CS480/680: Introduction to Machine Learning

Homework 3

Due: 11:59 pm, November 14, 2025, submit on LEARN and CrowdMark.

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: Adaboost (5 pts)

Recall the update rules of Adaboost:

$$p_i^t = \frac{w_i^t}{\sum_{j=1}^n w_j^t}, \quad i = 1, \dots, n \tag{1}$$

$$\epsilon_t = \epsilon_t(h_t) = \sum_{i=1}^n p_i^t \cdot [\![h_t(\mathbf{x}_i) \neq y_i]\!] \tag{2}$$

$$\beta_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t} \tag{3}$$

$$w_i^{t+1} = w_i^t \exp(-y_i \beta_t h_t(\mathbf{x}_i)), \quad i = 1, \dots, n. \tag{4}$$

Here we use i and t to index the training examples and iterations, respectively. $[\![A]\!] = 1$ if the event A holds and 0 otherwise. Here $y_i \in \{\pm 1\}$ and **we also assume** $h_t(\mathbf{x}_i) \in \{\pm 1\}$. In this exercise we offer additional insights about Adaboost.

a) (1 pt) Prove by induction that the updates defined above are equivalent to the following updates where the y's and h's are $\{0, 1\}$ rather than $\{\pm 1\}$:

$$\tilde{\beta}_t = \frac{\epsilon_t}{1-\epsilon_t} \tag{5}$$

$$\tilde{w}_i^{t+1} = \tilde{w}_i^t \tilde{\beta}_t^{1-|\tilde{h}_t(\mathbf{x}_i) - \tilde{y}_i|}, \tag{6}$$

where $\tilde{y}_i = \frac{y_i+1}{2}, \tilde{h}(\mathbf{x}_i) = \frac{h(\mathbf{x}_i)+1}{2} \in \{0, 1\}$.

[Hint: Show that p_i^t remains the same under the two seemingly different updates.]

Ans:

b) (1 pt) Recall in the logistic regression lecture we made the linear assumption on the log odds ratio:

$$\log \frac{p(y=1|\mathbf{X}=\mathbf{x})}{p(y=-1|\mathbf{X}=\mathbf{x})} \approx \mathbf{w}^\top \mathbf{x} + b. \tag{7}$$

Adaboost in effect tries to approximate the log odds ratio using *additive* functions:

$$\log \frac{p(y=1|\mathbf{X}=\mathbf{x})}{p(y=-1|\mathbf{X}=\mathbf{x})} \approx \sum_{t=1}^T \beta_t h_t(\mathbf{x}), \tag{8}$$

where each $h_t(\mathbf{x})$ is a weak classifier. Indeed, fixing $\mathbf{X} = \mathbf{x}$, prove that the minimizer of the following exponential loss

$$\min_{H \in \mathbb{R}} \quad \mathbb{E}[\exp(-yH)|\mathbf{X}=\mathbf{x}] \tag{9}$$

is (proportional to) the log odds ratio. Here the expectation is wrt the conditional distribution $p(y|\mathbf{X} = \mathbf{x})$.

[Any function can be approximated arbitrarily well by additive functions but clearly not by linear functions, thus the power of Adaboost.]

Ans:

c) (1 pt) Suppose h_t is a weak classifier whose error $\epsilon_t > 1/2$, i.e. worse than random guessing! In this case it makes sense to flip h_t to $\bar{h}_t(\mathbf{x}) = -h_t(\mathbf{x})$. Compute the error $\epsilon_t(\bar{h}_t)$ and the resulting $\bar{\beta}_t$. Do we get the same update for w in (4)? Explain.

Ans:

d) (1 pt) Adaboost is a greedy algorithm where we find the weak classifiers sequentially. At iteration t, the classifiers $h_s, s < t$ are already found along with their coefficients β_s. Suppose h_t is given by some oracle, to find the optimal coefficient β_t, we solve an empirical approximation of the exponential loss (9):

$$\min_{\beta \in \mathbb{R}} \quad \frac{1}{n} \sum_{i=1}^{n} \exp(-y_i[H_{t-1}(\mathbf{x}_i) + \beta h_t(\mathbf{x}_i)]), \tag{10}$$

where needless to say, $H_{t-1}(\mathbf{x}) := \sum_{s=1}^{t-1} \beta_s h_s(\mathbf{x})$. While it is possible to solve (10) directly, we gain more insights by defining a distribution over the training examples $(\mathbf{x}_i, y_i), i = 1, \ldots, n$:

$$p_i^t = \frac{\exp(-y_i H_{t-1}(\mathbf{x}_i))}{\sum_{i=1}^{n} \exp(-y_i H_{t-1}(\mathbf{x}_i))}, \tag{11}$$

so that we can rewrite (10) equivalently as:

$$\min_{\beta \in \mathbb{R}} \quad \hat{\mathbb{E}}_t \exp[-y\beta h_t(\mathbf{x})], \quad (\mathbf{x}, y) \sim \mathbf{p}^t. \tag{12}$$

