Курс: Методы оптимизации в машинном обучении, осень 2016 ВМК МГУ

Практическое задание 4. Минимизация суммы функций.

Начало выполнения задания: 14 декабря 2016 г.
Срок сдачи: 28 декабря (среда), 23:59.
Среда для выполнения задания: Python 3.

1 Краткое описание задания

В этом задании Вам предлагается реализовать и протестировать два популярных метода для минимизации суммы функций: SGD (Stochastic Gradient Descent) и SVRG (Stochastic Variance Reduced Gradient). Тестирование нужно будет провести на следующих двух моделях из области машинного обучения: 1) логистическая регрессия; 2) автокодировщик (глубокая нейронная сеть). В первом случае оптимизируемая функция является выпуклой, во втором — невыпуклой. Один из вопросов, на который Вам нужно будет ответить по окончании выполнения задания — есть ли существенная эмпирическая разница в поведении методов SGD/SVRG на выпуклой и невыпуклой задачах?

2 Необходимая теория

2.1 Задача минимизации суммы функций

Рассмотрим задачу безусловной минимизации суммы функций:

$$\min_{x \in \mathbf{R}^d} \left\{ f(x) := \frac{1}{n} \sum_{i=1}^{n} f_i(x) \right\}, \tag{1}$$

где $f_i : \mathbf{R}^d \to \mathbf{R}$ — дифференцируемые функции (не обязательно выпуклые).

Будем считать, что о структуре функций f_i, кроме того что они являются достаточно гладкими, ничего не известно (т. е. f_i заданы в виде «черного ящика»). Все, что доступно, — это *инкрементальный оракул первого порядка* — специальная процедура, которая по заданному индексу $i \in \{1, \ldots, n\}$ и точке $x \in \mathbf{R}^d$ возвращает $(f_i(x), \nabla f_i(x))$ — значение и градиент функции f_i в точке x. Задача состоит в том, чтобы найти приближенное решение (1) за как можно меньшее число вызовов оракула.

2.2 Методы минимизации суммы функций

Опишем два популярных метода для решения задачи (1).

2.2.1 Метод SGD

Метод SGD является рандомизированным аналогом метода градиентного спуска. Итерация метода имеет следующий вид:

$$x_{k+1} = x_k - h_k g_k,$$

где g_k — стохастический градиент функции f в точке x_k, т. е. случайный вектор, удовлетворяющий свойству несмещенности $\mathbf{E}\, g_k = \nabla f(x_k)$, а h_k — детерминированная положительная длина шага.

Типичным выбором стохастического градиента в задаче (1) является $g_k = \nabla f_{i_k}(x_k)$, где индекс i_k генерируется случайно (независимо на каждой итерации) из равномерного распределения на $\{1, \ldots, n\}$. В этом

случае сложность итерации метода SGD в n раз меньше, чем сложность итерации метода градиентного спуска, поскольку требуется совершить лишь один вызов оракула, а не n. (Тем не менее, важно понимать, что при этом число итераций, которые нужно выполнить методу SGD для достижения заданной точности, может быть на порядки больше, чем соответствующее число итераций для метода градиентного спуска.)

Существует несколько стратегий для выбора длин шагов h_k, а также формирования выходной точки в методе SGD. В данном задании Вам предлагается использовать постоянную длину шага $h_k \equiv h$, а в качестве выходной точки брать среднее всех x_k. Такая схема строго обоснована для выпуклой оптимизации; насколько хорошо она будет работать в невыпуклом случае — это Вам предлагается проверить экспериментально.

Псевдокод метода SGD приведен в алгоритме 1.

Алгоритм 1 Метод SGD

Вход: Начальная точка x_0; число итераций K; длина шага h.

