Dec 9, 2019

Performance of Bert encodings in Nvidia DeepLearningExamples

This document is a snapshot of the performance of the DeepLearningExamples (https://github.com/NVIDIA/DeepLearningExamples) Tensorflow and Pytorch encodings of the BERT (https://arxiv.org/abs/1810.04805) language representation model. As the code bases and underlying DNN libraries are regularly improved this only represents the performance for a clone of the source base taken at the end of September 2019.

The HW configuration for all measurements in this document is a single Nvidia DGX2 (not DGX2H) with 16 32GB V100s. running linux.

The sw configuration is using the Nvidia docker images (tenorflow and pytorch) built with the scripts and Dockerfiles supplied in the github repository. Thus using nvcr.io/nvidia/tensorrtserver:19.06-py3 for Tensorflow and nvcr.io/nvidia/pytorch:19.08-py3 for pytorch.

Tensorflow Baseline

The Bert baseline has been a transformer based (large) model with 24 layers, a hidden size of 1024, 16 attention heads, a feed forward filter size of 4096 and pre trained on Wikipedia + bookcorpus data blocked into sequences of 512 tokens and using the Adam optimizer. FP16-mixed mode numerical representation has been available for some time and is the only mode that will be considered here. The current code base also supports using a dataset blocked into sequences of 128 tokens. Thus, the performance impact of the sequence length can be evaluated. The current code base also supports several optimization techniques for the mixed mode calculations. The ones considered here for the Tensorflow code base will be using XLA and a manual FP16 optimization that appears to achieve faster computations and allows a larger minibatch size/gpu.

The performance data for pretraining is shown below. The maximum batch size that fp16 plus xla could support was 10, while the manual fp16 reached 14 for the sequence length of 512. Similarly, a larger batch size for sequence length 128 was also reached. The speeds were evaluated by averaging the last 15 values printed to stdout before stopping the jobs. Thus, they do not include overheads of opening new files, initial warmup, compilation, etc.

Tensorflow-Adam seq_len=512				
batch size	fp_mode	num_gpu	global seq/sec	seq/sec/gpu
10	fp16, xla	4	82.745	20.68625
10	manual_fp16	4	115.09	28.7725
14	manual_fp16	4	127.57	31.8925
14	manual_fp16	8	251.91	31.48875
14	manual_fp16	16	494.86	30.92875

The sequence length 128 data are shown below for the baseline runs. Only the manual-fp16 optimization was used in light of the results from sequence length 512.

Tensorflow-Adam seq_len=128				
batch size	fp_mode	num_gpu	global seq/sec	seq/sec/gpu
64	manual_fp16	4	609.26	152.315
64	manual_fp16	8	1208.03	151.00375
64	manual_fp16	16	2349.55	146.846875

LAMB Optimizer

The Lamb optimizer (https://arxiv.org/abs/1904.00962) has been added to the DeepLearningExamples code base in the current release. It enables a transformer based model to greatly increase the global batch size. The code base invoking the Lamb optimizer is numerically stable with global batch sizes up to 32768 for sequence lengths of 512 tokens and up to 65536 for sequence lengths of 128 tokens. These large global batch sizes mean that the weight updates and inter GPU communications are greatly reduced. The large global batch size also greatly increases the number of GPUs that can be used in parallel. However even with a single DGX2 there is a significant performance increase. In all the tables shown below the global batch size is labelled “global agg size”.

In the course of experimenting with the Lamb optimizer based Tensorflow code, the author had not appreciated the global batch size limit and how it was implemented through the invocation scripts. This caused some confusion as can be seen in the table of runs that were attempted, shown below. The runs that did not crash with NaNs were terminated with kill -9 as these were just short speed runs.

The moral is read the documentation carefully or pay the price 😊

Tensorflow-Lamb-seq_len=128							
batch size	fp_mode	num_gpu	global seq/sec	seq/sec/gpu	agg size	last	global agg size
48	fp16-xla	4	660.7	165.175	128	260	24576
48	fp16-xla	8	1281.3	160.1625	128	340	49152
48	fp16-xla	16	2519.1	157.44375	128	310-NaN	98304
48	manual_fp16	4	686.1	171.525	128	900	24576
48	manual_fp16	8	1333.5	166.6875	128	470-NaN	49152
64	manual_fp16	4	733.2	183.3	128	230-NaN	32768
64	manual_fp16	8	1418.6	177.325	128	190-NaN	65536

What also became apparent is that the manual_fp16 optimization was not stable with the large batch sizes and the Lamb optimizer for the sequence length of 128.

