Accelerating Recommender System Training 15x with RAPIDS

Sara Rabhi[*]
sara.rabhi@telecom-sudparis.eu
Télécom SudParis
Evry, France

Wenbo Sun[*]
ws3109@rit.edu
Rochester Institute of Technology
Rochester, USA

Julio Perez
jperez@nvidia.com
NVidia
Seattle, USA

Mads R. B. Kristensen
mkristensen@nvidia.com
NVidia
Copenhagen, Denmark

Jiwei Liu
jiweil@nvidia.com
NVidia
Pittsburgh, USA

Even Oldridge[†]
eoldridge@nvidia.com
NVidia
Vancouver, Canada

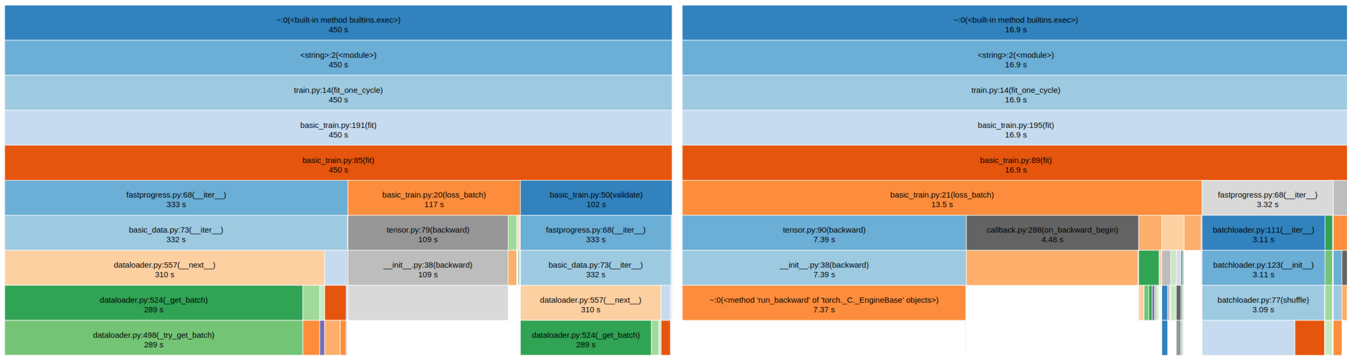

Figure 1: Unoptimized (450s) and optimized (15.4s) profiles of the execution of the model training loop following inclusion of the batch dataloader, embedding kernel improvements, and large batch size in conjunction with LAMB.

ABSTRACT

In this paper we present the novel aspects of our 15th place solution to the RecSys Challenge 2019 which are focused on the acceleration of feature generation and model training time. In our final solution we sped up training of our model by a factor of 15.6x, from a workflow of 891.8s (14m52s) to 57.2s, through a combination of the RAPIDS.AI cuDF library for preprocessing, a custom batch dataloader, LAMB and extreme batch sizes, and an update to the kernel responsible for calculating the embedding gradient in PyTorch. Using cuDF we also accelerated our feature generation by a factor of 9.7x by performing the computations on the GPU, reducing the time taken to generate the features used in our model from 51 minutes to 5. We demonstrate these optimizations on the fastai tabular model which we relied on extensively in our final ensemble. With training time so drastically reduced the iteration involved in generating new features and training new models is much more fluid, allowing for the rapid prototyping of deep learning based recommender systems in hours as opposed to days.

[*]Work done while while the authors were on internship at NVidia
[†]Corresponding Author

RecSys Challenge '19, September 20, 2019, Copenhagen, Denmark
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7667-9/19/09…$15.00
https://doi.org/10.1145/3359555.3359564

CCS CONCEPTS

• **Information systems → Content ranking**; **Recommender systems**; • **Computing methodologies → Neural networks**.