1: **for** $k \leftarrow 0$ **to** $K-1$ **do**
2: Выбрать $i_k \sim \mathrm{Unif}\{1, \ldots, n\}$
3: $x_{k+1} \leftarrow x_k - h\nabla f_{i_k}(x_k)$
4: **end for**

Выход: $\frac{1}{K+1}\sum_{k=0}^{K} x_k$

2.2.2 Метод SVRG

Основная идея метода SVRG заключается в выборе «более хорошего» стохастического градиента для функции (1), чем просто $\nabla f_{i_k}(x_k)$. Действительно, несмотря на то, что оценка $\nabla f_{i_k}(x_k)$ является несмещенной, ее дисперсия $\mathbf{E}\,\|\nabla f_{i_k}(x_k) - \nabla f(x_k)\|_2^2$ может быть достаточно большой и никак не уменьшаться с увеличением числа итераций k.

Для уменьшения дисперсии стохастического градиента в методе SVRG используется следующий прием. Фиксируется некоторая точка $\tilde{x}$, представляющая собой текущее приближенное решение задачи (1). Далее выполняется m итераций метода SGD (где m — параметр метода, обычно порядка $2n$), запущенного из точки $x_0 = \tilde{x}$, в котором в качестве стохастического градиента вместо $\nabla f_{i_k}(x_k)$ используется

$$g_k := \nabla f_{i_k}(x_k) - \nabla f_{i_k}(\tilde{x}) + \nabla f(\tilde{x}). \tag{2}$$

За счет того, что $\mathbf{E}\,\nabla f_{i_k}(\tilde{x}) = \nabla f(\tilde{x})$, вектор g_k по-прежнему является несмещенной оценкой полного градиента $\nabla f(x_k)$. Дополнительное слагаемое $-\nabla f_{i_k}(\tilde{x}) + \nabla f(\tilde{x})$ призвано уменьшить дисперсию этой оценки. По окончании m итераций точка $\tilde{x}$ изменяется на новое приближенное решение — результат работы метода SGD (в нашем случае это $\frac{1}{m+1}\sum_{k=0}^{m} x_k$), — и процесс повторяется снова (происходит перезапуск). В итоге, благодаря таким перезапускам, дисперсия $\mathbf{E}\,\|g_k - \nabla f(x_k)\|_2^2$ постепенно уменьшается и в пределе достигает нуля.

Таким образом, метод SVRG представляет собой двухуровневую схему (см. алгоритм 2). На внешних итерациях (называемых *стадиями*) происходит обновление точек $\tilde{x}$ и подсчет полных градиентов $\nabla f(\tilde{x})$. На внутренних итерациях выполняются шаги метода SGD, использующего «скорректированную» оценку стохастического градиента (2).

Заметим, что общее число вызовов оракула внутри одной стадии метода SVRG составляет $n + 2m$. Если значение m выбрано порядка n, то «средняя сложность итерации» метода SVRG лишь в небольшое число раз превосходит соответствующую сложность в методе SGD (например, если $m = 2n$, то это число равно $(n + 4n)/(2n) = 2.5$).

Обсудим выбор длины шага h в методе SVRG. Согласно теоретическому анализу, h следует выбирать порядка $0.1/L_f$, где L_f — константа Липшица для градиента функции f. В определенных ситуациях (например, в задачах минимизации эмпирического риска типа логистической регрессии) константу L_f вычислить несложно. Однако для более сложных функций эта информация обычно недоступна, и здесь

Алгоритм 2 Метод SVRG

Вход: Начальная точка x_0; число стадий S; продолжительность стадии m; длина шага h.

$\tilde{x}^1 \leftarrow x_0$
for $s \leftarrow 1$ **to** S **do**
  $\tilde{g}^s \leftarrow \frac{1}{n}\sum_{i=1}^n \nabla f_i(\tilde{x}^s)$
  $x_0^s \leftarrow \tilde{x}^s$
  for $k \leftarrow 0$ **to** $m-1$ **do**
    Выбрать $i_k^s \sim \mathrm{Unif}\{1,\dots,n\}$
    $g_k^s \leftarrow \nabla f_{i_k^s}(x_k^s) - \nabla f_{i_k^s}(\tilde{x}^s) + \tilde{g}^s$
    $x_{k+1}^s \leftarrow x_k^s - hg_k^s$
  end for
  $\tilde{x}^{s+1} \leftarrow \frac{1}{m+1}\sum_{k=0}^m x_k^s$
end for

