PennyLane: Automatic differentiation of hybrid quantum-classical computations

Ville Bergholm,[1] Josh Izaac,[1] Maria Schuld,[1] Christian Gogolin,[1] M. Sohaib Alam,[2] Shahnawaz Ahmed,[3] Juan Miguel Arrazola,[1] Carsten Blank,[4] Alain Delgado,[1] Soran Jahangiri,[1] Keri McKiernan,[2] Johannes Jakob Meyer,[5] Zeyue Niu,[1] Antal Száva,[1] and Nathan Killoran[1]

[1]*Xanadu, 777 Bay Street, Toronto, Canada*

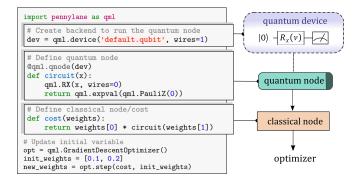[2]*Rigetti Computing, 2919 Seventh Street, Berkeley, CA 94710*

[3]*Wallenberg Centre for Quantum Technology, Department of Microtechnology and Nanoscience, Chalmers University of Technology, 412 96 Gothenburg, Sweden*

[4]*data cybernetics, Martin-Kolmsperger-Str 26, 86899 Landsberg, Germany*

[5]*Dahlem Center for Complex Quantum Systems, Freie Universität Berlin, 14195 Berlin, Germany*

PennyLane is a Python 3 software framework for optimization and machine learning of quantum and hybrid quantum-classical computations. The library provides a unified architecture for near-term quantum computing devices, supporting both qubit and continuous-variable paradigms. PennyLane's core feature is the ability to compute gradients of variational quantum circuits in a way that is compatible with classical techniques such as backpropagation. PennyLane thus extends the automatic differentiation algorithms common in optimization and machine learning to include quantum and hybrid computations. A plugin system makes the framework compatible with any gate-based quantum simulator or hardware. We provide plugins for Strawberry Fields, Rigetti Forest, Qiskit, Cirq, and ProjectQ, allowing PennyLane optimizations to be run on publicly accessible quantum devices provided by Rigetti and IBM Q. On the classical front, PennyLane interfaces with accelerated machine learning libraries such as TensorFlow, PyTorch, and autograd. PennyLane can be used for the optimization of variational quantum eigensolvers, quantum approximate optimization, quantum machine learning models, and many other applications.

Introduction

Recent progress in the development and commercialization of quantum technologies has had a profound impact on the landscape of quantum algorithms. Near-term quantum devices require routines that are of shallow depth and robust against errors. The design paradigm of *hybrid algorithms* which integrate quantum and classical processing has therefore become increasingly important. Possibly the most well-known class of hybrid algorithms is that of *variational circuits*, which are parameter-dependent quantum circuits that can be optimized by a classical computer with regards to a given objective.

Hybrid optimization with variational circuits opens up a number of new research avenues for near-term quantum computing with applications in quantum chemistry [1], quantum optimization [2], factoring [3], state diagonalization [4], and quantum machine learning [5–18]. In a reversal from the usual practices in quantum computing research, a lot of research for these mostly heuristic algorithms necessarily focuses on numerical experiments rather than rigorous mathematical analysis. Luckily, there are various publicly accessible platforms to simulate quantum algorithms [19–26] or even run them on real quantum devices through a cloud service [27, 28]. However, even though some frameworks are designed with variational circuits in mind [25, 29, 30], there is at this stage no unified tool for the hybrid optimization of quantum circuits across quantum platforms, treating all simulators and devices on the same footing.

PennyLane is an open-source Python 3 framework that facilitates the optimization of quantum and hybrid quantum-classical algorithms. It extends several seminal ma-

chine learning libraries — including *autograd* [31], *TensorFlow* [32], and *PyTorch* [33] — to handle modules of quantum information processing. This can be used to optimize variational quantum circuits in applications such as *quantum approximate optimization* [2] or *variational quantum eigensolvers* [1]. The framework can also handle more complex machine learing tasks such as training a hybrid quantum-classical machine learning model in a supervised fashion, or training a generative adverserial network, both when discriminator and generator are quantum models [14] and when one is quantum and the other is classical [34].

```python
import pennylane as qml

# Create backend to run the quantum node
dev = qml.device('default.qubit', wires=1)

# Define quantum node
@qml.qnode(dev)
def circuit(x):
    qml.RX(x, wires=0)
    return qml.expval(qml.PauliZ(0))

# Define classical node/cost
def cost(weights):
    return weights[0] * circuit(weights[1])

# Update initial variable
opt = qml.GradientDescentOptimizer()
init_weights = [0.1, 0.2]
new_weights = opt.step(cost, init_weights)
```

FIG. 1: *Basic example of a PennyLane program consisting of a quantum node followed by a classical node. The output of the classical node is the objective for optimization.*

PennyLane can in principle be used with any gate-based quantum computing platform as a backend, including both qubit and continuous-variable architectures, and has a simple Python-based user interface. Fig. 1 shows a simple example that illustrates the core idea of the framework. The user defines a quantum circuit in the function `circuit` connected to a device `dev`, as well as a "classical function" that calls `circuit` and computes a cost. The functions can be depicted as nodes in a directed acyclic *computational graph* that represents the flow of information in the computation. Each node may involve a number of input and output variables represented by the incoming and outgoing edges, respectively. A `GradientDescentOptimizer` is created that improves the initial candidate for these variables by one step, with the goal of decreasing the cost. PennyLane is able to automatically determine the gradients of all nodes — even if the computation is performed on quantum hardware — and can therefore compute the gradient of the final cost node with respect to any input variable.

PennyLane is an open-source software project. Anyone who contributes significantly to the library (new features, new plugins, etc.) will be acknowledged as a co-author of this whitepaper. The source code for PennyLane is available online on GitHub[1], while comprehensive documentation and tutorials are available on PennyLane.ai[2].

In the following, we will introduce the concept of hybrid optimization and discuss how gradients of quantum nodes are computed. We then present PennyLane's user interface through examples of optimization and supervised learning, and describe how to write new plugins that connect PennyLane to other quantum hardware and simulators.

Hybrid optimization

The goal of optimization in PennyLane is to find the minima of a cost function that quantifies the quality of a solution for a certain task. In hybrid quantum-classical optimization, the output of the cost function is a result of both classical and quantum processing, or a *hybrid computation*. We call the processing submodules *classical* and *quantum nodes*. Both classical and quantum nodes can depend on tunable parameters θ that we call *variables*, which are adjusted during optimization to minimize the cost. The nodes can receive inputs x from other nodes or directly from the global input to the hybrid computation, and they produce outputs $f(x; \theta)$. The computation can therefore be depicted as a Directed Acyclic Graph (DAG) that graphically represents the steps involved in computing the cost, which is produced by the final node in the DAG. By traversing the DAG, information about gradients can be accumulated via the rules of automatic differentiation [35, 36]. This is used to compute the gradient of the cost function with respect to all variables in order to minimize the cost with a gradient-descent-type algorithm. It is important to note that automatic differentiation only requires a small constant overhead compared to the "forward" computation by collecting and reusing intermediate results. However, quantum nodes are black boxes to automatic differentiation, which means that accumulation of partial results does not extend to the interior of quantum nodes.

Quantum nodes

While classical nodes (see Fig. 2(a)) can contain any numerical computations[3], quantum nodes have a more restricted layout. A quantum node (in PennyLane represented by the QNode class) is an encapsulation of a function $f(x; \theta) : \mathbb{R}^m \to \mathbb{R}^n$ that is executed by means of quantum information processing on a *quantum device*. The device can either refer to quantum hardware or a classical simulator.

