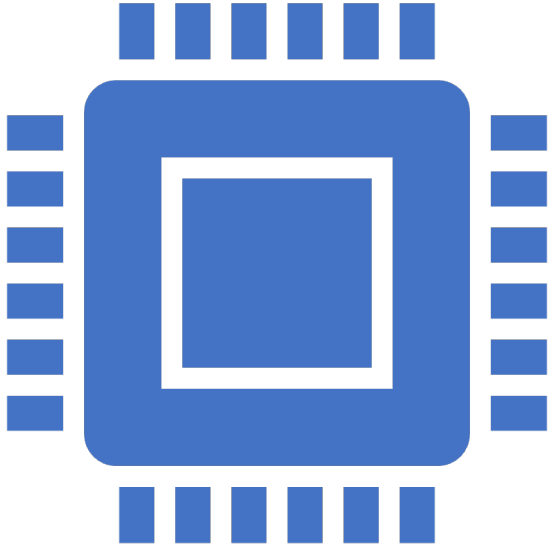


CuML и не только

О чем поговорим?

- CuML и Scikit-learn (CPU, GPU, Multi-GPU)
- Catboost (CPU, GPU, Multi-GPU)
- Pytorch (CPU, GPU, Multi-GPU)

CuML

CuML – scikit-learn на ~~максималках~~ GPU. Реализовано много различных методов, использовать которые очень удобно.
Опять синтаксис scikit-learn и cuml очень похожи между собой, что не может не радовать, только используйте.

Не все возможности sklearn есть, но все же, очень полезная библиотека и ее функционала часто будет достаточно. Ну а если нет, то в курсе еще поговорим о том, где взять недостающее)

Найдите ниже 10 отличий:

```
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

from cuml.ensemble import RandomForestClassifier as cuRFC
from cuml.datasets.classification import make_classification as make_classification_cu
from cuml.preprocessing.model_selection import train_test_split as train_test_split_cu
import cudf
from cuml.metrics import accuracy_score as accuracy_score_cu
```

Что есть в CuML

- Кластеризация: K-Means, DBSCAN, HDBSCAN
- Снижение размерности: PCA, Incremental PCA, tSVD, UMAP, Random Projection, tSNE
- Линейные модели: Linear Regression (L1, L2, ElasticNet), Logistic Regression, Naïve Bayess
- Нелинейные модели: Random Forest, KNN, SVM*
- Временные ряды: Holt-Winters Exponential Smoothing, ARIMA
- SHAP Values

https://github.com/rapidsai/cuml

Немного бенчмарков

На чем тестировались?
32 ядра, 256 ГБ оперативы, 4 Tesla V100 16ГБ

Генерировались датасеты для классификации, 2 класса, 20 фичей. Ниже 200к наблюдений, но тестируем на разных объемах). Обучали RandomForest.

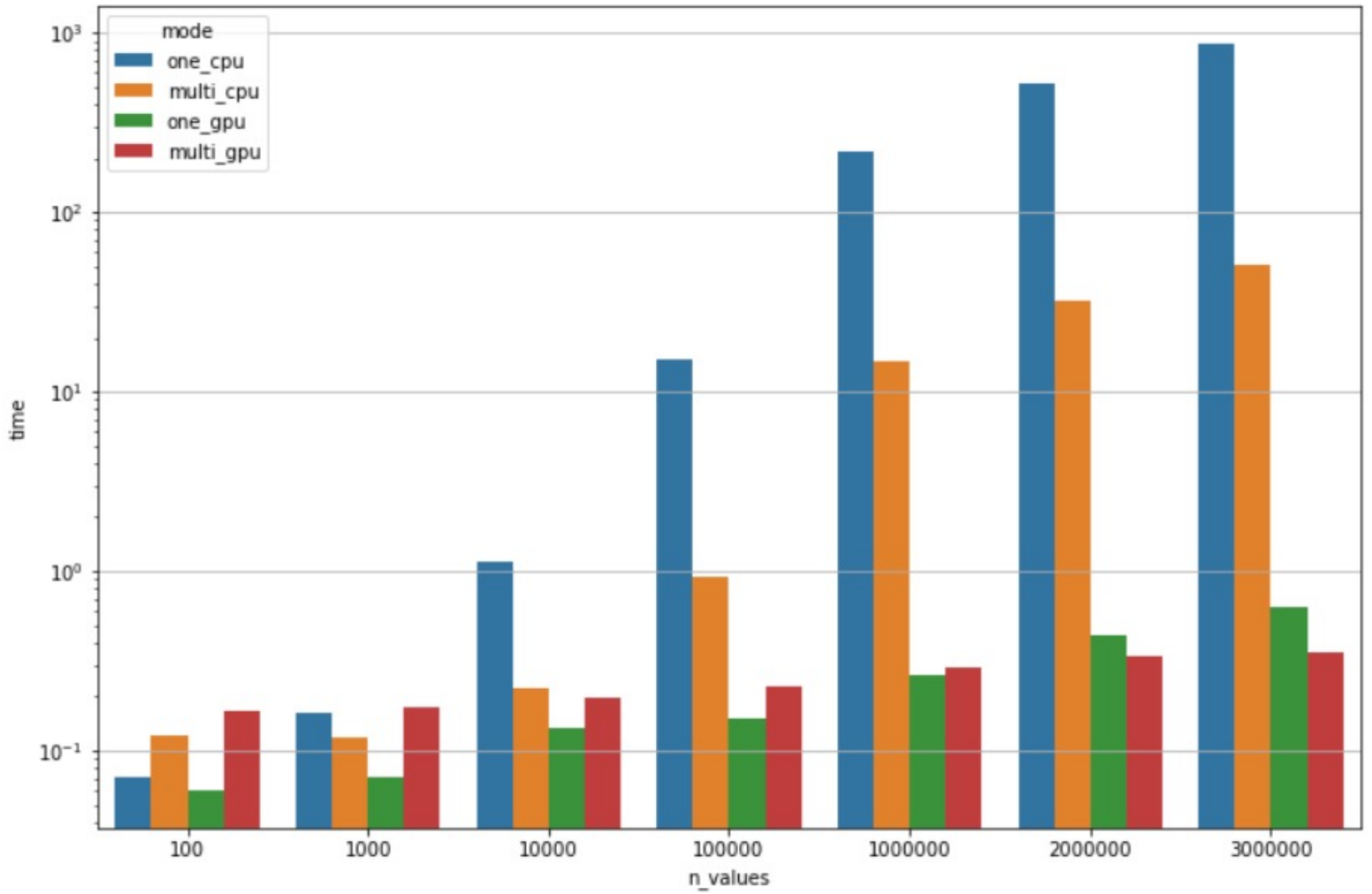
```
%%time
X, y = make_classification(n_classes=n_classes, n_features=n_features, n_samples=n_samples, random_state=0, class_sep=0.7)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
CPU times: user 476 ms, sys: 1.66 s, total: 2.13 s
Wall time: 302 ms
```

```
%%time
X_cu, y_cu = make_classification_cu(n_classes=n_classes, n_features=n_features, n_samples=n_samples, random_state=0, class_sep=0.
X_train_cu, X_test_cu, y_train_cu, y_test_cu = train_test_split_cu(X_cu, y_cu, random_state=0)
```

```
CPU times: user 20 ms, sys: 8 ms, total: 28 ms
Wall time: 25.6 ms
```