(Here the hat notation is to remind you that this is an empirical expectation specified by $\mathbf{p}^t$ over our training data.) Let ϵ_t be defined as in (2). Prove that the optimal β in (12) is given in (3).

[Hint: We remind again that both y and $h_t(\mathbf{x})$ are $\{\pm 1\}$-valued. Split training examples according to $h_t(\mathbf{x}_i) = y_i$ or not.]

Ans:

e) (1 pt) What is the training error

$$\epsilon_{t+1}(h_t) = \sum_{i=1}^{n} p_i^{t+1} \cdot [\![h_t(\mathbf{x}_i) \neq y_i]\!] \tag{13}$$

of the weak classifier h_t on the next round $t + 1$? Justify your answer. You may assume $0 < \epsilon_t < 1$ so that all quantities are well-defined.

[Hint: This exercise should be simple, given what you have done in d). Split training examples according to $h_t(\mathbf{x}_i) = y_i$ or not.]

Ans:

Exercise 2: CNN Implementation (8 pts)

Note: Please mention your Python version (and maybe the version of all other packages). For this exercise, you may submit a Python notebook.

In this exercise you are going to run some experiments involving CNNs. You need Python, matplotlib, and either PyTorch, TensorFlow, or JAX (PyTorch is the most popular right now and is thus strongly recommended). Be sure to install all necessary 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. You may find Colab useful for running things more efficiently (for free), but note that heavy usage may result in them taking away your GPU (unless you pay). Before start, we suggest you review what we learned about each layer in CNN, and read the documentation and tutorial for your framework of choice (either PyTorch, TensorFlow,

or JAX).

a) Train a VGG11 net on the MNIST dataset (which should be loadable via, e.g., torchvision). 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 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.]

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

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

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

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

(iv) (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:

c) 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:

(i) (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.

As one resource for those working in PyTorch, 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:

(ii) (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.

Again for those working in PyTorch, one way of approaching this is to 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:

d) (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 3: 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 $k = 1, \dots, K$

  $r_{\cdot k} \leftarrow \sum_{i=1}^n r_{ik}$

  $\pi_k \leftarrow r_{\cdot k}/n$

  $\boldsymbol{\mu}_k = \sum_{i=1}^n r_{ik}\mathbf{x}_i / r_{\cdot k}$

  $S_k \leftarrow \left(\sum_{i=1}^n r_{ik}\mathbf{x}_i\mathbf{x}_i^\top / r_{\cdot k}\right) - \boldsymbol{\mu}_k\boldsymbol{\mu}_k^\top$

a) (5 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 for a simpler case, and ensure you can derive the updates in Algorithm 1. Then adapt the steps to the simpler diagonal case. 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.]

[Warning: Either in this part or the next one, you may run into issues involving NaNs. You will have to diagnose these issues and fix them.]

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

$$\ell = -\sum_{i=1}^{n} \log \left[\sum_{k=1}^{K} \pi_k |2\pi S_k|^{-1/2} \exp[-\tfrac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top S_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k)] \right], \tag{14}$$

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 TOL $= 10^{-5}$.

Run your algorithm on `gmm_dataset.csv`, for $k = 1$ to 10. Generate a plot with k on the x-axis and the negative log-likelihood of the data under the final trained model on the y-axis. What do you think the most appropriate choice of k is? Explain and justify how and why you chose this value. [You may want to focus on more than just maximizing the log-likelihood.] For your chosen value of k, report the parameters (mixing weights, mean vectors, and vectors corresponding to the diagonals of the covariance matrices) of your trained model. When reporting them, sort the components in increasing order of mixing weights.

Ans:

b) (5 pts) Next, we apply (the adapted) Algorithm 1 in part a) 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:

$$\hat{y}(\mathbf{x}) = \arg\max_{c=0,\ldots,9} \underbrace{\Pr(Y = c) \cdot p(X = \mathbf{x}|Y = c)}_{\propto \Pr(Y=c|X=\mathbf{x})}, \tag{15}$$

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: