Lab 2 - PyTorch Autograd

Abhilash Pulickal Scaria
aps1n21@soton.ac.uk

Introduction

This lab focuses on using PyTorch's automatic differentiation framework to implement matrix factorisation using gradient descent and a simple MLP.

1 Implement matrix factorisation using gradient descent (take 2)

1.1 Implement gradient-based factorisation using PyTorch's AD

```
from typing import Tuple
import torch
import torch.nn.functional as F

def gd_factorise_ad(A: torch.Tensor, rank: int,
    num_epochs=1000, lr=0.01) ->
    Tuple[torch.Tensor, torch.Tensor]:
  U = torch.rand((A.shape[0], rank),
      requires_grad=True)
  V = torch.rand((A.shape[1], rank),
      requires_grad=True)

  for epoch in range(num_epochs):
    U.grad, V.grad = None, None
    e = F.mse_loss(A, U @ V.T, reduction = 'sum')
    e.backward()
    U.data = U - lr * U.grad
    V.data = V - lr * V.grad

  return U, V
```

1.2 Factorise and compute reconstruction error on real data

When the above function is used to compute rank -2 factorisation of iris dataset the reconstruction loss obtained is 15.2289. This is similar to the reconstruction loss of truncated SVD (15.2288). The gradient based matrix factorisation minimises the Frobenius norm and we obtain the reconstruction loss as 15.2289. Truncated SVD also gives a similar reconstruction loss (15.2288) this observation which is similar to what we observed in Lab 1 and is explained by the Eckart-Young Theorem.

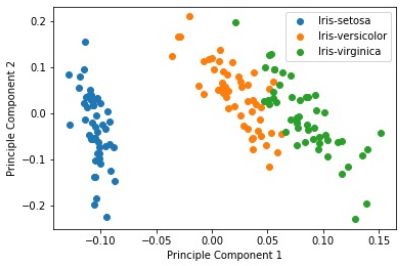
1.3 Compare against PCA

PCA is a technique which reduces dimensionality of dataset while keeping reconstruction loss to a minimum. The matrix $\boldsymbol{U}_t$ represents the projection of data from the original dataset onto its principle components. By observing $\hat{\boldsymbol{U}}$ we can see it is a scaled and rotated version of $\boldsymbol{U}_t$. So we can infer that orthogonal linear transform to lower dimensional space which maximises variance (PCA) is similar to minimising the reconstruction error.

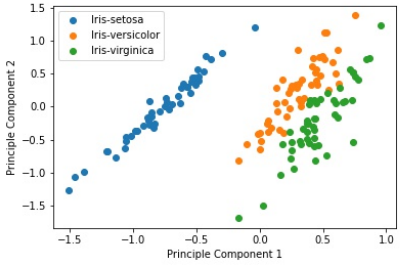
2 A simple MLP

2.1 Implement the MLP

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

def train_mlp(x_tr: torch.Tensor, y_tr:
    torch.Tensor, num_epochs: int = 100, lr: int
    = 0.01):

    W1 = torch.rand((4, 12), requires_grad=True)
    W2 = torch.rand((12, 3), requires_grad=True)
    b1 = torch.rand(1, requires_grad=True)
    b2 = torch.rand(1, requires_grad=True)

    for epoch in range(num_epochs):
        y_pred = mlp(x_tr, W1, W2, b1, b2)
        loss =
            torch.nn.functional.cross_entropy(y_pred,
            y_tr,reduction="sum")
        loss.backward()
        with torch.no_grad():
            W1 -= lr * W1.grad
            W2 -= lr * W2.grad
            b1 -= lr * b1.grad
            b2 -= lr * b2.grad

        for params in [W1, W2, b1, b2]:
            params.grad.zero_()

    return W1, W2, b1, b2
```



(a) SVD



(b) Matrix Factorisation

Figure 1: Projections of first two principle components.

2.2 Test the MLP

Table 1: Training and Validation accuracies (10 runs).

No	Training Accuracy	Validation Accuracy
0	0.96	0.82
1	0.97	0.82
2	0.65	0.68
3	0.95	0.88
4	0.96	0.82
5	0.94	0.86
6	0.95	0.82
7	0.96	0.86
8	0.96	0.82
9	0.65	0.68

It can be observed that 3rd run and final run have low accuracy. This may be due to bad initialisation of weights. Since this is a simple mlp the bad initialisation may have caused the model to converge to a local minima which resulted in the lower accuracy. But 8 out of 10 runs have good accuracy so simple MLP is a good model for this problem.