CS480/680: Introduction to Machine Learning

Homework 4

Due: 11:59 pm, March 24, 2021, submit on LEARN and CrowdMark (see Piazza).

Updated March 10, 2021

Include your name and student number!

Submit your writeup in pdf and all source code in a zip file (with proper documentation). Write a script for each programming exercise so that the TAs can easily run and verify your results. Make sure your code runs!
[Text in square brackets are hints that can be ignored.]

Exercise 1: CNN Implementation (8 pts)

Note: Please mention your Python version (and maybe the version of all other packages).

In this exercise you are going to run some experiments involving CNNs. You need to know Python and install the following libraries: Pytorch, matplotlib and all their dependencies. You can find detailed instructions and tutorials for each of these libraries on the respective websites.

For all experiments, running on CPU is sufficient. You do not need to run the code on GPUs. Before start, we suggest you review what we learned about each layer in CNN, and read at least this tutorial.

1. Train a VGG11 net on the MNIST dataset. VGG11 was an earlier version of VGG16 and can be found as model A in Table 1 of this paper, whose Section 2.1 also gives you all the details about each layer. The goal is to get the loss as close to 0 loss as possible. Note that our input dimension is different from the VGG paper. You need to resize each image in MNIST from its original size 28×28 to 32×32 [make sure you understand why this is].

   For your convenience, we list the details of the VGG11 architecture here. The convolutional layers are denoted as `Conv(number of input channels, number of output channels, kernel size, stride, padding)`; the batch normalization layers are denoted as `BatchNorm(number of channels)`; the max-pooling layers are denoted as `MaxPool(kernel size, stride)`; the fully-connected layers are denoted as `FC(number of input features, number of output features)`; the drop out layers are denoted as `Dropout(dropout ratio)`:

   ```
   - Conv(001, 064, 3, 1, 1) - BatchNorm(064) - ReLU - MaxPool(2, 2)
   - Conv(064, 128, 3, 1, 1) - BatchNorm(128) - ReLU - MaxPool(2, 2)
   - Conv(128, 256, 3, 1, 1) - BatchNorm(256) - ReLU
   - Conv(256, 256, 3, 1, 1) - BatchNorm(256) - ReLU - MaxPool(2, 2)
   - Conv(256, 512, 3, 1, 1) - BatchNorm(512) - ReLU
   - Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
   - Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU
   - Conv(512, 512, 3, 1, 1) - BatchNorm(512) - ReLU - MaxPool(2, 2)
   - FC(0512, 4096) - ReLU - Dropout(0.5)
   - FC(4096, 4096) - ReLU - Dropout(0.5)
   - FC(4096, 10)
   ```

   You should use the cross-entropy loss `torch.nn.CrossEntropyLoss` at the end.

   [This experiment will take up to 1 hour on a CPU, so please be cautious of your time. If this running time is not bearable, you may cut the training set to 1/10, so only have ∼600 images per class instead of the regular ∼6000.]

2. Once you've done the above, the next goal is to inspect the training process. Create the following plots:

   (a) (1 pt) test accuracy vs the number of epochs (say $3 \sim 5$)

   (b) (1 pt) training accuracy vs the number of epochs

   (c) (1 pt) test loss vs the number of epochs

   (d) (1 pt) training loss vs the number of epochs

[If running more than 1 epoch is computationally infeasible, simply run 1 epoch and try to record the accuracy/loss after every few minibatches.]

Ans:

3. Then, it is time to inspect the generalization properties of your final model. Flip and blur the test set images using any python library of your choice, and complete the following:

   (e) (1 pt) test accuracy vs type of flip. Try the following two types of flipping: flip each image from left to right, and from top to bottom. Report the test accuracy after each flip. What is the effect? Please explain the effect in one sentence.

   You can read this doc to learn how to build a complex transformation pipeline. We suggest the following command for performing flipping:

   ```
   torchvision.transforms.RandomHorizontalFlip(p=1)
   torchvision.transforms.RandomVerticalFlip(p=1)
   ```

   Ans:

   (f) (1 pt) test accuracy vs Gaussian noise. Try adding standard Gaussian noise to each test image with variance 0.01, 0.1, 1 and report the test accuracies. What is the effect? Please explain the effect in one sentence.

   For instance, you may apply a user-defined lambda as a new transform t which adds Gaussian noise with variance say 0.01:

   ```
   t = torchvision.transforms.Lambda(lambda x : x + 0.1*torch.randn_like(x))
   ```

   Ans:

4. (2 pts) Lastly, let us verify the effect of regularization. Retrain your model with data augmentation and test again as in part 3. Report the test accuracies and explain what kind of data augmentation you use in retraining.

   Ans:

Exercise 2: Gaussian Mixture Model (GMM) (10 pts)

Notation: For a matrix A, $|A|$ denotes its determinant. For a diagonal matrix $\text{diag}(\mathbf{s})$, $|\text{diag}(\mathbf{s})| = \prod_i s_i$.

Algorithm 1: EM for GMM.

