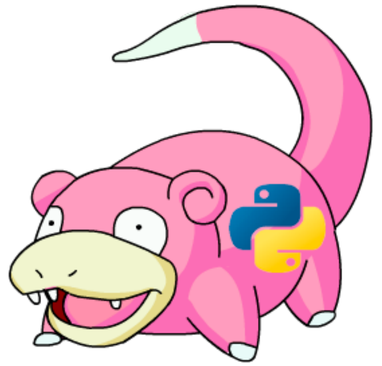
# Лекция №3.

## Часть 1.

## Профилирование кода.

01.12.2020



# Профилирование кода

**Профилирование -** процесс сбора статистики выполнения кода программы с целью дальнейшей оптимизации

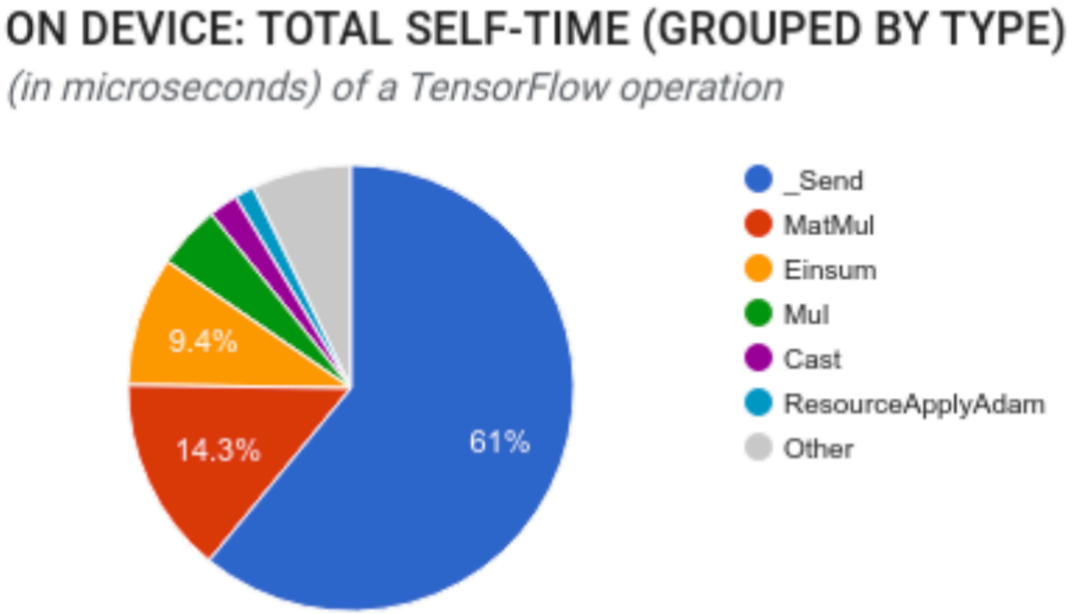
**Какую статистику обычно собирают?**

- Время выполнения отдельных строк
- Время выполнения отдельных функций
- % использования  CPU и памяти CPU
- Дерево вызовов функций
- Обнаружение **«hot-spots»***

*__hot-spots__ - участки программы, на который приходится наибольшая часть времени выполнения программы и/или загрузки процессора

# Зачем это надо?

**При профилировании кода** вы можете извлечь важные характеристики такие как: время выполнения и загрузку процессора. Соответственно после профилирования программист может найти наиболее медленные участки кода и провести их ***оптимизацию***

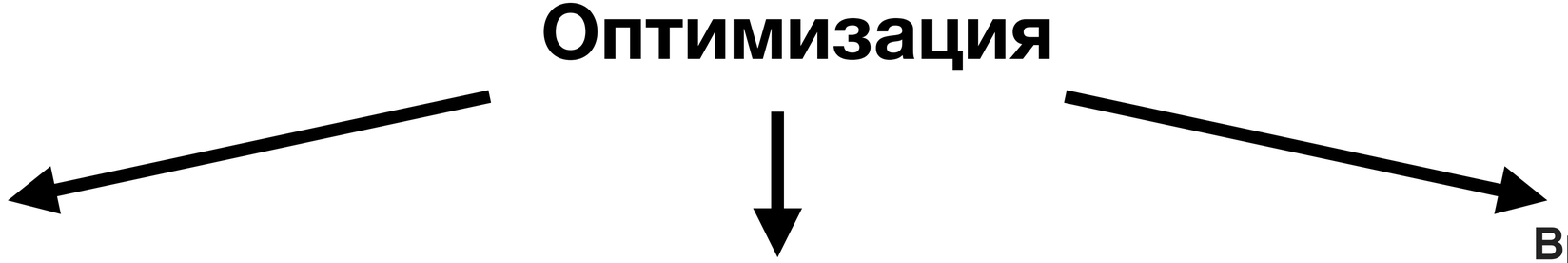



- Пример статистики собранной во время обучения нейронной сети. Профилирование с использованием методов из библиотеки Tensorflow (библиотека для глубокого обучения)

