Abstract

Error feedback (EF) так же известная как "компенсация ошибки" - популярный механизм стабилизации конвергенции (сходимости) в контексте распределенного обучения моделей машинного обучения с учителем, улучшенный за счет использования механизмов сжатия сообщений. Сейчас предлагают следующие решения: 1) либо применяем к настройке одной машины 2) либо полагаемся на очень сильные и часто необоснованные предположения, такие как глобальная ограниченность градиентов, или итеративно-зависимые предположения, которые не могут быть проверены априори и могут не выполняться на практике. 3) либо обходят проблемы за счет введения дополнительных несмещенных компрессоров, которые увеличивают стоимость связи.

В работе предлагают новый метод работы с ошибками без глобальных предположений, лишь с требованием разнородности данных.

Introduction

Поскольку модели могут иметь огромное количество параметров, современные эффективные методы оптимизации используют сжатие данных с потерями (lossy communication compress), чтобы уменьшить размер передаваемых сообщений. В следствие чего авторы ставят своей задачей решить *невыпуклую распределенную задачу оптимизации*

$$\min_{x \in \mathbb{R}^d} \left[f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x) \right],$$

где $x \in \mathbb{R}^d$ - обучаемый вектор параметров, n - количество машин, $f_i(x)$ - потери модели на x на данных, хранящихся на устройстве i.

Background and Motivation

2.1 Two families of compression operators

Определение смещенных и несмещенных операторов. Как известно (??) $\mathbb{U} \subset \mathbb{B}$. Пример biased - Top-k. Пример unbiased - Rand-k. \ Жадные алгоритмы хорошо сохраняют главную суть передаваемого сообщения. Действительно, и Top-k, и Rand-k в худшем случае вызывают одинаковые искажения, что происходит, когда энергия распределяется равномерно по всем элементам входного вектора, и нелегко теоретически понять, что в этом худшем случае ситуация не будет происходить неоднократно на протяжении итераций.

In particular, we say that a (possibly randomized) map $\mathcal{C} : \mathbb{R}^d \to \mathbb{R}^d$ is an *unbiased compression operator*, or simply just *unbiased compressor*, if there exists a constant $\omega \geq 0$ such that

$$\mathbb{E}\left[\mathcal{C}(x)\right] = x, \quad \mathbb{E}\left[\|\mathcal{C}(x) - x\|^2\right] \leq \omega \|x\|^2, \qquad \forall x \in \mathbb{R}^d. \tag{2}$$

The family of such operators will be denoted by $\mathbb{U}(\omega)$. Further, we say that a (possibly randomized) map $\mathcal{C} : \mathbb{R}^d \to \mathbb{R}^d$ is a *biased compression operator*, or simply just *biased compressor*, if there exists a constant $0 < \alpha \leq 1$ such that

$$\mathbb{E}\left[\|\mathcal{C}(x) - x\|^2\right] \leq (1 - \alpha) \|x\|^2, \qquad \forall x \in \mathbb{R}^d. \tag{3}$$

The family of such operators will be denoted by $\mathbb{B}(\alpha)$. It is well known that, in a certain sense, the latter

$\mathbb{B}$ строго больше, так как содержит компрессор, который не возникает в результате масштабирования несмещенного компрессора. Biased эмпирически превосходят Unbiased.

2.2 Error feedback: what it is good for, and what we still do not know

Несмещенность оказывается очень эффективным инструментом, облегчающим анализ распределенных методов первого порядка, использующих несмещенные компрессоры, и по этой причине ландшафт методов, использующих такие компрессоры, очень богат и относительно хорошо изучен. Например, используя несмещенные компрессоры, мы знаем (список).

Смещенные же наоборот сложны для изучения. Основная сложность в том, что наивная реализация может привести к расхождению

Example 1 *Consider $n = d = 3$ and define*

$$f_1(x) = \langle a, x\rangle^2 + \frac{1}{4}\|x\|_2^2, \qquad f_2(x) = \langle b, x\rangle^2 + \frac{1}{4}\|x\|_2^2, \qquad f_3(x) = \langle c, x\rangle^2 + \frac{1}{4}\|x\|_2^2,$$

where $a = (-3, 2, 2)$, $b = (2, -3, 2)$ and $c = (2, 2, -3)$. Let the starting iterate be $x^0 = (t, t, t)$, where $t > 0$. Then

$$\nabla f_1(x^0) = \frac{t}{2}(-11, 9, 9), \qquad \nabla f_2(x^0) = \frac{t}{2}(9, -11, 9), \qquad \nabla f_3(x^0) = \frac{t}{2}(9, 9, -11).$$

Using the Top-1 compressor, we get $\mathcal{C}(\nabla f_1(x^0)) = \frac{t}{2}(-11, 0, 0)$, $\mathcal{C}(\nabla f_2(x^0)) = \frac{t}{2}(0, -11, 0)$ and $\mathcal{C}(\nabla f_3(x^0)) = \frac{t}{2}(0, 0, -11)$. The next iterate of DCGD is

$$x^1 = x^0 - \eta \frac{1}{3} \sum_{i=1}^{3} \mathcal{C}(\nabla f_i(x^0)) = \left(1 + \frac{11\eta}{6}\right) x^0.$$

Repeated application gives

$$x^k = \left(1 + \frac{11\eta}{6}\right)^k x^0.$$

Since $\eta > 0$, the entries of x^k diverge exponentially fast to $+\infty$.