KEYWORDS

Recommender Systems, Neural Networks, GPU Acceleration

ACM Reference Format:
Sara Rabhi, Wenbo Sun, Julio Perez, Mads R. B. Kristensen, Jiwei Liu, and Even Oldridge. 2019. Accelerating Recommender System Training 15x with RAPIDS. In *Proceedings of the ACM Recommender Systems Challenge 2019 Workshop (RecSys Challenge '19), September 20, 2019, Copenhagen, Denmark.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3359555.3359564

1 INTRODUCTION

The training of recommender systems is a computationally intensive one, requiring significant compute resources and lengthy runtimes to generate features, prepare data, and train the model. In this paper we propose several novel optimizations to the fastai[2] tabular model training loop applicable to all recommender systems that use PyTorch[9]. These optimizations are evaluated on the RecSys Challenge 2019[5] which consists of session data taken from the hotel recommendation website Trivago. Our 15th place solution ensembled[13] several fastai tabular models with a lesser number of xgboost[1] and gbm[4] models. The ensembling method and models used are not particularly novel and we have instead chosen to share the methods used to accelerate training and feature generation.

Feature engineering and the preparation of data for deep learning models is a significant part of the development of any recommender system. The typical operations used in feature generation such as deltas, conditional existence, and crossing all benefit greatly from the massive parallelism that the GPU enables. Similarly the methods used to prepare categorical and continuous variables for a deep neural net model like encoding, normalization, null value filling, etc also follow the same format. These two stages were accelerated on the GPU with RAPIDS.AI cuDF library[10], speeding up both feature engineering and data preprocessing by an order of magnitude.

Profiling the recommender system during training also revealed bottlenecks when training tabular data in PyTorch[9]. The PyTorch dataloader was designed to load images which require significant CPU resources for preprocessing and data augmentation, with each item in the batch loaded one at a time. For tabular data typical of recommendation problems an item is a single row in the dataframe and is a very small tensor; accessing these one by one adds significant overhead. The dataloader was restructured to return a batch at a time in a single read from memory. A shuffle mechanism was also implemented to allow for the randomization of batches at the start of training and the beginning of each epoch.

Examining the GPU kernels that are most active within the GPU during training also revealed one kernel which accounted for 70% of all GPU training time. This kernel is used in calculating the gradients for embedding layers which most recommendation algorithms today make use of to represent categorical variables. The kernel was rewritten to achieve better GPU utilization, reducing it to 2.3% of the total GPU workload.

Finally, batch size of the model was scaled close to the memory limits of the 32 Gig Tesla V100 GPU used in training, utilizing the LAMB optimizer [15] to help with convergence. With a batch size of 204800 the training of the model took 15.4s with identical loss and MRR. The unoptimized fastai workflow with a similar batch size was 29.2x slower at 450s demonstrating the effectiveness of the batch dataloader at large batch sizes.

Section 2 covers the improvements to feature generation and preprocessing using RAPIDS. Section 3 outlines the batch dataloader and demonstrates its improved throughput. Section 4 explains in detail the improvements made to the embedding kernel. Section 5 outlines our findings regarding the LAMB optimizer and large batch sizes. Finally Section 6 summarizes the improvements and provide our conclusions and future work to improve the training time of deep recommender systems. Source code is provided[1] for the feature engineering phase along with a workflow for the end to end training of one of the models used in our ensemble.

2 FEATURE ENGINEERING AND PREPROCESSING

Feature engineering and preprocessing operations take the form of simple manipulations to the data sources. Data scientists and ML Engineers typically create these features manually through the manipulation of data in Pandas[6] or some other dataframe library where features can be combined, compared and contrasted. These features, along with the input features, must then be prepared for the model by encoding categoricals for embedding access and normalizing continuous variables into ranges that the neural network can more easily work with. In this Section we demonstrate the effectiveness of doing these operations on the GPU using the cuDF library.

