ECE521: Inference Algorithms and Machine Learning
University of Toronto

Assignment 3:
Unsupervised Learning and Probabilistic Models

TA: Use Piazza for Q&A
Due date: Mar. 24 11:59 pm, 2017
Electronic submission to: ece521ta@gmail.com

General Note:

- In this assignment, you will implement learning and inference procedures for some of the probabilistic models described in class, apply your solutions to some simulated datasets, and analyze the results.
- Full points are given for complete solutions, including justifying the choices or assumptions you made to solve the question. Both complete source code and program outputs should be included in the final submission.
- Homework assignments are to be solved in the assigned groups of two or three. You are encouraged to discuss the assignment with other students, but you must solve it within your own group. Make sure to be closely involved in all aspects of the assignment.

1 K-means

K-means clustering is one of the most widely used data analysis algorithms. It is used to summarize data by discovering a set of data prototypes that represent clusters of data. The data prototypes are usually referred to as cluster centers. Usually, K-means clustering proceeds by alternating between assigning data points to clusters and then updating the cluster centers. In this assignment, we will investigate a different learning algorithm that directly minimizes the K-means clustering loss function.

1.1 Learning K-means [10 pt.]

The K cluster centers can be thought of as K, D-dimensional parameter vectors and we can place them in a $K \times D$ parameter matrix $\boldsymbol{\mu}$, where the k^{th} row of the matrix denotes the k^{th} cluster

center $\boldsymbol{\mu}_k$. The goal of K-means clustering is to learn $\boldsymbol{\mu}$ such that it minimizes the loss function, $\mathcal{L}(\boldsymbol{\mu}) = \sum_{n=1}^{B} \min_{k=1}^{K} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|_2^2$. Even though the loss function is not smooth due to the "min" operation, one may still be able to find its solutions through iterative gradient-based optimization. The "min" operation leads to discontinuous derivatives, in a way that is similar to the effect of the ReLU activation function, but nonetheless, a good gradient-based optimizer can work effectively.

1. Is the loss function $\mathcal{L}(\boldsymbol{\mu})$ convex in $\boldsymbol{\mu}$? Why or why not? Give a rigorous explanation. [3 pt.]

2. For the dataset *data2D.npy*, set $K = 3$ and find the K-means clusters $\boldsymbol{\mu}$ by minimizing the $\mathcal{L}(\boldsymbol{\mu})$ using the gradient descent optimizer. The parameters $\boldsymbol{\mu}$ should be initialized by sampling from the standard normal distribution. Include a plot of the loss vs the number of updates. Hints: you may want to use the Adam optimizer for this assignment with following hyper-parameter *tf.train.AdamOptimizer(LEARNINGRATE, beta1=0.9, beta2=0.99, epsilon=1e-5)*. The learning should converge within a few hundred updates. [2 pt.]

3. Run the algorithm with $K = 1, 2, 3, 4, 5$ and for each of these values of K, compute and report the percentage of the data points belonging to each of the K clusters. Comment on how many clusters you think is "best" and why? (To answer this, it may be helpful discuss this value in the context of a 2D scatter plot of the data.) Include the 2D scatter plot of data points colored by their cluster assignments. [3 pt.]

4. Hold 1/3 of the data out for validation. For each value of K above, cluster the training data and then compute and report the loss for the validation data. How many clusters do you think is best? [2 pt.]

2 Mixtures of Gaussians [20 pt.]

Mixtures of Gaussians (MoG) can be interpreted as a probabilistic version of K-means clustering. For each data vector, MoG uses a latent variable z to represent the cluster assignment and uses a joint probability model of the cluster assignment variable and the data vector: $P(\mathbf{x}, z) = P(z)P(\mathbf{x} \,|\, z)$. For B IID training cases, we have $P(\mathbf{X}, \mathbf{z}) = \prod_{n=1}^{B} P(\mathbf{x}_n, z_n)$. The Expectation-Maximization (EM) algorithm is the most commonly used technique to learn a MoG. Like the standard K-means clustering algorithm, the EM algorithm alternates between updating the cluster assignment variables and the cluster parameters. What makes it different is that instead of making hard assignments of data vectors to cluster centers (the "min" operation above), the EM algorithm computes probabilities for different cluster centers, $P(z|\mathbf{x})$. These are computed from $P(z = k|\mathbf{x}) = P(\mathbf{x}, z = k)/\sum_{j=1}^{K} P(\mathbf{x}, z = j)$.

While the Expectation-Maximization (EM) algorithm is typically the go-to learning algorithm to train MoG and is guaranteed to converge to a local optimum, it suffers from slow convergence. In this assignment, we will explore a different learning algorithm that makes use of gradient descent.

2.1 The Gaussian cluster model [8 pt.]

Each of the K mixture components in the MoG model occurs with probability $\pi^k = P(z = k)$. The data model is a multivariate Gaussian distribution centered at the cluster mean (data center) $\boldsymbol{\mu}^k \in \mathbb{R}^D$. We will consider a MoG model where it is assumed that for the multivariate Gaussian for cluster k, different data dimensions are independent and have the same standard deviation, σ^k.