Выход: $\tilde{x}^{S+1}$

желательно иметь специальную процедуру динамической регулировки оценки константы L_f. К сожалению, никаких теоретически обоснованных процедур, осуществляющих эту регулировку, для метода SVRG не существует. Тем не менее, здесь можно предложить следующую эвристику, которая основана на схеме Нестерова для подбора соответствующей оценки в методе градиентного спуска (см. Практическое задание 3). Пусть уже имеется некоторая оценка L константы L_f, и на текущей итерации был выбран индекс i_k. Сначала выполняется «пробный» шаг $x \leftarrow x_k - hg_k$ метода SVRG с длиной шага $h = 0.1/L$. Далее вычисляется значение функции f_{i_k} в новой точке x и проверяется, соответствует ли оно тому, что предсказывает оценка Липшица для функции f_{i_k} в точках x_k и x. Если нет, то константа L удваивается, и процесс повторяется. Если да, то новая точка x принимается ($x_{k+1} \leftarrow x$), а константа L уменьшается в $2^{1/m}$ раз (чтобы максимальное уменьшение за одну стадию — аналог одной итерации градиентного спуска — составило 2). Псевдокод метода SVRG с адаптивным подбором константы Липшица приведен в алгоритме 3. (Насколько хорошо эта эвристическая процедура работает на практике — Вам предстоит выяснить при выполнении этого задания.)

2.3 Модели логистической регрессии и автокодировщика

2.3.1 Двухклассовая логистическая регрессия

Логистическая регрессия является стандартной моделью в задачах классификации. Для простоты рассмотрим лишь случай бинарной классификации. Неформально задача формулируется следующим образом. Имеется обучающая выборка $((a_i, b_i))_{i=1}^n$, состоящая из n векторов $a_i \in \mathbf{R}^d$ (называемых *признаками*) и соответствующих им чисел $b_i \in \{-1, 1\}$ (называемых *классами*). Нужно построить алгоритм $b(\cdot)$, который для произвольного нового вектора признаков a автоматически определит его класс $b(a) \in \{-1, 1\}$.

В модели логистической регрессии определение класса выполняется по знаку линейной комбинации компонент вектора a с некоторыми фиксированными коэффициентами $x \in \mathbf{R}^d$:

$$b(a) := \mathrm{sign}(a^\top x).$$

Коэффициенты x являются параметрами модели и настраиваются с помощью решения следующей оптимизационной задачи:

$$\min_{x \in \mathbf{R}^d} \left\{ \frac{1}{n}\sum_{i=1}^n \ln(1 + \exp(-b_i a_i^\top x)) + \frac{\lambda}{2}\|x\|_2^2 \right\}, \tag{3}$$

где $\lambda > 0$ — коэффициент регуляризации (параметр модели). Чтобы представить задачу (3) в форме (1), достаточно задать функции f_i следующим образом: $f_i(x) := \ln(1 + \exp(-b_i a_i^\top x)) + \frac{\lambda}{2}\|x\|_2^2$.

Алгоритм 3 Метод SVRG с адаптивным подбором константы Липшица

Вход: Начальная точка x_0; число стадий S; продолжительность стадии m; начальная оценка константы Липшица L_0.

$\tilde{x}^1 \leftarrow x_0$
$L \leftarrow L_0$
for $s \leftarrow 1$ **to** S **do**
  $\tilde{g}^s \leftarrow \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\tilde{x}^s)$
  $x_0^s \leftarrow \tilde{x}^s$
  for $k \leftarrow 0$ **to** $m-1$ **do**
    Выбрать $i_k^s \sim \mathrm{Unif}\{1,\dots,n\}$
    $g_k^s \leftarrow \nabla f_{i_k^s}(x_k^s) - \nabla f_{i_k^s}(\tilde{x}^s) + \tilde{g}^s$
    repeat
      $x \leftarrow x_k^s - (0.1/L) g_k^s$
      if $f_{i_k^s}(x) > f_{i_k^s}(x_k^s) + \nabla f_{i_k^s}(x_k^s)^\top (x - x_k^s) + \frac{L}{2}\|x - x_k^s\|_2^2$ **then**
        $L \leftarrow 2L$
      end if
    until $f_{i_k^s}(x) \le f_{i_k^s}(x_k^s) + \nabla f_{i_k^s}(x_k^s)^\top (x - x_k^s) + \frac{L}{2}\|x - x_k^s\|_2^2$
    $x_{k+1}^s \leftarrow x$
    $L \leftarrow \max\{L_0, L/2^{1/m}\}$
  end for
  $\tilde{x}^{s+1} \leftarrow \frac{1}{m+1}\sum_{k=0}^{m} x_k^s$
end for

