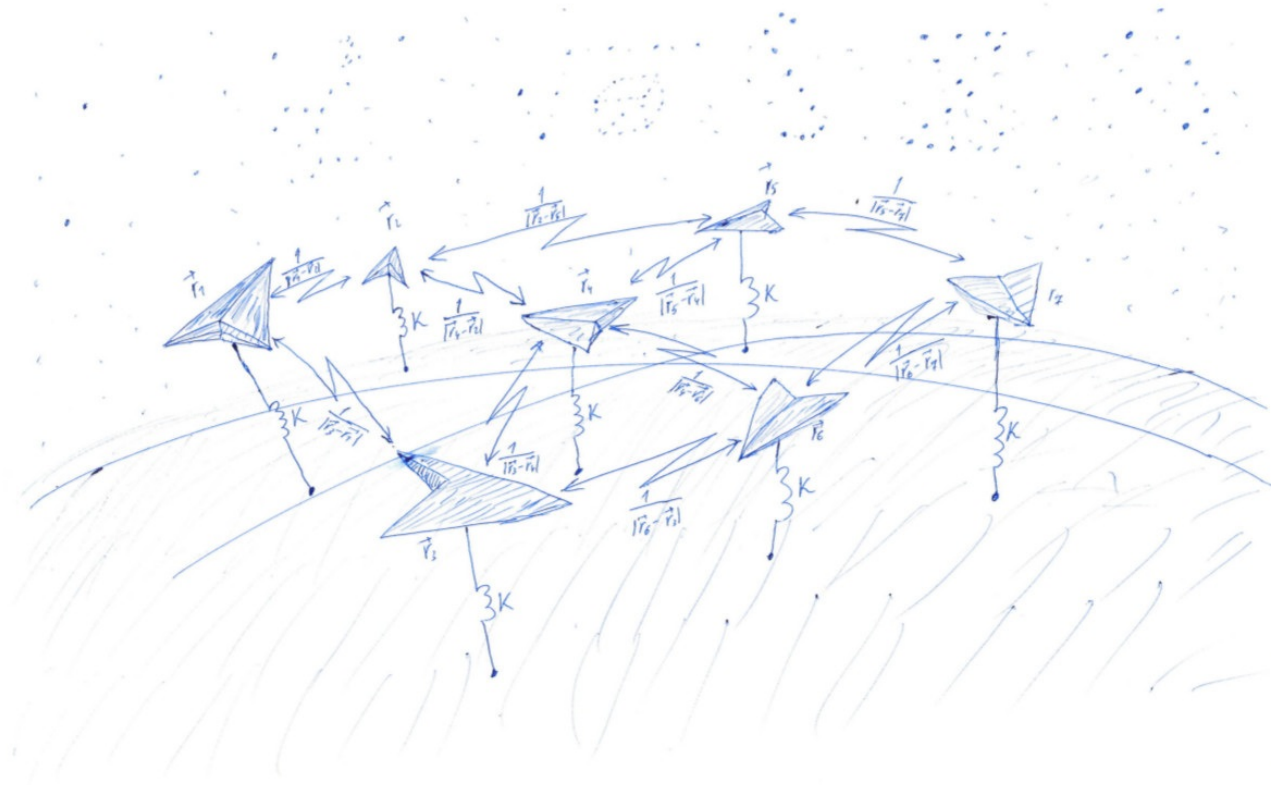
Равномерное распределение точек на сфере

Часть 1. Постановка задачи. Расчёт функционала.

- Аудитория: начальный уровень deep learning
- В результате прочтения:
  - опыт применения векторизированных матричных вычислений библиотек numpy, pytorch, tensorflow для решения реальных задач
  - более полное понимание разных методов минимизации функции многих переменных, их параметров
  - опыт визуализации данных с помощью PovRay и vapory



Откуда задача

> Если в вашей компании есть человек, целый день смотрящий видосики и чатящийся на форумах - это датасайнтист. И у него ОБУЧАЕТСЯ.

Чаще всего процесс обучения моделей machine learning (ML) - это подбор параметров, минимизирующих некоторый функционал. При этом контролировать этот процесс можно, глядя на одно единственное число. Давайте рассмотрим задачу, в которой мы будем минимизировать довольно сложный функционал, но при этом в любой момент времени

сможем видеть, что у нас получилось. Будем искать равномерное распределения по сфере заданного количества n точек. Такое распределение бывает нужно акустику для того, чтобы понять, в каком направлении запустить волну в кристалле. Связисту - чтобы узнать как расположить на орбите спутники для достижения наилучшего качества связи. Метеорологу для размещения станций слежения за погодой.

