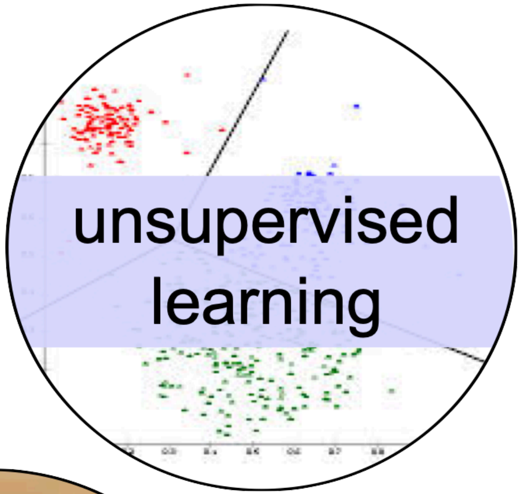
Software Environments for Quantum Machine Learning

Dániel Nagy

Types of machine learning

Need a lot of labeled data



Learning from unlabeled data

Learning from interactions with an environment

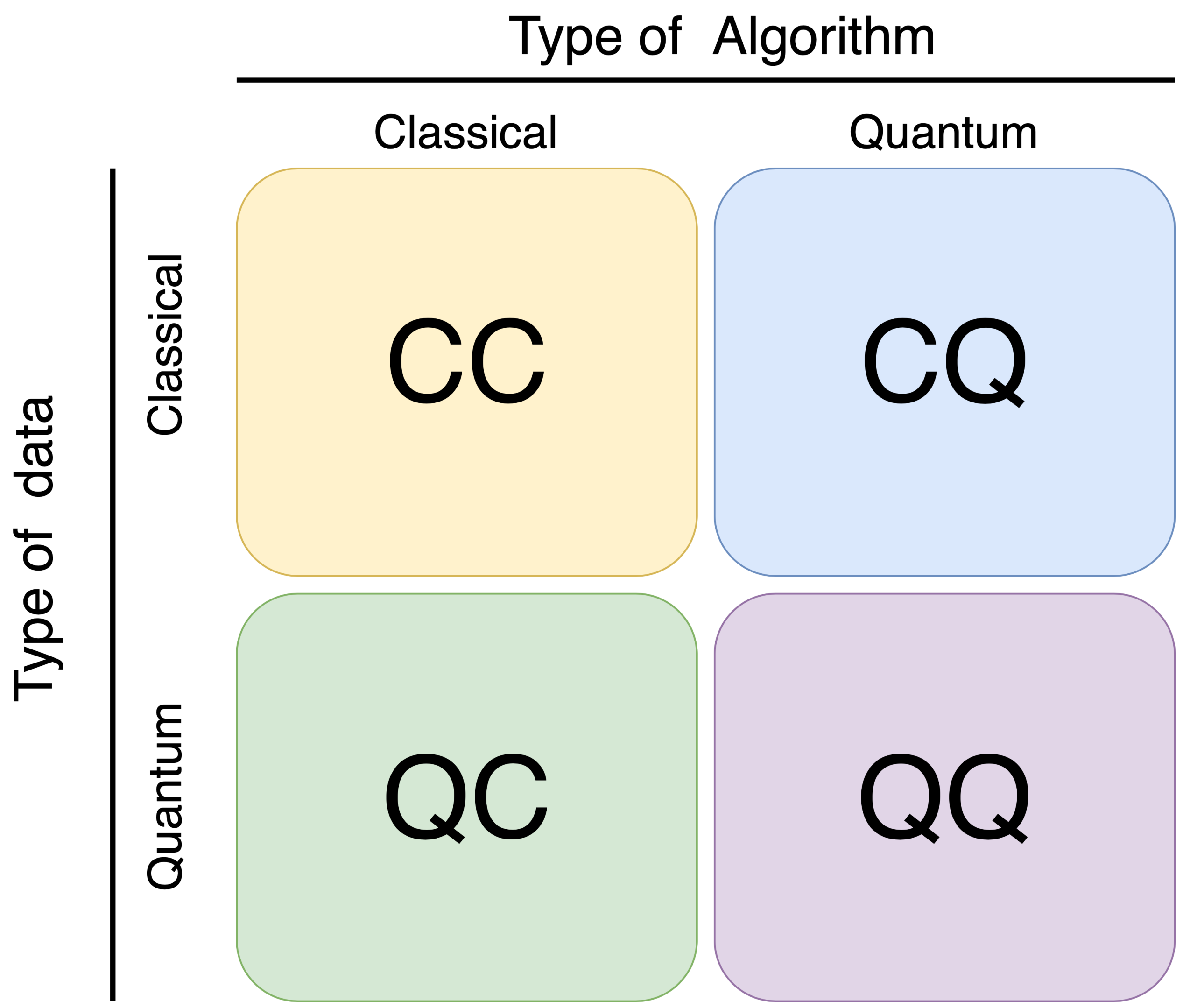
Why quantum machine learning?