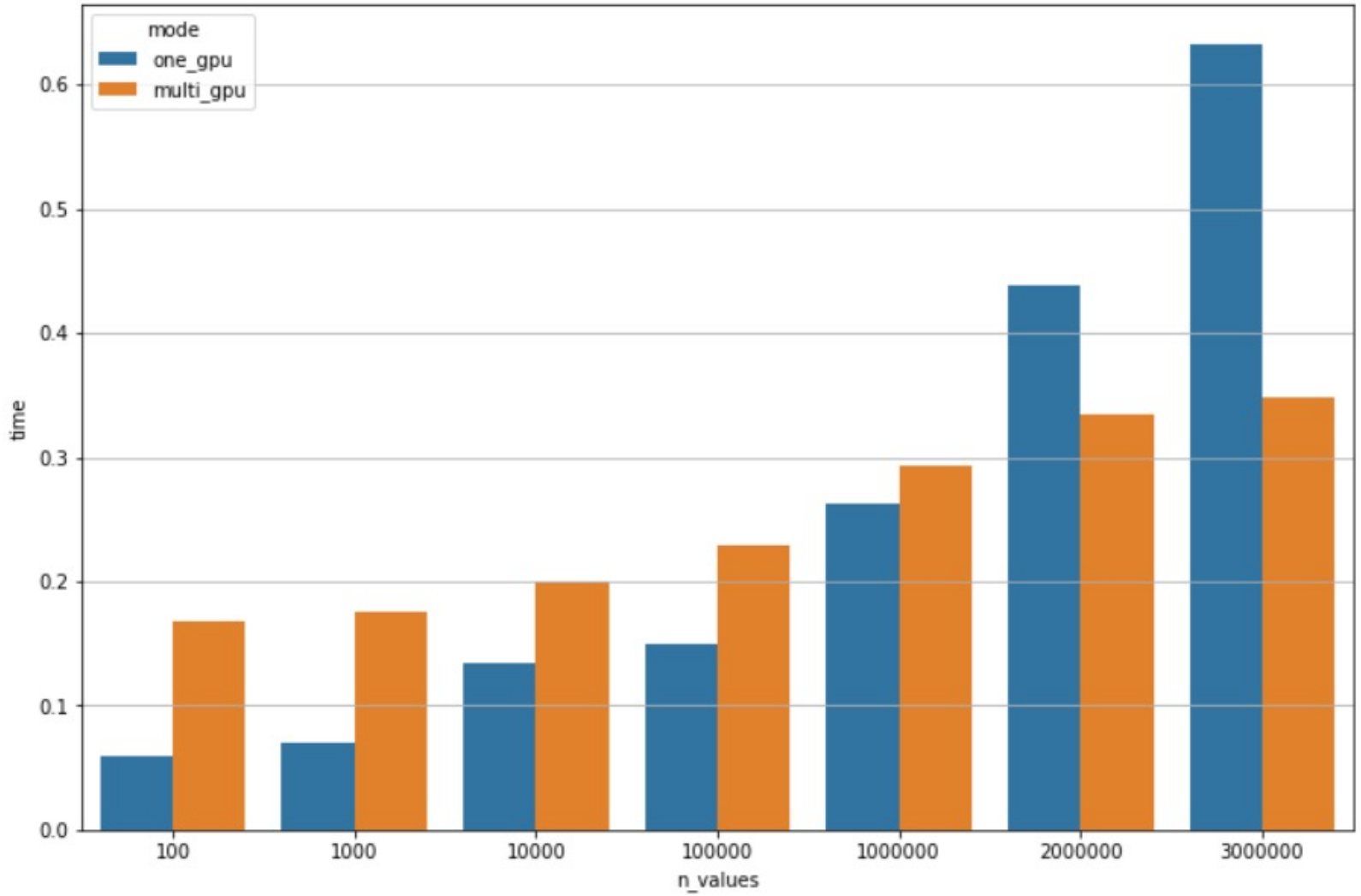
Немного бенчмарков



По времени шкала логарифмическая.

Какие выводы? Если датасет не очень большой, то 1 GPU сойдет за глаза и будет очень шустрым.
Использовать 4 GPU стало целесообразно при наличии от 2 кк наблюдений.

Немного бенчмарков



Тут уже обычная шкала по времени. Для сравнения 1 GPU vs 4 GPU.

Немного бенчмарков

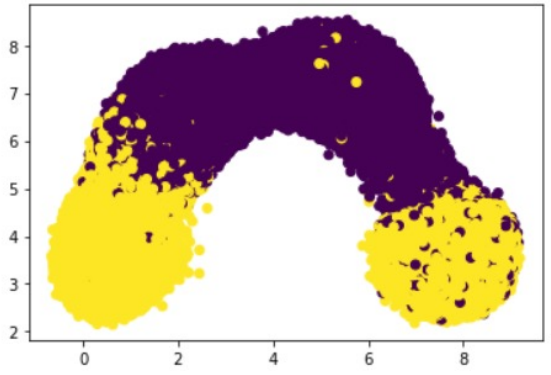
Все мы хотим посмотреть на пространство размерности больше 3. И все мы знаем алгоритм UMAP, который считается довольно точным и намного быстрее, чем tSNE. Итак, 20 фичей, 200к наблюдений. UMAP из CuML или из одноименной библиотеки? Используем 1 GPU=)

```
%%time
umap_cpu = umap.UMAP(n_neighbors=30, n_components=2).fit(X)
umap_cpu_embeddings = umap_cpu.transform(X)
```

```
CPU times: user 45min 5s, sys: 11min 53s, total: 56min 58s
Wall time: 5min 22s
```

```
a, b = zip(*umap_cpu_embeddings.tolist())
plt.scatter(a, b, c=y)
```

```
<matplotlib.collections.PathCollection at 0x7f1134550910>
```

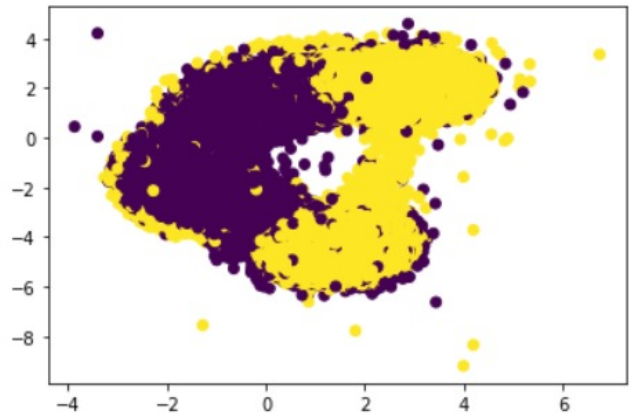



```
%%time
umap_gpu = cuUMAP(n_neighbors=30, n_components=2).fit(X)
umap_gpu_embeddings = umap_gpu.transform(X)
```

```
CPU times: user 1.04 s, sys: 960 ms, total: 2 s
Wall time: 1.99 s
```

```
a, b = zip(*umap_gpu_embeddings.tolist())
plt.scatter(a, b, c=y)
```

```
<matplotlib.collections.PathCollection at 0x7f10111e8b90>
```



Выводы

- Использовать GPU классно и просто
- Несколько GPU посложнее, но тоже можно
- Если вам хватает 1 GPU, то больше и не надо, берите несколько только тогда, когда иначе не хватает памяти, либо GPU больше никому не нужна

P.S. Лучше использовать float32, int32. На 64 будет ругаться)

CatBoost

Catboost

Как говорят разработчики, что при создании алгоритма ни одно животное не пострадало.
Очень хорошая библиотека, это и так известно.

Отличная новость: из коробки работает даже с Multi-GPU. Но надо помнить, кстати, что обучение на GPU является недетерминированным.

Инференс также на CPU.

Дискретизация переменных – единственный способ обучения таких моделей на GPU. По умолчанию дискретизация на 128 категорий, но можно менять.

https://www.youtube.com/watch?v=kRjC7nzSHWk – тут можно послушать самого разработчика Catboost.

