int to PDF

Just In Time Compilation with JAX

Contents

Open in Colab Open in Kaggle

Authors: Rosalia Schneider & Vladimir Mikulik

In this section, we will further explore how JAX works, and how we can make it performant. We will discuss the `jax.jit()` transform, which will perform *Just In Time* (JIT) compilation of a JAX Python function so it can be executed efficiently in XLA.

How JAX transforms work

In the previous section, we discussed that JAX allows us to transform Python functions. This is done by first converting the Python function into a simple intermediate language called jaxpr. The transformations then work on the jaxpr representation.

We can show a representation of the jaxpr of a function by using `jax.make_jaxpr`:

```
import jax
import jax.numpy as jnp

global_list = []

def log2(x):
  global_list.append(x)
  ln_x = jnp.log(x)
  ln_2 = jnp.log(2.0)
```

latest

Skip to main content

```
print(jax.make_jaxpr(log2)(3.0))
```

```
{ lambda ; a:f32[]. let
    b:f32[] = log a
    c:f32[] = log 2.0
    d:f32[] = div b c
  in (d,) }
```

The Understanding Jaxprs section of the documentation provides more information on the meaning of the above output.

Importantly, note how the jaxpr does not capture the side-effect of the function: there is nothing in it corresponding to `global_list.append(x)`. This is a feature, not a bug: JAX is designed to understand side-effect-free (a.k.a. functionally pure) code. If *pure function* and *side-effect* are unfamiliar terms, this is explained in a little more detail in 🔪 JAX - The Sharp Bits 🔪: Pure Functions.

Of course, impure functions can still be written and even run, but JAX gives no guarantees about their behaviour once converted to jaxpr. However, as a rule of thumb, you can expect (but shouldn't rely on) the side-effects of a JAX-transformed function to run once (during the first call), and never again. This is because of the way that JAX generates jaxpr, using a process called 'tracing'.

When tracing, JAX wraps each argument by a *tracer* object. These tracers then record all JAX operations performed on them during the function call (which happens in regular Python). Then, JAX uses the tracer records to reconstruct the entire function. The output of that reconstruction is the jaxpr. Since the tracers do not record the Python side-effects, they do not appear in the jaxpr. However, the side-effects still happen during the trace itself.

Note: the Python `print()` function is not pure: the text output is a side-effect of the function. Therefore, any `print()` calls will only happen during tracing, and will not appear in the jaxpr:

```
def log2_with_print(x):
  print("printed x:", x)
  ln_x = jnp.log(x)
  ln_2 = jnp.log(2.0)
  return ln_x / ln_2

print(jax.make_jaxpr(log2_with_print)(3.))
```

```
    b:f32[] = log a
    c:f32[] = log 2.0
    d:f32[] = div b c
  in (d,) }
```

See how the printed `x` is a `Traced` object? That's the JAX internals at work.

The fact that the Python code runs at least once is strictly an implementation detail, and so shouldn't be relied upon. However, it's useful to understand as you can use it when debugging to print out intermediate values of a computation.

A key thing to understand is that jaxpr captures the function as executed on the parameters given to it. For example, if we have a conditional, jaxpr will only know about the branch we take:

```
def log2_if_rank_2(x):
  if x.ndim == 2:
    ln_x = jnp.log(x)
    ln_2 = jnp.log(2.0)
    return ln_x / ln_2
  else:
    return x

print(jax.make_jaxpr(log2_if_rank_2)(jax.numpy.array([1, 2, 3])))
```

```
{ lambda ; a:i32[3]. let  in (a,) }
```

JIT compiling a function

As explained before, JAX enables operations to execute on CPU/GPU/TPU using the same code. Let's look at an example of computing a *Scaled Exponential Linear Unit* (SELU), an operation commonly used in deep learning:

```
import jax
import jax.numpy as jnp

def selu(x, alpha=1.67, lambda_=1.05):
  return lambda_ * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

x = jnp.arange(1000000)
%timeit selu(x).block_until_ready()
```

The code above is sending one operation at a time to the accelerator. This limits the ability of the XLA compiler to optimize our functions.

