1 JAX

JAX follows the functional programming paradigm. That is, JAX provides tools to transform a function into another function. Specifically, JAX can automatically compute the derivative of a function or composition of functions.

As an example, for \( f(x) = \frac{1}{2} \|x\|_2^2 \), JAX computes \( \nabla f : \mathbb{R}^n \to \mathbb{R}^n \) where \( \nabla f(x) = x \).

```
[ ]: import jax
import jax.numpy as jnp

def f(x):
    return jnp.sum(x**2)/2  # identical to numpy syntax
grad_f = jax.grad(f)  # compute the gradient function

x = jnp.array([0., 1., 2.])  # use JAX arrays!
print('x:', x)
print('f(x):', f(x))
print('grad_f(x):', grad_f(x))
```

WARNING:jax._src.lib.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)

\[
x: \quad [0. 1. 2.]
\]
\[
f(x): \quad 2.5
\]
\[
grad_f(x): \quad [0. 1. 2.]
\]

2 Automatic Differentiation

Automatic Differentiation (AD, autodiff) uses pre-defined derivatives and the chain rule to compute derivatives of more complex functions.

Consider the function \( f : \mathbb{R}^n \to \mathbb{R}^m \). The Jacobian of \( f \) evaluated at the point \( x \in \mathbb{R}^n \) is the matrix

\[
\partial f(x) = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_m}{\partial x_2}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x)
\end{bmatrix}^{m \times n} \in \mathbb{R}^{m \times n}.
\]
As for any matrix, the Jacobian \( \partial f(x) : \mathbb{R}^n \to \mathbb{R}^m \) is a linear map \( v \mapsto \partial f(x)v \) defined by the usual matrix-vector multiplication rules.

AD can be used to compute the *Jacobian-Vector Product (JVP)*

\[
\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m \\
v \mapsto \partial f(x)v
\]

and the *Vector-Jacobian Product (VJP)*

\[
\partial f(x)\top : \mathbb{R}^m \to \mathbb{R}^n \\
w \mapsto \partial f(x)\top w
\]

The maps \( v \mapsto \partial f(x)v \) and \( w \mapsto \partial f(x)\top w \) are also known as the *pushforward* and *pullback*, respectively, of \( f \) at \( x \). The vector \( x \) is referred to as the primal, while the vectors \( v \) and \( w \) are termed *seeds* in AD literature.

Consider the function composition

\[
h(x) = (f_N \circ f_{N-1} \circ \cdots \circ f_1)(x) = f_N(f_{N-1}(\cdots f_1(x) \cdots)),
\]

where each \( f_k : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}} \) is some differentiable map.

We can write this recursively as

\[
y_0 = x \in \mathbb{R}^n, \quad y_{k+1} = f_{k+1}(y_k) \in \mathbb{R}^{d_{k+1}}, \quad y_N = h(x) \in \mathbb{R}^{d_N}.
\]

By the chain rule, we have

\[
\partial h(x) = \partial f_N(y_{N-1}) \partial f_{N-1}(y_{N-2}) \cdots \partial f_1(y_0).
\]

This sequence of matrix multiplications that can get quickly get expensive for complicated functions!

It is more efficient and usually sufficient in practice to compute JVPs via the recursion

\[
\partial h(x)v_0 = \partial f_N(y_{N-1}) \partial f_{N-1}(y_{N-2}) \cdots \partial f_1(y_0)v_0 \\
= v_N, \\
v_k = \partial f_k(y_{k-1})v_{k-1}
\]

and VJPs via the recursion

\[
\partial h(x)\top w_0 = \partial f_1(y_0)\top \cdots \partial f_{N-1}(y_{N-2})\top \partial f_N(y_{N-1})\top w_0 \\
= w_N, \\
w_k = \partial f_{N-k+1}(y_{N-k})\top w_{k-1}
\]

VJPs require more memory than JVPs, since \( \{y_k\}_{k=1}^{N-1} \) must be computed and stored first (i.e., the *forward pass*) before recursing (i.e., the *backward pass*).
2.1 Example: VJP as a gradient

For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the Jacobian at $x$ is $\partial f(x) \in \mathbb{R}^{1 \times n}$, so

$$\nabla f(x) = \partial f(x)^\top.$$ 

E.g., if $f(x) = \frac{1}{2} \|x\|_2^2$, then $\nabla f(x) = x \cdot 1$.

```
[ ]: f = lambda x: jnp.sum(x**2)/2  # anonymous functions work as well
    x = jnp.array([0., 1., 2.])
    f_x, dfxT = jax.vjp(f, x)  # compute forward pass and VJP function
    dfxT_1 = dfxT(1.)

    print('x: ', x)
    print('f(x): ', f_x)
    print('dfxT(1): ', dfxT_1)
```

x: [0. 1. 2.]
f(x): 2.5
dfxT(1): (DeviceArray([0., 1., 2.], dtype=float32),)

