

Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации

Отчет по лабораторной работе №2

⟨Собрано 30 апреля 2023 г.⟩

Работу выполнили:

Бактурин Савелий Филиппович М32331

Вереня Андрей Тарасович М32331

Сотников Максим Владимирович М32331

Преподаватель:

ТВА

Стохастический градиентный спуск

Исследование с разными размерами батча

Реализуйте стохастический градиентный спуск для решения линейной регрессии. Исследуйте сходимость с разным размером батча (1 – SGD, 2, ..., $n - 1$ – Minibatch GD, n – GD из предыдущей работы).

Стохастический градиентный спуск

Стохастический градиентный спуск — модификация к основному методу итерационного поиска минимума через антиградиент дифференцируемой функции в рассматриваемой плоскости \mathbb{R}^n . Идея: пусть есть множество M – какой-то полный набор данных вычисленных оценок при спуске к минимуму, тогда в рассматриваемой версии мы будем брать случайное значение из выбранного $M' \subset M$. Как правило, такой подход преуменьшает вычислительные ресурсы, в особенности, следует помнить, что `float` считается крайне медленно, и ускоряет итерацию по количествам эпохам, но при этом мы теряем точность сходимости.

Пусть $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$ – координата в \mathbb{R}^n и задана функция $f(x_i) : \mathbb{R}^n \rightarrow \mathbb{R}$. Мы хотим нашу задачу свести к исследованию на некоторых специальных образцах заданной функции, для каждой точки из которых мы будем минимизировать ошибку для дальнейшего нахождения приближенного минимума рассматриваемой функции $f(x_i)$. Мы хотим найти линейную регрессию, представляющая из себя полином 1-ой степени от n переменных. Для начала мы найдем функцию ошибки S по следующей формуле:

$$S(f) = (X'^T \times X')^{-1} \times X'^T \times Y \times \vec{x},$$

где Y – матрица значений при множестве X (определение), X' – это матрица X , но в 1-ой колонке забитый единицами. По другому мы можем записать данную формулу следующим образом:

$$S(f) = \sum_{i=1}^N (y_i - x_i \cdot w_i)^2,$$

где $y_i \in D(f(x_i))$ (образ функции), $w_i \in W$ – сгенерированные веса, коэффициенты при линейной функции.

Рассмотрим идею алгоритма. При итерации, пока мы не превысили максимальное количество шагов или не сведем нашу функцию потери до некоторого ε , мы будем обобщать экспериментальные данные в виде случайных точек в некоторую многомерную линию. На каждом шаге мы будем изменять функцию потерь от измененной w .

Напишем идейный псевдокод. Скажем, что $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$ – координата в n -мерном пространстве \mathbb{R}^n , $y_i = \{y_i^0, y_i^1, \dots, y_i^{n-1}\}$ – образ функции $f(x_i)$.

```
1 function S(x, y, w):
2     return  $\sum_{i=1}^N (y_i - x_i \cdot w_i)^2$ 
3
4 function stochastic_descent(x, y):
5     w  $\leftarrow$  [ $w_i \in \mathbb{R}$ ] * n
```

```

6   prev ← INIT, предыдущее значение функции потери
7   next ← S(x, y, w), текущее значение функции потери
8   α ← const
9   while |prev − next| > ε:
10      prev ← next
11      i ← x ∈ [0, |Y|]
12      T ← [0] * n
13      ∀j ∈ |w| do
14          Tj ← (yi − xi × w) · xij
15      w ← w + α · T
16      next ← S(x, y, w)
17   return w
18

```

Minibatch градиентный спуск

Модификация *Minibatch* обобщает вариант стохастического градиентного спуска, тем, что во время итерации мы будем брать не одну случайную точку из посчитанных на предыдущем шаге и изменять функцию потерь как бы относительно её, а теперь возьмем выборку $M' \subset M$, причем, обязательно, чтобы $|M'| > 1$ и $|M'| < |M|$.

