k$ is the the k-th basis vector in $\mathbb{R}^{10}$. The cross entropy will be defined as

$$H(\mathbf{e}_k, \mathbf{y}) = \sum_i [\mathbf{e}_k]_i \log(\frac{1}{y_i})$$

where the sum is indexed over the non-null perceptron values. Thus, we in a sense want to see if the information of the true label $\mathbf{e}_k$ is uncertain given the probability distribution $\mathbf{y}$. Ideally, it would not be so as the predicted label would give a good probability distribution and capture the information of the label.

2.2.3 Results

The results are 95.5% predictive accuracy and a 24us to classify each image. These results are far greater than what we achieved using k-nearest neighbors. We could do even better with adjusting the model and choosing a better algorithm than the stochastic gradient descent.