A JAX introduction for QML

February 2025

This document aims to introduce JAX to researchers and practitioners in Quantum Machine Learning (QML) who may not yet be familiar with it. We will cover the fundamentals of JAX and showcase why its particularly well suited for working with quantum circuits in the context of QML with PennyLane. However, this is not an in-depth technical deep dive. For a more comprehensive exploration, we recommend referring to the official JAX and PennyLane documentation, which provide excellent resources.

If you prefer reading the Jupyter notebook equivalent of this document you can find it in this GitHub repository.

1 Introduction to JAX

In short, JAX is an array-oriented numerical computing library that enables composable transformations. These include just-in-time (JIT) compilation, automatic vectorization and automatic differentiation. On top of that, since it uses the XLA (Accelerated Linear Algebra) compiler it can run on CPUs, GPUs and TPUs natively.

JAX provides an array interface that mimics NumPy and can be used as a drop-in replacement for NumPy arrays. The most notable difference between `numpy`, usually referred as `np`, arrays and `jax.numpy`, usually referred as `jnp`, is that the latter are always immutable.

```
# Create a NumPy array
np_array = np.array([1, 2, 3])
print("Original␣NumPy␣array:", np_array)
np_array[0] = 10  # Mutating a NumPy array is allowed
print("Modified␣NumPy␣array:", np_array)

# Create a JAX array
jnp_array = jnp.array([1, 2, 3])

try:
    jnp_array[0] = 10  # Attempting to mutate a JAX array
except TypeError as e:
    print("\n[!]␣JAX␣array␣mutation␣error:", e)

# Instead of mutating, we create a new array
new_jnp_array = jnp_array.at[0].set(10)
print("\nOriginal␣JAX␣array:", jnp_array)
print("New␣JAX␣array␣after␣modification:", new_jnp_array)

> Original NumPy array: [1 2 3]
> Modified NumPy array: [10  2  3]
> [!] JAX array mutation error: JAX arrays are immutable and
     do not support in-place item assignment. Instead of x[
    idx] = y, use x = x.at[idx].set(y) or another .at[]
    method: https://jax.readthedocs.io/en/latest/_autosummary
    /jax.numpy.ndarray.at.html
> Original JAX array: [1 2 3]
> New JAX array after modification: [10  2  3]
```

All JAX operations are implemented in terms of XLA. This means that whenever we use `jnp` we will be taking advantage of the accelerated linear algebra compiler. However, we also have a lower level API available with `jax.lax`, which contains wrappers for primitive XLA operations. All `jnp` operations are implemented in terms of `jax.lax`.

JAX has an internal representation of programs called jaxpr language. We can use the `jax.make_jaxpr` function to obtain the jaxpr representation of a function. This can be useful for finding out how certain functions get transformed to lower level operations. For instance, we can see how `jnp.dot` translates to a more general `jax.lax.dot_general`.

```
def high_level(x, y):
    return jnp.dot(x, y)

def low_level(x, y):
    return lax.dot_general(
        x, y,
        dimension_numbers=(((1,), (0,)), ((), ()))  # Batch
            dimensions are empty
    )

A = jnp.array([[1, 2], [3, 4]])
B = jnp.array([[5, 6], [7, 8]])
jaxpr_high_level = jax.make_jaxpr(high_level)(A, B)
jaxpr_low_level = jax.make_jaxpr(low_level)(A, B)

print("JAXpr␣for␣jnp.dot:\n", jaxpr_high_level)
print("JAXpr␣for␣lax.dot_general:\n", jaxpr_low_level)

> JAXpr for jnp.dot:
> { lambda ; a:i32[2,2] b:i32[2,2]. let
>     c:i32[2,2] = dot_general[
>       dimension_numbers=(([1], [0]), ([], []))
>       preferred_element_type=int32
>     ] a b
>   in (c,) }
>
> JAXpr for lax.dot_general:
> { lambda ; a:i32[2,2] b:i32[2,2]. let
>     c:i32[2,2] = dot_general[dimension_numbers=(([1], [0]),
     ([], []))] a b
>   in (c,) }
```

Along with a NumPy-like API of functions that operate on arrays, JAX also includes a number of composable transformations which operate on functions. The ones we are interested the most in are:

- `jax.jit()`: just-in-time compilation.
- `jax.vmap()`: vectorizing transform, i.e. automatic vectorization.
- `jax.grad()`: gradient transform, i.e. automatic differentiation.