# Оптимизация

Википедия подсказывает нам, что:

**Оптимизация** — это модификация системы для улучшения её эффективности. Оптимизация стоит на трех «китах» — естественность, производительность, затраченное время. Давайте разберемся подробнее, что они означают:




**Естественность.**
Код должен быть аккуратным, модульным и легко читабельным. Каждый модуль должен естественно встраиваться в программу

**Производительность**.
В результате оптимизации вы должны получить прирост производительности программного продукта. Например ускорение на 20-30% в сравнение с исходным вариантом.

**Время**.
Оптимизация и последующая отладка должны занимать небольшой период времени. Оптимальными считаются сроки, не превышающие 10 – 15 % времени, затраченного на написание самого программного продукта. Иначе это будет нерентабельно.

# Оптимизация не всегда необходима!

**КАК ТАК?!**

**Перед проведением оптимизации подумайте - а надо ли это вообще? И вот почему:**

1. На оптимизацию тратится время.
2. Скорее всего код станет непонятнее.
3. Не все оптимизации полезны. Оптимизируя по времени, вы можете увеличить расход памяти.

Дональд Кнут* сделал следующее заявление об оптимизации:

*«Мы должны забыть о небольшой эффективности, скажем, примерно в 97% случаев: преждевременная оптимизация - это корень всего зла, но мы не должны упускать наши возможности в этих критических 3%*

*Дональд Кнут - американский ученый в области информатики, профессор Стэнфордского университета

# Подходы к профилированию кода

- **Визуально (метод «пристального взгляда»)**
- **Ручное профилирование**
- **Профилирование с использованием готовых инструментов**

# Визуальное профилирование

**Метод пристального взгляда** - один из самых неэффективных. Подходит разве что профессионалам с большим опытом.
При использовании данного метода, Вы просто смотрите построчно код и интуитивно пытаетесь понять, где и какая из строк кода использует наибольшее количество вычислительных ресурсов.

**Достоинства метода:**
**+** нет необходимости в использовании дополнительных библиотек
**-** Сложно оценить трудозатраты и результат после оптимизации

# Ручное профилирование

**Суть метода -** измерение времени исполнения того или иного участка кода «вручную». Под термином «вручную» подразумевается, например, сохранение в переменную количество времени, затраченного на выполнение спорного участка, затем его оптимизации и новая проверка - ускорила ли оптимизация время исполнения кода? Для достоверного результата следует повторить операцию N раз и взять среднее значение.

**Достоинства и недостатки этого метода:**

**+** очень простое применение
**+** зачастую не подходит для промышленного программирования
**–** вставка «чужеродного» кода в проект
**–** использование возможно не всегда
**–** никакой информации о программе, кроме времени выполнения анализируемого участка
**–** анализ результатов может быть затруднительным (следствие предыдущего пункта)

# Пример

Для ручного профилирования можно использовать «магические функции» **%timeit, %%timeit**

**%timeit** -это магическая функция ipython , которая может использоваться для определения времени выполнения определенного фрагмента кода (одного оператора выполнения или одного метода)

Чтобы использовать его, например, если мы хотим узнать, является ли использование `xrange` более быстрым , чем использование `range`, вы можете просто сделать это:

```
In [1]: %timeit for _ in range(1000): True
10000 loops, best of 3: 37.8 µs per loop

In [2]: %timeit for _ in xrange(1000): True
10000 loops, best of 3: 29.6 µs per loop
```

# Профилирование с помощью инструментов

Профилирование **с помощью инструментов** помогает, когда мы (по тем или иным причинам) не знаем, отчего программа работает не так, как следует, либо когда нам лень использовать ручное профилирование и анализировать его результаты.