Выход: $\tilde{x}^{S+1}$

2.3.2 Автокодировщик

Автокодировщик — это специальная нейронная сеть, используемая для обучения представлений, обычно с целью сокращения размерности. Неформально задача обучения представлений состоит в том, чтобы по заданной обучающей выборке $(a_i)_{i=1}^n$, где $a_i \in \mathbf{R}^p$ (признаковое описание i-го объекта), построить кодирующую функцию (алгоритм) $\phi : \mathbf{R}^p \to \mathbf{R}^{p'}$, выполняющей преобразование признаков (кодирование).

Опишем как работает автокодировщик. Обозначим число слоев автокодировщика через s, а число нейронов на k-м слое через d_k ($1 \le k \le s$). (Согласно дизайну $d_1 = d_s = p$.) Матрицы весов обозначим через $X_k \in \mathbf{R}^{d_{k+1} \times d_k}$ ($1 \le k \le s-1$), а полный список параметров автокодировщика (набор из $s-1$ матриц) через $X := (X_1, \dots, X_{s-1})$. Пусть на вход подается вектор $a \in \mathbf{R}^p$. Тогда значения $z_k(a; X) \in \mathbf{R}^{d_k}$ на слоях автокодировщика вычисляются последовательно по следующим рекуррентным формулам для $k = 1, \dots, s-1$:

$$\begin{aligned} z_1(a; X) &:= a, \\ u_{k+1}(a; X) &:= X_k z_k(a; X), \\ z_{k+1}(a; X) &:= \sigma_{k+1}(u_{k+1}(a; X)) \end{aligned}$$

Здесь $\sigma_{k+1} : \mathbf{R}^{d_{k+1}} \to \mathbf{R}^{d_{k+1}}$ — некоторые (гладкие) функции (называемые *активациями*), которые осуществляют (нелинейные) преобразования входов. Вектор $z_s(a; X)$ с самого последнего слоя называется выходом.

Параметры X автокодировщика настраиваются с целью сделать выход $z_s(a; X)$ похожим на вход a. Для этого решается следующая оптимизационная задача:

$$\min_{\substack{X=(X_1,\dots,X_{s-1}) \\ X_k \in \mathbf{R}^{d_{k+1} \times d_k}}} \frac{1}{2n} \sum_{i=1}^{n} \|z_s(a_i; X) - a_i\|_2^2. \tag{4}$$

Как только параметры X настроены, в качестве кодирующей функции $\phi(\cdot)$ можно взять любую из функций

$z_k(\cdot; X)$ с любого промежуточного слоя $k \in \{2, \dots, s-1\}$.

Чтобы переписать задачу (4) в более традиционной форме, перейдем от минимизации по набору матриц $X = (X_1, \dots, X_{s-1})$ к минимизации по одной векторной переменной $x \in \mathbf{R}^d$, где $d := \sum_{k=1}^{s-1} d_k d_{k+1}$ — общее число параметров автокодировщика. Для этого нужно ввести некоторое взаимно-однозначное отображение между множествами $\bigtimes_{k=1}^{s-1} \mathbf{R}^{d_{k+1} \times d_k}$ и $\mathbf{R}^d$. Зададим это отображение с помощью последовательной группировки матриц X_k в один (длинный) вектор:

$$\operatorname{pack}(X) := (\operatorname{vec}(X_1), \dots, \operatorname{vec}(X_{s-1})) \in \mathbf{R}^d,$$