Variational circuits

The quantum device executes a parametrized quantum circuit called a *variational circuit* [37] that consists of three basic operations:

---

[1] https://github.com/XanaduAI/pennylane/

[2] https://pennylane.ai

[3] Of course, in order to differentiate the classical nodes the computations have to be based on differentiable functions.

FIG. 2: *While a classical node consists of a numerical computation $\mathcal{A}$, a quantum node executes a variational circuit U on a quantum device and returns an estimate of the expectation value of an observable $\hat{B}$, estimated by averaging R measurements.*

1. Prepare an initial state (here assumed to be the ground or vacuum state $|0\rangle$).

2. Apply a sequence of unitary gates U (or more generally, quantum channels) to $|0\rangle$. Each gate is either a fixed operation, or it can depend on some of the inputs x or the variables θ. This prepares the final state $U(x, \theta)|0\rangle$.

3. Measure m mutually commuting scalar observables $\hat{B}_i$ in the final state.

Step 2 describes the way inputs x are encoded into the variational circuit, namely by associating them with gate parameters that are not used as trainable variables[4]. Step 3 describes the way quantum information is transformed back to the classical output of a quantum node as the expected values of the measured observables:

$$f_i(x; \theta) = \langle \hat{B}_i \rangle = \langle 0| U(x; \theta)^\dagger \hat{B}_i U(x; \theta) |0\rangle. \quad (1)$$

The observables $\hat{B}_i$ typically consist of a local observable for each wire (i.e., qubit or qumode) in the circuit, or just a subset of the wires. For example, $\hat{B}_i$ could be the Pauli-Z operator for one or more qubits.

Estimating the expectation values

The expectation values $\langle \hat{B}_i \rangle$ are estimated by averaging the measurement results obtained over R runs of the circuit. This estimator, denoted f_i^*, is unbiased, $\langle f_i^* \rangle = f_i(x; \theta)$, and it has variance

$$\mathrm{Var}(f_i^*) = \frac{\mathrm{Var}(\hat{B}_i)}{R} = \frac{\langle \hat{B}_i^2 \rangle - \langle \hat{B}_i \rangle^2}{R}, \quad (2)$$

which depends on the variance of the operator $\hat{B}_i$, as well as the number of measurements ('shots') R. Note that setting $R = 1$ estimates the expectation value from a single measurement sample. Simulator devices can also choose to compute the exact expectation value numerically (in PennyLane this is the default behavior, represented by setting
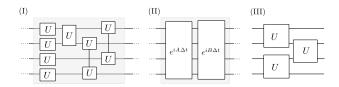
---

[4] This *input embedding* can also be interpreted as a feature map that maps x to the Hilbert space of the quantum system [9].

FIG. 3: *Different types of architectures for variational circuits: (I) layered gate architecture, (II) alternating operator architecture [2], and (III) an example of a tensor network architecture [39].*

R=0). The refined graphical representation of quantum nodes is shown in Fig. 2(b). We will drop the index R in the following.

Circuit architectures

The heart of a variational circuit is the *architecture*, or the fixed gate sequence that is the skeleton of the algorithm. Three common types of architectures are sketched in Fig. 3. The strength of an architecture varies depending on the desired use-case, and it is not always clear what makes a good ansatz. Investigations of the expressive power of different approaches are also ongoing [38]. One goal of PennyLane is to facilitate such studies across various architectures and hardware platforms.
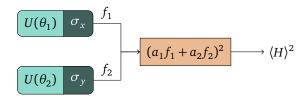
Examples of hybrid optimization tasks

Fig. 4 shows three examples of hybrid optimization tasks depicted as a DAG. Each of these models is available as a worked example in the PennyLane documentation [40]. Fig. 4(a) illustrates a variational quantum eigensolver, in which expectation values of two Pauli operators are combined with weights a_1, a_2 to return the squared global energy expectation $\langle H \rangle^2$. Fig. 4(b) shows a variational quantum classifier predicting a label y given a data input x for a supervised learning task. The input is preprocessed by a routine $\mathcal{P}$ and fed into a variational circuit with variables θ_W. A classical node adds a bias variable θ_b to the Pauli-Z expectation of a designated qubit. In Fig. 4(c) one can see a quantum generative adversarial network (QGAN) example. It consists of two variational circuits. One represents the "real data" circuit R together with a discriminator circuit D, and the other has a "fake" generator circuit G replacing R. The result is postprocessed by $\mathcal{P}_{R,G}$ and used to construct the cost function of the discriminator as well as the generator. The goal of a GAN is to train the discriminator and generator in an adversarial fashion until the generator produces data that is indistinguishable from the true distribution.

Computing gradients

PennyLane focuses on optimization via gradient-based algorithms, such as gradient descent and its variations. To minimize the cost via gradient descent, in every step the

(a) Variational quantum eigensolver

(b) Variational quantum classifier
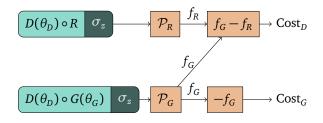

(c) Quantum generative adversarial network (QGAN)

FIG. 4: *DAGs of hybrid optimization examples. These models and more are available as worked examples in the PennyLane docs [40].*

Path p=1

Path p=2

FIG. 5: *Example illustration of the two paths that lead from the cost function back to a quantum node.*

individual variables $\mu \in \Theta$ are updated according to the following rule:

1: **procedure** GRADIENT DESCENT STEP
2:     **for** $\mu \in \Theta$ **do**
3:         $\mu^{(t+1)} = \mu^{(t)} - \eta^{(t)} \partial_\mu C(\Theta)$

The learning rate $\eta^{(t)}$ can be adapted in each step, depending either on the step number, or on the gradient itself.

Backpropagating through the graph

A step of gradient descent requires us to compute the gradient $\nabla_\Theta C(\Theta)$ of the cost with respect to all variables Θ. The gradient consists of partial derivatives $\partial_\mu C(\Theta)$ with respect to the individual variables $\mu \in \Theta$. In modern machine learning libraries like *TensorFlow* [32], *PyTorch* [33], or *autograd* [31], this computation is performed using *automatic differentiation* techniques such as the backpropagation algorithm. PennyLane extends these capabilities to computations involving quantum nodes, allowing computational models in these three machine learning libraries (including those with GPU-accelerated components) to seamlessly include quantum nodes. This makes PennyLane completely compatible with standard automatic differentiation techniques commonly used in machine learning.
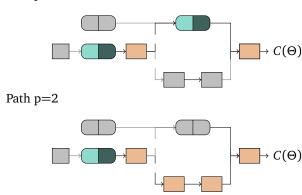
While the backpropagation method — a classical algo-

rithm — cannot resolve the quantum information inside quantum nodes, it is sufficient for us to compute the gradient or Jacobian of quantum nodes with respect to their (classical) inputs and variables. The key insight is to use the *same quantum device* (hardware or simulator) that implements a quantum node to also compute gradients or Jacobians of that quantum node.

Assume that only the node n^* depends on the subset of variables $\theta \subseteq \Theta$, and that μ is in θ. Let $C \circ n_1^{(p)} \circ \cdots \circ n^*$ be the path through the DAG of (quantum or classical) nodes that emerges from following the cost in the opposite direction of the directed edges until we reach node n^*. Since there may be $N_p \geq 1$ of those paths (see Fig. 5), we use a superscript p to denote the path index. All branches that do not lead back to θ are independent of μ and can be thought of as constants. The chain rule prescribes that the derivative with respect to the variable $\mu \in \theta$ is given by[5]

$$\partial_\mu C(\Theta) = \sum_{p=1}^{N_p} \frac{\partial C}{\partial n_1^{(p)}} \frac{\partial n_1^{(p)}}{\partial n_2^{(p)}} \cdots \frac{\partial n^*}{\partial \mu}.$$

In conclusion, we need to be able to compute two types of gradients for each node: the derivative $\frac{\partial n_i^{(p)}}{\partial n_{i-1}^{(p)}}$ with respect to the input from a previous node, as well as the derivative with respect to a node variable $\frac{\partial n}{\partial \mu}$.

