Lab 2 Exercise - PyTorch Autograd

1 Implement matrix factorisation using gradient descent

1.1 Implement gradient-based factorisation using PyTorch's AD

The auto-grad version factorisation was implemented: running on the matrix A used in last lab, it turned out 0.0135 reconstruction loss after 1000 epochs of GD. Actually, it nearly stopped updating and converged on 600th epoch.

1.2 Factorise and compute reconstruction error on real data

Implemented on iris dataset, this rank-2 gradient-based factorisation ended with loss 0.025, while a rank-2 truncated Singular Value Decomposition did the same with it—0.025.

1.3 Compare against PCA

when I turn the scatter plot of SVD upside down, it become similar to PCA's result.

These orthogonal linear transforms project a matrix to lower dimensional space so that it can be described by lower-dimension matrices(or vectors) with least information loss. Furthermore, These transforms denoise the matrix in each original dimension.

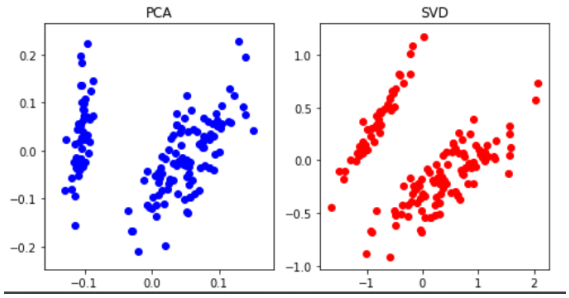

Figure 1: The scatter plots of PCA and SVD's reconstruction result

2 A simple MLP

2.1 Implement the MLP

I implemented MLP by writing 3 functions: one for forward propagation, one for training, one for evaluation.

```
import torch.nn.functional as F

def MLP(data, W1, b1, W2, b2):
    return torch.relu(data @ W1 + b1) @ W2 + b2

def train(data, lr = 0.01, numepochs = 100):
    assert data.shape[1] == 4
    W1 = torch.rand((4,12), requires_grad = True)
    W2 = torch.rand((12,3), requires_grad = True)
    b1 = torch.tensor([0.0], requires_grad = True)
    b2 = torch.tensor([0.0], requires_grad = True)
    for epoch in range(numepochs):
        logits = MLP(data, W1, b1, W2, b2)
        loss = F.cross_entropy(logits, targetstr)
        loss.backward()
        #update
        W1.data = W1.data - W1.grad * lr
        W2.data = W2.data - W2.grad * lr
        b1.data = b1.data - b1.grad * lr
        b2.data = b2.data - b2.grad * lr

        if (epoch)%200==0:
            print('epoch',epoch,' loss:',loss.item())

    return W1,b1,W2,b2

def evaluate(data, targets, W1, b1, W2, b2):
    assert data.shape[1] == 4
    logits = MLP(data, W1, b1, W2, b2)
    pred = torch.max(logits,1)
    a = torch.eq(pred[1], targets, out=None)
    print('acc:',float(a.sum())/float(a.shape[0]))
    return pred[1]


W1, b1, W2, b2 = train(datatr,0.002,2000)
print('training ', end = '')
evaluate(datatr,targetstr, W1, b1, W2, b2)
print('validation ', end = '')
evaluate(datava,targetsva, W1, b1, W2, b2)
```

Figure 2: The snippet of my code

2.2 Test the MLP

By default, model is not stable since its default learning rate is too high, leading it dependent too much of decent initial weights. However, if lr is tuned down to 0.001 and epoch up 20 times to 2000, training accuracy and validation acc stabilise quite a lot: the training accuracy is fluctuating around 95%~99%, while the former is slightly (or 1-8% experimentally) lower.

```
epoch 0   loss: 0.9812206029891968
epoch 250   loss: 0.06764069944620132
epoch 500   loss: 0.053915757685899734
epoch 750   loss: 0.04093702882528305
epoch 1000   loss: 1.1086402196269773e-07
epoch 1250   loss: 0.00037317140959203243
epoch 1500   loss: 0.00280552380718290 8
epoch 1750   loss: 0.0020774754229933023
training acc: 1.0
validation acc: 0.98
```

Figure 2: An example of my MLP implementation