где $\operatorname{vec}(\cdot)$ — операция вытягивания матрицы в вектор[1]. Легко видеть, что $\operatorname{pack}(\cdot)$ действительно задает взаимно-однозначное отображение: для любого $x \in \mathbf{R}^d$ всегда найдется, и при том единственный, набор матриц $X \in \bigtimes_{k=1}^{s-1} \mathbf{R}^{d_{k+1} \times d_k}$ такой, что $\operatorname{pack}(X) = x$. Таким образом, к операции $\operatorname{pack}(\cdot)$ существует обратная операция; обозначим ее через $\operatorname{unpack}(\cdot)$. В итоге, задачу (4) можно переписать эквивалентным образом в следующем виде:

$$\min_{x \in \mathbf{R}^d} \frac{1}{2n} \sum_{i=1}^{n} \|z_s(a_i; \operatorname{unpack}(x)) - a_i\|_2^2. \tag{5}$$

Положив $f_i(x) := \frac{1}{2}\|z_s(a_i; \operatorname{unpack}(x)) - a_i\|_2^2$, получим представление задачи (5) в виде (1).

3 Формулировка задания

1. Запрограммировать методы SGD и SVRG.

   Замечание: Обратите внимание, что в качестве выходных точек в методах SGD/SVRG используются средние поисковых точек x_k, а не сами эти точки. Именно эти средние и нужно передавать в качестве `x_out` в процедуру `logger.record_point` (см. раздел 5.1). (В методе SVRG логирование нужно производить на каждой итерации внутреннего цикла.)

2. Воспроизвести примеры с логистической регрессией и автокодировщиком из раздела 5.

3. Рассмотрим следующие конфигурации эксперимента:

   (a) **Тестовая функция:** Логистическая регрессия (функционал (3)) на наборах данных *w5a* (пример из раздела 5.2) и *a9a* из LIBSVM[2]. Коэффициент регуляризации стандартный: $\lambda = 1/n$.
   Начальная точка: $x_0 = 0$.

   (b) **Тестовая функция:** Автокодировщик (функционал (5)) на (нормированных) данных *digits* из scikit-learn с двумя вариантами архитектуры:

      i. $s = 3,\ d_1 = d_3 = 64,\ d_2 = 2,\ \sigma_2(x) = x,\ \sigma_3(x) = \operatorname{sigm}(x)$ (пример из раздела 5.3).

      ii. $s = 5,\ d_1 = d_5 = 64,\ d_2 = d_4 = 30,\ d_3 = 2,\ \sigma_3(x) = x,\ \sigma_2(x) = \sigma_4(x) = \sigma_5(x) = \operatorname{sigm}(x)$,

   где $\operatorname{sigm}(x) := 1/(1 + e^{-x})$ — сигмоидная функция. (Подразумевается, что все функции $\sigma_i(\cdot)$ действуют поэлементно, как в примере из раздела 5.3.)
   Начальная точка: $x_0 \sim \mathcal{N}(0, I_d)$ (случайная точка из стандартного нормального распределения).

   Для каждой из этих конфигураций:

   (a) Построить графики сходимости[3] метода SGD в зависимости от длины шага h. (Длину шага перебрать по экспоненциальной сетке.)

[1] Если $A \in \mathbf{R}^{m \times n}$, то $\operatorname{vec}(A)$ — это вектор размера mn, составленный из столбцов матрицы A. Например, если $A \in \mathbf{R}^{2 \times 3}$, то $\operatorname{vec}(A) = (a_{11}, a_{21}, a_{12}, a_{22}, a_{13}, a_{23}) \in \mathbf{R}^6$.

[2] `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

[3] Под *графиками сходимости* подразумеваются 1) зависимость значения (полной) функции от эпохи; 2) зависимость (логарифма) нормы градиента (полной функции) от эпохи.

(b) Нарисовать на одном графике метод SGD (с лучшей длиной шага) против SVRG.

(c) Построить графики сходимости метода SVRG в зависимости от параметра m (число итераций внутреннего цикла).

Какие выводы можно сделать?

4. Написать отчет с описанием всех проведенных исследований.