Input: $X \in \mathbb{R}^{n\times d}$, $K \in \mathbb{N}$, initialization for *model*
// *model* includes $\pi \in \mathbb{R}_+^K$ and for each $1 \le k \le K$, $\boldsymbol{\mu}_k \in \mathbb{R}^d$ and $S_k \in \mathbb{S}_+^d$
// $\pi_k \ge 0$, $\sum_{k=1}^K \pi_k = 1$, S_k symmetric and positive definite.
// random initialization suffices for full credit.
// alternatively, can initialize r by randomly assigning each data to one of the K components
Output: *model*, ℓ

for $iter = 1 : \text{MAXITER}$ **do**
  // step 2, for each $i = 1, \dots, n$
  for $k = 1, \dots, K$ **do**
    $r_{ik} \leftarrow \pi_k |S_k|^{-1/2} \exp[-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top S_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k)]$ // compute responsibility
  // for each $i = 1, \dots, n$
  $r_{i\cdot} \leftarrow \sum_{k=1}^K r_{ik}$
  // for each $k = 1, \dots, K$ and $i = 1, \dots, n$
  $r_{ik} \leftarrow r_{ik}/r_{i\cdot}$ // normalize
  // compute negative log-likelihood
  $\ell(iter) = -\sum_{i=1}^n \log(r_{i\cdot})$
  if $iter > 1$ && $|\ell(iter) - \ell(iter-1)| \le \text{TOL} * |\ell(iter)|$ **then**
    break
  // step 1, for each $k = 1, \dots, K$
  $r_{\cdot k} \leftarrow \sum_{i=1}^n r_{ik}$
  $\pi_k \leftarrow r_{\cdot k}/n$
  $\boldsymbol{\mu}_k = \sum_{i=1}^n r_{ik}\mathbf{x}_i / r_{\cdot k}$
  $S_k \leftarrow \left(\sum_{i=1}^n r_{ik}\mathbf{x}_i\mathbf{x}_i^\top / r_{\cdot k}\right) - \boldsymbol{\mu}_k\boldsymbol{\mu}_k^\top$

1. (2 pts) Derive and implement the EM algorithm for the diagonal Gaussian mixture model, where all covariance matrices are constrained to be diagonal. Algorithm 1 recaps all the essential steps and serves as a hint rather than a verbatim instruction. In particular, you must change the highlighted steps accordingly (with each S_k being a diagonal matrix), along with formal explanations. Analyze the space and time complexity of your implementation.

   [You might want to review the steps we took in class (lecture 16) to get the updates in Algorithm 1 and adapt them to the simpler case here. The solution should look like $s_j = \frac{\sum_{i=1}^n r_{ik}(x_{ij}-\mu_j)^2}{\sum_{i=1}^n r_{ik}} = \frac{\sum_{i=1}^n r_{ik}x_{ij}^2}{\sum_{i=1}^n r_{ik}} - \mu_j^2$ for the j-th diagonal. Multiplying an $n \times p$ matrix with a $p \times m$ matrix costs $O(mnp)$. Do not maintain a diagonal matrix explicitly; using a vector for its diagonal suffices.]

   To stop the algorithm, set a maximum number of iterations (say $\text{MAXITER} = 500$) and also monitor the change of the negative log-likelihood ℓ:

$$\ell = -\sum_{i=1}^n \log\left[\sum_{k=1}^K \pi_k |2\pi S_k|^{-1/2} \exp[-\tfrac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top S_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k)]\right], \tag{1}$$

   where $\mathbf{x}_i$ is the i-th column of $X^\top$. As a debug tool, note that ℓ should decrease from step to step, and we can stop the algorithm if the decrease is smaller than a predefined threshold, say $\text{TOL} = 10^{-5}$.

   Ans:

2. (2 pts) Redo Ex 2.1 with the spherical Gaussian mixture model, where each covariance matrix $S_k = s_k I$ is a multiple of the identity matrix I. Derive the update for s_k and implement the resulting EM algorithm. Analyze the space and time complexity of your implementation.

   Ans:

3. (2 pts) Redo Ex 2.1 where we fit d GMMs (each with K components) to each feature $X_{:,j}$, separately. Implement the resulting EM algorithm. Analyze the space and time complexity of your implementation.

   Ans:

4. (4 pts) Next, we apply (the adapted) Algorithm 1 in Ex 2.1 to the MNIST dataset. For each of the 10 classes (digits), we can use its (only its) training images to estimate its (class-conditional) distribution by fitting a GMM (with say $K = 5$, roughly corresponding to 5 styles of writing this digit). This gives us the density estimate $p(\mathbf{x}|y)$ where $\mathbf{x}$ is an image (of some digit) and y is the class (digit). We can now classify the test set using the Bayes classifier:

$$\hat{y}(\mathbf{x}) = \arg\max_{c=0,\ldots,9} \underbrace{\Pr(Y = c) \cdot p(X = \mathbf{x}|Y = c)}_{\propto \Pr(Y=c|X=\mathbf{x})}, \tag{2}$$

   where the probabilities $\Pr(Y = c)$ can be estimated using the training set, e.g., the proportion of the c-th class in the training set, and the density $p(X = \mathbf{x}|Y = c)$ is estimated using GMM for each class c separately. Report your error rate on the test set as a function of K (if time is a concern, using $K = 5$ will receive full credit).

   [Optional: Reduce dimension by PCA may boost accuracy quite a bit. Your running time should be on the order of minutes (for one K), if you do not introduce extra for-loops in Algorithm 1.]

   [In case you are wondering, our classification procedure above belongs to the so-called plug-in estimators (plug the estimated densities to the known optimal Bayes classifier). However, note that estimating the density $p(X = \mathbf{x}|Y = c)$ is actually harder than classification. Solving a problem (e.g. classification) through some intermediate harder problem (e.g. density estimation) is almost always a bad idea.]

   Ans: