Model Training & Evaluation

Page • 1 backlink • uni

Ботлнеки при обучении

Обычно есть какая-то база данных (S3 или что-то в том духе), с которой по сети (сетевой диск, но может быть и NVME, но это сейчас реже) данные передаются на разные машины

```python
def train():
    while True:
        batch = read()
        F(batch)
```

Тут может быть замедление как на чтении данных, так и на логике обучения

Идеально: вычитывать из сети с той скоростью, с которой идет обучение

- чтение можно делать параллельно в несколько потоков (одним выжать сеть не получится)
- пайплайн: асинхронное чтение и обучение: пока первый чанк на GPU, второй используется для обучения
- обычно `F(batch)` состоит из препроцессинга и применения модели, первое можно вынести наружу
- следить за балансом нагрузки CPU/GPU: CPU как занимается прокачкой, так и будет делать аугментации и общаться с RAM при обучении, обратно на прокачку мощностей может не хватить (нужно профилировать)

Как делать модели быстрее?

- fuse more – "фьюзинг" операции
- use tensor cores – лучше знать оборудование
- reduce overhead – pytorch не оптимизирован на 100%
- quantize – быстрее инфер, но хуже качество (намного ли?)
- use a custom kernel – сложно, но может дать большой выхлоп (см. flash attention)

Устройство видеокарты

- много SM (streamnig mutliprocessor): умеют в векторные инструкции
- SM может запросить блок непрерывной памяти (64-128 байт), под каждым SM есть ALU, которые проводят вычисления в этой памяти
- абстракция: grid потоков, состоящий из блоков, состоящий из warp-ов (в случае NVIDIA)
- у NVIDIA есть warp-ы, которые (ранбше был общий isp на warp, сейчас на каждый поток свой, но физически вычисления выполняются одновременно – есть маскирование операций и прочие эвристики)
- память: если тензор был в CPU, GPU HBM, GPU SRAM (L1-cache)
- еще есть L2-cache, которым можно управлять вручную
- мораль – FLOPS много, память не успевает, поэтому как правило, видеокарта упирается в чтение (даже внутри уровней памяти видеокарты, GPU сама обеспечивает синхронизацию)
- а что с этим делать? смотрим на выполняемые операции, Arithmetic intensity: пропорция вычислений на единиц доступа к памяти (отсюда понимаем, задача CPU-bound или memory-bound)

До pytorch 2.0 стандартной картиной был большой оверхед на запуск операций и round-trip-ы через глобальную память, как бороться:

- CUDA Graph (не зависит от pytorch): если оверхэд на выполнение первой операции большой, то набор последовательных операций можно выполнить без задержек (новый вариант, кэшировать память для выполнения операций на GPU) (?) – *запоминаем граф последовательных операций, последующие запуски делаем скопом в одной памяти*
- избавиться от хождений в глобальную память – `torch.compile(model, mode="reduce-overhead")` для фьюза операций

Tensor-Core: операции вида C = AX + B (B,C – FP16/FP32, A,X – FP16)

Quantization: больше FLOPS у операций над данными более легковесных типов (меняется Arithmetic intensity)

Ускорение можно получить за счет учета распределения вычислений по SM (например, размер эмбеддинга делится на какое-то странное число), чтобы полностью утилизировать блоки

Все это было при обучении на одной GPU

NUMA (non-unified memory access): в системах на несколько CPU и GPU может получиться, что CPU и GPU будут "в разных местах" и соединены

линком, который будет тормозить перегон данных (помогает NUMA affinity)

Распределенное обучение

- асинхронный и синхронный градиентный спуск: в первом случае независимое обновление, в последнем – ждем все апдейты и усредняем
- multi-gpu training: allreduce локальных градиентов и data parallelism
- у разных сетей разный баланс по памяти для градиентов, для параметров, дла активаций, а также по длине, с которой модель работает – балансируем между устройствами и делаем pipeline parallelism
- делать allreduce по сети: сеть – слабое место (со всех машин отправим на одну – перегрузим сеть, а "по кольцу" быстрее) (в NCCL – библиотеке для point-to-point коммуникаций – подобные алгоритмы есть, также для связи NVIDIA предлагает использовать NVLink)
- NCCL смотрит на топологию сети, делает поиск оптимальных колец и деревьев для агрегации и использует оптимальные kernel-ы

AutoML – поиск оптимального решения поверх инфраструктуры с больших компьютом

Оценка качества

- в рамках разработки
- метрики качества (в частности уже по всей системе)

Вначале оценивается относительно простой эвристики, потом относительно "человеческого" бейзлайна

Подходы к оценке

- бенчмарки
- оценка с помощью людей
- оценка готового продукта

Тесты

- perturbation tests: изменяем данные: вносим шумы, делаем перестановки
- при использовании аугментаций: нужна тестовая аугментация, которой нет в обучающем датасете
- invariance tests: замена признаков, которые не должны повлиять на ответ

- expectations tests: замена признаков, которые должны повлиять на ответ (сюда же compliance)
- slices: оценка на срезах

Парадокс Симпсона: если разбиение не имеет фиксированные доли, может оказаться, что одна модель на обоих срезах из двух работает лучше, а в совокупности работает хуже

Нужно выравнивать классификаторы: если модель говорит 0.6, то она должна ошибаться с шансом 40% при таком предикте

Иерархия метрик

Бизнес: EBITDA, GMV

Куча команд, каждая улучшает что-то свое по техническим метрикам

При этом хочется, чтобы команды могли делать выкатки независимо и при этом не просаживать бизнес

Критерий: зеленый прокрас на своем, серые метрики коллег

Приемочная метрика: можем ли мы катить и надеяться, что KPI не сломается (сам KPI, как правило, считается долго)

И приемочные метрики, и KPI бывают онлайн (удобно для SaaS-сервисов) и оффлайн

В частности, если приемочные метрики зеленые, а KPI красный – если проблемы с приемочными метриками

KPI не должен меняться от не наших изменений

Корреляцию KPI и приемочных метрик можно проверить с помощью ухудшающих экспериментов