- ML is linear algebra + nonlinearities
- QM is linear algebra + measurements
- ML deals with probability distributions, which naturally appear in QM.

Why quantum machine learning?

- The dimensionality of the Hilbert-space scales exponentially with the number of qubits.
- Quantum circuits can generate probability distributions that are impossible to generate with classical computers.
- They may be able to learn distributions that would be infeasible on classical computers.

Classifying QML algorithms by quantumness



- CC: every classical ML algorithm
- CQ: variational quantum regression, variational quantum classification e.g. q-SVM [arxiv]
- QC: ML assisted quantum error correction [arxiv]
- QQ: Quantum Autoencoders to Denoise Quantum Data [PRL], VQC for state tomography [arxiv]

Supervised Learning – Gradient descent

$$\begin{tikzcd} x \arrow[r] & y_{pred} = f(x;\theta) \arrow[d] \\ y_{true} \arrow[r] & L = \mathcal{L}(y_{pred}, y_{true}) \end{tikzcd}$$

The model is represented by an almost everywhere differentiable function, $f(x;\theta)$

$$\theta_{t+1} = \theta_t - \eta_t \frac{\partial L}{\partial \theta}$$

Quantum Supervised Learning – Gradient descent

$$\begin{tikzcd}
x \arrow[r, "\text{embedding}", "\text{Performance depends heavily on proper embedding}"'] & |\psi\rangle \arrow[r] & y_{pred} = f(x;\theta) = \langle\psi|U(\theta)|\psi\rangle \arrow[d, "\text{Actually, estimates of the expval}"] \\
& y_{true} \arrow[r] & L = \mathcal{L}(y_{pred}, y_{true})
\end{tikzcd}$$

The model is represented by a unitary, $U(\theta)$.

$$\theta_{t+1} = \theta_t - \eta_t \frac{\partial L}{\partial \theta}$$

How to calculate the gradient of a quantum node?

