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 practical 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, and for debugging purposes. 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 the most interested 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 arrays and input parameters of a function by abstract placeholders with the same shape and type. Working at this abstract level provides a way to determine the sequence of operations of a function and their effect on arrays without taking into account their content. Of course, this will come with the limitation of not being able to do branching operations based on the content of those arrays or parameters, a small price to pay.

To see how JAX sees input arrays inside a function subject to a transform we can simply print them:

```
@jax.jit
def f(x, a):
  y = jnp.arange(5)
  z = 5
  print("inside␣the␣function␣we␣see␣x␣as", x)
  print("inside␣the␣function␣we␣see␣y␣as", y)
  print("inside␣the␣function␣we␣see␣z␣as", z)
  print("inside␣the␣function␣we␣see␣a␣as", a)
  return x + 1

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

> 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>

> inside the function we see y as Traced<ShapedArray(int32
    [5])>with<DynamicJaxprTrace>

> inside the function we see z as 5
> inside the function we see a as Traced<ShapedArray(int32
    [], weak_type=True)>with<DynamicJaxprTrace>

> f(x, 5) =  [1 2 3 4 5]
```

In the example above, the variable x, which is passed as an argument, appears inside the function as `Traced<ShapedArray(int32[5])>`. This indicates that JAX has replaced it with a traced representation that preserves its shape (`[5]`) and type (`int32`), but not its actual values. Similarly, y, which is created within the function using `jnp.arange`, is also traced because it is derived from a JAX operation. In contrast, z, which is assigned a static integer value, remains a standard Python integer and is not subject to tracing. The variable a, another function argument, is traced in the same manner as x, but with an additional property: it is weakly typed, meaning that its type can be implicitly upcast, e.g. `int32` to `int64`, depending on how it is used in the computation.

1.1 JIT compilation

By default, JAX executes operations eagerly, tracing and staging each operation individually for XLA execution without ahead-of-time compilation. This introduces a minor overhead for small-scale computations and may even be slower than NumPy.

The `jax.jit` transformation enables the batching of multiple computations for optimized execution via XLA. By leveraging JAX's tracing mechanism, it captures the entire computation, allowing XLA to optimize, fuse, and compile sequences of operations into a single efficient execution.

When using JIT, we exchange a slow first execution for much faster subsequent ones. Therefore, JIT is the most useful when we have expensive numerical functions that we need to run a bunch of times.

Despite being very powerful, JIT compilation still has some limitations. Like we previously mentioned, due the tracing mechanism, it requires that branching operations do not depend on the contents of arrays or traced parameters. This implies that the following functions will not work with JIT.

In this case because the value of n will not be known at trace-time:

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

try:
    jax.jit(g)(10, 20)  # Raises an error
except Exception as e:
    print("[!]␣JAX␣JIT␣error:", e)

> [!] JAX JIT error: Attempted boolean conversion of traced
   array with shape bool[].
```

In this case because the contents of the x array will not be known at trace-time:

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

x = jnp.array(np.random.randn(10))

try:
    jax.jit(get_negatives)(x)  # Raises an error
except Exception as e:
    print("[!]␣JAX␣JIT␣error:", e)

> [!] JAX JIT error: Array boolean indices must be concrete;
    got ShapedArray(bool[10])
```

Another way to see this limitation is that branching statements are only allowed in JIT functions if and only if they are based on static attributes, like shape and type.

A simple solution to deal with functions like the previous ones is to separate the statement that comes after the branching in an external function and compile that rather than the whole function:

```
@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

print(g_inner_jitted(10, 20))

> 30
```

Alternatively, if we really need to compile the whole function we can mark certain arguments as static, making the compiled version of the function depend on the static arguments:

```
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))

> 30
```

This will make it so JAX has to re-compile the function for every new static input. Therefore, it is only a good strategy if we know that there is only a limited number cases.

Subsequent calls of JIT compiled functions use the cached code, which they find via the hash of the function. This implies that we have to be a bit careful when applying the JIT transformation. In some cases, by mistake, we may redefine an equivalent function in a way that modifies its hash and makes it impossible for JIT to use the cached compiled code. Here is an example:

```
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

%timeit g_inner_jitted_partial(10, 20).block_until_ready()
%timeit g_inner_jitted_lambda(10, 20).block_until_ready()
%timeit g_inner_jitted_normal(10, 20).block_until_ready()

> 221 ms +- 9.75 ms per loop
> 220 ms +- 6.08 ms per loop
> 0.565 ms +- 0.256 ms per loop
```

As we can see in the time per loop, the first two are much slower due the fact of not being able to take advantage of the cached code.

1.2 Automatic vectorization

The automatic vectorization, `jax.vmap`, transform allows us to extend a function defined on single inputs to support batch operations, while leveraging hardware acceleration whenever possible.