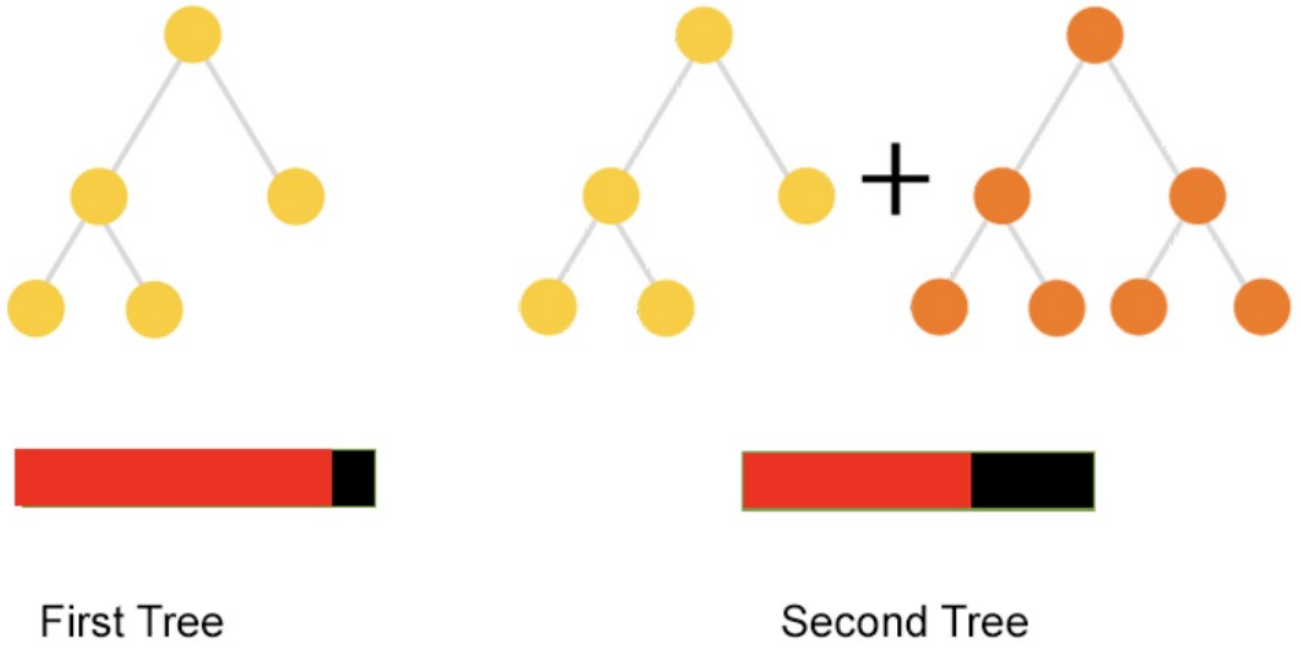
Catboost – интересные моменты

1) Нельзя обучать по батчам, а значит надо как-то уметь работать с крупными датасетами. Catboost оптимально использует память и из примерно на 7ГБ данных тратит 0.5Гб памяти.
2) Написан на C++
3) По словам разработчиков(разработчика) инференс в 40 раз быстрее XGBoost и LGBM
4) Не нужно самому делать OHE! Это очень плохо и теряется возможность оптимизации внутри модели, не делайте так.
5) gpu_eat_features_storage – если много категориальных переменных и памяти на GPU не хватает, то жертвуем скоростью и переносим хранилище категорий на CPU
6) Если обучаете на GPU, сохраняете модель и видите, что объем файла измеряется в сотнях Мб или даже в Гб..бывает, алгоритм составления комбинаций из категориальных переменных на GPU не оптимизирован с точки зрениях штрафов за большое количество комбинаций (на CPU оптимизировано). Выход: max_ctr_complexity = 1 или 2 вам помогут, по точности ударить не должно (наверное).
7) gpu_ram_part – ограничит % занимаемой памяти на GPU, по умолчанию берет 95%.
8) Используйте Pool. Да, Catboost всеяден, внутри все равно переведет в Pool, но это же лишние проверки)
9) Чем больше фичей, тем быстрее Catboost.
10) В некоторых задачах, когда LightGBM показывал смещение прогноза, CatBoost хорошо справлялся.

Отсылка к уже известному

Все мы хорошо знаем, как работают и учатся бустинги.

Учим итеративно, оптимизируя любую непрерывную целевую функцию.



https://developer.nvidia.com/blog/catboost-fast-gradient-boosting-decision-trees/

Как учим

1) Считаем градиенты оптимизируемой функции потерь по каждому наблюдению

2) Учим дерево решений, которое предсказывает градиенты функции потерь (часто еще называют корректирующим вектором)

