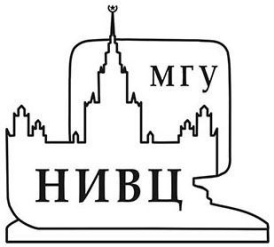
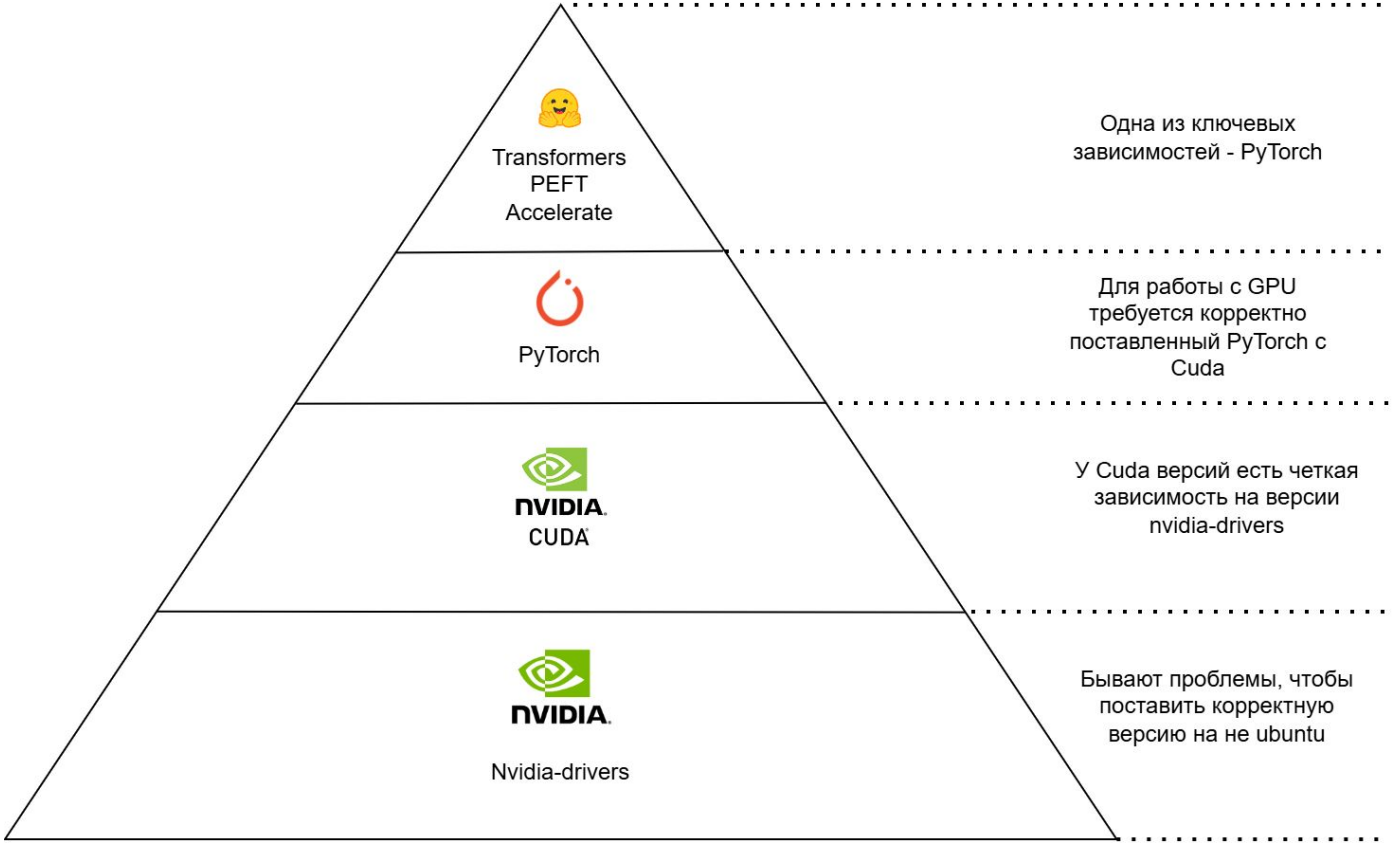


Технические основы

к.ф.-м.н. Тихомиров М.М.

НИВЦ МГУ имени М. В. Ломоносова

“Пирамида” технологий LLM



Pytorch

```
[1] import torch
    torch.cuda.is_available()
```

```
True
```

PyTorch Build	**Stable (2.5.1)**			Preview (Nightly)	
Your OS	Linux		Mac	**Windows**	
Package	Conda	**Pip**		LibTorch	Source
Language	**Python**			C++ / Java	
Compute Platform	**CUDA 11.8**	CUDA 12.1	CUDA 12.4	~~ROCm 6.2~~	CPU
Run this Command:	`pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`				