Встроенные в Python инструменты для профилирования:

- **cProfile** — относительно новый (с версии 2.5) модуль, написанный на C и оттого быстрый
- **profile** — нативная реализация профайлера (написан на чистом питоне), медленный, и поэтому не рекомендуется к использованию
- **hotshot** — экспериментальный модуль на си, очень быстрый, но больше не поддерживается и в любой момент может быть удалён из стандартных библиотек

# Пример

## cProfiler

Позволяет собрать аналитику по вызовам функций:

- **ncals** - кол-во вызовов. Если в этой колонке стоит два числа 3/1, то это значит, что функция рекурсивная. Первое число - общее кол-во вызовов, второе - кол-во нерекурсивных вызовов.
- **totime** - время исполнения функции без учета времени вызова подфункций
- **cumtime** - время исполнения функции с учетом времени вызова подфункций

Напишем функцию:

```
[ ]  def fib(n):
         if n == 0:
             return 1
         if n == 1:
             return 1
         return fib(n - 1) + fib(n - 2)
```

Импортируем модуль **cProfile** и запустим метод **run:**

```
[ ]  import cProfile
     cProfile.run('fib(30)', sort='tottime')
```

```
         2692540 function calls (4 primitive calls) in 0.668 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
2692537/1    0.668    0.000    0.668    0.668 <ipython-input-37-99a0d869b1b2>:1(fib)
        1    0.000    0.000    0.668    0.668 {built-in method builtins.exec}
        1    0.000    0.000    0.668    0.668 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

# Пример

**Tensorflow profiler**
Отслеживание производительности моделей из библиотеки Tensorflow (библиотека для глубокого обучения)

Summary of input-pipeline analysis

*Your program is HIGHLY input-bound because 81.4% of the total step time sampled is waiting for input. Therefore, you should first focus on reducing the input time.*

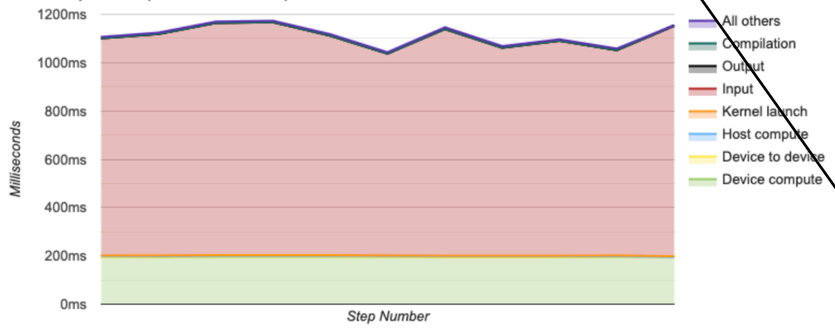
**Recommendation for next step:**
*Look at Section 3 for the breakdown of input time on the host.*

Device-side analysis details

Device step time

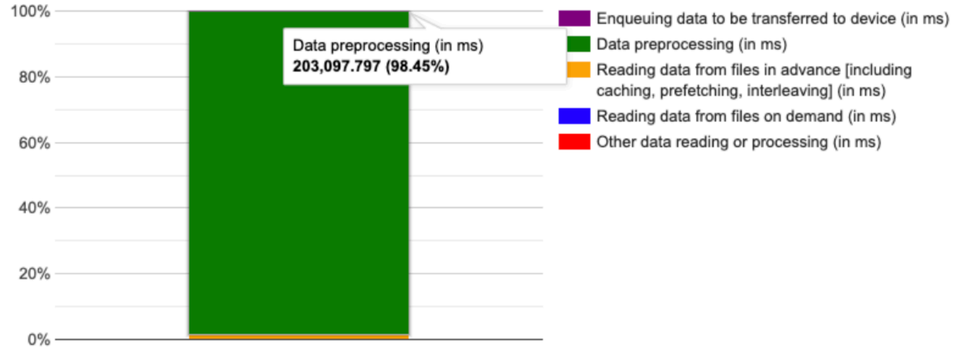
