I. Lab One - 31298516

A. Implement Gradient-Based Factorisation

The implementation of the Gradient-Based Matrix Factorisation follows the provided pseudocode is shown in the code snippet below:

```
def sgd_factorise_sub(A: torch.Tensor, rank:
    ↪  int, epoch=1000, lr=0.01) -> Tuple[
    ↪ torch.Tensor, torch.Tensor]:
    U_estimate = torch.rand(m, rank)
    V_estimate = torch.rand(n, rank)
    for _ in range(epoch):
        for r in range(m):
            for c in range(n):
                e = A[r, c] - U_estimate[r] @
                    ↪  (V_estimate[c].T)
                U_estimate[r] = U_estimate[r]
                    ↪  + lr*e*V_estimate[c]
                V_estimate[c] = V_estimate[c]
                    ↪  + lr*e*U_estimate[r]
    return U_estimate, V_estimate
```

B. Factorise and Compute Reconstruction Error

In this problem, reconstruction error could be directly expressed as $loss = |A - \hat{U}\hat{V}^T|$ which means the absolute value of distance between A and $\hat{U}\hat{V}^T$. Then we can record that value in each epoch and draw a graph as the loss function of our optimisation process as shown in 'Figure 1', and the data occurred in this runtime will be given in the next subsection.

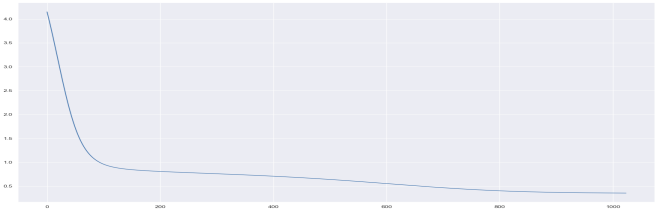

Figure 1: Curve of the loss function of our Matrix Factorisation implementation while epochs=1000 and lr=0.005.

C. Compare to the truncated-SVD

The truncated-SVD is very similar to the randomised-SVD which sacrifices part of precision for increasing the stability of solution and increase the generalisation ability. The following data could namely show the results of the reconstructed matrixes $\hat{A}$ and the final errors while factorising the matrix A by our approach and then by the randomised-SVD provided by 'Pytorch'. By making a comparison between both results, we could tell that our implementation is very close (or similar) to the official implementation.

```
[[0.2339, 0.4606, 0.3574],
[3.2534, 0.0054, 1.9679],
[3.0365, 0.5861, 2.1094]]
0.3542
[[ 0.22452754 0.5211662 0.3591739 ]
 [ 3.2530487 -0.00903386 1.9736899 ]
 [ 3.0377667 0.5983243 2.1023285 ]]
0.3492
```

D. Implement Masked Factorisation

From my point of view, we can directly set a gate to filter the values of matrix A instead of using a redundant mask matrix. Anyway, this is an excellent example of completing the broken data, which contains the thought of collaborative filtering. It is implemented in the following code snippet according to the pseudocode provided.

```
def sgd_factorise_masked(A: torch.Tensor, M:
    ↪  torch.Tensor, rank: int, epochs
    ↪ =1000, lr=0.01) -> Tuple[torch.Tensor
    ↪ , torch.Tensor]:
    for _ in range(epochs):
        for c in range(m):
            for r in range(n):
                if M[c, r] == 1:
                    pass
    return U_estimate, V_estimate
```

E. Reconstruct a Matrix

My estimate of the reconstructed matrix $\hat{A}$ and its final error from unmasked matrix A namely are:

```
[[0.3374, 0.6005, 0.1735],
[2.2843, 0.0492, 1.8374],
[2.9407, 0.6897, 2.2620]]
1.0637
```

The result shows this approach performs quite well on solving matrix completion problems. Also, the curve of its loss function is shown in 'Figure 2'.

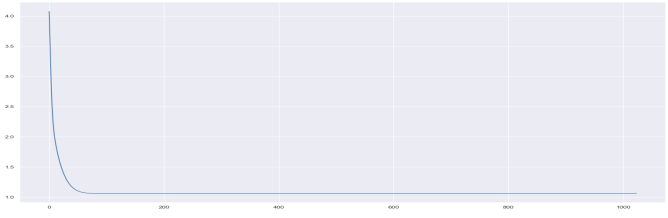

Figure 2: Loss function of the Masked Matrix Factorisation while epochs=1000 and lr=0.01.