COMP6248 Lab 2 Exercise – PyTorch Autograd

Wei Chien Teoh (Eugene)
wct1c16@soton.ac.uk

29 April 2021

Introduction

The results are seeded using torch.manual_seed(0) to provide reproducible results.

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]:
    m, n = A.shape
    U = torch.rand((m, rank), requires_grad=True)
    V = torch.rand((n, rank), requires_grad=True)

    for epoch in range(num_epochs):

        U.grad, V.grad = None, None

        loss = F.mse_loss(A, U @ V.T,
            reduction='sum')

        loss.backward()

        U.data -= lr * U.grad
        V.data -= lr * V.grad

    return U, V
```

1.2 Factorise and compute reconstruction error on real data

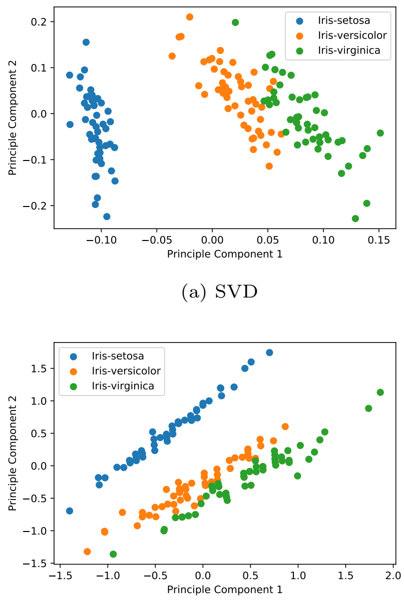
The results for both gradient-based matrix factorisation (MF) and truncated SVD are shown below.

$$\text{GD Loss} = 15.228897094726562$$
$$\text{Truncated Loss} = 15.22883415222168$$

Both results are almost identical. GD factorisation performs gradient-based minimisation on the Frobenius norm. Truncated SVD directly performs low-rank approximation by reducing the SVD to rank 2.

1.3 Compare against PCA

The principle components computed by SVD, $\boldsymbol{U}_t$ and by MF, $\hat{\boldsymbol{U}}$ are illustrated in Fig. 1a and Fig. 1b respectively. $\hat{\boldsymbol{U}}$ is observed to be a rotated and scaled version of $\boldsymbol{U}_t$. As such, maximising variance is analogous to minimising reconstruction error.



(a) SVD

(b) Matrix Factorisation

Figure 1: Projections of first two principle components.

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(data_tr: torch.Tensor, targets_tr:
    torch.Tensor, num_epochs: int = 100, lr: int
    = 0.01):
    # initialise weights and biases
    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):
        logits = mlp(data_tr, W1, W2, b1, b2)
        loss = torch.nn.functional.cross_entropy(
            logits,
            targets_tr,
            reduction="sum")
        loss.backward()

        # update weights
        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
```

2.2 Test the MLP

Table 1: Accuracies of repeated MLP training.

#	Training Accuracy	Validation Accuracy
0	0.92	0.88
1	0.66	0.66
2	0.92	0.88
3	0.71	0.70
4	0.92	0.88
5	0.92	0.88
6	0.88	0.82
7	0.95	0.88
8	0.85	0.90
9	0.77	0.72

Table 1 shows the results of the repeated MLP training. It is observed that bad random initialisation of weights and biases may majorly impact the accuracy. Examples of bad initialisation are experiments #1, #3 and #9. This may be caused by being stuck at a bad local minima or the vanishing gradient problem. The latter might be due to the weights being initialised between the values 0 to 1.