> Возможность ли получить линейно сходящийся метод с компенсацией ошибок в общих условиях гетерогенных данных, опираясь только на смещенные компрессоры, все еще остается открытой проблемой.

[Koloskova et al. 2020] показали, что если считать полный градиент на каждой ноде (в отличие от стохастического метода), то после T раундов коммуникации можно найти случайный вектор, для которого гарантированного выполняется

$$\mathbb{E}\left[||\nabla f(x)||^2\right] = \mathbb{O}\left(\frac{G^{2/3}}{T^{2/3}}\right),$$

в предположении

$$||\nabla f_i(x)||^2 \leq G^2, \forall x \in \mathbb{R}^d, i \in 1, 2, \ldots, n$$

Algorithm	sCVX	nCVX	DIST	key limitation
EF Stich et al. [2018]	✓	✗	✓	bounded gradients; sublinear rate in sCVX case
EF-SGD Stich and Karimireddy [2019]	✓	✓	✗	single node only
EF Ajalloeian and Stich [2020]	✓	✓	✗	single node only
SignSGD Karimireddy et al. [2019]	✗	✓	✗	moment bound; single node only
EC-SGD Beznosikov et al. [2020]	✓	✗	✓	linear rate only if $\nabla f_i(x^\star) = 0 \; \forall i$
EC-SGD Gorbunov et al. [2020b]	✓	✗	✓	linear rate only using an extra unbiased compressor
DoubleSqueeze Tang et al. [2020]	✗	✓	✓	bounded compression error; slow $O(1/T^{2/3})$ rate in nCVX case
Qsparse-SGD, CSER Basu et al. [2019], Xie et al. [2020]	✓	✓	✓	bounded gradients; slow $O(1/T^{1/2})$ rate in nCVX case
EC-SGD Koloskova et al. [2020]	✗	✓	✓†	bounded gradients; slow $O(1/T^{2/3})$ rate in nCVX case

Table 1: Known results for first order methods using biased compressors. sCVX = supports strongly convex functions, nCVX= supports nonconvex functions, DIST = works in the distributed regime. †decentralized method

Текущая ситуация с анализом смещенных компрессоров.

2.3 Summary of contributions

- Новый error feedback механизм. Методы работают в условиях распределенных гетерогенных (разнообразных, например разных форматов или типов: числа, текст) данных.
- Упростились *условия, накладываемые на данные*:
  - L_i-гладкость функций f_i
  - существование глобального минимума $f^{inf} \in \mathbb{R}^d$
- Линейная скорость для функций Поляка-Лоясевича

Main Results

Раньше схема сжатия была построена на следующем способе. В каждой ноде вычисляем градиент, сжимаем его с помощью biased метода и отправляем сжатый градиент на сервер. Сервер агрегирует все посылки от n агентов и делает шаг.

$$x^{t+1} = x^t - \frac{\gamma}{n} \sum_{i=1}^{n} C(\nabla f_i(x^t))$$

Но возникала ситуация, когда этот метод мог расходится. Можно посмотреть на эту задачу под другим углом. Предполагая, что $\nabla f_i(x^t)$ сходится к некоторму минимуму мы можем записать:

$$\mathbb{E}\left[||C(\nabla f_i(x^t)) - \nabla f_i(x^t)||^2\right] \leq (1 - \alpha)||\nabla f_i(x^t)||^2$$

Однако в минимуме градиент функции не обязан быть равен 0, а потому правая часть может быть очень большой, поэтому метод как бы не сходится к некому $\tilde{x}$.

Тогда сделаем следующее: вместо градиентов будем сжимать разные векторы, которые, как мы надеемся, будут сходиться к нулям. Поскольку ввиду (3) применение C к убывающим векторам вносит постепенно убывающие искажения, проблема стабилизации будет решена. Осталось понять, какие векторы брать.

3.1-3.2 Компрессора Маркова

Предположим, нам дана последовательность входных векторов $\{v^t\}_{t \geq 0}$ (например, градиентов), сгенерированная некоторым алгоритмом. Эта последовательность не обязательно сходится к нулю. Наша цель — создать последовательность «хороших» и «легких для передачи» (некоторому объекту, который мы будем называть «главным») оценок этих векторов, используя компрессор $\mathbb{C} \in \mathbb{B}(\alpha)$. Давайте пройдем несколько этапов открытия.

Определим рекрсивный способ построения компрессора, который назовем *компрессором Маркова*.

$$\mathbb{M}(v^0) := C(v^0)$$

$$\mathbb{M}(v^{t+1}) := \mathbb{M}(v^t) + C(v^{t+1} - \mathbb{M}(v^t)), t \geq 0$$

In [1]:
```
import numpy as np
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import torch
from sklearn.model_selection import train_test_split

import math
```

Реализуем этот алгоритм. Рассмотрим `Top-k`, который работает по следующему принципу.

$$\mathbb{C}(x) := \sum_{i=d-k+1}^{d} x_{(i)} e_{(i)},$$

где x - некоторый вектор, который мы хотим сжать, причем его координаты отсортированы по возрастанию, т.е. $|x_{(1)}| \leq |x_{(2)}| \leq \ldots \leq |x_{(d)}|$.

Пример. Пусть у нас есть вектор $x = (-11, 9, 9)$. Тогда после применения *Top-1*, получим $\mathbb{C}(x) = (-11, 0, 0)$.