1. Derive the expression for the latent variable posterior distribution of a data point $P(z \,|\, \mathbf{x})$ in terms of the MoG parameters, $\{\boldsymbol{\mu}^k, \sigma^k, \pi^k\}$. [3 pt.]

2. Modify the K-means distance function we derived above to compute the log probability density function for cluster k: $\log \mathcal{N}(\mathbf{x}\,;\, \boldsymbol{\mu}^k, {\sigma^k}^2)$ for all pair of B data points and K clusters. Include the snippets of the Python code [2 pt.]

3. Write a vectorized Tensorflow Python function that computes the log probability of the cluster variable z given the data vector $\mathbf{x}$: $\log P(z|\mathbf{x})$. The log Gaussian pdf function implemented above should come in handy. The implementation should use the provided *utils.logsumexp* function. Include the snippets of the Python code and comment on why it is important to use the log-sum-exp function instead of using *tf.reduce_sum*. [3 pt.]

2.2 Learning the MoG [12 pt.]

The marginal data likelihood for the MoG model is as follows (here "marginal" refers to summing over the cluster assignment variables):

$$P(\mathbf{X}) = \prod_{n=1}^{B} P(\mathbf{x}_n) = \prod_{n=1}^{B} \sum_{k=1}^{K} P(z_n = k) P(\mathbf{x}_n \,|\, z_n = k)$$
$$= \prod_{n} \sum_{k} \pi^k \mathcal{N}(\mathbf{x}_n\,;\, \boldsymbol{\mu}^k, {\sigma^k}^2)$$

The loss function we will minimize is the negative log likelihood $\mathcal{L}(\boldsymbol{\mu}, \sigma, \pi) = -\log P(\mathbf{X})$. The maximum likelihood estimate (MLE) is a set of the model parameters $\boldsymbol{\mu}, \sigma, \pi$ that maximize the log likelihood or, equivalently, minimize the negative log likelihood.

1. Direct gradient-based optimization appears to learn the MoG parameters without inferring the cluster assignment variables, that is, without computing $P(z|\mathbf{x})$. In fact, this inference is implicit in the gradient computation. Show that for a single training example, the gradient of the marginal log likelihood function is the expected gradient of the log joint probability under its posterior distribution, $\nabla_{\boldsymbol{\mu}} \log P(\mathbf{x}) = \sum_k P(z = k \,|\, \mathbf{x}) \nabla_{\boldsymbol{\mu}} \log P(\mathbf{x}, z = k)$. [2 pt.]

2. Implement the loss function using log-sum-exp function and perform MLE by directly optimizing the log likelihood function using gradient descent in Tensorflow. Note that the standard deviation has the constraint of $\sigma \in [0, \infty)$. One way to deal with this constraint is to replace σ^2 with $\exp(\phi)$ in the math and the software, where ϕ is an unconstrained parameter. In addition, π has a simplex constraint, that is $\sum_k \pi^k = 1$. We

can again replace this constrain with unconstrained parameter ψ through a softmax function $\pi^k = \exp(\psi^k)/\sum_{k'}\exp(\psi^{k'})$. A log-softmax function is provided for convenience, *utils.logsoftmax*. For the dataset *data2D.npy*, set $K = 3$ and report the best model parameters it has learnt. Include a plot of the loss vs the number of updates. [6 pt.]

3. Hold out 1/3 of the data for validation and for each value of $K = 1, 2, 3, 4, 5$, train a MoG model. For each K, compute and report the loss function for the validation data and explain which value of K is best. Include a 2D scatter plot of data points colored by their cluster assignments. [2 pt.]

4. Run both the K-means and the MoG learning algorithms on *data100D.npy*. Comment on how many clusters you think are within the dataset and compare the learnt results of K-means and MoG. [2 pt.]

3 Discover Latent Dimensions

3.1 Factor Analysis [Bonus: 6 pt.]

So far we have considered K-means and MoG for clustering the data. In both of these cases we assume that each data point 'belongs to' or 'is generated by' one of K prototypes or causes. In K-means, we make a hard decision about choosing one prototype for each data observation point. In MoG, we assign points to clusters in a soft way, reflecting our uncertainty about the underlying cause of each point by modelling the softmax distribution. However, these soft assignments merely represent a probabilistic view over which of the K latent causes; we still believe that only one underlying cause was used to generate each data point. In this question, we use Factor Analysis to relax this constraint: there is now no restriction on the number of latent causes that generate each point.