[1]https://github.com/rapidsai/dataloaders/tree/master/RecSys2019

Table 1: Feature Engineering Timing (s) and Speedup

	CPU	GPU	Speedup
read csv	57.6	5.2	11.0
string comparsion and masking	14.8	3.0	4.9
string column split & expand	70.0	0.6	114.0
assign string columns to dataframe	44.2	26.8	1.6
create data pair	117.3	89.9	1.3
prepare for target encoding	52.5	26.6	2.0
mean target encoding	1071.8	108.6	9.9
constrain string	13.8	2.6	5.3
count items	56.5	5.9	9.5
merge	128.1	12.4	10.3
binary features from text	266.0	20.8	12.8
difference features	1161.0	13.9	83.3
total	3053.5	316.4	9.7

2.1 Feature Engineering

The goal of feature engineering is to explicitly provide the model with information likely to be important in predicting the target that it would otherwise have a hard time computing on its own. These can be relationships that are not easy for a neural network to compute, like the delta between two input features, or they can be data that comes from an external source or event. An example of the latter from the competition is whether the item being predicted was the last item that the user interacted with. The process is generally an iterative one, requiring the exploration of different options and the evaluation of their performance improvements upon the model. One of the significant benefits of cuDF in this context is the speed with which features can be prototyped, allowing for rapid exploration of new features. Combined with the training speedup outlined in other sections of the paper this allows for the quick evaluation of a wide range of features.

In our case we generated features in four main groupings:

- User-item Impression pairs
- Item Metadata features
- Item Interaction features
- Relative Item Interaction features

First, user-item pairs were generated from the impressions to transform the problem into a binary classification problem. Additional information about the item, user-item interactions, and relative item interactions were computed independently and joined on to the original interaction pairs to generate the final output. Table 1 demonstrates the 9.7x speedup (relative to a pandas based workflow) of this stage of the workflow and the influence of individual features generated.

The total time to create the set of additional features and to generate the initial dataset was 51 minutes using Pandas. By performing these operations using cuDF where they can be accelerated by the GPU we improved the total feature creation time to just over 5 minutes, with each grouping taking just over a minute. This 9.7x speedup allowed us to quickly explore a wider range of features than would have otherwise been possible in such a short time frame.

2.2 Preprocessing

The fastai tabular model contains several preprocessing steps which modify the data, getting it into a format that is model ready. Categorical variables are encoded into their numerical representations while continuous variables null values are mapped to the median with a binary is_null variable created to indicate rows where the value has been replaced.

The original fastai implementation of these steps utilizes Pandas before eventually converting to Numpy[14], stacking the categorical and continuous variables separately and converting to a longtensor and a floattensor respectively. The cuDF implementation takes advantage of the byte transfer capabilities of its parquet reader and avoids any unnecessary conversions, copying the parquet files created during feature generation directly into GPU memory. After preprocessing has taken place the cuDF dataframes are memory mapped directly into tensors using the dlpack zero copy transfer of data. These tensors are optionally transferred to the CPU to be used in the PyTorch dataset. This transfer is unnecessary if your GPU has enough memory to hold both the dataset and the model.

The unoptimized workflow took 401.7 seconds to preprocess the feature engineered data and get it ready for model training. The optimized cuDF workflow shortens this to 47.41 seconds when using CPU memory to store the tensors, or 41.8 seconds if the tensor can stay on the GPU. This speedup of 9.6x, similar to what was achieved in feature generation, is typical of the benefit that GPU acceleration can provide to dataframe operations.

3 BATCH DATALOADERS

Dataloaders are the mechanism by which data is passed in batches to the model. In the pytorch dataloader this takes the form of an iterator that randomly selects indices from the dataset, grabs the data, collates the results into a batch, and then passes that batch to the GPU. This process of grabbing from the dataset item by item is relatively inexpensive but does add overhead, and in order to keep the GPU utilized it is necessary to assign multiple workers to create batches in parallel. RAPIDS is built upon CUDA which cannot be forked once launched due to resource ownership issues, preventing us from using the multiworker dataloader without using the much slower spawn method of generating workers. Fortunately there is another method that does not require multiple workers.