In [2]:
```
def compressor(np_tensor, ratio=0.5):
    with torch.no_grad():
        tensor = torch.from_numpy(np.asarray(np_tensor))
        numel = tensor.numel()
        k = max(int(numel * ratio), 1)

        values, indexes = torch.topk(torch.abs(tensor), k=k)
        values = tensor[indexes]

        # np.append(values_, values)
        # np.append(indexes_, indexes)
        return tensor, indexes, values

def get_compr_object(tensor, ratio=0.5):
    tensor, indexes, values = compressor(tensor, ratio)
    for idx in range(len(tensor)):
        if idx not in indexes:
            tensor[idx] = 0
    return tensor.numpy()
```

In [3]:
```
def topk(tensor, k):
  indexes = np.abs(tensor).argsort()[-k:]
  return indexes

def other_get_compr(tensor, k=1):
  true_k = min(len(tensor), k)
  indexes = topk(tensor, true_k)
  for idx in range(len(tensor)):
        if idx not in indexes:
            tensor[idx] = 0
  return tensor
```

Пример работы метода Top-K.

In [4]:
```
vector = torch.tensor([9, -11, 2, 3, 4])
print(get_compr_object(vector, ratio=0.5))
```

```
[  9 -11   0   0   0]
```

Для оптимизации возьмем задачу минимизации эмпирического риска:

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} l(g(w, x_i), y_i) + \frac{\lambda}{2}\|w\|_2^2. \tag{1}$$

Будем рассматривать линейную модель $g(w, x) = w^T x$ и логистическую/сигмоидную функцию потерь $l(z, y) = \ln(1 + \exp(-yz))$ (**Важно:** y **должен принимать значения** -1 **или** 1). Такую задачу еще называют логистической регрессией.

$$\nabla f(w) = \frac{1}{n} \sum_{i=1}^{n} \frac{-x_i y_i exp(-\omega^T \cdot x_i \cdot y_i)}{1 + exp(-\omega^T \cdot x_i \cdot y_i)} + \lambda\omega \tag{2}$$

Вычисление константы Липшица. Ранее мы подбирали learning rate руками, что вообще говоря плохо. Поэтому попробуем вычислить его, основываясь на оценке константы Липшица.

$$||\nabla f(x) - \nabla f(y)||_2 \leq L||x - y||_2$$

В нашем случае $f(t) = ln(1 + exp(t))$. Тогда $L = \frac{1}{2}\sum_{i=1}^{n} |y_i|^2 ||x_i||^2$.

In [5]:
```
dataset = "mushrooms.txt"
#файл должен лежать в той же деректории, что и notebook

from sklearn.datasets import load_svmlight_file
data = load_svmlight_file(dataset)
X, y = data[0].toarray(), data[1]
n, d = X.shape

y = y * 2 - 3

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, shuffle=True)
```

In [6]:
```
def grad(x, y, w, lambd):
    n, d = x.shape
    gradient = np.zeros(d)
    for i in range(n):
        exp = np.exp(-(np.dot(w, x[i, :]) * y[i]))
        gradient += 1 / n * ((-x[i, :] * y[i] * exp) / (1 + exp))
    gradient += lambd * w
    return gradient

def loss_func(y, w, x):
    '''
    y - вектор скаляров {0, 1}
    w - веса
    x - данные
    '''

    return np.log(1 + np.exp(-1 * np.hstack(np.vstack(y)) * (np.vstack(x) @ np.asarray(w))))

def Lipschitz_constant(x, y):
    n, d = x.shape
    lipthiz = 0
    for i in range(n):
        lipthiz += y[i] * (np.linalg.norm(x[i, :], ord=2) ** 2)
    return lipthiz / 2
```

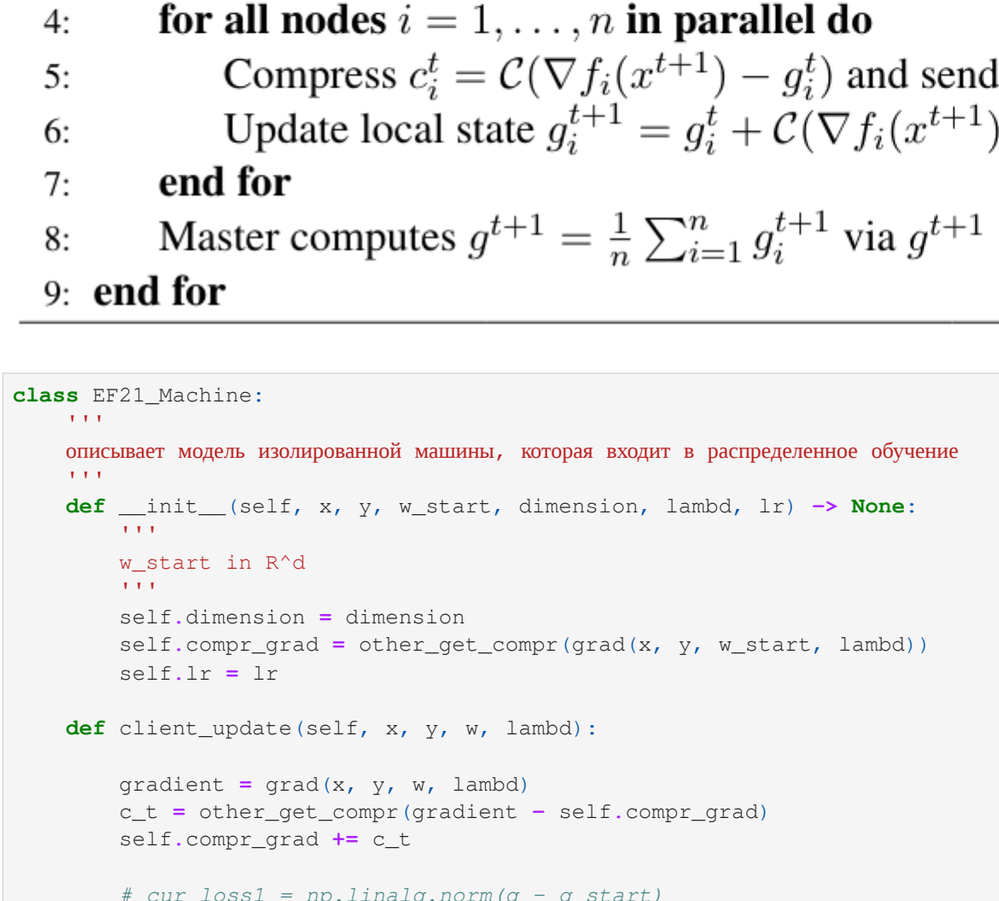
Algorithm 1 EF21 (Single node)