Тогда псевдокод от предыдущего рассмотренного варианта почти ничем не отличается. Скажем также, что $x_i = \{x_i^0, x_i^1, \dots, x_i^{n-1}\}$ – координата в n -мерном пространстве \mathbb{R}^n , $y_i = \{y_i^0, y_i^1, \dots, y_i^{n-1}\}$ – образ функции $f(x_i)$; значение m – выступает в роли мощности подмножества M' .

```

1  function S(x, y, w):
2      return  $\sum_{i=1}^N (y_i - x_i \cdot w_i)^2$ 
3
4  function stochastic_descent(x, y, m):
5      w ← [wi ∈ ℝ] * n
6      prev ← INIT, предыдущее значение функции потери
7      next ← S(x, y, w), текущее значение функции потери
8      α ← const
9      while |prev − next| > ε:
10         prev ← next
11         ∀i ∈ [0, m] do
12             T ← [0] * n
13             ∀j ∈ |w| do
14                 Tj ← (yi − xi × w) · xij
15             w ← w + α · T
16         next ← S(x, y, w)
17     return w
18

```

Градиентный спуск

Наконец, самый общий случай и являющийся самым быстроходным среди двух рассмотренных ранее, благодаря тому, что мы учитываем все точки $M' \equiv M$. Данный метод работает крайне медленно, но является одним из самых быстрых в сходимости

к приближенной точке минимума.

Пусть m – есть мощность множества M , тогда идейным псевдокодом-решением задачи будет являться тот же код, что и для предыдущего варианта, то есть

```
1 function S(x, y, w):
2     return  $\sum_{i=1}^N (y_i - x_i \cdot w_i)^2$ 
3
4 function stochastic_descent(x, y, m):
5     w  $\leftarrow$  [wi  $\in \mathbb{R}$ ] * n
6     prev  $\leftarrow$  INIT, предыдущее значение функции потери
7     next  $\leftarrow$  S(x, y, w), текущее значение функции потери
8      $\alpha \leftarrow$  const
9     while |prev - next| >  $\varepsilon$ :
10         prev  $\leftarrow$  next
11          $\forall i \in [0, m]$  do
12             T  $\leftarrow$  [0] * n
13              $\forall j \in |w|$  do
14                 Tj  $\leftarrow$  (yi - xi  $\times$  w)  $\cdot$  xij
15             w  $\leftarrow$  w +  $\alpha \cdot T$ 
16     next  $\leftarrow$  S(x, y, w)
17     return w
18
```

Learning rate scheduling

Задачей мы поставим подбор функции изменения шага, чтобы улучшить сходимость из предыдущего пункта. Тогда, мы использовали самый простой способ – *константный*, однако у него есть недостаток: иногда шаг в `const_learning_rate` может «перепрыгнуть» через минимум и/или начать «прыгать» через него бесконечно много раз, в таком случае мы хотим, чтобы шаг был уменьшен. Другой случай: когда такой шаг может привести к долгому ожиданию, пока алгоритм не дойдет до минимума. Вот тут и возникает задача подбора такой функции изменения шага, чтобы алгоритм за какое-то конечное k эпох добрался до минимума быстрее. В качестве такой мы возьмем *экспоненциальную функцию*. Экспоненциальная функция, в общем случае, для изменения сходимости выглядит так:

$$\text{learning_rate} = \text{start_learning_rate} \cdot e^{-k \cdot \text{epoch}},$$

где `learning_rate` – текущий размер шага, `start_learning_rate` – стартовая длина, k – некий параметр, который может зависеть от размера выбранного батча, и, наконец, `epoch` – текущий номер эпохи.

Исследование различных модификаций

Исследуйте модификации градиентного спуска (Nesterov, Momentum, AdaGrad, RMSProp, Adam).

Momentum

Пусть нам дана функция $f(x_i)$, где $x_i = \{x_i^1, x_i^2, \dots, x_i^n\}$ – координата точки x_i в n -мерном пространстве – и пусть у данной f есть множество локальных точек минимума, образующийся «впадиной» функции, и «горы» – максимумов. Рассмотрим некоторый объект \mathfrak{D} , который обладает свойством «скольжения» по поверхности нашей функции f , определяемый следующим образом:

- а) если он «сходит» с горы, то он не останавливается при «схождении» с локального максимума;
- б) если он «перескакивает» локальный минимум, то он либо остановится на идеально гладкой части поверхности функции f , либо он «сойдет» к локальному минимуму.

Идея модификации градиентного спуска *Momentum* в создании алгоритма, обладающий свойством \mathfrak{D} в имени свойства импульса.

