1 Vectorized Implementation in Numpy

1.1 Short Theory

During the lab sessions so far, you have been extensively using the Numpy library, which allows efficient matrix and algebraic operations in Python, like you usually do in Matlab. Today we will look at an essential feature to further utilize the power of Numpy, namely, *vectorized implementation*. In fact, vectorization implementation is not unique to Numpy. We can apply it to Matlab as well. The basic principle is "*writing as few for loop as possible*", and make use of Numpy's vector/matrix operators as much as possible.

As a demonstration of vectorization implementation, we will consider the problem of pairwise Euclidean distance calculation between two matrices.

Suppose we have two matrices, $A \in \mathbb{R}^{n\times d}$ and $B \in \mathbb{R}^{m\times d}$. We can think of A as containing n samples of d dimension, and of B in a similar way. The task is to compute a distance matrix $D \in \mathbb{R}^{n\times m}$, where an element D_{ij}, $i = 1,\ldots,n$, $j = 1,\ldots,m$, is the Euclidean distance between $A_i \in \mathbb{R}^d$ (row i of A) and $B_j \in \mathbb{R}^d$ (row j of B). This distance calculation task is the center of many machine learning algorithm, such as *K-means clustering*, *K-nearest neighbor classification* and *ranking and retrieval*.

Recall that the Euclidean distance between vector $A_i \in \mathbb{R}^d$ and $B_j \in \mathbb{R}^d$ is calculated as follows:

$$\begin{aligned} D_{ij} &= \sqrt{(A_i^{(1)} - B_j^{(1)})^2 + (A_i^{(2)} - B_j^{(2)})^2 + \cdots + (A_i^{(d)} - B_j^{(d)})^2} \\ &= \sqrt{\sum_{k=1}^{d} \left(A_i^{(k)} - B_j^{(k)}\right)^2} \end{aligned} \tag{1}$$

1.1.1 Straightforward implementation with for loops

A straightforward way to implement a function performing this task is to loop over all pairs of row vectors between the two matrices A, B and calculate their distances using eq.(1). This approach is simple, but it is not efficient (in Numpy) as it does not utilize the highly optimized vector/matrix operations in Numpy.

1.1.2 Vectorized implementation

Expand eq. (1) further, we have:

$$
\begin{aligned}
D_{ij} &= \sqrt{\sum_{k=1}^{d} \left(A_i^k\right)^2 - 2A_i^k A_j^k + \left(B_j^k\right)^2} \\
&= \sqrt{\left(\sum_{k=1}^{d} \left(A_i^k\right)^2\right) + \left(\sum_{k=1}^{d} \left(B_j^k\right)^2\right) - 2\left(\sum_{k=1}^{d} A_i^k A_j^k\right)} \\
&= \sqrt{\|A_i\|_2^2 + \|B_j\|_2^2 - 2A_i^T B_j} \qquad (2)
\end{aligned}
$$

As eq. (2) suggests, in order to calculate the distance between a row i in A to row j in B, we need to calculate the length of A_i, the length of B_j and their inner product. A simple way to do it over all pairs of rows in A and B is, again, performing `for` loop over all i and j. However, a better way is to (i) first calculate the lengths of all rows in A, the length of all rows in B and the inner product between all pairs of rows in A and B; and (ii) sum them over and take the element-wise square root to get the distance matrix D.

1.2 Python Exercise

In this exercise, you will implement and experiment with the two approaches to compute the distance matrix D as presented above.

Step 1: Euclidean distance calculation with `for` loop. In this step, you will follow the straightforward approach, as presented in 1.1.1 to calculate the distance matrix D.

Step 2: Vectorized implementation of Euclidean distance calculation. In this step, you will follow the vectorized approach, as presented in 1.1.2 to calculate the distance matrix D. Note that a correct implementation for this step should not contain any `for` loop.

Step 3: Checking the correctness. Either approach you follow, you should get the same results for the distance matrix D (though this does not necessarily means you have correct results). Run the code cell to check this.

Step 4: Comparing the running time. Run the code cell and compare the difference in running time between the two implementations. The vectorized implementation should be significantly faster than the straightforward implementation. The speed up factor depends on the size of the matrices. With big matrices, you can have very large gain in speed using vectorized implementation.

2 Recommendation System

Recommendation System (RS) concerns the task of giving recommendations to users. Nowadays, this task is everywhere over the Internet, for example, Netflix uses its RS engine to recommend suitable movies to viewers, Amazon uses RS to recommend potential products to customers, Facebook uses RS to rank and suggest posts of interest to users, etc. As a result, research in RS has been receiving a lot of attention in recent years.

One of the most common approaches for RS is *collaborative filtering*. In this exercise, we will consider the most basic but effective approach to perform ollaborative filtering, namely *matrix factorization*.

