P. Koumoutsakos, P. Hadjidoukas
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2017

Set 7 - PCA with Neural Networks

Issued: November 10, 2017
Hand in (optional): November 17, 2017 8:00am

Question 1: Principal Component Analysis with Oja's rule

Consider a linear network which maps each element of a set of input vectors $\mathbf{x}^i \in \mathbb{R}^D, i = \{1, \ldots, N\}$ to some feature space $y_i \in \mathbb{R}$ through a weight matrix $\mathbf{w} \in \mathbb{R}^D$:

$$y^i = \mathbf{w}^T \mathbf{x}^i. \tag{1}$$

this operation, computing the output of the neurons in a neural network, is commonly referred to as prediction or forward propagation. We have seen that the Hebbian rule for updating the weights $\Delta \mathbf{w} = \beta y^i \mathbf{x}^i$ can lead to unstable growth or decay of the weight matrix. Oja's algorithm [1], which regularises the Hebbian learning rule, is given by (for small learning rates $\beta \ll 1$):

$$\mathbf{w}^{i+1} = \mathbf{w}^i + \beta y^i (\mathbf{x}^i - \mathbf{w}^i y^i) \tag{2}$$

here we preserved the original notation from [1], where the iterative learning process shares the same index i as the dataset, but this algorithm is valid also when performing multiple sweeps over a fixed dataset of size N. The update rule imposes to the weight vector to have unit length and the iterative learning has a fixed point for $\mathbf{w}$ equal to the first eigenvector of the set of input vectors.

In order to find the first M eigenvectors of a dataset, we extend the network by introducing a set of neurons:

$$y_j^i = \mathbf{w}_j^T \mathbf{x}^i, \quad j = \{1, \ldots, M\} \tag{3}$$

If trained with equation 2, all weight vectors would converge to the same dominant eigenvector. In order to ensure orthonormality of the weight vectors Oja's rule has to be modified as [2]:

$$\mathbf{w}_j^{i+1} = \mathbf{w}_j^i + \beta y_j^i \left(\mathbf{x}^i - \mathbf{w}_j^i y_j^i - 2 \sum_{k<j} y_k^i \mathbf{w}_k^i \right) \tag{4}$$

a) Apply Oja's learning rule to find the first 10 principal components of the MNIST dataset. The MNIST dataset (figure 1) is a widely used benchmark dataset of 28 by 28 grayscale pixel images of handwritten digits. Interpret your results.

The skeleton code is already equipped to read the dataset and extract the principal components. Your task is to implement prediction and Oja's rule by modifying `Layer_Normal.h`. You can visualize the weight vectors with `visualize_components.py`.

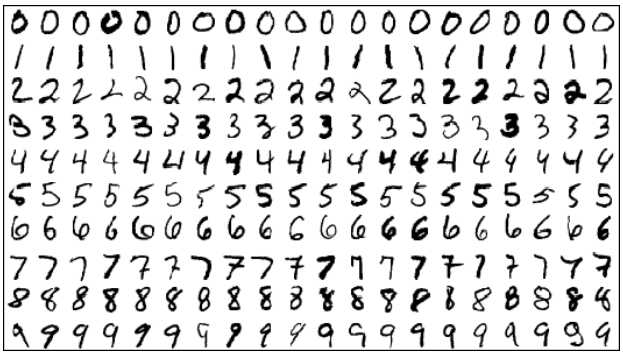

Figure 1: Examples from the MNIST dataset

Note that in order to simplify the notation we are omitting mention of the bias vector. Since we do not assume that the data is centered, each neuron should behave according to $y_j^i = \mathbf{w}_j^T \mathbf{x}^i + b_j$. Previous and following relation hold true by using the convention of hiding bias into the weight vectors according to $\mathbf{x}^i \leftarrow \{\mathbf{x}^i, 1\}$ and $\mathbf{w}_j \leftarrow \{\mathbf{w}_j, b_j\}$.

b) Parallelize with OpenMP threads the functions you just implemented, discuss and address any load-balancing issues, and report the scaling of your implementation.