Рассмотрим последовательность $\{u_i\}$, которую мы назовем «скоростью», скажем, что $\{g_i\} \equiv \{-\nabla f(x_i)\}$, тогда определим скорость следующим образом:

$$v_{i+1} = \lambda \cdot v_i + (1 - \lambda) \cdot g_i, \quad \lambda \in (0, 1)$$

и зададим изменение на $i + 1$ -ой итерации наших весов точек:

$$w_{i+1} = w_i - \alpha \cdot v_{i+1}, \quad \alpha = \text{const}$$

Nesterov

Модификация *Nesterov* по своей сути почти ничем не отличается ранее рассмотренного Momentum: единственное, что отделяет братьев по крови, так это то, что Nesterov считает градиент не в текущей точке, на которой мы стоим, а той, куда мы бы могли пойти, следуя импульсу. Тогда, наши основные формулы немного меняются, для скорости:

$$v_{i+1} = \lambda \cdot v_i + (1 - \lambda) \cdot g(w_i - \alpha \cdot \lambda \cdot v_i), \quad \alpha = \text{const}$$

И для весов:

$$w_{i+1} = w_i - \alpha \cdot v_{i+1}$$

AdaGrad

Рассмотрим некий `start_learning_rate` и стандартный метод стохастического спуска. Как мы уже видели, одним из недостатков такого метода является появления слишком больших или, наоборот, маленьких компонент относительно друг друга. Для исправления мы введём матрицу Ω , которую определим следующим образом:

$$\Omega = \sum_{j=1}^i g_j \times g_j^T,$$

где $g_j = \nabla f(x_j)$. Теперь рассмотрим основную формулу и поделим часть, где идет «шаг» алгоритма, на $\Omega_{i,i}$:

$$w_{i+1} = w_i - \frac{\alpha \cdot g_j}{\sqrt{\Omega_{i,i}}}$$

Теперь же, там, где мы потенциально делаем большие шаги, мы делим на большое $\Omega_{i,i}$ и таким образом уменьшаем количество больших скачков. Тут мы получаем глобальную проблему в виде потенциально слишком высокой скорости уменьшения шага.

RMSProp

Модификация *RMSProp* решает возникшую проблему с AdaGrad, в качестве решения было предложено усреднять значения. Пусть $(g_i)^2$ – покомпонентное возведение в квадрат, δ – некое число порядка $[10^{-10}, 10^{-7}]$, дабы по формуле не было деления на ноль. Тогда, в качестве скорости возьмем:

$$s_{i+1} = \lambda \cdot s_i + (1 - \lambda) \cdot (g_i)^2, \quad \lambda \in (0, 1)$$

Наконец, изменение весов:

$$w_{i+1} = w_i - \alpha \cdot \frac{g_i}{\sqrt{s_{i+1} + \delta}}, \quad \alpha = \text{const}$$

Adam

Начинаем объединение весов предыдущих. В отличие от всех его предшественников модификация *Adam* является наиболее распространенным и используемым.

$$\begin{aligned} v_{i+1} &= \lambda_1 \cdot v_i + (1 - \lambda_1) \cdot g_i \\ s_{i+1} &= \lambda_2 \cdot s_i + (1 - \lambda_2) \cdot (g_i)^2 \\ \hat{v}_{i+1} &= \frac{v_{i+1}}{1 - \lambda_1^{i+1}} \\ \hat{s}_{i+1} &= \frac{s_{i+1}}{1 - \lambda_2^{i+1}} \end{aligned}$$

Наконец, изменение весов будем рассчитывать по следующей формуле:

$$w_{i+1} = w_i - \lambda \cdot \frac{\hat{v}_{i+1}}{\sqrt{\hat{s}_{i+1} + \delta}},$$

где $\lambda_1 = 0.9$, $\lambda_2 = 0.999$ и $\delta = 10^{-8}$.

Сравнение модификаций

Сходимость

Траектории

Полиномиальная регрессия

1. Реализуйте полиномиальную регрессию. Постройте графики восстановленной регрессии для полиномов разной степени.
2. Модифицируйте полиномиальную регрессию добавлением регуляризации в модель (L1, L2, Elastic регуляризации).
3. Исследуйте влияние регуляризации на восстановление регрессии.

Введение

Представим себе некую функцию $f : \mathbb{R} \rightarrow \mathbb{R}$ причем такую, что она не зависит от каких-то побочных факторов (случайные значения и тому подобное), и при этом получаемые экспериментальные данные были как бы «разбросаны» относительно некой обобщающей кривой. Нашей задачей состоит в понимании, что это за кривая – по сути, это множество пар $M = \{\langle x, y \rangle : x \in \mathbb{R}, y \in E(f(x))\}$, которые при переходе от заданного $x' \rightarrow x''$ по некоторому δ , подает предугадываемое значение y'' на основе полученных ранее начальных экспериментальных данных. Говоря простыми словами, с помощью данной прямой мы хотим получать прогноз.