Для некоторых n задача решается легко. Например, если $n = 8$, то мы можем взять куб и его вершины будут являться ответом к задаче. Также нам повезёт, если n будет равно количеству вершин икосаэдра, додекаэдра или другого платонова тела. В противном случае задача не столь проста.

Для достаточно большого количества точек есть формула с эмпирически подобранными коэффициентами, ещё один вариант - здесь. Но есть и более универсальное, хотя и более сложное решение, которому посвящена данная статья.

Решим задачу, очень похожую на задачу Томсона (wiki). Разбросаем n точек случайно, а потом заставим их притягиваться к какой-то поверхности, например, к сфере и отталкиваться друг от друга. Притяжение и отталкивание определяются функцией - потенциалом. При минимальном значении потенциальной функции точки будут лежать на сфере наиболее равномерно.

Считаем потенциал

Случайно разбрасываем точки в кубе со стороной 1. Пусть у нас будет 2000 точек, а упругое взаимодействие в 1000 раз более значимо, чем электростатическое:

```
import numpy as np

n = 2000
k = 1000
x = np.random.rand(n, 3)
```

Сила упругого взаимодействия определяется формулой $F = -k_1 x$, сила электростатического - $F = k_2/r^2$. Потенциал упругого взаимодействия двух точек $u_1 = k_1 r^2/2$, электростатического - $u_2 = k_2/r$. Полный потенциал складывается из электростатического взаимодействия всех пар точек и упругого взаимодействия каждой точки с поверхностью сферы:

$$U(x_1, \ldots, x_n) = \sum_{p=1}^{n-1} \sum_{q=p+1}^{n} \frac{1}{|\vec{x_p} - \vec{x_q}|} + k \cdot \sum_{p=1}^{n} (1 - |\vec{x_p}|)^2 \to \min$$

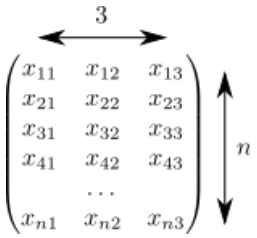
В принципе, можно посчитать значение потенциала по этой формуле:

```
L_for = 0
L_for_inv = 0
L_for_sq = 0
for p in range(n):
    p_distance = 0
    for i in range(3):
        p_distance += x[p, i]**2
    p_distance = math.sqrt(p_distance)
    L_for_sq += k * (1 - p_distance)**2  # квадрат расстояния от поверхности сферы, умноженный на
 константу упругости
```

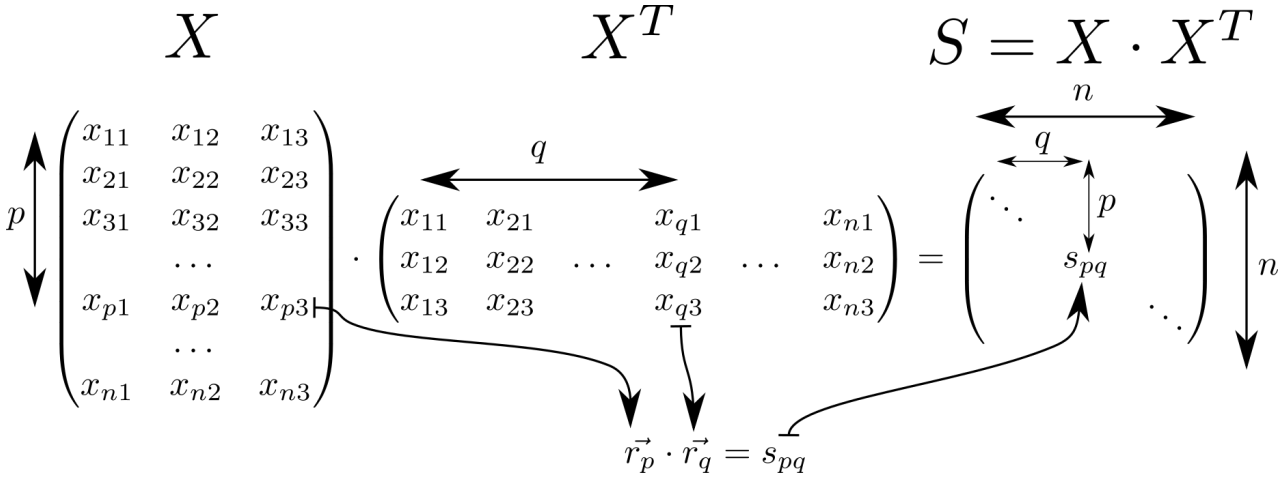
```
    for q in range(p + 1, n):
        if q != p:
            pq_distance = 0
            for i in range(3):
                pq_distance += (x[p, i] - x[q, i]) ** 2
            pq_distance = math.sqrt(pq_distance)
            L_for_inv += 1 / pq_distance  # обратное расстояние между двумя точками
L_for = (L_for_inv + L_for_sq) / n
print('loss =', L_for)
```

