XNumPy

Отдел перспективных исследований
АО НПК Криптонит
xnumpy@kryptonite.ru

23 октября 2023 г.

1 Введение

Библиотека XNumPy расширяет широко распространённую NumPy и дополняет её классами для работы с числами типа float с учётом числа *точных битов*. Потеря точности возникает фактически при всех операциях со значениями с плавающей точкой. Это связано с тем, что значения 64-битных (и любых других) значений с плавающей точкой неточно представляют вещественные числа, а операции с ними приближают точные результаты, но не всегда им равны.

Простейшим известным примером неточности операций с плавающей точкой является существование *машинного ноля*:

```
>>> 1.0 + 1e-16 == 1.0
True
```

Проблема неточности значений с плавающей точкой имеет общий характер и начинается уже с обработки литералов:

```
>>> 0.1 + 0.2 == 0.3
False
```

Неточности возникают из-за следующих особенностей представления вещественных чисел в виде двоичных чисел с плавающей точкой:

- представимо конечное множество чисел, в то время как множество вещественных чисел бесконечно (точность представления зависит от числа бит в мантиссе числа с плавающей точкой, а диапазон представимых чисел — от числа бит в его порядке),
- операции сложения и умножения неассоциативны, то есть в общем случае

$$a + (b + c) \neq (a + b) + c$$
$$a \cdot (b \cdot c) \neq (a \cdot b) \cdot c$$

  при работе с числами с плавающей точкой, существуют ошибки округлений младших разрядов.

Таким образом, неточность в числах с плавающей точкой присутствует практически всегда.

При операциях с такими числами неточные знаки распространяются и появляются в результатах вычислений. Наиболее ярко это проявляется при вычитании близких чисел. В этом случае количество неточных битов возрастает особенно быстро, и иногда достаточно единственной операции, чтобы точных знаков практически не осталось.

Большинство операций не увеличивают точность (кроме вряд ли осмысленной операции умножения результата вычисления на нуль).

Наличие неточных битов в ряде случаев может привести к их распространению по мантиссе числа, превращая её в «цифровой шум», в результате чего на часть полученных значений станет нельзя опираться и делать из них любые выводы. Потеря точности может возникать как по причине неточных начальных данных, так и по причине накопления ошибок из-за очень большого числа операций, что особенно актуально для длительных вычислений, таких, например, как обучение нейросетей или моделирование длительных процессов.

Показано[1], что цифровые шумы, как правило, рано или поздно приводят к полной потере точности при обучении нейросетей. Это является дополнительным аргументом к тому, чтобы контролировать количество точных битов, иначе сложно иметь представление о том, насколько можно доверять результатам вычислений.

При разработке многих вычислительных библиотек учитывается такой параметр, как *численная устойчивость* алгоритмов, характеризующая степень влияния цифровых шумов аргументов на результат вычисления. Однако, также широко распространены алгоритмы, которые имеют пониженную численную устойчивость относительно явного вычисления по определению, но при этом дающие очень существенный прирост производительности. Примером является умножение матриц, реализованное в библиотеке `OpenBLAS`, используемое для умножения матриц в `NumPy`.

При разработке сложных алгоритмов, имеющих внутри много потенциально численно неустойчивых операций, есть риск потерять значимость и начать интерпретировать цифровой шум вместо осмысленных величин. Таким образом, оказывается важным отслеживать накапливаемые неточности от операции к операции. Библиотека XNumPy служит для реализации этой цели.

В библиотеке XNumPy вводятся новые классы:

- `xf64` — числовой класс, содержащий `float64` (aka `double` in `C/C++` or `f64` in `Rust`) и количество точных битов (`uint8`).
- `xf64array` — класс массива (аналогичный `numpy.ndarray` с `dtype=numpy.float64`).

При работе с этими классами вычисление точных битов происходит автоматически. В большинстве случаев, где по умолчанию `numpy` конструи-

[1] https://www.ijainn.latticescipub.com/wp-content/uploads/papers/v3i1/A1061123122.pdf

рует массив с типом данных `float64`, тип по умолчанию заменён на `xf64`. Соответственно, создаётся не `numpy.ndarray`, а `xnumpy.xf64array`. Таким образом, в большинстве случаев требуемые изменения в коде (помимо замены импортируемой библиотеки с NumPy на XNumPy) минимальны.

