“ALGORITHMS FOR MASSIVE DATASETS” AND “STATISTICAL METHODS FOR MACHINE LEARNING” JOINT PROJECT REPORT

Matteo Rusconi

Contents

1 Introduction

This is the report of the joint project between the two courses “Algorithm for massive datasets” and “Statistical methods for machine learning”. The task was to train a neural network to classify photos of human faces, according to if they wear glasses or not. The original dataset provided by Kaggle contains 5000 labeled photos. I decided to implement the neural network algorithm from scratch, trying to build it as simple as possible. In fact i implemented a trivial multi-layer perceptron, which takes a photo as input and outputs the prediction.

2 Data organization

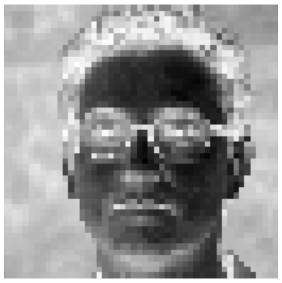
Of the 5000 photos in the dataset i partitioned the first 4200 as training set and the ones from 4201 to 4500 as test set. The labels are organized as csv files; training labels are in *train.csv* and test labels are in *test.csv.* These csv files have 2 columns, *id* and *label*, and respectively 4200 and 300 rows.

3 Preprocessing techniques

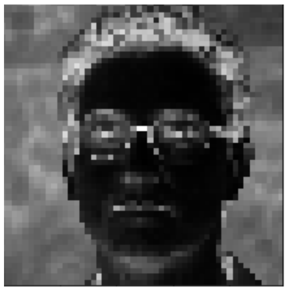
As my neural network is a classic MLP, some preprocessing had to be done because i wanted the input layer to have one neuron per pixel and the downloaded photos, as seen below here,

had a resolution of 1024 x 1024 pixels, so a 1048576 neuron input layer is ovbiously not computationally easy to handle. The first thing i did is in fact downscaling the photos to a 50 x 50 pixel resolution. I found out that this resolution was a good compromise between image quality and number of pixels. Then i converted the photos from RGB to grayscale in order to have a single float value for every pixel in the images.

Since the majority of glasses are black or gray colored, i thought of converting grayscale to negative grayscale, in order to have pixels corresponding to glasses parts brighter (and higher corresponding float value) than the surrounding skin of the subject.



This was not necessary but i liked the idea of having parts of glasses to cause higher activations to the corresponding neurons in the MLP. Then i figured out that the easiest part of glasses to be recognized by the algorithm is probably the part that goes on the nose, because it cannot really be confused with hair or eyebrows. So i applied a filter that enhances the brightness of that part (only if there is already some brightness i.e. it is likely that there is glasses) and lowers the brightness of the rest of the image (*detectWhite* function in *preprocess-script.py*).



In this way the photo is less noisy and the neural network can more easily focus on parts of the images that matters in the classification task. At last to further reduce the number of neurons in the input layer i decided to crop the images and keep only the part of the face where there can be glasses.

This works because glasses are always in the same portion of the images. Cropping also removes noise caused from hair which are tipically really bright in the negative pictures. The final images have a 34 x 10 pixel resolution, so i managed to have a 340 neuron only input layer.

Float values of the photos are redistributed over a [0,1] interval, and then normalized by subtracting mean and deviding by standard deviation. A final one-dimension list is created with the 340 values for each image, and the data is finally ready to be processed by the neural network. All of these preprocessing operation are executed by the script *preprocess-script.py*, which is inserted in a cell on the colab notebook.

In addition, i noticed that the original dataset contains some wrong labeled photos. Since they caused heavy overfitting i decided to look for them all manually and, once figured out which ones were wrong, i implemented a script that fixed them. I found a total of 526 wrong labeled photos, which is roughly 10% of them, and this can explain the significant increase in test error that i experienced when training error went below 10%. I decided to not include in section 5 experiments done with the noisy dataset.