но есть небольшая беда. Для жалких 2000 точек эта программа будет работать 2 секунды.

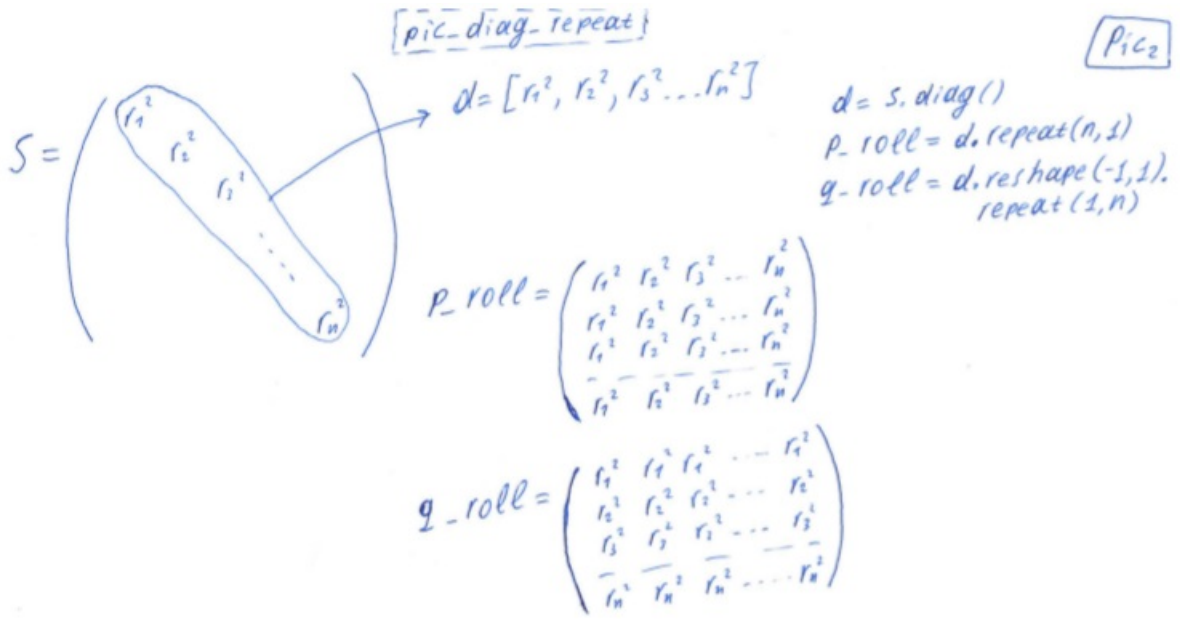
Чтобы посчитать потенциал попарного взаимодействия n точек быстрее, запишем их координаты в виде матрицы X размером $3 \times n$.



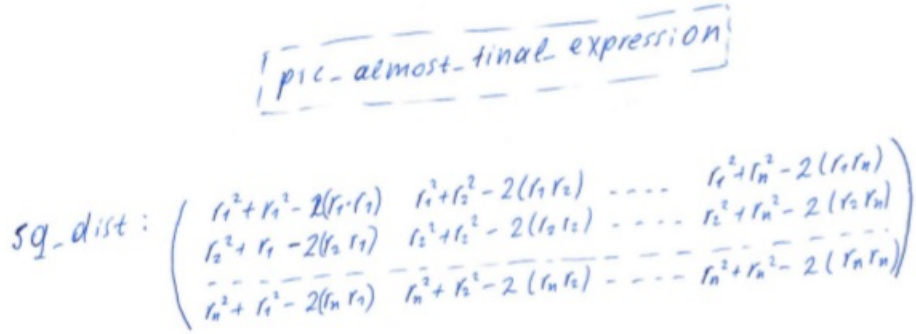
Матрица $S = X \cdot X^T$ обладает многими интересными свойствами и часто встречается в выкладках, связанных с теорией линейных классификаторов в ML. Так, если мы посмотрим на строчки матрицы X с индексами p и q как на вектора трёхмерного пространства $\vec{r_p}, \vec{r_q}$ то обнаружим что матрица S состоит из скалярных произведений этих векторов. Таких векторов n штук, значит размерность матрицы S равна $n \times n$.



