67750: Advanced Practical Machine Learning 2018/9

Exercise 1 - Unsupervised Image Denoising

Prof. Yair Weiss *TA: Daniel Gissin*

preface and grading

I found this exercise interesting and I still use the concepts presented here in other contexts. I hope you will find it interesting too. Pay attention that the GSM and ICA models were not presented in class, there is a very good explanation of them in the third and forth page of this exercise.

This exercise is hard so you can be very proud to submit partial solution. In order to encourage you to solve it, We promise that all the submissions will get at least 70, including partial submissions. In fact, since we normalize the grades to gaussian around 85 and we expect to get almost no full submissions, most of the partial submissions will get between 80 to 90, and some of them above 90.

In particular, the ICA model is way more complex, and we will assign only 5 points for implementing it.

Good luck,

Omri, 10.11.2020

1 Theoretical Questions

1.1 MLE in the EM algorithm

In class we saw that the expected log likelihood function for the Gaussian mixture model can be written as:

$$\mathbb{E}[\ell(S,\theta)] = \sum_{i=1}^{N}\sum_{y=1}^{k} c_{i,y} log\big(\pi_y \mathcal{N}(x_i;\mu_y,\Sigma_y)\big)$$

Show that the MLE of the multinomial distribution of our mixture weights is indeed:

$$\pi_y = \frac{1}{N}\sum_{i=1}^{N} c_{i,y}$$

Hint: don't forget to the constraint on π_y...

1.2 MLE in the GSM Model

The GSM model assumes that image patches are samples from a mixture of k Gaussians with the distribution $\mathcal{N}(0, r_y^2\Sigma)$ for a shared Σ between the Gaussians. Show that the maximization update

for r_y^2 is:

$$r_y^2 = \frac{\sum_{i=1}^n c_{i,y} x_i^T \Sigma^{-1} x_i}{d \sum_{i=1}^n c_{i,y}}$$

Here are some linear algebra identities and the Gaussian distribution to help get you started (A is an $n \times n$ matrix, c is a scalar):

$$\mathcal{N}(x; \mu, \Sigma) = \frac{1}{\sqrt{|2\pi\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

$$det(cA) = c^n det(A)$$

$$(cA)^{-1} = c^{-1} A^{-1}$$

Hint: simplify the expression as much as you can before taking the derivative.

1.3 EM Initialization

Yossi didn't listen in class and decided to initialize all of the Gaussians in his GMM to have the same parameters:

$$\forall y : \pi_y = \frac{1}{k}, \mu_y = \mu, \Sigma_y = \Sigma$$

Write down the first two iterations of the EM algorithm for this initialization. What's the problem?

2 Practical Exercise

Please submit a single tar file named "ex1_<YOUR_ID>". This file should contain your code, along with an "Answers.pdf" file in which you should write your answers to the theoretical and practical sections and provide all figures/data to support your answers (write your ID in there as well, just in case). Your code will be checked manually so please write readable, well documented code.

If you have any constructive remarks regarding the exercise (e.g. question X wasn't clear enough, we lacked the theoretical background to complete question Y...) we'll be happy to read them in your Answers file.

2.1 Model Reminder

In class we learned how to do non-blind denoising under the assumption that $x \sim GMM$:

- For learning $p_\theta(x)$, we use the EM algorithm to estimate $\theta = \{\pi_i, \mu_i, \Sigma_i\}_{i=1}^k$.
- For denoising, given a noisy input y, we use a weighted average of k different Wiener filters and calculate the optimal estimate of the clean image x:

$$Weiner(y) = \left(\Sigma^{-1} + \frac{1}{\sigma^2} I\right)^{-1} \left(\Sigma^{-1}\mu + \frac{1}{\sigma^2} y\right)$$

$$x^* = \sum_{i=1}^{k} c_i Weiner_i(y)$$

ההסתברות לכל גאוסיאן (הסתברות מותנה)

c_i here is the posterior probability that our image patch came from Gaussian i. This posterior probability is calculated in a similar way to how we calculate the $c_{i,y}$s in the EM algorithm, it is $\mathbb{P}(Gaussian = i|y)$ where y is our noisy image patch. That can be calculated using Bayes rule:

$$c_i = \frac{\mathbb{P}(Gaussian = i, y)}{\sum_{j=1}^{k} \mathbb{P}(Gaussian = j, y)} = \frac{\mathbb{P}(Gaussian = i)\mathbb{P}(y|Gaussian = i)}{\sum_{j=1}^{k} \mathbb{P}(Gaussian = j)\mathbb{P}(y|Gaussian = j)} \rightarrow$$

$$c_i = \frac{\pi_i \mathcal{N}(y; \mu_i, \Sigma_i + \sigma^2 I)}{\sum_{j=1}^{k} \pi_j \mathcal{N}(y; \mu_j, \Sigma_j + \sigma^2 I)}$$