1: **Input:** starting point $x^0 \in \mathbb{R}^d$, learning rate $\gamma > 0$, $g^0 = \mathcal{C}(\nabla f(x^0))$
2: **for** $t = 0, 1, 2, \ldots, T-1$ **do**
3: $\quad x^{t+1} = x^t - \gamma g^t$
4: $\quad g^{t+1} = g^t + \mathcal{C}(\nabla f(x^{t+1}) - g^t)$
5: **end for**

In [7]:
```
# algo1 EF21 (single node) for logistic regression problem
def ef21_singlenode(x, y, w_start, lr, iterations=1000, accuracy=0.01, lambd=0.1):
    '''
    w_start - starting point in R^d
    lr - learning rate (lr > 0)
    g_start = C(grad(f(x_start)))
    '''
    w_temp = w_start
    g_start = other_get_compr(grad(x, y, w_start, lambd), k=1)
    history1 = []   # distance(g - g_start)
    history2 = []   # dist(gradient - 0)
    for _ in range(iterations):
        w = w_temp - lr * g_start
        gradient = grad(x, y, w, lambd)
        g = g_start + other_get_compr(gradient - g_start, k=1)

        cur_loss1 = np.linalg.norm(g - g_start, ord=2)
        history1.append(cur_loss1)
        cur_loss2 = np.linalg.norm(gradient)
        history2.append(cur_loss2)

        # if (cur_loss1 < accuracy):
        #     return [history1, history2]

        w_temp = w
        g_start = g
    return [history1, history2]
```

In [8]:
```
# using Top-1
g_d = ef21_singlenode(x=X_train, y=y_train, w_start=np.zeros(X_train.shape[1]), lr=0.000576, iterations=500, accuracy=0.001, lambd=0.1)
```

In [23]:
```
plt.plot(range(len(g_d[0][10:])), g_d[0][10:])
plt.title('Gradient Descent (EF21_single_node)')
plt.xlabel('Iterations')
plt.ylabel('$||g_i - g\_start_i||$')
plt.grid()
plt.show()
```



3.3 Compressed gradient descent using the Markov compressor

Algorithm 2 EF21 (Multiple nodes)

1: **Input:** starting point $x^0 \in \mathbb{R}^d$; $g_i^0 = \mathcal{C}(\nabla f_i(x^0))$ for $i = 1, \ldots, n$ (known by nodes and the master); learning rate $\gamma > 0$; $g^0 = \frac{1}{n}\sum_{i=1}^{n} g_i^0$ (known by master)
2: **for** $t = 0, 1, 2, \ldots, T-1$ **do**
3: $\quad$ Master computes $x^{t+1} = x^t - \gamma g^t$ and broadcasts x^{t+1} to all nodes
4: $\quad$ **for all nodes** $i = 1, \ldots, n$ **in parallel do**
5: $\qquad$ Compress $c_i^t = \mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$ and send c_i^t to the master
6: $\qquad$ Update local state $g_i^{t+1} = g_i^t + \mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$
7: $\quad$ **end for**
8: $\quad$ Master computes $g^{t+1} = \frac{1}{n}\sum_{i=1}^{n} g_i^{t+1}$ via $g^{t+1} = g^t + \frac{1}{n}\sum_{i=1}^{n} c_i^t$
9: **end for**