На диагонали матрицы S стоят квадраты длин векторов $\vec{r_p}$: $s_{pp} = r_p^2$. Зная это, давайте считать полный потенциал взаимодействия. Начнём с расчёта двух вспомогательных матриц. В одной диагональ матрицы S будет повторяться в строках, в другой - в столбцах.



Посмотрим теперь на значение выражения `p_roll + q_roll - 2 * S`



Элемент с индексами (p, q) матрицы `sq_dist` равен $r_p^2 + r_q^2 - 2 \cdot (\overrightarrow{r_p}, \overrightarrow{r_q}) = (\overrightarrow{r_p} - \overrightarrow{r_q})^2$. То есть, у нас получилась матрица квадратов расстояний между точками. Выражение sq_dist.sum() даст одно-единственное число - полный потенциал попарного упругого взаимодействия между всеми точками. Попытки минимизировать этот потенциал не дают ничего хорошего. При данном потенциале взаимодействия точка сильнее чувствует точки, которые находятся от неё далеко. А чтобы получить равномерное распределение, нам нужно чтобы она в первую очередь обращала внимание на своих ближайших соседей.

Электростатическое отталкивание на сфере

`dist = torch.sqrt(sq_dist)` - матрица расстояний между точками. Нам нужно посчитать потенциал, учитывающий отталкивание точек между собой и притяжение к сфере. Поставим на диагональ единицы и заменим каждый элемент на обратный ему (только не подумайте, что мы при этом обратили матрицу!):
`rec_dist_one = 1 / (dist + torch.eye(n))`. Получилась матрица, на диагонали которой стоят единицы, другие элементы - потенциалы электростатического взаимодействия между точками.

Добавим теперь квадратичный потенциал притяжения к поверхности единичной сферы.

Расстояние от поверхности сферы $(1 - r)$. Возводим его в квадрат и умножаем на k, который задаёт соотношение между ролью электростатического отталкивания частиц и притяжения сферы. Итого `k = 1000`, `all_interactions = rec_dist_one - torch.eye(n) + (d.sqrt() - torch.ones(n))**2`. Долгожданный таргет, который мы будем минимизировать: `t = all_interactions.sum()`

Программа, рассчитывающая потенциал с помощью библиотеки numpy:

```
%%time
xxt = x.dot(x.T)
pp_sq_dist = np.diag(xxt)
p_roll = pp_sq_dist.reshape(1, -1).repeat(n, axis=0)
q_roll = pp_sq_dist.reshape(-1, 1).repeat(n, axis=1)
pq_sq_dist = p_roll + q_roll - 2 * xxt
pq_dist = np.sqrt(pq_sq_dist)
pp_dist = np.sqrt(pp_sq_dist)
surface_dist_sq = (pp_dist - np.ones(n)) ** 2
rec_pq_dist = 1 / (pq_dist + np.eye(n)) - np.eye(n)
L_np_rec = rec_pq_dist.sum() / 2
L_np_surf = k * surface_dist_sq.sum()
L_np = (L_np_rec  + L_np_surf) / n
print('loss =', L_np)
```

Здесь дела обстоят чуть лучше - 200 мс на 2000 точек.

Используем pytorch:

```
import torch

%%time
pt_x = torch.from_numpy(x)
pt_xxt = pt_x.mm(pt_x.transpose(0, 1))
pt_pp_sq_dist = pt_xxt.diag()
pt_p_roll = pt_pp_sq_dist.repeat(n, 1)
pt_q_roll = pt_pp_sq_dist.reshape(-1, 1).repeat(1, n)
pt_pq_sq_dist = pt_p_roll + pt_q_roll - 2 * pt_xxt
pt_pq_dist = pt_pq_sq_dist.sqrt()
pt_pp_dist = pt_pp_sq_dist.sqrt()
pt_surface_dist_sq = (pt_pp_dist - torch.ones(n, dtype=torch.float64)) ** 2
pt_rec_pq_dist = 1/ (pt_pq_dist + torch.eye(n, dtype=torch.float64)) - torch.eye(n, dtype=torch.f
loat64)
L_pt = (pt_rec_pq_dist.sum() / 2 + k * pt_surface_dist_sq.sum()) / n
print('loss =', float(L_pt))
```

