II. Lab Two - 31298516

A. Implement Gradient-Based Factorisation using PyTorch's AD

Our goal is to minimise $\|A - \hat{U}\hat{V}^T\|_F^2$ and get the optimal matrixes $\hat{U}$ and $\hat{V}$ which are initialised with values from uniform distribution $\mathcal{U}(0,1)$, while A is known. Therefore it can be considered as solving a stochastic gradient descent(SGD) problem, and it has been implemented based on the automatic differentiation of 'Pytorch'. In the backpropagation process of minimising a function, the gradient(derivative) of a variable is equivalent to the error it needs to fix; hence the only thing we need to do in each epoch is to calculate the value of our goal function and to adjust the weights of $\hat{U}$ and $\hat{V}$ by letting themselves minus their gradients as shown in the following code snippet. (The mathematical inference of this approach is omitted due to the pages limitation.)

```
def gd_factorise_ad(A: torch.Tensor, rank:
    ↪ int, epochs=1000, lr=0.01) -> Tuple[
    ↪ torch.Tensor, torch.Tensor]:
    U_estimate = torch.tensor(torch.rand(
        [*A.size()][0], rank), requires_grad
            ↪ =True)
    V_estimate = torch.tensor(torch.rand(
        [*A.size()][1], rank), requires_grad
            ↪ =True)
    for _ in range(epochs):
        loss = torch.norm(A - U_estimate@(
            ↪ V_estimate.T))
        loss.backward()
        loss_list.append(float(loss))
        with torch.no_grad():
            U_estimate -= lr*U_estimate.grad
            V_estimate -= lr*V_estimate.grad
            V_estimate.grad.zero_()
            U_estimate.grad.zero_()
    return U_estimate, V_estimate
```

B. Factorise and Compute Reconstruction Error on Real Data

The 'Figure 3' below is the loss curve of our matrix factorisation approach based on SGD while testing on the iris dataset. Its final loss between the estimate $\hat{A}$ and original matrix A is 7.1619 while the one computed by truncated-SVD is 3.9024. By making a comparison between both results, the performance of our approach is still acceptable. There is a small gap between our result and the result from truncated-SVD. And as can be seen, our approach is stuck in a straight bottom line, which I think should represent the local optimum. Hence it should be helpful if we can introduce a heuristic algorithm like simulated annealing.

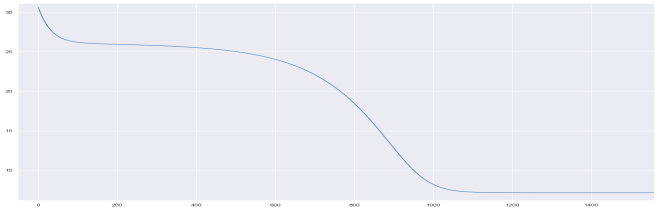

Figure 3: Loss function of the Masked Matrix Factorisation while epochs=1000 and lr=0.01.

C. Implement the MLP

As I previously mentioned, we can simply update the weights by letting themselves minus their gradients. I generate all the weights and biases matrixes randomly and under the same normal distribution, but also the thought of Xavier initialisation [**?**] is added which implemented by multiplying with $\frac{1}{\sqrt{n}}$. Plus, I divided the training set into batches with size 16 according to the idea of Masters and Luschi [**?**]. The code snippet is omitted due to the page limitation.

D. Test the MLP

The code snippet of the validation module is also shown below, and the accuracy is 0.92 while the loss curve of training process while on the iris dataset is shown in 'Figure 4'.

```
def validate(model, validation_data,
    ↪ validation_labels):
    W1,W2,b1,b2 = model[0],model[1],model
        ↪ [2],model[3]
    pred_raw = torch.relu(data_va @ W1 + b1)
        ↪  @ W2 +b2
    pred_labels = list(map(lambda x:np.
        ↪ argmax(x), pred_raw.data.numpy())
        ↪ )
    acc_valid = sum(pred_labels ==
        ↪ validation_labels.numpy())/len(
        ↪ validation_labels)
validate(train(data_tr, targets_tr),data_va,
    ↪ targets_va)
```

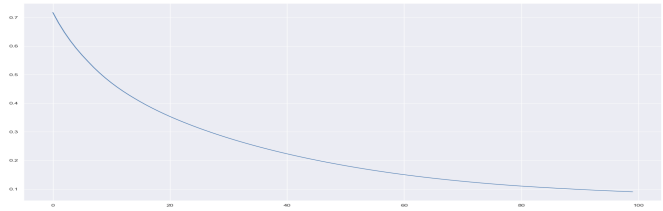

Figure 4: Loss curve of the training process with epochs=100 and lr=0.01.