Scientific computing and machine learning in 

Patrick Kidger
Google X

vs PyTorch?

JAX is:

- Faster for scientific computing.
- Advanced features:
  - forward-mode autodiff
  - vmap
  - stronger scientific ecosystem (=maintained)
- (Downside: sometimes trickier to use.)

vs Julia?

JAX has:

- Fewer correctness issues.
- Much more mature autodiff.
- Large-scale autoparallel.
- (Downside: smaller ecosystem.)

Numerical Python

NumPy: array-based programming model

SciPy: scientific computing operations

CuPy: GPU-backed arrays

Numba: JIT compilation

Autograd: autodifferentiation

TensorFlow: machine learning operations

PyTorch: eager execution; model-building syntax

Familiar API

JAX provides a familiar NumPy-style API.

```
import jax.numpy as jnp

def forward(params, inputs):
  for W, b in params:
    outputs = inputs @ W + b
    inputs = jnp.maximum(outputs, 0)
  return outputs

def loss(params, x, y):
  predicted_y = forward(params, x)
  return jnp.sum((predicted_y - y) ** 2)
```

Familiar API

JAX provides a familiar NumPy-style API.

Transformations

JAX includes composable function transformations for compilation, batching, autodiff, and autoparallel.

```
import jax.numpy as jnp
from jax import grad, jit

def forward(params, inputs):
  ...

def loss(params, x, y):
  ...

grad_fn = jit(grad(loss))
```

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def forward(params, inputs):
  ...

def loss(params, x, y):
  ...

grad_fn = jit(grad(loss))
perexample_grad_fn = jit(vmap(grad(loss), in_axes=(None, 0, 0)))
```

Familiar API

JAX provides a familiar NumPy-style API.

Transformations

JAX includes composable function transformations for compilation, batching, autodiff, and autoparallel.

Run Anywhere

The same code executes on multiple backends, including CPU, GPU, & TPU

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def forward(params, inputs):
  ...

def loss(params, x, y):
  ...

grad_fn = jit(grad(loss))
perexample_grad_fn = jit(vmap(grad(loss), in_axes=(None, 0, 0)))
```

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def forward(params, inputs):
  ...

def loss(params, x, y):
  ...

grad_fn = jit(grad(loss))
perexample_grad_fn = jit(vmap(grad(loss), in_axes=(None, 0, 0)))


grads = perexample_grad_fn(params, x, y)
```

Runs on a single CPU

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def forward(params, inputs):
  ...

def loss(params, x, y):
  ...

grad_fn = jit(grad(loss))
perexample_grad_fn = jit(vmap(grad(loss), in_axes=(None, 0, 0)))
sharding = jax.sharding.PositionalSharding(devices)
x, y = jax.device_put((x, y), sharding)
grads = perexample_grad_fn(params, x, y)
```

Runs on a cluster of 32 GPUs

Runs on a single CPU

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def forward(params, inputs):
  ...

def loss(params, x, y):
  ...

grad_fn = jit(grad(loss))
perexample_grad_fn = jit(vmap(grad(loss), in_axes=(None, 0, 0)))
sharding = jax.sharding.PositionalSharding(devices)
x, y = jax.device_put((x, y), sharding)
grads = perexample_grad_fn(params, x, y)
```

Runs on a cluster of ~~32 GPUs~~
...or even more!

Runs on a single CPU

Familiar API

JAX provides a familiar NumPy-style API.

Transformations

JAX includes composable function transformations for compilation, batching, autodiff, and autoparallel.

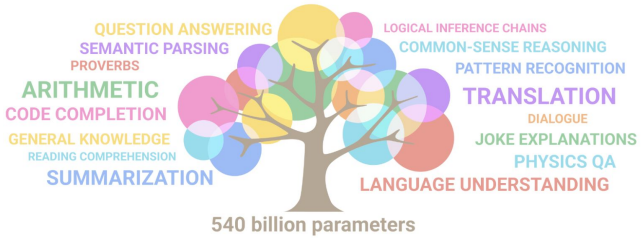
Run Anywhere

The same code executes on multiple backends, including CPU, GPU, & TPU

Why does JAX exist?

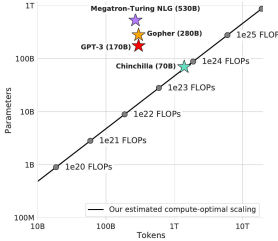
JAX powering large neural networks

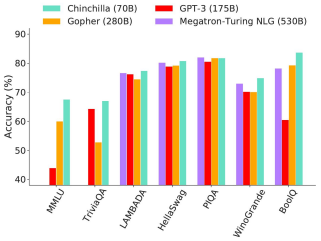
PaLM: Scaling Language Modeling with Pathways



DeepMind

Training Compute-Optimal Large Language Models





Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding

DeepMind 2022-5-19

A Generalist Agent

Minerva: Solving Quantitative Reasoning Problems with Language Models

Question: A line parallel to $y = 4x + 6$ passes through $(5, 10)$. What is the y-coordinate of the point where this line crosses the y-axis?

Model output: The line is parallel to $y = 4x+6$, which means that it has the same slope as $y = 4x+6$. Since the line passes through $(5, 10)$, we can use the point-slope form of a line to find the equation of the line:

$$y - 10 = 4(x - 5)$$
$$y - 10 = 4x - 20$$
$$y = 4x - 10$$

Now that we have the equation of the line, we can find the y-coordinate of the point where the line crosses the y-axis by substituting $x = 0$ into the equation:

$$y = 4 \cdot 0 - 10 = \boxed{-10}.$$

DeepMind 2022-3-16

Competition-Level Code Generation with AlphaCode