To do transformations, JAX uses the concept of tracing a function. Tracing works by replacing the array inputs of a function by abstract placeholders with the same shape and type. This allows JAX to determine the sequence of operations of a function and their effect on input arrays, independently of their content.

We can see how JAX sees traced arrays by printing inside a function subject to a transform:

```
@jax.jit
def f(x):
  print("inside␣the␣function␣we␣see␣x␣as␣", x)
  return x + 1

x = jnp.arange(5)
print("outside␣the␣function␣we␣see␣x␣as␣", x)
y = f(x)
print("f(x)␣=␣", y)

> outside the function we see x as [0 1 2 3 4]
> inside the function we see x as Traced<ShapedArray(int32
    [5])>with<DynamicJaxprTrace>
> f(x) =  [1 2 3 4 5]
```

The printed value we see inside the function is not the array x, but rather an abstract traced representation that has the same shape and type. In this case, the JIT transform has used the traced information to create a compiled version of f.

1.1 JIT compilation

By default, JAX executes operations eagerly, dispatching each operation individually to XLA without ahead-of-time compilation. The `jax.jit` transform leverages JAX's tracing mechanism to capture entire computations, allowing the XLA compiler to optimize, fuse, and compile sequences of operations into a single efficient execution.

JIT compilation is very powerful but it has some limitations. In particular, it requires that array branching operations must be determined and trace-time. This implies that the following functions will not work with JIT:

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

jax.jit(g)(10, 20)  # Raises an error
```

```
def get_negatives(x):
  return x[x < 0]
```

```python
x = jnp.array(np.random.randn(10))
jax.jit(get_negatives)(x) # Raises an error
```

Another way to see this limitation is that branching statements are allowed in JIT functions as long as they are based on static attributes, e.g. shape, type... What we can do to deal with this is to only compile certain parts of the function:

```python
# 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)
```

If we really need to compile the whole function then we can mark certain arguments as static. This will make the compiled version of the function depend on the static arguments, so JAX will have to re-compile the function for every new static input. This is only a good strategy if the you know that there is only a limited number of static values.

```python
from functools import partial

@partial(jax.jit, static_argnames=['n'])
def g_jit_decorated(x, n):
  i = 0
  while i < n:
    i += 1
  return x + i

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

The first time we call a JIT function it gets compiled and the resulting XLA code is cached. Subsequent calls will then reuse the cached code. If we specify static arguments, the cached code will only be used for the same values of static arguments. To find the cached code JAX uses the hash of the function, this implies that we should not redefine equivalent functions in ways that can modify the function hash.

```python
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
```

1.2 Automatic vectorization

Automatic vectorization allows us to extend a function defined on single input to support batch operations, while capitalizing on hardware acceleration.

The `jax.vmap` transform, like `jax.jit`, works by tracing the function and automatically adding batch axes at the beginning of each input. With the arguments `in_axes` and `out_axes` we can specify the input and output batch dimensions. For more information on how to use these two parameters this article is a really good reference.

All JAX transformations are composable, this means that we can combine `jax.jit` and `jax.vmap` to create just-in-time compiled-vectorized functions. It's important to note that these two transformations commute, it doesn't matter if we compile or vectorize first, it will always create a compiled version of the vectorized function rather than a vectorized version of compiled functions.

