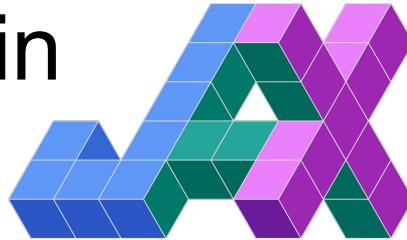


Scientific computing and machine learning in



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

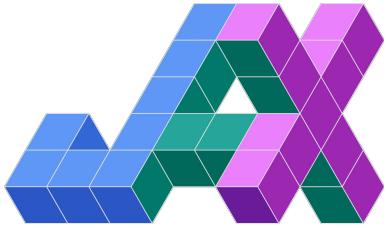
[CuPy](#): GPU-backed arrays

[Numba](#): JIT compilation

[Autograd](#): autodifferentiation

[TensorFlow](#): machine learning operations

[PyTorch](#): eager execution; model-building syntax



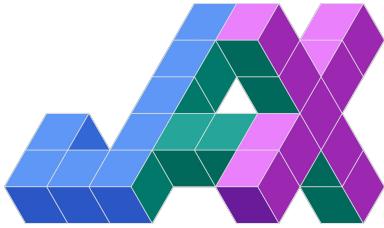
Familiar API

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):
    ...

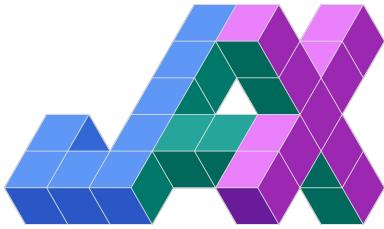
grad_fn = jit(grad(loss))
```

```
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) 
```

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

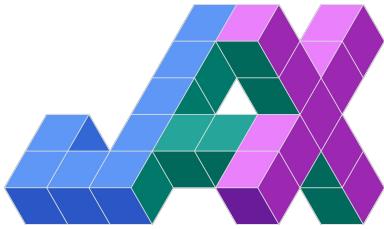
def forward(params, inputs):
    ...

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.

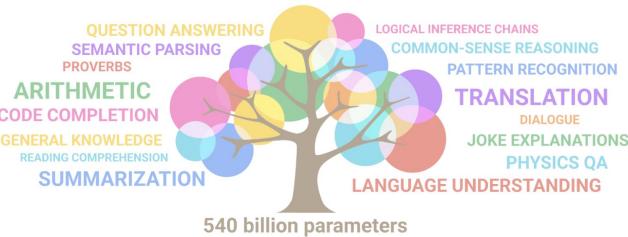
Run Anywhere

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



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



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:

$$\begin{aligned}y - 10 &= 4(x - 5) \\y - 10 &= 4x - 20 \\y &= 4x - 10\end{aligned}$$

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:

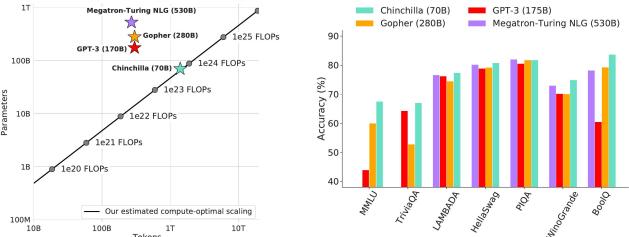
$$y = 4 \cdot 0 - 10 = \boxed{-10}.$$



2022-3-16

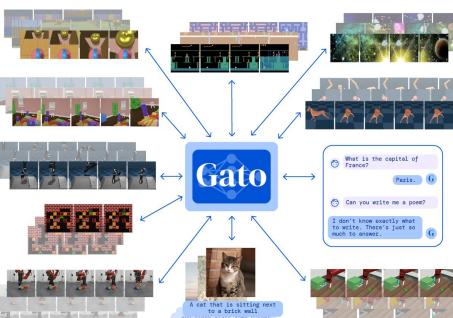


Training Compute-Optimal Large Language Models



2022-5-19

A Generalist Agent



Competition-Level Code Generation with AlphaCode

```
1 t=int(input())
2 for i in range(t):
3     s=input()
4     t=input()
5     a=[]
6     b=[]
7     for j in s:
8         a.append(j)
9     for j in t:
10        b.append(j)
11    a.reverse()
12    b.reverse()
13    c=[]
14    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