Naturally, what we want to do is give the XLA compiler as much code as possible, so it can fully optimize it. For this purpose, JAX provides the `jax.jit` transformation, which will JIT compile a JAX-compatible function. The example below shows how to use JIT to speed up the previous function.

```
selu_jit = jax.jit(selu)

# Warm up
selu_jit(x).block_until_ready()

%timeit selu_jit(x).block_until_ready()
```

```
10000 loops, best of 5: 150 µs per loop
```

Here's what just happened:

1. We defined `selu_jit` as the compiled version of `selu`.
2. We called `selu_jit` once on `x`. This is where JAX does its tracing – it needs to have some inputs to wrap in tracers, after all. The jaxpr is then compiled using XLA into very efficient code optimized for your GPU or TPU. Finally, the compiled code is executed to satisfy the call. Subsequent calls to `selu_jit` will use the compiled code directly, skipping the python implementation entirely.

(If we didn't include the warm-up call separately, everything would still work, but then the compilation time would be included in the benchmark. It would still be faster, because we run many loops in the benchmark, but it wouldn't be a fair comparison.)

3. We timed the execution speed of the compiled version. (Note the use of `block_until_ready()`, which is required due to JAX's Asynchronous execution model).

Why can't we just JIT everything?

After going through the example above, you might be wondering whether we should simply apply `jax.jit` to every function. To understand why this is not the case, and when we should/shouldn't apply `jit`, let's first check some cases where JIT doesn't work.

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

f_jit = jax.jit(f)
f_jit(10)  # Should raise an error.
```

```
---------------------------------------------------------------------------
UnfilteredStackTrace                      Traceback (most recent call last)
<ipython-input-12-2c1a07641e48> in <module>()
      9 f_jit = jax.jit(f)
---> 10 f_jit(10)  # Should raise an error.

/usr/local/lib/python3.7/dist-packages/jax/_src/traceback_util.py in reraise_with_fi
    161     try:
--> 162       return fun(*args, **kwargs)
    163     except Exception as e:

/usr/local/lib/python3.7/dist-packages/jax/_src/api.py in cache_miss(*args, **kwargs
    418         device=device, backend=backend, name=flat_fun.__name__,
--> 419         donated_invars=donated_invars, inline=inline)
    420     out_pytree_def = out_tree()

/usr/local/lib/python3.7/dist-packages/jax/core.py in bind(self, fun, *args, **param
   1631   def bind(self, fun, *args, **params):
-> 1632     return call_bind(self, fun, *args, **params)
   1633

/usr/local/lib/python3.7/dist-packages/jax/core.py in call_bind(primitive, fun, *arg
   1622   tracers = map(top_trace.full_raise, args)
-> 1623   outs = primitive.process(top_trace, fun, tracers, params)
   1624   return map(full_lower, apply_todos(env_trace_todo(), outs))

/usr/local/lib/python3.7/dist-packages/jax/core.py in process(self, trace, fun, trac
   1634   def process(self, trace, fun, tracers, params):
-> 1635     return trace.process_call(self, fun, tracers, params)
   1636

/usr/local/lib/python3.7/dist-packages/jax/core.py in process_call(self, primitive,
    626   def process_call(self, primitive, f, tracers, params):
--> 627     return primitive.impl(f, *tracers, **params)
    628   process_map = process_call

/usr/local/lib/python3.7/dist-packages/jax/interpreters/xla.py in _xla_call_impl(**
    687   compiled_fun = _xla_callable(fun, device, backend, name, donated_invars,
--> 688                                *unsafe_map(arg_spec, args))
    689   try:

/usr/local/lib/python3.7/dist-packages/jax/linear_util.py in memoized_fun(fun, *args
    262     else:
--> 263       ans = call(fun, *args)
    264       cache[key] = (ans, fun.stores)

/usr/local/lib/python3.7/dist-packages/jax/interpreters/xla.py in _xla_callable_unca
```

```
    761

/usr/local/lib/python3.7/dist-packages/jax/interpreters/xla.py in lower_xla_callable
    771   jaxpr, out_avals, consts = pe.trace_to_jaxpr_final(
--> 772       fun, abstract_args, pe.debug_info_final(fun, "jit"))
    773   if any(isinstance(c, core.Tracer) for c in consts):

/usr/local/lib/python3.7/dist-packages/jax/interpreters/partial_eval.py in trace_to_
   1541     with core.new_sublevel():
-> 1542       jaxpr, out_avals, consts = trace_to_subjaxpr_dynamic(fun, main, in_ava
   1543     del fun, main