При печати данных типа `xf64` и `xf64array` на месте неточных цифр отображаются вопросительные знаки.

В версии `1.0.0` библиотеки XNumPy перегружено 288 наиболее употребительных методов из 378 методов библиотеки NumPy. В следующих версиях набор перегруженных методов планируется дополнять.

2 Использование новых классов

Создать явно `xf64` можно при помощи конструкторов:

```
>>> xf64(42)
42 with 54 exact bits (approx 16 digits)
>>> xf64.with_precision(42, 4)
42 with 4 exact bits (approx 1 digits)
```

Точность указывается **в битах, а не в десятичных цифрах**.

Тип `xf64` может быть приведён к `float` и `int` стандартным способом. Получить количество точных битов можно с помощью метода `exact_bits`:

```
>>> a = xf64.with_precision(1.1, 8)
>>> b = xf64.with_precision(2.2, 8)
>>> a + b
3.3000000000000003 with 8 exact bits (approx 2 digits)
>>> (a + b).exact_bits()
8
```

Для массива `xf64array` значения типа `float64` можно получить как свойство `values`, возвращающее `ndarray` той же формы. Свойство `exact_bits` позволяет получить массив значений точных битов элементов (тип значений `numpy.uint8`).

3 Форматированный вывод

При печати неточные знаки заменяются вопросительными знаками:

```
>>> np.pi
3.141592653589793
>>> pi_ = xf64.with_precision(3.14, 12)
>>> print(pi_)
3.14
>>> print(f"{pi_:.5f}")
3.14???
```

Аналогично для массивов:

```
>>> arr = np.array([
...     [xf64.with_precision(np.pi, i)] for i in range(0, 53, 4)
... ])
>>> print(arr)
[[ ?.????????????????e+00]
,[ 3.????????????????e+00]
,[ 3.1???????????????e+00]
,[ 3.14??????????????e+00]
,[ 3.141?????????????e+00]
,[ 3.14159?????????e+00]
,[ 3.141592????????e+00]
,[ 3.1415926???????e+00]
,[ 3.14159265??????e+00]
,[ 3.141592653?????e+00]
,[ 3.14159265358???e+00]
,[ 3.1415926535897?e+00]
,[ 3.14159265358979e+00]]
>>> arr.exact_bits
array([[ 0],
       [ 4],
       [ 8],
       [12],
       [16],
       [20],
       [24],
       [28],
       [32],
       [36],
       [40],
       [44],
       [48],
       [52]], dtype=uint8)
```

Точность в XNumPy хранится в битах двоичного представления числа, которые не отображаются однозначно на цифры десятичного представления результата, поэтому на печати вопросами отображаются *гарантированно неточные десятичные цифры*.

Аналогичный подход (см. далее) применяется и при вычислении числа неточных битов, биты, маркированные как «неточные», являются гарантированно неточными.

4 Сохранение и загрузка

Функции `save, load, savez, savez_compressed` перегружены и действуют следующим образом:

- функция `save` получает путь/дескриптор и объект,
  - если объект не является объектом расширения, вызывается `numpy.save` от тех же аргументов,
  - если объект является строкой или путём, к нему добавляется суффикс `.npy`, если он отсутствует,
  - значения сохраняются по указанному пути, точности сохраняются по пути с добавленным суффиксом `.prec` (в дополнение к `.npy`),
  - если переданы объект расширения и что-то другое (например, дескриптор, полученный из `open(filename, "wb")`), функция кидает ошибку.
- функция `savez` работает аналогично:
  - если среди переданных объектов есть объекты расширения и передан не имя файла или путь, кидается ошибка,
  - имя дополняется суффиксом `.npz` при его отсутствии,
  - объекты сохраняются словарём при помощи `numpy.savez`, от объектов расширения хранятся значения,
  - для точностей создаётся дополнительный файл с именем, дополненным суффиксом `.prec`, куда при помощи `numpy.savez` сохраняются точности объектов расширения (для других ничего не сохраняется туда).
- функция `savez_compressed` работает аналогично `savez`,
- функция `load` работает в обратном порядке:
  - загружает из файла содержимое с помощью `numpy.load`,
  - смотрит, есть ли рядом файл с дополнительным суффиксом `.prec` (если нет), возвращает содержимое как есть,
  - если файл рядом есть, читает и дополняет объекты данными о точности.

