Accelerated machine-learning research via composable function transformations in Python

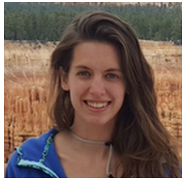

mattjj@


frostig@


leary@


dougalm@


phawkins@


skyewm@


jekbradbury@


necula@

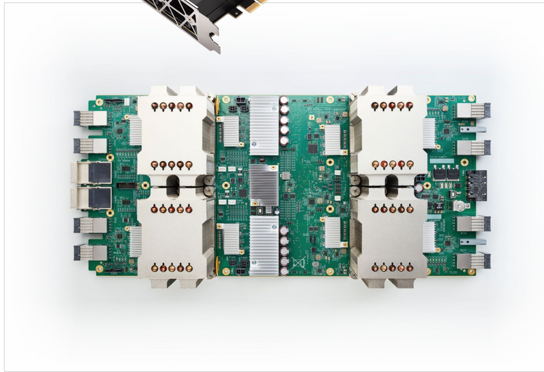
...@google.com

What is **JAX**

```python
import jax.numpy as np
from jax import jit, grad, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = np.dot(inputs, W) + b
    inputs = np.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return np.sum((preds - targets) ** 2)
```



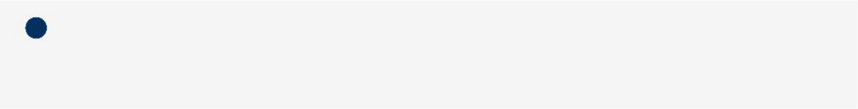
```python
gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), (None, 0)))
```

JAX is an extensible system for **composable function transformations** of Python+NumPy code.

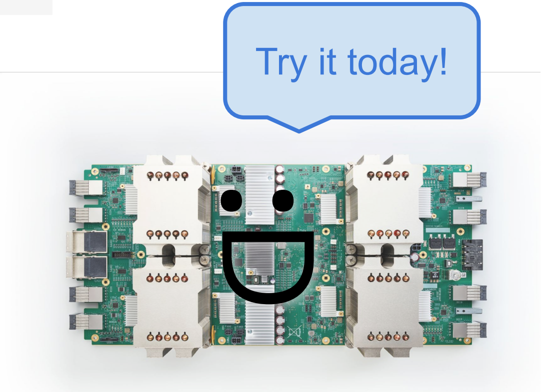
You can use JAX for free on **Cloud TPUs** in Colab!

bit.ly/jax-tpu

(github.com/google/jax/tree/master/cloud_tpu_colabs)



Wave simulation from the "Wave Equation" notebook



Demo!

How **JAX** works

Step 1: Python function → JAX IR

```python
def f(x):
  return x + 2
```

```python
class EspressoDelegator(object):

  def __add__(self, num_espressos):
    subprocess.popen(["ssh", ...])
```

Step 1: Python function → JAX IR

```
def f(x::f32):
  return x + 2
```

Step 1: Python function → JAX IR

```python
def f(x):
  return x + 2
```

How does `f` behave on...

```
ShapedArray(f32, (3,))

ShapedArray(f32, (2, 2))

ConcreteArray(f32, [[1., 2.], [3., 4.]])
```

Abstract value

Step 1: Python function → JAX IR

```python
def f(x):
  return x + 2
```

How does `f` behave on...

`ShapedArray(f32, (3,))`

`ShapedArray(f32, (2, 2))`

~~ConcreteArray(f32, [[1., 2.], [3., 4.]])~~

Abstract value

Step 1: Python function → JAX IR

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

Calls to JAX **primitive operations**, the elementary operations we know how to transform.