- Parameter-shift rule:
  - If $f(\mu)$ is a quantum node, then

$$\partial_\mu f(\mu) = c[f(\mu + s) - f(\mu - s)]$$

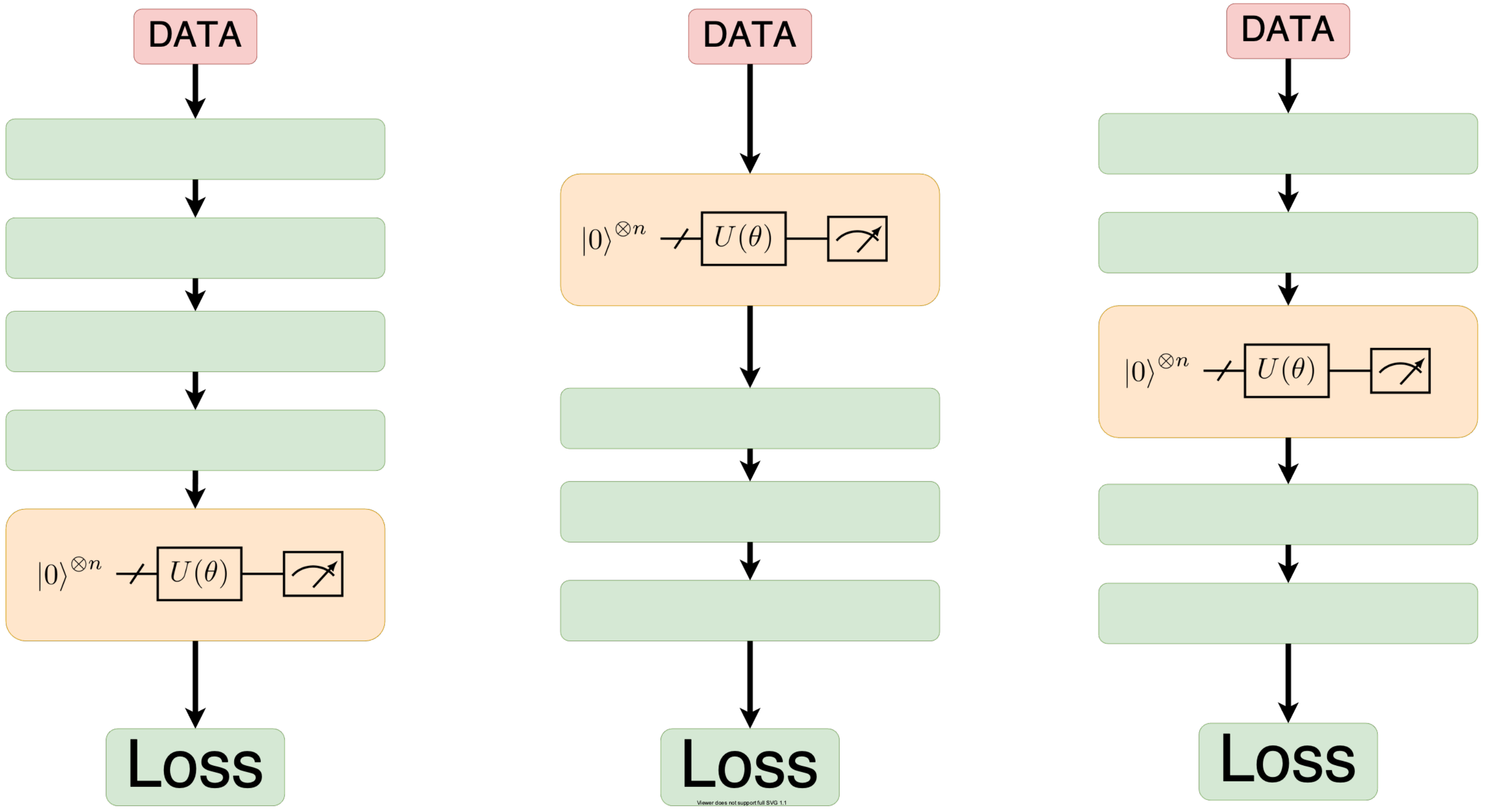
(c and s are finite parameters from a lookup table)