The core functionality of the dataloader is an iterator over the dataset that returns batches. This functionality has been implemented in the form of a batch dataloader that loads the entirety of the batch from CPU memory into a tensor in a single memory access. BatchDataloader and BatchDataset[2]. replace the PyTorch Dataloader and Dataset classes, but are otherwise called in the same way. In order to maintain the randomization required when training DNN models effectively they also provide a mechanism to shuffle the data before training and at the beginning of each epoch.

Loading data in this way has significant benefits. First, no multiprocessing is required, which reduces errors and significantly speeds up workflows in environments like Windows where multiprocessing is slower. Second, performance of the batch dataloader is better than that of the multi-worker dataloader on tabular datasets. Third, it allows the use of CUDA before the dataloader, which is required by RAPIDS libraries.

Figure 1 shows the before and after profiles of the training loop for an extreme batch size of 204,800. Initial testing was done at a more modest batch size of 4096, however the profiles are similar in nature. As in Figure 1, in the batch size 4096 unoptimized profile the dataloader is a largest component of the profile, taking 260s or 54.6% of the total training runtime. The batch dataloader improves this to 2.45s, only 2.08% of the total training time. This is a relative speedup of over 100x for data loading and a speed up of total training by more than 2x. This improvement of two orders of magnitude partly comes from the usage of the PyTorch dataloader in fastai, which does tensor conversions during __get_item__ rather than during the initialization phase in order to flexibly allow for data augmentation at the time of batch load. More typically we see a 5-50% improvement over the multi-worker dataloader for similar sized batches across different workloads, depending on the batch and tensor size.

[2]https://github.com/rapidsai/dataloaders/tree/master/pytorch/batch_dataloader

4 KERNEL IMPROVEMENTS

Using the batch dataloader training is no longer dominated by dataloading and batch size can be scaled to better utilize the GPU. Analyzing the model using the NVProf tool[8] in Figure 2 we see that 69.6% of the work on the GPU is happening within a single kernel EmbeddingBag_accGradParametersKernel_sum_avg (EmbGPK).

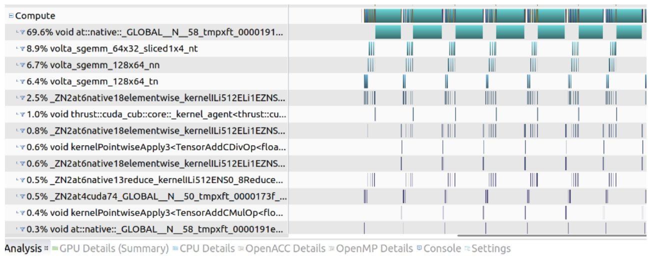
Compute
69.6% void at::native::_GLOBAL__N__58_tmpxft_0000191...
8.9% volta_sgemm_64x32_sliced1x4_nt
6.7% volta_sgemm_128x64_nn
6.4% volta_sgemm_128x64_tn
2.5% _ZN2at6native18elementwise_kernelILi512ELi1EZNS...
1.0% void thrust::cuda_cub::core::_kernel_agent<thrust::cu...
0.8% _ZN2at6native18elementwise_kernelILi512ELi1EZNS...
0.6% void kernelPointwiseApply3<TensorAddCDivOp<floa...
0.6% _ZN2at6native18elementwise_kernelILi512ELi1EZNS...
0.5% _ZN2at6native13reduce_kernelILi512ENS0_8Reduce...
0.5% _ZN2at4cuda74_GLOBAL__N__50_tmpxft_0000173f_...
0.4% void kernelPointwiseApply3<TensorAddCMulOp<flo...
0.3% void at::native::_GLOBAL__N__58_tmpxft_0000191e...
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
OpenMP Details
Console
Settings

Figure 2: Nvprof profile of the fastai model on GPU.