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```python
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

Replace argument **x** with a special tracer object

```python
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
{ lambda  ;  ; a.
  let b = log a
```

```python
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2) # ln_2 = 0.693147
  return ln_x / ln_2
```

```
{ lambda  ;  ; a.
  let b = log a
```

Trace doesn't include log(2) because no **data dependence** on tracer object

```
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
```

```python
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```python
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  global_list.append(x)
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

Behavior not captured by jaxpr!

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```python
x = np.array(...)
y = jit(log2)(x)
```

Traced function must be **pure**
(no side effects visible outside the function,
output fully determined by input)

Step 1: Python function → JAX IR

```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2)
  return ln_x / ln_2
```

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```python
x = np.array(...)
y = jit(log2)(x)
```

Step 1: Python function → JAX IR

```
def f(x):
  if x.ndim == 0:
    return 2*x**3.
  else:
    return 3*x
```

```
jit(f)(0.)
```

```
{ lambda  ;  ; a.
  let b = pow a 3.0
      c = mul b 2.0
  in [c] }
```

```
jit(f)(np.ones(4.))
```

```
{ lambda  ;  ; a.
  let b = mul a 3.0
  in [b] }
```

Step 1: Python function → JAX IR

```
def f(x):
  if x > 0: # ERROR!
    return 2*x**3.
  else:
    return 3*x
```

```
jit(f)(0.)
```

```
TypeError: Abstract value passed to
`bool`, which requires a concrete value.
```

Step 1: Python function → JAX IR

```python
def f(x):
  if x > 0:
    return 2*x**3.
  else:
    return 3*x
```

```
grad(f)(1.)
```

```
{ lambda  ;  ; a.
  let b = pow a 3.0
      c = mul b 2.0
  in [c] }
```

```
grad(f)(-1.)
```

```
{ lambda  ;  ; a.
  let b = mul a 3.0
  in [b] }
```

Step 1: Python function → JAX IR

$\top$

↑

`... Unshaped(f32) ...`

```python
# no control flow allowed
z = cos(x + y)
```

↑

jit, vmap → `... Shaped(f32, (2,2)) ...`

```python
# can branch on shape
if x.shape[0] > 2: ...
for subarray in array: ...
```

↑

grad → `... EpsilonBall(f32,[[1.,2.],[3.,4.]]) ...`

```python
# can branch on value if x.val != 0
if x > 0: ...
```

↑

eval → `... Concrete(f32,[[1.,2.],[3.,4.]]) ...`

```python
# can always branch on value
if x > 0: ...
```

↑

$\perp$

Step 2: transform jaxpr

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

Step 2: transform jaxpr

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```
def log_jvp(x, t):
  return lax.div(t, x)
```

```
def div_jvp(x, y, tx, ty):
  return (ty / y,
          -x * ty / y**2)
```