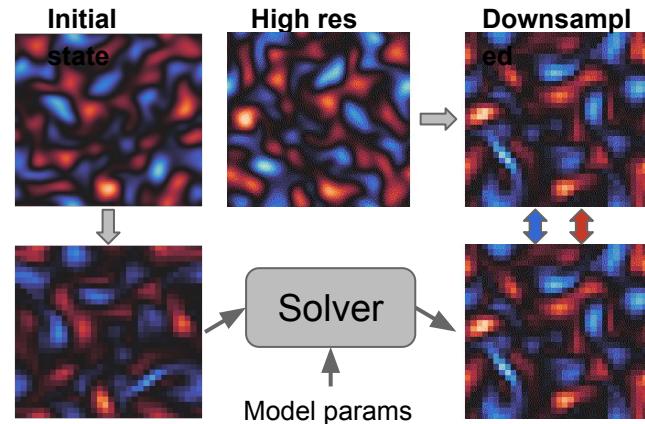
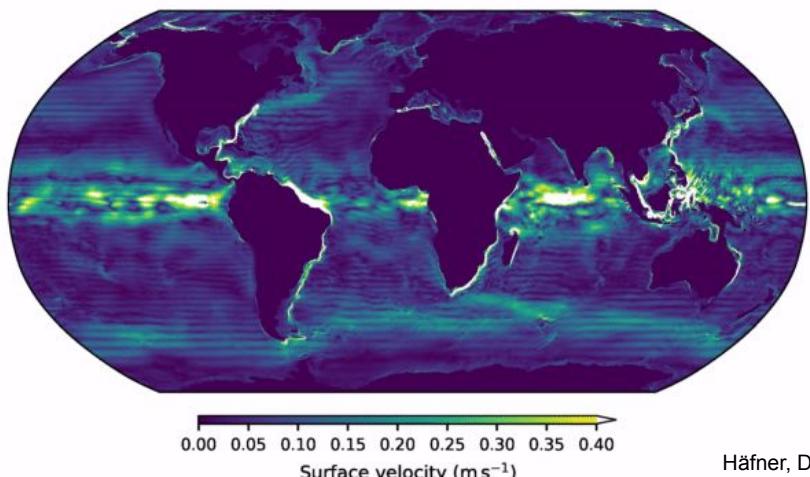
GPU/TPU native

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

Kochkov et al, [Machine learning accelerated computational fluid dynamics \(PNAS, 2021\)](#)

High-resolution ocean simulation with Veros and JAX

Simulated on 16 NVIDIA A100 GPUs in 24h (equivalent to >2000 CPU cores)



Higher-level geophysical modeling

Unfortunately, there is no way to use the same Numba implementation for both CPU and GPU, as efficient GPU code generation typically requires a vectorized approach instead of explicit loops.

JAX on the other hand reads very similarly to NumPy, but its JIT compiler generates code that is competitive with Numba / Fortran on CPU and has great performance on GPU (with speedups of 40x — 3000x over NumPy). The only major restrictions are that JAX arrays are immutable and all JAX functions have to be pure (i.e., have no side effects).

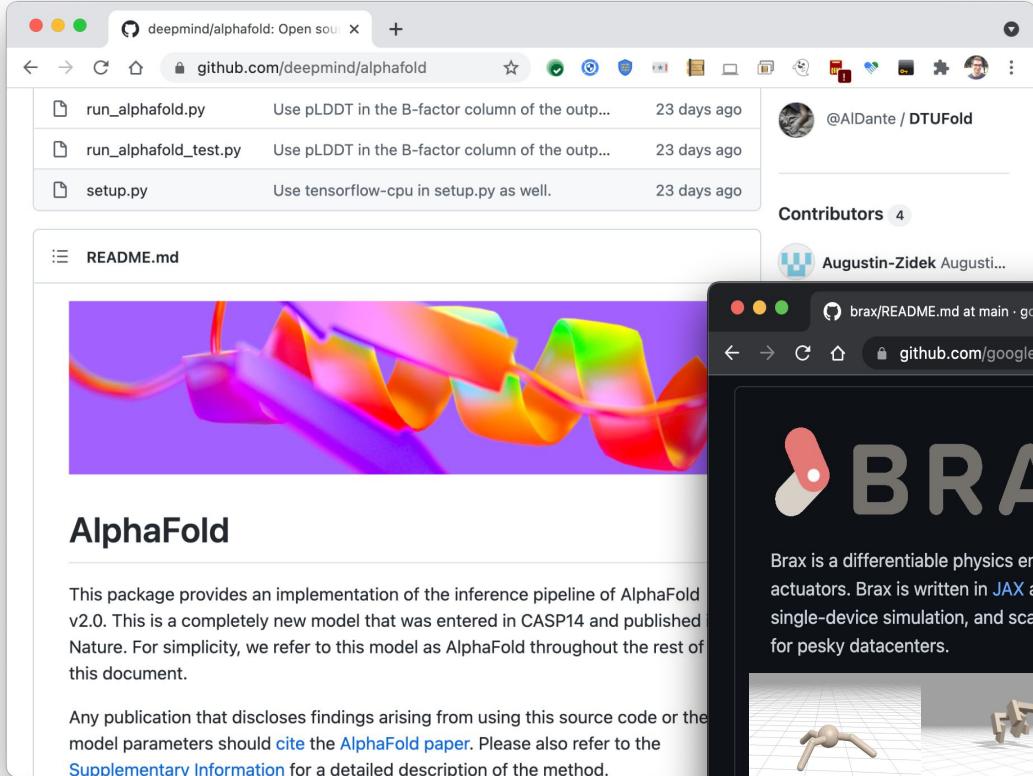
We have therefore decided on JAX as the new computational backend for Veros. First benchmarks of the full model show that, with JAX, high-resolution setups are ~10% slower than Fortran on CPU, and about as fast as 50 Fortran CPUs on a single high-end GPU.