Question 2: Auto-associative Neural Network

We now consider a neural network with two layers and linear activation functions that is trained via backpropagation. The vector of outputs $\mathbf{y}^i = \{y_1^i, \ldots, y_M^i\} \in \mathbb{R}^M$ of the first layer of neurons is produced by multiplying an input $\mathbf{x}^i$ with the neurons' weight matrix $W_\mathsf{a} = \{\mathbf{w}_{\mathsf{a},0}, \ldots, \mathbf{w}_{\mathsf{a},M}\}^T \in \mathbb{R}^{M \times D}$:

$$\mathbf{y}^i = W_\mathsf{a} \mathbf{x}^i$$

The second (and final layer) receives the output of the first layer and operates with the matrix $W_\mathsf{b} \in \mathbb{R}^{D \times M}$ so that we define the auto-associative output:

$$\tilde{\mathbf{x}}^i = W_\mathsf{b} \mathbf{y}^i = W_\mathsf{b} W_\mathsf{a} \mathbf{x}^i \tag{5}$$

When the network is trained to produce an output $\tilde{\mathbf{x}}^i$ that is equal to the input $\mathbf{x}^i$ then it can be shown [3] that this (so called "auto-associative network") performs PCA.

In order to train the network we introduce the cost function:

$$\mathcal{L}(\mathbf{x}^i, W) = \frac{1}{2} \|\tilde{\mathbf{x}}^i - \mathbf{x}^i\|^2 \tag{6}$$

The weights can be updated through backpropagation by computing the gradient of the error with respect to W (which stands for both W_a and W_b): $\mathsf{d}\mathcal{L}(\mathbf{x}^i, W)/\mathsf{d}W$. Since the gradient is expected to be noisy, it is common practice to update the weights by averaging a 'mini-batch' of gradients:

$$W^{k+1} = W^k - \frac{\beta}{B} \sum_i^B \frac{\mathsf{d}\mathcal{L}(\mathbf{x}^i, W^k)}{\mathsf{d}W^k} \tag{7}$$

where β is the learning rate and B is the batch-size.

a) Implement the backpropagation algorithm and train the network to find the first 10 principal components of the MNIST dataset. What do you observe ?

You can check the correctness of your implementation (in addition to passing the test already implemented in `main_backprop.cpp`) by comparing your output with that of the TensorFlow implementation in `mnist_tf.py`. For this point you only need to modify `backPropagate` in `Layer_Normal.h`.

Upon convergence to the global optimum, the principal components of the dataset should compose the rows of W_{a} and the columns of W_{b}. The skeleton code only looks at W_{b}. Are there differences in the two sets of modes?

Note that backpropagation may encounter saddle points in the cases of linear networks. Changing the activation from linear to any type of non-linear function does not help.

b) Parallelize the update such that each thread performs one element of the minibatch. Report your scaling.

If for the first question you introduced multi-threading in the `propagate` function of `Layer_Normal.h`, you will have to disable it for this question. HINT: Refer to the `predict` and `backProp` functions of `Network.h`. You will notice that the memory layout to carry out the network operations is already thread-safe. Therefore you will only have to modify the main training loop in `main_backprop.cpp`.

c) Use TensorFlow to train auto-associative networks with different depths and activation functions.

For all models include a compression layer of size 10 in the middle of the network. Upon convergence, you can compare the modes of the dataset to the network outputs following an individual activation in the compression layer. Consider only the half of the network that goes from the compression layer to the output layer. For each of the 10 compression layer neurons in turn, set their output to 1 and the output of the others to 0, and report the final output of the network. Can you propose a method that will overcome the saddle points? Can you achieve better accuracy with deeper models?

References

[1] Oja, Erkki Simplified neuron model as a principal component analyzer, Journal of mathematical biology, 1982.

[2] Oja, Erkki, Principal components, minor components, and linear neural networks, Neural networks, 1992.

[3] Baldi, Pierre and Hornik, Kurt, Neural networks and principal component analysis: Learning from examples without local minima, Neural networks, 1989.