- Finite difference method
  - If $f(\mu)$ is a quantum node, then

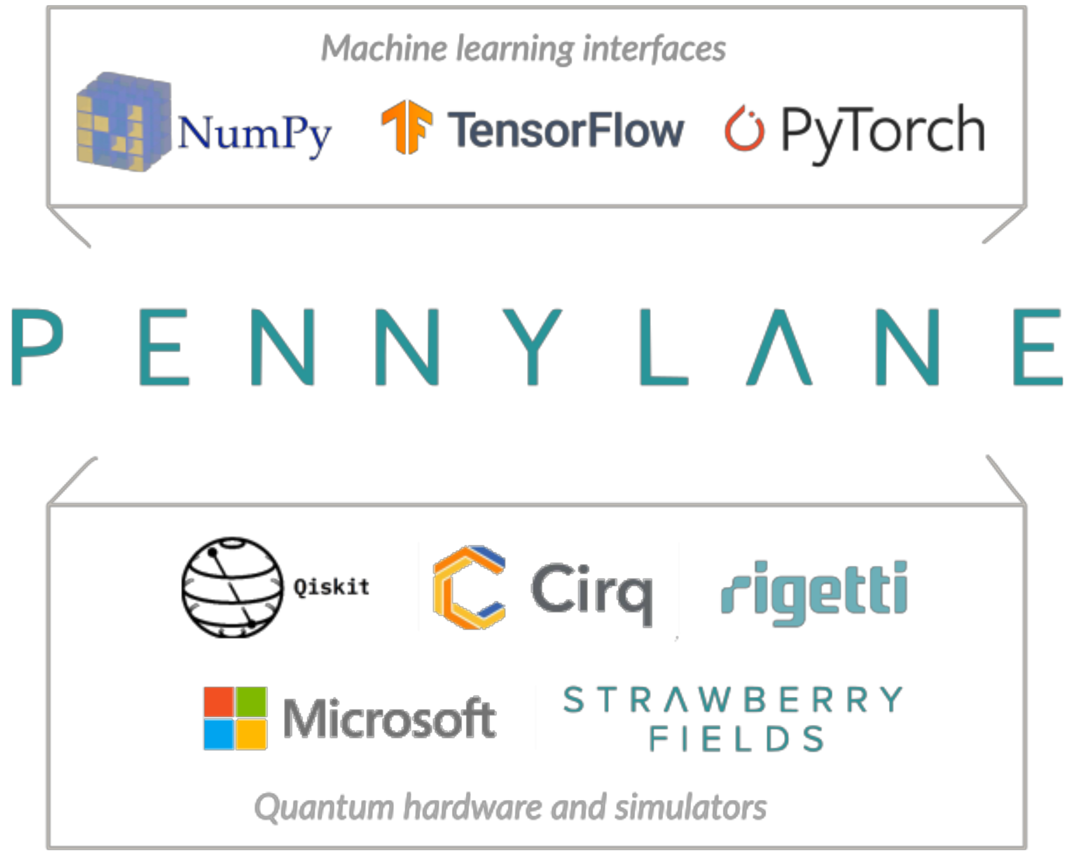
$$\partial_\mu f(\mu) \approx \frac{f\left(\mu + \frac{1}{2}\Delta\mu\right) - f(\mu - \frac{1}{2}\Delta\mu)}{\Delta\mu}$$

What do we have now? Hybrid devices

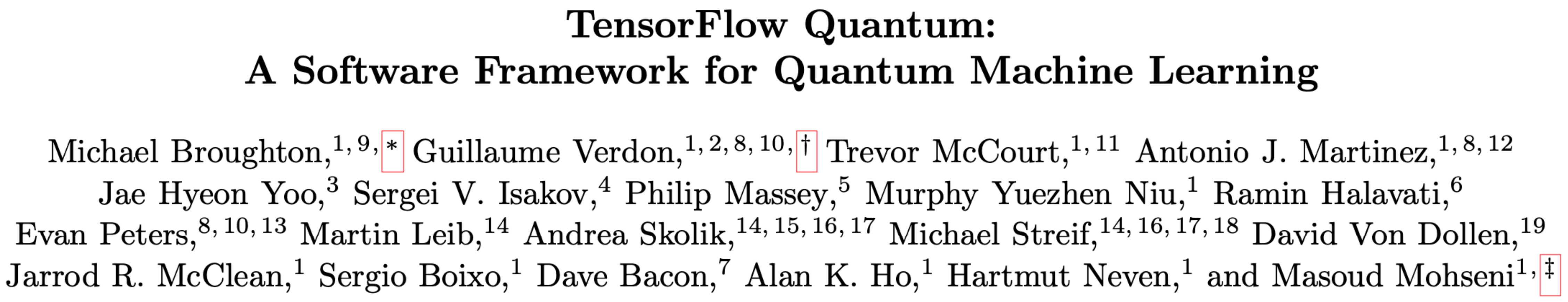
- High performance classical computers
- Small and noisy quantum devices
- => Build hybrid quantum-classical algorithms
- Compose models from both quantum and classical nodes