Pytorch Version

It had already been observed by the author that versions of BERT written in the Pytorch framework could run faster than those in Tensorflow (see Megatron-LM article https://github.com/David-Levinthal/machine-learning/blob/master/bert_vs_GPT2.pdf). This is born out with the DeepLearningExamples code base as well. The runs at sequence length 128 ran at ~ 200 seq/sec/gpu with the pytorch based code even on 16 GPUs or the lower batch size of 48. The Tensorflow runs perhaps reached 170 with 4 GPUs, but likely had stability issues with more GPUs. Probably a value of ~ 160 is more representative.

Again, it is clear in the data below that going beyond the recommended global batch size is not a good idea.

Pytorch-lamb seq_len=128							
batch size	num_gpu	it/s	global seq/sec	seq/sec/gpu	agg_size	last	global agg size
64	4	3.19	816.64	204.16	128	200 full	32768
64	8	3.18	1628.16	203.52	128	200 full	65536
64	16	3.15	3225.6	201.6	128	236-Nan	131072
48	16	4.16	3194.88	199.68	128	590-NaN	98304
64	16	3.14	3215.36	200.96	64	7038 full	65536

The first two short runs were terminated by setting the number of steps to a rather small number to avoid killing the applications. The last run in the table above was taken from a full run to convergence. This will be discussed in more detail later.

The speeds were evaluated by averaging the last 15 final values of iterations/sec (it/s) printed by the tqdm invocations. The use of tqdm in a code like this strikes the author as a poor idea as it greatly increases the size of the log file, adds overhead and makes post processing rather difficult.

Getting just the final value (shown below, 3.18 iterations/sec) is a bit of a trick. As the iteration/sec value increases over the duration of the step it is critical to get the final value. It would be easier if a simple call to time were used as is done in the Tensorflow versions.

```
Iteration:  33%|\u2588\u2588\u2588\u258e      | 3655/11057 [19:17<38:45,  3.18it/s]
```

Averaging these speeds ignores some overhead due to opening files and some warmup. As a completed run also prints out the total time this overhead can be evaluated by comparing a calculated training time to the total measured time of 148313.2 seconds, a difference of a bit more than 3%

it/s	agg_size	total_steps	calc total time
3.14	64	7038	143449.7

The Lamb run is done in two phases, with the first being run with the 128 token sequence length data, followed by a second phase using the 512 token length data. Again the 512 token length training cycle is faster with the pytorch code than the Tensorflow version shown earlier.

Pytorch-lamb seq_len=512						
batch size	num_gpu	it/s	global seq/sec	seq/sec/gpu	agg_size	global agg size
10	4	4.04	161.6	40.4	128	5120
10	8	4.04	323.2	40.4	128	10240
10	16	4.03	644.8	40.3	128	20480
8	4	4.77	152.64	38.16	512	16384
8	8	4.78	305.92	38.24	512	32768
8	16	4.75	608	38	512	65536

The last run was terminated after 35 steps and therefore never ran into NaNs.

The default phase2 run uses a local batch size of 8 and 1536 steps of a global batch size of 32768. This is compared with the phase1 run using the 128 token length data where the batch size was 64 and there were 7038 steps of a global batch size of 65536.

The author ran into dimm induced machine check errors during the second phase and had to restart from checkpoints several times. The speed from the final restart is shown below.

batch size	num_gpu	it/s	global seq/sec	seq/sec/gpu	agg_size	global agg size
8	16	4.72	604.16	37.76	256	32768

As the first several runs did not finish normally the total measured runtime was not available for the second phase. If one assumes the same overhead for phase2 as was observed in phase 1 the total time for phase 2 can be computed by using the training time (num_steps*aggregation_size/average iterations per sec)

it/s	agg_size	total_steps	calc total time	corrected_p2_time
4.72	256	1536	83308.47	86132.97

The total number of samples for the two phases are shown below

	batch_size	num_gpu	agg_size	total_steps	total_seq
Phase1	64	16	64	7038	461242368
Phase2	8	16	256	1536	50331648

While approximately 90% of the samples are from the 128 token length run, the total times of 148313.2 seconds or 41.2 hours and 86133 seconds or 23.9 hours are more in the ratio of 2 to 1, with a total time on a single DGX2 of around 65 hours.

