Lab 1 Exercise - Playing with gradients and matrices in PyTorch

Jonathon Hare (jsh2@ecs.soton.ac.uk)

February 5, 2020

This is the first of a series of exercises that you need to work through **on your own** after completing the accompanying lab session. You'll need to write up your results/answers/findings and submit this to ECS handin as a PDF document along with the other lab exercises near the end of the module (1 pdf document per lab).

We expect that you should use $_n o m o r e_t h a n o n e s i d e o f A 4 t o c o v e r y o u r r e s p o n s e s t o t h i s e x e r c i s e . T h i s e x e r c i s e i s w o r t h 5 \% o f y o u r o$

1 Implement a matrix factorisation using gradient descent

You saw (briefly) in the lectures that a form of low-rank matrix factorisation can be achieved by considering the following optimisation problem:

$$\min_{\hat{\boldsymbol{U}},\hat{\boldsymbol{V}}}(\|\boldsymbol{A}-\hat{\boldsymbol{U}}\hat{\boldsymbol{V}}^\top\|_{\mathrm{F}}^2) \tag{1}$$

where $\boldsymbol{A}\in\mathbb{R}^{m\times n}$, $\hat{\boldsymbol{U}}\in\mathbb{R}^{m\times r}$, $\hat{\boldsymbol{V}}\in\mathbb{R}^{m\times r}$ and $r<\min(m,n)$. The following pseudocode gives an algorithm that uses gradient descent to optimise this problem (note that $\hat{\boldsymbol{U}}_{r,*}$ refers to the entire r-th row of $\hat{\boldsymbol{U}}$):

```
Inputs:
    A: ℝ^{m×n} input matrix.
    r: rank of the factorisation.
    N: number of epochs.
    η: learning rate.

Outputs:
    Û: ℝ^{m×r} matrix initialised with values from 𝒰(0,1).
    V̂: ℝ^{n×r} matrix initialised with values from 𝒰(0,1).

Algorithm:
    For epoch = 1 to N
        For r = 1 to m
            For c = 1 to n
                e = A_rc − Û_{r,*}(V̂_{c,*})^⊤
                Û_{r,*} ← Û_{r,*} + η · e · V̂_{c,*}
                V̂_{c,*} ← V̂_{c,*} + η · e · Û_{r,*}
    return Û, V̂
```

1.1 Implement gradient-based factorisation (1 mark)

Implement the above code method using PyTorch by completing the following method:

```
from typing import Tuple

def sgd_factorise(A: torch.Tensor, rank: int, num_epochs=1000,
        lr=0.01) -> Tuple[torch.Tensor, torch.Tensor]:
```

1.2 Factorise and compute reconstruction error (1 mark)

Use your code to compute the rank-2 factorisation of the following matrix and state the result (use the default values for learning rate and number of epochs):

$$\begin{bmatrix} 0.3374 & 0.6005 & 0.1735 \\ 3.3359 & 0.0492 & 1.8374 \\ 2.9407 & 0.5301 & 2.2620 \end{bmatrix}$$

Compute the value of the reconstruction loss[a] $\|\boldsymbol{A} - \hat{\boldsymbol{U}}\hat{\boldsymbol{V}}^\top\|_{\mathrm{F}}^2$.

[a]Hint: you can use `torch.nn.functional.mse_loss` with `reduction='sum'` to do this.

2 Compare your result to truncated SVD

A truncated Singular Value Decomposition is defined as:

$$\tilde{\boldsymbol{A}} = \boldsymbol{U}_t \boldsymbol{\Sigma}_t \boldsymbol{V}_t^\top \tag{2}$$

where $\tilde{\boldsymbol{A}} \in \mathbb{R}^{m \times n}$, $\boldsymbol{U} \in \mathbb{R}^{m \times t}$, $\boldsymbol{S} \in \mathbb{R}^{t \times t}$, $\boldsymbol{V} \in \mathbb{R}^{m \times t}$ and $t < \min(m, n)$.

2.1 Compare to the truncated-SVD (1 mark)

Compute the SVD of the matrix $\boldsymbol{A}$ using `torch.svd`. Set the last singular value to zero, and compute the reconstruction. What is the error? How does it compare to the result from your algorithm above? Why is this the case[a]?

[a]Maybe google Eckart-Young theorem

3 Matrix completion

One of the really neat things we can do with our gradient-based approach to matrix factorisation is to perform a task called *matrix completion*. Assume for a moment that you have a problem where you have a number of observations corresponding to different instances, but that these observations are both noisy and incomplete (you don't have observations of all features for all instances). You might ask if you can use unsupervised learning to *impute* the missing observations, and (potentially) remove some of the noise — this is exactly what matrix completion is about.

Of course, to perform matrix completion, you do have to make some kind of assumption about the processes that produced the data you have. One assumption that you could make is that the process is controlled by a small number of *latent variables*[1]. Perhaps the simplest latent variable model is one based on low-rank factorisation: the chosen rank in such a case literally controls the number of latent variables, and thus the complexity of the model. We have known about such models since at least the late 1980s (where a technique called Latent Semantic Indexing was invented), but it wasn't until 2006[2] that we saw a gradient-based approach being proposed and used with a very large dataset.

The only difference between the pseudocode in Section 1, and one that works with incomplete observations is that we would only perform gradient updates where we have observations:

```
For epoch = 1 to N
    For r = 1 to m
        For c = 1 to n
            if A_rc was observed
                e = A_rc − Û_{r,*}(V̂_{c,*})^⊤
                Û_{r,*} ← Û_{r,*} + η · e · V̂_{c,*}
                V̂_{c,*} ← V̂_{c,*} + η · e · Û_{r,*}
return Û, V̂
```

[1]you'll see many forms of latent variable model throughout this module, and also in advanced machine learning and in data-mining if you're taking them.

[2]http://sifter.org/ simon/journal/20061211.html

3.1 Implement masked factorisation (1 mark)

Create a new version of your factorisation code that additionally takes a second matrix of the same shape of the matrix being factorised. This second matrix should be a binary mask, with 1's where the value is valid, and zeros elsewhere. Use the mask to only compute updates using the valid values.
Your implementation should have the following signature:

```
def sgd_factorise_masked(A: torch.Tensor, M: torch.Tensor, rank: int,
        num_epochs=1000, lr=0.01) -> Tuple[torch.Tensor, torch.Tensor]:
```

3.2 Reconstruct a matrix (1 mark)

Use your function to compute the rank-2 factorisation of the following matrix (again using default learning rate and number of epochs):

$$\begin{bmatrix} 0.3374 & 0.6005 & 0.1735 \\ & 0.0492 & 1.8374 \\ 2.9407 & & 2.2620 \end{bmatrix}$$

What is your estimate of the completed matrix? How does this compare to the original matrix $\boldsymbol{A}$? What does this tell you?