И, наконец, tensorflow:

```
import tensorflow as tf

tf_x = tf.placeholder(name='x', dtype=tf.float64)
tf_xxt = tf.matmul(tf_x, tf.transpose(tf_x))
tf_pp_sq_dist = tf.diag_part(tf_xxt)
tf_p_roll = tf.tile(tf.reshape(tf_pp_sq_dist, (1, -1)), (n, 1))
tf_q_roll = tf.tile(tf.reshape(tf_pp_sq_dist, (-1, 1)), (1, n))
tf_pq_sq_dist = tf_p_roll + tf_q_roll - 2 * tf_xxt
tf_pq_dist = tf.sqrt(tf_pq_sq_dist)
tf_pp_dist = tf.sqrt(tf_pp_sq_dist)
```

```
tf_surface_dist_sq = (tf_pp_dist - tf.ones(n, dtype=tf.float64)) ** 2
tf_rec_pq_dist = 1 / (tf_pq_dist + tf.eye(n, dtype=tf.float64)) - tf.eye(n, dtype=tf.float64)
L_tf = (tf.reduce_sum(tf_rec_pq_dist) / 2 + k * tf.reduce_sum(tf_surface_dist_sq)) / n

glob_init = tf.local_variables_initializer()

%%time
with tf.Session() as tf_s:
    glob_init.run()
    res, = tf_s.run([L_tf], feed_dict={tf_x: x})
    print(res)
```

Сравниваем производительность этих трёх подходов:

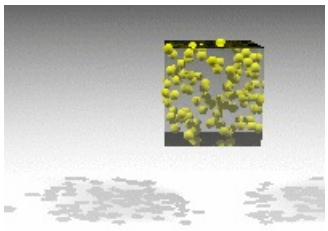
N	python	numpy	pytorch	tensorflow
2000	4.03	0.083	1.11	0.205
10000	99	2.82	2.18	7.9

Векторизованные вычисления дают выигрыш более чем на полтора десятичных порядка относительно кода на чистом python. Виден "boiler plate" у pytorch: вычисления малого объёма занимают заметное время, но оно почти не изменяется при увеличении объёма вычислений.

Визуализация

В наше время данные можно визуализировать средствами огромного количества пакетов, таких как Matlab, Wolfram Mathematics, Mapple, Matplolib и т.д. и т.п. В этих пакетах очень много сложных функций, делающих сложные вещи. К сожалению, если перед тобой стоит простая, но нестандартная задача, ты оказываешься безоружен. Моё любимое решение в такой ситуации - povray. Это очень мощная программа, которую обычно применяют для создания фотореалистичных изображений, но её можно использовать как "ассемблер визуализации". Обычно, сколь бы сложной не была поверхность, которую хочется отобразить, достаточно попросить povray нарисовать сферы с центрами, лежащими на этой поверхности.

С помощью библиотеки vapory можно создать povray сцену прямо в python, отрендерить её и посмотреть на результат. Сейчас он выглядит так:



Картинка получена так:

```
import vapory
from PIL import Image

def create_scene(moment):
```

```
    angle = 2 * math.pi * moment / 360
    r_camera = 7
    camera = vapory.Camera('location', [r_camera * math.cos(angle), 1.5, r_camera * math.sin(angl
e)], 'look_at', [0, 0, 0], 'angle', 30)
    light1 = vapory.LightSource([2, 4, -3], 'color', [1, 1, 1])
    light2 = vapory.LightSource([2, 4, 3], 'color', [1, 1, 1])
    plane = vapory.Plane([0, 1, 0], -1, vapory.Pigment('color', [1, 1, 1]))
    box = vapory.Box([0, 0, 0], [1, 1, 1],
                     vapory.Pigment('Col_Glass_Clear'),
                     vapory.Finish('F_Glass9'),
                     vapory.Interior('I_Glass1'))
    spheres = [vapory.Sphere( [float(r[0]), float(r[1]), float(r[2])], 0.05, vapory.Texture(vapor
y.Pigment('color', [1, 1, 0]))) for r in x]
    return vapory.Scene(camera, objects=[light1, light2, plane, box] + spheres, included=['glass.
inc'])
```

Далее будет рассказано о том, как запустить минимизацию функционала, чтобы точки вышли из куба и равномерно расползлись по сфере.