Derivatives of quantum nodes

There are three types of methods to compute derivatives[6] of quantum nodes with respect to a variable or input: an-

---

[5] While $\partial_\mu C(\Theta)$ is a partial derivative and one entry of the gradient vector $\nabla C(\Theta)$, intermediate DAG nodes may map multiple inputs to multiple outputs. In this case, we deal with 2-dimensional Jacobian matrices rather than gradients.

[6] When we speak of derivatives here, we actually refer to estimates of derivatives that result from estimates of expectation values. Numerically computed derivatives in turn are approximations of the true derivatives,

alytical, numerical, or device-provided. By default, PennyLane uses the device or analytical derivatives wherever it can. Most types of quantum nodes support analytic derivatives, even if they are executed on quantum hardware.

Analytic derivatives

Recent advances in the quantum machine learning literature [8, 10, 11, 41] have suggested ways to estimate analytic derivatives by computing linear combinations of different quantum circuits. These rules are summarized and extended in a companion paper [42], which provides the theoretical foundation for derivative computations in PennyLane. In a nutshell, PennyLane makes two circuit evaluations, taking place at shifted parameters, in order to compute analytic derivatives. This recipe works for qubit gates of the form $e^{-i\mu P}$, where the Hermitian generator P has only two unique eigenvalues (which includes e.g., all single-qubit rotation gates), as well as continuous-variable circuits with Gaussian operations[7].

If $f(x;\theta) = f(\mu)$ is the output of the quantum node, we have

$$\partial_\mu f(\mu) = c\big(f(\mu + s) - f(\mu - s)\big), \qquad (3)$$

where $c, s \in \mathbb{R}$ are fixed parameters for each type of gate. While this equation bears some structural resemblance to numerical formulas (discussed next), there are two key differences. First, the numbers c and s are not infinitesimal, but finite; second, Eq. (3) gives the *exact* derivatives. Thus, while analytic derivative evaluations are constrained by device noise and statistical imprecision in the averaging of measurements, they are not subject to numerical issues. To analytically compute derivatives of qubit gates or gates in a Gaussian circuit, PennyLane automatically looks up the appropriate derivative recipe (the numbers c and s) for a gate, evaluates the original circuit twice (shifting the argument of the relevant gate by $\pm s$), subtracts the results, and scales by c.

Numerical derivatives

Numerical derivative methods require only 'black-box' evaluations of the model. We estimate the partial derivative of a node by evaluating its output, $f(x;\theta) = f(\mu)$, at several values which are close to the current value $\mu \in \theta$ (μ can be either a variable or an input here). The approximation of the derivative is given by

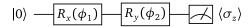$$\partial_\mu f(\mu) \approx \frac{f(\mu + \Delta\mu) - f(\mu)}{\Delta\mu} \qquad (4)$$

FIG. 6: *Variational circuit of the qubit rotation example.*

for the *forward finite-differences* method, and by

$$\partial_\mu f(\mu) \approx \frac{f(\mu + \frac{1}{2}\Delta\mu) - f(\mu - \frac{1}{2}\Delta\mu)}{\Delta\mu} \qquad (5)$$

for the *centered finite-differences* method. Of course, there is a tradeoff in choice of the difference $\Delta\mu$ for noisy hardware.

Device derivatives

In addition to the analytic and numeric derivative implementations described above — which are supported by all simulator and hardware devices — PennyLane also supports directly querying the device for the derivative, if known. For example, a simulator written using a classical automatic differentiation library, such as TensorFlow or PyTorch, can make use of backpropagation algorithms internally to calculate derivatives. Compared to the analytic method on simulators, this may lead to significant time savings, as the information required to compute the derivative is stored and reused from the forward circuit evaluation — simply adding constant overhead. Furthermore, the device derivative may also be used when interfacing with hardware devices that provide their own custom gradient formulations.

User API

A thorough introduction and review of PennyLane's API can be found in the online documentation. The documentation also provides several examples for optimization and machine learning of quantum and hybrid models in both continuous-variable and qubit architectures, as well as tutorials that walk through the features step-by-step.

Optimization

To see how PennyLane allows the easy construction and optimization of variational circuits, let us consider the simple task of optimizing the rotation of a single qubit — the PennyLane version of 'Hello world!'.

The task at hand is to optimize the variational circuit of Fig. 6 with two rotation gates in order to flip a single qubit from state $|0\rangle$ to state $|1\rangle$. After the rotations, the qubit is in state $|\psi\rangle = R_y(\phi_2)R_x(\phi_1)|0\rangle$ and we measure the expectation value

$$f(\phi_1, \phi_2) = \langle\psi|\sigma_z|\psi\rangle = \cos(\phi_1)\cos(\phi_2)$$

of the Pauli-Z operator. Depending on the variables ϕ_1 and ϕ_2, the output expectation lies between 1 (if $|\psi\rangle = |0\rangle$)

---

even if the quantum nodes were giving exact expectations (e.g., by using a classical simulator device).

[7] For cases that do not fall into the above two categories, using an ancilla may provide an alternate strategy to evaluate derivatives [42]. This option is not currently implemented in PennyLane.

and -1 (if $|\psi\rangle = |1\rangle$).

PennyLane code for this example — using the default *autograd* interface for classical processing — is shown below in Codeblock 1. It is a self-contained example that defines a quantum node, binds it to a computational device, and optimizes the output of the quantum node to reach a desired target.

```python
import pennylane as qml
from pennylane import expval
from pennylane.optimize import
↪   GradientDescentOptimizer

# Create device
dev = qml.device('default.qubit', wires=1)

# Quantum node
@qml.qnode(dev)
def circuit1(var):
    qml.RX(var[0], wires=0)
    qml.RY(var[1], wires=0)
    return expval(qml.PauliZ(0))

# Create optimizer
opt = GradientDescentOptimizer(0.25)

# Optimize circuit output
var = [0.1, 0.2]
for it in range(30):
    var = opt.step(circuit1, var)
    print("Step {}: cost: {}".format(it,
    ↪   circuit1(var)))
```

Codeblock 1: *Optimizing two rotation angles to flip a qubit.*

We now discuss each element in the above example. After the initial import statements, line 5 declares the device `dev` on which we run the quantum node, while lines 7–12 define the quantum node itself. PennyLane uses the name `wires` to refer to quantum subsystems (qubits or qumodes) since they are represented by horizontal wires in a circuit diagram. The decorator `@qml.qnode(dev)` is a shortcut that transforms the function `circuit1` into a quantum node of the same name. If PennyLane is used with another supported machine learning library, such as PyTorch or TensorFlow, the QNode interface should be specified when using the decorator, via the `interface` keyword argument (`interface='torch'` and `interface='tf'` respectively). This allows the QNode to accept objects native to that interface, such as Torch or TensorFlow tensors.

Note that we could alternatively create the QNode by hand, without the use of the decorator:

```python
def circuit1():
    ...

circuit1 = qml.QNode(dev, circuit1)
```

Codeblock 2: *Creating a quantum node without the decorator.*

Finally, the free variables of this computation are automatically optimized through repeated calls to the `step` method of the provided optimizer.