For the n^{th} data point, let $\mathbf{s}_n \in \mathbb{R}^K$ be a vector of real-valued latent variables that have generated the observation feature vector $\mathbf{x}_n \in \mathbb{R}^D$. We assume the distribution of the latent variables is modeled as a zero mean Gaussian with an identity covariance matrix: $p(\mathbf{s}_n) = \mathcal{N}(\mathbf{s}_n\,;\,\mathbf{0}, I)$. Our observation feature vector $\mathbf{x}_n$'s are real numbers, therefore allowing us to model the likelihood with a Gaussian as well: $p(\mathbf{x}_n|\mathbf{s}_n) = \mathcal{N}(\mathbf{x}_n\,;\,W\mathbf{s}_n + \boldsymbol{\mu}, \Psi)$. Here, $\boldsymbol{\mu}$ is the average value of the input features, W is the weight matrix that projects the K-dimensional latent variables to the D-dimensional input space and $\Psi = \begin{bmatrix} \psi_1 & 0 & \dots & 0 \\ 0 & \psi_2 & \dots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \dots & \psi_D \end{bmatrix}$ is a *diagonal* covariance matrix.

Consider the marginal likelihood defined as:

$$P(\mathbf{X}) = \prod_{n=1}^{B} P(\mathbf{x}_n) = \prod_{n=1}^{B} \int_{\mathbf{s}_n} P(\mathbf{s}_n) P(\mathbf{x}_n \,|\, \mathbf{s}_n) d\mathbf{s}_n$$
$$= \prod_{n=1}^{B} \int_{\mathbf{s}_n} \mathcal{N}(\mathbf{s}_n \,;\, \mathbf{0}, I) \mathcal{N}(\mathbf{x}_n \,;\, W\mathbf{s}_n + \boldsymbol{\mu}, \Psi) d\mathbf{s}_n$$

Intuitively, Factor Analysis is a probabilistic version of PCA in the same way as MoG to K-means.

1. Deriving the marginal log likelihood of the factor analysis model for a single training example is a Gaussian distribution with the following mean and covariance matrix: $\log P(\mathbf{x}) = \log \int_{\mathbf{s}} P(\mathbf{x} \,|\, \mathbf{s}) P(\mathbf{s}) d\mathbf{s} = \mathcal{N}(\mathbf{x} \,;\, \boldsymbol{\mu}, \Psi + WW^T)$. (You may directly quote the multivariate Gaussian results at the end of this handout.) [1 pt.]

2. Write a TensorFlow implementation that learns Factor Analysis models by directly maximizing the log likelihood function. Namely, we would like to adapt the weight matrix, the mean of the data and the data covariance matrix by maximizing the marginal log likelihood:

$$\max_{W, \boldsymbol{\mu}, \Psi} \quad \sum_{n=1}^{B} \log P(\mathbf{x}_n)$$

   Note that for the determinant of the covariance matrix, a numerical stable implementation is to use a Cholesky decomposition that is $\log \det\{A\} = \sum_i \log \operatorname{diag}\{L\}_i^2$ and L is the Cholesky factor. This trick can be implemented in TensorFlow as:

```
log_det = 2.0 * tf.reduce_sum(tf.log(tf.diag_part(tf.cholesky(A))))
```

   For the tiny hand-written digits dataset containing two classes "3" and "5" *tinymnist.npy*, train a factor analysis model by setting the number of latent dimension K = 4 and report training, validation and test marginal log likelihood. Plot each row of the learnt weight matrix as a set of 8x8 images similar to the neural network visualization in assignment 2. Comment on the visualization and discuss what kind of latent dimensions factor analysis has discovered from the dataset. (You would like to discuss what kind of variability has the weight matrix captured about the handwritten digits of "3" and "5", e.g. one latent dimension is used to model the variability of the top part of those digits.) [3 pt.]

3. Geoffrey Hinton's explanation on PCA and FA: Generate a toy dataset of 200 3-dimensional data points $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(200)}\}$ by first generating the latent states $\mathbf{s}$ from a 3-D multivariate Gaussian distribution with zero mean and identity covariance matrix $\mathbf{s} \sim \mathcal{N}(\mathbf{s} \,;\, \mathbf{0}, I)$, $\mathbf{s} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} \in \mathbb{R}^3$. Now transform the latent states to 3-dimensional observations $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ using the following formula:

$$x_1 = s_1$$
$$x_2 = s_1 + 0.001 s_2$$
$$x_3 = 10 s_3$$

Use such dataset to train a PCA with a single principle component and a factor analysis model with a single latent dimension. Show that PCA learns the maximum variance direction (i.e. x_3 direction) while FA learns the maximum correlation direction(i.e. $x_1 + x_2$ direction). [2 pt.]

Multivariate Gaussian Results

$$P(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Lambda^{-1}) \tag{1}$$

$$P(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{L}^{-1}) \tag{2}$$

$$P(\mathbf{y}) = \mathcal{N}(\mathbf{y}; \mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{L}^{-1} + \mathbf{A}\Lambda^{-1}\mathbf{A}^T) \tag{3}$$

$$P(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}; \Sigma\{\mathbf{A}^T\mathbf{L}(\mathbf{y} - \mathbf{b}) + \Lambda\boldsymbol{\mu}, \Sigma) \tag{4}$$

$$\text{where, } \Sigma = (\Lambda + \mathbf{A}^T\mathbf{L}\mathbf{A})^{-1} \tag{5}$$