Таким образом, эти функции работают совместимым образом. Чтобы не потерять данные о точности при передаче файлов, важно передавать их вместе с файлами, содержащими точность!

5 Примеры

Библиотека снабжена примерами использования (см. директорию `./examples` репозитория). Примеры демонстрируют, что существенная потеря точности может случаться даже в таких, вроде бы простых случаях, как решение квадратного уравнения, вычисление тригонометрических функций большого аргумента и вычисление определителя небольшой матрицы.

6 Предостережения для пользователей

Версии `1.0.*` не подразумевают учёт точности для работы со следующими группами функций.

6.1 astype

Метод `astype` класса `numpy.ndarray` реализован в `NumPy` и не принимает аргумент `xf64`, используйте другие способы приведения типа, например, `xnumpy.asarray` с ключевым аргументом `dtype=xf64`.

6.2 Произведение матриц и оператор `@`

При перемножении матриц типа `ndarray` (при отсутствии среди них `xf64array`) для вычисления точностей произведения следует использовать не бинарный оператор `@`, а функцию `xnumpy.matmul`, так как оператор `@` ссылается на код `numpy` и не возвращает `xf64array`.

Аналогично следует использовать функции вместо операторов при применении к `numpy.ndarray` вместе с переменными типа `xf64`, так как методы класса `numpy.ndarray` определяют переменные типа `xf64` как имеющие для массива тип данных `object` вместо `xf64` и потому могут некорректно обрабатывать точность и отбрасывать данные о точности при дальнейших операциях с полученным массивом.

6.3 Многочлены

Класс `np.poly` и алгоритмы для работы с ним не перегружены в версии `1.0`.

6.4 Комплексные числа

Оптимальная структура для комплексных чисел на данный момент не выбрана. Для работы с комплексными числами рекомендуются использовать "овеществлённые"версии алгоритмов, то есть сводить работу к парам массивов вещественной и мнимой части.

6.5 Методы на основе `np.unique`

Методы на основе `np.unqiue` основаны на сравнении чисел с плавающей точкой на равенство, что представляется малоосмысленным. Для сравнения чисел с плавающей точкой с дополнением ещё и количеством точных битов выбор набора уникальных чисел из массива окончательно теряет всякий смысл, поэтому такие методы выдают ошибку.

Использование таких методов на `xf64` приводит к ошибке, а в целом для значений с плавающей точкой использование таких методов крайне не рекомендуется.

6.6 float32

Учёт точности в версии `1.0.*` происходит для `float64` и не происходит для `float32`.

6.7 Низкоуровневые средства

Не рекомендуется пытаться работать с `xf64array` низкоуровневыми средствами, такими как указатели (что возможно, например, при написании расширений на `Cython`), так как внутренне `xf64array` устроен не как массив значений типа `xf64`. Это может быть допустимо для `numpy.ndarray`, но не для `xf64array`.

7 Установка

```
pip install xnumpy-1.0.1*.whl
pip install xnumpy_base-1.0.1*.whl
```

8 Обоснования

8.1 Определение ошибок

В этом параграфе мы приведём некоторые пояснения способа вычисления погрешностей. Эти алгоритмы и оценки хорошо известны (учебник Бахвалова по численным методам). Для полноты изложения всё же приведём некоторые обозначения.

Пусть дана вещественная переменная x. Обозначим её приближение через $\hat{x}$. Тогда *абсолютная погрешность* равна $\Delta x = |x - \hat{x}|$, *относительная погрешность* равна δ_x, где $\dfrac{\Delta x}{x} = 1 + \delta_x$.

