Open in Colab

🚀 Запуск модели в Google Colab

⚙️ Требования

- Убедитесь, что выбрано **GPU** как тип оборудования.
- Рекомендуется использовать **Tesla T4** — она доступна бесплатно в Google Colab.

🟢 Инструкция по запуску

1. Откройте вкладку **«Среда выполнения»** (*Runtime*) в верхнем меню.
2. Выберите **«Сменить тип среды выполнения»** (*Change runtime type*).
3. В поле **Аппаратное ускорение** (*Hardware accelerator*) выберите **GPU** и нажмите **«Сохранить»**.
4. Затем снова откройте **«Среда выполнения»** и выберите **«Выполнить всё»** (*Run all*), чтобы запустить все ячейки.

💡 Подсказка

При первом запуске Google Colab может запросить разрешение на выполнение кода — подтвердите, если уверены в источнике.

💻 Установка Unsloth на локальный компьютер

Чтобы установить **Unsloth** на свой компьютер, следуйте инструкциям по установке на странице GitHub:
👉 Официальная инструкция по установке

После установки вы узнаете:

- 📂 Как подготовить данные
- 🧠 Как обучить модель
- ⚙️ Как запустить модель для инференса
- 💾 Как сохранить модель

Инсталляцмя зависимостей

```
%%capture
import os
if "COLAB_" not in "".join(os.environ.keys()):
    !pip install unsloth
else:
    # Do this only in Colab notebooks! Otherwise use pip install unsloth
    !pip install --no-deps bitsandbytes accelerate xformers==0.0.29.post3 peft trl==0.15.2 triton cut_cross_entropy unsloth_zoo
    !pip install sentencepiece protobuf datasets huggingface_hub hf_transfer
    !pip install --no-deps unsloth
```

Unsloth

Unsloth — это фреймворк (или набор инструментов) с открытым исходным кодом, предназначенный для ускоренного и эффективного обучения больших языковых моделей (LLM), таких как LLaMA, Mistral и других, на вашем собственном железе (в том числе ноутбуках и локальных серверах).

Основные особенности Unsloth:
• 🔧 Поддержка LLaMA и Mistral: оптимизирована для LLaMA 2 и других моделей с открытым исходным кодом.
• ⚡ Очень быстрая дообучаемость (fine-tuning): обещают 2–5× более быстрое обучение по сравнению с обычным PyTorch или Hugging Face.
• 💾 Низкие требования к ресурсам: позволяет тренировать 7B моделей даже на одной GPU с 8 ГБ VRAM.
• 🧠 Интеграция с PEFT/LoRA: обучение происходит с помощью LoRA (Low-Rank Adaptation), что делает его гораздо легче и дешевле.
• 🤝 Совместим с Hugging Face: можно загружать модели и датасеты напрямую из Hugging Face Hub.
• 🔄 Интеграция с Colab и Kaggle: можно запускать и обучать прямо в облаке.

```
from unsloth import FastLanguageModel
import torch
max_seq_length = 2048 # Choose any! We auto support RoPE Scaling internally!
dtype = None # None for auto detection. Float16 for Tesla T4, V100, Bfloat16 for Ampere+
load_in_4bit = True # Use 4bit quantization to reduce memory usage. Can be False.

# 4bit pre quantized models we support for 4x faster downloading + no OOMs.
fourbit_models = [
    "unsloth/Meta-Llama-3.1-8B-bnb-4bit",      # Llama-3.1 15 trillion tokens model 2x faster!
    "unsloth/Meta-Llama-3.1-8B-Instruct-bnb-4bit",
    "unsloth/Meta-Llama-3.1-70B-bnb-4bit",
    "unsloth/Meta-Llama-3.1-405B-bnb-4bit",    # We also uploaded 4bit for 405b!
    "unsloth/Mistral-Nemo-Base-2407-bnb-4bit", # New Mistral 12b 2x faster!
    "unsloth/Mistral-Nemo-Instruct-2407-bnb-4bit",
    "unsloth/mistral-7b-v0.3-bnb-4bit",        # Mistral v3 2x faster!
    "unsloth/mistral-7b-instruct-v0.3-bnb-4bit",
    "unsloth/Phi-3.5-mini-instruct",           # Phi-3.5 2x faster!
    "unsloth/Phi-3-medium-4k-instruct",
    "unsloth/gemma-2-9b-bnb-4bit",
    "unsloth/gemma-2-27b-bnb-4bit",            # Gemma 2x faster!
] # More models at https://huggingface.co/unsloth

model, tokenizer = FastLanguageModel.from_pretrained(
    # Can select any from the below:
    # "unsloth/Qwen2.5-0.5B", "unsloth/Qwen2.5-1.5B", "unsloth/Qwen2.5-3B"
    # "unsloth/Qwen2.5-14B",  "unsloth/Qwen2.5-32B",  "unsloth/Qwen2.5-72B",
    # And also all Instruct versions and Math. Coding verisons!
    model_name = "unsloth/Qwen2.5-7B",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,
    # token = "hf_...", # use one if using gated models like meta-llama/Llama-2-7b-hf
)
```