CUDA

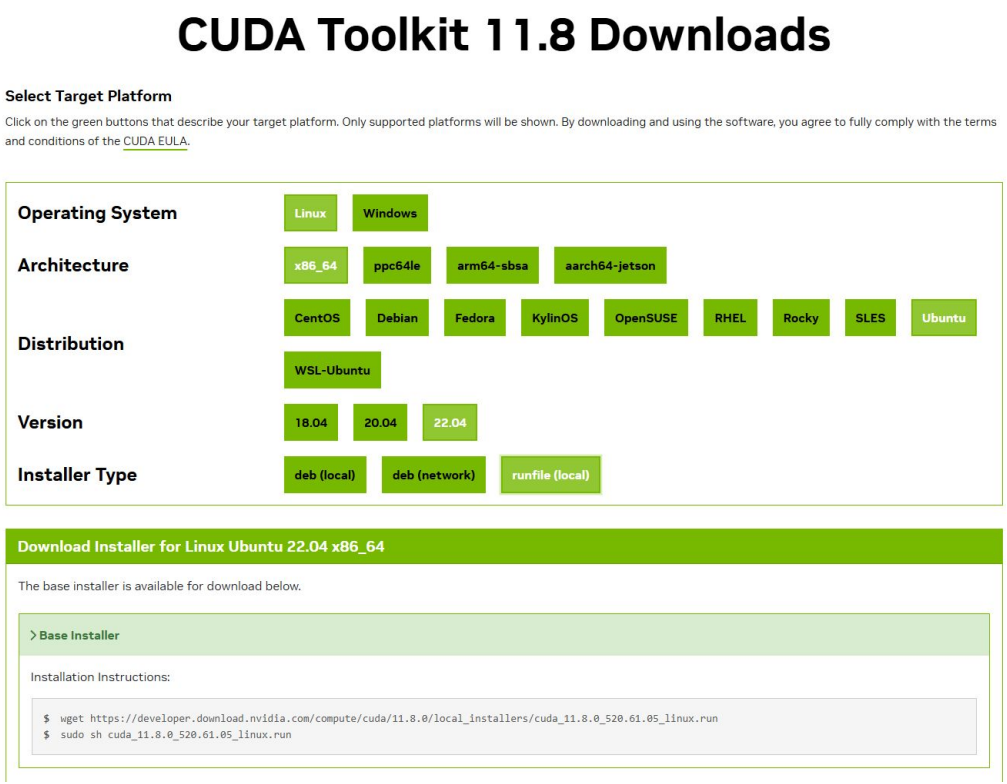
Библиотека для вычислений на GPU

В отличие от драйверов может быть поставлена в docker контейнере.

CUDA Toolkit	Minimum Required Driver Version for CUDA Minor Version Compatibility*	
	Linux x86_64 Driver Version	Windows x86_64 Driver Version
CUDA 12.x	>=525.60.13	>=528.33
CUDA 11.8.x CUDA 11.7.x CUDA 11.6.x CUDA 11.5.x CUDA 11.4.x CUDA 11.3.x CUDA 11.2.x CUDA 11.1.x	>=450.80.02	>=452.39
CUDA 11.0 (11.0.3)	>=450.36.06**	>=451.22**

https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html

CUDA и nvidia-drivers



https://developer.nvidia.com/cuda-downloads?target_os=Linux&target_arch=x86_64&Distribution=Ubuntu&target_version=24.04&target_type=runfile_local

nvidia-drivers

Можно ставить и через apt-get / аналоги

```
sudo ubuntu-drivers install nvidia:535
```

Предварительно рекомендуется удалить все предыдущие версии

```
sudo apt remove --purge '^nvidia-.*'
sudo apt remove --purge '^libnvidia-.*'
```

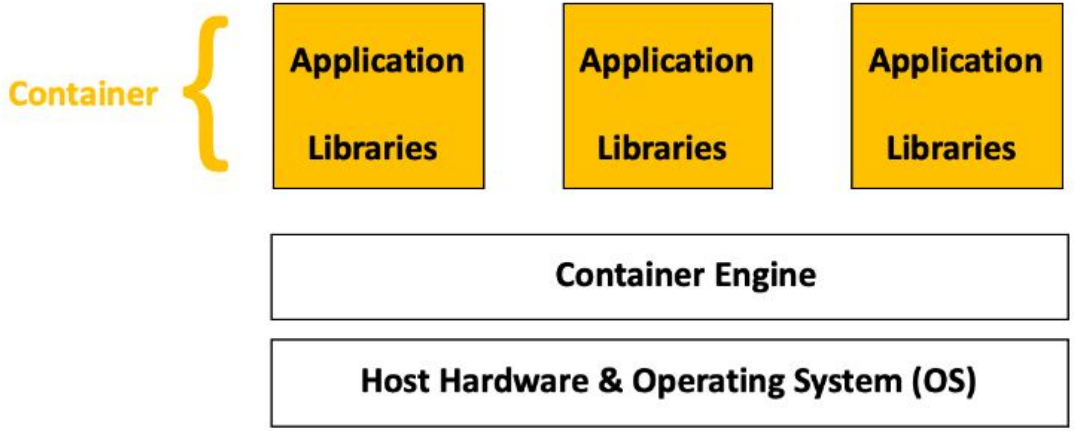
Но обычно проще ставить через installer без установки cuda, если она не нужна.