/usr/local/lib/python3.7/dist-packages/jax/interpreters/partial_eval.py in trace_to_
   1519     in_tracers = map(trace.new_arg, in_avals)
-> 1520     ans = fun.call_wrapped(*in_tracers)
   1521     out_tracers = map(trace.full_raise, ans)

/usr/local/lib/python3.7/dist-packages/jax/linear_util.py in call_wrapped(self, *arg
    165     try:
--> 166       ans = self.f(*args, **dict(self.params, **kwargs))
    167     except:

<ipython-input-12-2c1a07641e48> in f(x)
      3 def f(x):
----> 4   if x > 0:
      5     return x

/usr/local/lib/python3.7/dist-packages/jax/core.py in __bool__(self)
    548   def __nonzero__(self): return self.aval._nonzero(self)
--> 549   def __bool__(self): return self.aval._bool(self)
    550   def __int__(self): return self.aval._int(self)

/usr/local/lib/python3.7/dist-packages/jax/core.py in error(self, arg)
    999   def error(self, arg):
-> 1000     raise ConcretizationTypeError(arg, fname_context)
   1001   return error

UnfilteredStackTrace: jax._src.errors.ConcretizationTypeError: Abstract tracer value
The problem arose with the bool function.
While tracing the function f at <ipython-input-12-2c1a07641e48>:3 for jit, this conc

See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeEr

The stack trace below excludes JAX-internal frames.
The preceding is the original exception that occurred, unmodified.

--------------------

The above exception was the direct cause of the following exception:

ConcretizationTypeError                 Traceback (most recent call last)
<ipython-input-12-2c1a07641e48> in <module>()
      8
      9 f_jit = jax.jit(f)
---> 10 f_jit(10)  # Should raise an error.

<ipython-input-12-2c1a07641e48> in f(x)
      2
```

```
      5     return x
      6   else:

ConcretizationTypeError: Abstract tracer value encountered where concrete value is e
The problem arose with the bool function.
While tracing the function f at <ipython-input-12-2c1a07641e48>:3 for jit, this conc

See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeEr
```

```
# While loop conditioned on x and n.

def g(x, n):
  i = 0
  while i < n:
    i += 1
  return x + i

g_jit = jax.jit(g)
g_jit(10, 20)  # Should raise an error.
```

```
---------------------------------------------------------------------------
UnfilteredStackTrace                      Traceback (most recent call last)
<ipython-input-13-2aa78f448d5d> in <module>()
      9 g_jit = jax.jit(g)
---> 10 g_jit(10, 20)  # Should raise an error.

/usr/local/lib/python3.7/dist-packages/jax/_src/traceback_util.py in reraise_with_fi
    161     try:
--> 162       return fun(*args, **kwargs)
    163     except Exception as e:

/usr/local/lib/python3.7/dist-packages/jax/_src/api.py in cache_miss(*args, **kwargs
    418         device=device, backend=backend, name=flat_fun.__name__,
--> 419         donated_invars=donated_invars, inline=inline)
    420     out_pytree_def = out_tree()

/usr/local/lib/python3.7/dist-packages/jax/core.py in bind(self, fun, *args, **param
   1631   def bind(self, fun, *args, **params):
-> 1632     return call_bind(self, fun, *args, **params)
   1633

/usr/local/lib/python3.7/dist-packages/jax/core.py in call_bind(primitive, fun, *arg
   1622   tracers = map(top_trace.full_raise, args)
-> 1623   outs = primitive.process(top_trace, fun, tracers, params)
   1624   return map(full_lower, apply_todos(env_trace_todo(), outs))

/usr/local/lib/python3.7/dist-packages/jax/core.py in process(self, trace, fun, trac
   1634   def process(self, trace, fun, tracers, params):
-> 1635     return trace.process_call(self, fun, tracers, params)
   1636

/usr/local/lib/python3.7/dist-packages/jax/core.py in process_c
    626   def process_call(self, primitive, f, tracers, params):
```

```
/usr/local/lib/python3.7/dist-packages/jax/interpreters/xla.py in _xla_call_impl(**
    687   compiled_fun = _xla_callable(fun, device, backend, name, donated_invars,
--> 688                                *unsafe_map(arg_spec, args))
    689   try:

/usr/local/lib/python3.7/dist-packages/jax/linear_util.py in memoized_fun(fun, *args
    262     else:
--> 263       ans = call(fun, *args)
    264       cache[key] = (ans, fun.stores)