It is worth noting that the original distribution using only one data set and the Adam optimizer required 2.25 million steps on 16 GPUs with a batch size of 14, or 504 million sequences of length 512 tokens. The use of 2 data sets reduced the number of sequences run on the 512 token data set to 1536*32768 or 50.3 million, ie 10X less. If one scaled up that number by a factor of 3 for the fraction of total time on the 512 token sequence length data set, the effective reduction in the number of sequences still a factor of over 3X.

Model Quality

As the code keeps the last 3 checkpoint files from phase1 and the checkpoint files from phase2 on can compare the model quality at the end of phase1 to the final model quality. To do this the author used the squad application in the pytorch code base (scripts/run_squad.sh, etc) using the chkpt.7038 (final phase1 chkpt file) and then rerunning using the chkpt.8574 file (final phase2 chkpt file).

Squad uses a sequence length of 384, but of course the number of parameters in the model is independent of this. The final converged model is truly converged as running Squad V1.1 on the final model checkpoint, with 2 epochs of fine tuning results in an accuracy of 84.5.

{“exact_match”: 84.47492904446547, “f1”: 90.96846445339581}

The squad run using the checkpoint from the end of phase 1 yields an accuracy is quite a bit lower, 73.5.

{“exact_match”: 73.50993377483444, “f1”: 81.8947410846232}

It is quite clear that the 2 phase approach greatly speeds up training to a converged model.

Bert Base: A Convergence study

The two-phase training of language models is rather standard now, using two data sets made from the same text data but divided into different sequence lengths. The short sequence length data set used to get a rough model quickly and then a second full sequence length data set to get the best model. This creates a 2-dimensional problem when trying to study convergence speed and finding an optimal strategy.

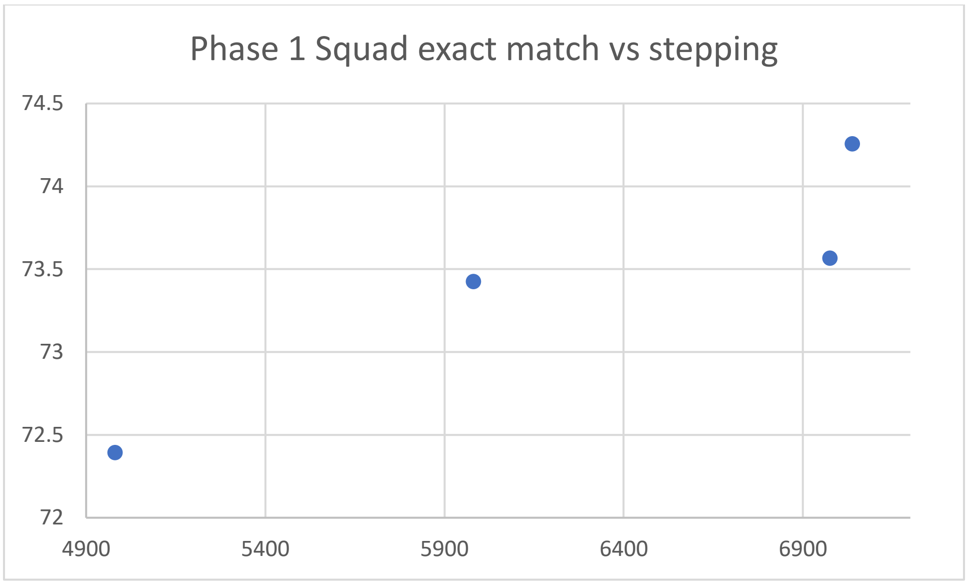
Due to the 2 dimensional nature of an exploration of a two phase training, BERT base (hidden size 768, 12 layers, 12 attention heads, 110 Million parameters) was used for an example analysis. It should be noted that smaller models show less differentiation in performance versus the selected hardware, so any conclusions based on the data herein should keep that in mind. (https://github.com/David-Levinthal/machine-learning/blob/master/Performance%20properties%20of%20some%20deep%20neural%20networks.pdf, https://github.com/David-Levinthal/machine-learning/blob/master/Evaluating%20RNN%20performance%20across%20HW%20platforms.pdf)

The basic DeepLearningExamples/Pytorch/LanguageModel/Bert/scripts formed the basis for the study, merely changing the model to Bert base, the stride of checkpointing and splitting the run_training.sh into a phase1 and phase 2 script. Then run_squad.sh was used to evaluate the model quality for the 2-dimensional set of checkpoints. Checkpoints from phase 1 (128 token sequence length pretraining) were written every 1000 steps of the global batch size of 65536. A series of phase2 pretraininings were then seeded from the last 4 phase 1 checkpoints. For the phase 2 runs the local batch size was increased from 8 to 16 as this improved throughput by 10%. The global batch size was held constant at 32768. The model quality was then determined with a standard run of the run_squad.sh script modified to refer to the Bert_base configuration. This uses 2 epochs of fine tuning on the Squad V1.1 data set, more will be said of this later. The data of the exact match values is shown below. The times are calculated from the number of steps to reach the checkpoint.

