582744 Advanced Course in Machine Learning

Exercise 4 **Due April 11, 23:55 AM**

Rules:

1. Return your solutions in Moodle by the deadline.
2. The submission consists of two parts: (a) A single PDF file containing your answers to all questions. (b) A single file containing your code (either a single plain text source code or a compressed file). Do not include datasets, plots etc in this file, only the code.
3. If you feel comfortable, add an estimate of how many hours you worked on the problems in the beginning of your report.
4. Please typeset your work using appropriate software such as LaTeX. However, there is no need to typeset the pen and paper answers – you can also include a scanned hand-written version.
5. Pay attention to how you present the results. Be concise.
6. Spotted a mistake? Something is unclear? Ask for clarifications in Moodle.

This set of exercises is due on Tuesday April 11, before 23:55 AM.

0 How many hours did you work on these?

1 Regularized linear regression (2 pts)

During the lectures it was stated that ridge regression pulls the solution vector towards zero, but we did not clearly describe how exactly it happens. If we store individual samples as rows of the input matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$, then the solution for the ridge regression problem

$$\operatorname*{argmin}_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda \|\mathbf{w}\|^2$$

is

$$\mathbf{w}_\lambda = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$

and the solution for the unregularized problem is obtained when $\lambda = 0$. Characterize the relationship between $\mathbf{w}_\lambda$ and $\mathbf{w}_0$, the solution for the unregularized problem. You can either try directly expressing $\mathbf{w}_\lambda$ as a function of $\mathbf{w}_0$, or you can look at the the gradients of the regularized problem at $\mathbf{w}_0$. Describe how the estimate changes for increasing regularization constant λ.

Hint: In case you are unable to solve the problem for vector-valued $\mathbf{x}$, you can start with scalars x instead to get a rough idea of the solution.

2 Stochastic neighbor embedding (programming, 3 points)

Implement stochastic neighbor embedding, using the loss function and gradient presented in the lecture slides. For more information, you can consult the original research article http://papers.nips.cc/paper/2276-stochastic-neighbor-embedding.pdf.

In this exercise you will use it for exploring the MNIST data set of handwritten digits, mapping them to a two-dimensional embedding space. Download the data set and useful code snippets for handling it from http://www.cs.helsinki.fi/u/sorkhei/mnist.tar.gz – this has earlier been used on the "Introduction to ML" course.

To fit SNE follow the instructions below. They do not exactly match the suggestions provided in the paper, but result in a plot that should look the same for everyone.

- Initialize the algorithm using the first two PCA components, using at least 5000 of the examples for computing the principal components. Normalize the scores by their standard deviation, so that the variance for each dimension is one.

- Write a function that computes the neighborhoods using

$$d_{ij}^2 = \sum_d \frac{(\mathbf{x}_{id} - \mathbf{x}_{jd})^2}{2\sigma^2}$$

$$p_{ij} = \frac{e^{-d_{ij}^2}}{\sum_{k \neq j} e^{-d_{ik}^2}},$$

  further assuming $d_{ij} = \infty$ to exlude the point itself from the neighborhood

- Compute the neighborhoods p_{ij} in the original data space using $\sigma^2 = 10000^2$ in the above formula, and the neighborhoods q_{ij} in the embedding space using $\sigma^2 = 1$. How do these two parameters control the algorithm? Why did we need very large value for the former?

- Optimize the loss function

$$L(\mathbf{z}) = \sum_{i,j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

using gradient descent with the gradient

$$\frac{\partial L}{\partial \mathbf{z}_i} = 2 \sum_j (\mathbf{z}_i - \mathbf{z}_j)(p_{ij} - q_{ij} + p_{ji} - q_{ji}).$$

Start with standard gradient descent using fixed step size of 0.05, but if you wish you can also try faster alternatives. Plot the training loss as a function of the algorithm iteration.

- Visualize both the initial PCA representation and the final SNE representation with scatter-plots, coloring the samples according to the class labels. Does either one look better?

Note that the algorithm is rather slow, so you probably will not want to learn the embedding for all of the samples. Try making an implementation that works at least for 1000 samples, but if your code is too slow then use a smaller subset. Remember to mention in your report how many samples you used.

If you want, you can also play around with different initializations, optimization heuristics, and different values for σ. The choices above were given to make grading easier (that is, so that the result plots and errors would be comparable across your solutions) and they are definitely not optimal for solving the problem itself. The original paper also has a range of suggestions for improving the learning – feel free to try some of those if you feel like it.

3 Non-negative matrix factorization (programming, 3 points)

The non-negative matrix factorization was presented in the lectures, but you should take a look at the original research article `http://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf` before solving the paper to improve the motivation and understanding.

(a) Read in the data set (from Moodle) that contains small image patches extracted from natural images, represented as row-wise concatenation of 16×16 grayscale pixel representations. Store it as a data matrix $\mathbf{X}$ so that each image patch is represented as one column of $D = 16 * 16 = 256$ dimensions. The dataset is originally from `http://mldata.org/repository/data/viewslug/natural-scenes-data/` but was modified for this exercise.

(b) Implement the NMF algorithm using the update rules:

$$\mathbf{H}_{t+1} = \mathbf{H}_t \times \frac{\mathbf{W}_t^T \mathbf{X}}{\mathbf{W}_t^T \mathbf{W}_t \mathbf{H}_t}$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t \times \frac{\mathbf{X}\mathbf{H}_{t+1}^T}{\mathbf{W}_t \mathbf{H}_{t+1} \mathbf{H}_{t+1}^T}$$

Here the products in the numerator and denumerator are regular matrix products, but the division and the multplication with $\mathbf{H}_t$ and $\mathbf{W}_t$ are element-wise.

(c) Apply the algorithm on data $\mathbf{X}$ with three different choices for the number of factors: $K = 8$, $K = 16$ and $K = 64$. Compute also the final (training) loss for each choice. Describe what happens for the loss when you increase K.

(d) Apply PCA on the same data (you can use ready-made implementations here), using the same choices for K. Which model fits the data better (that is, obtains lower training error), PCA or NMF? Can you think of why?

(e) Visualize the resulting basis vectors (columns of $\mathbf{W}$) in image format, converting the vectors back to matrices for visualization. For PCA you should order the components according to the variance explained, but for NMF there is no natural ordering. Which of the three choices for K looks most useful?

To visualize the results (or the input images) you can use the following code snippet that takes as an input a matrix that holds one vectorize image or basis vector on each row, as well as l that tells the width and length of the image.

```
#
# If X contains the images you want to show then
# use visualize(X[:16,:], 16) to see first 16 of those
#
def visualize(image, l):
    if image.ndim == 1:
        image = np.array([image])
    cols = int(np.ceil(np.sqrt(image.shape[0])))
    img_number = 0
    for row in xrange(0, cols):
        for col in xrange(0, cols):
            if img_number > image.shape[0] - 1:
                break
            else:
                ax = plt.subplot2grid((cols, cols), (row, col))
                ax.axes.axes.get_xaxis().set_visible(False)
                ax.axes.axes.get_yaxis().set_visible(False)
                imgplot = ax.imshow(image[img_number].reshape(l, l), cmap='Greys
_r')
                imgplot.set_interpolation('nearest')
                ax.xaxis.set_ticks_position('top')
                ax.yaxis.set_ticks_position('left')
                img_number += 1
    plt.show()
```