🧩 Добавление адаптеров LoRA

Теперь мы добавляем **LoRA-адаптеры**, благодаря чему для обучения потребуется обновлять всего **от 1 до 10% параметров** модели!

Это значительно снижает потребление памяти и ускоряет обучение, особенно на ограниченных ресурсах (например, на одной GPU).

```
model = FastLanguageModel.get_peft_model(
    model,
    r = 16, # Choose any number > 0 ! Suggested 8, 16, 32, 64, 128
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0, # Supports any, but = 0 is optimized
    bias = "none",    # Supports any, but = "none" is optimized
    # [NEW] "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
    use_gradient_checkpointing = "unsloth", # True or "unsloth" for very long context
    random_state = 3407,
    use_rslora = False,  # We support rank stabilized LoRA
    loftq_config = None, # And LoftQ
)
```

Подготовка данных

🦙 Использование датасета Alpaca

Мы используем датасет **Alpaca** от yahma — это отфильтрованная версия оригинального датасета Alpaca (52K), созданного Стэнфордом.

Вы можете заменить этот раздел кода своей собственной процедурой подготовки данных.

⚠️ Важные замечания

> **[ПРИМЕЧАНИЕ]** Чтобы обучать модель **только на продолжениях** (игнорируя пользовательский ввод), ознакомьтесь с документацией TRL:
> 🔗 Train on completions only

> **[ПРИМЕЧАНИЕ]** Обязательно добавляйте **EOS_TOKEN** (токен конца последовательности) в токенизированный выход!
> Без него модель может генерировать бесконечный текст.

📘 Рекомендуемые ноутбуки

- 💬 Для использования шаблона `llama-3` с датасетами в формате **ShareGPT**:
  Conversational Notebook (Llama 3, 8B)-Alpaca.ipynb)
- 📝 Для генерации продолжений текста (например, написание новелл):
  Text Completion Notebook (Mistral, 7B)-Text_Completion.ipynb)

```
alpaca_prompt = """Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately compl

### Instruction:
{}

### Input:
{}

### Response:
{}"""

EOS_TOKEN = tokenizer.eos_token # Must add EOS_TOKEN
def formatting_prompts_func(examples):
    instructions = examples["instruction"]
    inputs       = examples["input"]
    outputs      = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        # Must add EOS_TOKEN, otherwise your generation will go on forever!
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return { "text" : texts, }
pass

from datasets import load_dataset
dataset = load_dataset("yahma/alpaca-cleaned", split = "train")
dataset = dataset.map(formatting_prompts_func, batched = True,)
```

```
import pandas as pd

# Преобразование в DataFrame
df = dataset.to_pandas()
df.head()
```

🧠 Обучение модели

Теперь воспользуемся `SFTTrainer` из библиотеки Hugging Face TRL!