2.2 Example: JVP as a directional derivative

The directional derivative of $f : \mathbb{R}^n \to \mathbb{R}$ at $x \in \mathbb{R}^n$ along $v \in \mathbb{R}^n$ is

$$\nabla f(x)^\top v = \partial f(x)v.$$ 

E.g., if $f(x) = \frac{1}{2} \|x\|_2^2$, then $\nabla f(x)^\top v = x^\top v$.

```
[ ]: f = lambda x: jnp.sum(x**2)/2
    x = jnp.array([0., 1., 2.])
    v = jnp.array([1., 1., 1.])

    # use tuples to separate inputs from seeds
    f_x, dfx_v = jax.jvp(f, (x,), (v,))

    print('x: ', x)
    print('f(x): ', f_x)
    print('dfx(v): ', dfx_v)
```

x: [0. 1. 2.]
f(x): 2.5
dfx(v): 3.0

2.3 Example: Multi-input, multi-output VJP

Let’s try something more complicated:
\[ f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R} \]

\[
(x, y) \mapsto \left( \frac{1}{2} \|x\|_2^2 + \frac{1}{2} \|y\|_2^2, \sum_{i=1}^n x_i \right)
\]

\[ f = [f_1, f_2] \]

---

```python
import jax
import jnp

def f(x, y):
    f1 = jnp.sum(x**2) / 2 + jnp.sum(y**2) / 2
    f2 = jnp.sum(x)
    return f1, f2

x = jnp.array([0., 1., 2.])
y = jnp.array([0., 1., 2.])

f_xy, dfT = jax.vjp(f, x, y)

print('x,y: ', x, y)
print('f(x,y): ', f_xy)
print('dfT(1,1): ', dfT((1., 1.)))  # provide tuple as input
```

\[ x, y: \begin{bmatrix} 0. & 1. & 2. \end{bmatrix} \begin{bmatrix} 0. & 1. & 2. \end{bmatrix} \]

\[ f(x,y): (\text{DeviceArray}(5., \text{dtype=float32}), \text{DeviceArray}(3., \text{dtype=float32})) \]

\[ dfT(1,1): (\text{DeviceArray}([1., 2., 3.], \text{dtype=float32}), \text{DeviceArray}([0., 1., 2.], \text{dtype=float32})) \]

---

2.4 Example: VJP and JVP for a Matrix Input

We can generalize VJPs and JVPs to non-vector inputs as well:

\[ f : \mathbb{R}^{n \times n} \to \mathbb{R} \]

\[ X \mapsto a^\top X b \]

```python
import jnp
import jnp.eye

def f(X):
    a, b = jnp.array([0., 1., 2.]), jnp.array([0., 1., 2.])
    return a @ (X @ b)

X = jnp.ones((3, 3))
f_x = f(X)
w, V = jnp.array(1.), jnp.eye(3)
f_x, dfT = jax.vjp(f, X)
f_x, df_v = jax.jvp(f, (X,), (V,))

print('X:\n', X, '\n', 'f(X): ', f_x, '\n', sep='')
print('dfT(1):\n', dfT(w), '\n', 'df(1): ', df_v, sep='')
```

\[ X: \begin{bmatrix} 1. & 1. & 1. \\ 1. & 1. & 1. \end{bmatrix} \]

---

4
3 Auto-Vectorizing Functions with \texttt{jax.vmap}

For some complicated function $f: \mathbb{R}^n \to \mathbb{R}^m$, we want to calculate $f(x)$ for many different values of $x$ without looping.

This is known as \textit{vectorizing} a function. JAX can do this automatically!

```python
f = lambda x: jnp.array([jnp.sum(x**2)/2, jnp.linalg.norm(x, jnp.inf)]
```

```python
f = jax.vmap(f)  # \texttt{axis = 0}
```

```python
batch_size, n = 100, 3
x = jnp.ones((batch_size, n))  # dummy values with desired shape
```

```python
print(x.shape)
print(f(x).shape)
```

```
(100, 3)
(100, 2)
```