4 Прототипы реализуемых функций

Методы оптимизации, которые Вам необходимо реализовать, будут работать с функционалами вида (1) с помощью объекта `fsum` (абстрактного) класса `FuncSum`. Объект `fsum` содержит внутри себя поле `n_funcs`, хранящее число функций n, а также метод `call_ith(i, x)`, который принимает на вход число `i` (типа `int`) и вектор `x` (типа `np.ndarray` размера d) и возвращает пару `(fi, gi)`, где `fi` (типа `float`) — значение функции $f_{i+1}(x)$, а `gi` (типа `np.ndarray` размера d) — соответствующий градиент $\nabla f_{i+1}(x)$.

Замечание: Обратите внимание, что в методе `call_ith` используется сдвинутая нумерация по сравнению с текстом этого задания: чтобы получить функцию f_1 нужно использовать индекс `i=0`, функцию f_2 — индекс `i=1` и т. д.

От Вас требуется реализовать следующие функции:

1. Метод SGD (алгоритм 1):

Модуль:	`optim`
Функция:	`sgd(fsum, x0, n_iters=1000, step_size=0.1, trace=False)`

Параметры:	`fsum: FuncSum`
	Оптимизируемая функция, объект класса `FuncSum` (см. выше).
	`x0: np.ndarray`
	Начальная точка $x_0 \in \mathbf{R}^d$.
	`n_iters: int`, опционально
	Число итераций K метода.
	`step_size: float`, опционально
	Длина шага h.
	`trace: bool`, опционально
	Отслеживать прогресс метода для возврата истории или нет.

Возврат:	`x_out: np.ndarray`
	Приближенное решение (результат работы метода).
	`hist: dict`, возвращается только если `trace=True`
	История процесса оптимизации. Словарь со следующими полями:
	`epoch: np.ndarray`
	Номер эпохи: {суммарное число вызовов оракула} / {число функций n}.
	`f: np.ndarray`
	Значение полной функции.
	`norm_g: np.ndarray`
	(Бесконечная) норма градиента полной функции.

	Значения полей в истории должны быть согласованы: `hist['f'][j]` и `hist['norm_g'][j]` равны соответственно значению и норме градиента полной функции в текущем приближенном решении `x_out`, полученным спустя `hist['epoch'][j]` эпох.

2. Метод SVRG с адаптивным подбором длины шага (алгоритм 3):

Модуль:	`optim`
Функция:	`svrg(fsum, x0, n_stages=10, n_inner_iters=None, tol=1e-4, trace=False, L0=1)`

Параметры:	`fsum: FuncSum`
	Оптимизируемая функция, объект класса `FuncSum` (см. выше).
	`x0: np.ndarray`
	Начальная точка $x_0 \in \mathbf{R}^d$.
	`n_stages: int`, опционально
	Число стадий S.
	`n_inner_iters: int` или `None`, опционально
	Число внутренних итераций m. Если `None`, то взять равным $2n$.
	`tol: float`, опционально
	Точность для критерия остановки: если $\|\tilde{g}\|_\infty < \mathrm{tol}$, то выход.
	`trace: bool`, опционально
	Отслеживать прогресс метода для возврата истории или нет.
	`L0: float`, опционально
	Начальная оценка константы Липшица.

Возврат:	`x_out: np.ndarray`
	Приближенное решение (результат работы метода).
	`hist: dict`, возвращается только если `trace=True`
	История процесса оптимизации. Полностью аналогично `hist` в `sgd` (см. выше).

5 Примеры

5.1 Формирование истории внутри методов

Вместе с текстом этого задания также выдается файл `logutils.py`, содержащий (уже реализованные) вспомогательные классы `FuncSumWrapper` и `Logger`. Эти классы удобно использовать внутри методов оптимизации для отслеживания их прогресса (и формирования истории). Рассмотрим как это делается.

В самом начале кода метода оптимизации нужно написать следующие строчки:

```
from logutils import FuncSumWrapper, Logger

fsum = FuncSumWrapper(fsum)
logger = Logger(fsum)
```

В результате этого переменная `fsum` (оптимизируемая функция) перезапишется на новый объект класса `FuncSumWrapper`. Этот новый объект полностью повторяет функционал исходного объекта (т. е. содержит поле `n_funcs` и метод `call_ith`) за исключением того, что теперь при каждом вызове метода `call_ith`

счетчик суммарного числа вызовов оракула автоматически увеличивается на один. Также здесь создается новый вспомогательный объект `logger`, который будет осуществлять логирование (формирование истории).