```
t=int(input())
for i in range(t):
    s=input()
    t=input()
    a=[]
    b=[]
    for j in s:
        a.append(j)
    for j in t:
        b.append(j)
    a.reverse()
    b.reverse()
    c=[]
    while len(b)!=0 and len(a)!=0:
```

Figure 3 | **Solution to Figure 2 generated by AlphaCode.** The model successfully extracted the information necessary to solve the problem from the natural language description:

1. The problem is to figure out if it is possible to convert one phrase to another by pressing backspace instead of typing some letters. So first we read the two phrases (lines 3-4).
2. If the letters at the end of both phrases don't

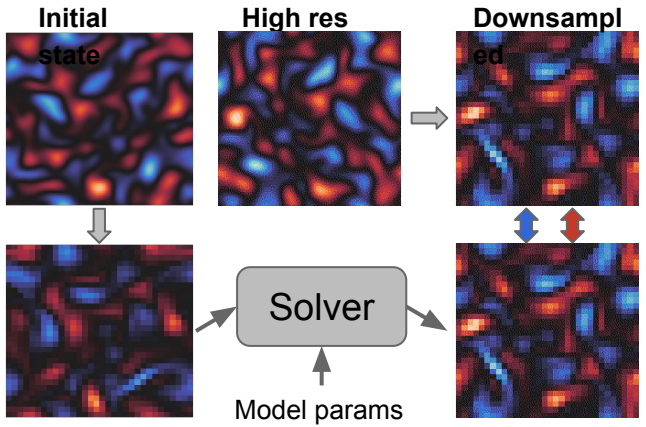
JAX in scientific computing

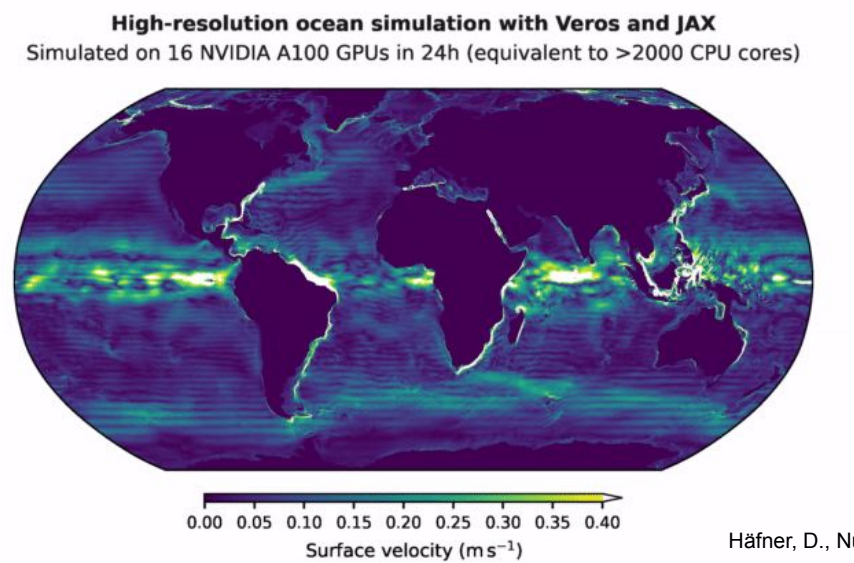
JAX-CFD is:

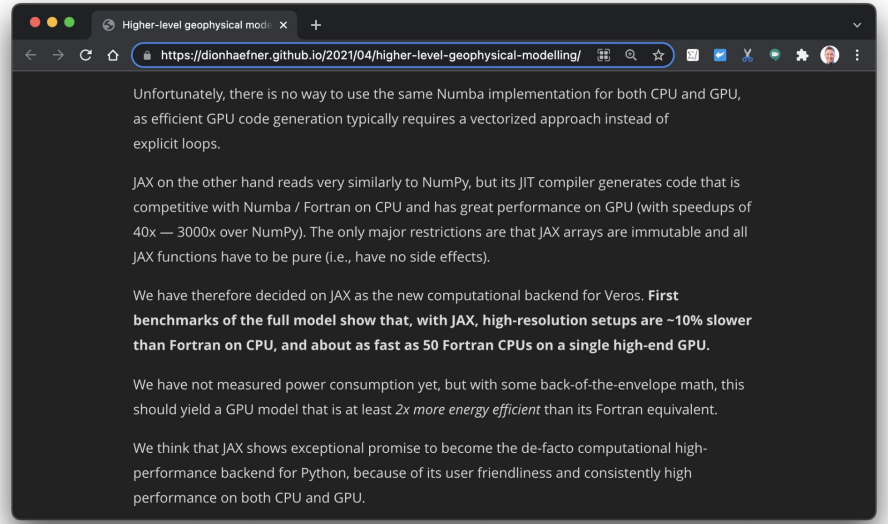
Fully **differentiable**
Designed for **hybrid ML/physics** models
GPU/TPU native

https://github.com/google/jax-cfd

Kochkov et al, Machine learning accelerated computational fluid dynamics (PNAS, 2021)