The calculation of EmbGPK is a segmented sum over the rows of a matrix to compute the embedding gradients. In the original implementation index access was not distributed evenly and there were not enough threads, resulting in the underutilization of the processors on the GPU shown in Figure 3 (top). Instead of doing a flat sum where a single variable is used to accumulate the weights, the kernel was modified to compute the sum of the weights in two-steps:

- Each GPU warp sums 'NROWS_PER_THREAD' number of row given by 'indices'
- Sum each partial-sum from 1) and scatter into 'grad_weight'

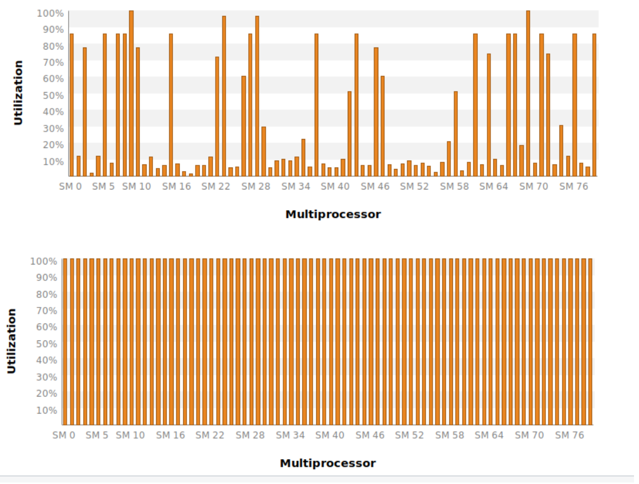

Figure 3: GPU Streaming Multiprocessor utilization before (top) and after (bottom) improvements to EmbGPK

Not only does this increase the utilization, it also improves numerical stability. Figure 3 (bottom) shows the utilization of the GPU under the new kernel where we see that all GPU Streaming Multiprocessors are utilized fully. The optimizations have been merged into PyTorch and any method using embeddings at scale should see an improvement in performance. When we examine the model performance again in NVProf we see a significantly improved graph. The calculation of embedding gradients has gone from 70% of the workload to 2.3% and the GPU computation is dominated by matrix multiply kernels which are already heavily optimized.

5 LAMB AND EXTREME BATCH SIZES

One of the further advantages of the batch dataloader introduced in Section 3 is the ability to scale to extreme batch sizes. On the Tesla V100 GPUs used for evaluation the maximum batch size of the tabular model to was 819600. Scaling to this degree sped up training to 15s/epoch, however the model took two epochs to converge. When batch size was reduced to 204800 the model converged in a single epoch in 15.4 seconds. This is a 7.5x speedup over the 4096 batch size GPU Memory variant. Comparatively the relative speedup for the unoptimized fastai workflow with the larger batch size was only 4.7%, in part due to increased dataloader costs.

The scaling of batch size to this degree required the use of layer-wise adaptive large batch optimization [15], more commonly known as the LAMB optimizer, which scales the learning rate for each layer of the network. We validated model performance by evaluating mean and stdev of AUC and MRR across five runs of each batch size. The 4096 batch size version had an AUC of 0.8827 ± 0.0049 and an MRR of 0.6143 ± 0.005 while the 204800 batch version had an AUC of 0.8816 ± 0.0012 and an MRR of 0.6136 ± 0.0008 for our baseline model, a difference of 0.1% for both metrics.

6 CONCLUSIONS AND FUTURE WORK

This paper demonstrates several novel optimizations that can be used when training deep learning recommender systems in PyTorch or when using the fastai library. Source code for all opimizations is provided and researchers are encouraged to try these methods out on their own recommendation workflows.