Рассмотрим идею *полиномиальной регрессии*. Пусть даны $x_i \in X$ и $y_i \in Y$. Тогда уравнением полинома будет иметь следующий вид в нашей задаче:

$$y_i = \sum_{j=0}^k a_j \cdot x_i^j,$$

где $\{a\}$ – это коэффициенты полинома, где a_0 , их мы ищем ровно тем же способом, что и в первой задаче – по методу наименьших квадратов. Наконец, по аналогии с задачей градиентного спуска, мы получаем следующее выражение:

$$L = \omega \cdot \left\| \sum_{i=1}^n (\mathbf{y}_i - y_i)^2 \right\| \rightarrow \min,$$

где $\{\mathbf{y}\}$ – это значения полинома в точках $\{x\}$, тогда наше выражение преобразуется:

$$L = \omega \cdot \left\| \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i)^2 \right) \right\| \rightarrow \min,$$

где $\omega = \frac{1}{n}$, для того чтобы усреднить квадратичные ошибки на всех обучающих примерах. Это позволяет получить более сбалансированную и интерпретируемую метрику ошибки, не зависящую от размера обучающей выборки. Для получения градиента исследуемой функции мы воспользуемся производной от L по некоторому коэффициенту a_i , тогда

$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i) \cdot x_i^i \right)$$

Исследование различных модификация

Как и в задаче с градиентным спуском нам бы хотелось получать более реальные «гладкие» оценки предсказания значения, вместо имения «гор» и «впадин» кривой. Эта проблема может быть решена с помощью модификаций к методу, которые, по своей сути, представляют из себя отдельные слагаемые.

L1

Идея *L1 регуляризации*. Давайте с каждой новой итерации мы будем неким способом «наказывать» нашу модель за высокие веса, чтобы в будущем увеличить обобщающую способность. Эта регуляризация основана на добавлении еще одного слагаемого в основную сумму, равного абсолютному значению коэффициентов полинома.

$$L = \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i)^2 \right) + \lambda \cdot \sum_{i=0}^k (|a_i|),$$

где λ – есть общий коэффициент для регуляризации влияния штрафов на модель. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i) \cdot x_i^i \right) + \lambda$$

L2

Идея *L2 регуляризации*. Данная модификация является небольшим дополнением к предыдущему в виде большей гладкости производной и возможностью лучшей работой с градиентным спуском. Проблема L1 заключалась в том, что в некоторых местах, где производная не может быть установлена, имеет резкие «горы», «впадины» и вообще разрывы. Тогда, на помощь приходит немного иное штрафное значение функции – вместо суммы значений коэффициентов полинома мы возьмем квадраты. Отличает этот подход от L1 тем, что теперь мы штрафует больше, но при этом никогда не зануляемся, как это было с предыдущей модификацией.

$$L = \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i)^2 \right) + \lambda \cdot \sum_{i=0}^k (a_i^2),$$

где λ – есть общий коэффициент для регуляризации влияния штрафов на модель. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_i} = 2 \cdot \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i) \cdot x_i^i \right) + \lambda \cdot a_i$$

Elastic регуляризация

Идея *Elastic регуляризация*. Объединение двух предыдущих идей, как и полезных особенностей, свойств обоих методов, при этом, корреляция переменных такая, что

данная регуляризация не обнуляет некоторые из них, как в случае с L1.

$$L = \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i)^2 \right) + \lambda_1 \cdot \sum_{i=0}^k (a_i^2) + \lambda_2 \cdot \sum_{i=0}^k (a_i^2),$$

где λ_1 и λ_2 – есть коэффициенты для регуляризации влияния штрафов L1 и L2 на модель соответственно. Градиентом для модели будет выступать тогда такая формула:

$$\frac{\partial L}{\partial a_{\mathbf{i}}} = 2 \cdot \omega \cdot \sum_{i=1}^n \left(\sum_{j=0}^k (a_j \cdot x_i^j - y_i) \cdot x_{\mathbf{i}}^j \right) + \lambda_1 + \lambda_2 \cdot a_{\mathbf{i}}$$