In order for a quantum node to work properly within PennyLane, the function declaring the quantum circuit must adhere to a few rules. It can only contain quantum gates (no classical processing within the circuit), and must return expectation values of one or more observables on separate wires. In the latter case, the expectation values should be returned together as a tuple.

```python
dev2 = qml.device('default.qubit', wires=2)

@qml.qnode(dev2)
def circuit2(var):
    qml.RX(var[0], wires=0)
    qml.CNOT(wires=[0,1])
    qml.RY(var[1], wires=1)
    return expval(PauliZ(0)),
    ↪   expval(PauliZ(1))
```

Codeblock 3: *A quantum node that returns two expectations.*

As long as at least one expectation value is returned, not every wire needs to be measured. In addition to expectation values, PennyLane also supports returning variances (`qml.var()`) and samples (`qml.sample()`), although the latter is not differentiable. Tensor observables may also be specified using the `@` notation, for example `qml.expval(qml.PauliZ(0) @ qml.PauliY(2))`. Note that since PennyLane treats hardware and simulators on the same footing, the user does not have access to the quantum state itself.

Multiple quantum nodes can be bound to the same device, and the same circuit can be run on different devices. In the latter case, the QNode will need to be created manually. These use-cases are shown in Codeblock 4.[8]

```python
# Create devices
sim = qml.device('qiskit.aer', wires=1)
hardware = qml.device('qiskit.ibm',
↪   backend="ibmqx5" wires=1)

# Define quantum circuits
def circuitA(var):
    qml.RX(var[0], wires=0)
    qml.RY(var[1], wires=0)
    return expval(qml.PauliZ(0))

def circuitB(var):
    qml.RY(var[0], wires=0)
    qml.RX(var[1], wires=0)
    return expval(qml.PauliZ(0))
```

---

[8] This particular example leverages the Qiskit [43] plugin for PennyLane [44]. This code will not run without the plugin being installed and without hardware access credentials being provided.

```
# QNode running Circuit A on hardware
A_s = qml.QNode(circuitA, sim)

# QNode running Circuit B on hardware
A_hw = qml.QNode(circuitA, hardware)

# QNode running Circuit B on simulator
B_hw = qml.QNode(circuitB, hardware)
```

Codeblock 4: *Constructing multiple quantum nodes from various circuits and devices.*

If we have multiple quantum nodes, we can combine the outputs with a classical node to compute a final cost function:

```
# Classical node
def cost(var):
    return (A_s(var[0])-A_hw(var[1]))**2

opt = GradientDescentOptimizer()

var = [0.1, 0.2]
for it in range(10):
    var = opt.step(cost, var)
```

Codeblock 5: *A classical node combining two quantum nodes.*

This cost compares a simulator and a hardware, and finds values of the variables for which the two produce the same result. This simple example hints that automatic optimization tools could be used to correct for systematic errors on quantum hardware.

In summary, quantum and classical nodes can be combined in many different ways to build a larger hybrid computation, which can then be optimized automatically in PennyLane.

Supervised learning

PennyLane has been designed with quantum and hybrid quantum-classical machine learning applications in mind. To demonstrate how this works, we consider a basic implementation of a variational classifier. A variational classifier is a model where part of the computation of a class prediction is executed by a variational circuit. The circuit takes an input x as well as some trainable variables and computes a prediction y.

```
def loss(labels, predictions):
    # Compute loss
    ...

def regularizer(var):
    # Compute regularization penalty
    ...

def statepreparation(x):
```

```
    # Encode x into the quantum state
    ...

def layer(W):
    # Layer of the model
    ...

def circuit3(x, weights):
    # Encode input x into quantum state
    statepreparation(x)
    # Execute layers
    for W in weights:
        layer(W)
    return ... # Return expectation(s)

def model(x, var):
    weights = var[0]
    bias = var[1]
    return circuit3(x, weights) + bias

def cost(var, X, Y):
    # Compute prediction for each input
    preds = [model(x, var) for x in X]
    # Compute the cost
    loss = loss(Y, preds)
    regul = regularizer(var)
    return loss + 0.01 * regul
```

Codeblock 6: *Code stub for creating a variational quantum classifier.*

In Codeblock 6, the machine learning model is defined in the `model` function. It retrieves two types of variables from `var`, a scalar bias and a list of layer weights. It then computes the output of the variational circuit and adds the bias. The variational circuit, in turn, first refers to a routine that encodes the input into a quantum state, and then applies layers of a certain gate sequence, after which an expectation is returned.

We can train the classifier to generalize the input-output relation of a training dataset.

```
# Training inputs
X = ...
# Training targets
Y = ...

# Create optimizer
opt = AdamOptimizer(0.005, beta1=0.9,
↪   beta2=0.9)

# Initialize variables
n_layers = ...
n_gates = ...
var = (np.random.randn(n_layers, n_gates), 0.)

# Train the model
for it in range(50):
    var = opt.step(lambda v: cost(v, X, Y),
    ↪   var)
```

Codeblock 7: *Code stub for optimizing the variational classifier.*

The variables are initialized as a tuple containing the bias and the weight matrix. In the optimization loop, we feed a Python lambda function into the optimizer. Since the optimizer expects a function with a single input argument, this is a way to feed both X and Y into the cost.

PennyLane can straightforwardly incorporate various standard machine learning practices. Examples include: optimizing minibatches of data with stochastic gradient descent, adding more terms to the cost, saving variables to a file, and continuing optimization with a warm start. For full worked-out examples, see the PennyLane documentation [40].

Behind the scenes

The core of PennyLane is the `grad` method for functions with scalar outputs, as well as the `jacobian` method for multi-dimensional functions. `grad` and `jacobian` compute gradients of classical or quantum nodes. Let us switch to "interactive mode" and look at `circuit1` and `circuit2` from above.

```
>>> from pennylane import numpy as np
>>> var = np.array([0.4, 0.1])
>>> g1 = qml.grad(circuit1, argnum=0)
>>> print(g1(var))
[-0.38747287, -0.09195267]

>>> j2 = qml.jacobian(circuit2, argnum=0)
>>> print(j2(var))
[[-0.38941834  0.         ]
 [-0.38747287 -0.09195267]]
```

Codeblock 8: *Computing gradients of hybrid functions.*

As expected, the gradient of a `QNode` with 2 inputs and 1 output is a 1-dimensional array, while the Jacobian of a `QNode` with 2 inputs and 2 outputs is a 2×2 array. The `Optimizer` class uses gradients and Jacobians computed this way to update variables. PennyLane currently has seven built-in optimizers, which work with the default Autograd interface: standard gradient descent, gradient descent with momentum, gradient descent with Nesterov momentum, Adagrad, Adam, RMSprop, and quantum natural gradient descent [45]. For the PyTorch and TensorFlow, the optimizers provided by those libraries can be used.

While automatic differentiation with gradients and Jacobians is a handy feature, sometimes we want certain parts of our computational pipeline (e.g., the inputs x in the supervised learning example) not to be part of a gradient. In PennyLane *all positional arguments to quantum nodes can be differentiated*, while *keyword arguments are never differentiated*. Thus, when using the gradient-descent-based optimizers included in PennyLane, all numerical parameters appearing in non-keyword arguments will be updated, while numerical values included as keyword arguments will not be updated. Once defined, keyword arguments *must always*

be passed as keyword arguments, and not as positional arguments.

Algorithms and features

PennyLane also provides a higher-level interface for easily and automatically creating and processing QNodes. This includes a library of circuit ansätze or 'templates' from across the quantum machine learning literature, tools to map a single ansatz across multiple observables or devices, and the ability to easily create cost functions for common quantum variational algorithms.

