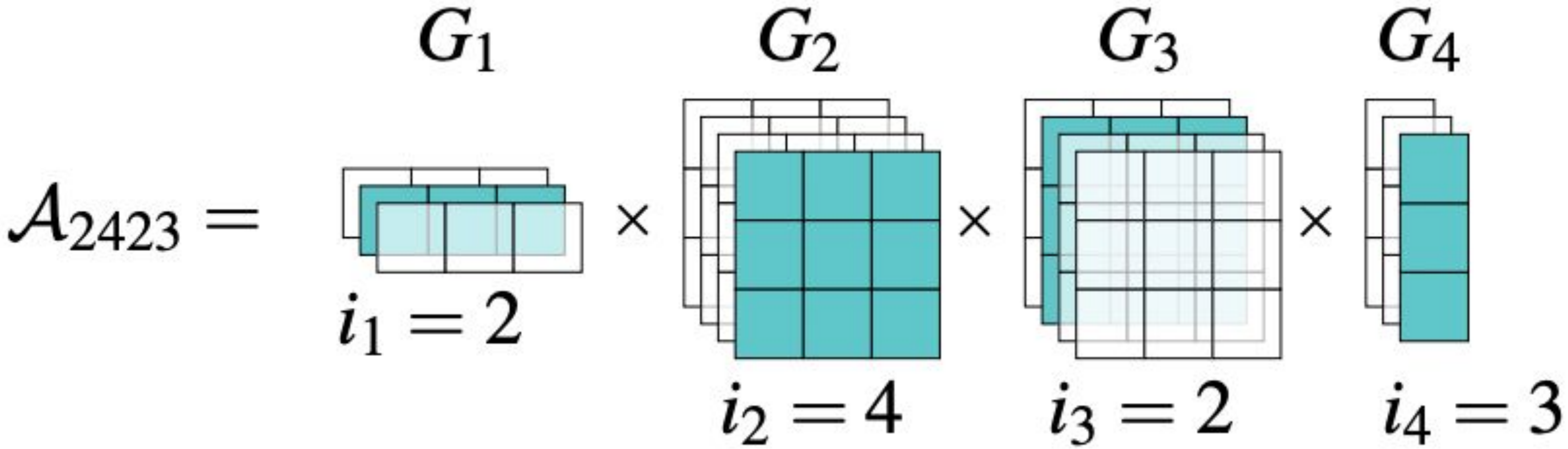
Tensor Train decomposition with Jax mindset

Alexander Novikov, Dmitry Belousov, Ivan Oseledets

Tensor Train decomposition (aka MPS)

Allows to represent a big tensor as a product of small factors



Tensor Train decomposition (aka MPS)

Allows to represent a big tensor as a product of small factors

Supports arithmetic:

Can build factors of (A + B) or (A * B) or A.dot(B) from factors of A and B without ever materializing the full tensor

Tensor Train decomposition (aka MPS)

Allows to represent a big tensor as a product of small factors

Supports arithmetic:

Can build factors of (A + B) or (A * B) or A.dot(B) from factors of A and B without ever materializing the full tensor

TT-rank controls the compression, and it increases after operations

Jax

A library for linear algebra, automatic differentiation, and machine learning

Supports ~any numpy functions

Jax is functional (i.e. functions can not have side effects)

Most operations are decorators (transformations of functions)

Allows to use cluster of GPUs / TPUs easily

Jax example

```
def f(x):
  return x**2


f_prime = jax.grad(f)


f_prime(1)  # returns 2
```

Jax example 2

```
import jax.numpy as jnp


def f(x):
  values = jnp.linalg.svd(x, compute_uv=False)
  return jnp.sum(values)


f_prime = jax.grad(f)
```

TTAX basics

Tensor Train implemented on Jax

TTAX basics

Tensor Train implemented on Jax

```
import ttax
seed = jax.random.PRNGKey(42)
tt_matrix = ttax.random.matrix(seed, ((2, 3, 4), (5, 6, 7)), tt_rank=10)

tt_matrix
```

```
shape (24, 210), tt_rank 10
```