Для того, чтобы записать в историю очередную точку `x_out` (типа `np.ndarray`), используется команда

```
logger.record_point(x_out)
```

Метод `record_point` работает следующим образом. Во-первых, проверяется, сколько прошло вызовов оракула с момента последней записи в историю. Если достаточно (что определяется параметром `record_freq` внутри `logger`), то выполняется вычисление значения и градиента полной функции в точке `x_out` и происходит добавление записи в историю. В противном случае (если последняя запись была недавно) ничего не происходит. Таким образом, несмотря на то, что команда `record_point` может вызываться очень часто (например, после каждого вызова оракула, как в методе SGD), в действительности запись в историю происходит гораздо реже (по умолчанию один раз в эпоху). (Такое поведение требуется из соображений эффективности. Поскольку вычисление полной функции занимает достаточно большое время по сравнению со временем работы одной итерации методов SGD/SVRG, то схема, в которой история действительно обновляется на каждой итерации, работала бы «вечность».)

Наконец, чтобы извлечь из `logger` историю метода (уже сформированную в соответствии с требованиями к прототипам), нужно воспользоваться методом `get_hist`:

```
hist = logger.get_hist()
```

5.2 Логистическая регрессия

Файл `logistic.py` содержит процедуры для работы с логистической регрессией.

Пример запуска методов SGD и SVRG на данных *w5a* из LIBSVM:

```
from sklearn.datasets import load_svmlight_file
from logistic import LossFuncSum
from optim import sgd, svrg

# Load training data
A_train, b_train = load_svmlight_file('w5a')

# Create function object
fsum = LossFuncSum(A_train, b_train, reg_coef=1/A_train.shape[0])

# Set up starting point
x0 = np.zeros(A_train.shape[1])

# Run SGD
x_sgd, hist_sgd = sgd(fsum, x0, n_iters=10*fsum.n_funcs, step_size=0.01, trace=True)

# Run SVRG
x_svrg, hist_svrg = svrg(fsum, x0, n_stages=2, trace=True)
```

Оценить качество полученной точки `x_sgd` можно с помощью тестовой выборки:

```
from logistic import predict_labels

# Load testing data
A_test, b_test = load_svmlight_file('w5a.t', n_features=A_train.shape[1])
```

```
# Define auxiliary function for evaluating accuracy
def calc_error(x):
    b_hat = predict_labels(A_test, x)
    err = np.mean(b_test != b_hat)
    return err

# Evaluate ‘x0‘ and ‘x_sgd‘
print('Initial error: %g' % calc_error(x0))
print('SGD result: %g' % calc_error(x_sgd))
print('SVRG result: %g' % calc_error(x_svrg))
```

Если все реализовано правильно, то результат должен быть примерно таким:

```
Initial error: 0.969946
SGD result: 0.103585
SVRG result: 0.103886
```

Графики сходимости можно нарисовать следующим образом:

```
# Function value vs epoch
plt.figure()
plt.plot(hist_sgd['epoch'], hist_sgd['f'], linewidth=4, label='SGD')
plt.plot(hist_svrg['epoch'], hist_svrg['f'], linewidth=4, label='SVRG')
plt.xlabel('Epoch')
plt.ylabel('Function value')
plt.grid()
plt.legend()

# Gradient norm vs epoch
plt.figure()
plt.semilogy(hist_sgd['epoch'], hist_sgd['norm_g'], linewidth=4, label='SGD')
plt.semilogy(hist_svrg['epoch'], hist_svrg['norm_g'], linewidth=4, label='SVRG')
plt.xlabel('Epoch')
plt.ylabel('Gradient norm')
plt.grid()
plt.legend()
```

5.3 Автокодировщик

Файл `autoencoder.py` содержит процедуры для работы с автокодировщиком.