/usr/local/lib/python3.7/dist-packages/jax/interpreters/xla.py in _xla_callable_unca
    759   return lower_xla_callable(fun, device, backend, name, donated_invars,
--> 760                             *arg_specs).compile().unsafe_call
    761

/usr/local/lib/python3.7/dist-packages/jax/interpreters/xla.py in lower_xla_callable
    771   jaxpr, out_avals, consts = pe.trace_to_jaxpr_final(
--> 772       fun, abstract_args, pe.debug_info_final(fun, "jit"))
    773   if any(isinstance(c, core.Tracer) for c in consts):

/usr/local/lib/python3.7/dist-packages/jax/interpreters/partial_eval.py in trace_to_
   1541     with core.new_sublevel():
-> 1542       jaxpr, out_avals, consts = trace_to_subjaxpr_dynamic(fun, main, in_ava
   1543     del fun, main

/usr/local/lib/python3.7/dist-packages/jax/interpreters/partial_eval.py in trace_to_
   1519     in_tracers = map(trace.new_arg, in_avals)
-> 1520     ans = fun.call_wrapped(*in_tracers)
   1521     out_tracers = map(trace.full_raise, ans)

/usr/local/lib/python3.7/dist-packages/jax/linear_util.py in call_wrapped(self, *arg
    165     try:
--> 166       ans = self.f(*args, **dict(self.params, **kwargs))
    167     except:

<ipython-input-13-2aa78f448d5d> in g(x, n)
      4   i = 0
----> 5   while i < n:
      6     i += 1

/usr/local/lib/python3.7/dist-packages/jax/core.py in __bool__(self)
    548   def __nonzero__(self): return self.aval._nonzero(self)
--> 549   def __bool__(self): return self.aval._bool(self)
    550   def __int__(self): return self.aval._int(self)

/usr/local/lib/python3.7/dist-packages/jax/core.py in error(self, arg)
    999   def error(self, arg):
-> 1000     raise ConcretizationTypeError(arg, fname_context)
   1001   return error

UnfilteredStackTrace: jax._src.errors.ConcretizationTypeError: Abstract tracer value
The problem arose with the bool function.
While tracing the function g at <ipython-input-13-2aa78f448d5d>:3 for jit, this conc

See https://jax.readthedocs.io/en/latest/errors.html#jax.errors

The stack trace below excludes JAX-internal frames.
```

```
--------------------

The above exception was the direct cause of the following exception:

ConcretizationTypeError                 Traceback (most recent call last)
<ipython-input-13-2aa78f448d5d> in <module>()
      8
      9 g_jit = jax.jit(g)
---> 10 g_jit(10, 20)  # Should raise an error.

<ipython-input-13-2aa78f448d5d> in g(x, n)
      3 def g(x, n):
      4   i = 0
----> 5   while i < n:
      6     i += 1
      7   return x + i

ConcretizationTypeError: Abstract tracer value encountered where concrete value is e
The problem arose with the bool function.
While tracing the function g at <ipython-input-13-2aa78f448d5d>:3 for jit, this conc

See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeEr
```

The problem is that we tried to condition on the *value* of an input to the function being jitted. The reason we can't do this is related to the fact mentioned above that jaxpr depends on the actual values used to trace it.

The more specific information about the values we use in the trace, the more we can use standard Python control flow to express ourselves. However, being too specific means we can't reuse the same traced function for other values. JAX solves this by tracing at different levels of abstraction for different purposes.

For `jax.jit`, the default level is `ShapedArray` – that is, each tracer has a concrete shape (which we're allowed to condition on), but no concrete value. This allows the compiled function to work on all possible inputs with the same shape – the standard use case in machine learning. However, because the tracers have no concrete value, if we attempt to condition on one, we get the error above.

In `jax.grad`, the constraints are more relaxed, so you can do more. If you compose several transformations, however, you must satisfy the constraints of the most strict one. So, if you `jit(grad(f))`, `f` mustn't condition on value. For more detail on the interaction between Python control flow and JAX, see 🔪 JAX - The Sharp Bits 🔪: Control Flow.

One way to deal with this problem is to rewrite the code to avoid conditionals on value. Another is to use special control flow operators like `jax.lax.cond`. Ho
is impossible. In that case, you can consider jitting only part of the func

that inner part (though make sure to check the next section on caching to avoid shooting yourself in the foot):

```
# While loop conditioned on x and n with a jitted body.