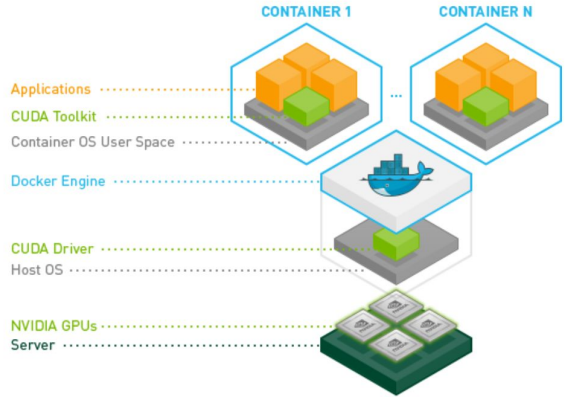
Docker

Виртуализация на уровне ОС, библиотек и приложений.

Драйвера все еще идут от Host OS, например, nvidia-drivers

OS, python, CUDA, pytorch … все можно ставить в docker контейнере

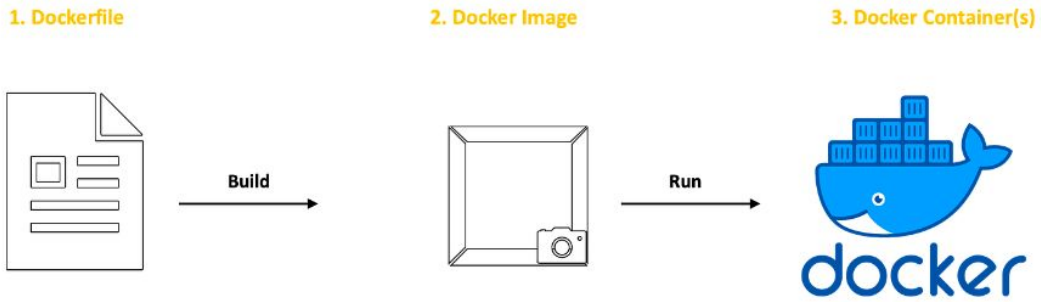




https://www.freecodecamp.org/news/how-docker-containers-work/

Docker: основные понятия

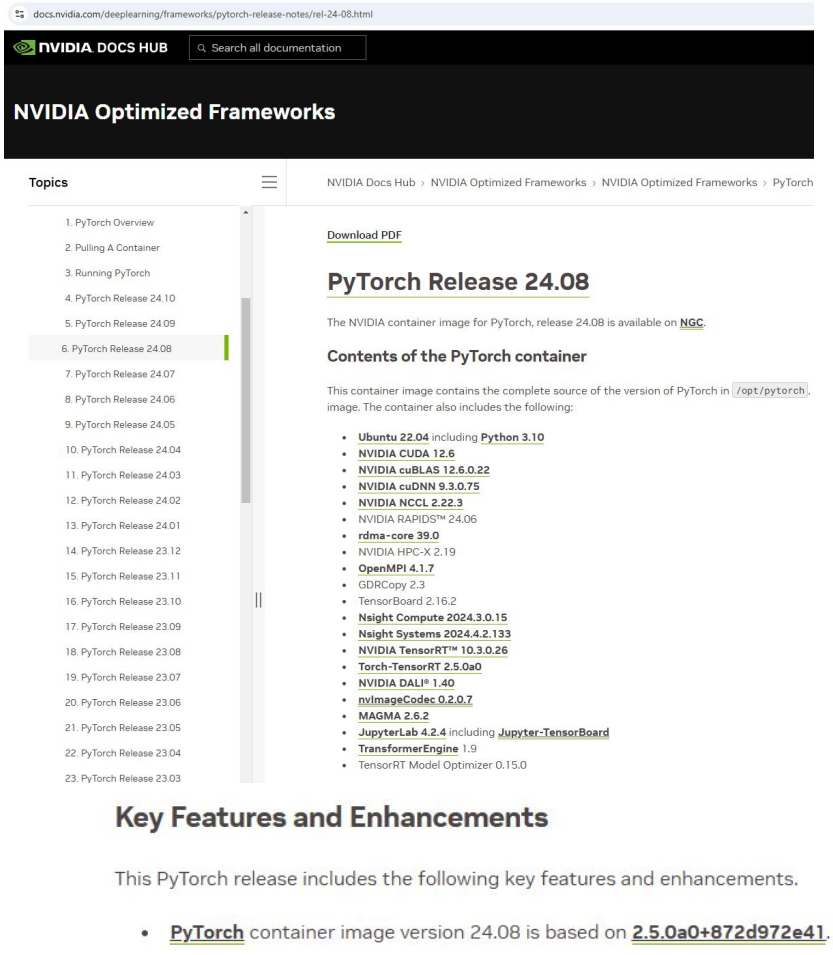
- Dockerfile
  - Конфигурация нашего образа
- Docker image
  - Собранный образ, может быть экспортирован
- Docker container
  - Запущенный docker image



Docker image: ngc

Для обучения в **multi-node** настоятельно рекомендуется брать ngc контейнер за базу (и следить, чтобы не переставлялся pytorch)!