Software packages for QML

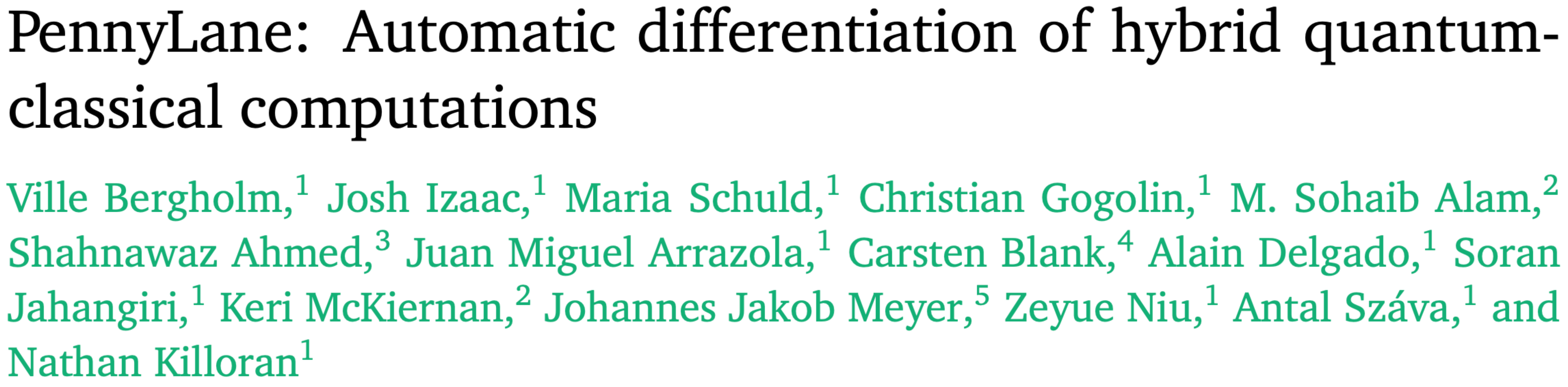


TensorFlow Quantum



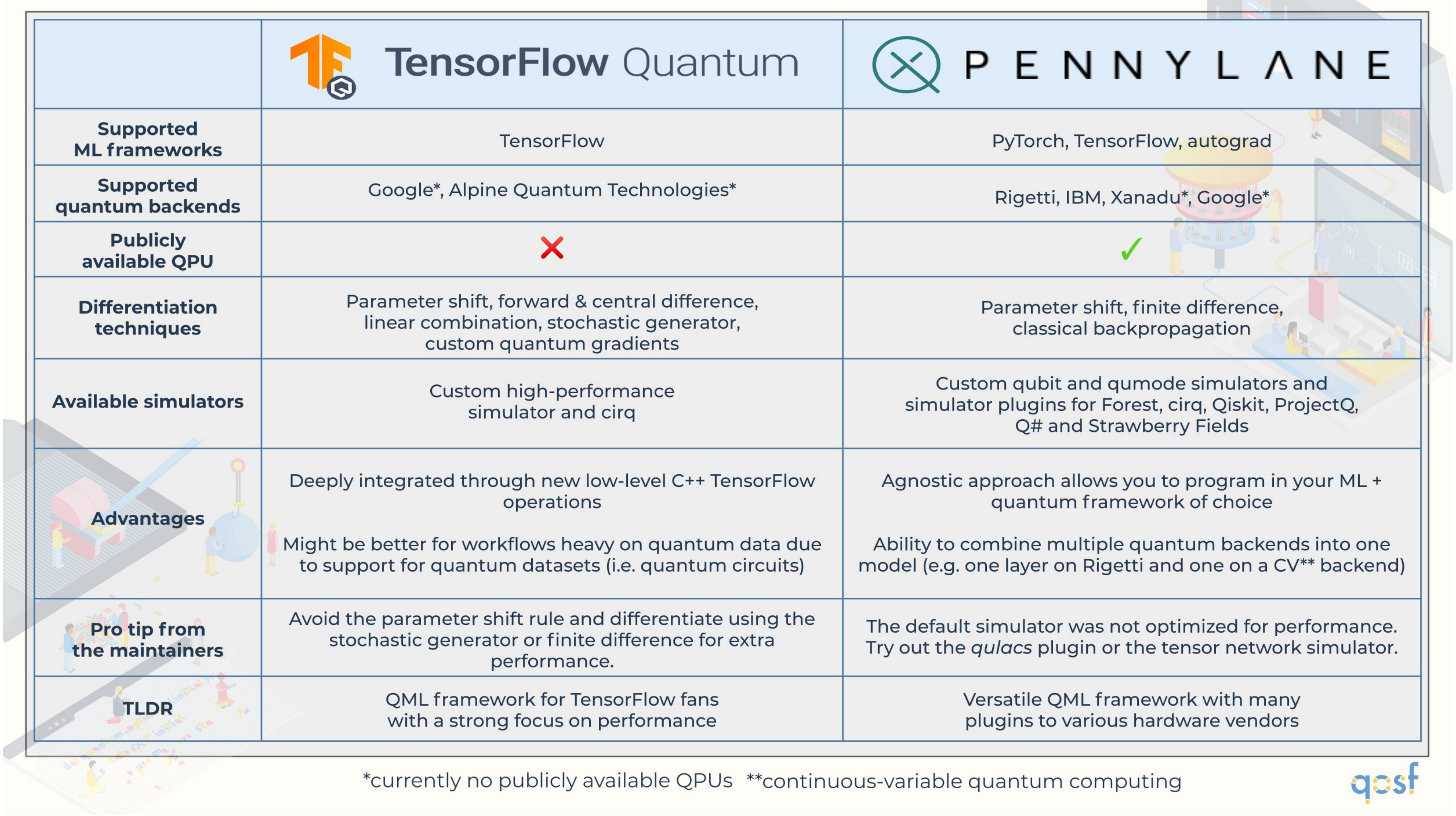
TensorFlow Quantum:
A Software Framework for Quantum Machine Learning

Michael Broughton,[1, 9, *] Guillaume Verdon,[1, 2, 8, 10, †] Trevor McCourt,[1, 11] Antonio J. Martinez,[1, 8, 12]
Jae Hyeon Yoo,[3] Sergei V. Isakov,[4] Philip Massey,[5] Murphy Yuezhen Niu,[1] Ramin Halavati,[6]
Evan Peters,[8, 10, 13] Martin Leib,[14] Andrea Skolik,[14, 15, 16, 17] Michael Streif,[14, 16, 17, 18] David Von Dollen,[19]
Jarrod R. McClean,[1] Sergio Boixo,[1] Dave Bacon,[7] Alan K. Ho,[1] Hartmut Neven,[1] and Masoud Mohseni[1, ‡]



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]