To use the `jax.vmap` we just have to specify how to vectorize the function with the `in_axes` and `out_axes` parameters. The `in_axes` parameter specifies along which axes we vectorize each input. In this case we specify to vectorize along the rows, 0, for the first two parameters and to not vectorize along the third parameter:

```
def weighted_array_sum(scale: float, weights: jnp.ndarray,
    epsilon: float):
    return scale * jnp.sum(weights) + epsilon

scales = jnp.array([0.5, 1.0, 1.5])
weights = jnp.array([
    [1.0, 0.2, 0.3],
    [0.5, 0.8, 1.0],
    [0.7, 0.1, 0.4]
])
epsilon = 1e-4

vectorized_weighted_array_sum = jax.vmap(weighted_array_sum,
     in_axes=(0, 0, None))

vectorized_weighted_array_sum(scales, weights, epsilon)

> Array([0.7501    , 2.3000998, 1.8001001], dtype=float32)
```

The `out_axes` parameter specifies how to store the batched results. In this case we show the difference of storing the batched results as rows or as columns:

```
def identity(x):
    return x

print("id_rows(x)␣=␣\n", jax.vmap(identity, in_axes=0,
    out_axes=0)(weights), "\n")
print("id_cols(x)␣=␣\n", jax.vmap(identity, in_axes=0,
    out_axes=1)(weights))

> id_rows(x) =
 [[1.  0.2 0.3]
 [0.5 0.8 1. ]
 [0.7 0.1 0.4]]

> id_cols(x) =
 [[1.   0.5  0.7]
 [0.2  0.8  0.1]
 [0.3  1.   0.4]]
```

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`, allows us to get the gradient of any function. We can apply it any number of times to obtain a higher order gradients with respect to any parameters.

```
def func(x, y):
    return x**2 + y**3

grad_x = jax.grad(func, argnums=0)        # Derivative w.r.t
    . x
grad_y = jax.grad(func, argnums=1)        # Derivative w.r.t
    . y
grad   = jax.grad(func, argnums= (0, 1))  # Both

x, y = 2.0, 3.0
print("Gradient␣w.r.t␣x:", grad_x(x, y))  # 2*x = 4
print("Gradient␣w.r.t␣y:", grad_y(x, y))  # 3*y^2 = 27
print("Gradient␣both:", grad(x, y))       # (4, 27)

> Gradient w.r.t x: 4.0
> Gradient w.r.t y: 27.0
> Gradient both: (Array(4., dtype=float32, weak_type=True),
    Array(27., dtype=float32, weak_type=True))
```

Automatic differentiation also works for parameters specified in a dictionary:

```
def loss_fn(params, x):
    w, b = params["w"], params["b"]
    return jnp.sum((w * x + b) ** 2)

params = {"w": 2.0, "b": 1.0}
x = jnp.array([1.0, 2.0, 3.0])

grad_fn = jax.grad(loss_fn)
grads = grad_fn(params, x)
print("Gradients:", grads)  # {'w': ..., 'b': ...}

> Gradients: {'b': Array(30., dtype=float32, weak_type=True)
    , 'w': Array(68., dtype=float32, weak_type=True)}
```

2 Using JAX with PennyLane

Disclaimer: Despite not being explicitly mentioned in the documentation, 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.

```
n_qubits = 16
dev = qml.device("default.qubit", wires=n_qubits)

@qml.qnode(dev, interface="jax-jit") # If jax version is >
    0.4.28, some transformations won't work if the interface
    is not specified as "jax-jit" due using `jax.core.
    ConcreteArray` which is deprecated.
def circuit(param):
    for idx in range(n_qubits):
        qml.Hadamard(wires=idx)
        qml.RX(param, wires=idx) # we use the same parameter
            for simplicity.

    for idx in range(n_qubits-1):
        qml.CNOT(wires=[idx, idx+1])

    qml.CNOT(wires=[n_qubits-1, 0])

    return qml.expval(qml.PauliZ(0))
```

Once we have created a circuit with the JAX interface, we can use any transform. From the point of view of JAX, the circuit is just a regular function, we can even print the corresponding jaxpr.

We can create compiled and vectorized versions of our circuit like any other function:

```
jit_circuit  = jax.jit(circuit)
vcircuit     = jax.vmap(circuit)
jit_vcircuit = jax.jit(vcircuit)
```

In the figure 1 we show how the runtime scales as the batch size increases for each version of the circuit. And in the figure 2 we focus on the smaller batch sizes.