Иначе:



Docker container

- Контейнер - запущенный image
- Все внутреннее состояние контейнера независимо от остальных контейнеров (не считая связанных volume):
  - Запустили контейнер, сделали что-то в нем, перезапустили - контейнер снова в начальном состоянии
- Чтобы пробрасывать gpu должен стоять nvidia-container-toolkit / nvidia-docker2

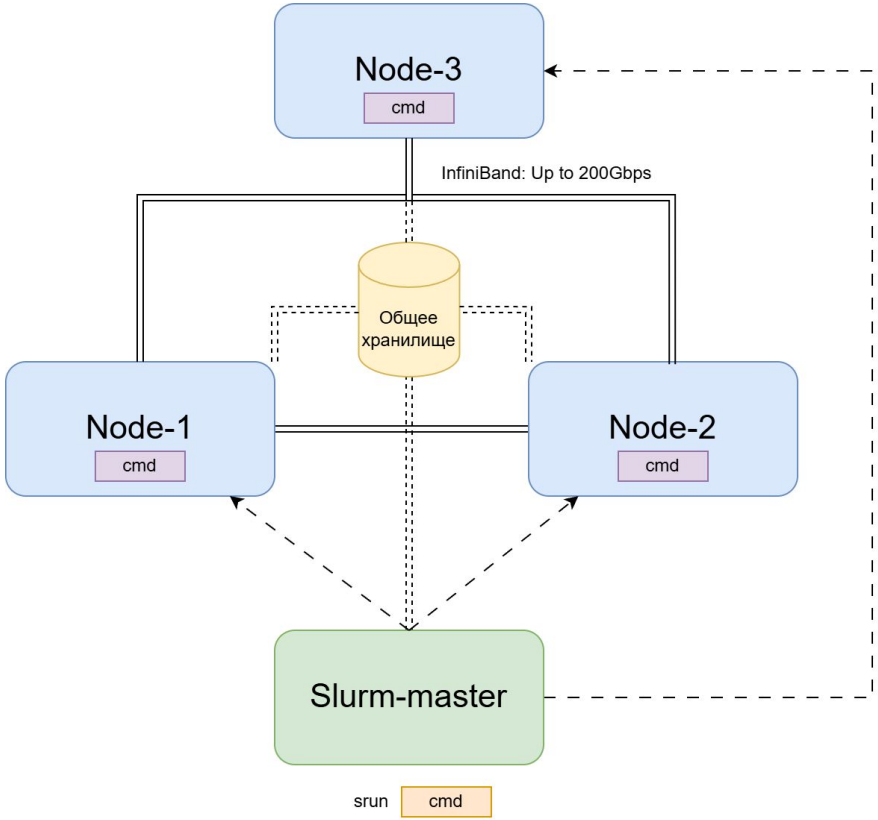
Docker container

```
docker run -v /home/ubuntu:/workdir -p 8000:8888 -it --gpus '"device=0,1"' --rm --name jupyter01 ngc_cuda_pytorch_vllm_11_10_24_v7 bash
```

- -v /host_path:/container_path
- -p host_port:container_port
  - При таком варианте открывает его наружу!
- --gpus all “пробросит” в контейнер все gpu машины
- --name jupyter01 - имя контейнера для удобства, --rm - после завершения работы контейнера, он будет удален (не image!)
- ngc_cuda… название image
- bash - команда на запуске, -it - интерактивный режим
- По умолчанию требуются sudo права, если не настроено иначе

Slurm

- Открытый пакет для менеджмента и распределения нагрузке в кластере.
- Прямого доступа на **вычислительный узел** по умолчанию нет
- Со slurm-master происходит запуск задач:
  - Задача ставится в выбранную очередь (partition)
  - При успешном запуске на каждом узле запускается целевой скрипт в заданном докер-контейнере



NCCL

Высокооптимизированная библиотека для multi-gpu коммуникаций, которая организует обмен данными между GPU, минимизируя “оверхед” на передачу данных в рамках процесса обучения.

- **Broadcast** - копирует данные с одной GPU (root GPU) на все остальные связанные с задачей GPU
- **Reduce** - производит операцию (сумма, максимум и тп) по всем участвующим GPU и сохраняет на root GPU.
- **AllReduce** - По сути как и Reduce, но сохраняет результат на каждой GPU
- **ReduceScatter** - Производит Reduce, но затем нарезает и распространяет результат по участвующим GPU одинаковыми блоками.
- **AllGather** - собирает данные со всех GPU, конкатенирует и затем распространяет результат по ним.