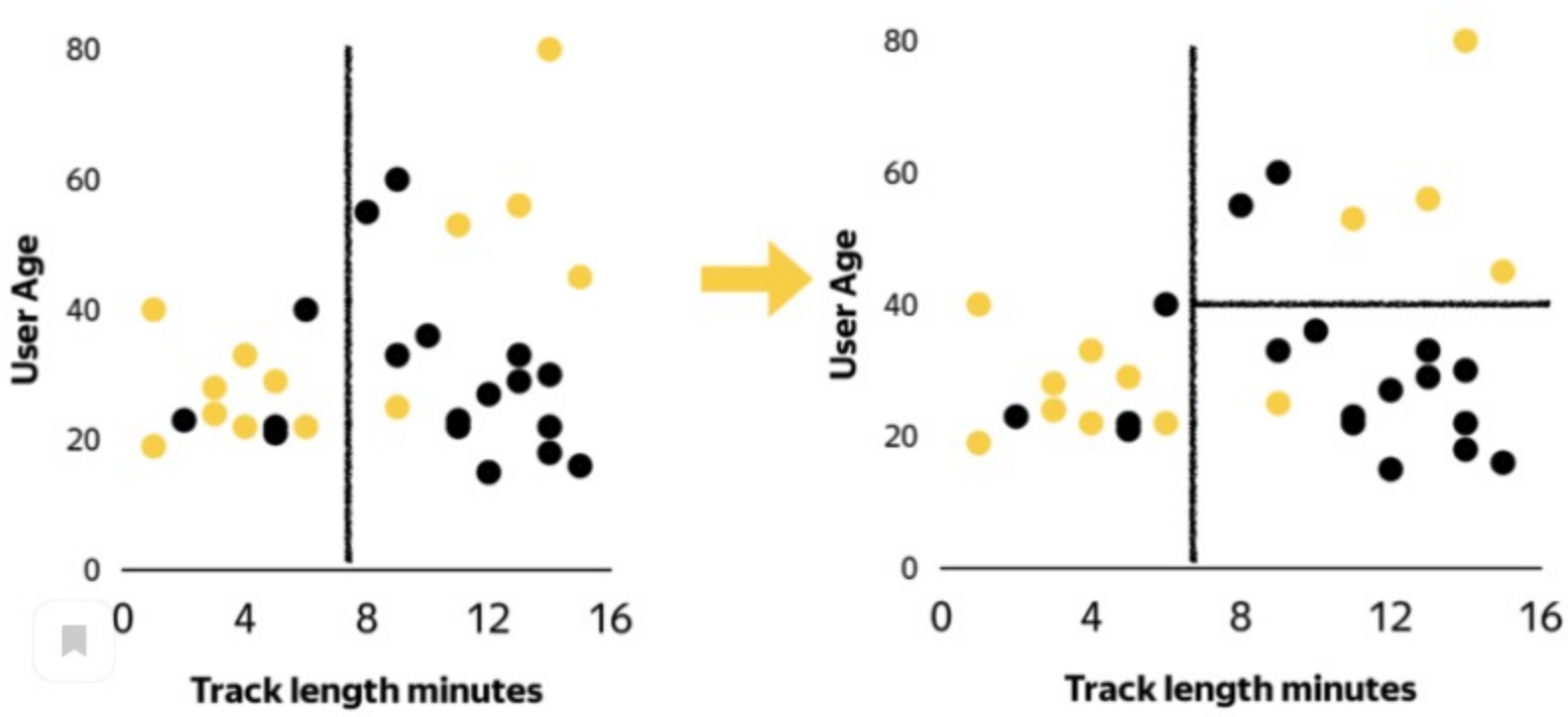
В чем сложность

1) Градиенты считать легко, тут на CPU и GPU проблем нет

2) А вот поиск наилучшего дерева решений является вычислительно очень затратной операцией, занимающей почти все время работы алгоритма

Да и как найти тот уровень отсечения, который будет наилучшим для листа?

Зачем нужна дискретизация



Мы определяем то разделение, которое наибольшим образом оптимизирует нашу функцию потерь.

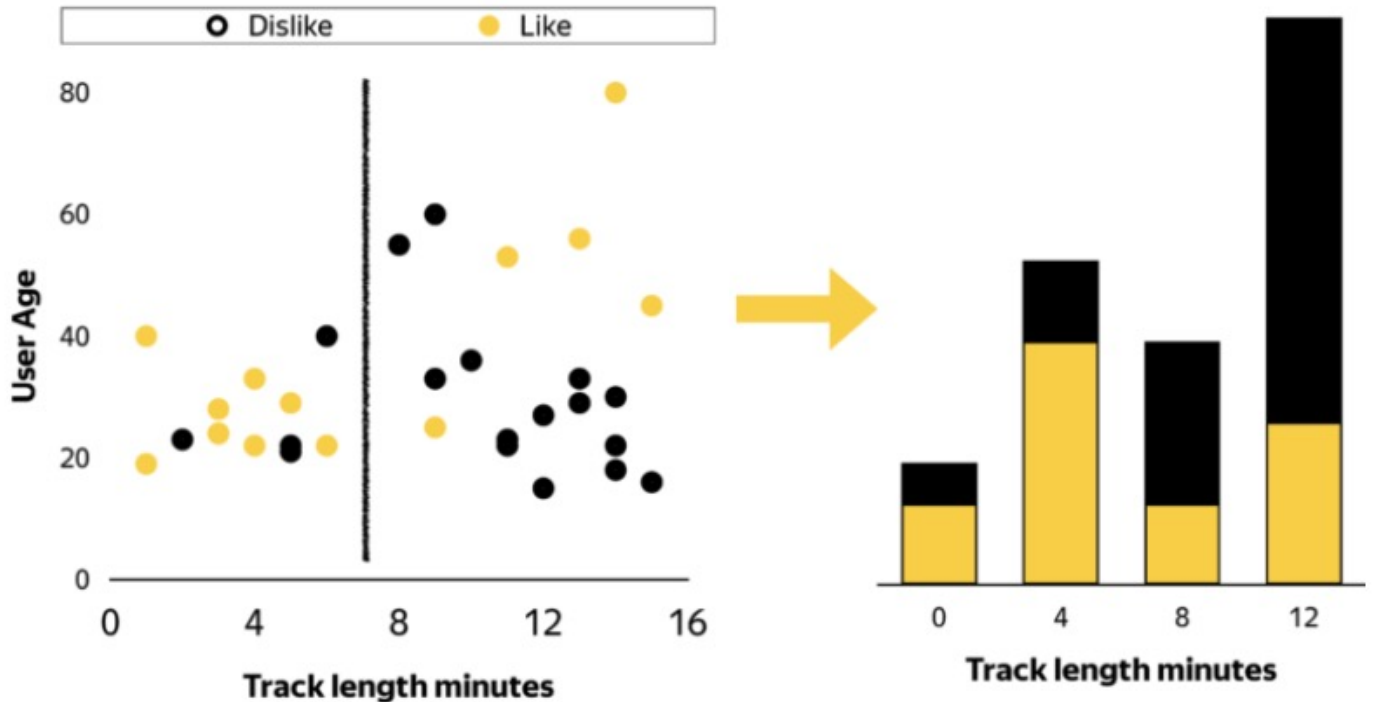
Фиксируем разделение.