2.1 Short Theory

Let's consider the Netflix problem: recommending movies to users. Suppose we are given a rating matrix $R \in \mathbb{R}^{n \times m}$, in which each row corresponds to a user and each column corresponds to a movie. An element R_{ij} in R corresponds to the rating user i gives to movie j. For movie rating, the rating values are normally integers in the range $1, \ldots, 5$. Since the number of movies is very large and each user can only give ratings for small number of movies, the rating matrix R is highly sparse, meaning that there is only a small portion of the entries is known. The task is to predict the unknown rating entries from the known rating entries. If we can accurately do this, we can predict the rating of a user to moviews he or she has not seen, and then give suitable recommendations.

In matrix factorization, we assume that the rating matrix R can be factorized into two matrices $U \in \mathbb{R}^{n \times d}, V \in \mathbb{R}^{m \times d}$: $R = UV^T$. U can be considered as containing the users' features, i.e. each row U_i encodes the preference of user i; while V can be considered as containing the items' features, i.e. each

row V_j encodes the characteristics of item j.

Denote Ω the set of known entries. The objective function which we need to minimize is the mean square error over the knwon entries, as follows:

$$L = \frac{1}{2}\frac{\sum_{i,j\in\Omega}\left(U_i^T V_j - R_{ij}\right)^2}{|\Omega|}, \tag{3}$$

with $|\Omega|$ the number of known entries.

Clearly if we fix V and only solve eq. (3) for U_i, we will arrive at the linear regression (least square) problem. As a result, an approach to minimize L in eq. (3) is to alternate between solving for U and for V. This approach is called *Alternating least square* (ALS). However, as we know that gradient descent is more scalable than least square, we will minimize the loss L using gradient descent method.

From Ω, we can create a mask $M \in 0, 1^{n\times m}$ for the rating matrix, such that $M_{ij} = 1$ if $ij \in \Omega$ (or R_{ij} is known. The loss in eq. (3) can be written in Numpy form as:

$$L = \frac{1}{2}\frac{\texttt{sum}\left(M \odot \left(UV^T - R\right)^{**} 2\right)}{\texttt{sum}(M)}, \tag{4}$$

where `sum` represents the summation of all elements in a matrix, $\odot$ represent the element-wise matrix multiplication.

Similar to the logistic regression case, to avoid overfitting, we need to add regularization terms to our parameters, which are U and V in this case. This results in the final loss function (in Numpy form):

$$L = \frac{1}{2}\frac{\texttt{sum}\left(M \odot \left(UV^T - R\right)^{**} 2\right)}{\texttt{sum}(M)} + \frac{1}{2}\lambda.\texttt{sum}\left(U^{**}2\right) + \frac{1}{2}\lambda.\texttt{sum}\left(V^{**}2\right) \tag{5}$$

As we are using gradient to solve for U and V, we need to calculate the gradients of the loss L in eq. 5 w.r.t. U and V. These gradients can be

expressed as follows:

$$\frac{\partial L}{\partial U} = \left(\left(UV^T - R\right) \odot M\right) V + \lambda U \tag{6}$$

$$\frac{\partial L}{\partial V} = \left(\left(UV^T - R\right) \odot M\right)^T U + \lambda V \tag{7}$$

$$\tag{8}$$

Having the gradients calculated using the equations above, we can apply gradient descent for a number of iterations to learn for U and V. Obtaining U and V, we can then make prediction for the whole rating matrix as UV^T.

2.2 Python Exercise

In this exercise, you will implement the collaborative filtering using matrix factorization algorithm. You will use a standard movie rating datasetm namely,"MovieLens100K". The dataset contains 943 users and 1682 movies. There are $100,000$ integer ratings, with values in range $[1, 5]$. As you can see, this dataset is highly sparse with only 6.3% of ratings are known. For your convenience, the function to load the dataset has been given and the dataset has been split into a training and a testing set.

Step 1: Load dataset and create mask matrix. Finish the `create_mask` function. In the original rating matrix, known entries have positive values while unknown entries are all set to 0. As a result, you can create the mask by comparing the original rating matrix with 0.

Step 2: Implement cost and gradient functions. You will implement functions *compute_cost* and *compute_gradient*. Follow the equations presented above for your implementation.

Step 3: Learn the users' features and movies' features. Use gradient descent with a predifined number of learning iteration and regularization weights to learn for U and V.

Step 4: Write evaluation function. In this step, you need to write the two functions which are commonly used to evaluate matrix factorization methods, namely the *root mean square error (RMSE)* and *mean absolute error (MAE)*. Note that both RMSE and MAE are calculated **only** over the entries in a given mask.

Step 5: Make prediction and evaluate. Use the learned features U and V to complete the rating matrix. After that, you can evaluate your results.