The pretraining was done using the 16GPUs in a DGX2 as were the earlier measurements in the paper. The Squad fine tuning runs were done on a Dell C4140 with 4 V100s

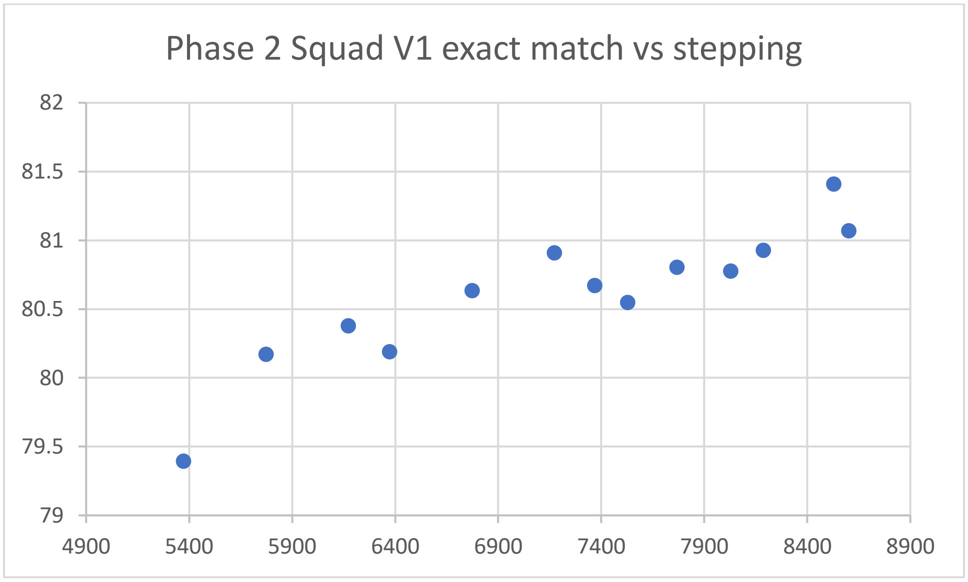
stepping	last seq len	total duration (hrs)	exact match	f1
4981	128	11.33	72.39	81.29
5373	512	13.50	79.39	86.89
5773	512	15.72	80.17	87.50
6173	512	17.94	80.38	87.75
5981	128	13.60	73.42	82.09
6373	512	15.78	80.19	87.44
6773	512	18.00	80.63	87.94
7173	512	20.22	80.91	88.01
6976	128	15.87	73.57	82.31
7368	512	18.04	80.67	87.73
7768	512	20.26	80.80	87.96
8186	512	22.58	80.93	88.08
7038	128	16.02	74.26	82.66
7528	512	18.74	80.55	87.67
8028	512	21.52	80.78	88.09
8528	512	24.29	81.41	88.47
8601	512	24.70	81.07	88.45

A few graphs make it a bit easier to absorb the data.
first a graph of the exact match values at the phase 1 checkpoints.