4 Algorithm implementation

4.1 Network structure

The multi-layer perceptron is implemented from scratch, as a python multi-dimensional list. The first dimension specifies the layer, the second one specifies the neuron, the third one selects between activation values or weight values and the last one specifies which activation or which weight. The data structure is composed on object creation with a nested for loop

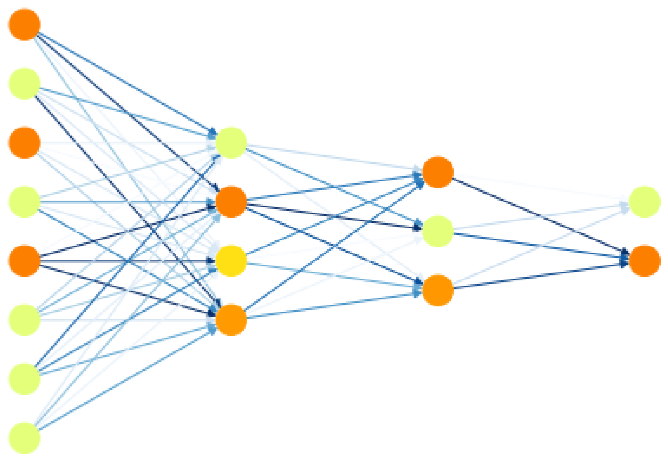
```python
    def __init__(self, networkStructure):

        self.nn = []

        for i in range(len(networkStructure)):
            self.nn.append([])
            for j in range(networkStructure[i]):
                if (i < len(networkStructure) - 1):
                    self.nn[i].append([])
                    self.nn[i][j].append([1, 0])
                    self.nn[i][j].append([self.randomNum() for\
 x in range(networkStructure[i + 1])])
                else:
                    self.nn[i].append([])
                    self.nn[i][j].append([1, 0])
```

and initializes the weights with random numbers drawn from a distribution with mean 0 and standard deviation 0.35. To create and initialize a neural network it is just required to specify a tuple with the numbers of neurons for each layer:

```
nn = NeuralNetwork((8, 4, 3, 2))
```

To visualize the neural network i impelemented a print function which relies on the *networkx* library. To make this possible i implemented a function that converts the neural network list to an adjacency matrix of the graph representing the network, and then i let networkx do the job, with a couple of graph layout adjustments.

```
nn.setInput((1,0,1,0,1,0,0,0))
nn.evaluateNetwork()
nn.print()
```



In this printed image of the network the intensity of color reflects the value of the corresponding activation or weight. The *evaluateNetwork* function propagates the activations coming from the input layer deeply into the network.

The print function was really handful at early stages of backpropagation implementation, while it becomes not so useful when networks becomes lager, because there are so many neurons and edges that they become not anymore distinguishable.

4.2 Backpropagation

The classic learning algorithm for neural networks is gradient descent with backpropagation. I implemented the mini-batched version of gradient descent. This is the generic formula for weight adjustment:

$$w_{i,j} \leftarrow -\eta \frac{1}{|S_t|} \sum_{s \in S_t} \frac{\partial \ell_s(W)}{\partial w_{i,j}} \qquad (i,j) \in E \tag{1}$$

where η is the learning rate, S_t is the fixed mini-batch size, $\ell(W)$ is the loss function on the current weight matrix W and E is the set of edges. We can write $\frac{\partial \ell_s(W)}{\partial w_{i,j}}$ as $\frac{\partial \ell_s(W)}{\partial s_j}\frac{\partial s_j}{\partial w_{i,j}} = \delta_j v_i$ using the derivative chain rule where $s_j = \boldsymbol{w}(j)^{\mathrm{T}}\boldsymbol{v}(j)$, $v_j = \sigma(s_j)$ and σ is the activation function. This is the recursive definition for generic δ_j:

$$\delta_j = \frac{\partial \ell_s(W)}{\partial s_j} = \begin{cases} \ell'(v_j)\sigma'(s_j) & \text{if output layer} \\ \sigma'(s_j)\sum_{k:(j,k)} \delta_k w_{j,k} & \text{otherwise} \end{cases}$$

My implementation calculates the deltas for every neuron in every mini-batch, and then updates weights as described in (1).

4.3 Parameters

As activation function i utilized the leaky ReLU on all layers except the output one:

$$\sigma(z) = \begin{cases} \alpha z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

where $\alpha = 10^{-2}$.
On the last layer i utilized the softmax function in order to have an easy gradient calculation in combination with the log loss function. Softmax function is

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \text{for } i = 1,...,K \text{ and } \mathbf{z} = (z_1,..., z_k) \in \mathbb{R}^K \tag{2}$$

and log loss function for binary classification is

$$-(y\log(p) + (1-y)\log(1-p)) \tag{3}$$

It is demonstrated that the gradient of softmax with log loss is simply

$$\delta_i = v_i - y_i$$

where $\mathbf{y}$ is the label associated to the network's input.