По умолчанию выполняется **60 шагов**, чтобы ускорить процесс обучения. Однако вы можете задать полный проход по данным, установив:

- `num_train_epochs=1`
- и отключив ограничение по количеству шагов: `max_steps=None`

Также поддерживается обучение с использованием `DPOTrainer` от TRL для продвинутых сценариев тонкой настройки!

```
from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported

trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    dataset_num_proc = 2,
    packing = False, # Can make training 5x faster for short sequences.
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 5,
        # num_train_epochs = 1, # Set this for 1 full training run.
        max_steps = 60,
        learning_rate = 2e-4,
        fp16 = not is_bfloat16_supported(),
        bf16 = is_bfloat16_supported(),
        logging_steps = 1,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
        report_to = "none", # Use this for WandB etc
    ),
)
```

```
# @title Show current memory stats
gpu_stats = torch.cuda.get_device_properties(0)
start_gpu_memory = round(torch.cuda.max_memory_reserved() / 1024 / 1024 / 1024, 3)
max_memory = round(gpu_stats.total_memory / 1024 / 1024 / 1024, 3)
print(f"GPU = {gpu_stats.name}. Max memory = {max_memory} GB.")
print(f"{start_gpu_memory} GB of memory reserved.")
```

```
trainer_stats = trainer.train()
```

```
# @title Show final memory and time stats
used_memory = round(torch.cuda.max_memory_reserved() / 1024 / 1024 / 1024, 3)
used_memory_for_lora = round(used_memory - start_gpu_memory, 3)
used_percentage = round(used_memory / max_memory * 100, 3)
lora_percentage = round(used_memory_for_lora / max_memory * 100, 3)
print(f"{trainer_stats.metrics['train_runtime']} seconds used for training.")
print(
    f"{round(trainer_stats.metrics['train_runtime']/60, 2)} minutes used for training."
)
print(f"Peak reserved memory = {used_memory} GB.")
print(f"Peak reserved memory for training = {used_memory_for_lora} GB.")
print(f"Peak reserved memory % of max memory = {used_percentage} %.")
print(f"Peak reserved memory for training % of max memory = {lora_percentage} %.")
```

🔍 Инференс (использование модели)

Запустим модель! Вы можете изменить **инструкцию** и **ввод**, оставив поле **вывода** пустым — модель сгенерирует ответ автоматически.

Инференс (от англ. inference) в контексте машинного обучения и нейросетей — это процесс применения обученной модели для получения предсказаний на новых, ранее не виденных данных.

🔍 Другими словами:

Если обучение — это то, когда модель "учится" на примерах, то инференс — это когда она "отвечает" на реальные запросы.

📌 Примеры инференса: • Вы вводите вопрос в чат-бот → модель генерирует ответ → это инференс. • Вы загружаете изображение в систему → она распознаёт, что на нём кот → тоже инференс. • Автокомплит текста в IDE → инференс модели кода.

🧠 Технически:

Инференс включает: • Токенизацию входа • Прогон входа через обученную нейросеть • Получение выходных токенов (предсказаний) • Декодирование ответа

```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Continue the fibonnaci sequence.", # instruction
        "1, 1, 2, 3, 5, 8", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

outputs = model.generate(**inputs, max_new_tokens = 64, use_cache = True)
tokenizer.batch_decode(outputs)
```

Использование `TextStreamer` для непрерывного инференса

Для реализации непрерывного вывода токенов при генерации текста можно использовать компонент `TextStreamer`. Это позволяет получать результаты по мере их появления — токен за токеном — без необходимости ожидания завершения всей генерации.

Преимущества использования `TextStreamer`:

- **Реализация стриминга вывода** в реальном времени в интерфейсе пользователя.
- **Повышенная отзывчивость** при работе с длинными запросами или большими языковыми моделями.
- **Удобство отладки** и наблюдения за процессом генерации текста по шагам.

```
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Continue the fibonnaci sequence.", # instruction
        "1, 1, 2, 3, 5, 8", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 128)
```

Сохранение и загрузка дообученных моделей

Для сохранения результатов дообучения в формате LoRA-адаптеров используйте один из следующих методов:

- `push_to_hub()` — для загрузки адаптеров напрямую в репозиторий Hugging Face.
- `save_pretrained()` — для локального сохранения адаптеров на диск.