TTAX basics

Tensor Train implemented on Jax

```
import ttax
seed = jax.random.PRNGKey(42)
tt_matrix = ttax.random.matrix(seed, ((2, 3, 4), (5, 6, 7)), tt_rank=10)

tt_matrix
```

```
shape (24, 210), tt_rank 10
```

```
tt_vector = ttax.random.matrix(seed, ((5, 6, 7), (1, 1, 1)), tt_rank=3)

tt_vector
```

```
shape (210, 1), tt_rank 3
```

TTAX basics

```
import ttax
seed = jax.random.PRNGKey(42)
tt_matrix = ttax.random.matrix(seed, ((2, 3, 4), (5, 6, 7)), tt_rank=10)

tt_matrix
```

```
shape (24, 210), tt_rank 10
```

```
tt_vector = ttax.random.matrix(seed, ((5, 6, 7), (1, 1, 1)), tt_rank=3)

tt_vector
```

```
shape (210, 1), tt_rank 3
```

```
tt_product = tt_matrix @ tt_vector

tt_product
```

```
shape (24, 1), tt_rank 30
```

Power iteration

```python
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Power iteration

What if matrix is 10^10 x 10^10
but has structure?

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Rakhuba, Maxim, Alexander Novikov, and Ivan Oseledets. "Low-rank Riemannian eigensolver for high-dimensional Hamiltonians." Journal of Computational Physics (2019)

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5, tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(3):
  tt_vector = tt_matrix @ tt_vector
  tt_vector = (1./norm(tt_vector)) * tt_vector
```

TT power iteration

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Vanilla power iteration

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5, tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(3):
  tt_vector = tt_matrix @ tt_vector
  tt_vector = (1./norm(tt_vector)) * tt_vector
  print(tt_vector)
```

```
shape (100000, 1), tt_rank 30
shape (100000, 1), tt_rank 300
shape (100000, 1), tt_rank 3000
```

TT power iteration

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Vanilla power iteration

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5 , tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  tt_vector = tt_matrix @ tt_vector
  tt_vector = ttax.round(tt_vector, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
  print(tt_vector)
```

```
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
```

TT power iteration

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Vanilla power iteration

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5 , tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  tt_vector = tt_matrix @ tt_vector
  tt_vector = ttax.round(tt_vector, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
  print(tt_vector)
```

```
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
```

TT power iteration

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Vanilla power iteration

Can we do these two ops together more efficiently?

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5 , tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  tt_vector = tt_matrix @ tt_vector
  tt_vector = ttax.round(tt_vector, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
  print(tt_vector)
```

```
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
```

TT power iteration

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Vanilla power iteration

Can we do these two ops together more efficiently?
No

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5 , tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  tt_vector = tt_matrix @ tt_vector
  tt_vector = ttax.round(tt_vector, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
  print(tt_vector)
```

```
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
shape (100000, 1), tt_rank 3
```

TT power iteration

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Vanilla power iteration

Can we do these two ops together more efficiently?
No, but

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5, tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  intermidiate = tt_matrix @ tt_vector
  intermidiate = ttax.project(intermidiate, tt_vector)
  tt_vector = ttax.round(intermidiate, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
```

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5, tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  intermidiate = tt_matrix @ tt_vector
  intermidiate = ttax.project(intermidiate, tt_vector)
  tt_vector = ttax.round(intermidiate, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
```

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Can do together (asymptotically) faster than separately!

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5, tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  intermidiate = tt_matrix @ tt_vector
  intermidiate = ttax.project(intermidiate, tt_vector)
  tt_vector = ttax.round(intermidiate, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
```

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Can do together (asymptotically) faster than separately!

