Homework #4

CSE 446: Machine Learning
Prof. Sewoong Oh
Due: **Thursday** 12/5/2019 11:59 PM
100 points

Please review all homework guidance posted on the website before submitting to Gradescope. Please provide succinct answers along with succinct reasoning for all your answers. Points may be deducted if long answers demonstrate a lack of clarity. Similarly, when discussing the experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. In other words, all your explanations, tables, and figures for any particular part of a question must be grouped together.

The following problems can be done in any order, though the deep learning problem will get you familiar with pytorch, which may be useful for movie recommendation.

Expectation Maximization

1. *Pandora* is a streaming music company like *Spotify* that was known to buck the collaborative filtering trend[1] and instead paid an army of employees to create feature vectors for each song by hand. Suppose you work at a Pandora clone and have feature vectors $x_1, \dots, x_n \in \mathbb{R}^d$ for all n songs in your database, and a particular user, for some subset $\mathcal{S} \subset \{1, \dots, n\}$, has listened to song $i \in \mathcal{S}$ exactly $Y_i \in \{1, 2, \dots\}$ times. You would like to make a playlist for this user so you assume Y_i is Poisson distributed with mean $\mathbb{E}[Y_i|x_i] =: \lambda_i = e^{w^T x_i}$ for some weight vector $w \in \mathbb{R}^d$ reflecting the user's preferences. That is,

$$p(Y_i = y|x_i, w) = \frac{\lambda_i^y}{y!} e^{-\lambda_i} = \frac{e^{y x_i^T w}}{y!} e^{-e^{w^T x_i}}.$$

The maximum likelihood estimator is $\widehat{w} = \arg\max_w \prod_{i \in \mathcal{S}} p(y_i|x_i, w)$. The idea is that you would then construct an m song playlist out of the m songs that maximize $x_i^T \widehat{w}$.

a. *[3 points]* The estimate $\widehat{w}$ has no closed-form solution. Can the optimization problem be transformed into a *convex* optimization problem? If so, suggest a method of solving for $\widehat{w}$ given $\{(x_i, y_i)\}_{i \in \mathcal{S}}$. (Hint: one can pose this as a convex optimization problem.).

b. *[7 points]* You solve for the $\widehat{w}$ for this user and make a playlist for her. Weeks later you look at her listening history and observe that sometimes she listens to a particular set of songs and skips over others, and at some other point she listens to a different set of songs and skips over others. You have the epiphany that users are human beings whose preferences differ with their mood (e.g., music for workouts, studying, being sad, etc.). You decide she has k music moods and aim to make k playlists, one for each mood that could be modeled by a different weight vector w. The problem is that you don't know which observation $i \in \mathcal{S}$ is assigned to which mood. Describe how you would use the EM algorithm to make these k playlists by introducing additional variables z_{ij} that indicate whether song i is suitable for mood j. Your description should specify in math exactly what computations will be performed in the E step, exactly what computations will be performed in the M step, how you initialize your parameters (doesn't have to be fancy), and what your criterion for convergence is. Make sure that all quantities are defined precisely. See Murphy Ch. 11 and other references on the course website for a review of EM and ideas.

[1] Methods like matrix completion can leverage massive user-bases rating lots of items, but suffer from the "cold-start" problem: you recommend songs based on people's rating history, but to learn who would like a *new* song you need lots of people to listen to that song, but that requires you to suggest it and possibly degrade recommendation performance.

Recommendation System

2. You will build a personalized movie recommender system. There are $m = 1682$ movies and $n = 943$ users. As historical data, every user rated at least 20 movies but some watched many more; the total dataset we will use has $100,000$ total ratings from all users. The goal is to recommend movies the users haven't seen. Namely, consider a matrix $R \in \mathbb{R}^{m \times n}$ where the entry $R_{i,j}$ represents the jth user's rating on movie i on a scale of $\{1, \dots, 5\}$; a higher value represents that the user is more satisfied with the movie. We may think of our historical data as some observed entries of this matrix while many remain unknown, and we wish to estimate the unknowns (presumably, we would then suggest the movies that appear most appealing to the user). We will use the 100K MovieLens dataset available here: `https://grouplens.org/datasets/movielens/100k/`. The dataset contains lots of metadata like the titles and genre of the movies, and even information about the users like their age. While all this metadata can be useful in predicting whether a user will like a movie or not, we will ignore it and just use the ratings contained in the `u.data` file. Use this data file and the following python code to construct a training and test set:

```
import csv
import numpy as np
data = []
with open(’u.data’) as csvfile:
    spamreader = csv.reader(csvfile, delimiter=’\t’)
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])
data = np.array(data)

num_observations = len(data)   # num_observations = 100,000
num_users = max(data[:,0])+1   # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1   # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*num_observations)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train],:]
test = data[perm[num_train::],:]
```