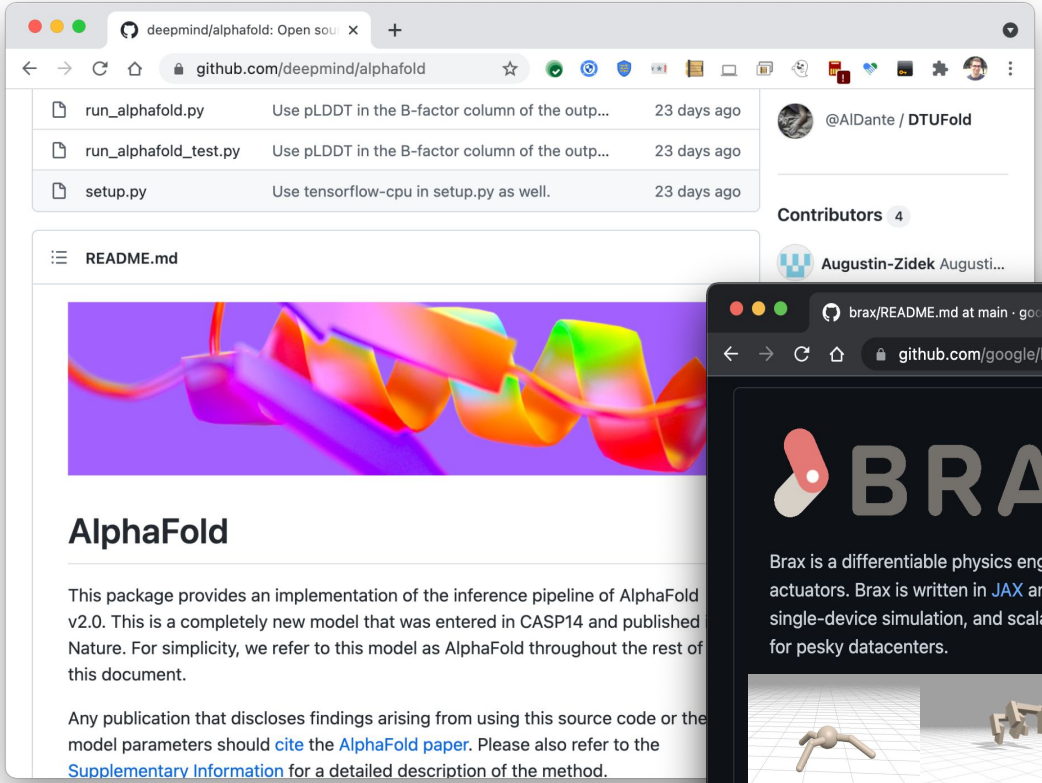


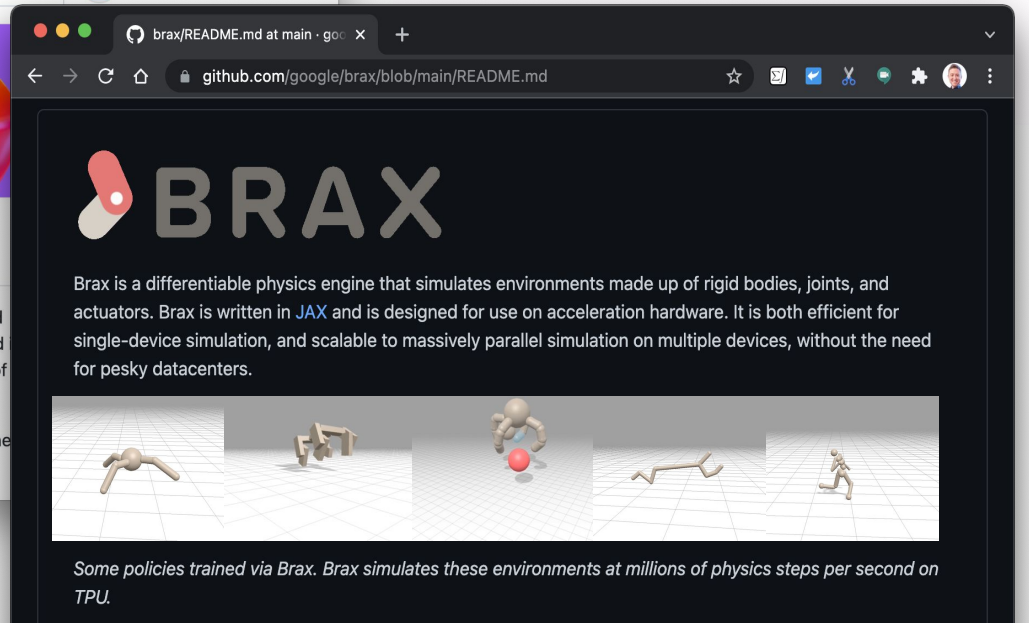




Häfner, D., Nuterman, R, & Jochum, M. (2021). Journal of Advances in Modeling Earth Systems.

JAX in scientific computing





So far.

Sales pitch! An invitation to try JAX.

High-level examples.

Coming up.

Scientific computing libraries

Limitations of JAX. (I'll be honest with you!)

Ecosystem

- **Linear solvers, QR decompositions, etc.**
- **ODE/SDE/PDE solvers**
- **Neural networks**
- **Optimisers**
- **…**

jax.scipy

jax.scipy.fft

jax.scipy.linalg

cholesky
eigh
qr
...

jax.scipy.stats

bernoulli
gamma
uniform
...

Notably missing: ***jax.scipy.integrate*** *and* ***jax.scipy.optimize***

Diffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```python
from diffrax import diffeqsolve, ODETerm, Kvaerno5
import jax.numpy as jnp

term = ODETerm(...)
solver = Kvaerno5()  # implicit solver; Jacobian found using autodiff!
y0 = jnp.array(...)

solution = diffeqsolve(term, solver, t0=0, t1=1, dt0=0.1, y0=y0)
```