But hard to implement for every single combination like project(matmul)

Power iteration

```
tt_matrix  # Of size 10^5 x 10^5, tt_rank=10

shape = ((10, 10, 10, 10, 10), (1, 1, 1, 1, 1))
tt_vector = ttax.random.matrix(seed, shape, tt_rank=3)

for _ in range(100):
  intermidiate = tt_matrix @ tt_vector
  intermidiate = ttax.project(intermidiate, tt_vector)
  tt_vector = ttax.round(intermidiate, max_tt_rank=3)
  tt_vector = (1./norm(tt_vector)) * tt_vector
```

```
matrix  # Of size 10 x 10

vector = np.random.randn(10, 1)

for _ in range(100):
  vector = matrix @ vector
  vector = vector / np.linalg.norm(vector)
```

Can do together (asymptotically) faster than separately!

But hard to implement for every single combination like project(matmul)

So we built an einsum compiler that does this automatically

Einsum compiler for asymptotic speedups

```
def slow_project_matmul(matrix, vector):
  matvec = matrix @ vector
  return ttax.project(matvec, vector)

fast_project_matmul = ttax.fuse(slow_project_matmul)
```

Einsum compiler for asymptotic speedups

```
def slow_project_matmul(matrix, vector):
  matvec = matrix @ vector
  return ttax.project(matvec, vector)

fast_project_matmul = ttax.fuse(slow_project_matmul)
```

```
tt_matrix = ttax.random.matrix(seed, matrix_shape, tt_rank=10)
tt_vector = ttax.random.matrix(seed, vector_shape, tt_rank=10)
```

```
benchmark(slow_project_matmul, tt_matrix, tt_vector)
```

```
100 loops, best of 5: 4 ms per loop
```

```
benchmark(fast_project_matmul, tt_matrix, tt_vector)
```

```
The slowest run took 1106.56 times longer than the fastest. This could mean that an intermediate result is being cache
1 loop, best of 5: 1.84 ms per loop
```

Einsum compiler for asymptotic speedups

```
def slow_project_matmul(matrix, vector):
  matvec = matrix @ vector
  return ttax.project(matvec, vector)

fast_project_matmul = ttax.fuse(slow_project_matmul)
```

```
tt_matrix = ttax.random.matrix(seed, matrix_shape, tt_rank=20)
tt_vector = ttax.random.matrix(seed, vector_shape, tt_rank=20)
```

```
benchmark(slow_project_matmul, tt_matrix, tt_vector)
```

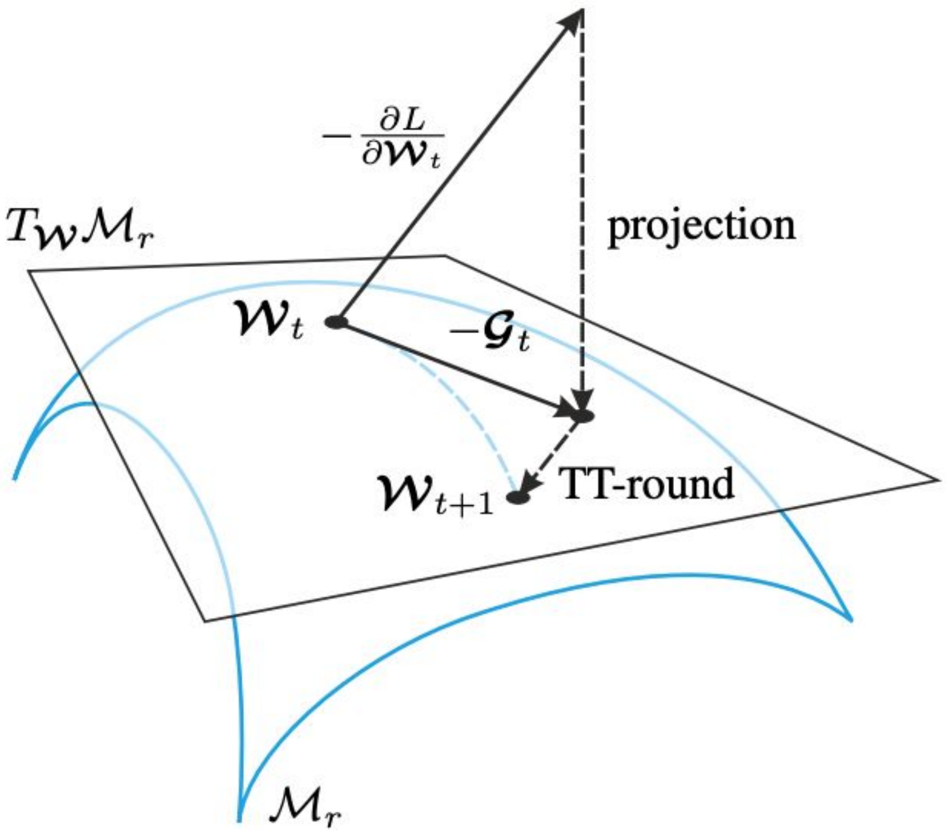
```
10 loops, best of 5: 70.1 ms per loop
```

```
benchmark(fast_project_matmul, tt_matrix, tt_vector)
```

```
100 loops, best of 5: 7.28 ms per loop
```