The arrays `train` and `test` contain the user-movie-score data representing the training set and the test set, respectively. Each line takes the form "j, i, s", where j is the user index, i is the movie index, and s is the user's score in $\{1, 2, 3, 4, 5\}$ describing how much they liked the movie (higher is better), which we're calling $R_{i,j}$. Using `train` you will train a model that can predict how any user would rate any movie, if such a rating was made. You will evaluate your model based on the average squared-error on the data in `test`. Specifically, you will use `train` to build a model $\widehat{R} \in \mathbb{R}^{m \times n}$ that hopefully has small test error:

$$\mathcal{E}_{\text{test}}(\widehat{R}) = \frac{1}{|\text{test}|} \sum_{(i,j,R_{i,j}) \in \text{test}} (\widehat{R}_{i,j} - R_{i,j})^2.$$

Note that I am using $\widehat{R}$ and R above even though typically it is almost never necessary to allocate an $m \times n$ matrix to perform this calculation; doing so would be incredibly wasteful from a memory perspective since only about 6.3% of the entries of the matrix R are known in the entire dataset. It is much more efficient to just keep track of the indices and ratings that are known of than allocating the matrix.

Low-rank matrix factorization is a baseline method for personalized recommendation. It learns a vector representation $u_i \in \mathbb{R}^d$ for each movie and a vector representation $v_j \in \mathbb{R}^d$ for each user, such that the inner product $\langle u_i, v_j \rangle$ approximates the rating $R_{i,j}$. You will build a simple latent factor model.

You will implement multiple estimators and use the inner product $\langle u_i, v_j \rangle$ to predict if user j likes movie i in the test data. For simplicity, we will put aside best practices and choose hyperparameters by using those that minimize the test error. You may use fundamental operators from Numpy or PyTorch in your implementation (i.e., numpy.linalg.lstsq, SVD, autograd, etc.) but not any precooked algorithm from a package like Sci-Kit Learn. If there is a question whether some package crosses the line and is not appropriate for use, it probably is not appropriate.

a. *[5 points]* Our first estimator just pools all the users together and, for each movie, outputs as its prediction the average user rating of that movie in `train`. That is, if $\mu \in \mathbb{R}^m$ is a vector where μ_i is the average rating of the users that rated the ith movie, write this estimator $\widehat{R}$ as a rank-one matrix. What is the $\mathcal{E}_{\text{test}}$ for this simple estimator?

b. *[5 points]* In this part, we actually will allocate an $m \times n$ matrix since our data is relatively small and it is instructive. Allocate a matrix $\widetilde{R}_{i,j} \in \mathbb{R}^{m \times n}$ and set its entries equal to the known values in the training set, and 0 otherwise (that is, $\widetilde{R}$ will be a sparse matrix). For each $d = 1, 2, 5, 10, 20, 50$ let $\widehat{R}^{(d)}$ be the best rank-d approximation (in terms of squared error) approximation to $\widetilde{R}$. This is equivalent to computing the singular value decomposition (SVD) and using just the top d singular values. This learns a lower dimensional vector representation for users and movies. Refer to the lecture materials and linked notes on SVD, PCA and dimensionality reduction. You should use an efficient solver; we recommend `scipy.sparse.linalg.svds`. For each $d = 1, 2, 5, 10, 20, 50$ compute the estimator $\widehat{R}^{(d)}$ and plot the average squared error of predictions on the training set and test set on a single plot, as a function of d.

c. *[10 points]* Repeat the previous problem but instead replace the unknown entries in $\widetilde{R}$ with the average of the known entries in the training set instead of 0. Briefly, in words, compare your results to that of part a–is it surprising? One takeaway of this assignment is that there are many ways to deal with missing data, and some are much better than others.

d. *[10 points]* Replacing all missing values by a constant is not a completely satisfying solution or starting point. A more reasonable choice is to minimize the MSE (mean squared error) only on rated movies. Let's define a loss function:

$$L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) := \sum_{(i,j,R_{i,j}) \in \text{train}} (\langle u_i, v_j \rangle - R_{i,j})^2 + \lambda \sum_{i=1}^{m} \|u_i\|_2^2 + \lambda \sum_{j=1}^{n} \|v_j\|_2^2,$$

where $\lambda > 0$ is the regularization coefficient. We will implement algorithms to learn vector representations by minimizing the loss function $L(\{u_i\}, \{v_j\})$.

