AutoFD: Automatic Functional Differentiation in JAX

Min Lin, Sea AI Lab

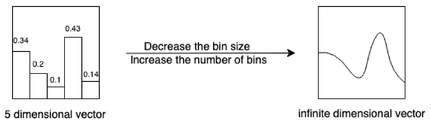
TLDR

Given: $F[f] = \int \exp(f(x))dx$ Find: $\frac{\delta F[f]}{\delta f}$

```python
from autofd import function
import autofd.operators as o

# define the functional
def F(f):
  return o.integrate(o.compose(jnp.exp, f))
# define a function
@function
def f(x: Float32[Array, ""]) -> Float32[Array, ""]:
  ...
# take the functional derivative
dFdf = jax.grad(F)(f)
# dFdf is invokable!
dFdf(1.)
```

Functions generalize vectors

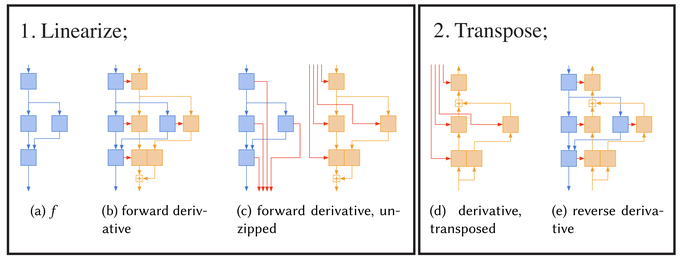
	Vector $\boldsymbol{v}$	Function f
Visualize		
Indexing	$\boldsymbol{v}_i$ (discrete)	$f(x)$ (continuous)
Summation	$\sum_i v_i$	$\int f(x)dx$
Linear projection	$\boldsymbol{u} = M^\top \boldsymbol{v}$	$g(y) = \int K(y,x)f(x)dx$
Differentiation	$\frac{df(\boldsymbol{v})}{d\boldsymbol{v}}$	$\frac{\delta F[f]}{\delta f}$

Types of Functionals & Operators

Local Operator	$\hat{\Phi}_L(f) : x \mapsto \phi_L(x, f(x)).$
Semilocal Operator	$\hat{\Phi}_S(f) : x \mapsto \phi_S(x, f(x), \nabla f(x), \cdots, \nabla^{(n)} f(x)).$
Linear Nonlocal Operator	$\hat{\Phi}_I(f) : u \mapsto \int \phi_I(u,x)f(x)dx.$
Integral Functional	$F : f \mapsto \int \hat{\Phi}(f)(x)dx.$

Autodiff Mechanism of JAX

Radul et al. (2023)



Example jvp and transpose rules

```python
def exp_jvp_rule(x, tangent):
  return exp(x) * tangent
```

```python
def matvec_transpose(cotangent, m, v):
  if ad.is_undefined_primal(m):
    return [cotangent @ v.T, None]
  else:
    return [None, m.T @ cotangent]
```

Relating JAX API to math symbols

$$\texttt{jax.jvp}(f, \underbrace{x}_{\text{primal}}, \underbrace{\delta x}_{\text{tangent}}) = J_f(x)\delta x = \underbrace{D(f)(x)(\delta x)}_{\text{Fréchet notation}}.$$

$$\texttt{jax.linear\_transpose}(f, \underbrace{x}_{\text{primal}})(\underbrace{y}_{\text{cotangent}}) = f^\top(y) = T(f)(x)(y)$$

AD for Functionals & Operators

	JVP ⟶ Gateaux derivative	Transpose ⟶ Conjugate operator
Example:	**Compose operator** $\hat{C}(f,g) : x \mapsto f(g(x))$	**Nabla operator** $\nabla(f) : x \mapsto f'(x)$
Python:	`def compose_impl(f, g):` `  return lambda *args: f(g(*args))`	`def nabla_impl(f):` `  return jax.grad(f)`
Math:	$\partial_g(\hat{C})(f,g)(\delta g)(x)$ $= \lim_{\tau\to 0} \frac{f(g(x)+\tau\delta g(x)) - f(g(x))}{\tau}$ $= \frac{d}{d\tau} f(g(x)+\tau\delta g(x))\,|_{\tau=0}$ $= \nabla(f)(g(x)) \cdot \delta g(x)$ $= \hat{C}(\hat{L}(f), g, \delta g)(x)$	Conjugate operator $\langle \hat{O}f, g\rangle = \langle f, \hat{O}^* g\rangle$ $\langle \nabla f, g\rangle = \int dy g(y) \int dx \delta'(x-y) f(x)$ $= \int dx f(x) \int dy \delta'(x-y) g(y)$ $= \langle f, -\nabla g\rangle$