Templates

The `pennylane.templates` module provides a growing library of pre-coded templates of common variational circuit architectures that can be used to build, evaluate, and train more complex models. In the literature, such architectures are commonly known as an ansatz. PennyLane conceptually distinguishes two types of templates, layer architectures and input embeddings. Most templates are complemented by functions that provide an array of random initial parameters.

```
import pennylane as qml
from pennylane.templates import
↪   AngleEmbedding, StronglyEntanglingLayers
from pennylane.init import
↪   strong_ent_layers_uniform

dev = qml.device('default.qubit', wires=2)

@qml.qnode(dev)
def circuit(weights, x=None):
    AngleEmbedding(x, [0,1])
    StronglyEntanglingLayers(weights, [0,1])
    return qml.expval(qml.PauliZ(0))

init_weights =
↪   strong_ent_layers_uniform(n_layers=3,
↪   n_wires=2)
print(circuit(init_weights, x=[1., 2.]))
```

Codeblock 9: *The embedding template* `AngleEmbedding` *is used to embed data within the QNode, and the layer template StronglyEntanglingLayers used as the variational ansatz with a uniform parameter initialization strategy.*

Templates provided include `AmplitudeEmbedding`, `QAOAEmbedding`, `CVNeuralNetLayers`, among others. In addition, custom templates can be created; simply decorate a Python function that applies quantum gates with the `template` decorator:

```
@qml.template
def bell_state_preparation(wires):
    qml.Hadamard(wires=wires[0])
    qml.CNOT(wires=wires)
```

Codeblock 10: *Defining a custom template.*

The custom template can then be used within any valid QNode.

QNode collections

A number of variational algorithms, such as the variational quantum eigensolver, require numerous quantum circuit evaluations per optimization time-step. In PennyLane, this corresponds to constructing and evaluating multiple QNodes. PennyLane provides a high-level framework for processing and manipulating groups of (possibly independent) QNodes, known as a `QNodeCollection`. QNode collections are sequences of QNodes, each bound to potentially different devices, that can be evaluated independently — i.e., the input of any QNode in the collection does not depend on the output of another.

```python
@qml.qnode(dev1)
def circuit1(x):
    qml.RX(x[0], wires=0)
    qml.RY(x[1], wires=0)
    return qml.expval(qml.PauliZ(0))

@qml.qnode(dev2)
def circuit2(x):
    qml.RZ(x[0], wires=0)
    qml.RZ(x[1], wires=0)
    return qml.var(qml.PauliY(0))

qnodes = qml.QNodeCollection([circuit1,
    circuit2])
qnodes([0.3, 0.2])
```

Codeblock 11: *Creating a QNode collection.*

QNode collections can also be created by mapping a template over a list of observables and devices; each QNode within the collection will be evaluated on the corresponding device.

```python
def ansatz(x, **kwargs):
    qml.Hadamard(wires=0)
    qml.RX(x[0], wires=0)
    qml.CNOT(wires=[0, 1])

obs = [
    qml.PauliX(0),
    qml.PauliY(0) @ qml.PauliZ(1)
]
devs = [
    qml.device("forest.qpu",
        device="Aspen-4-4Q-B"),
    qml.device("qiskit.ibmq",
        backend="ibmqx2")
]
qnodes = qml.map(ansatz, obs, devs)
```

Codeblock 12: *Creating a QNode collection with hardware devices.*

The key advantage of QNode collections is that, since the QNodes are independent, they can be evaluated simultaneously and are embarrassingly parallelizable. If the user has access to multiple hardware devices, the QNode collection also supports asynchronous quantum evaluation by passing the `parallel=True` keyword argument, significantly decreasing the wall time per optimization step compared to sequential evaluation.

Within the QNode collection abstraction, the user has access to several composition functions that act on and process QNode collections when evaluated. These include `qml.sum()`, `qml.dot()`, and `qml.apply()`.

```python
>>> coeffs = torch.tensor([-1.3, 0.2],
    dtype=torch.float64)
>>> cost = qml.dot(coeffs,
    qml.apply(torch.sin, qnodes))
>>> cost([0.2, 0.1], parallel=True)
tensor(-0.9078, dtype=torch.float64)
```

Codeblock 13: *A QNode collection using the PyTorch interface has the* `sin` *function applied using* `qml.apply`. *Then, the dot product of the QNode collection is taken with a PyTorch tensor. Note that the function composition is lazy — the quantum evaluation only occurs once* `cost` *is called.*

Quantum Chemistry

The variational quantum eigensolver (VQE) algorithm is frequently applied to quantum chemistry problems [1]. In VQE, a quantum computer is first used to prepare the trial wave function of a molecule, and the expectation value of its electronic Hamiltonian is measured. A classical optimizer then adjusts the quantum circuit parameters to find the lowest eigenvalue of the Hamiltonian.

The starting point of the VQE is an electronic Hamiltonian expressed in the Pauli basis — however, determining the Pauli-basis representation from the molecular structure is highly non-trivial, requiring use of both self-consistent field methods as well as mapping of Fermionic states and operators to qubits. PennyLane provides a quantum chemistry package that, with a single line of code, can be used to generate the electronic Hamiltonian of a molecule. It employs the quantum chemistry packages PySCF [46], Psi4 [47, 48], and OpenFermion [49]. It can be installed using the command `pip install pennylane-qchem`.

To build the Hamiltonian, it is necessary to specify the geometry of the molecule. This can be input using an XYZ file format, containing the total number of atoms in the molecule, their atomic symbols, and positions in Cartesian coordinates. If Open Babel is installed [50], any format it recognizes, such as `.mol` or `.sdf`, is also supported by PennyLane. Additional information includes the charge of the molecule, the spin-multiplicity of the Hartree-Fock state, the atomic basis set, and the fermionic-to-qubit mapping. The following example code generates the qubit Hamiltonian for the neutral hydrogen molecule using the `sto-3g` basis set for atomic orbitals and the Jordan-Wigner

fermionic-to-qubit mapping:

```
h, nr_qubits = qml.qchem.generate_hamiltonian(
    mol_name='h2',
    mol_geo_file='h2.xyz',
    mol_charge=0,
    multiplicity=1,
    basis_set='sto-3g',
    mapping='jordan_wigner'
)
```

Codeblock 14: *Generating the electronic Hamiltonian of the Hydrogen molecule using the input file* h2.xyz, *which encodes the molecular structure.*

Once the Hamiltonian has been generated, the VQE cost function can be constructed using the VQECost class:

```
import pennylane as qml
from qml.init import strong_ent_layers_normal
from qml.templates import
↪ StronglyEntanglingLayers

dev = qml.device('default.qubit', wires=4)
hf_state = np.array([1, 1, 0, 0])

def ansatz(x, wires):
    qml.BasisState(hf_state, wires=wires)
    StronglyEntanglingLayers(x, wires=wires)

cost = qml.VQECost(ansatz, h, dev, 'torch')
init_params = strong_ent_layers_normal(5, 4)
params = torch.tensor(init_params)
cost(params)
```

Codeblock 15: *Constructing a VQE cost function using the PyTorch interface.*

Note that the cost function can also be constructed as a QNode collection:

```
qnodes = qml.map(ansaztz, h.ops, dev, "torch")
cost = qml.dot(h.coeffs, qnodes)
```

Codeblock 16: *Constructing a VQE cost function using map to create the QNode collection explicitly.*

Writing a plugin

PennyLane was designed with extensibility in mind, providing an API for both hardware devices and software simulators to easily connect and allow PennyLane access to their frameworks. This enables the automatic differentiation and optimization features of PennyLane to be used on an external framework with minimal effort. As a result, PennyLane is inherently hardware agnostic — the user is able to construct hybrid computational graphs containing QNodes executed on an arbitrary number of different devices, and even

reuse quantum circuits across different devices. As of version 0.8, PennyLane has plugins available for Strawberry Fields [25, 51], Rigetti Forest [52, 53], Qiskit [43, 44], Cirq [54], ProjectQ [22, 55]. These bring access to the following devices, respectively:

- strawberryfields.fock, and strawberryfields.gaussian;

- forest.numpywavefunction, forest.wavefunction, forest.qvm, and forest.qpu;

- qiskit.basicaer, qiskit.aer, and qiskit.ibm;

- cirq.simulator;

- projectq.ibm, projectq.simulator, and projectq.classical.