https://github.com/patrick-kidger/diffrax

Equinox

For parameterised functions. (E.g. neural networks.)

```
import equinox as eqx

class Affine(eqx.Module):
    weight: jnp.ndarray
    bias: jnp.ndarray

    def __call__(self, x):
        return self.weight @ x + self.bias
```

https://github.com/patrick-kidger/equinox

Equinox is good for scientific computing.

There are also:

Flax (Google Research)

Haiku (DeepMind)

For neural networks only.

Interlude: I really like ABCs for defining interfaces, so Equinox has strong built-in support for these.

```python
class AbstractSolver(eqx.Module):
    @abstractmethod
    def step(...): ...

class AbstractRungeKutta(AbstractSolver):
    def step(...): ...

    tableau: AbstractClassVar[ButcherTableau]

class KenCarp3(AbstractRungeKutta):
    tableau = ...
```

Interlude: recursive binary checkpointing (“treeverse”)... with unknown termination condition!

```
eqx.internal.while_loop(..., kind="checkpointed", checkpoints=4)

# x
# xx
# xxx
# xxxx
# x.xxx
# x.x.xx
# x.x.x.x
# x.x.x.x.
# x...x.x.x
```

jaxtyping

Compile-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):
    # x and y must have the same shape
    ...

from jaxtyping import Array, Float

def batch_outer_product(x: Float[Array, "*batch dim_x"],
                        y: Float[Array, "*batch dim_y"]
                       ) -> Float[Array, "*batch dim_x dim_y"]:
    ...
```

https://github.com/google/jaxtyping

Lineax

Linear solvers ($Ax=b \Rightarrow x=A^{-1}b$) and linear least squares (pseudoinverse solution: $x=A^{\dagger}b$).