Every **transform** has a rule for every primitive

Step 2: transform jaxpr

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```python
def jvp_transform(jaxpr, x, t):
  env = {jaxpr.invar: (x, t)}
  for eqn in jaxpr.eqns:
    rule = jvp_rules[eqn.prim]
    xs, ts = zip(*[env[v] for v in eqn.ins])
    env[eqn.out] = rule(xs, ts)
  return env[jaxpr.outvar]
```

Transform itself is a simple jaxpr **interpreter**

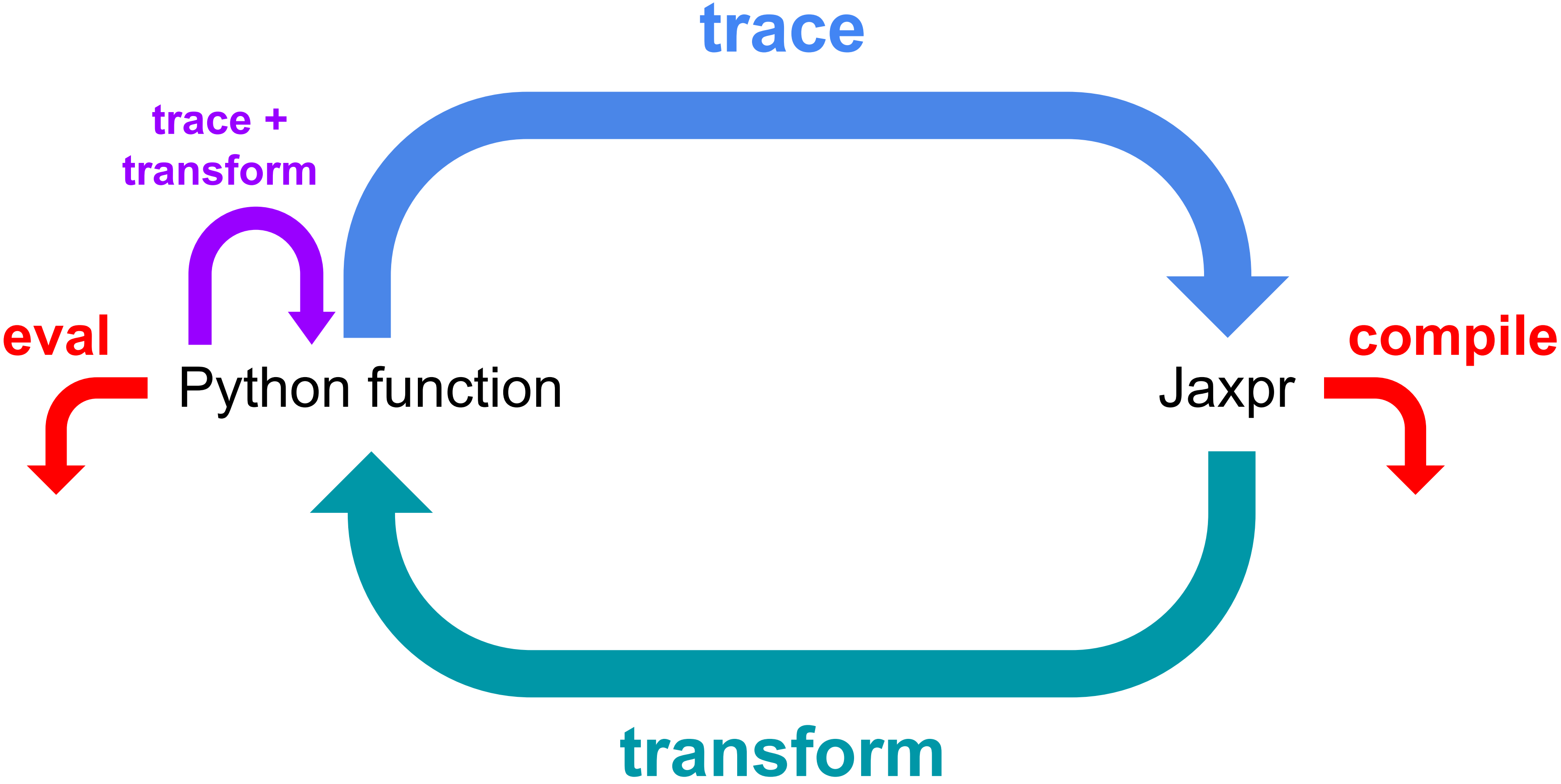
Step 2: transform jaxpr

Replace arguments with tracer objects

```
{ lambda  ;  ; a.
  let b = log a
      c = div b 0.693147
  in [c] }
```

```python
def jvp_transform(jaxpr, x, t):
  env = {jaxpr.invar: (x, t)}
  for eqn in jaxpr.eqns:
    rule = jvp_rules[eqn.prim]
    xs, ts = zip(*[env[v] for v in eqn.ins])
    env[eqn.out] = rule(xs, ts)
  return env[jaxpr.outvar]
```