PennyLane also contains a built-in reference plugin with two devices default.gaussian and default.qubit, as well as a tensor network plugin default.tensor that uses the TensorNetwork library [56] with the TensorFlow backend for high performance simulation. In addition, there is the community contributed PennyLane-Qulacs plugin [57], which makes available the Qulacs high performance C++ simulator [58] as a PennyLane device.

In PennyLane, there is a subtle distinction between the terms 'plugin' and 'device':

- A plugin is an external Python package that provides additional quantum *devices* to PennyLane.

- Each plugin may provide one (or more) devices, that are accessible directly by PennyLane, as well as any additional private functions or classes.

Once installed, these devices can be loaded directly from PennyLane without any additional steps required by the user. Depending on the scope of the plugin, a plugin can also provide custom quantum operations, observables, and functions that extend PennyLane — for example by converting from the target framework's quantum circuit representation directly to a QNode supporting autodifferentiation[9]. In the remainder of this section, we briefly describe the plugin API of PennyLane, and how it can be used to provide new quantum devices.

Devices

When performing a hybrid computation using PennyLane, one of the first steps is to specify the quantum devices which will be used by quantum nodes. As seen above, this is done as follows:

---

[9] One example being the PennyLane-Qiskit plugin, which provides conversion functions qml.from_qasm() and qml.from_qiskit() — allowing QNodes to be created from QASM and Qiskit quantum programs respectively.

```
import pennylane as qml
dev1 = qml.device(short_name, wires=2)
```

Codeblock 17: Loading a PennyLane-compatible device.

where `short_name` is a string which uniquely identifies the device provided. In general, the short name has the following form: `pluginname.devicename`.

Creating a new device

The first step in making a PennyLane plugin is creating the device class. This is as simple as importing the abstract base class `Device` from PennyLane, and subclassing it [10]:

```
from pennylane import Device

class MyDevice(Device):
    """MyDevice docstring"""
    name = 'My custom device'
    short_name = 'example.mydevice'
    pennylane_requires = '0.1.0'
    version = '0.0.1'
    author = 'Ada Lovelace'
```

Codeblock 18: Creating a custom PennyLane-compatible device.

Here, we have begun defining some important class attributes ('identifiers') that allow PennyLane to recognize the device. These include:

- `Device.name`: a string containing the official name of the device
- `Device.short_name`: the string used to identify and load the device by users of PennyLane
- `Device.pennylane_requires`: the version number(s) of PennyLane that this device is compatible with; if the user attempts to load the device on a different version of PennyLane, a `DeviceError` will be raised
- `Device.version`: the version number of the device
- `Device.author`: the author of the device

Defining all these attributes is mandatory.

Supporting operations and expectations

Plugins must inform PennyLane about the operations and expectations that the device supports, as well as potentially further capabilities, by providing the following class attributes/properties:

- `Device.operations`: a set of the supported PennyLane operations as strings, e.g., `operations = {"CNOT", "PauliX"}`. This is used to decide whether an operation is supported by your device in the default implementation of the public method `Device.supported()`.

- `Device.observables`: a set of the supported PennyLane observables as strings, e.g., `observables = {"PauliX", "Hadamard", "Hermitian"}`. This is used to decide whether an observable is supported by your device in the default implementation of the public method `Device.supported()`.

- `Device._capabilities`: a dictionary containing information about the capabilities of the device. For example, the key `'model'`, which has value either `'qubit'` or `'CV'`, indicates to PennyLane the computing model supported by the device. This class dictionary may also be used to return additional information to the user — this is accessible from the PennyLane frontend via the public method `Device.capabilities`.

A subclass of the `Device` class, `QubitDevice`, is provided for easy integration with simulators and hardware devices that utilize the qubit model. `QubitDevice` provides automatic support for all supported observables, including tensor observables. For a better idea of how these required device properties work, refer to the two reference devices.

Applying operations and measuring statistics

Once all the class attributes are defined, it is necessary to define some required class methods, to allow PennyLane to apply operations to your device. In the following examples, we focus on the `QubitDevice` subclass. When PennyLane evaluates a QNode, it calls the `Device.execute` method, which performs the following process:

```
self.check_validity(circuit.operations,
↪   circuit.observables)

# apply all circuit operations
self.apply(circuit.operations,
↪   rotations=circuit.diagonalizing_gates)

# generate computational basis samples
if (not self.analytic) or circuit.is_sampled:
    self._samples = self.generate_samples()

# compute the required statistics
results = self.statistics(circuit.observables)

return self._asarray(results)
```

Codeblock 19: The PennyLane `Device.execute` method, called whenever a quantum node is evaluated.

In most cases, there are a minimum of two methods that need to be defined:

- `Device.apply`: Accepts a list of PennyLane Operations to be applied. The corresponding quantum operations are applied to the device, the circuit rotated into the measurement basis, and, if relevant, the quantum circuit compiled and executed.

- `Device.probability`: Returns the (marginal) probability of each computational basis state from the last run of the device.

In addition, if the device generates/returns its own computational basis samples for measured modes after execution, the following method must also be defined:

- `Device.generate_samples`: Generate computational basis samples for all wires. If `Device.generate_samples` is not defined, PennyLane will automatically generate samples using the output of the device probability.

Once the required methods are defined, the inherited methods `Device.expval`, `Device.var`, and `Device.sample` can be passed an observable (or tensor product of observables), returning the corresponding measurement statistic.

Installation and testing

PennyLane uses a `setuptools entry_points` approach to plugin integration. In order to make a plugin accessible to PennyLane, the following keyword argument to the `setup` function must be provided in the plugin's `setup.py` file:

```python
devices_list = [
'myplugin.mydev1 = MyMod.MySubMod:MyDev1'
'myplugin.mydev2 = MyMod.MySubMod:MyDev2'
],
setup(entry_points={'pennylane.plugins':
↪ devices_list})
```

Codeblock 20: Creating the PennyLane device entry points.

Here, `devices_list` is a list of devices to be registered, `myplugin.mydev1` is the short name of the device, and `MyMod.MySubMod` is the path to the Device class, `MyDev1`. To ensure the device is working as expected, it can be installed in developer mode using `pip install -e pluginpath`, where `pluginpath` is the location of the plugin. It will then be accessible via PennyLane.

All plugins should come with unit tests, to ensure that the device supports the correct gates and observables, and is applying them correctly. For an example of a plugin test suite, see `tests/test_default_qubit.py` and `tests/test_default_gaussian.py` in the main PennyLane repository. In general, as all supported operations have their gradient formula defined and tested by PennyLane, testing that the device calculates the correct gradients is not required — it is sufficient to test that it *applies* and *measures* quantum operations and observables correctly.

Supporting new operations

PennyLane also provides the ability to add custom operations or observables to be executed on the plugin device, that may not be currently supported by PennyLane. For qubit architectures this is done by subclassing the `Operation` and `Observable` classes, defining the number of parameters the operation takes, and the number of wires the operation acts on. In addition, if the method of analytic differentiation of the operation with respect to any parameters is known, the corresponding `grad_recipe` should be provided, to open up analytic differentiation support in PennyLane.