3.1 Example: Batch Evaluation of a Neural Network

```python
def f(x, W, b):
    return W[1] @ jnp.tanh(W[0] @ x + b[0]) + b[1]
```

```python
f = jax.vmap(f, in_axes=(0, None, None))
f = jax.vmap(f, in_axes=(0, None, None))
```

```python
n, m = 3, 5
batch_size = 100
hdim = 32
```

```python
W = (jnp.ones((hdim, n)), jnp.ones((m, hdim)))
b = (jnp.ones(hdim), jnp.ones(m))
x = jnp.ones((40, batch_size, n))
```

```python
print(x.shape)
print(f(x, W, b).shape)
```

```
(40, 100, 3)
(40, 100, 5)
```
3.2 Example: Jacobian Matrix from JVPs and VJPs

Let \( e_k^{(d)} \in \{0,1\}^d \) denote the \( k \)th coordinate vector in \( d \) dimensions.

For \( f : \mathbb{R}^n \to \mathbb{R}^m \), we can compute the full Jacobian \( \frac{\partial f(x)}{\partial x} \in \mathbb{R}^{m \times n} \) with either \( n \) JVPs

\[
\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial x} I_n = \left[ \frac{\partial f(x)}{\partial x} e_1 \right] \ldots \left[ \frac{\partial f(x)}{\partial x} e_n \right],
\]

or \( m \) VJPs

\[
\frac{\partial f(x)}{\partial x}^\top = \frac{\partial f(x)}{\partial x}^\top I_m = \left[ \frac{\partial f(x)}{\partial x}^\top e_1 \right] \ldots \left[ \frac{\partial f(x)}{\partial x}^\top e_m \right].
\]

This is what the source code for \( \text{jax.jacfwd} \) and \( \text{jac.jacrev} \) does.

```python
n, m = 3, 2
x = jnp.ones(n)
Jx = jax.vmap(df, in_axes=(None, 0))(x, jnp.eye(n))
JxT = jax.vmap(dfT, in_axes=(None, 0))(x, jnp.eye(m))
print('Jacobian (forward AD):')
print(Jx)
print('Jacobian (reverse AD):')
print(JxT)
```

```
Jacobian (forward AD):
[[1. 2.]
 [0. 0.]
 [0. 2.]]
```

3.3 Example: Linearizing Dynamics at Many Points

For \( \dot{x} = f(x,u) \) with \( x \in \mathbb{R}^n \) and \( u \in \mathbb{R}^m \), recall the first-order Taylor approximation

\[
f(x,u) \approx f(\bar{x}_k, \bar{u}_k) + \frac{\partial f(\bar{x}_k, \bar{u}_k)}{\partial x}(x - \bar{x}) + \frac{\partial f(\bar{x}_k, \bar{u}_k)}{\partial u}(u - \bar{u}).
\]

\[
f(\bar{x}_k, \bar{u}_k) + \Delta x_k \quad \Delta t \quad \Delta u_k
\]
We want $A_k \Delta x_t$, $B_k \Delta u_t$, and $c_k$ for $\{(\bar{x}_k, \bar{u}_k)\}^K_{k=1}$ and $\{(\Delta x_t, \Delta u_t)\}^T_{t=1}$.

This scenario may correspond to evaluating Taylor approximations for $T$ perturbations $(\Delta x_t, \Delta u_t)$ that we want to test at the $K$ points $(\bar{x}_k, \bar{u}_k)$.

```python
# Inverted pendulum (with unit mass and unit length)
f = lambda x, u: jnp.array([x[1], 9.81*jnp.sin(x[0]) + u[0]])

def taylor(\bar{x}, \bar{u}, \Delta x, \Delta u):
    f_\bar{x}\bar{u}, A\Delta x = jax.jvp(lambda x: f(x, \bar{u}), (\bar{x},), (\Delta x,))
    f_\bar{x}\bar{u}, B\Delta u = jax.jvp(lambda u: f(\bar{x}, u), (\bar{u},), (\Delta u,))
    return f_\bar{x}\bar{u}, A\Delta x, B\Delta u

print(type(taylor))

n, m = 2, 1
K, T = 5, 10
\bar{x}, \bar{u} = jnp.ones((K, n)), jnp.ones((K, m))
\Delta x, \Delta u = jnp.ones((T, n)), jnp.ones((T, m))
taylor = jax.vmap(taylor, in_axes=(None, None, 0, 0))
print(type(taylor))
taylor = jax.vmap(taylor, in_axes=(0, 0, None, None))
print(type(taylor))

c, Ax, Bu = taylor(\bar{x}, \bar{u}, \Delta x, \Delta u)
print(c.shape)
print(Ax.shape)
print(Bu.shape)
```