JAX rules:	$\partial_f(\hat{C})(f,g)(\delta f)(x)$
	$= \lim_{\tau\to 0} \frac{f(g(x))+\tau\delta f(g(x)) - f(g(x))}{\tau}$
	$= \delta f(g(x))$
	$= \hat{C}(\delta f, g)(x)$
	`def compose_jvp_rule(primals, tangents):`
	`  f, g = primals`
	`  df, dg = tangents`
	`  primal_out = compose(f, g)`
	`  tangent_out = sum(`
	`  compose(linearize(f), g, dg),`
	`  compose(df, g),`
	`  )`
	`  return primal_out, tangent_out`
		`def nabla_transpose_rule(cotangent, primal):`
		`  return -nabla(cotangent)`

Primitive Operators

$$\hat{C}(f,g) : x \mapsto f(g(x))$$
$$\nabla(f) : x \mapsto f'(x)$$
$$\hat{L}(f) : x, \delta x \mapsto f'(x)\delta x$$
$$\hat{T}(f) : x \mapsto f^\top(x)$$
$$\hat{I} : f \mapsto \int f(x)dx$$

$$\text{PermuteArgs}(f) : x, y \mapsto f(y, x)$$
$$\text{Zip}(f,g) : x, y \mapsto f(x), g(y)$$
$$\text{Partial}(f, x) : y \mapsto f(x, y)$$
$$\cdots$$

Neural Operator

```python
def f(x: Float32[Array, ""]) -> Float32[Array, ""]:
  return jnp.sin(4 * x * jnp.pi)

def b(x: Float32[Array, ""]) -> Float32[Array, ""]:
  return jnp.sin(x * jnp.pi)

def y(x: Float32[Array, ""]) -> Float32[Array, ""]:
  return jnp.cos(x * jnp.pi)

def k(y: Float32[Array, ""], x: Float32[Array, ""])
  return jnp.sin(y) + jnp.cos(x)

def layer(k, b, f, activate=True):
  # here k @ f is syntatic sugar for
  # o.integrate(k * broadcast(f), argnums=1)
  g = k @ f + b
  if activate:
    a = o.numpy.tanh(g)
    return a
  else:
    return g

def loss(params, f, t):
  # two layer mlp
  k1, b1, k2, b2 = params
  h1 = layer(k1, b1, f, activation=True)
  h2 = layer(k2, b2, h1, activation=False)
  return o.integrate((h2 - t)**2)
```

Density Functional Theory

Exchange Correlation Functionals

1. Mostly semi-local
2. Derivative of energy functional gives potential.
3. Crucial for constructing Kohn-Sham Hamiltonian.
4. Traditionally hand waved using Euler-Lagrange.

```python
from autofd.operators import integrate
# a point in 3D
r = jnp.array([0.1, 0.2, 0.3])
# Exc, exchange-correlation energy functional
def exc(rho: Callable) -> Float:
  epsilon_xc = gga_x_pbe(rho)
  return integrate(epsilon_xc * rho)
# Vxc is the functional derivative of Exc
vxc = jax.grad(exc)(rho)
# Vxc is itself a function callable with r as input
vxc(r)
```

Calculating Band Structure for Solids

1. Physicists write simple equations in function space.
2. Implementation is in coefficient space.
3. One of the difficulties in scientific computing is the math to implementation gap.

$$\left(-\frac{1}{2}\nabla^2 + \hat{V}_{\text{eff}}\right)\psi = \epsilon\psi$$

$$\hat{V}_{\text{eff}} = \frac{\delta E[\rho]}{\delta\rho}$$

```python
def band_structure(crystal, fixed_rho):

  def potential_energy(rho):
    return (
      e.hartree(rho) +
      e.external(rho, crystal) +
      jax_xc.energy.lda_x(rho)
    )

  _, eff_energy = jax.linearize(potential_energy, (fixed_rho,))

  def energy_under_veff(param, k):
    psi = o.partial(wave_ansatz, args=(param,), argnums=(0,))
    return e.kinetic(psi) + eff_energy(psi_to_rho(psi))

  bands = {}
  for k in k_vectors:
    bands[k] = jnp.eigh(jax.hessian(energy_under_veff)(param, k))[0]
  return bands
```