```
{ lambda ; ; a b.
  let c = log a
      d = div c 0.693147
      e = div b a
      f = div e 0.693147
  in [d, f] }
```


trace
trace +
transform
eval
Python function
Jaxpr
compile
transform

Why researchers like JAX

1. JAX is **easy to use**
   - Minimal + expressive API (NumPy + function transformations)
   - Can understand “what it’s doing”
   - Same API for CPU/GPU/TPU
2. JAX is **fast**
   - Good performance out-of-the-box
   - Simple parallelization model (pmap)
3. Robust and powerful **transformations**
4. **Functional** programming model
   - Aligns well with math
   - Reproducible results
   - Easier to debug
   - The key to JAX’s superpowers

Current limitations

1. Limited **higher-level libraries** for layers/models
   - Stay tuned!

2. **Per-op dispatch overhead** not fully optimized
   - Solution 1: keep optimizing
   - Solution 2: more jit

3. Transforms only work on **pure functions**
   - User-promised

“Eager-mode” performance with jit

Composable jit means we can write readable and efficient library code.

```
def adam(step_size, b1=0.9, b2=0.999, eps=1e-8):

  ...

  @jit
  def update(i, g, state):
    x, m, v = state
    m = (1 - b1) * g + b1 * m
    v = (1 - b2) * (g ** 2) + b2 * v
    mhat = m / (2 - b1 ** (i + 1))
    vhat = v / (2 - b2 ** (i + 1))
    x = x - step_size(i) * mhat / (np.sqrt(vhat) + eps)
    return x, m, v
```

All computations are JIT-compiled with XLA.
JAX has almost no handwritten kernels.

Current limitations

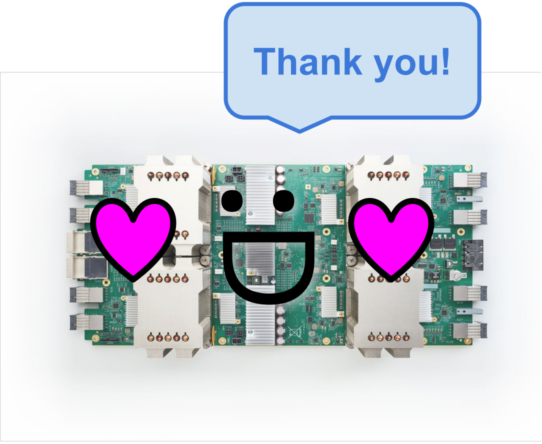
1. Limited **higher-level libraries** for layers/models
    - Stay tuned!

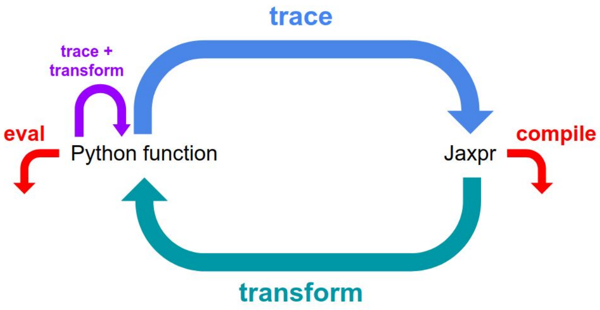
2. **Per-op dispatch overhead** not fully optimized
    - Solution 1: keep optimizing
    - Solution 2: more jit

3. Transforms only work on **pure functions**
    - User-promised

Many projects are already using JAX!

1. Studying neural net training with **advanced autodiff**
   - **neural-tangents**: experiments with the Neural Tangent Kernel
   - **spectral-density**: estimating loss function Hessian spectra
2. Algorithms for **robotics** and **control**
   - asynchronous **model-predictive control**
3. **Bayesian** models and inference
   - **NumPyro**: probabilistic programming and NUTS
4. Simulation and **science**
   - **jax-md**: differentiable, hardware-accelerated molecular dynamics for physics
   - **Time Machine**: molecular dynamics for biology with meta-optimization
   - **comp-thru-dynamics**: dynamics in artificial and biological neural systems
5. Large-scale **neural network** training
   - **trax**: Tensor2Tensor in JAX





github.com/google/jax

Demo: **bit.ly/jax-tpu**

Stickers!

Google