

Supplementary Material for: Data-Aware Low-Rank Compression for Large NLP Models

A PROOF OF THEOREM 1

Theorem 1. Assume $\text{rank}(W) = r$ and $\text{rank}(X) = t$. The closed form solution M^* of the optimization problem in equation 2 is

$$M^* = V_{W,r} S_{W,r}^{-1} Z_k S_{X,t}^{-1} V_{X,t}^T, \quad (7)$$

where Z_k is the rank- k truncated SVD of $Z = S_{W,r} V_{W,r}^T V_{X,t} S_{X,t}$.

Proof. We firstly consider the unconstrained problem:

$$\begin{aligned} M^* &= \underset{M}{\operatorname{argmin}} \|WX - WMX\|_F^2 \\ &= \underset{M}{\operatorname{argmin}} \|U_W^T W X U_X - U_W^T W M U_X\|_F^2 \\ &= \underset{M}{\operatorname{argmin}} \|S_W V_W^T V_X S_X - S_W V_W^T M V_X S_X\|_F^2, \end{aligned}$$

where the second equality holds due to the fact that U_W and U_X are orthonormal matrices. Note that we could expand the term $S_W V_W^T V_X S_X$ as:

$$\begin{aligned} S_W V_W^T V_X S_X &= \begin{bmatrix} S_{W,r} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{W,r}^T \\ \bar{V}_{W,r}^T \end{bmatrix} \begin{bmatrix} V_{X,t} & \bar{V}_{X,t} \end{bmatrix} \begin{bmatrix} S_{X,t} & 0 \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} S_{W,r} V_{W,r}^T & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{X,t} S_{X,t} & 0 \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} S_{W,r} V_{W,r}^T V_{X,t} S_{X,t} & 0 \\ 0 & 0 \end{bmatrix}. \end{aligned}$$

Similarly, we will have

$$S_W V_W^T M V_X S_X = \begin{bmatrix} S_{W,r} V_{W,r}^T M V_{X,t} S_{X,t} & 0 \\ 0 & 0 \end{bmatrix}.$$

Therefore, we could continue above unconstrained problem as:

$$\begin{aligned} M^* &= \underset{M}{\operatorname{argmin}} \|S_W V_W^T V_X S_X - S_W V_W^T M V_X S_X\|_F^2 \\ &= \underset{M}{\operatorname{argmin}} \left\| \begin{bmatrix} S_{W,r} V_{W,r}^T V_{X,t} S_{X,t} - S_{W,r} V_{W,r}^T M V_{X,t} S_{X,t} & 0 \\ 0 & 0 \end{bmatrix} \right\|_F^2 \\ &= \underset{M}{\operatorname{argmin}} \|S_{W,r} V_{W,r}^T V_{X,t} S_{X,t} - S_{W,r} V_{W,r}^T M V_{X,t} S_{X,t}\|_F^2 \\ &= \underset{M}{\operatorname{argmin}} \|Z - S_{W,r} V_{W,r}^T M V_{X,t} S_{X,t}\|_F^2. \end{aligned}$$

The above minimization problem obtains the optimal value if $S_{W,r} V_{W,r}^T M V_{X,t} S_{X,t}$ equals the rank- k truncated SVD of Z by the fundamental property of SVD decomposition. Thus, we will have:

$$Z_k = S_{W,r} V_{W,r}^T M^* V_{X,t} S_{X,t} \implies M^* = V_{W,r} S_{W,r}^{-1} Z_k S_{X,t}^{-1} V_{X,t}^T.$$

□

B AN ALGORITHM TO SEARCH OF RANKS UNDER DRONE

In Algorithm 2, we illustrate how to select the rank of each module by applying DRONE illustrated in Algorithm 1. The input to Algorithm 2 consists of training data, the model with all parameters of weight matrices and original training loss. In addition, a pre-defined search grid is also necessary.