**Device step-time statistics**
Average: 1114.5 ms (σ = 44.8 ms)
Range: 1043.4 - 1173.6 ms

**Step Time (in milliseconds)**




Host-side analysis details

**Breakdown of input processing time on the host**



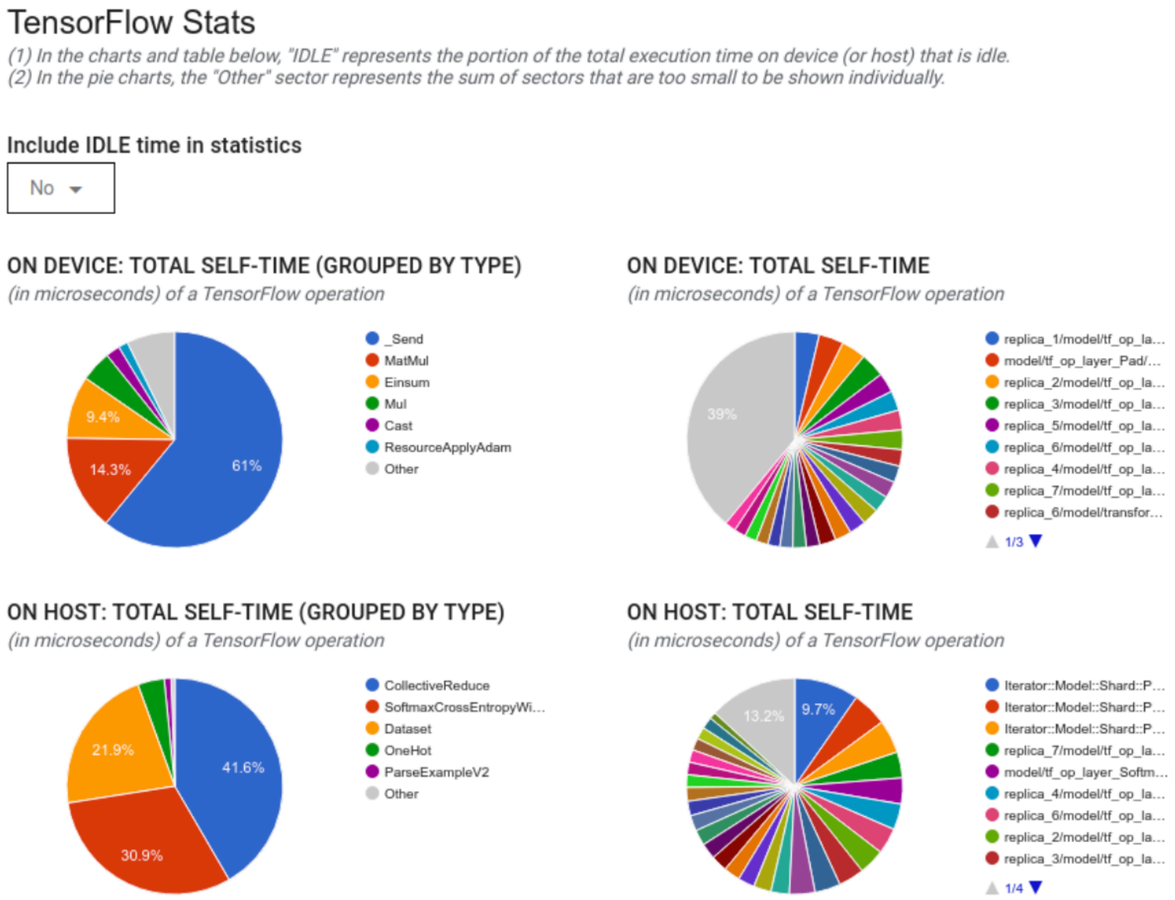

**What can be done to reduce above components of the host input time:**
Enqueuing data: you may want to combine small input data chunks into fewer but larger chunks.
Data preprocessing: you may increase num_parallel_calls in Dataset map() or preprocess the data OFFLINE.
Reading data from files in advance: you may tune parameters in the following tf.data API (prefetch size, interleave cycle_length, reader buffer_size)
Reading data from files on demand: you should read data IN ADVANCE using the following tf.data API (prefetch, interleave, reader buffer)
Other data reading or processing: you may consider using the tf.data API (if you are not using it now)

**Автоматические рекомендации!**

# Пример

**Tensorflow profiler**

Отслеживание производительности моделей из библиотеки Tensorflow (библиотека для глубокого обучения)




На верхней панели отображается до четырех круговых диаграмм:

1. Распределение времени самостоятельного выполнения каждой операции на хосте
2. Распределение времени самостоятельного выполнения каждого типа операции на хосте
3. Распределение времени самостоятельного выполнения каждой операции на устройстве
4. Распределение времени самостоятельного выполнения каждого типа операции на устройств

# Вывод

- Прежде чем переходить к оптимизации - оцените ее необходимость и уместность в конкретной задаче

- Существует множество способов профилирования «с инструментами», в том числе уже встроенных в Python: **cProfile, profile, hotspot**

- Каким бы хорошим не был Python-программист, у него практически всегда имеются проблемы со скоростью выполнения программы или же используемой памятью

- Оптимизировать можно практически бесконечно. Ищите «золотую середину»

**Ссылки на полезные статьи о профилировании:**

**О профилировании кода прямо в Jupyter Notebook**

**Ускорение кода Python средствами самого языка**