Lab 2 Exercise - PyTorch Autograd

Jonathon Hare & Antonia Marcu
{j.s.hare, a.marcu}@soton.ac.uk

February 6, 2025

This is the exercise that you need to work through **on your own** after completing the second 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).

You should use *no more* than one side of A4 to cover your responses to *this* exercise. This exercise is worth 5% of your overall module grade.

1 Implement matrix factorisation using gradient descent (take 2)

Last week we implemented a low-rank matrix factorisation by initially considering gradient updates to a single element at a time, and then introducing a vectorised algorithm that computed the factorisation all at once, but with manually computed gradients. Let's revisit that problem, but this time we'll use PyTorch's AD framework to compute the gradients for us on the entire problem, and then use these with a simple gradient descent algorithm. As a recap, the optimisation problem we are solving is:

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

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

1.1 Implement gradient-based factorisation using PyTorch's AD (1 mark)

Implement the optimisation problem using PyTorch's automatic differentiation framework to compute the gradients associated with the tensors $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}$, by completing the following method:

```
from typing import Tuple

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

Remember, unlike last week you won't be updating an element at a time, but rather you'll compute and apply all the gradients of $\hat{\boldsymbol{U}}$ to $\hat{\boldsymbol{U}}$, and of $\hat{\boldsymbol{V}}$ to $\hat{\boldsymbol{V}}$ once per epoch. $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}$ should be initially created with uniform random values. To compute the squared frobenius norm loss (reconstruction loss), use `torch.nn.functional.mse_loss` with `reduction='sum'`[a].

[a]Note that if you really wanted to use your implementation for real problems you would most likely choose to use `reduction='mean'`, which would decouple the learning rate from the size of the problem and make tuning easier.

1.2 Factorise and compute reconstruction error on real data (1 mark)

Use the following code to download a dataset of 150 instances and 4 features into a mean-centered tensor called `data`:

```
import pandas as pd
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases'
        +'/iris/iris.data', header=None)
data = torch.tensor(df.iloc[:, [0,1,2,3]].values)
data = data - data.mean(dim=0)
```

Use your `gd_factorise_ad` function to compute the rank-2 factorisation of the `data` matrix using the default values for learning rate and number of epochs. What is the reconstruction loss? How does this reconstruction loss compare to the loss of a rank-2 reconstruction computed using a truncated Singular Value Decomposition of the same data?

1.3 Compare against PCA (1 mark)

Given that our `data` matrix consisted of mean-centered data points encoded in each row, then the U matrix from SVD represents a projection of the data onto its principle directions (i.e. it's the result of applying PCA to data).
Create a scatter plot of the data projected onto the first two principle axes computed by SVD. Now create a second scatter plot from the data in matrix $\hat{\boldsymbol{U}}$. What do you observe? Can you infer a relationship between an orthogonal linear transform to a lower dimensional space which maximises variance (which is what PCA does), to minimising reconstruction error?

2 A simple MLP

We'll now turn our attention to a simple MLP implementation that we'll train with gradient descent. Normally, we would utilise other aspects of the PyTorch library to simplify this, however for this exercise we're just going to use raw tensors and the AD framework to track their gradients.

Firstly, we'll set up some training and validation data:

```
import pandas as pd
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases'
        +'/iris/iris.data', header=None)
df = df.sample(frac=1) #shuffle

# add label indices column
mapping = {k: v for v, k in enumerate(df[4].unique())}
df[5] = df[4].map(mapping)

# normalise data
alldata = torch.tensor(df.iloc[:, [0,1,2,3]].values, dtype=torch.float)
alldata = (alldata - alldata.mean(dim=0)) / alldata.var(dim=0)

# create datasets
targets_tr = torch.tensor(df.iloc[:100, 5].values, dtype=torch.long)
targets_va = torch.tensor(df.iloc[100:, 5].values, dtype=torch.long)
data_tr = alldata[:100]
data_va = alldata[100:]
```

Generally speaking[1], PyTorch expects the first axis of a tensor to be the batch axis; this means that in the data tensors above, each instance is in a row. The simple MLP you're going to build will have the following form:

```
logits = torch.relu(data @ W1 + b1) @ W2 + b2
```

where W1 $\in \mathbb{R}^{4,12}$, W2 $\in \mathbb{R}^{12,3}$, and both b1 and b2 $\in \mathbb{R}$. Pay attention to the broadcasting semantics of the `@` operator which performs matrix multiplication (see the `Tensor.matmul` method), and can broadcast

[1] except when dealing with recurrent models

over the batch dimension. **Note** that this isn't a normal MLP — normal MLPs have a bias term per neuron, whereas the one you are building has a single shared scalar bias per layer. Put another way, normal `Linear` layers with i inputs and o outputs would have weights of shape $\mathbb{R}^{i,o}$ and a bias *vector* of shape $\mathbb{R}^{o}$; in this case we wanted to demonstrate that its entirely possible to do something a little different to 'normal' by *structurally regularising* the model so it has less learnable parameters.

2.1 Implement the MLP (1 mark)

Given the above form of the MLP, write the code to perform 100 epochs of *gradient descent* with a learning rate of 0.01 on the training (_tr) data. Use PyTorch's cross-entropy loss (`torch.nn.functional.cross_entropy(logits, targets_tr)`) as the objective. Initialise `W1` and `W2` with normally distributed random numbers, and `b1` and `b2` as zeros. You'll need to manually apply the gradient updates to `W1`, `W2`, `b1` and `b2` each epoch.

2.2 Test the MLP (1 mark)

Run your MLP on the training data, and compute both the training and validation accuracies at the end of training. Repeat this a few times and report what you observe and what you infer from those observations.