Note that you may need to tune the hyper-parameter λ to optimize the performance. Also note that this is a non-convex optimization problem so your initial starting point may affect the quality of the final solution (since it may just be a local minimum). Common choices for initializing the $\{u_i\}_{i=1}^m$, $\{v_j\}_{j=1}^n$ vectors are with entries drawn from `np.random.rand()` or `np.random.randn()` scaled by some scale factor $\sigma > 0$ (σ is an additional hyperparameter). Another popular initialization is to use one of the solutions from part b or c.

Alternating minimization: First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then minimize the loss function with respective to $\{u_i\}$ by treating $\{v_j\}$ as constant vectors, and then minimize the loss function with respect to $\{v_j\}$ by treating $\{u_i\}$ as constant vectors. Iterate these two steps until both $\{u_i\}$ and $\{v_j\}$ converge. Note that when one of $\{u_i\}$ or $\{v_j\}$ is given, minimizing the loss function with respect to the other part has closed-form solutions.

Try $d = 1, 2, 5, 10, 20, 50$ and plot the mean squared error of train and test as a function of d. You should never be allocating an $m \times n$ matrix for this problem. Try tuning λ and σ to coarsely optimize the test error.

e. *[10 points]* Repeat part d, but instead of using alternating minimization, use batched stochastic gradient descent.

Stochastic Gradient Descent: First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then take a minibatch of random samples from your training set and compute a gradient step, repeat until convergence. This is non-convex optimization so the initialization matters, and so does the batch size and step size η used (η is a hyperparameter). One strategy is to pick the largest constant value of η such that the loss L still tends to decrease. Another strategy is to pick a relatively large value of η and then scale it by some factor $\beta \in (0,1)$ so that $\eta \mapsto \beta\eta$ every time a number of examples are seen that exceeds the size of the training set.

Feel free to modify the loss function to, say, different regularizers if it helps reduce the test error. See `http://www.optimization-online.org/DB_FILE/2011/04/3012.pdf` for some ideas. Note that λ, σ, η and the step size are all hyperparameters.

f. *[5 points]* Briefly, in words, compare the results of parts d and e. This is an example where the loss functions are identical, but the *algorithm* used has drastic impact on how well the model overfits or generalizes to new, unseen data.

g. (Extra credit: *[5 points]*). Using any algorithm you'd like (as long as you code it up and don't just call some pre-cooked machine learning method), find an estimator and clearly describe how to reproduce it, that achieves a test error of no more than 0.9.

Deep learning architectures

3. In this problem we will explore different deep learning architectures for a classification task. Go to `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html` and complete the following tutorials

- *What is PyTorch?*
- *Autograd: automatic differentiation*
- *Neural Networks*
- *Training a classifier*

The final tutorial will leave you with a network for classifying the CIFAR-10 dataset, which is where this problem starts. Just following these tutorials could take a number of hours but they are excellent, so start early. After completing them, you should be familiar with tensors, two-dimensional convolutions (`nn.Conv2d`) and fully connected layers (`nn.Linear`), ReLu non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`); if there is any doubt of their inputs/outputs or whether the layers include an offset or not, consult the API `http://pytorch.org/docs/master/`.

A few preliminaries:

- Using a GPU may considerably speed up computations but it is not necessary for these small networks (one can get away with using one's laptop).

- Conceptually, each network maps an image $x^{in} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output layer $x^{out} \in \mathbb{R}^{10}$ where the image's class label is predicted as $\arg\max_{i=0,1,\dots,9} x_i^{out}$. An error occurs if the predicted label differs from its true label.

- In this problem, the network is trained via cross-entropy loss, the same loss we used for multi-class logistic regression. Specifically, for an input image and label pair (x^{in}, c) where $c \in \{0,1,\dots,9\}$, if the network's output layer is $x^{out} \in \mathbb{R}^{10}$, the loss is $-\log(\frac{\exp(x_c^{out})}{\sum_{c'=0}^{9}\exp(x_{c'}^{out})})$.

- For computational efficiency reasons, this particular network considers *mini-batches* of images per training step meaning the network actually maps $B = 4$ images per feed-forward so that $\widetilde{x}^{in} \in \mathbb{R}^{B\times32\times32\times3}$ and $\widetilde{x}^{out} \in \mathbb{R}^{B\times10}$. This is ignored in the network descriptions below but it is something to be aware of.

- The cross-entropy loss for a neural network is, in general, non-convex. This means that the optimization method may converge to different *local minima* based on different hyperparameters of the optimization procedure (e.g., stepsize). Usually one can find a good setting for these hyperparameters by just observing the relative progress of training over the first epoch or two (how fast is it decreasing) but you are warned that early progress is not necessarily indicative of the final convergence value (you may converge quickly to a poor local minimum whereas a different step size could have poor early performance but converge to a better final value).