sbatch

Запуск через команду sbatch.

```
#!/bin/bash

#SBATCH --nodes=8                   # number of nodes
#SBATCH --gres=gpu:8                # number of GPUs per node
#SBATCH --time=4-23:59:59

export NNODES=8
export GPUS_PER_NODE=8

export head_node=( $( scontrol show hostnames $SLURM_JOB_NODELIST ) )
export head_node_ip=$(srun --nodes=1 --ntasks=1 --gpus 0 -w "$head_node" hostname --ip-address)

export HF_HOME=/scratch/tikhomirov/workdir/data/.cache/

echo Head Node: $head_node
echo Head Node IP: $head_node_ip
echo "${head_node_ip: -1}"
srun --container-image /scratch/tikhomirov/ngc_cuda_pytorch_24_04_v1+latest.sqsh --container-workdir /scratch/tikhomirov --container-mounts /scratch/tikhomirov:/scratch/tikhomirov bash -c "cd /scratch/tikhomirov/workdir/projects/ruadapt_training  && ./run_train_peft_32_128.sh"
```

- Конфигурация узлов и GPU по-умолчанию, а также лимит времени задается через #SBATCH
- head_node_ip нужен для синхронизации в скрипте запуска обучения через torchrun
- Логи пишутся в slurm-task_id.out

Скрипт, запускаемый уже на узлах

```
rdzv_id="512${head_node_ip: -1}"
rdzv_port="2650${head_node_ip: -1}"
echo $rdzv_id
echo $rdzv_port

torchrun --nnodes=$NNODES --nproc-per-node=$GPUS_PER_NODE --rdzv-id=$rdzv_id --rdzv-backend=c10d --rdzv-endpoint=$head_node_ip:$rdzv_port train_trainer.py \
```

- Multi-gpu/multi-node задачи запускаются через **torchrun** команду.
- Вся логика распараллеливания вычисления уже реализована в Trainer от huggingface.
- Требуется только правильно запустить скрипт!
  - Оптимальный batch_size и total batch size
  - По GPU должно влезать в одну карточку (в случае ddp/zero-2)

Алгоритм работы с кластером

1. Подготавливается скрипт обучения + данные в нужном формате
2. Подготавливается docker image с **нужным для запуска** скрипта обучения **окружением**
   a. Подготовка docker image идет где-то на сторонней машине
   b. В данном случае **docker image** должен быть еще **преобразован** в специальный формат **enroot**
      i. sudo enroot import dockerd://docker_name
      ii. enroot нужно предварительно поставить на машину
3. Подготавливается скрипт **run.sh**, который будет запускаться через **sbatch**
   a. Настраивается количество GPU, nodes, макс. время задачи, partition, oversubscribe

Алгоритм работы с кластером

4. Все скрипты и данные копируются на slurm-master
5. Командой **sbatch run.sh** запускается задача
6. Командой **squeue** смотрим себя в очереди
7. Командой **sinfo** можно посмотреть, сколько свободных (idle) задач в очереди
8. Командой **scancel** job_id можно принудительно завершить задачу
   a. Полезно, когда задача зависла, но при этом по сути упала и это видно в логах
9. В директории с run.sh будет создан **slurm-*taskid*.out** файл, в него все поднятые в рамках задачи процессы будут выводить свой output, соответственно, например, **cat slurm-*123123*.out**

Встречаемые проблемы

- Иногда какой-то узел не отрабатывает штатно, весь запуск падает и/или зависает. В основном помогает перезапуск задачи.
  - Иногда ошибки были на стороне CUDA, иногда на стороне NCCL.
- Однажды была проблема, когда узел упал и завис так, что scancel не помогал завершить задачу (7 из 8 узлов были “отпущены”, а проблемный держал задачу).

Разбор кода скрипта обучения LLM

https://github.com/RefalMachine/ruadapt/blob/main/ruadapt/pretraining/train_trainer.py

Разбор кода скрипта обучения LLM

326 - инициализируем парсер параметров

332 - парсим наши данные

363 - логгируем полезную информацию

```
321  def main():
322      # See all possible arguments in src/transformers/training_args.py
323      # or by passing the --help flag to this script.
324      # We now keep distinct sets of args, for a cleaner separation of concerns.
325
326      parser = HfArgumentParser((ModelArguments, DataTrainingArguments, LoraTrainingArguments))
327      if len(sys.argv) == 2 and sys.argv[1].endswith(".json"):
328          # If we pass only one argument to the script and it's the path to a json file,
329          # let's parse it to get our arguments.
330          model_args, data_args, training_args = parser.parse_json_file(json_file=os.path.abspath(sys.argv[1]))
331      else:
332          model_args, data_args, training_args = parser.parse_args_into_dataclasses()
333
363      logger.warning(
364          f"Process rank: {training_args.local_rank}, device: {training_args.device}, n_gpu: {training_args.n_gpu}"
365          + f"distributed training: {training_args.parallel_mode.value == 'distributed'}, 16-bits training: {training_args.fp16}"
366      )
```