For example, to define the U2 gate, which depends on parameters ϕ and λ, we create the following class:

```python
class U2(Operation):
  """U2 gate."""
  num_params = 2
  num_wires = 1
  par_domain = "R"
  grad_method = "A"
  grad_recipe = None

  @staticmethod
  def _matrix(*params):
    phi, lam = params
    return PhaseShift._matrix(phi+lam) @
    ↪  Rot._matrix(lam, np.pi/2, -lam)

  @staticmethod
  def decomposition(phi, lam, wires):
    decomp_ops = [
      Rot(lam, np.pi / 2, -lam, wires=wires),
      PhaseShift(lam, wires=wires),
      PhaseShift(phi, wires=wires),
    ]
    return decomp_ops
```

Codeblock 21: Creating a custom qubit operation.

where the following quantities *must* be declared:

- `Operation.num_params`: the number of parameters the operation takes

- `Operation.num_wires`: the number of wires the operation acts on

- `Operation.par_domain`: the domain of the gate parameters; `'N'` for natural numbers (including zero), `'R'` for floats, `'A'` for arrays of floats/complex numbers, and `None` if the gate does not have free parameters

- `Operation.grad_method`: the gradient computation method; `'A'` for the analytic method, `'F'` for finite differences, and `None` if the operation may not be differentiated

- `Operation.grad_recipe`: the gradient recipe for the analytic `'A'` method. This is a list with one tuple

per operation parameter. For parameter k, the tuple is of the form (c_k, s_k), resulting in a gradient recipe of

$$\frac{d}{d\phi_k}O = c_k\left[O(\phi_k + s_k) - O(\phi_k - s_k)\right].$$

Note that if `grad_recipe = None`, the default gradient recipe $(c_k, s_k) = (1/2, \pi/2)$ is used for every parameter.

- `Operation._matrix` (optional): returns the matrix representation of the operator in the computational basis for the provided parameter values.

- `Operation.decomposition` (optional): returns a list of PennyLane operations that decompose the custom gate for the provided parameter values.

The user can then import this operation directly from your plugin, and use it when defining a `QNode`:

```python
import pennylane as qml
from MyModule.MySubModule import Ising

@qnode(dev1)
def my_qfunc(phi):
    qml.Hadamard(wires=0)
    Ising(phi, wires=[0,1])
    return qml.expval.PauliZ(1)
```

Codeblock 22: *Using a plugin-provided custom operation.*

In this case, as the plugin is providing a custom operation not supported by PennyLane, it is recommended that the plugin unit tests *do* provide tests to ensure that PennyLane returns the correct gradient for the custom operations.

Custom observables

Custom observables can be added in an identical manner to operations above, but with three small changes:

- The `Observable` class should instead be subclassed.

- The class attribute `Observable.eigvals` should be defined, returning a one-dimensional array of eigenvalues of the observable.

- The method `Observable.diagonalizing_gates` should be defined. This method returns a list of PennyLane `Operation` objects that diagonalize the observable in the computational basis. This is used to support devices that can only perform measurements in the computational basis.

Custom CV operations and expectations

For custom continuous-variable operations or expectations, the `CVOperation` or `CVObservable` classes must be subclassed instead. In addition, for CV operations with known analytic gradient formulas (such as Gaussian operations), the static class method `CV._heisenberg_rep` must be defined:

```python
class Custom(CVOperation):
    """Custom gate"""
    n_params = 2
    n_wires = 1
    par_domain = 'R'
    grad_method = 'A'
    grad_recipe = None

    @staticmethod
    def _heisenberg_rep(params):
        return function(params)
```

Codeblock 23: *Creating a custom continuous-variable operation.*

For operations, the `_heisenberg_rep` method should return the Heisenberg representation of the operation, i.e., the matrix of the linear transformation carried out by the operation for the given parameter values[11]. This is used internally for calculating the gradient using the analytic method (`grad_method = 'A'`). For observables, this method should return a real vector (first-order observables) or symmetric matrix (second-order observables) of coefficients which represent the expansion of the observable in the basis of monomials of the quadrature operators. For single-mode operations we use the basis $\mathbf{r} = (\mathbb{I}, \hat{x}, \hat{p})$, and for multi-mode operations the basis $\mathbf{r} = (\mathbb{I}, \hat{x}_0, \hat{p}_0, \hat{x}_1, \hat{p}_1, \ldots)$, where $\hat{x}_k$ and $\hat{p}_k$ are the quadrature operators of qumode k. Note that, for every gate, even if the analytic gradient formula is not known or if `_heisenberg_rep` is not provided, PennyLane continues to support the finite difference method of gradient computation.

Conclusion

We have introduced PennyLane, a Python package that extends automatic differentiation to quantum and hybrid classical-quantum information processing. This is accomplished by introducing a new *quantum node* abstraction which interfaces cleanly with existing DAG-based automatic differentiation methods like the backpropagation algorithm. The ability to compute gradients of variational quantum circuits – and to integrate these seamlessly as part of larger hybrid computations – opens up a wealth of potential applications, in particular for optimization and machine learning tasks.

We envision PennyLane as a powerful tool for many research directions in quantum computing and quantum machine learning, similar to how libraries like TensorFlow or PyTorch have become indispensible for research in deep learning. With small quantum processors becoming pub-

---

[11] Specifically, if the operation carries out a unitary transformation U, this method should return the matrix for the adjoint action $U^\dagger(\cdot)U$.

licly available, and with the emergence of variational quantum circuits as a new algorithmic paradigm, the quantum computing community has begun to embrace heuristic algorithms more and more. This spirit is already common in the classical machine learning community and has – together with dedicated software enabling rapid exploration of computational models – allowed that field to develop at a remarkable pace. With PennyLane, tools are now freely available to investigate model structures, training strategies, and optimization landscapes within hybrid and quantum machine learning, to explore existing and new variational circuit architectures, and to design completely new algorithms by circuit learning.

[1] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'Brien, "A variational eigenvalue solver on a photonic quantum processor," Nature Communications **5**, 4213 (2014).

[2] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann, "A quantum approximate optimization algorithm," arXiv preprint (2014), arxiv:1411.4028.

[3] Eric R Anschuetz, Jonathan P Olson, Alán Aspuru-Guzik, and Yudong Cao, "Variational quantum factoring," arXiv preprint (2018), arxiv:1808.08927.

[4] Ryan LaRose, Arkin Tikku, Étude O'Neel-Judy, Lukasz Cincio, and Patrick J Coles, "Variational quantum state diagonalization," arXiv preprint (2018), arxiv:1810.10506.

[5] Jonathan Romero, Jonathan P Olson, and Alan Aspuru-Guzik, "Quantum autoencoders for efficient compression of quantum data," Quantum Science and Technology **2**, 045001 (2017).

[6] Peter D Johnson, Jonathan Romero, Jonathan Olson, Yudong Cao, and Alán Aspuru-Guzik, "QVECTOR: an algorithm for device-tailored quantum error correction," arXiv preprint (2017), arxiv:1711.02249.

[7] Guillaume Verdon, Michael Broughton, and Jacob Biamonte, "A quantum algorithm to train neural networks using low-depth circuits," arXiv preprint (2017), arxiv:1712.05304.

[8] Edward Farhi and Hartmut Neven, "Classification with quantum neural networks on near term processors," arXiv preprint (2018), arxiv:1802.06002.

[9] Maria Schuld and Nathan Killoran, "Quantum machine learning in feature Hilbert spaces," arXiv preprint (2018), arxiv:1803.07128.

[10] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii, "Quantum circuit learning," Phys. Rev. A **98**, 032309 (2018), arxiv:1803.00745.

[11] Maria Schuld, Alex Bocharov, Krysta Svore, and Nathan Wiebe, "Circuit-centric quantum classifiers," arXiv preprint (2018), arxiv:1804.00633.

