Supplementary Materials for Efficient Optimization Methods for Extreme Similarity Learning with Nonlinear Embeddings

Bowen Yuan
National Taiwan University
f03944049@csie.ntu.edu.tw

Yu-Sheng Li
National Taiwan University
r07922087@csie.ntu.edu.tw

Pengrui Quan
University of California, Los Angeles
prquan@g.ucla.edu

Chih-Jen Lin
National Taiwan University
cjlin@csie.ntu.edu.tw

1 MORE IMPLEMENTATION DETAILS

In evaluating $L(\boldsymbol{\theta})$, the required $\boldsymbol{P}, \boldsymbol{Q}, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c, \boldsymbol{P}_c, \boldsymbol{Q}_c$, and Frobenius inner products can be directly computed by the forward and other built-in functions. However, when we use GPUs for large data sets, we face two difficulties. First, as the memory consumption of a forward process is proportional to the number of data, calculating the whole $\boldsymbol{P}$ and $\boldsymbol{Q}$ at once may be infeasible for GPUs. Second, for later computations, we need to cache $\boldsymbol{P}$ and $\boldsymbol{Q}$ in $O((m+n)k)$ space, which may also be infeasible.

For the first difficulty, fortunately, as the computation of each row of $\boldsymbol{P}$ and $\boldsymbol{Q}$ is independent of others, we follow [2] to have a mini-batch setting. Specifically, by splitting $\mathbb{U}$ and $\mathbb{V}$ into multiple subsets, we sequentially calculate rows of $\boldsymbol{P}$ or $\boldsymbol{Q}$ corresponding to indices in each subset. For $\tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c, \boldsymbol{P}_c$, and $\boldsymbol{Q}_c$, each of which is the sum of m or n matrices of size $k \times k$, we can calculate the partial results on each subset and accumulate them for the final output.

To overcome the second difficulty, we apply a heterogeneous computing scheme by storing $\boldsymbol{P}$ and $\boldsymbol{Q}$ in CPU, which has a larger memory capacity than GPU. After finishing the above computation in GPU on each subset, we move the resulting partial $\boldsymbol{P}$ and $\boldsymbol{Q}$ to the CPU memory. Some subsequent sparse operations such as $L^+(\boldsymbol{\theta})$ in (10) are conducted in CPU. As the advantage of GPU over CPU is more significant on dense operations, our setting of having sparse operations on CPU is appropriate. On the contrary, for $\tilde{\boldsymbol{P}}_c, \tilde{\boldsymbol{Q}}_c, \hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c, \boldsymbol{P}_c$, and $\boldsymbol{Q}_c$, we store these $k \times k$ matrices in the GPU memory. Then we compute $L^-(\boldsymbol{\theta})$ involving dense operations of Frobenius inner products on GPU.

To compute $\nabla L(\boldsymbol{\theta})$ in (27), similar to $L^+(\boldsymbol{\theta})$, $\boldsymbol{XQ}$ and $\boldsymbol{X}^\top \boldsymbol{P}$ are computed on CPUs. Then by applying the mini-batch setting, for each subset, we feed the corresponding part of $\boldsymbol{XQ}$, $\boldsymbol{X}^\top \boldsymbol{P}$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{P}$, $\boldsymbol{Q}$, $\tilde{\boldsymbol{P}}$ and $\tilde{\boldsymbol{Q}}$ into GPUs. With the cached $\hat{\boldsymbol{P}}_c, \hat{\boldsymbol{Q}}_c, \boldsymbol{P}_c$, and $\boldsymbol{Q}_c$, we first conduct dense operations to compute $\boldsymbol{XQ} + \omega(\boldsymbol{APQ}_c - \boldsymbol{A}\tilde{\boldsymbol{P}}\hat{\boldsymbol{Q}}_c)$ and $\boldsymbol{X}^\top \boldsymbol{P} + \omega(\boldsymbol{BQP}_c - \boldsymbol{B}\tilde{\boldsymbol{Q}}\hat{\boldsymbol{P}}_c)$ on GPU. Then we call the automatic differentiation function (e.g., `tf.gradients` in TensorFlow) to finish the transposed Jacobian-vector products.

To compute $\boldsymbol{Gd}$ in (40), as we mentioned, the Jacobian-vector products can be implemented with the forward-mode automatic differentiation. However, for compatibility with older versions of libraries where the forward-mode automatic differentiation is not supported, we apply the trick provided in [1] to compute $\boldsymbol{W}$ and $\boldsymbol{H}$ with the reverse-mode automatic differentiation. Similar to $\boldsymbol{P}$ and $\boldsymbol{Q}$, we apply the mini-batch setting and cache $\boldsymbol{W}$ and $\boldsymbol{H}$ in the CPU memory. Then with the cached $\boldsymbol{W}$, $\boldsymbol{H}$, $\boldsymbol{P}$, and $\boldsymbol{Q}$, we compute $\boldsymbol{ZQ}$ and $\boldsymbol{Z}^\top \boldsymbol{P}$ on CPUs. Next, similar to $\nabla L(\boldsymbol{\theta})$, by the mini-batch setting, for each subset, we feed the required partial matrices to compute $\boldsymbol{AWQ}_c + \boldsymbol{APH}_c$ and $\boldsymbol{BHP}_c + \boldsymbol{BQW}_c$ in GPU. Finally, we call the automatic differentiation function to finish the remaining transposed Jacobian-vector products.

REFERENCES

[1] Jamie Townsend. A new trick for calculating Jacobian vector products, 2017.
[2] Chien-Chih Wang, Kent-Loong Tan, Chun-Ting Chen, Yu-Hsiang Lin, S. Sathiya Keerthi, Dhruv Mahajan, Sellamanickam Sundararajan, and Chih-Jen Lin. Distributed Newton methods for deep learning. *Neural Comput.*, 30:1673–1724, 2018.