Исходя из него ищем следующее.

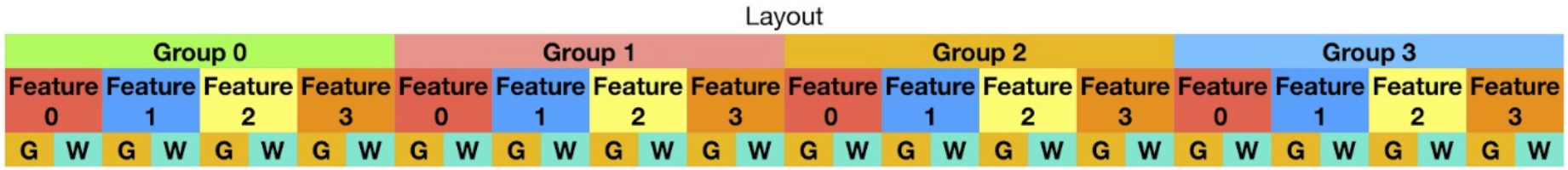
А если переменная непрерывная? Верно, дискретизация (аналог квантизации для нейронок)

https://developer.nvidia.com/blog/catboost-fast-gradient-boosting-decision-trees/

Зачем нужна дискретизация



Алгорит жадный, анализиурем гистограммы по всем фичам. Для классификации анализируется отношение классов.
Но фичи также могут объединяться в группы по 4, чтобы экономить память.
Для обучения дерева считается сумма весов и сумма градиентов.
Все это позволяет оптимально реализовать алгоритм обучения.



https://developer.nvidia.com/blog/catboost-fast-gradient-boosting-decision-trees/

Pytorch

Стратегии обучения нейронных сетей

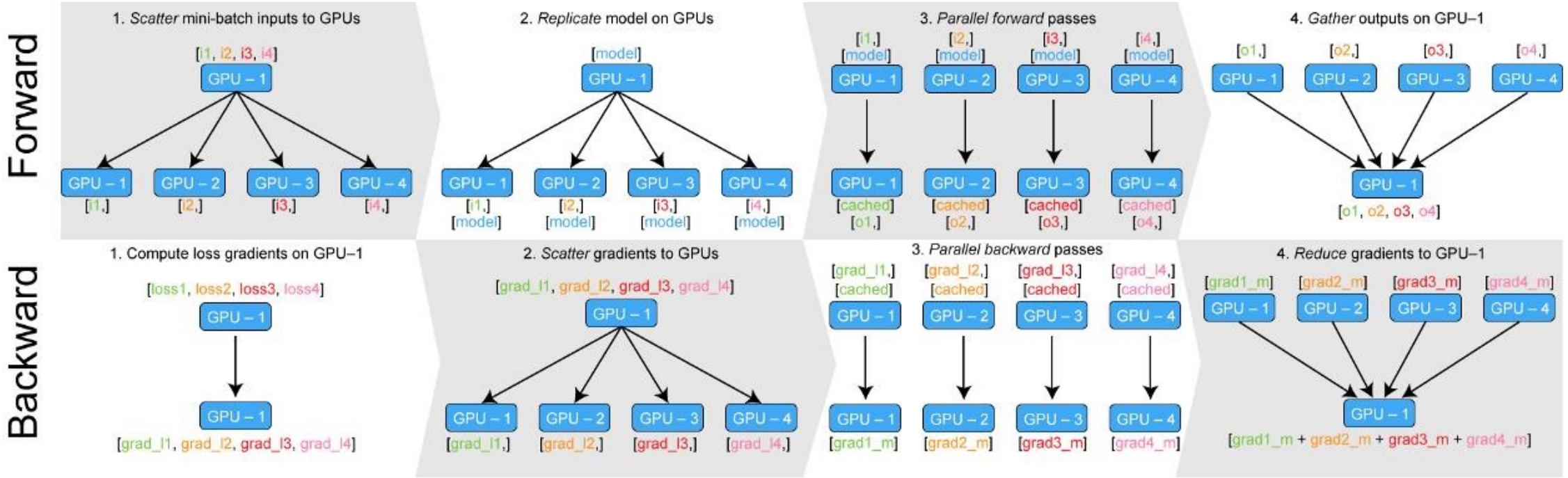
Обучение на CPU и одном GPU проходят стандартно, отличия в том, что веса модели, расчеты и батчи данных находятся на соответствующих устройствах.

