1 Implement a matrix factorisation using gradient descent

1.1 Implementation of gradient-based factorisation, see figure 2a.

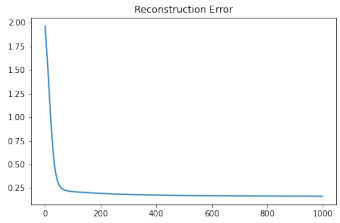

Figure 1: Reconstruction error of rank-2 factorisation.

1.2 Reconstruction loss = 0.1219

2 Compare result to truncated SVD

The difference in loss from the two method = 3.7417e-05. The difference is due to Eckart-Young theorem which states that A_k is the best approximation of A by a rank k matrix.

3 Matrix Completion

3.1 Implementation of masked factorisation, see figure 2b.

3.2 Reconstructed matrix with masked factorisation: $\begin{bmatrix} 0.3340 & 0.6009 & 0.1778 \\ 2.3819 & 0.0492 & 1.8376 \\ 2.9411 & 0.1168 & 2.2615 \end{bmatrix}$

The approximation is correct to at least 1 decimal places in all cases.

```python
def sgd_factorise(A: torch.Tensor, rank: int, num_epochs=1000, lr=0.01)
    m,n = A.shape
    u_hat = torch.rand(m, rank)
    v_hat = torch.rand(n, rank)
    err = torch.rand(num_epochs)
    for epoch in range(num_epochs):
        for r in range(m):
            for c in range(n):
                e = A[r][c] - u_hat[r] @ v_hat[c].T
                u_hat[r] = u_hat[r] + lr * e * v_hat[c]
                v_hat[c] = v_hat[c] + lr * e * u_hat[r]
        err[epoch] = e
    return (u_hat, v_hat, err)
```

(a) Gradient-based factorisation

```python
def sgd_factorise_masked(A: torch.tensor, M: torch.tensor, rank: int,
                         num_epochs=1000, lr=0.01) -> Tuple[torch.Tensor, torch.Tensor]:
    m,n = A.shape
    u_hat = torch.rand(m, rank)
    v_hat = torch.rand(n, rank)
    for epoch in range(num_epochs):
        for r in range(m):
            for c in range(n):
                if M[r,c] == 1:
                    e = A[r][c] - u_hat[r] @ v_hat[c].T
                    u_hat[r] = u_hat[r] + lr * e * v_hat[c]
                    v_hat[c] = v_hat[c] + lr * e * u_hat[r]
    return u_hat, v_hat
```

(b) Masked factorisation.

Figure 2: Implementations