COMP6248 Lab 2 Exercise

Adrian Azzareli, aab1g18@soton.ac.uk

1 Matrix Factorisation using Gradient Descent

Task 1.1 implementation:

```
def gd_factorise_ad(A:torch.Tensor, rank:int,
                num_epochs =1000, lr =0.01) ->
                Tuple[torch.Tensor, torch.Tensor]:
  r,c = A.size()
  U = torch.rand((r, rank), requires_grad=True)
  V = torch.rand((c,rank), requires_grad=True)
  for i in range(num_epochs):
    U.grad, V.grad = None, None
    loss = torch.nn.functional.mse_loss(A,
                   U@V.t(), reduction='sum')
    loss.backward()
    U.data = U - lr*U.grad
    V.data = V - lr*V.grad
  return U, V
```

Running this algorithm with the popular *iris data set* (over eight iterations) provides us with an average reconstruction loss of $\mathcal{L}_{gd} = 15.2289$. On the same data, truncated SVD provides a reconstruction loss of $\mathcal{L}_{svd} = 15.2288$. Evidently, both results are almost identical hence the algorithms perform similarly.

Figures 1 and 2 illustrate scatter plots of the data projected onto the two principle axis' of the gradient descent algorithm and truncated SVD, respectively. It is evident that they are scaled reflection and translations of each other, thus we could infer the relationship: maximising variance while transforming into lower dimension is analogous to minimising the reconstruction error.

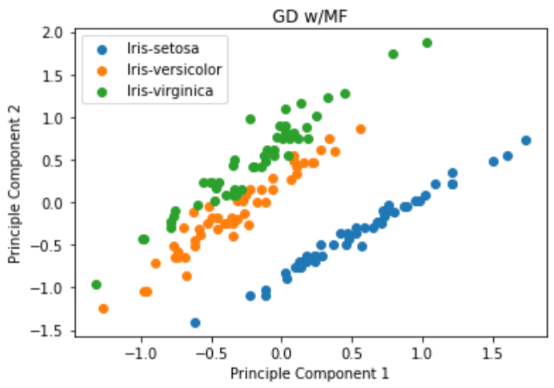

Fig. 1. Projection of data onto two principle components of Gradient Descent with Matrix Factorisation

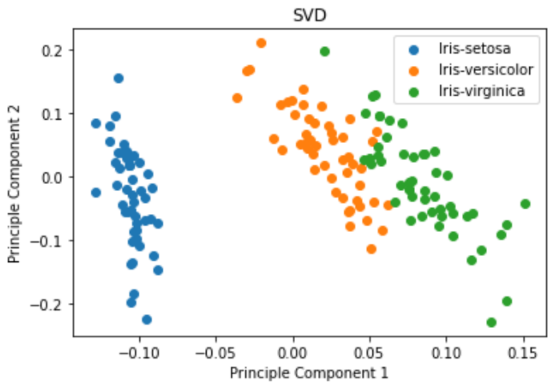

Fig. 2. Projection of data onto two principle components of SVD

2 Simple MLP

gd_train_mlp(.) was implemented to train the MLP using gradient descent. We also defined our objective function as *objective_function(.)*.

Accuracy was determined by finding the percentage of correctly estimated targets. The results in Table 1 indicate that a good training accuracy is *likely* to lead to a good validation accuracy (e.g. #3 and #6). It is also noticeable that a training accuracy of 0.69 repeatedly results in a validation of accuracy of 0.62. This suggests these iterations were similarly initialised and perhaps indicates the existence of a local minimal which they get stuck in - hence the same validation accuracy. Further investigation is necessary to be certain.

```
def gd_train_mlp(data_tr:torch.Tensor,
                 target_tr:torch.Tensor,
                 num_epochs=100, lr=0.01):
  W1 = torch.rand((4, 12),requires_grad=True)
  W2 = torch.rand((12, 3),requires_grad=True)
  b1 = torch.zeros((1),requires_grad=True)
  b2 = torch.zeros((1),requires_grad=True)
  for e in range(num_epochs):
    for p in [W1,W2,b1,b2]:
      p.grad = None # zero param grads
    loss = objective_function(data_tr,
                W1,W2,b1,b2,targets_tr)
    loss.backward()
    for p in [W1,W2,b1,b2]:
      p.data -= lr*p.grad # perform descent
  return W1, W2, b1, b2

def objective_function(data,W1,W2,b1,b2,
                       targets):
    logits = torch.relu(data@W1+b1)@W2+b2
    return torch.nn.functional.cross_entropy
                         (logits, targets)
```

Table 1. Accuracy of MLP over 8 iterations

#	Training Accuracy	Validation Accuracy
0	0.69	0.62
1	0.71	0.64
2	0.69	0.62
3	0.79	0.80
4	0.65	0.56
5	0.65	0.64
6	0.69	0.62
7	0.62	0.54