4.4 Data managment

As data processed by the algorithm could be relly large, i implemented it in a way that the size of data to be used is not bounded on main memory size. In fact, only one mini-batch at a time is loaded in main memory, while the rest remains on hard disk. This applies also to the preprocessing phase, where only a fixed portion of the photos is loaded in RAM, and also to the training/test error calculation, where data is loaded accordingly to mini-batch size.

4.5 Algorithm complexity

The training algorithm is divided in two steps: feed-forward propagation of signal and backpropagation of weight adjustments.

Feed-forward
This step is composed of the computation of many weighted sums, depending on the network structure. Assuming that our network has 3 layers, we call $|a|, |b|$ and $|c|$ the number of neurons in each layer. For each pair of adjacent layers (x, y), there will be computed $|y|$ weighted sums of complexity $\Theta(|x|)$ causing a composed complexity of $\Theta(|x||y|)$. The total complexity of a feed-forward phase in a three layered neural network is then $\Theta(|a||b| + |b||c|)$

Backpropagation
This step is composed of the computation of the deltas for every neuron, and every delta in a specific layer is computed by a weighted sum of the deltas in the subsequent layer. After this, every weight in the network is adjusted. Since there are $|a||b| + |b||c|$ edges, the complexity of a three layered network is $\Theta(|a||b| + |b||c| + |a||b| + |b||c) = \Theta(|a||b| + |b||c|)$

We've seen that the training complexity on one example is $\Theta(|a||b| + |b||c|)$. If we have n examples and we train on t epochs the total complexity becomes $\Theta(nt(|a||b| + |b||c|))$

5 Experiments and final results

Since my neural network is a dense MLP with a lot of neurons, training time is usually not very short, so at the end i had to give priority to short time experiments and tune hyperparameters towards small networks.

That said, my first attempts did actually follow the classic heuristic that says to use only one hidden layer with a number of neurons equal to mean value between input and output neurons, which in my case is

$$(340 + 2)/2 = 171$$

as i have 2 output neurons that classify inputs as glasses or no glasses. I started with a learning rate of 0.01, a learning rate decay of 1/10 per epoch, a total of 10 epochs and a minibatch size of 1 as other hyperparameters. So my first network has been a (340, 171, 2) and led to these results:

(340, 171, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**1.38%**	**1.33%**	1	109.68

where training error and test error are referred to the best epoch (i.e. the one with lowest test error) and total time is expressed in minutes. We can see that this result is already satisfing in terms of generalization of the outputed predictor, in fact a test error of 1.33% is not bad at all. But the problem with this network becomes clear when we have a look on the total time of execution of the algorithm, which is very close to 2 hours. This is because the network has fully connected layers, and we've seen that the complexity of the algorithm is $\Theta(ab + bc)$ where a, b and c are the number of neurons over the 3 layers (n and t are fixed). This means that what we need to avoid networks that have a large amount of neurons in 2 adjacent layers.

This brought me to test the performance of the algorithm on a significantly smaller network, which is a (340, 10, 2) structured neural networks. I left untouched the other hyperparameters to see what the difference of neurons number alone does to the predictive power of the network:

(340, 10, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.47%**	**2.33%**	1	8.68

Now, as expected, the total execution time is reduced to less than 10 minutes. On the other hand we can see that the performance are similar. In order to increase the number of neurons without increasing too much the execution time, adding a new layer is certainly a good approach as it won't be adiacent to the first layer. So i added a new layer with 5 neurons and these are the results:

(340, 10, 5, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.64%**	**1.33%**	1	8.76

As expected execution time is only slightly increased, and we notice that test error is decreased. It is also noticeable that there is some strange behavior here, as test error is lower than training error. This could mean that there are still some wrong labeled examples in the training set, so that even if the predictor classifies them correctly they are still counted as wrong predictions.

This result is already not bad considered the fact that i am using a general purpose multi-layer perceptron instead of a more sophisticated model like Convulutional Neural Networks, which with their capability of recognizing figures and shapes, and with their efficient data dimensionality reduction through the usage of filters and pooling layers, surely are a more solid candidate to solve this image feature recognition problem.