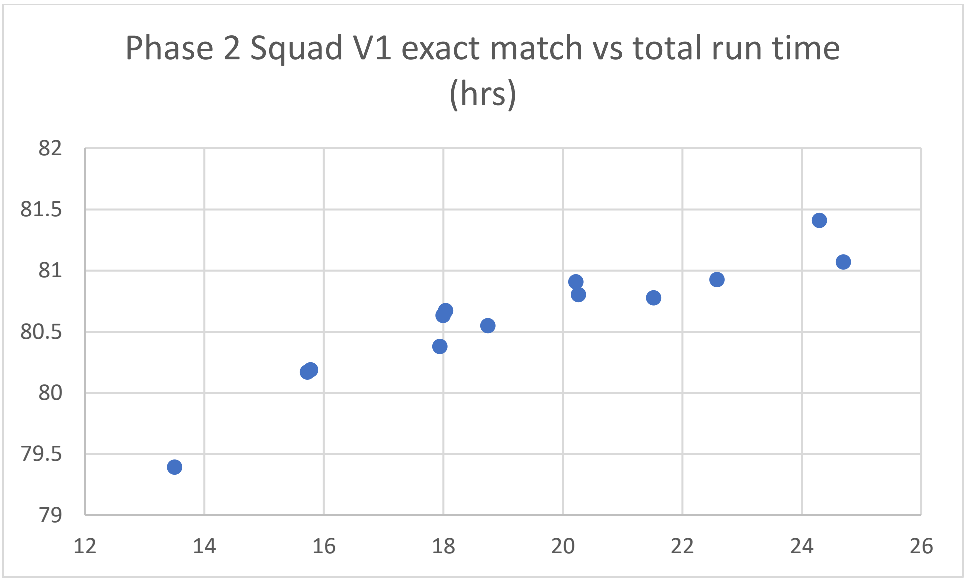


The last point shows a jump of almost 1 percent. A different run resulted in a value of 73.07. This illustrates the variations that show up in the convergence and in the way its accuracy is determined.

The data from the phase 2 checkpoints is shown below with a steady rise as more total steppings in phase1 and phase 2 are used in pre-training

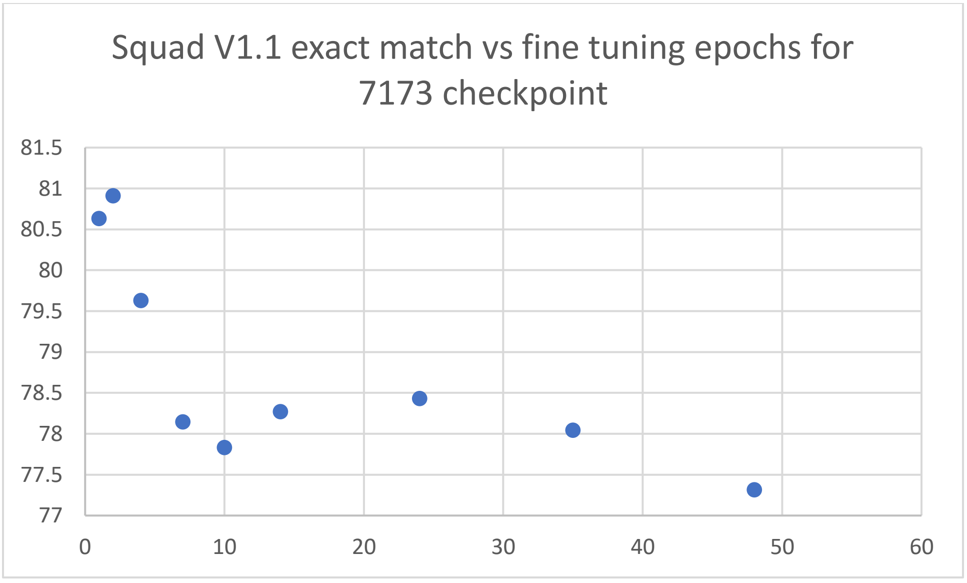


The data shown this way is still a bit confusing as the point just above 7400 is actually from the 4th phase 1 checkpoint. A better way to illustrate the phase 2 convergence is to plot the exact match values vs the total pretraining time as shown below:



And a much more monotonic result is shown with points with nearly equal total pre training times but with different phase 1 seeds having almost identical exact match scores.

Finally a series of squad runs were made varying the number of fine tuning epochs to explore the effect of that third phase of training. It is somewhat surprising that more than just a couple epochs results in a decrease in the exact match score. Perhaps the V1.1 data set is not so good or there is some other reason that was not apparent. A suggestion from my colleague, Andy Wagner, is that the Squad data set overtrains very quickly which would explain the observed behavior.