Разбор кода скрипта обучения LLM

блок 423 - 444 - при загрузке датасета **с диска** в случае подачи train и val файлов.

```
423         else:
424             data_files = {}
425             dataset_args = {}
426             if data_args.train_file is not None:
427                 data_files["train"] = data_args.train_file
428             if data_args.validation_file is not None:
429                 data_files["validation"] = data_args.validation_file
430             extension = (
431                 data_args.train_file.split(".")[-1]
432                 if data_args.train_file is not None
433                 else data_args.validation_file.split(".")[-1]
434             )
435             if extension == "txt":
436                 extension = "text"
437                 dataset_args["keep_linebreaks"] = data_args.keep_linebreaks
438             raw_datasets = load_dataset(
439                 extension,
440                 data_files=data_files,
441                 cache_dir=model_args.cache_dir,
442                 token=model_args.token,
443                 **dataset_args,
444             )
```

Разбор кода скрипта обучения LLM: пример входных данных

{"text": "The MK5000C was powered by the 5000 hp Caterpillar V12 3612 diesel engine. This diesel engine remains one of the largest engine blocks used in rail service in North America. The Cat 3612 features a 280 mm (11.0 in) bore with a 30
0 mm (11.8 in) stroke and has a 1,121 cu in (18.37 l) displacement per cylinder, 13,456 cu in (220.50 l) total. The 3612 has dual turbochargers that are liquid aftercooled. The 3612 idles at 300 rpm and has a maximum speed of 1000 rpm.\nTh
e Caterpillar 3612 drove a KATO 16P12-27000 main alternator which was capable of handling 8400 amperes at 1315 V DC at 1000 rpm. The power generated by the main alternator drove 6 MK1000 traction motors, each with a gear ratio of 83:20 and
 connected to 40-inch (1,016 mm) wheels which allowed the MK5000C a maximum speed of 70 mph (110 km/h). The MK5000C rode on two 3 axle Dofasco designed bolster-less trucks, the same that many Canadian MLW and GE designed locomotives ride o
n.\nThe first 3 MK5000C were 71 ft 2 in (21.69 m) long, while the last three were 73 ft 4 in (22.35 m) in length, all 6 were 15 ft 11 1/2 in (4.864 m) tall and 10 ft (3.0 m) wide. The MK5000C weighed 396,000 lb (180 t). \nLike most modern
locomotives the MK5000C was microprocessor controlled, using an in-house designed system called the MK-LOC. This system monitored the performance of all aspects of the locomotive and controlled the power output as well as the traction cont
rol/adhesion of the locomotive. The MK5000C also had electro-pneumatic braking, provided by the EPIC 3102 air brake system which can be found on locomotives of other builders.\nThe MK5000C carried 5300 US gallons (20,100 L) of diesel, 246
 US gal (931 L) of lubricating oil, and 320 gallons (1,210 L) of coolant. This coolant system was unlike that of most other North American locomotives, using a water/antifreeze mix; only with a few Caterpillar repowered switchers and the E
lectro Motive Division SD90MAC share this trait.\nThe MK5000C generated 118,000 lbf (525 kN) of continuous tractive effort, and produced around 35% adhesion on dry rail.\nThe MK5000C at first look appears similar to many 1990s era EMD loco
motives. The MK5000C has a fuel tank and long hood that appear very similar to EMD designs, however mechanically the MK5000C shares very little in common with any EMD product.\nA total of 6 examples were built, three in August 1994 for dem
onstration on the Southern Pacific Railroad, and another three in August 1995 for demonstration on the Union Pacific Railroad. Due to termination of the MK Rail high horsepower program neither railroad purchased the model, and the units we
re returned after one year of demonstrations. Production was stopped after the sale of MK Rail in 1996, and 3 more partially built units sat in storage until 2001 when their frames were scrapped by MK Rail successor MotivePower Industries.
 One of the MK5000C cabs was used on DM&E 5000, a former ATSF SD45B rebuilt into a \"SD50M-3\". This unit is still in service as MPEX 5000, and can be found in lease service on various railroads in North America.\nIn 2001 the Utah Railway
tested and later acquired all 6 units from Wabtec, the owner of MotivePower Industries. However, after one year of operation, all units were out of service due to problems with the main bearings on the Caterpillar 3612 diesel engine and Ka
to main alternator. The units were returned to Wabtec and had the CAT 3612 and Kato main alternator removed and replaced with an EMD AR11 main alternator. At the same time, the engine blocks were replaced by EMD 3500 Horsepower 16-645F3B d
iesel engines from 5 retired Union Pacific EMD SD50 and 1 retired Union Pacific EMD GP50 locomotives. The 6 units were reclassed with the designation MK50-3 and are now back in service with the Utah Railway.\nIn March 2017 four units were
prepared to be shipped to the Kyle Railroad, a few months after Utah Railway's coal train contracts expired. A BNSF train picked up the four units and left with them on March 14, 2017. A fifth unit, left the Utah Railway in late March/Apri
l, with the final unit Utah Railway 5003 departing on April 6, 2017", "domain": "enwiki"}

.jsonl файл, где каждая строка - dict с key “text” и “domain”

Тексты бывают очень длинные, что сказывается на скорости работы токенайзера, требуется спец. обработка.

