

# CS1952Q HW 1 Code

David Heffren

February 2023

## 1 Project Report

In this assignment, to find  $X$  and  $Y$  minimizing  $f(X, Y) = \sum_{(i,j) \in \Omega} (M_{ij} - XY_{ij}^T)^2$ , I used batch gradient descent in python (considering a single row a batch of 1), alternating between  $X$  and  $Y$ .

I did this only using the numpy and cupy packages in python. I split the given examples into 3 groups: training, validation, and test, with a split of .8/.1/.1. The basic algorithm was to: per each epoch, pick a random batch of rows, use the gradient to update the Xbatch. Then, do the same over  $Y$ . Then, repeat for a different batch. You evaluate it intermittently with the validation loss.

To deal with the issue that we didn't have all values of  $M$ , and thus wanted to ignore some terms in the sum (but to avoid using slow for loops in python), I created a matrix  $M$  with a value of 0 if we didn't have the pair  $(i,j)$  in the training data, and a boolean matrix  $B$  with 1 if we have the pair and 0 if we don't.

With this, I could rewrite the sums in matrix form, and speed up the calculations with numpy, avoiding for loops. Thus, the loss function became:

$$f(X, Y) = ||(M - XY^T) * B||^2 = \sum_{i,j} (M - XY^T * B)_{ij}^2$$
where  $*$  is the elementwise product. Note when DISPLAYING the loss, I divided it by the number of elements in our data, to reflect the test loss used by graders.

For the gradient of  $X, Y$  I calculated it in matrix form as:

$$\begin{aligned}\frac{\partial f}{\partial X} &= -2(M - XY^T * B)Y \\ \frac{\partial f}{\partial Y} &= -2(M - XY^T * B)^T X\end{aligned}$$

Note that originally I just took the gradient of the WHOLE matrix  $X$  and  $Y$  at once, and updated them that way, but this caused the problem that the gradients would explode and i'd end up with nans. So, i first implemented it by taking the gradient of each row individually (calculating the entire gradient then taking a row of that was way too inefficient), and updating  $X$  and  $Y$  row by row. However, this was too slow, so I instead updated the gradient via batches of  $X$  and  $Y$ , updating bs adjacent rows of  $X$ , then bs adjacent rows of  $Y$ . With this, I got about 1 second per epoch.

I tried a couple of different initializations for X and Y, as I first just made each element  $N(0,1)$  for both. This worked well enough. Then, i tried the rank r approximation of the SVD of M(with zeros). However, this was way too slow and didn't really work for this implementation.

To choose r, I essentially used trial and error. I wanted to pick the minimum r value s.t it achieved very good results. When r was too large, it wouldn't learn much. I ended up settling on an r of 6, although r=4 worked well, and took many fewer epochs. Anything higher than 10 didn't work well for me at all. However, this may have been because I didn't spend enough time playing with the learning rate.

To calculate the loss itself, I used the above stated loss function/number of examples, in order to match what the graders use. It essentially is the MSE but only over the given examples.

With  $r = 6$ ,  $lr = .00004$ ,  $epochs = 500$ ,  $bs = 10$ , I managed to get pretty good results:

train: .642,  
val: .6645,  
test: .664.

As you can see, this .664 is well below the threshold .8 for the assignment. Since this test data was sampled randomly from the list of all given data, it should be an accurate predictor of the randomly sampled data used by the graders.

For  $r = 4$ , my best results were train: .7265, val: .7419, test: .7432, with learning rate .00005,  $bs = 10$ ,  $epochs = 100$ . Note that this learning rate became too large towards the end, as the training and validation loss began to increase.

I wanted to try a learning rate with exponential decay. However, for  $r=6$  the learning rate .00004 is the MAXIMUM possible without gradients exploding.

Note I originally found decent values for r on the smaller dataset, because it was loads faster to run (Especially before I optimized). It turns out the good values for r doesn't depend on the size of the dataset. However, the learning rate definitely does.

I added a regulariser which penalizes  $XY^T$  which go above 5 or below .5, in order to incentivize the results are in the right range. This decreases the test loss to around .6.

At the end, I decided to round the values below .5 to .5 and those above 5 down to 5, which makes sense given the range. I also tried rounding each answer to the nearest .5, but that turned out to make the overall loss go up rather than go down. Therefore, I just did that first rounding and turned in the result.

My best possible results were:  $b = 6$ ,  $epochs = 1000$ , regularizer with coeff 1,  $batchSize = 100$ . trainLoss = .554, valLoss = .576, testLoss = .575.