If, instead, we have $K = 5$ trajectories $\{(\bar{x}_k, \bar{u}_k)\}^K_{k=1}$ and each trajectory $\bar{x}_k$ has $T = 10$ timesteps $\{(\bar{x}_{k,t}, \bar{u}_{k,t})\}^T_{t=1}$, and similarly for $(\Delta x, \Delta u)$, then we can evaluate Taylor approximations for all these trajectories with two calls to vmap as below.

```python
# Inverted pendulum (with unit mass and unit length)
f = lambda x, u: jnp.array([x[1], 9.81*jnp.sin(x[0]) + u[0]])

def taylor(\bar{x}, \bar{u}, \Delta x, \Delta u):
    f_\bar{x}\bar{u}, A\Delta x = jax.jvp(lambda x: f(x, \bar{u}), (\bar{x},), (\Delta x,))
    f_\bar{x}\bar{u}, B\Delta u = jax.jvp(lambda u: f(\bar{x}, u), (\bar{u},), (\Delta u,))
    return f_\bar{x}\bar{u}, A\Delta x, B\Delta u
```
\[ n, m = 2, 1 \]
\[ K, T = 5, 10 \]
\[ \bar{x} = \text{jnp.ones}((K, T, n)) \quad \# \text{note the different sizes} \]
\[ \bar{u} = \text{jnp.ones}((K, T, m)) \]
\[ \Delta x, \Delta u = \text{jnp.ones}((K, T, n)), \text{jnp.ones}((K, T, m)) \]

# two successive calls to vmap:
# we linearize for the K trajectories that each have T timesteps
taylor = \text{jax.vmap}(taylor)
taylor = \text{jax.vmap}(taylor)

c, Ax, Bu = taylor(\bar{x}, \bar{u}, \Delta x, \Delta u)

4 Other Features and Nuances of JAX

See the JAX documentation for more details.

4.1 Just-In-Time (JIT) Compilation

JAX can compile code to run fast on both CPUs and GPUs. The first call to a “jitted” function will compile and cache the function; subsequent calls are then much faster.

[ ]: \text{def selu}(x, alpha=1.67, lmbda=1.05):
    \return lmbda * \text{jnp.where}(x > 0, x, alpha * \text{jnp.exp}(x) - alpha)

x = \text{jnp.ones}(\text{int}(1e7))
\text{\%timeit} -r10 -n100 selu(x).\text{block\_until\_ready}()

selu_jit = \text{jax.jit}(selu)
\text{\%timeit} -r10 -n100 selu_jit(x).\text{block\_until\_ready}()

42.5 ms ± 2.95 ms per loop (mean ± std. dev. of 10 runs, 100 loops each)
10.4 ms ± 426 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)

4.2 In-Place Updates

JAX arrays are immutable. In keeping with the functional programming paradigm, updates to array values at indices are done via JAX functions.

[ ]: X = \text{jnp.zeros}((3,3))
\text{try:}
except Exception as e:
    print("Exception: {}".format(e))
print('\nX:
X, sep='')
Y = X.at[0, :].set(1.)
print('\nY:
Y, sep='')

Exception: '<class 'jaxlib.xla_extension.DeviceArray'>' object does not support item assignment. JAX arrays are immutable. Instead of `x[idx] = y`, use `x = x.at[idx].set(y)` or another .at[] method:

X:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

Y:
[[1. 1. 1.]
 [0. 0. 0.]
 [0. 0. 0.]]

4.3 Pseudo-Random Number Generation (PRNG)

JAX does explicit PRNG; after initializing a PRNG state, it can be forked into new PRNG states for parallel stochastic generation.

This enables reproducible results; propagate the key and make new subkeys whenever new random numbers are needed.

seed = 0
key = jax.random.PRNGKey(seed)
print(jax.random.normal(key, shape=(1,)))
print(jax.random.normal(key, shape=(1,)))  # same value sampled!

print('\nkey', key)

key, *subkeys = jax.random.split(key, 3)
print('|-- SPLIT --&gt; key ', key)
print(' &gt;&gt;&gt; subkeys', subkeys[0],
     ' &gt;&gt;&gt; normal', jax.random.normal(subkeys[0], shape=(1,)))
print(' ', subkeys[1],
     ' &gt;&gt;&gt; normal', jax.random.normal(subkeys[1], shape=(1,)))

[-0.20584226]
[-0.20584226]

key [0 0]
|-- SPLIT --&gt; key [2467461003 428148500]
--> subkeys [3186719485 3840466878] --> normal [0.5781488]
[2562233961 1946702221] --> normal [0.8535516]