Разбор кода скрипта обучения LLM

Загрузка конфига и токенайзера

```
479         if model_args.config_name:
480             config = AutoConfig.from_pretrained(model_args.config_name, **config_kwargs)
481         elif model_args.model_name_or_path:
482             config = AutoConfig.from_pretrained(model_args.model_name_or_path, **config_kwargs)
-------------------------------------------------------------------------------------------------
498         if model_args.tokenizer_name:
499             tokenizer = AutoTokenizer.from_pretrained(model_args.tokenizer_name, **tokenizer_kwargs)
500         elif model_args.model_name_or_path:
501             tokenizer = AutoTokenizer.from_pretrained(model_args.model_name_or_path, **tokenizer_kwargs)
```

Разбор кода скрипта обучения LLM

522 - функция токенизации текстов из содержимого поля “text” в каждом примере.

524 - custom_tokenize - в данном случае для обработки длинных текстов

535 - применение функции на данных в num_workers потоках (будет запущено на главном процессе **на каждом** задействованном **узле**!)

Как следствие, рекомендация: сначала запуск **на 1 узле** для предобработки данных (**кэшируются**), затем уже на обучение на N узлах.

```
522     def tokenize_function(examples):
523         with CaptureLogger(tok_logger) as cl:
524             output = custom_tokenize(examples[text_column_name], tokenizer,
525
526         if "Token indices sequence length is longer than the" in cl.out:
527             tok_logger.warning(
528                 "^^^^^^^^^^^^^^^^ Please ignore the warning above - this lor
529                 " before being passed to the model."
530             )
531         return output
532
533     with training_args.main_process_first(desc="dataset map tokenization"):
534         if not data_args.streaming:
535             tokenized_datasets = raw_datasets.map(
536                 tokenize_function,
537                 batched=False,
538                 num_proc=data_args.preprocessing_num_workers,
539                 remove_columns=column_names,
540                 load_from_cache_file=not data_args.overwrite_cache,
541                 desc="Running tokenizer on dataset",
542             )
```

Разбор кода скрипта обучения LLM

582 - группировка токенизированных текстов!

В варианте от hf токены всех текстов конкатенируются, а затем **нарезаются на block_size** как получится.

В данной реализации каждый элемент батча обязан стартовать либо с начала документа, либо с начала абзаца

595 - спец. переменная окружения, чтобы завершить исполнение кода в этом месте

```
577         def group_texts_function(examples):
578             return group_texts(examples, tokenizer, tokenizer_prop, block_size,
579 
580         with training_args.main_process_first(desc="grouping texts together"):
581             if not data_args.streaming:
582                 lm_datasets = tokenized_datasets.map(
583                     group_texts_function,
584                     batched=True,
585                     num_proc=data_args.preprocessing_num_workers,
586                     load_from_cache_file=not data_args.overwrite_cache,
587                     desc=f"Grouping texts in chunks of {block_size}",
588                 )
589             else:
590                 lm_datasets = tokenized_datasets.map(
591                     group_texts_function,
592                     batched=True,
593                 )
594 
595         if int(os.environ.get('RUADAPT_NO_TRAIN', 0)) == 1:
596             return 0
597 
```

Разбор кода скрипта обучения LLM

605 - загрузка моделей на cpu, но можно грузить сразу на нужный девайс.

Положение загрузки моделей изменено относительно исходного скрипта от HF - тут после подготовки датасета, у них до токенизации

```
598         MODEL_CLASS = AutoModelForCausalLM
599         if model_args.model_name_or_path:
600             torch_dtype = (
601                 model_args.torch_dtype
602                 if model_args.torch_dtype in ["auto", None]
603                 else getattr(torch, model_args.torch_dtype)
604             )
605             model = MODEL_CLASS.from_pretrained(
606                 model_args.model_name_or_path,
607                 from_tf=bool(".ckpt" in model_args.model_name_or_path),
608                 config=config,
609                 cache_dir=model_args.cache_dir,
610                 revision=model_args.model_revision,
611                 token=model_args.token,
612                 torch_dtype=torch_dtype,
613                 low_cpu_mem_usage=True,
614                 attn_implementation="flash_attention_2"
615                 #device_map={"": f"cuda:{training_args.local_rank}"}
616             )
```

Разбор кода скрипта обучения LLM

```
664         if training_args.peft:
665             logger.info("Init new peft model")
666             target_modules = training_args.trainable.split(',')
667             modules_to_save = training_args.modules_to_save
668             if modules_to_save is not None and len(modules_to_save) > 0 and modules_to_save != 'None' a
669                 modules_to_save = modules_to_save.split(',')
670             else:
671                 modules_to_save = None
672 
673             lora_rank = training_args.lora_rank
674             lora_dropout = training_args.lora_dropout
675             lora_alpha = training_args.lora_alpha
676             use_dora = training_args.use_dora
677             logger.info(f"use dora: {use_dora}")
678             logger.info(f"target_modules: {target_modules}")
679             logger.info(f"default lora_rank: {lora_rank}")
680             rank_pattern = {}
681             alpha_pattern = {}
682             if training_args.rank_pattern_path is not None:
683                 rank_pattern = json.load(codecs.open(training_args.rank_pattern_path, 'r', 'utf-8'))
684             if training_args.alpha_pattern_path is not None:
685                 alpha_pattern = json.load(codecs.open(training_args.alpha_pattern_path, 'r', 'utf-8'))
```

Разбор кода скрипта обучения LLM

В случае peft=True инициализируются необходимые переменные и модель

Иначе в данном случае все слои кроме входных и выходных эмбеддингов замораживаются - обучаются только эмбеддинги.