1.3 Automatic differentiation

The automatic differentiation transform, `jax.grad`, takes in a function and returns its gradient-function. This means that we can apply this transform multiple times to the same input to differentiate multiple times.

```
import jax
import jax.numpy as jnp
from jax import grad

print(grad(grad(jnp.tanh))(2.0))
```

A very useful function is `jax.value_and_grad()` which efficiently computes both the function value and gradient in one pass.

```
def sigmoid(x):
  return 0.5 * (jnp.tanh(x / 2) + 1)

def predict(W, b, inputs):
  return sigmoid(jnp.dot(inputs, W) + b)

inputs = jnp.array([[0.52, 1.12,  0.77],
                    [0.88, -1.08, 0.15],
                    [0.52, 0.06, -1.30],
                    [0.74, -2.49, 1.39]])
targets = jnp.array([True, True, False, True])

def loss(W, b):
  preds = predict(W, b, inputs)
  label_probs = preds * targets + (1 - preds) * (1 - targets
      )
  return -jnp.sum(jnp.log(label_probs))

key, W_key, b_key = jax.random.split(key, 3)
W = jax.random.normal(W_key, (3,))
b = jax.random.normal(b_key, ())

loss_value, Wb_grad = jax.value_and_grad(loss, (0, 1))(W, b)
print('loss␣value', loss_value)
print('loss␣value', loss(W, b))

> loss value 2.9729187
> loss value 2.9729187
```

For a more advanced and in-depth description of JAX's automatic differentiation, you can check the advanced auto-diff section from the official documentation.

2 Using JAX with PennyLane

Disclaimer: PennyLane only officially supports JAX version 0.4.28.

In PennyLane we can construct a JAX-based QNode by specifying `interface='jax'`. This will make the QNode use and returns JAX arrays, while also being compatible with JAX transformations.

To JIT-compile a QNode we can simply add the `@jax.jit` decorator, or call the transform via jax.jit(),

```
dev = qml.device('default.qubit', wires=2)
@jax.jit
@qml.qnode(dev, interface='jax')
def circuit(phi, theta):
    qml.RX(phi[0], wires=0)
    qml.RZ(phi[1], wires=1)
    qml.CNOT(wires=[0, 1])
    qml.RX(theta, wires=0)
    return qml.expval(qml.PauliZ(0))
```

For differentiation methods other than `backprop`, when the `jax` interface is specified, PennyLane will attempt to determine if the computation was JIT-compiled and change the interface to `jax-jit`. This interaction is broken if you use a version of JAX greater than 0.4.28, so you have to manually specify the `jax-jit` interface for it to work.

To vectorize a QNode we can use the `vmap` transform like we would do on any other function.

```
@qml.qnode(dev, interface="jax")
def circuit(param):
    qml.RX(param, wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(0))

vcircuit = jax.vmap(circuit)

# We runs 3 quantum circuits in parallel.
batch_params = jnp.array([1.02, 0.123, -0.571])
    batched_results = vcircuit(batch_params)

> Batched result: [0.52336595 0.99244503 0.84136092]
```

Calculating the gradients with respect to parameters is as simple with any regular Python function:

```
@qml.qnode(dev, interface='jax')
def circuit(phi, theta):
    qml.RX(phi[0], wires=0)
    qml.RY(phi[1], wires=1)
    qml.CNOT(wires=[0, 1])
    qml.PhaseShift(theta, wires=0)
    return qml.expval(qml.PauliZ(0))

phi = jnp.array([0.5, 0.1])
theta = jnp.array(0.2)
grads = jax.grad(circuit3, argnums=(0, 1))
phi_grad, theta_grad = grads(phi, theta)
```

2.1 Shots and Samples

In JAX, all random number generators must be explicitly seeded. This means that when simulations include randomness, the QNode requires a `jax.random.PRNGKey`. If we want to use JIT, then we must create the QNode inside the JIT decorator's scope.

```
@jax.jit
def sample_circuit(phi, theta, shots, key):
    dev = qml.device('default.qubit', wires=2, seed=key,
        shots=100)

    @qml.qnode(dev, interface='jax', shots=shots)
    def circuit(phi, theta):
        qml.RX(phi[0], wires=0)
        qml.RZ(phi[1], wires=1)
        qml.CNOT(wires=[0, 1])
        qml.RX(theta, wires=0)
        return qml.sample()

    return circuit(phi, theta)
```

3 Conclusions