2.1.1 Gaussian Scale Mixture (GSM)

Here, instead of assuming that the images were samples from a mixture of any k Gaussians, we will assume that the k Gaussians all share the same covariance matrix, and the only thing that is different is the scale by which the covariance is multiplied. In other words, we will assume our patches are first sampled from a Gaussian with distribution $\mathcal{N}(0, \Sigma)$, and then a coin is flipped to determine by which of the k possible scalars will x be multiplied:

$$x \sim \mathcal{N}(0, r_i^2 \Sigma)$$

This model is less expressive than the normal Gaussian mixture model, but what we lose in expressiveness we gain in making the convergence of EM to the optimal solution more likely. There are also other justifications for this model which we didn't go into, that stem from the statistics of natural images.

In this model, the parameters that we need to learn are the parameters Σ, π and the k possible scalars $\{r_i\}_{i=1}^k$. To make things simpler for us, **we will fix Σ to be the sample covariance matrix of our data** (like we do in the single multivariate Gaussian MLE calculation). The rest of the parameters will be learned using the EM algorithm, where the only new parameters that you haven't seen before are the scalars $\{r_i\}_{i=1}^k$. Those can be learned in the M-step of the EM algorithm using the following update (see theoretical questions):

$$r_y^2 = \frac{\sum_{i=1}^{n} c_{i,y} \cdot x_i^T \Sigma^{-1} x_i}{d \cdot \sum_{i=1}^{n} c_{i,y}}$$

Here $c_{i,y}$ is like we saw in class, the probability that x_i came from Gaussian y. d is simply the dimension of the Gaussian (64 if we're dealing with 8×8 patches).

Denoising for this model is identical to denoising for GMM, where now the covariance matrices will be $\{r_i^2 \Sigma\}_{i=1}^k$.

2.1.2 Independent Component Analysis (ICA)

ICA is a general model which is used in many fields, where we want to separate a multivariate variable into maximally independent sub-components. This is done by assuming each of the variables

is created using a fixed linear combination of scalar independent, non Gaussian random variables. For our purposes, we will assume that there are d independent scalar random variables $\{s_i\}_{i=1}^{d}$, each distributed using a 1D GMM of k Gaussians ($k \geq 2$). We then assume our image patch is sampled in the following way:

$$x = As$$

This means that we first sample the s variables (each from a different mixture of k Gaussians), and then we take the vector s and multiply it by some square matrix A to get our image patch x (also d dimensional).

It is possible to learn both the parameters of the mixture of Gaussians from s and A together using maximum likelihood estimation, but we'll make things simple and use a heuristic for estimating A, and only learn the parameters of the uni-variate mixture of Gaussians. Assuming Σ is the empirical covariance matrix of our data, we will choose A to be the following matrix:

$$\Sigma = P\Lambda P^T$$

$$A = P$$

So simply diagonalizing our sample covariance can give us our A.

Now that we've made these assumptions, we can simplify our problem by denoising in the s-domain instead of the x-domain. Since $s = A^{-1}x = P^T x$, we can look at any image in the s domain by multiplying it by P^T.

Assuming we know the parameters of the d mixtures of k uni-variate Gaussians, we can take any noisy image y and perform the following denoising procedure:

1. Move y to the s domain: $s_{noisy} = P^T y$.
2. Denoise each of the coordinates of s_{noisy} separately using the Wiener filter to get $s_{denoised}$.
3. Move back to the image patch domain to get our denoised image: $x_{denoised} = P \cdot s_{denoised}$.

To learn the parameters of the d mixtures of k uni-variate Gaussians, we perform the following learning procedure:

1. Calculate P from our training set by diagonalizing the empirical covariance matrix.
2. Transform our training set to the s domain: $s = P^T x$
3. Learn each of the d coordinates in the s domain separately using the EM algorithm, assuming each coordinate is sampled from a mixture of k uni-variate Gaussians.