In [31]:
```
class EF21_Machine:
    '''
    описывает модель изолированной машины, которая входит в распределенное обучение
    '''
    def __init__(self, x, y, w_start, dimension, lambd, lr) -> None:
        '''
        w_start in R^d
        '''
        self.dimension = dimension
        self.compr_grad = other_get_compr(grad(x, y, w_start, lambd))
        self.lr = lr

    def client_update(self, x, y, w, lambd):

        gradient = grad(x, y, w, lambd)
        c_t = other_get_compr(gradient - self.compr_grad)
        self.compr_grad += c_t

        # cur_loss1 = np.linalg.norm(g - g_start)
        # history1.append(cur_loss1)
        # cur_loss2 = np.linalg.norm(gradient)
        # history2.append(cur_loss2)

        return c_t

    def get_g_start(self, x, y, w_start, lambd):
        return other_get_compr(grad(x, y, w_start, lambd))


class EF21_Server:
    def __init__(self, x, y, lambd, number_machine, lr, iterations) -> None:
        '''
        x, y, w_start, lambd - параметры для инициализации машин
        number_machine - количество машин, на которых распределяем вычисления
        lr - learning rate
        iterations - количество итераций
        '''
        dimension = x[0].shape[1]
        self.x = x
        self.y = y
        self.w = np.zeros(dimension)
        self.machines = [EF21_Machine(self.x[i], self.y[i], self.w, dimension, lambd, lr) for i in range(number_machine)]
        self.g_start = np.sum(np.array([self.machines[i].get_g_start(x=self.x[i], y=self.y[i], w_start=self.w, lambd=lambd) for i in range(number_machine)]), axis=0) / number_machine
        self.number_machine = number_machine
        self.lr = lr
        self.iterations = iterations
        self.lambd = lambd

    def server_executes(self, accuracy):
        '''
        accuracy - точность вычислений
        работаем на данных self.x, self.y
        '''
        # history = []
        history1 = []

        for t in range(self.iterations):
            w_next = self.w - self.lr * self.g_start
            grads_from_machine = [self.machines[i].client_update(self.x[i], self.y[i], w=w_next, lambd=self.lambd) for i in range(len(self.machines))]
            g_next = self.g_start + np.sum(np.array(grads_from_machine), axis=0) / self.number_machine

            cur_loss1 = np.linalg.norm(g_next - self.g_start)
            history1.append(cur_loss1)
            # if history1[t] < accuracy:
            #     return self.w, history1

            self.w = w_next
            self.g_start = g_next


        return self.w, history1
```

Смоделируем ситуацию распределенного обучения. Разделим исходный датасет на 5 частей. Они будут представлять из себя 5 устройств.

In [28]:
```
NUM_MACHINE = 5

X_train = X_train[:X_train.shape[0] // NUM_MACHINE * NUM_MACHINE]
y_train = y_train[:y_train.shape[0] // NUM_MACHINE * NUM_MACHINE]
X_train_set = np.vsplit(X_train, NUM_MACHINE)
y_train_set = np.split(y_train, NUM_MACHINE)
```

Вычислим необходимый learning rate из теоремы 1:

$$0 \leq \gamma \leq \left(L + \tilde{L}\sqrt{\frac{\beta}{\theta}}\right)^{-1},$$

где $\sqrt{\frac{\beta}{\theta}} := \frac{1+\sqrt{1-\alpha}}{\alpha} - 1, \alpha \in (0, 1]$.

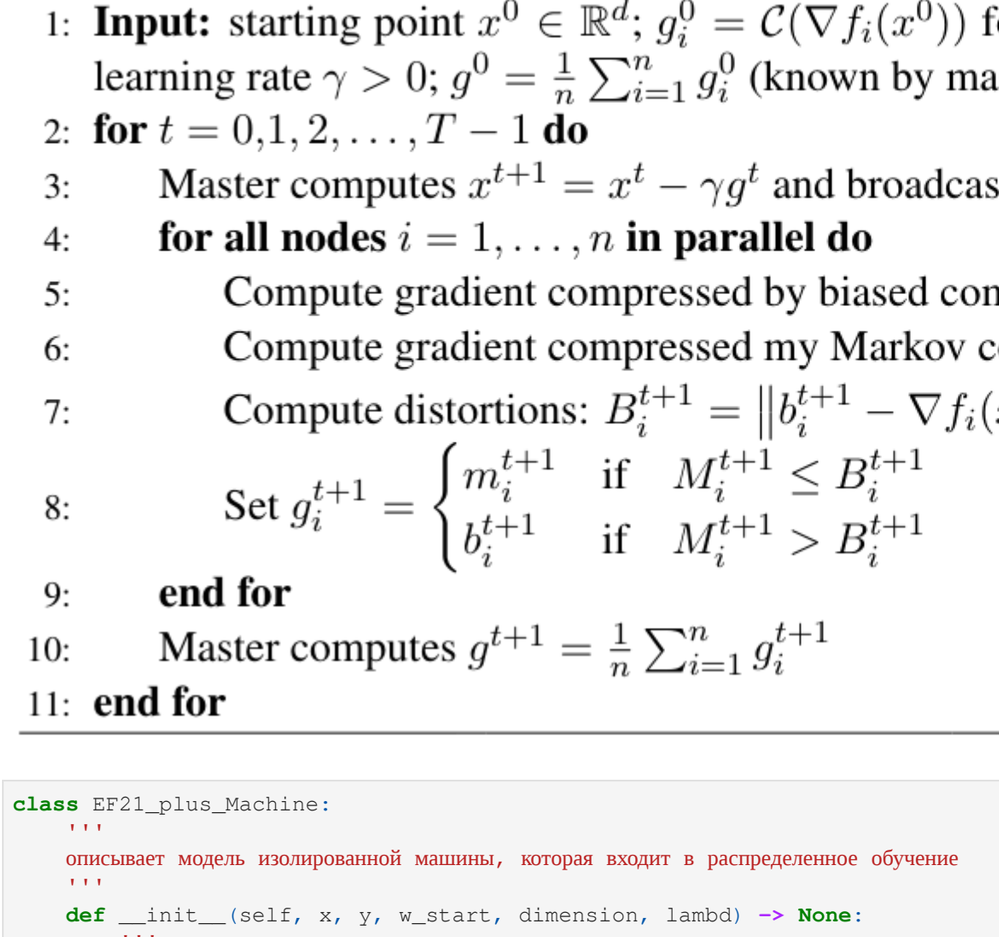
Во всех экспериментах использовался регуляровочный коэффициент $\lambda = 0.1$.

In [12]:
```
alpha = 0.5
L_set = np.array([Lipschitz_constant(X_train_set[i], y_train_set[i]) for i in range(NUM_MACHINE)])
L = np.sum(L_set) / NUM_MACHINE
L_tild = (np.sum(L_set ** 2) / NUM_MACHINE) ** 0.5
learn_rate = 1 / (L + L_tild * ((1 + np.sqrt(1 - alpha))/alpha - 1))
learn_rate
```

Out[12]:
```
0.0005186731844279578
```

In [32]:
```
ef21_server = EF21_Server(x=X_train_set, y=y_train_set, lambd=0.1, number_machine=NUM_MACHINE, lr=learn_rate, iterations=500)
ef21_final_weight, ef21_comput_history = ef21_server.server_executes(accuracy=0.001)
```

In [33]:
```
plt.plot(range(len(ef21_comput_history)), ef21_comput_history)

plt.title('Gradient Descent (EF21)')
plt.xlabel('Iterations')
plt.ylabel('$||g_i - g\_start_i||$')
plt.grid()
plt.show()
```



Итак, метод сходится за ~ 100 итераций.

3.5 EF21+: Use C or Markov compressor, whichever is better.

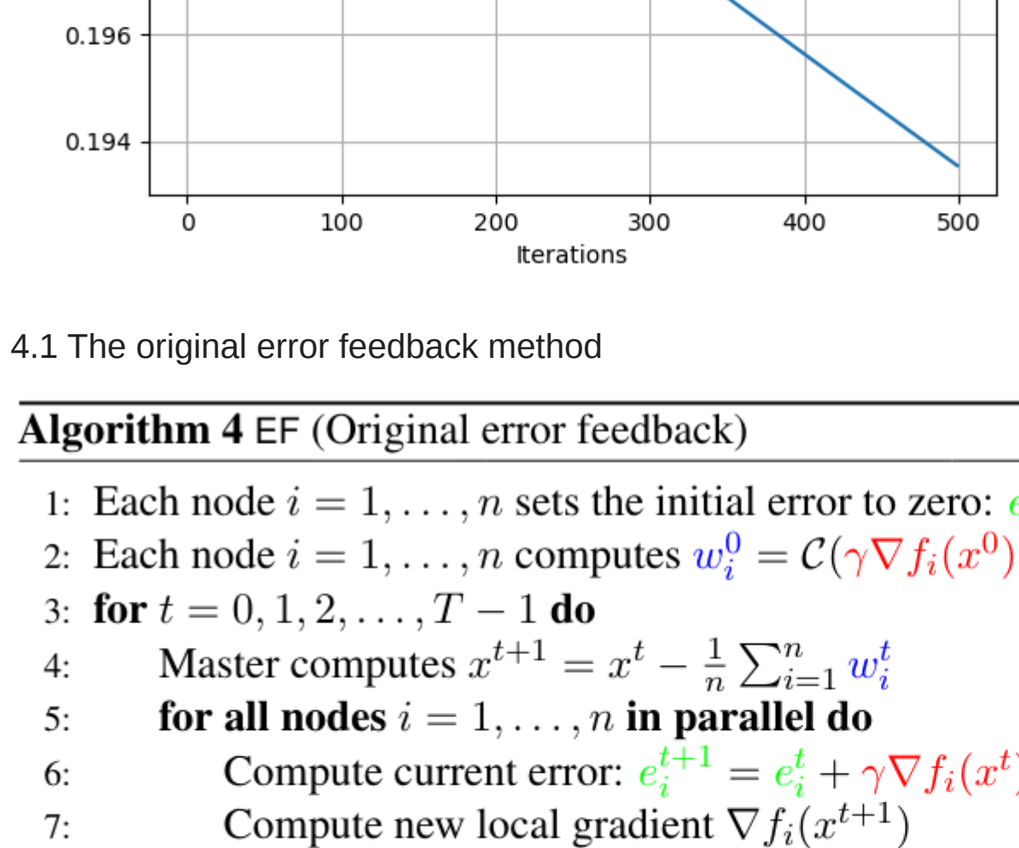
Algorithm 3 EF21+ (Multiple nodes)

1: **Input:** starting point $x^0 \in \mathbb{R}^d$; $g_i^0 = \mathcal{C}(\nabla f_i(x^0))$ for $i = 1, \ldots, n$ (known by nodes and the master); learning rate $\gamma > 0$; $g^0 = \frac{1}{n}\sum_{i=1}^{n} g_i^0$ (known by master)
2: **for** $t = 0, 1, 2, \ldots, T-1$ **do**
3: $\quad$ Master computes $x^{t+1} = x^t - \gamma g^t$ and broadcasts x^{t+1} to all nodes
4: $\quad$ **for all nodes** $i = 1, \ldots, n$ **in parallel do**
5: $\qquad$ Compute gradient compressed by biased compressor $b_i^{t+1} = \mathcal{C}(\nabla f_i(x^{t+1}))$
6: $\qquad$ Compute gradient compressed my Markov compressor $m_i^{t+1} = g_i^t + \mathcal{C}(\nabla f_i(x^{t+1}) - g_i^t)$
7: $\qquad$ Compute distortions: $B_i^{t+1} = \left\|b_i^{t+1} - \nabla f_i(x^{t+1})\right\|^2$; $M_i^{t+1} = \left\|m_i^{t+1} - \nabla f_i(x^{t+1})\right\|^2$
8: $\qquad$ Set $g_i^{t+1} = \begin{cases} m_i^{t+1} & \text{if} \quad M_i^{t+1} \leq B_i^{t+1} \\ b_i^{t+1} & \text{if} \quad M_i^{t+1} > B_i^{t+1} \end{cases}$
9: $\quad$ **end for**
10: $\quad$ Master computes $g^{t+1} = \frac{1}{n}\sum_{i=1}^{n} g_i^{t+1}$
11: **end for**