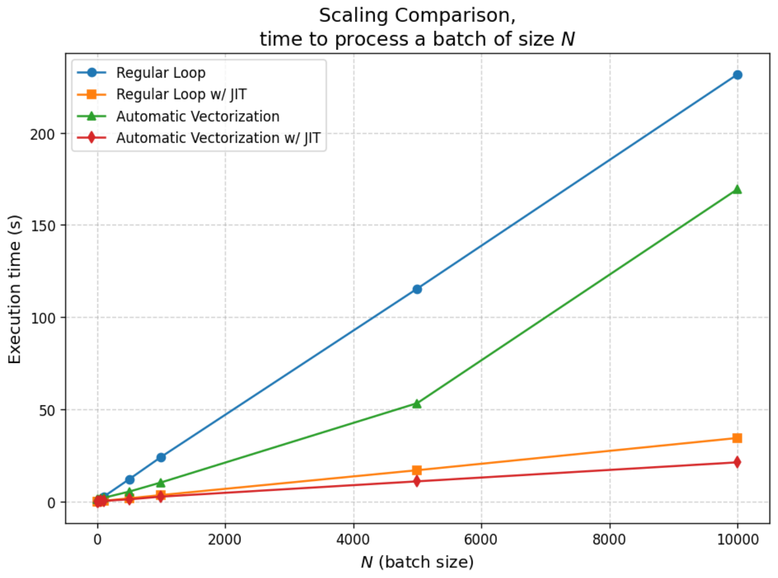

Figure 1: Runtime scaling of a circuit given different transformation.

The experiment results indicate that the combination of just-in-time compilation and automatic vectorization achieves the best execution times, significantly outperforming either technique alone. JIT compilation alone delivers the second-best performance, even surpassing the JIT + vectorization approach at small batch sizes. Automatic vectorization alone ranks third, providing speedup benefits but struggling to match JIT's performance, particularly for larger batch sizes.

Despite its lower standalone performance, automatic vectorization remains valuable due to its practicality and its ability to enhance JIT when combined.

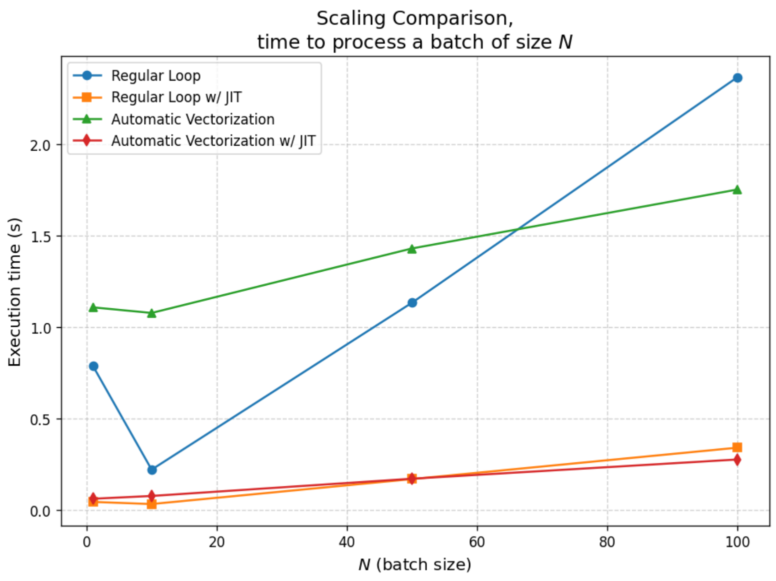

Figure 2: Focus on small batch sizes of the runtime scaling.

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 circuit(key, param):
    dev = qml.device("default.qubit", wires=2, shots=10,
        seed=key)
    @qml.qnode(dev, interface="jax", diff_method=None)
    def sampling_circuit():
        qml.RX(param, wires=0)
        qml.CNOT(wires=[0, 1])
        return qml.sample(qml.PauliZ(0))

    return sampling_circuit()

key1 = jax.random.PRNGKey(0)
key2 = jax.random.PRNGKey(1)

result1 = circuit(key1, jnp.pi/2)
result2 = circuit(key1, jnp.pi/2)
result3 = circuit(key2, jnp.pi/2)
```

```
print("Result␣with␣key1␣(first␣call):", result1)
print("Result␣with␣key1␣(second␣call):", result2)
print("Result␣with␣key2:", result3)

> Result with key1 (first call): [ 1.  1.  1. -1.  1. -1.
   -1.  1. -1.  1.]
> Result with key1 (second call): [ 1.  1.  1. -1.  1. -1.
   -1.  1. -1.  1.]
> Result with key2: [ 1.  1. -1.  1. -1. -1.  1.  1.  1.
   -1.]
```

3 Conclusions

In conclusion, JAX offers a powerful framework for numerical computing that fits seamlessly into the realm of quantum machine learning. Its array-based approach, combined with composable transformations such as just-in-time compilation, automatic vectorization, and differentiation, enables researchers to write clean, high-performance code that is both scalable and efficient. This design makes JAX an ideal choice for implementing quantum circuits, particularly when integrated with PennyLane, where the benefits of accelerated execution and precise gradient computations directly contribute to more effective QML applications.