Movie Rating Predictor (Low-Rank Matrix Completion)

Vedant Gupta

February 23, 2024

To explain my code, I will start with an overview of my overall final code structure. Following this, I will explain the iterative process that got me to the final structure, including the different ideas I tried, tested, and discarded, and my final results.

1 Code Structure

I wrote all my code in a single file: script.py using Pytorch. This file is divided into the following functions and classes:

- **process_data()**: This function reads the provided file *mat_comp* and stores the data into tensors. Specifically, I store the values n, m, and k in a tensor called $parameters$. The training data is shuffled and then split into a training set and validation set (90% of the provided data is stored in the training set). I store this data as six 1D tensors: three containing the user indexes, movie indexes, and respective ratings for the training set and three analogous tensors for the validation set. Finally, I read and store the test data as two tensors containing user and movie indexes

- **load_data()**: This functions loads all the tensors stored by *process_data* and returns them. [I choose to store the input data to avoid the data processing overhead in successive runs]

- **class MovieRecoModel**: This is the main class implementing a low-rank matrix factorization model. The *__init__* function takes in the number of users/movies and the guess of low rank and accordingly initializes two matrices ($user\_to\_feature$ and $movie\_to\_feature$) the $forward$ function takes in a batch of user and movie indexes and returns the dot products of the corresponding row of $user\_to\_feature$ and $movie\_to\_feature$.

- **run_train_loop()**: This function runs a single pass over the training data. It takes in shuffled indexes of users and movies and the corresponding ratings. This is divided into batches of *batch_size*. Each batch is passed through the model and loss function (*MSELoss*), and then a backpropagation step is taken. This function returns the average training loss over the epoch.

- **run_validation()**: This function takes in the validation data and returns the MSE validation loss.
- **write_predictions()**: This function takes in the trained model along with the test data set, and writes the predictions in *mat_comp_ans* as desired.
- **train()**: This is the main function training the model. It does the following:
  - Call *load_data* to get training data
  - Initialize model, optimizer (*Adam*) and loss function
  - Iterate for *EPOCHS*, in each epoch, shuffle the training data and call *run_train_loop*. Then calculate validation loss. Print losses
  - Save the model and predictions to file.
- **main()**: This is the entry point into the program. The *mode* argument indicates whether we want to process input data (in which case *process_data* is called) or train the model. For the latter, we initialize lists of desired values for all the hyperparameters: rank, batch size, learning rate, and the amount of regularization (using the built-in *weight_decay* option with Adam). A model is trained for all possible combinations of hyperparameters and a summary of results is written to a csv file for comparison.

2 Process

Initially, I did not use regularization and used SGD and a batch size of 1. This resulted in a large gap between test and validation loss. Hence I switched to Adam and used *weight_decay* for regularization. This reduced the gap between losses and also increased rate of convergence. With a batch size of 1, training was extremely slow and noisy, and I wasn't able to complete a single epoch with multiple hours of training. Hence, refactored my code to support batching, resulting in a dramatic speedup (about 20 seconds per epoch). I also tried using SVD to initialize my parameters, but this resulted in a gradient explosion (probably due to how I assigned my parameters and the large values of the singular values). Dividing the SVD by 10 fixed this issue, but did not offer many benefits in training. Hence, I decided to stick with initializing with a standard normal distribution.

To finetune my model performance, I selected different values of the hyperparameters and trained my model over all possible combinations (total training took over 9 hours due to the number of combinations!). Based on the results, I decided to stick with a batch size of 1000, rank of 25, learning rate of 0.001 and regularization factor of 0.00001. With these parameters, I found out that validation loss is minimized after ≈ 32 epochs after which validation loss starts to increase due to overfitting. The final training and validation losses obtained were 0.051 and 0.63 respectively.

Note on AI use: I used ChatGPT to help me figure out Pytorch syntax and with a fraction of process_data.