JAX

The core of JAX in brief, NumPy that runs on GPUs and TPUs with composable transforms for autodiff, compilation, and vmap.

Arrays & Devices

Make arrays, see your accelerator, move data onto it.

PURPOSE	COMMAND	RESULT
Array from a Python list.	`jnp.array([1., 2., 3.])`	
Range / evenly spaced.	`jnp.arange(5) jnp.linspace(0,1,3)`	
Convert a NumPy array.	`jnp.asarray(np_array)`	
List the devices.	`jax.devices() jax.default_backend()`	
Put data on a device.	`jax.device_put(x, devices()[0])`	
Wait for async compute.	`x.block_until_ready()`	

a JAX array is immutable and lives on the accelerator; the default float dtype is float32, not float64

NumPy API & Immutable Updates

jnp mirrors numpy, but arrays never change in place.

PURPOSE	COMMAND	RESULT
Element-wise math (vectorized).	`jnp.exp(x) x * 2 + 1`	
Matrix multiply.	`a @ bjnp.dot(a, b)`	
Reduce over an axis.	`x.sum(axis=0) x.mean()`	
Indexing reads like numpy.	`m[1, 2] m[:, 0]`	
Functional "set" (returns copy).	`x.at[0].set(99)`	
Functional "add" at index.	`x.at[1].add(10)`	

arrays cannot be mutated: x.at[idx].set(...) / .add(...) returns a new array and leaves the original untouched

Automatic Differentiation

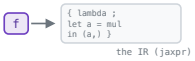
grad turns a function into its gradient function.

PURPOSE	COMMAND	RESULT
Gradient of a scalar function.	`grad(f)(2.0)`	
Value and gradient together.	`value_and_grad(loss)(w, x, y)`	
Differentiate chosen args.	`grad(loss, argnums=(0, 1))(w, b, x)`	
Higher-order derivative.	`grad(grad(f))(2.0)`	
Full Jacobian matrix.	`jax.jacobian(f)(x)`	
Hessian (curvature) matrix.	`jax.hessian(g)(x)`	

grad is a higher-order transform: give it a scalar function, get a new function computing the gradient

JIT Compilation

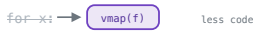
jit traces a function and compiles it with XLA.

PURPOSE	COMMAND	RESULT
Compile a function.	`jit(f)@jit`	
Trace once, then reuse.	`f(x); f(x)# 2nd call fast`	
Mark compile-time constants.	`partial(jit, static_argnums=(1,))`	
Inspect the traced program.	`jax.make_jaxpr(f)(x)`	
Time it correctly.	`f(x).block_until_ready()`	
Avoid: Python side effects.	`print()/ .item() inside a jit'd fn`	

jit traces once on shaped inputs and caches the XLA binary; the function must be pure, no Python if/for over array values

Auto-Vectorization (vmap)

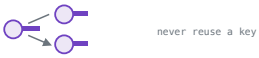
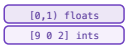
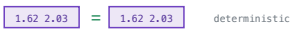
Write the single-example function; vmap adds the batch axis.

PURPOSE	COMMAND	RESULT
Batch a per-example function.	`vmap(f)(batch)`	
Map over the leading axis.	`vmap(dot)(A, B)`	
Choose which axis to map.	`vmap(f, in_axes=(0, None))`	
Control the output axis.	`vmap(f, out_axes=1)`	
Compose with grad (per-sample).	`vmap(grad(f))(xs)`	
The payoff: no batch loop.	~~`for x in batch:`~~ `vmap(f)`	

write a function for one example; in_axes/out_axes set the batch axis (None holds an arg fixed); composes with grad and jit

Random Numbers (Explicit Keys)

No global seed: you pass and split an explicit key.

PURPOSE	COMMAND	RESULT
Create a key from a seed.	`key = jax.random.key(0)`	
Split into independent keys.	`k1, k2 = jax.random.split(key)`	
Draw standard normals.	`jax.random.normal(key, (3,))`	
Uniform / integer draws.	`jax.random.uniform` / randint(key,...)	
Per-step keys in a loop.	`jax.random.fold_in(key, step)`	
Same key, same draw.	`random.normal(key, (3,)) x2`	

the golden rule: never reuse a key; split it into fresh independent keys before each new draw

Control Flow & Loops

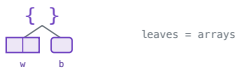
Inside jit, use lax primitives instead of Python branches.

PURPOSE	COMMAND	RESULT
Data-dependent branch.	`lax.cond(p, true_fn, false_fn, x)`	
Fixed-count loop.	`lax.fori_loop(0, 5, body, init)`	
Loop while a condition holds.	`lax.while_loop(cond, body, init)`	
Carry + collect (the RNN loop).	`lax.scan(body, init, xs)`	
Why not a Python for?	`forover a traced array -> unrolls`	
Stop a gradient flowing.	`jax.lax.stop_gradient(x)`	

inside a jit/grad trace, a Python if/for runs on the trace; data-dependent flow uses cond / fori_loop / while_loop / scan

Pytrees & Parameters

Nested dicts/lists of arrays are first-class; map over them.

PURPOSE	COMMAND	RESULT
A parameter pytree.	`params = {"w": ..., "b": ...}`	
Map a function over leaves.	`jax.tree.map(lambda x: x*2, params)`	
List just the leaves.	`jax.tree.leaves(params)`	
Flatten / unflatten.	`jax.tree.flatten(params)`	
One SGD step over a tree.	`jax.tree.map(lambda p,g: p - lr*g, params, grads)`	
Grad returns a matching tree.	`grad(loss)(params, batch)`	

a pytree is any nested dict/list/tuple of arrays; jax.tree.map applies over every leaf, preserving structure