Taking $W \in R^{768 \times 768}$ as an example, we can perform a grid search for a proper low rank k over $[1, 768]$ such as $\{96, 192, 288, 384, \dots, 768\}$. The finer the grid, the more compressed model we could get at the cost of longer running time of the DRONE method. With these input parameters, we firstly distribute the total allowed loss into each individual module. We then iteratively apply Algorithm 1 following the computational sequence illustrated in Figure 1. For the compression of each module, we search the rank k by going through the grid. If the approximated result will not increase the allowed loss increase ratio of the component, we will end the search and tie the found rank to the component and move on. The procedure will continue until all components are compressed. The whole process could guarantee us that the final loss L' of the compressed model \hat{M} would not be greater than $(1 + r)L$, where L is the original loss before approximation.

Algorithm 2: Overall Algorithm of Grid Search of Low-rank Model Approximation

Input: training data D_{train} ; Original weight matrix W ; Prediction Model M , total allowed loss increase ratio r , Search grids of ranks for each module G , original Training loss L ,

Output: Low-rank Model \hat{M}

```

1  $R \leftarrow$  Distribute allowed ratio  $r$  into each module.
2 for  $l = 1, \dots, \text{total layers}$  do
3   foreach module  $m_i \in M_l$  do
4      $W_{l,i} \leftarrow$   $l$ -th layer parameter of module  $m_i$  (e.g., 2nd feed-forward matrix in first layer.)
5     for  $i = 1, \dots, |G_{l,i}|$  do
6        $k \leftarrow G_{l,i}$ 
7        $U, V \leftarrow$  Algorithm 1 ( $k, D_{train}, W_{l,i}, M$ )
8        $\hat{M} \leftarrow M$  with  $W_{l,i}$  replaced by  $U, V$ .
9       Evaluate new loss  $L_{new} = \hat{M}(D_{train})$ 
10      if  $L_{new}/L < 1 + R_{l,i}$  then
11         $M \leftarrow \hat{M}$ 
12      break
```

C LSTM RESULT

Models	LSTM-1	LSTM-2	Softmax	Others	Total Time	Perplexity
PTB-Large	1.27ms	1.30ms	1.09ms	0.13ms	3.79ms	78.32
PTB-Large-SVD	-	-	-	-	-	81.09
PTB-Large-DRONE	-	-	-	-	-	80.87
PTB-Large-DRONE-Retrain	0.24ms	0.34ms	0.42ms	0.11ms	1.11ms(3.4x)	79.01

Table 4: The average inference time of each component in the model of 2-layer LSTM model. Both proposed methods and SVD use same ranks so the inference time is approximately the same. The unit is in millisecond and the number in parenthesis shows the ratio respective to the overall inference time.

D RESULTS OF SVD-BASED RETRAINING

Models	Accuracy (%)
BERT-MRPC	89.5
BERT-MRPC-DRONE	86.8
BERT-MRPC-SVD	63.8
BERT-MRPC-SVD-Retrain	85.8

Table 5: Illustration of SVD fine-tuning. Using the same rank as the proposed method, SVD accuracy will drop significantly. After fine-tuning on the SVD-based approximation, the accuracy could be recovered. But it's still less competitive than the proposed method.

E PYTHON PSEUDO CODE OF SOLVING EQUATION 2

```

1 import numpy as np
2
3 def OPTsolver(x, y, k):
4     '''
5     compute the best rank k projection M such that  $\|x \cdot y' - x \cdot M \cdot y'\|_F$ 
6     is minimized
7     x \in shape n x d
8     y \in shape m x d
9     '''
10    xSS = np.matmul(x.transpose(), x)
11    kSS = np.matmul(y.transpose(), y)
12    U1, S1, V1 = np.linalg.svd(xSS, False)
13    S1 = S1 ** 0.5
14    I1 = np.eye(S1.shape[0])
15    U2, S2, V2 = np.linalg.svd(kSS, False)
16    S2 = S2 ** 0.5
17    I2 = np.eye(S2.shape[0])
18    YK = np.dot(np.dot(I1 * S1, V1), np.dot(V2.transpose(), I2 * S2))
19    U, S, V = np.linalg.svd(YK, False)
20    L = np.dot(V1.transpose(), I1 * (1/S1))
21    R = np.dot(I2 * (1/S2), V2)
22    M = np.dot(U[:, :k] * S[:k], V[:, k:])
23    return L, R, U, S, V

```

Listing 1: The python function to solve the equation 2.