- The training method used in this example uses a form of stochastic gradient descent (SGD) that uses a technique called *momentum* which incorporates scaled versions of previous gradients into the current descent direction[2]. Practically speaking, momentum is another optimization hyperparameter in addition to the step size. If this bothers you, you can obtain all the same results using regular stochastic gradient descent.

- We will not be using a validation set for this exercise. Hyperparameters like network architecture and step size should be chosen based on the performance on the test set. This is very bad practice for all the reasons we have discussed over the quarter, but we aim to make this exercise as simple as possible.

- You should modify the training code such that at the end of each epoch (one pass over the training data) you compute and print the training and test classification accuracy (you may find the running calculation that the code initially uses useful to calculate the training accuracy).

- While one would usually train a network for hundreds of epochs for it to converge, this can be prohibitively time consuming so feel free to train your networks for just a dozen or so epochs.

You will construct a number of different network architectures and compare their performance. For all, it is highly recommended that you copy and modify the existing (working) network you are left with at the end of the tutorial *Training a classifier*. For all of the following perform a hyperparameter selection (manually by hand, random search, etc.) using the test set, report the hyperparameters you found, and plot the training and test classification accuracy as a function of iteration (one plot per network). **Highly sub-optimal hyperparameter choices that lead to drastically worse error rates than your peers will result in points off (but don't over do it).**

Here are the network architectures you will construct and compare.

a. *[15 points]* Fully connected output, 0 hidden layers (logistic regression): we begin with the simplest network possible that has no hidden layers and simply linearly maps the input layer to the output layer. That is, conceptually it could be written as

$$x^{out} = W \text{vec}(x^{in}) + b$$

where $x^{out} \in \mathbb{R}^{10}$, $x^{in} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ where $3072 = 32 \cdot 32 \cdot 3$. For a tensor $x \in \mathbb{R}^{a\times b\times c}$, we let $\text{vec}(x) \in \mathbb{R}^{abc}$ be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).

b. *[15 points]* Fully connected output, 1 fully connected hidden layer: we will have one hidden layer denoted as $x^{hidden} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The nonlinearity applied to the hidden layer will be the relu ($\text{relu}(x) = \max\{0, x\}$, elementwise). Conceptually, one could write this network as

$$x^{out} = W_2 \text{relu}(W_1 \text{vec}(x^{in}) + b_1) + b_2$$

where $W_1 \in \mathbb{R}^{M\times3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$.

c. *[15 points]* Fully connected output, 1 convolutional layer with max-pool: for a convolutional layer W_1 with individual filters of size $p \times p \times 3$ and output size M (reasonable choices are $M = 100$, $p = 5$) we have that $\text{Conv2d}(x^{in}, W_1) \in \mathbb{R}^{(33-p)\times(33-p)\times M}$. Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{in}, W) + b_1$ where b_1 is parameterized in $\mathbb{R}^M$. We will then apply a relu (relu doesn't change the tensor shape) and pool. If we use a max-pool of size N (a reasonable choice is $N = 14$ to pool to 2×2 with $p = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{in}, W_1) + b_1)) \in \mathbb{R}^{\lfloor\frac{33-p}{N}\rfloor\times\lfloor\frac{33-p}{N}\rfloor\times M}$. We will then apply a fully connected layer to the output to get a final network given as

$$x^{output} = W_2 \text{vec}(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{input}, W_1) + b_1))) + b_2$$

[2] See `http://www.cs.toronto.edu/~hinton/absps/momentum.pdf` for the deep learning perspective on this method.

where $W_2 \in \mathbb{R}^{10 \times M(\lfloor \frac{33-p}{N} \rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$. The parameters M, p, N (in addition to the step size and momentum) are all hyperparameters.

d. (Extra credit: *[5 points]*) Returning to the original network you were left with at the end of the tutorial *Training a classifier*, tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully connected layers, stepsize, etc.) and train for many epochs to achieve a *test accuracy* of at least 87%.

The number of hyperparameters to tune in the last exercise combined with the slow training times will hopefully give you a taste of how difficult it is to construct good performing networks. It should be emphasized that the networks we constructed are **tiny**; typical networks have dozens of layers, each with hyperparameters to tune. Additional hyperparameters you are welcome to play with if you are so interested: replacing relu $\max\{0, x\}$ with a sigmoid $1/(1 + e^{-x})$, max-pool with average-pool, and experimenting with batch-normalization or dropout.