$f(x)$	$\mathcal{C}_f(x)$
$x + a$	$\frac{x}{x+a}$
ax	1
$1/x$	1
x^n	$\lvert n \rvert$
$\ln(x)$	$\left\lvert \frac{1}{\ln x} \right\rvert$
$\cos x$	$\lvert x \tan x \rvert$

Таблица 1: Примеры чисел обусловленности

8.2 Распространение ошибок

Пусть задано приближение $\hat{x}$ вещественного числа x и требуется найти приближение $\hat{y}$ значения $y = f(x)$. Для оценки абсолютно погрешности обычно используют приближение

$$|\Delta y| = |\hat{y} - y| = |f(x + \Delta x) - f(x)| \approx \left| \frac{\partial f(x)}{\partial x} \right|_{x=\hat{x}} \cdot |\Delta x|.$$

Для относительной погрешности получаем оценку

$$\left| \frac{\hat{y} - y}{\hat{y}} \right| = \left| \frac{\hat{x}}{f(\hat{x})} \cdot f'(\hat{x}) \right|.$$

Величина $\left| \frac{\hat{x}}{f(\hat{x})} \cdot f'(\hat{x}) \right|$. называется *числом обусловленности* $\mathcal{C}_f(\hat{x})$ функции f в точке $\hat{x}$. Эта величина соответствует коэффициенту распространения ошибки функции $f(x)$ в точке $\hat{x}$. Это означает, что относительная ошибка величины δ в $\hat{x}$ приводит к относительной ошибке величины $\delta \cdot \mathcal{C}_f(\hat{x})$ в приближении $f(\hat{x})$.

Легко проверить, что если $\hat{x}$ имеет s точных битов, то $f(\hat{x})$ имеет $s - \log_2 \mathcal{C}_f(\hat{x})$ точных битов.

Таким образом мы можем вычислить количество точных битов для функций одной переменной. Оценка погрешностей функций многих переменных нетривиальна, и мы ограничиваем её снизу. Таким образом, биты, обозначенные как неточные, гарантированно содержат информационный шум. Это значит, что ошибки не меньше найденных.

8.3 Распространение ошибок в функциях многих аргументов

Для перегрузки методов `NumPy` нам не потребуются произвольные функции двух и более аргументов. Разберём несколько явных случаев:

- Сумма. При сложении нескольких чисел абсолютная разряд абсолютной погрешности суммы равен максимуму разрядов абсолютных погрешностей аргументов.

- Произведение. При перемножении нескольких чисел разряд относительной погрешности произведения равен максимуму разрядов относительных погрешностей аргументов, что соответствует минимуму из количеств точных битов аргументов.

- Минимум и максимум. При вычислении минимума нескольких чисел значение (`float64`) получается как минимум значений, а точность (`uint8`) как минимум точностей тех из аргументов, которые в точной части совпадают с минимумов значений. Для максимума аналогично.

  Таким образом, минимум из набора значений типа `xf64` может не оказаться равным ни одному из значений, что отличает `xf64` от прочих числовых типов.

Хотя умножение матриц в `NumPy` (выполняемое при помощи библиотеки `BLAS`) даёт результат, отличающийся от получаемого буквальным вычислением по формуле, это вычисление сильно оптимизировано под конкретную целевую платформу, что может влиять на алгоритм вычисления так, что результат будет меняться. Поэтому, по крайней мере в начальных версиях библиотеки `XNumPy`, мы ограничиваем точность снизу тем значением, которое соответствует формуле произведения матриц, а не получается композицией вычисления точностей для меньших элементарных операций.

В общем случае мы стараемся, чтобы вычисление точных битов выполнялось для наименьших элементарных операций, а не больших блоков функций, так как иначе мы потеряем часть информации о результирующей точности при конкретном алгоритме реализации. Однако в hardware dependent cases мы будем вместо элементарных операций вычислять значение точности для блоков, деление которых на меньшие части будет меняться.

Так, в случае умножения матриц могут использоваться SIMD-инструкции процессора (такие, как SSE и AVX2), наличие которых зависит от типа процессора. Это может влиять на последовательность сложений в суммировании и в итоге может таким образом влиять на сам результат суммирования по причине неассоциативности сложения `float64`.

9 Планируется в следующих версиях

В следующих версиях планируется:

- добавить оценку точности быстрого преобразования Фурье,
- поддержку комплексных чисел, а также оценок точности для `float32` и `float16`,
- добавить перегруженные методы библиотек `scipy` и `sklearn`,
- поддержку вычислений с оценкой точности на GPU.

Вопросы, замечания и предложения можно направлять по адресу xnumpy@kryptonite.ru.