ММП 2025/2026

Викладач Канцедал Георгій Олегович

Основні особливості DataFrame

- Однопоточність обробки даних
- Використання «широких» дата типів
- Великий функціонал
- Обмеженість роботи з великими файлами

На противагу цім як позитивним так і негативним якостям є декілька варіантів бібліотек для роботи з двохмірними даними. Хоча вони і мають менший функціонал для аналітики але більш придатні для роботи з великими даними. Зокрема Tensorflow, PyTorch, Jax.

Аналоги для матричних операцій. Порівняння

На процесорі відміність між бібліотеками виникає зарахунок власне алгоритмів роботи з оперативною памятю. Так Джакс виявився менш придатним до операцій траспонування – яка є однією з найпростіших серед матричних, в той час як зі скаладними дійсно векторними операціями дійсно отримані непогані результати. Для матричних операцій без використання вектоних функцій рекомендую використовувати Тензорфлоу.

				на процесорі		
	[JAX] Size: 1000	[TORCH] Size: 1	[TF] Size: 10000	ошибка JAX	ошибка TORCH	ошибка TF
A @ x	0.00011	0.06584	**0.01712**	0.00017	0.03574	0.00615
A @ A	0.00004	9.58179	18.29044	0.00004	0.89274	6.20058
A.T	**0.00011**	**0.00005**	**0.07268**	0.00015	0.00008	0.00679
A^-1	0.00003	28.21943	64.33344	0.00004	1.22017	0.65505
A + A	0.00005	0.12034	**0.03307**	0.00006	0.04826	0.00247
				ВЫВОДЫ		

Аналоги для матричних операцій. Порівняння

Для наступного порівняння було використано лише куду (тобто відеокарту). Тензорфлоу виявився надзвичайно вибагливим до обєму памяті і нежаль не зміг відпрацювати коректно. Однак Торч і Джакс мають меншу потребу до відео памяті і тому на них тест пройшов. Результати свідчать про повне домінування джаксі з операціями матричного множення і обернення. В той час як пайторч виявився більш придатним для простіших вектоних операцій. Це пітверджує висновки поперднього слайду.

	[JAX] Size: 10000x10000	[TORCH] Size: 10000x1000	[TF] Size: 10000x10000
A @ x	0.05568	**0.0182**	все вмерло
A @ A	**0.15301**	3.65884	все вмерло
A.T	0.00675	**0.00002**	все вмерло
A^-1	**0.63435**	6.07527	все вмерло
A + A	**0.00551**	0.06468	все вмерло

Аналоги для матричних операцій. Порівняння

Окрема ворто відмітити можливість Джаксу використовувати XLA ядра що недоступні при використанні куди і зазвичай зустрічаються в професійних відокартах для рендерингу та проведенню симуляцій. Очікувано тести виявили незначне покращення порівняно з кудою.

JAX	на куде			
	JAX	JAX XLA	ошибка JAX	ошибка JAX XLA
A @ x	0.00526	**0.00424**	0.0399	**0.02704**
A @ A	**0.10278**	0.10292	**0.06933**	0.06947
A.T	**0.0226**	0.05102	**0.27915**	0.48115
A^-1	0.51259	**0.48646**	**0.28203**	0.40107
A + A	0.00736	**0.00726**	0.04811	**0.04772**

Базові матричні операції з JAX

JAX — це бібліотека Python для високопродуктивних обчислень на масивах з підтримкою автоматичного диференціювання, JIT-компіляції та паралельного виконання на CPU, GPU та TPU.Вона поєднує знайомий API NumPy з потужними трансформаціями програм, що дозволяє ефективно реалізовувати складні обчислювальні задачі.

```python
import jax.numpy as jnp

# Створення матриць
A = jnp.array([[1, 2], [3, 4]])
B = jnp.array([[5, 6], [7, 8]])

# Матриця + Матриця
print("A + B =\n", A + B)

# Матриця * Матриця (елементно)
print("A * B =\n", A * B)

# Матричне множення
print("A @ B =\n", A @ B)

# Транспонування
print("A.T =\n", A.T)
```

JIT-компіляція для прискорення

@jit автоматично компілює функцію, підвищуючи продуктивність на GPU або TPU.

```
from jax import jit

# Функція множення двох матриць
@jit
def matmul_jit(x, y):
    return x @ y

# Перевіримо
result = matmul_jit(A, B)
print("JIT A @ B =\n", result)
```

Автоматичне диференціювання

@jit автоматично компілює функцію, підвищуючи продуктивність на GPU або TPU.

```
def sigmoid(x):
    return 1 / (1 + jnp.exp(-x))

def loss_fn(params, x, y):
    preds = sigmoid(jnp.dot(x, params))
    return -jnp.mean(y * jnp.log(preds) + (1 - y) * jnp.log(1 - preds))

from jax import grad

grad_loss = grad(loss_fn)
```

```
from jax import grad

# Скалярна функція: визначимо її як функцію від елементів матриці
def loss_fn(x):
    return jnp.sum((x @ x.T)**2)

# Обчислимо градієнт по x
grad_fn = grad(loss_fn)
G = grad_fn(A)

print("Градієнт функції по A =\n", G)
```

