Probabilistic Programming by Transformation in JAX

Sharad Vikram, Matthew D. Hoffman, Alexey Radul, James Bradbury, Matthew Johnson, Sergio Guadarrama
Google Research

What is JAX?
JAX is a Python numerical computing library based on *composable function transformations*.

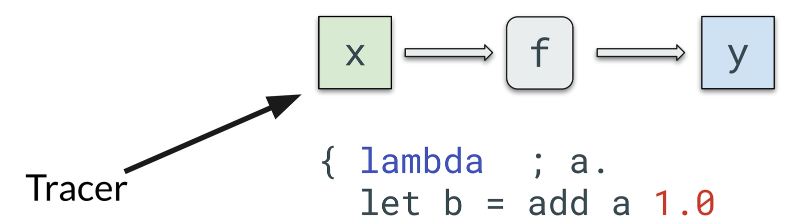
Examples transformations are:

- `grad(f)` - Automatic differentiation
- `vmap(f)` - Vectorized map
- `jit(f)` - JIT compilation
- `pmap(f)` - Distributed map

Transformations are implemented by tracing their input functions.

```
@jit
def f(x):
  return jnp.exp(x + 1.)

f(jnp.ones(5))
```




```
{ lambda  ; a.
  let b = add a 1.0
      c = exp b
  in (c,) }
```

Challenge:
How do we build a probabilistic programming system on top of JAX that is fully compatible with JAX transformations?

Proposal: Oryx
Oryx adds new function transformations to JAX that enable a novel probabilistic programming system.

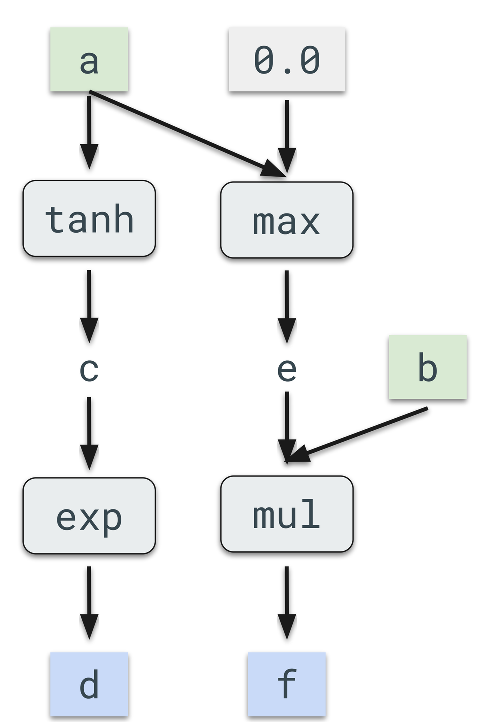
- `inverse(f)` - Function inversion
- `log_prob(f)` - Log density computation
- `harvest(f)` - Tagging-based effect handling

Automatic Function Inversion

Building an intermediate representation
Python functions are traced to build a JAX expression, or JAXPR.

```
def g(x, y):
  return (
    jnp.exp(jnp.tanh(x)),
    jnp.maximum(x, 0.) * y)
```

```
{ lambda  ; a b.
  let c = tanh a
      d = exp c
      e = max a 0.0
      f = mul e b
  in (d, f) }
```

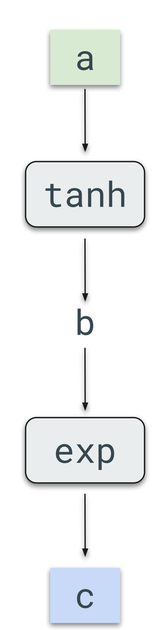


JAXPRs are composed of JAX *primitives*, or the lowest level traceable operations in JAX.

Propagation Algorithm

By initializing the output nodes in the graphs to known values, we can "fill in" the missing values in the graph until we have values for the inputs, using rules of the following form for each JAX primitive:

```
def rule(invals, outvals):
  ...
  return new_invals, new_outvals
```




```
{
    a: ?,
    b: ?,
    c: 5.
}
```

Execute rule for `exp`

```
{
    a: ?,
    b: 1.609,
    c: 5.
}
```

Execute rule for `tanh`

```
{
    a: 1.015,
    b: 1.609,
    c: 5.
}
```

Log Density Transformation

Probabilistic Programs in Oryx
In Oryx, probabilistic programs are just JAX functions that take in a pseudorandom number generation key as their first input.

```
def sample(key):
  x = random_variable(tfd.Normal(0., 1.))(key)
  return jnp.exp(x / 2.) + 2.
sample(random.PRNGKey(0)) # ==> 2.902198
```

Propagating inverses and ILDJs
To convert a program into its log-density function, we can run a propagation on it similar to function inversion, but additionally keep track of inverse log-det Jacobians.

```
log_prob(sample)(3.)  # ==> -0.22579134
```

Effect Handling

Tagging-based effect system
JAX transformations require pure (side-effect free) functions as inputs. How can we find random samples located in a program?

```
TAG = 'intermediate'
def f(x):
  y = sow(x + 1., name='y', tag=TAG)
  return y ** 2
f(1.)  # ==> 4.
```

We present a general-purpose function transformation that enables *collecting* and *injecting* tagged values into a program.
`sow` - tags values (semantically is identity)
`harvest` - "functionalizes"

```
reap(f, tag=TAG)(1.)  # ==> {'y': 2.}

plant(f, tag=TAG)(dict(y=5.), 1.)  # ==> 25.

harvest(f, tag=TAG)({}, 1.)  # ==> (4., {'y': 2})
harvest(f, tag=TAG)(dict(y=5.), 1.)  # ==> (25., {})
```

Probabilistic Programming Transformations

With the base transformations available, we can now implement some PPL-specific transformations:

`joint_sample` - converts a program into one that returns latent random samples (based on `harvest`)

`intervene` - inserts values for random samples in probabilistic programs

Oryx transformations compose with JAX ones!

```
def latent_normal(key):
  z_key, x_key = random.split(key)
  z = random_variable(
        tfd.Normal(0., 1.), name='z')(z_key)
  x = random_variable(
        tfd.Normal(z, 1.), name='x')(x_key)
  return x
joint_sample(latent_normal)(
     random.PRNGKey(0)) # ==> {'x': -1.1076, 'z': 0.14389}
log_prob(joint_sample(latent_normal))(
     dict(x=0., z=0.)) # ==> -1.837877
intervene(latent_normal, x=5.)(
     random.PRNGKey(0)) # ⇒ 5.
```

Applications

Because Oryx and JAX transformations are interoperable, we can easily do large scale Bayesian inference on GPUs and TPUs. Automatic inversion enables writing complex, trainable distributions for applications like normalizing flows. Finally, the function transformation paradigm enables applications like automatically constructing surrogate posteriors for variational inference.

Learn more at tensorflow.org/probability/oryx and try it yourself with `pip install oryx`.