@jax.jit
def loop_body(prev_i):
  return prev_i + 1

def g_inner_jitted(x, n):
  i = 0
  while i < n:
    i = loop_body(i)
  return x + i

g_inner_jitted(10, 20)
```

```
Array(30, dtype=int32, weak_type=True)
```

If we really need to JIT a function that has a condition on the value of an input, we can tell JAX to help itself to a less abstract tracer for a particular input by specifying `static_argnums` or `static_argnames`. The cost of this is that the resulting jaxpr is less flexible, so JAX will have to re-compile the function for every new value of the specified static input. It is only a good strategy if the function is guaranteed to get limited different values.

```
f_jit_correct = jax.jit(f, static_argnums=0)
print(f_jit_correct(10))
```

```
10
```

```
g_jit_correct = jax.jit(g, static_argnames=['n'])
print(g_jit_correct(10, 20))
```

```
30
```

To specify such arguments when using `jit` as a decorator, a common pattern is to use python's `functools.partial`:

```
from functools import partial

@partial(jax.jit, static_argnames=['n'])
```

```
  while i < n:
    i += 1
  return x + i

print(g_jit_decorated(10, 20))
```

```
30
```

When to use JIT

In many of the examples above, jitting is not worth it:

```
print("g jitted:")
%timeit g_jit_correct(10, 20).block_until_ready()

print("g:")
%timeit g(10, 20)
```

```
g jitted:
The slowest run took 13.54 times longer than the fastest. This could mean that an ir
1000 loops, best of 5: 229 µs per loop
g:
The slowest run took 11.72 times longer than the fastest. This could mean that an ir
1000000 loops, best of 5: 1.2 µs per loop
```

This is because `jax.jit` introduces some overhead itself. Therefore, it usually only saves time if the compiled function is complex and you will run it numerous times. Fortunately, this is common in machine learning, where we tend to compile a large, complicated model, then run it for millions of iterations.

Generally, you want to jit the largest possible chunk of your computation; ideally, the entire update step. This gives the compiler maximum freedom to optimise.

Caching

It's important to understand the caching behaviour of `jax.jit`.

Suppose I define `f = jax.jit(g)`. When I first invoke `f`, it will get compiled, and the resulting XLA code will get cached. Subsequent calls of `f` will reuse the

how `jax.jit` makes up for the up-front cost of compilation.

If I specify `static_argnums`, then the cached code will be used only for the same values of arguments labelled as static. If any of them change, recompilation occurs. If there are many values, then your program might spend more time compiling than it would have executing ops one-by-one.

Avoid calling `jax.jit` inside loops. For most cases, JAX will be able to use the compiled, cached function in subsequent calls to `jax.jit`. However, because the cache relies on the hash of the function, it becomes problematic when equivalent functions are redefined. This will cause unnecessary compilation each time in the loop:

```
from functools import partial

def unjitted_loop_body(prev_i):
  return prev_i + 1

def g_inner_jitted_partial(x, n):
  i = 0
  while i < n:
    # Don't do this! each time the partial returns
    # a function with different hash
    i = jax.jit(partial(unjitted_loop_body))(i)
  return x + i

def g_inner_jitted_lambda(x, n):
  i = 0
  while i < n:
    # Don't do this!, lambda will also return
    # a function with a different hash
    i = jax.jit(lambda x: unjitted_loop_body(x))(i)
  return x + i

def g_inner_jitted_normal(x, n):
  i = 0
  while i < n:
    # this is OK, since JAX can find the
    # cached, compiled function
    i = jax.jit(unjitted_loop_body)(i)
  return x + i

print("jit called in a loop with partials:")
%timeit g_inner_jitted_partial(10, 20).block_until_ready()

print("jit called in a loop with lambdas:")
%timeit g_inner_jitted_lambda(10, 20).block_until_ready()

print("jit called in a loop with caching:")
%timeit g_inner_jitted_normal(10, 20).block_until_ready()
```

```
jit called in a loop with partials:
1 loop, best of 5: 192 ms per loop
jit called in a loop with lambdas:
```

```
jit called in a loop with caching:
10 loops, best of 5: 21.6 ms per loop
```

Previous
JAX As Accelerated NumPy

Next
Automatic Vectorization in JAX