Векторизація з vmap

vmap дозволяє застосовувати функцію до кожного рядка без явного циклу.

```
from jax import vmap

# Приклад: нормалізація рядків матриці
def normalize_row(row):
    return row / jnp.linalg.norm(row)

# Векторизована версія
normalize_matrix = vmap(normalize_row, in_axes=0)

normalized = normalize_matrix(A)
print("Нормалізована матриця A =\n", normalized)
```

Векторизація з vmap

vmap дозволяє застосовувати функцію до кожного рядка без явного циклу.

```
from jax import vmap

# Приклад: нормалізація рядків матриці
def normalize_row(row):
    return row / jnp.linalg.norm(row)

# Векторизована версія
normalize_matrix = vmap(normalize_row, in_axes=0)

normalized = normalize_matrix(A)
print("Нормалізована матриця A =\n", normalized)
```

Зовнішні колбеки в JAX

Зовнішні колбеки дозволяють JAX виконувати Python-код на хості під час виконання програми, навіть у контексті JAX-трансформацій, таких як jit(), vmap() та grad().

Основні типи колбеків у JAX:

- jax.pure_callback: для чистих функцій без побічних ефектів.
- jax.experimental.io_callback: для функцій з побічними ефектами, таких як введення/виведення.
- jax.debug.callback: для налагодження та виводу інформації під час виконання.

```
import jax

@jax.jit
def f(x):
  y = x + 1
  print("intermediate value: {}".format(y))
  return y * 2

result = f(2)
```

```
intermediate value: Traced<~int32[]>with<DynamicJaxprTrace>
```

```
@jax.jit
def f(x):
  y = x + 1
  jax.debug.print("intermediate value: {}", y)
  return y * 2

result = f(2)
```

```
intermediate value: 3
```

jax.pure_callback — чисті колбеки

- jax.pure_callback дозволяє викликати чисту Python-функцію host_fn під час виконання JAX-програми. Функція повинна бути детермінованою та без побічних ефектів.
- jax.ShapeDtypeStruct використовується для вказівки форми та типу даних результату колбеку.

```
import jax
import jax.numpy as jnp
import numpy as np

def f_host(x):
  # call a numpy (not jax.numpy) operation:
  return np.sin(x).astype(x.dtype)

def f(x):
  result_shape = jax.ShapeDtypeStruct(x.shape, x.dtype)
  return jax.pure_callback(f_host, result_shape, x, vmap_method='sequential')

x = jnp.arange(5.0)
f(x)
```

```
Array([ 0.       ,  0.841471 ,  0.9092974,  0.14112  , -0.7568025],      dtype=float32)
```

```
jax.jit(f)(x)
```

```
Array([ 0.       ,  0.841471 ,  0.9092974,  0.14112  , -0.7568025],      dtype=float32)
```

experimental.io_callback

- На відміну від jax.pure_callback(), функція jax.experimental.io_callback() **призначена спеціально для використання з нечистими функціями**, тобто такими, що **мають побічні ефекти**.
- Наприклад, ось колбек, який використовує глобальний генератор випадкових чисел з NumPy на стороні хоста. Це **нечиста операція**, оскільки побічним ефектом генерації випадкового числа в NumPy є **зміна внутрішнього стану генератора випадкових чисел**.

```
from jax.experimental import io_callback
from functools import partial

global_rng = np.random.default_rng(0)

def host_side_random_like(x):
  """Generate a random array like x using the global_rng state"""
  # We have two side-effects here:
  # - printing the shape and dtype
  # - calling global_rng, thus updating its state
  print(f'generating {x.dtype}{list(x.shape)}')
  return global_rng.uniform(size=x.shape).astype(x.dtype)

@jax.jit
def numpy_random_like(x):
  return io_callback(host_side_random_like, x, x)

x = jnp.zeros(5)
numpy_random_like(x)
```

```
generating float32[5]
```

```
Array([0.6369617 , 0.26978672, 0.04097353, 0.01652764, 0.8132702 ],      dtype=float32)
```

jax.debug.callback — налагодження

- jax.debug.callback використовується для налагодження та виводу інформації під час виконання JAX-програми. Це дозволяє вставляти точки виводу без порушення трансформацій JAX.
- Зверніть увагу, що колбеки можуть бути дубльовані або опущені під час оптимізації, тому використовуйте цей метод лише для налагодження.

```python
import jax
import jax.numpy as jnp

def debug_fn(x):
    print("Debug: x =", x)

@jax.jit
def f(x):
    jax.debug.callback(debug_fn, x)
    return x * 2

result = f(jnp.array([1., 2., 3.]))
```

Висновок

- Jax потужний
- Інколи непогано оптимізовує швидкість
- Не передбачає інтеграцію з не джаксовими матрицями
- Не бажано змішувати джакс і не джакс
- Бажано писати великі шматки джаксу і компінлювати його
- Бажано не використовувати колбеки крім дебагу

End