[12] Edward Grant, Marcello Benedetti, Shuxiang Cao, Andrew Hallam, Joshua Lockhart, Vid Stojevic, Andrew G Green, and Simone Severini, "Hierarchical quantum classifiers," arXiv preprint (2018), arxiv:1804.03680.

[13] Jin-Guo Liu and Lei Wang, "Differentiable learning of quantum circuit Born machine," arXiv preprint (2018), arxiv:1804.04168.

[14] Pierre-Luc Dallaire-Demers and Nathan Killoran, "Quantum generative adversarial networks," Physical Review A **98**, 012324 (2018).

[15] Vojtech Havlicek, Antonio D Córcoles, Kristan Temme, Aram W Harrow, Jerry M Chow, and Jay M Gambetta, "Supervised learning with quantum enhanced feature spaces," arXiv preprint (2018), arxiv:1804.11326.

[16] Hongxiang Chen, Leonard Wossnig, Simone Severini, Hartmut Neven, and Masoud Mohseni, "Universal discriminative quantum neural networks," arXiv preprint (2018), arxiv:1805.08654.

[17] Nathan Killoran, Thomas R Bromley, Juan Miguel Arrazola, Maria Schuld, Nicolás Quesada, and Seth Lloyd, "Continuous-variable quantum neural networks," arXiv preprint (2018), arxiv:1806.06871.

[18] Gregory R Steinbrecher, Jonathan P Olson, Dirk Englund, and Jacques Carolan, "Quantum optical neural networks," arXiv preprint (2018), arxiv:1808.10047.

[19] Dave Wecker and Krysta M. Svore, "LIQUi|>: A software design architecture and domain-specific language for quantum computing," arXiv preprint (2014), arxiv:1402.4467.

[20] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik, "qHiPSTER: the quantum high performance software testing environment," arXiv preprint (2016), arxiv:1601.07195.

[21] IBM Corporation, "Qiskit," (2016).

[22] Damian S Steiger, Thomas Häner, and Matthias Troyer, "ProjectQ: an open source software framework for quantum computing," Quantum **2**, 49 (2018).

[23] Rigetti Computing, "Forest SDK," (2017).

[24] Microsoft Corporation, "Quantum Development Kit," (2017).

[25] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook, "Strawberry Fields: A software platform for photonic quantum computing," arXiv preprint (2018), arxiv:1804.03159.

[26] Google Inc., "Cirq," (2018).

[27] IBM Corporation, "IBM Quantum Experience," (2016).

[28] Rigetti Computing, "QPU Specifications," (2017).

[29] Xanadu Inc., "Quantum machine learning toolbox," (2018).

[30] Sukin Sim, Yudong Cao, Jonathan Romero, Peter D Johnson, and Alan Aspuru-Guzik, "A framework for algorithm deployment on cloud-based quantum computers," arXiv preprint (2018), arxiv:1810.10576.

[31] Dougal Maclaurin, David Duvenaud, and Ryan P Adams, "Autograd: Effortless gradients in numpy," in *ICML 2015 AutoML Workshop* (2015).

[32] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: a system for large-scale machine learning." in *OSDI*, Vol. 16 (USENIX Association, Berkeley, CA, USA, 2016) pp. 265–283.

[33] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer, "Automatic dif-

ferentiation in PyTorch," (2017).

[34] Seth Lloyd and Christian Weedbrook, "Quantum generative adversarial learning," Physical Review Letters **121**, 040502 (2018).

[35] Dougal Maclaurin, *Modeling, inference and optimization with composable differentiable procedures*, Ph.D. thesis, Harvard University, Graduate School of Arts & Sciences (2016).

[36] Atılım Güneş Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind, "Automatic differentiation in machine learning: a survey." Journal of Machine Learning Research **18**, 1–153 (2018).

[37] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik, "The theory of variational hybrid quantum-classical algorithms," New Journal of Physics **18**, 023023 (2016).

[38] Yuxuan Du, Min-Hsiu Hsieh, Tongliang Liu, and Dacheng Tao, "The expressive power of parameterized quantum circuits," arXiv preprint (2018), arxiv:1810.11922.

[39] William Huggins, Piyush Patel, K Birgitta Whaley, and E Miles Stoudenmire, "Towards quantum machine learning with tensor networks," arXiv preprint (2018), arxiv:1803.11537.

[40] Xanadu Inc., "PennyLane," (2018).

[41] Gian Giacomo Guerreschi and Mikhail Smelyanskiy, "Practical optimization for hybrid quantum-classical algorithms," arXiv preprint (2017), arxiv:1701.01450.

[42] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran, "Evaluating analytic gradients on quantum hardware," arXiv preprint (2018), arxiv:1811.11184.

[43] David C McKay, Thomas Alexander, Luciano Bello, Michael J Biercuk, Lev Bishop, Jiayin Chen, Jerry M Chow, Antonio D Córcoles, Daniel Egger, Stefan Filipp, *et al.*, "Qiskit backend specifications for openqasm and openpulse experiments," arXiv preprint arXiv:1809.03452 (2018).

[44] Carsten Blank, "PennyLane Qiskit plugin," (2019).

[45] James Stokes, Josh Izaac, Nathan Killoran, and Giuseppe Carleo, "Quantum natural gradient," arXiv preprint arXiv:1909.02108 (2019).

[46] Qiming Sun, Timothy C Berkelbach, Nick S Blunt, George H Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D McClain, Elvira R Sayfutyarova, Sandeep Sharma, *et al.*, "Pyscf: the python-based simulations of chemistry framework," Wiley Interdisciplinary Reviews: Computational Molecular Science **8**, e1340 (2018).

[47] Justin M Turney, Andrew C Simmonett, Robert M Parrish, Edward G Hohenstein, Francesco A Evangelista, Justin T Fermann, Benjamin J Mintz, Lori A Burns, Jeremiah J Wilke, Micah L Abrams, *et al.*, "Psi4: an open-source ab initio electronic structure program," Wiley Interdisciplinary Reviews: Computational Molecular Science **2**, 556–565 (2012).

[48] Robert M Parrish, Lori A Burns, Daniel GA Smith, Andrew C Simmonett, A Eugene DePrince III, Edward G Hohenstein, Ugur Bozkaya, Alexander Yu Sokolov, Roberto Di Remigio, Ryan M Richard, *et al.*, "Psi4 1.1: An open-source electronic structure program emphasizing automation, advanced libraries, and interoperability," Journal of Chemical Theory and Computation **13**, 3185–3197 (2017).

[49] Jarrod R McClean, Kevin J Sung, Ian D Kivlichan, Yudong Cao, Chengyu Dai, E Schuyler Fried, Craig Gidney, Brendan Gimby, Pranav Gokhale, Thomas Häner, *et al.*, "Openfermion: the electronic structure package for quantum computers," arXiv:1710.07629 (2017).

[50] Noel M O'Boyle, Michael Banck, Craig A James, Chris Morley, Tim Vandermeersch, and Geoffrey R Hutchison, "Open babel: An open chemical toolbox," Journal of cheminformatics **3**, 33 (2011).

[51] Xanadu Inc., "PennyLane Strawberry Fields plugin," (2018).

[52] Robert S Smith, Michael J Curtis, and William J Zeng, "A practical quantum instruction set architecture," arXiv preprint arXiv:1608.03355 (2016).

[53] Rigetti Computing, "PennyLane Forest plugin," (2019).

[54] Johannes Jakob Meyer, "PennyLane Cirq plugin," (2019).

[55] Xanadu Inc., "PennyLane ProjectQ plugin," (2018).

[56] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer, "Tensornetwork: A library for physics and machine learning," arXiv preprint arXiv:1905.01330 (2019).

[57] Steven Oud, "PennyLane Qulacs plugin," (2019).

[58] QunaSys, "Qulacs," (2019).