```
from lineax import JacobianLinearOperator, linear_solve, QR

def f(y: Float[Array, "3"], args
     ) -> tuple[Float[Array, "3"], Float[Array, "1 2"]]:
    ...

y = jnp.array([1.0, -1.5, 2.0])
op = JacobianLinearOperator(f, y, args=None)  # First-class autodiff!
vector = (jnp.array([4.0, -2.0, 1.0]), jnp.array([[-10.0, -0.7]]))  # pytrees!
solver = QR()
solution = linear_solve(op, vector, solver)
```

https://github.com/google/lineax

Optimistix

(a) root finding (b) fixed-point finding (c) minimisation (d) [nonlinear] least squares

```python
import optimistix as optx

# BFGS + Levenberg-Marquardt + Dogleg all hybridised together!
class HybridiseEverything(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f,
              solver=HybridiseEverything(rtol=1e-6, atol=1e-6),
              adjoint=optx.ImplicitAdjoint())  # IFT
```

https://github.com/???? (To be released soon!)

Case study: Diffrax vs …

…SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

…torchdiffeq?

- Is actively maintained
- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

…DifferentialEquations.jl?

- Python, NumPy
- Reliable autodiff, autoparallel
- Same speed
- (But a few less solvers!)

1. ODE + SDE (+ some PDE) solvers;
   a. DDE and DAE solvers coming soon!
2. High order, implicit, IMEX, symplectic;
3. Save at time points; dense output;
4. Multiple methods for backpropagation, e.g. Griewank-style recursive checkpointing.
5. Adaptive step size controllers (PID etc.)
6. Event handling; discontinuities;
7. Distributed computing; GPU/TPU support;

Diffrax also introduces a new, unified approach to combining diffeq numerics, via rough path theory. "SDDAEs" anyone?!

Full example: neural ODE

```python
import diffrax as dfx
import equinox as eqx
import jax
import optax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size="scalar", width_size=32, depth=1,
                 activation=eqx.nn.PReLU(), final_activation=eqx.nn.PReLU(),
                 key=jax.random.PRNGKey(0))

def solve(model, y0: Float[Array, "dim"]):
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: jax.grad(model)(y)),
                          dfx.Tsit5(), t0=0, t1=1, dt0=0.1, y0=y0)
    return sol.ys

def loss(model, batch_y0: Float[Array, "batch dim"],
         batch_y1: Float[Array, "batch dim"]):
    pred_batch_y1 = jax.vmap(solve, in_axes=(None, 0))(model, batch_y0)
    return jax.numpy.mean((pred_batch_y1 - batch_y1)**2)

@jax.jit
def make_step(model, opt_state, batch_y0: Float[Array, "batch dim"],
              batch_y1: Float[Array, "batch dim"]):
    grads = jax.grad(loss)(model, batch_y0, batch_y1)
    updates, opt_state = optim.update(grads, opt_state)
    model = optax.apply_updates(model, grads)
    return model, opt_state

optim = optax.adam(learning_rate=3e-4)
opt_state = optim.init(mlp)
dataset = zip(jax.numpy.array([[[1., 2., 3., 4.]]]), jax.numpy.array([[[5., 6., 7., 8.]]]))  # toy dataset
for batch_y0, batch_y1 in dataset:
    mlp, opt_state = make_step(mlp, opt_state, batch_y0, batch_y1)
```

Given

$$f(y, \theta) = \text{NeuralNetwork}(y, \theta)$$

$$y_i(0) = y_{i,0}, \qquad \frac{\mathrm{d}y_i}{\mathrm{d}t} = \nabla_y f(y_i(t), \theta)$$

Minimise with SGD:

$$\frac{1}{n} \sum_{i=1}^{n} ||y_i(1) - y_{i,1}||^2$$

JAX limitations

JAX is not for general-purpose computation!

- Optimised for **arrays** and **linear algebra**. *If you want to do a branch-and-bound combinatorial optimisation algorithm, then you're out of luck.*

JAX has a "skill floor", i.e. it takes some time to learn.

- **Some gotchas**. *E.g. in-place updates require care.*
- Functional programming. *Not always familiar.*
- Compilation time. *You need to know how to manage this, e.g.* **jax.lax.scan** *instead of* **for**.

Questions?

Links:

- JAX: http://github.com/google/jax
- Diffrax: https://github.com/patrick-kidger/diffrax
    - Has links onwards to the rest of the JAX ecosystem.

My contacts:

- kidger@google.com
- twitter.com/PatrickKidger