- Подготовили batch данных;
- Сделали forward pass;
- Вычислили loss и градиенты;
- Сделали backward pass;
- Обновили веса.

А как это сделать распределенно?

- DataParallel;
- DistributedDataParallel.

DataParallel

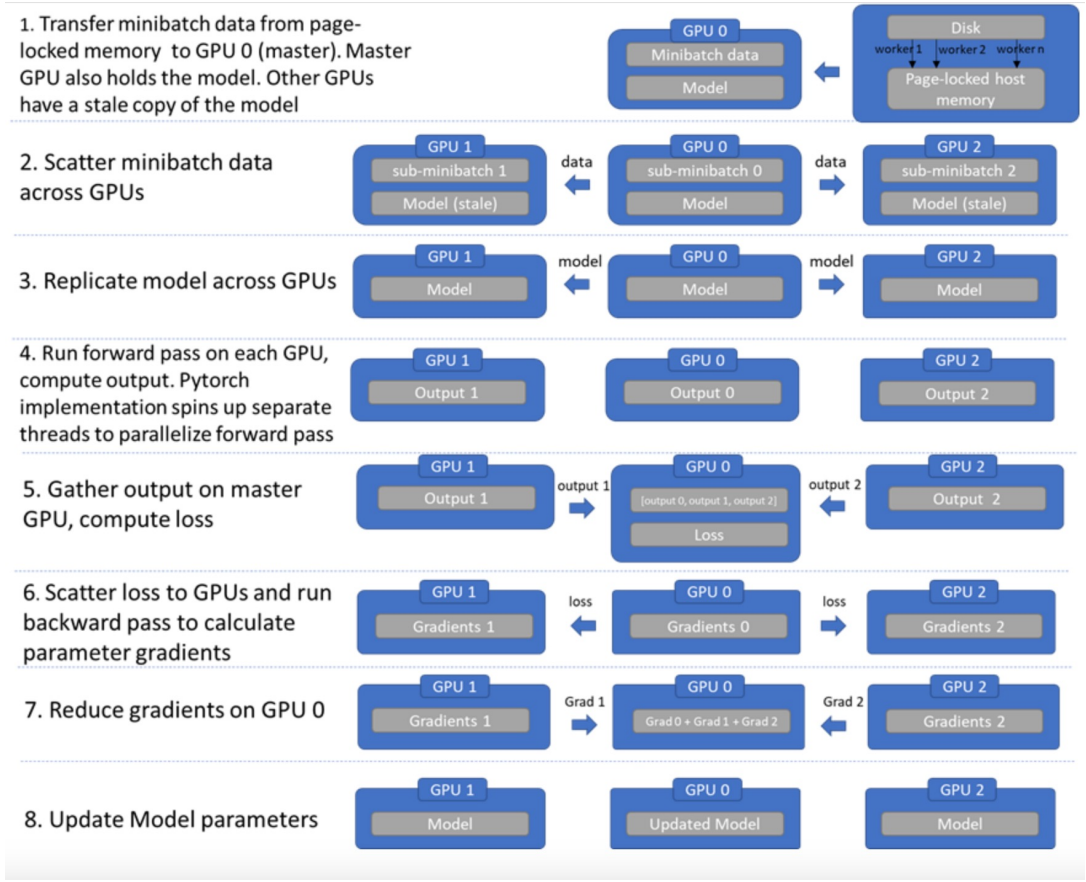


Главная особенность – все GPU могут находиться только на одной ноде.

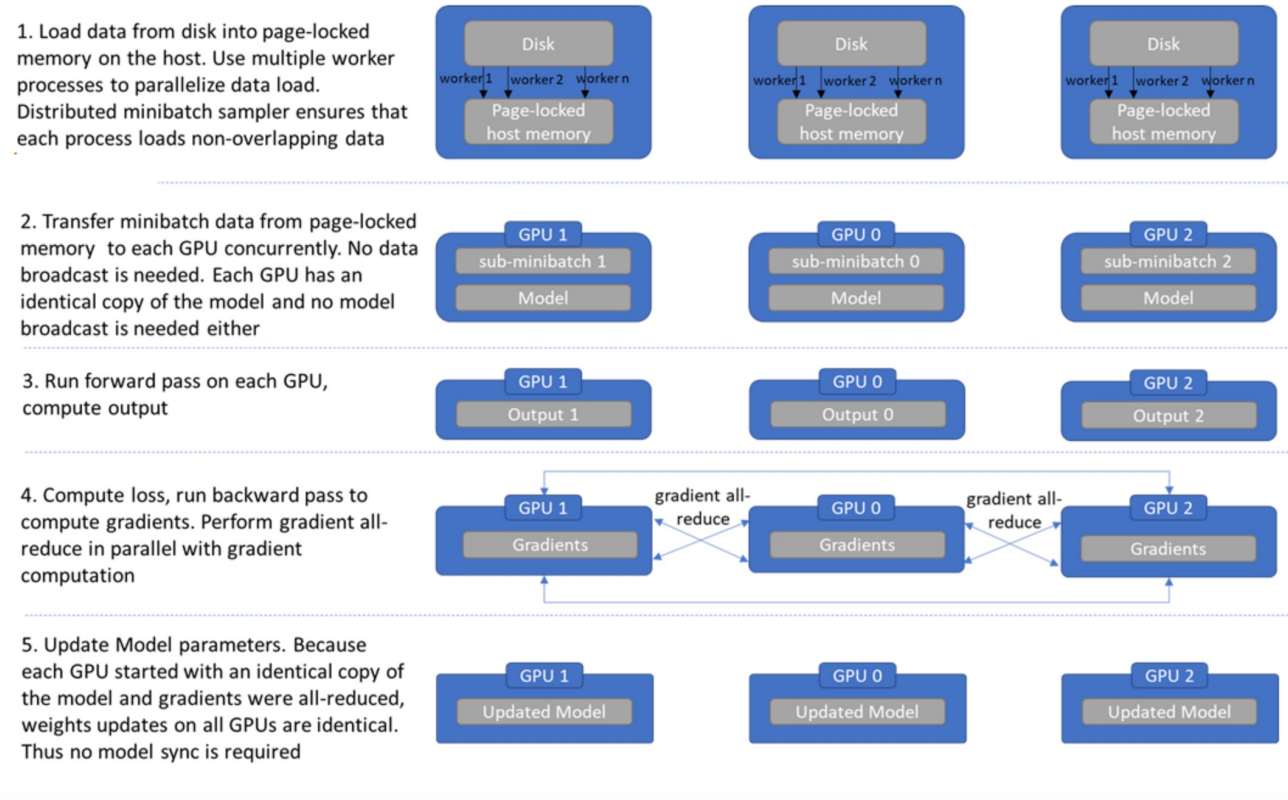
А также много копирований, все аккумулируется, веса обновляются на master gpu.