F PYTHON PSEUDO CODE OF RANK SEARCHING

```

1
2 import os

```



```

3 import numpy as np
4 import torch
5 import subprocess as sp
6
7 cuda_num = 7
8 n_heads = 12
9 total_layer = 12
10
11 prev_loss = .11159391902588509 # Initial Loss
12 the_model_name = 'bertSST2'
13
14 time_attn = 117.5 # Empirical Inference Time on Attention Module
15 time_0 = 34.27 # Empirical Inference Time on Attention FFL Module
16 time_1 = 133.11 # Empirical Inference Time on Feedforward 1 layer
17 time_2 = 128.84 # Empirical Inference Time on Feedforward 2 layer
18 minimal_time = min(time_attn,time_0,time_1,time_2)
19 multiplier = (time_attn+time_0+time_1+time_2)/(minimal_time)
20 tolerant = 2. # allowed loss increase ratio.  $\epsilon$  in Algorithm 2.
21
22 # Code to Distribute the  $\epsilon$  into individual Modules.
23 # The distribution depends on empirical inference time of each module and
  number of layers.
24 basic_tolerance = np.exp(np.log(tolerant)/multiplier)
25 tol_attn = np.exp(np.log(basic_tolerance**(time_attn/minimal_time))/
  n_layer)
26 tol_0 = np.exp(np.log(basic_tolerance**(time_0/minimal_time))/n_layer)
27 tol_1 = np.exp(np.log(basic_tolerance**(time_1/minimal_time))/n_layer)
28 tol_2 = np.exp(np.log(basic_tolerance**(time_2/minimal_time))/n_layer)
29
30
31 ##### Omitted Code ###
32 # This part of the code is to change some parameters of the underlying
  huggingface framework in order to extract the training distribution  $X$ 
  of each module from the model.
33 ### Omitted Code ###
34
35
36
37 for i in range(total_layer):
38     for each module in the layer: # This line is pseudo code for clarity
  reason.
39         # This part of the code extracts  $\epsilon_{l,i}$  (named the_tol here) in
  Algorithm 2.
40         if save_symbol == "E":
41             the_tol = tol_attn
42         elif save_symbol == "F0":
43             the_tol = tol_0
44         elif save_symbol == "F1":
45             the_tol = tol_1
46         else:
47             the_tol = tol_2
48         # Update the allowed increase of loss
49         prev_loss = prev_loss * the_tol
50
51         rank = 16 if save_symbol == "E" else 96 # initial search rank for
  Attention(16) and FFL layers(96)
52         tps = 64 if save_symbol == "E" else 768 # Maximal rank specified
  in the original models.
53         while rank <= tps:
54             ### Omitted Code ###
55             ## Write the tried rank into huggingface framework##
56             ### Omitted Code ###
57
58             # This line run the inference in the command line
59             os.system('CUDA_VISIBLE_DEVICES="'+str(cuda_num)+'" python
  run_glue.py --model_type bert --model_name_or_path /data/TinySeries/

```



```

60 SST2/OriginalSST2/ --task_name SST-2 --do_eval --data_dir /data/
61 glue_data/SST-2/ --output_dir /tmp/sst-2 --per_gpu_eval_batch_size
62 100 --per_gpu_train_batch_size 100 --max_seq_length 128 > /tmp/tmp0')
63
64     with open('/tmp/tmp0', 'r') as file:
65         data = file.readlines()
66         new_r = float(data[-1])
67         if new_r < prev_loss:
68             break
69         if save_symbol == "E": # Attention module, we increase search
70             rank 16 at a time.
71             rank += 16
72         else:
73             #rank += 96 # For FFL layer, we increase search rank 96 a
74             time.
75             if rank == 384:
76                 rank = 768
77                 break
78             else:
79                 rank += 96
80
81     ### Omitted Code ###
82     # This part of code update the model #

```

Listing 2: A mixed of real code and pseudo code to illustrate the search algorithm.