In [15]:
```
class EF21_plus_Machine:
    '''
    описывает модель изолированной машины, которая входит в распределенное обучение
    '''
    def __init__(self, x, y, w_start, dimension, lambd) -> None:
        '''
        w_start in R^d
        '''
        self.dimension = dimension
        self.compr_grad = other_get_compr(grad(x, y, w_start, lambd), k=1)

    def client_update(self, x, y, w, lambd):
        # gradient = grad(x, y, w, lambd)
        # c_t = other_get_compr(gradient - self.compr_grad)
        # self.compr_grad += c_t

        gradient = grad(x, y, w, lambd)
        biased_compressor_t = other_get_compr(gradient, k=1)
        markov_compressor_t = self.compr_grad + other_get_compr(gradient - self.compr_grad, k=1)

        # compute distortions
        biased_distor_t = np.linalg.norm(biased_compressor_t - gradient, ord=2)
        markov_distor_t = np.linalg.norm(markov_compressor_t - gradient, ord=2)

        if (markov_distor_t > biased_distor_t):
            self.compr_grad = biased_compressor_t
        else:
            self.compr_grad = markov_compressor_t

        return self.compr_grad

    def get_g_start(self):
        return self.compr_grad


class EF21_plus_Server:
    def __init__(self, x, y, lambd, number_machine, lr, iterations) -> None:
        '''
        x, y, w_start, lambd - параметры для инициализации машин
        number_machine - количество машин, на которых проводим вычисления
        lr - learning rate
        iterations - количество итераций
        '''
        dimension = x[0].shape[1]
        self.x = x
        self.y = y
        self.number_machine = number_machine
        self.lr = lr
        self.iterations = iterations
        self.lambd = lambd

        self.w = np.zeros(dimension)
        self.machines = [EF21_plus_Machine(self.x[i], self.y[i], self.w, dimension, lambd) for i in range(number_machine)]

    def server_executes(self, accuracy):
        '''
        accuracy - точность вычислений
        работаем на данных self.x, self.y
        '''
        history1 = []
        g_start = np.sum(np.array([self.machines[i].get_g_start() for i in range(self.number_machine)]), axis=0) / self.number_machine

        for t in range(self.iterations):
            w_next = self.w - self.lr * g_start
            grads_from_machine = [self.machines[i].client_update(self.x[i], self.y[i], w=w_next, lambd=self.lambd) for i in range(len(self.machines))]
            g_start = np.sum(np.array(grads_from_machine), axis=0) / self.number_machine

            cur_loss1 = np.linalg.norm(g_start, ord=2)
            history1.append(cur_loss1)

            self.w = w_next


        return self.w, history1
```

In [16]:
```
ef21_plus_server = EF21_plus_Server(x=X_train_set, y=y_train_set, lambd=0.1, number_machine=NUM_MACHINE, lr=learn_rate, iterations=500)
ef21_plus_final_weight, ef21_plus_comput_history = ef21_plus_server.server_executes(accuracy=0.001)
```

In [25]:
```
plt.plot(range(len(ef21_plus_comput_history)), ef21_plus_comput_history)
plt.title('Gradient Descent (EF21+)')
plt.xlabel('Iterations')
plt.ylabel('$||g^t(w)||$')
plt.grid()
plt.show()
```



4.1 The original error feedback method

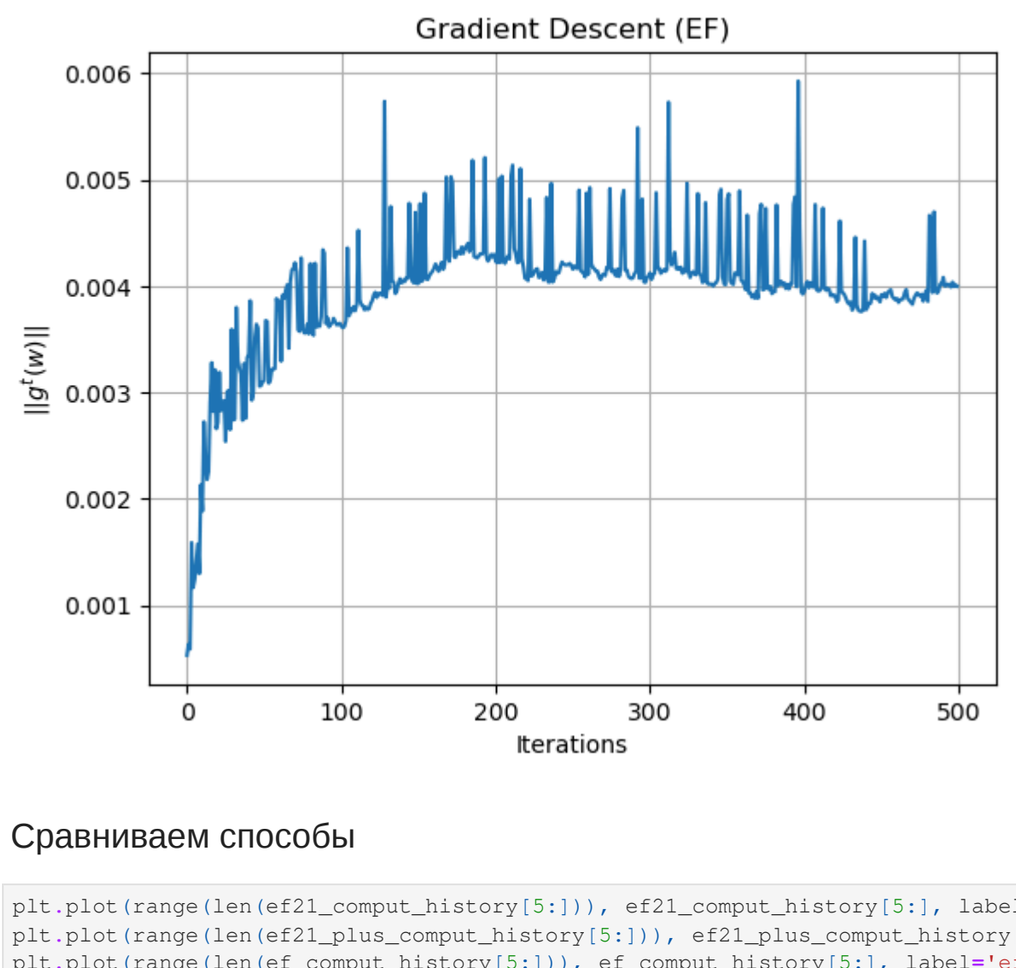
Algorithm 4 EF (Original error feedback)