Table 2: Training Time (s)

Batch Size 4096	Prep	Dataload	Training	Total
UnOptimized	401.7	268.0	222.0	891.7
CPU Memory	47.4	7.4	119.2	174.0
GPU Memory	41.8	2.6	115.4	159.8
Batch Size 20480	**Prep**	**Dataload**	**Training**	**Total**
UnOptimized	401.7	310.0	140.0	851.7
GPU Memory	41.8	3.1	12.3	57.2

Using the RAPIDS.AI cuDF library the preprocessing steps that prepare data for the model were improved by a factor of 9.6x, performing categorical encoding, normalization and null value filling on the GPU. A novel batch dataloader which loads an entire batch from memory in a single read accelerating the PyTorch dataloader by a factor of over 100x, and improving training time by 2x. The kernel responsible for calculating embedding gradients in PyTorch was optimized reducing it from 70% of the workload to 2.3% and improving the training time by 1.9x. Then, taking advantage of the LAMB optimizer batch size was scaled to the limits of GPU memory, achieving a further 2.12x speedup in training time. A final 15% improvement in performance was achieved by keeping the entire dataset in GPU memory during training.

These optimizations collectively improve the end to end training time by a factor of 15.6x. Table 2 highlights the improvements to each stage of the recommendation pipeline. Reducing the training time from 15 minutes to 57.2s enables a much richer exploration of feature space in hours instead of days, and significantly reduces the cost of training deep learning based recommenders. The iterative nature of feature engineering and model training mean that these speedups compound; long feature creation and training times result in a lot of cognitive downtime and a disrupted work process. Shortening feature creation time and training time so dramatically provides a more immediate feedback loop where Data Scientists and ML Engineers can stay in the flow of their work.

In addition to these improvements we plan to explore FP16 mixed-precision training using Apex [7] and multi-gpu training via Hogwild [11], Horovod[12] and BytePS[3], improving model performance at the cost of additional hardware. Further, these performance improvements are not limited to tabular data or recommendation and should be applicable to Natural Language Processing, Graph Data, Time Series, Tabular Data, and other problems where dataloading and preprocessing plays a significant role in the total time taken to train a model. We hope to demonstrate their effectiveness in other domains and provide easy to use examples for the community.

ACKNOWLEDGMENTS

The authors wish to thank the RAPIDS.AI team for their support on this competition and in particular Joshua Patterson for his vision of a GPU accelerated data science workflow and Nicolas Koumchatzky for his guidance and recommender system expertise.

REFERENCES

[1] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785
[2] Jeremy Howard, Rachel Thomas, and Sylvain Gugger. 2018. Fast.ai Library. http://docs.fast.ai
[3] Bytedance Inc. 2019. BytePS. https://github.com/bytedance/byteps
[4] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 3146–3154. http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf
[5] Peter Knees, Yashar Deldjoo, Farshad Bakhshandegan Moghaddam, Jens Adamczak, Gerard-Paul Leyson, and Philipp Monreal. 2019. RecSys Challenge 2019: Session-based Hotel Recommendations. In *Proceedings of the Thirteenth ACM Conference on Recommender Systems (RecSys '19)*. ACM, New York, NY, USA, 2. https://doi.org/10.1145/3298689.3346974
[6] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.
[7] NVidia. 2017. APEX. https://github.com/NVIDIA/apex
[8] NVidia. 2019. CUDA Profiler Users Guide. https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf
[9] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*.
[10] RAPIDS.AI. 2019. RAPIDS.AI cuDF repository. https://github.com/rapidsai/cuDF
[11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 693–701. http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf
[12] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799
[13] Mark J van der Laan, Eric C Polley, and Alan E Hubbard. 2007. Super Learner. In *Journal of the American Statistical Applications in Genetics and Molecular Biology*, Vol. 6. Issue 1.
[14] StÃľfan van der Walt, S. Chris Colbert, and GaÃńl Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30. https://doi.org/10.1109/MCSE.2011.37 arXiv:https://aip.scitation.org/doi/pdf/10.1109/MCSE.2011.37
[15] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. 2019. Reducing BERT Pre-Training Time from 3 Days to 76 Minutes. *CoRR* abs/1904.00962 (2019). arXiv:1904.00962 http://arxiv.org/abs/1904.00962