Riemannian optimization

$\mathcal{M}_r$ – all tensors with fixed TT-rank (say 5)



Computing Riemannian gradient

```
# Loss(x): 0.5 * <x, A x>
```

```
def rimennian_gradient(x):
  return ttax.project(A @ x, x)
```

Computing Riemannian gradient

```
# Rayleigh quotient (loss for solving eigenvalue problems): <x, A x> / <x, x>

def rayleigh_quotient(x):
  xAx = ttax.flat_inner(A @ x, x)
  norm = ttax.norm(x)
  return xAx / norm
```

```
def rimennian_gradient(x):
  Ax = A @ x
  norm = ttax.norm(x)
  coef = 2 / norm
  first = ttax.project(coef * Ax, x)
  second = coef * rayleigh_quotient(x) * x
  return first - second
```

Computing Riemannian gradient

```
# Rayleigh quotient (loss for solving eigenvalue problems): <x, A x> / <x, x>

def rayleigh_quotient(x):
  xAx = ttax.flat_inner(A @ x, x)
  norm = ttax.norm(x)
  return xAx / norm
```

If you need Riemannian Hessian-by-vector it's going to be ...

$$\nabla^2 f(\mathbf{X})\ \mathbf{Z} = \frac{2}{\langle \mathbf{X}, \mathbf{X} \rangle} \mathrm{A}\mathbf{Z} - 2\frac{f(\mathbf{X})}{\langle \mathbf{X}, \mathbf{X} \rangle}\mathbf{Z} - 4\frac{\langle \mathrm{A}\mathbf{X}, \mathbf{Z} \rangle}{\langle \mathbf{X}, \mathbf{X} \rangle^2}\mathbf{X} \\ - 4\frac{\langle \mathbf{X}, \mathbf{Z} \rangle}{\langle \mathbf{X}, \mathbf{X} \rangle^2}\mathrm{A}\mathbf{X} + 8f(\mathbf{X})\frac{\langle \mathbf{X}, \mathbf{Z} \rangle}{\langle \mathbf{X}, \mathbf{X} \rangle^2}\mathbf{X}$$

Autodiff

```
# Rayleigh quotient (loss for solving eigenvalue problems): <x, A x> / <x, x>

def rayleigh_quotient(x):
  xAx = ttax.flat_inner(A @ x, x)
  norm = ttax.norm(x)
  return xAx / norm
```

Just do this!

```
riemannian_gradient = ttax.grad(rayleigh_quotient)
riemannian_hessian_by_vector = ttax.hessian_by_vector(rayleigh_quotient)
```

Novikov, Alexander, Maxim Rakhuba, and Ivan Oseledets. "Automatic differentiation for Riemannian optimization on low-rank matrix and tensor-train manifolds." arXiv (2021)

Conclusion

- TTAX is a library for working with TT-decomposition written on Jax

- We built an einsum compiler which asymptotically speeds up your code by fusing a few operations into a single one

- We support Riemannian autodiff, which computes Riemannian gradient and Riemannian Hessian-by-vector product for an arbitrary given function with optimal asymptotics