Пример запуска метода SGD на данных *digits* из scikit-learn:

```
from sklearn.datasets import load_digits
from scipy.special import expit
from autoencoder import LossFuncSum, n_params_total
from optim import sgd

# Load data
digits = load_digits()
A = digits['data']
labels = digits['target']
```

```
# Normalize values into segment [0, 1]
A = A / np.max(A)

# Define (element-wise) identity function and its derivative
linfun = (lambda x: x)
dlinfun = (lambda x: np.ones_like(x))
# Define (element-wise) sigmoid function and its derivative
sigmfun = expit
dsigmfun = (lambda x: expit(x) * (1 - expit(x)))
# Describe autoencoder architecture
arch = {
    'n_layers': 3,                 # Number of layers: s
    'sizes': [64, 2, 64],          # Layer sizes: d_1, d_2, d_3
    'afuns': [linfun, sigmfun],    # Activation functions: sigma_2, sigma_3
    'dafuns': [dlinfun, dsigmfun], # Derivatives of act. functions: sigma_2', sigma_3'
}

# Create function object
fsum = LossFuncSum(A, arch)

# Set up starting point
np.random.seed(0)
x0 = np.random.randn(n_params_total(arch))

# Run SGD
x_sgd = sgd(fsum, x0, n_iters=10*fsum.n_funcs, step_size=0.1)
```

Оценить качество полученной точки `x_sgd` можно следующим образом:

```
from autoencoder import compute_vals, unpack

# Define auxuiliary function for plotting results of autoencoder
def plot_result(x, arch, title, digits=[3, 4, 7], colors=['r', 'b', 'g']):
    X_list = unpack(x, arch)
    Z_list = compute_vals(A, X_list, arch)[0]

    Z = Z_list[arch['n_layers']//2] # middle layer

    plt.figure()
    hs = []
    for dig, col in zip(digits, colors):
        mask = (labels == dig)
        h = plt.scatter(Z[0, mask], Z[1, mask], color=col, s=50, alpha=0.5)
        hs.append(h)
    plt.title(title)
    plt.legend(hs, ['Digit %s'%dig for dig in digits], scatterpoints=1)

# Plot results
plot_result(x0, arch, 'Initial approximation')
plot_result(x_sgd, arch, 'SGD result')
```

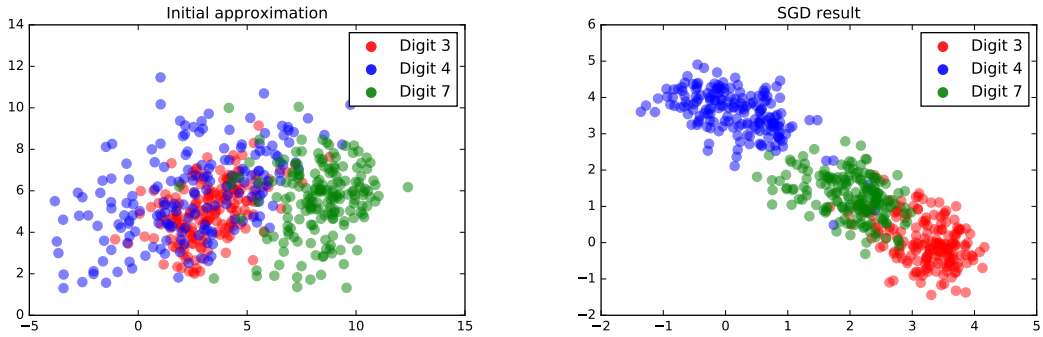

Рис. 1: Результат кодирования векторов длины 64 из набора данных *digits* в векторы размера 2 (точки на плоскости). Слева: параметры равны x_0 (никакой оптимизации); справа: параметры найдены с помощью оптимизации методом SGD.

Если все реализовано правильно, то Вы должны получить картинки, похожие на рис. 1.

6 Оформление задания

Результатом выполнения задания являются 1) отчет в формате PDF с описанием проведенных исследований и 2) файл `optim.py`, содержащий исходные коды методов SGD и SVRG. Выполненное задание следует отправить письмом по адресу bayesml@gmail.com с заголовком

[BMK MOMO16] Задание 4, Фамилия Имя