1: Each node $i = 1, \ldots, n$ sets the initial error to zero: $e_i^0 = 0$
2: Each node $i = 1, \ldots, n$ computes $w_i^0 = \mathcal{C}(\gamma\nabla f_i(x^0))$ and sends this to the master
3: **for** $t = 0, 1, 2, \ldots, T-1$ **do**
4: $\quad$ Master computes $x^{t+1} = x^t - \frac{1}{n}\sum_{i=1}^{n} w_i^t$
5: $\quad$ **for all nodes** $i = 1, \ldots, n$ **in parallel do**
6: $\qquad$ Compute current error: $e_i^{t+1} = e_i^t + \gamma\nabla f_i(x^t) - w_i^t$
7: $\qquad$ Compute new local gradient $\nabla f_i(x^{t+1})$
8: $\qquad$ Compute error-compensated (stepsize-scaled) gradient $w_i^{t+1} = \mathcal{C}(e_i^{t+1} + \gamma\nabla f_i(x^{t+1}))$
9: $\qquad$ Send w_i^{t+1} to the master
10: $\quad$ **end for**
11: **end for**

In [38]:
```
class EF_Machine:
    '''
    описывает модель изолированной машины, которая входит в распределенное обучение
    '''
    def __init__(self, x, y, w_start, lambd, dimension, lr) -> None:
        self.w_start = w_start
        self.error = np.zeros(dimension)
        self.compr_grad = other_get_compr(lr * grad(x, y, w_start, lambd))

    def client_update(self, x, y, w, lambd, lr):
        self.error += lr * grad(x, y, self.w_start, lambd) - self.compr_grad
        gradient = grad(x, y, w, lambd)
        self.compr_grad = other_get_compr(self.error + lr * gradient)
        self.w_start = w
        return self.compr_grad

    def get_g_start(self):
        return self.compr_grad


class EF_Server:
    def __init__(self, x, y, lambd, number_machine, lr, iterations) -> None:
        '''
        number_machine - количество машин, на которых распределяем вычисления
        lr - learning rate
        iterations - количество итераций
        '''
        self.dimension = x[0].shape[1]
        self.x = x
        self.y = y
        self.w_start = np.zeros(self.dimension)
        self.machines = [EF_Machine(self.x[i], self.y[i], self.w_start, lambd, self.dimension, lr) for i in range(number_machine)]
        self.number_machine = number_machine
        # self.point_from_machine = None
        self.lr = lr
        self.lambd = lambd
        self.iterations = iterations

    def server_executes(self, accuracy):
        '''
        accuracy - точность вычислений
        '''
        history = []
        point_start = [self.machines[i].get_g_start() for i in range(self.number_machine)]

        for t in range(self.iterations):
            g_start = np.sum(np.array(point_start), axis=0)
            w_next = self.w_start - g_start / self.number_machine
            point_start = [self.machines[i].client_update(self.x[i], self.y[i], w=w_next, lambd=self.lambd, lr=self.lr) for i in range(self.number_machine)]

            loss = np.linalg.norm(g_start, ord=2)
            history.append(loss)

            self.w_start = w_next

        return self.w_start, history
```

In [39]:
```
ef_server = EF_Server(x=X_train_set, y=y_train_set, lambd=0.1, number_machine=NUM_MACHINE, lr=learn_rate, iterations=500)
ef_final_weight, ef_comput_history = ef_server.server_executes(accuracy=0.001)
```

In [40]:
```
plt.plot(range(len(ef_comput_history)), ef_comput_history)
plt.title('Gradient Descent (EF)')
plt.xlabel('Iterations')
plt.ylabel('$||g^t(w)||$')
plt.grid()
plt.show()
```



Сравниваем способы

In [21]:
```
plt.plot(range(len(ef21_comput_history[5:])), ef21_comput_history[5:], label='ef21')
plt.plot(range(len(ef21_plus_comput_history[5:])), ef21_plus_comput_history[5:], label='ef21+')
plt.plot(range(len(ef_comput_history[5:])), ef_comput_history[5:], label='ef')
plt.title("All method")
plt.legend()
plt.show()
```