Lab 1 Exercise - Playing with gradients and matrices in PyTorch

Jonathon Hare (jsh2@ecs.soton.ac.uk)

February 5, 2024

This is the first of a series of exercises that you need to work through **on your own** after completing the accompanying lab session. You'll need to write up your results/answers/findings and submit this to ECS handin as a PDF document along with the other lab exercises near the end of the module (1 pdf document per lab). If you are asked to implement a function, please supply a code snippet in your file.

You should use *no more* than one side of A4 to cover your responses to *this* exercise. This exercise is worth 5% of your overall module grade.

1 Implement a matrix factorisation using gradient descent

You saw (briefly) in the lectures that a form of low-rank matrix factorisation can be achieved by considering the following optimisation problem:

$$\min_{\hat{U},\hat{V}}(\|A - \hat{U}\hat{V}^\top\|_{\mathrm{F}}^2) \tag{1}$$

where $A \in \mathbb{R}^{m \times n}$, $\hat{U} \in \mathbb{R}^{m \times r}$, $\hat{V} \in \mathbb{R}^{n \times r}$ and $r < \min(m, n)$. The following pseudocode gives an algorithm that uses gradient descent to optimise this problem (note that $\hat{U}_{r,*}$ refers to the entire r-th row of $\hat{U}$):

```
Inputs:
    A: ℝ^{m×n} input matrix.
    r: rank of the factorisation.
    N: number of epochs.
    η: learning rate.

Outputs:
    Û: ℝ^{m×r} matrix initialised with values from 𝒰(0,1).
    V̂: ℝ^{n×r} matrix initialised with values from 𝒰(0,1).

Algorithm:
    For epoch = 1 to N
        For r = 1 to m
            For c = 1 to n
                e = A_rc - Û_{r,*}(V̂_{c,*})^⊤
                Û_{r,*} ← Û_{r,*} + η · e · V̂_{c,*}
                V̂_{c,*} ← V̂_{c,*} + η · e · Û_{r,*}
    return Û, V̂
```

1.1 Implement gradient-based factorisation (1 mark)

Implement the above code method using PyTorch by completing the following method:

from typing **import** Tuple

def sgd_factorise(A: torch.Tensor, rank: **int**, num_epochs=1000,
        lr=0.01) -> Tuple[torch.Tensor, torch.Tensor]:

Include the code in your report.

---

2 Compare your result to truncated SVD

A truncated Singular Value Decomposition is defined as:

$$\tilde{\boldsymbol{A}} = \boldsymbol{U}_t \boldsymbol{\Sigma}_t \boldsymbol{V}_t^{\top} \tag{2}$$

where $\tilde{\boldsymbol{A}} \in \mathbb{R}^{m \times n}$, $\boldsymbol{U} \in \mathbb{R}^{m \times t}$, $\boldsymbol{S} \in \mathbb{R}^{t \times t}$, $\boldsymbol{V} \in \mathbb{R}^{m \times t}$ and $t < \min(m, n)$.

3 Matrix completion

One of the really neat things we can do with our gradient-based approach to matrix factorisation is to perform a task called *matrix completion*. Assume for a moment that you have a problem where you have a number of observations corresponding to different instances, but that these observations are both noisy and incomplete (you don't have observations of all features for all instances). You might ask if you can use unsupervised learning to *impute* the missing observations, and (potentially) remove some of the noise — this is exactly what matrix completion is about.

Of course, to perform matrix completion, you do have to make some kind of assumption about the processes that produced the data you have. One assumption that you could make is that the process is controlled by a small number of *latent variables*[1]. Perhaps the simplest latent variable model is one based on low-rank factorisation: the chosen rank in such a case literally controls the number of latent variables, and thus the complexity of the model. We have known about such models since at least the late 1980s (where a technique called Latent Semantic Indexing was invented), but it wasn't until 2006[2] that we saw a gradient-based approach being proposed and used with a very large dataset.

The only difference between the pseudocode in Section 1, and one that works with incomplete observations is that we would only perform gradient updates where we have observations:

```
For epoch = 1 to N
    For r = 1 to m
        For c = 1 to n
            if A_rc was observed
```
$$e = \boldsymbol{A}_{rc} - \hat{\boldsymbol{U}}_{r,*}(\hat{\boldsymbol{V}}_{c,*})^{\top}$$
$$\hat{\boldsymbol{U}}_{r,*} \leftarrow \hat{\boldsymbol{U}}_{r,*} + \eta \cdot e \cdot \hat{\boldsymbol{V}}_{c,*}$$
$$\hat{\boldsymbol{V}}_{c,*} \leftarrow \hat{\boldsymbol{V}}_{c,*} + \eta \cdot e \cdot \hat{\boldsymbol{U}}_{r,*}$$
```
    return  Û, V̂
```

---

[1]you'll see many forms of latent variable model throughout this module, and also in advanced machine learning and in data-mining if you're taking them.

[2]http://sifter.org/~simon/journal/20061211.html

4 Movie Recommendation

Matrix completion can be used to solve real-world problems, such as building a recommender system. In fact, the approach you implemented in the previous section arose because of a competition to build a recommender system for *Netflix*. The original *Netflix Challenge* provided ratings of movies (on a scale of 1 to 5) given by different viewers, and the challenge was to predict how a user might rate a film that they hadn't seen before. This challenge is a specific form of recommendation called *collaborative filtering*, which it turns out can be viewed as a matrix completion problem. Collaborative filtering approaches work on the basis that people often share similar tastes — the underlying assumption being that there are a small number of latent groupings of people with similar likes and dislikes.

Consider that the raw data consists of tuples (user, movie, rating). This data could be converted into a matrix where each row corresponds to a movie, each column to a user, and the elements to the rating given by a particular user to a particular movie. Clearly not all users will have watched and rated all movies, so there will be many missing entries. Matrix completion provides a way of estimating what values those missing entries should take — essentially providing guesses for how a user would rate a movie that they haven't yet watched based on the ratings of others with similar patterns of taste.

The stochastic, one-element at-a-time, algorithm you wrote in part 3, can be used to perform matrix completion for very large datasets (it does in fact mirror the basic approach that was originally developed for the Netflix Challenge), however, the python implementation is very very slow. The following code implements a vectorised gradient descent that updates all the values at once (by summing all the updates to the U and V matrices) and is faster:

```
def gd_factorise_masked(A: torch.Tensor, M: torch.Tensor, rank: int,
        num_epochs: int=1000, lr: float=1e-5) -> Tuple[torch.Tensor, torch.Tensor]:
    U = torch.rand(A.shape[0], rank)
    V = torch.rand(A.shape[1], rank)

    for e in range(num_epochs):
        err = (A - U @ V.t()) * M
        U += lr * err @ V
        V += lr * err.t() @ U

    return U, V
```

4.1 Predict a movie rating (1 mark)

I've created a pytorch tensor that contains the ratings of the top-1000 movies from the top 10000 most active users within the original Netflix dataset. You can download a serialised version of this tensor from `http://ecs-vlc.github.io/COMP6258/labs/lab1/ratings.pt` and load it using `torch.load('ratings.pt')`. I've also created a list that maps the row-index of the ratings tensor to the title of each movie, and provided that as a CSV file at `http://ecs-vlc.github.io/COMP6258/labs/lab1/titles.csv`.

Using the `gd_factorise_masked` function (default learning rate and epochs, and a rank of 5), what rating do you predict the 5th user (column index 4) might give to the movie "A Beautiful Mind"? What is the *sum of squared errors* of the resultant matrix, computed over the valid (e.g. unmasked) values?