Added comment on TQDM

The following is from the end of a (different) full run of phase1 (sequence length 128). The accumulation size was 64 and the local batch size was 64
The final throughput summary of the run was
training throughput phase1: 2723.84 sequences/second
we can use this, the accumulation size and number of nodes to compute an average rate of iterations/sec that the base node would print out. This yields:
2724/(16*64) = 2.66 it/sec

We can process the log to show tqdms' last value (of 64 printed per step)

```
cat bert_16gpu_phase1_pyt.log | tail -n 25
Iteration:   6%|\u258c        | 661/11337 [03:37<57:35,  3.09it/s]Step 7032 LR 0.00017518714874213283
Step:7032 Average Loss = 1.4574737548828125 Step Loss = 1.6171875 LR 0.006
Iteration:   6%|\u258b        | 725/11337 [03:57<57:13,  3.09it/s]Step 7033 LR 0.00015992325525180656
Step:7033 Average Loss = 1.443267822265625 Step Loss = 1.4541015625 LR 0.006
Iteration:   7%|\u258b        | 789/11337 [04:18<56:53,  3.09it/s]Step 7034 LR 0.0001430397079704331
Step:7034 Average Loss = 1.4429473876953125 Step Loss = 1.462890625 LR 0.006
Iteration:   8%|\u258a        | 853/11337 [04:39<56:36,  3.09it/s]Step 7035 LR 0.0001238760208522985
Step:7035 Average Loss = 1.449462890625 Step Loss = 1.4208984375 LR 0.006
Iteration:   8%|\u258a        | 917/11337 [05:00<56:11,  3.09it/s]Step 7036 LR 0.00010114434748482682
Step:7036 Average Loss = 1.446441650390625 Step Loss = 1.435546875 LR 0.006
Iteration:   9%|\u258a        | 981/11337 [05:21<55:52,  3.09it/s]Step 7037 LR 7.151985398522354e-05
Step:7037 Average Loss = 1.46856689453125 Step Loss = 1.4853515625 LR 0.006
Iteration:   9%|\u2589        | 1045/11337 [05:42<55:32,  3.09it/s]Step 7038 LR 0.0
11/13/2019 19:47:17 - INFO - __main__ -   Total Steps:7048.0 Final Loss = 1.4431438446044922
11/13/2019 19:47:17 - INFO - __main__ -   ** ** * Saving fine - tuned model ** ** *
Total time taken 149858.59364962578

*****************************************
Setting OMP_NUM_THREADS environment variable for each process to be 1 in default, to avoid your system being overloaded, please further tune the variable for optimal performance in your application as needed.
*****************************************
+ set +x
finished pretraining, starting benchmarking
```

training throughput phase1: 2723.84 sequences/second

```
average loss: 1.46856689453125
final loss: 1.4431438446044922
```

On the other hand, we can extract tqdms' first value and one might ask
which one is more likely to yield 2724 seq/sec? 😊

```
$ cat bert_16gpu_phase1_pyt.log | tail -n 25 | cut -f1 -d]
Iteration:   5%|\u258c        | 598/11337 [03:16<1:03:39,  2.81it/s
Step:7032 Average Loss = 1.4574737548828125 Step Loss = 1.6171875 LR 0.006
Iteration:   6%|\u258c        | 662/11337 [03:37<1:05:44,  2.71it/s
Step:7033 Average Loss = 1.443267822265625 Step Loss = 1.4541015625 LR 0.006
```

```
Iteration:   6%|\u258b        | 726/11337 [03:58<1:06:21,  2.66it/s
Step:7034 Average Loss = 1.4429473876953125 Step Loss = 1.462890625 LR 0.006
Iteration:   7%|\u258b        | 790/11337 [04:19<1:04:03,  2.74it/s
Step:7035 Average Loss = 1.449462890625 Step Loss = 1.4208984375 LR 0.006
Iteration:   8%|\u258a        | 854/11337 [04:40<1:01:47,  2.83it/s
Step:7036 Average Loss = 1.446441650390625 Step Loss = 1.435546875 LR 0.006
Iteration:   8%|\u258a        | 918/11337 [05:01<1:04:16,  2.70it/s
Step:7037 Average Loss = 1.46856689453125 Step Loss = 1.4853515625 LR 0.006
Iteration:   9%|\u258a        | 982/11337 [05:21<1:04:48,  2.66it/s
11/13/2019 19:47:17 - INFO - __main__ -   Total Steps:7048.0 Final Loss = 1.4431438446044922
11/13/2019 19:47:17 - INFO - __main__ -   ** ** * Saving fine - tuned model ** ** *
Total time taken 149858.59364962578

*****************************************
Setting OMP_NUM_THREADS environment variable for each process to be 1 in default, to avoid your system being overloaded, please further tune the variable for optimal performance in your application as needed.
*****************************************
+ set +x
finished pretraining, starting benchmarking
training throughput phase1: 2723.84 sequences/second
average loss: 1.46856689453125
final loss: 1.4431438446044922
```

Conclusion: using tqdm in production code might not be the best choice.