For the sake of completeness i now tested some variations of these deeper small networks:

(340, 10, 5, 4, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.11%**	**1.66%**	1	8.85

(340, 10, 10, 10, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.40%**	**3.33%**	1	8.92

(340, 20, 10, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.71%**	**2.66%**	1	14.86

(340, 10, 7, 5, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.57%**	**2.33%**	1	9.03

(340, 10, 8, 6, 4, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.26%**	**1.66%**	1	9.14

(340, 30, 20, 10, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**2.26%**	**1.66%**	1	22.11

We can see that these networks do perform more or less similarly, and the differences may be caused by randomization on edge weights, so i decided to pick the best one so far and apply some changes on learning rate and mini-batch size:

Learning rate

(340, 10, 5, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.001	1/10	4200	**5.83%**	**6%**	1	8.76
20	0.001	1/10	4200	**4.80%**	**5.66%**	1	16.94
20	0.001	1/20	4200	**3.83%**	**2%**	1	16.94
20	0.001	1/25	4200	**4.61%**	**2%**	1	16.94
20	0.001	1/30	4200	**3.97%**	**3.66%**	1	16.94
10	0.03	1/10	4200	**1.54%**	**0.66%**	1	8.76
10	0.03	1/15	4200	**2.57%**	**1.66%**	1	8.76
10	0.03	1/20	4200	**2.80%**	**1.33%**	1	8.76

We can see that lowering learning rate affected negatively generalization power, while highering it did make the model perform slightly better. In these experiments i tried to also tune learning rate decay and epochs number.

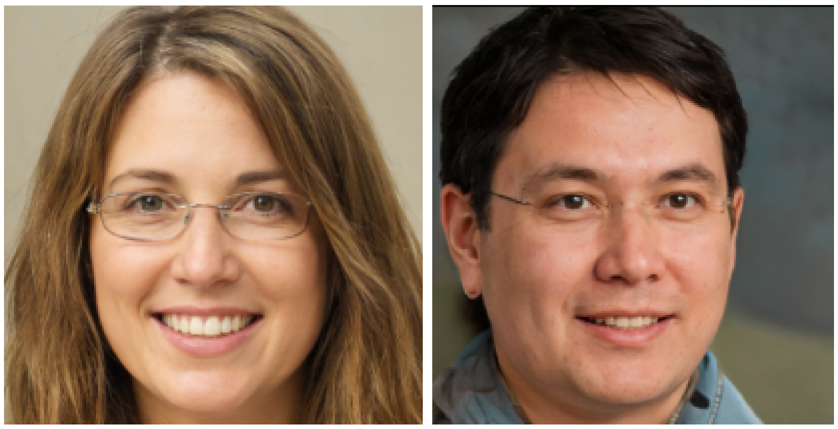
Mini-batch size

(340, 10, 5, 2)

epochs	l_rate	l_dec	t_size	**tr_err**	**ts_err**	minib_size	tot_time
10	0.01	1/10	4200	**46.11%**	**43.99%**	50	6.08
10	0.001	1/10	4200	**48.69%**	**48%**	50	6.08
10	0.03	1/10	4200	**34.16%**	**34.66%**	50	6.08
10	0.01	0	4200	**7.26%**	**5%**	3	7.27
15	0.01	0	4200	**9.35%**	**8%**	5	10.11
20	0.01	1/15	4200	**11.11%**	**8%**	3	14.26

We can see that highering the mini-batch size makes the learning process slower. This is due to the fact that the higher the mini-batch size is, the more accurate the step towards better solution is, but it also increases the time required to make this step. The key is to find the right compromise between mini-batch size and number of epochs needed to converge towards a good model. Unfortunatly i couldn't find a combination of epoch number, learning rate and mini-batch size which performs better than the previouses networks with mini-batch size of 1.

At last it's interesting to see which photos are the most difficult to be recognized by the trained model:



These are the two photos of the test set wrongly classified by my best predictor. The left one has some bright and thin glasses, which mesh partially with her hair, and this could be the reason of the wrong classification. The one on the right is instead clearly strange, since the lens part of the glasses is completely missing. This is a wrong generated photo (the glasses in the images are generated by a machine learning model too) and it is totally acceptable that photos like these are missclassified.