При желании обучать всю модель полным тюном, часть с requires_grad = False убрать.

```
686             peft_config = LoraConfig(
687                 task_type=TaskType.CAUSAL_LM,
688                 target_modules=target_modules,
689                 inference_mode=False,
690                 r=lora_rank, lora_alpha=lora_alpha,
691                 lora_dropout=lora_dropout,
692                 modules_to_save=modules_to_save,
693                 rank_pattern=rank_pattern,
694                 alpha_pattern=alpha_pattern,
695                 use_dora=use_dora)
696             model = get_peft_model(model, peft_config)
697             model.print_trainable_parameters()
698         else:
699             logger.info("Ruadapt default")
700             for param_name, param in model.model.named_parameters():
701                 if 'embed_tokens' not in param_name:
702                     param.requires_grad = False
```

Разбор кода скрипта обучения LLM

709 - инициализация Trainer: передаем модель, параметры, датасеты и тп.

730 - важный момент для обучения peft + gradient_checkpointing. Без этого работать не будет

742 - запускаем обучение!

```
708         TRAINER_CLASS = TrainerNoBaseSave if training_args.peft else Trainer
709         trainer = TRAINER_CLASS(
710             model=model,
711             args=training_args,
712             train_dataset=train_dataset if training_args.do_train else None,
713             eval_dataset=eval_dataset if training_args.do_eval else None,
714             tokenizer=tokenizer,
715             # Data collator will default to DataCollatorWithPadding, so we change it.
716             data_collator=default_data_collator,
717             compute_metrics=compute_metrics if training_args.do_eval and not is_torch_
718             preprocess_logits_for_metrics=preprocess_logits_for_metrics
719             if training_args.do_eval and not is_torch_tpu_available()
720             else None,
721         )
730         if training_args.peft:
731             if training_args.gradient_checkpointing:
732                 model.enable_input_require_grads()
736         if training_args.do_train:
737             checkpoint = None
738             if training_args.resume_from_checkpoint is not None:
739                 checkpoint = training_args.resume_from_checkpoint
740             elif last_checkpoint is not None:
741                 checkpoint = last_checkpoint
742             train_result = trainer.train(resume_from_checkpoint=checkpoint)
743             trainer.save_model()  # Saves the tokenizer too for easy upload
```

Разбор кода скрипта обучения LLM: типовой скрипт

```
torchrun --nnodes=$NNODES --nproc-per-node=$GPUS_PER_NODE --rdzv-id=$rdzv_id --rdzv-backend=c10d --rdzv-endpoint=$head_node_ip:$rdzv_port train_trainer.py \
--model_name_or_path $MODEL_NAME_OR_PATH \
--train_file $TRAIN_FILE_PATH \
--validation_file /scratch/tikhomirov/workdir/data/darulm_20_05_24/val.json \
--block_size 4096 \
--preprocessing_num_workers 96 \
--output_dir $OUTPUT_DIR \
--overwrite_output_dir \
--do_train \
--do_eval \
--evaluation_strategy steps \
--per_device_train_batch_size 1 \
--per_device_eval_batch_size 1 \
--learning_rate $LR \
--weight_decay 0.1 \
--adam_beta1 0.9 \
--adam_beta2 0.95 \
--adam_epsilon 1e-05 \
--num_train_epochs 1.0 \
--lr_scheduler_type cosine \
--warmup_steps 100 \
--save_step 10000 \
--logging_steps 10 \
--save_total_limit 8 \
--bf16 \
--bf16_full_eval \
--torch_dtype bfloat16 \
--gradient_accumulation_steps 4 \
--eval_steps 1000 \
--log_on_each_node false \
--peft $PEFT \
--lora_rank ${lora_rank} \
--lora_alpha ${lora_alpha} \
--trainable ${lora_trainable} \
--lora_dropout ${lora_dropout} \
--modules_to_save ${modules_to_save}
```

https://github.com/RefalMachine/ruadapt/blob/main/ruadapt/pretraining/run_train_ruadapt.sh

Подводя итог: запуск задачи на кластере

- Подготовили docker контейнер, с правильным окружением
  - наследуемся от ngc, torch не трогаем, во время обучения проверяем, что ускорение кратное в зависимости от количества GPU
  - конвертация через enroot
- Подготовили данные, модель + .py скрипт обучения + .sh скрипт, который запускает .py скрипт с нужными параметрами
- Подготовили .sh файл, который будет запускаться через sbatch - соответственно запускать наше обучение на каждой из выделенных на задачу GPU
- Готово!