We have not measured power consumption yet, but with some back-of-the-envelope math, this should yield a GPU model that is at least 2x more energy efficient than its Fortran equivalent.

We think that JAX shows exceptional promise to become the de-facto computational high-performance backend for Python, because of its user friendliness and consistently high performance on both CPU and GPU.

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

JAX in scientific computing

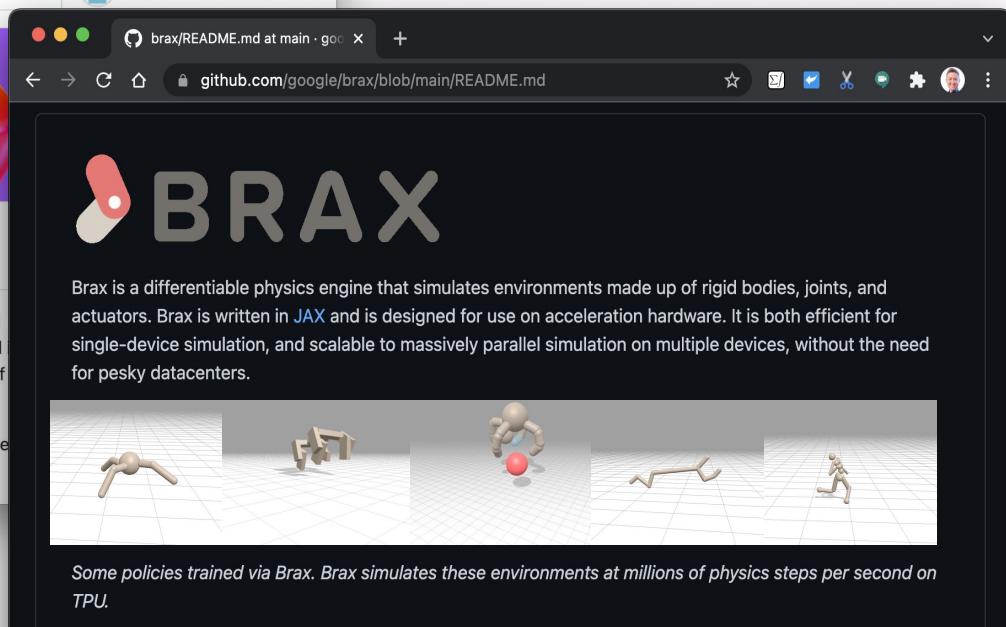


The screenshot shows the GitHub repository for AlphaFold. It includes a list of files like `run_alphaFold.py`, `run_alphaFold_test.py`, and `setup.py`. Below the files is a `README.md` section featuring a vibrant, multi-colored 3D visualization of a protein structure.

AlphaFold

This package provides an implementation of the inference pipeline of AlphaFold v2.0. This is a completely new model that was entered in CASP14 and published Nature. For simplicity, we refer to this model as AlphaFold throughout the rest of this document.

Any publication that discloses findings arising from using this source code or the model parameters should cite the [AlphaFold paper](#). Please also refer to the [Supplementary Information](#) for a detailed description of the method.



The screenshot shows the GitHub repository for Brax. The `README.md` page features a large, stylized logo consisting of colored hexagons. Below the logo, the text describes Brax as a differentiable physics engine. It includes a section titled "Some policies trained via Brax. Brax simulates these environments at millions of physics steps per second on TPU." followed by five small images showing various 3D physics simulations.

BRAX

Brax is a differentiable physics engine that simulates environments made up of rigid bodies, joints, and actuators. Brax is written in **JAX** and is designed for use on acceleration hardware. It is both efficient for single-device simulation, and scalable to massively parallel simulation on multiple devices, without the need for pesky datacenters.

Some policies trained via Brax. Brax simulates these environments at millions of physics steps per second on TPU.

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](#)

[jax.scipy.stats](#)

cholesky

eigh

qr

...

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!

```
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
```

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.

```
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"]:
    ...

...
```



Lineax

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

```
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

```
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
```

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

```
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.y

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

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

$$y_i(0) = y_{i,0}, \quad \frac{dy_i}{dt} = \nabla_y f(y_i(t), \theta)$$

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