DistributedDataParallel

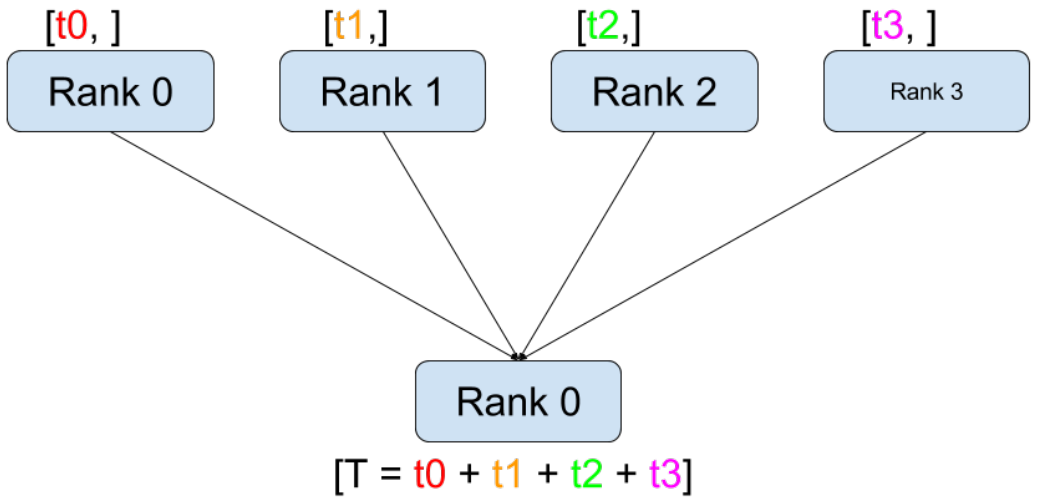
DataParallel



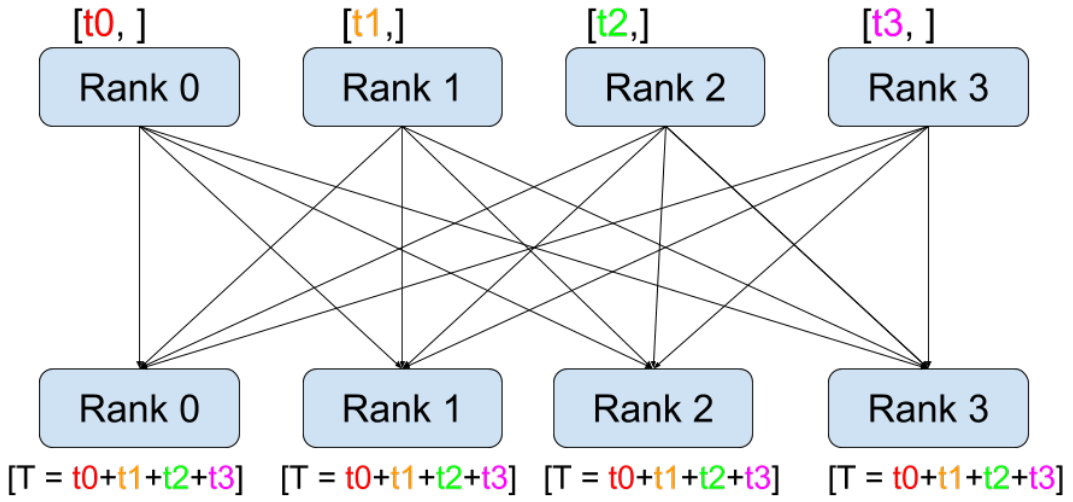
DistributedDataParallel



Reduce/All-Reduce



Reduce



All-Reduce