2.2 Working with Image Patches

In order to avoid having to invert huge matrices, we will work with small image patches in this exercise (instead of full images). This means that we will assume that each image was created by stitching together $d \times d$ sized image patches (d will usually be 8) which were sampled from one the distribution of one of our models.

The actual procedure of building an image patch data set from full-sized images, along with the procedure of denoising a full-sized image using a model of image patches, is supplied in the supplementary code. You may go over the code if you want to learn how this can be done, but in general the idea is that every pixel in the noisy image is denoised by denoising the image patch around it and returning the single pixel's value after the full patch was denoised.

2.3 Exercise Requirements

The three models that you will need to implement in this exercise are the single multivariate Gaussian model (MVN), the GSM model and the ICA model. You will begin by implementing functions for learning the three models from the original training data. Then, you will implement functions for testing how the training set is explained by your model with log likelihood calculation functions. Finally, you will test your models by denoising images and checking how effective your models were. The overall requirements will be:

1. Write a function for evaluating the log likelihood for each model. You may use the NumPy basic Gaussian likelihood and log likelihood functions here.
2. Write a function for learning each model using the EM algorithm. Show plots of the log likelihood of the data as a function of the EM iterations and show EM indeed converges.
3. Write a function for denoising pictures for each model.
4. Provide a comparison of the three models in terms of run time, the log-likelihood of the clean test set and the MSE of the reconstruction of the test set images. Also, write which model you think is best and why. Provide qualitative evidence to support your conclusions (i.e. noisy and denoised images).
5. Provide a basic main function which demonstrates your implementation.

The supplementary code, along with the function you are required to implement, can be found in the "Image Denoising.py" file.

2.4 Data

A pickle data set of natural images can be found along with the supplementary code (separate data set for training and testing) - you can either use this dataset, or any other collection of images you wish. In any event, be sure to transform the images to gray-scale, re-scale the values to $[0, 1]$, and remove the mean of the dataset (you may use the supplied function that does that). There is also a function called "images_example" which demonstrates how to load the data and plot it.

2.5 Supplementary Code & Function Headers

This exercise comes with some helper functions that should make handling the images a little bit easier, so you can focus on the actual machine learning task. You may use the supplied code as it is or change it as you see fit (just make sure to let us know in your "Answers.PDF" file

if you changed anything). The supplied code allows you to standardize your images (using the "grayscale_and_standardize" function) and sample patches from a given image (using the "sample_patches" function). You can look at the "images_example" function to see how you can load the data so that it's ready for denoising (the "sample_patches" function also does the standardizing.

We also provide you with a function for denoising an image given pointers to your denoising function and model (using the "denoise_image" function) and a basic function for testing your implementation ("test_denoising"), but you are encouraged to test your code in other ways (for example, by creating synthetic data and seeing if your EM implementation learn the original parameters). You may look at the function headers to learn more about these supplementary functions.

Finally, we have also supplied you with different basically implemented classes for the different models. That being said, you may implement this exercise in any way you want (you can change any function signature you want, add functions, delete functions and so on), just as long as your code is readable and clear (and works).

2.6 Final Advice

2.6.1 Log Space

As discussed in class, since our image patches are high dimensional, you may experience numerical instabilities when calculating the likelihood of patches. To deal with these issues, it would be wise to work in log space, saving each likelihood in its log likelihood form. Scipy has a few functions that can help with these kinds of calculation, like "logsumexp" which accepts arrays in log space and calculates $log(\sum_i exp(x_i))$ (basically a sum function that keeps things in log space). You can see an example of using this kind of function in the supplied function "normalize_log_likelihood", which normalizes a matrix in log space in a specified axis. You may use this function in your implementation as well.

Once you finish your calculations in log-space and you need to access the actual probabilities, just take the exponent.

2.6.2 Efficient Coding

As in most machine learning tasks, there is quite a bit of matrix and vector manipulation in this exercise. Try and avoid loops as much as possible, and use NumPy functions as much as you can.

Still, your run time might not be too fast, especially in the ICA model which has to run many EMs in order to learn the parameters. We suggest pickling your models during training so you can easily test your trained models without having to keep retraining them all the time.