> **[ПРИМЕЧАНИЕ]**
> Эти методы сохраняют **только LoRA-адаптеры**, а не полную модель.
> Если требуется сохранить модель в формате 16-бит или GGUF, смотрите соответствующий раздел ниже.

```
model.save_pretrained("lora_model")  # Local saving
tokenizer.save_pretrained("lora_model")
# model.push_to_hub("your_name/lora_model", token = "...") # Online saving
# tokenizer.push_to_hub("your_name/lora_model", token = "...") # Online saving
```

Загрузка LoRA-адаптеров для инференса

Примечание: Переключение соответствующего флага (False → True) активирует использование LoRA-адаптеров при инференсе. Это позволяет использовать дообученные параметры без необходимости загружать полную модель.

```
if False:
    from unsloth import FastLanguageModel
    model, tokenizer = FastLanguageModel.from_pretrained(
        model_name = "lora_model", # YOUR MODEL YOU USED FOR TRAINING
        max_seq_length = max_seq_length,
        dtype = dtype,
        load_in_4bit = load_in_4bit,
    )
    FastLanguageModel.for_inference(model) # Enable native 2x faster inference

# alpaca_prompt = You MUST copy from above!

inputs = tokenizer(
[
    alpaca_prompt.format(
        "What is a famous tall tower in Paris?", # instruction
        "", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 128)
```

Вопрос к fine-tune модели

```
if False:
    from unsloth import FastLanguageModel
    model, tokenizer = FastLanguageModel.from_pretrained(
        model_name = "lora_model", # YOUR MODEL YOU USED FOR TRAINING
        max_seq_length = max_seq_length,
        dtype = dtype,
        load_in_4bit = load_in_4bit,
    )
    FastLanguageModel.for_inference(model) # Enable native 2x faster inference

# alpaca_prompt = You MUST copy from above!

inputs = tokenizer(
[
    alpaca_prompt.format(
        "Дай три совета чтобы оставаться здоровым",
        "", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

from transformers import TextStreamer
text_streamer = TextStreamer(tokenizer)
_ = model.generate(**inputs, streamer = text_streamer, max_new_tokens = 200)
```

Альтернатива: `AutoModelForPeftCausalLM` от Hugging Face

Вы также можете использовать `AutoModelForPeftCausalLM` от Hugging Face для загрузки моделей с LoRA-адаптерами:

```
from transformers import AutoModelForPeftCausalLM

model = AutoModelForPeftCausalLM.from_pretrained("path/to/peft-model")

⚠️ Важные замечания

Используйте этот подход только в случае, если у вас не установлен unsloth.
Причины:
    •   Поддержка загрузки моделей в 4bit отсутствует, что приводит к значительному увеличению времени загрузки.
    •   Инференс через unsloth работает в 2 раза быстрее, особенно при использовании оптимизаций с bitsandbytes.
```

```
if False:
    # I highly do NOT suggest - use Unsloth if possible
    from peft import AutoPeftModelForCausalLM
    from transformers import AutoTokenizer
    model = AutoPeftModelForCausalLM.from_pretrained(
        "lora_model", # YOUR MODEL YOU USED FOR TRAINING
        load_in_4bit = load_in_4bit,
    )
    tokenizer = AutoTokenizer.from_pretrained("lora_model")
```

Сохранение в `float16` для VLLM

Поддерживается прямое сохранение модели в формате `float16`, совместимом с VLLM.
Для этого выберите один из следующих режимов:

- `merged_16bit` — сохраняет полную модель в формате **float16**.
- `merged_4bit` — сохраняет модель в **int4** (для экономии памяти).
- `lora` — сохраняет только **LoRA-адаптеры** (в качестве запасного варианта).

Загрузка модели на Hugging Face

Для публикации объединённой модели используйте метод `push_to_hub_merged()`:

```
model.push_to_hub_merged("your-username/model-name", token="your_hf_token")
```

```
# Merge to 16bit
if False: model.save_pretrained_merged("model", tokenizer, save_method = "merged_16bit",)
if False: model.push_to_hub_merged("hf/model", tokenizer, save_method = "merged_16bit", token = "")

# Merge to 4bit
if False: model.save_pretrained_merged("model", tokenizer, save_method = "merged_4bit",)
if False: model.push_to_hub_merged("hf/model", tokenizer, save_method = "merged_4bit", token = "")

# Just LoRA adapters
if False: model.save_pretrained_merged("model", tokenizer, save_method = "lora",)
if False: model.push_to_hub_merged("hf/model", tokenizer, save_method = "lora", token = "")
```

Конвертация в GGUF / llama.cpp

Теперь мы **нативно поддерживаем сохранение в формате GGUF** (совместимо с `llama.cpp`)!

Основные возможности:

- Автоматическое клонирование репозитория `llama.cpp` (если требуется).
- По умолчанию модель сохраняется в формате `q8_0`.
- Поддерживаются и другие схемы квантования, включая `q4_k_m`, `q5_k_m` и т.д.

Методы сохранения:

- `save_pretrained_gguf()` — локальное сохранение GGUF-модели.
- `push_to_hub_gguf()` — загрузка GGUF-файла в ваш репозиторий Hugging Face.

Поддерживаемые методы квантования *(подробнее в Wiki)*:

- **`q8_0`** — Быстрая конвертация. Высокая точность, но требует больше ресурсов.
- **`q4_k_m`** — **Рекомендуемый вариант.** Использует `Q6_K` для половины тензоров (`attention.wv`, `feed_forward.w2`), остальные — `Q4_K`.
- **`q5_k_m`** — Также **рекомендуется.** Похож на `q4_k_m`, но с использованием `Q5_K`.

🔄 Автоэкспорт в Ollama

> **[НОВОЕ]**
> Хотите дообучить модель и сразу экспортировать её в Ollama? Воспользуйтесь Colab-блокнотом-Ollama.ipynb) для автоматизации этого процесса.

```
# Save to 8bit Q8_0
if False: model.save_pretrained_gguf("model", tokenizer,)
# Remember to go to https://huggingface.co/settings/tokens for a token!
# And change hf to your username!
if False: model.push_to_hub_gguf("hf/model", tokenizer, token = "")

# Save to 16bit GGUF
if False: model.save_pretrained_gguf("model", tokenizer, quantization_method = "f16")
if False: model.push_to_hub_gguf("hf/model", tokenizer, quantization_method = "f16", token = "")

# Save to q4_k_m GGUF
if False: model.save_pretrained_gguf("model", tokenizer, quantization_method = "q4_k_m")
if False: model.push_to_hub_gguf("hf/model", tokenizer, quantization_method = "q4_k_m", token = "")

# Save to multiple GGUF options - much faster if you want multiple!
if False:
    model.push_to_hub_gguf(
        "hf/model", # Change hf to your username!
        tokenizer,
        quantization_method = ["q4_k_m", "q8_0", "q5_k_m",],
        token = "", # Get a token at https://huggingface.co/settings/tokens
    )
```

Использование `.gguf` модели

После конвертации модели вы можете использовать полученный файл:

- `model-unsloth.gguf`
- или квантованную версию: `model-unsloth-Q4_K_M.gguf`

Поддерживаемые окружения:

- `llama.cpp` — CLI-интерфейс для запуска модели.
- `Jan` — локальный UI для работы с LLM.
- `Open WebUI` — удобный веб-интерфейс, совместимый с множеством LLM-бэкендов.

Вы можете установить Jan и Open WebUI по следующим ссылкам:

- 📦 Jan: https://github.com/janhq/jan
- 🌐 Open WebUI: https://github.com/open-webui/open-webui

🎉 Всё готово!