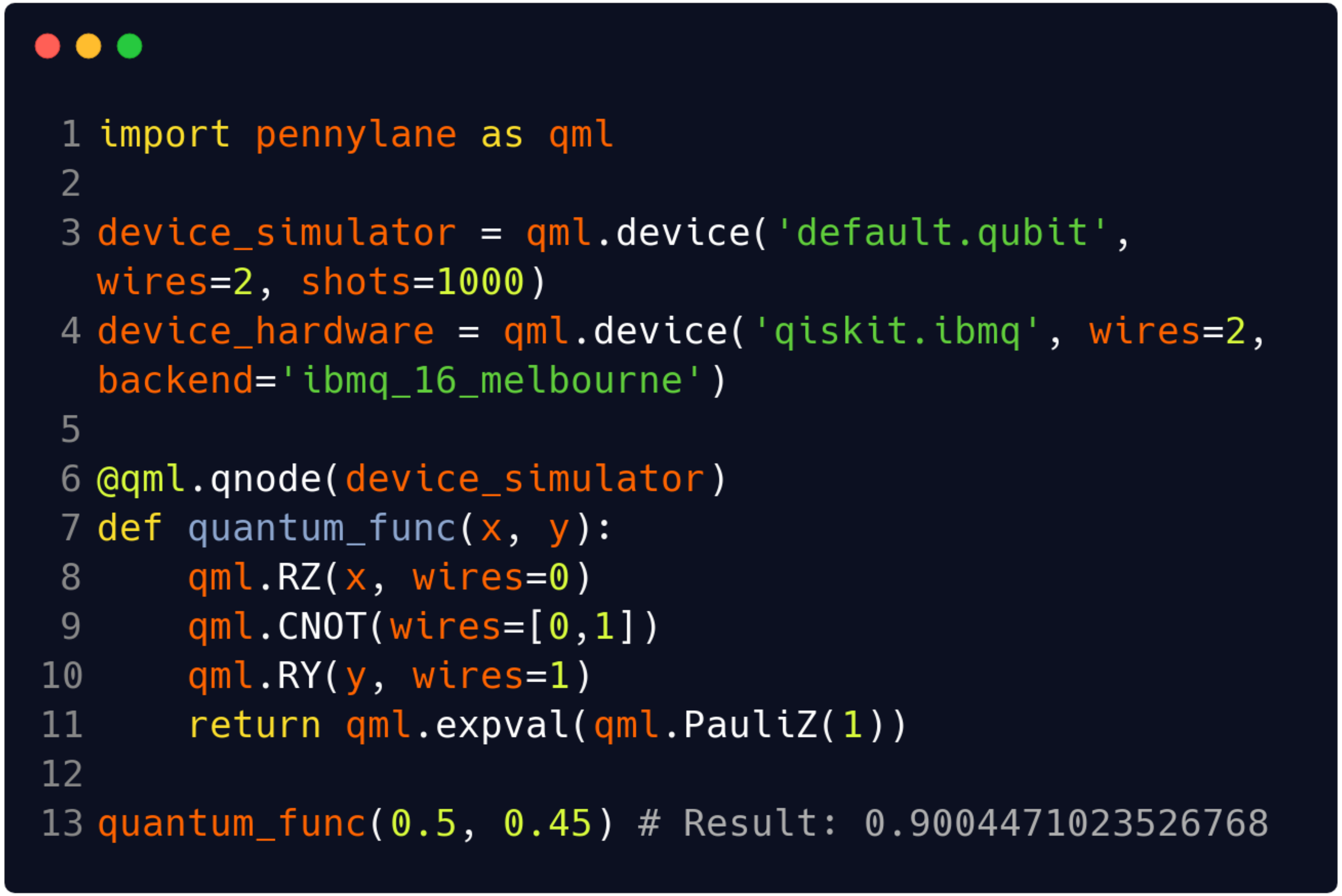
	TensorFlow Quantum	PENNYLANE
Supported ML frameworks	TensorFlow	PyTorch, TensorFlow, autograd
Supported quantum backends	Google*, Alpine Quantum Technologies*	Rigetti, IBM, Xanadu*, Google*
Publicly available QPU	✗	✓
Differentiation techniques	Parameter shift, forward & central difference, linear combination, stochastic generator, custom quantum gradients	Parameter shift, finite difference, classical backpropagation
Available simulators	Custom high-performance simulator and cirq	Custom qubit and qumode simulators and simulator plugins for Forest, cirq, Qiskit, ProjectQ, Q# and Strawberry Fields
Advantages	Deeply integrated through new low-level C++ TensorFlow operations Might be better for workflows heavy on quantum data due to support for quantum datasets (i.e. quantum circuits)	Agnostic approach allows you to program in your ML + quantum framework of choice Ability to combine multiple quantum backends into one model (e.g. one layer on Rigetti and one on a CV** backend)
Pro tip from the maintainers	Avoid the parameter shift rule and differentiate using the stochastic generator or finite difference for extra performance.	The default simulator was not optimized for performance. Try out the *qulacs* plugin or the tensor network simulator.
TLDR	QML framework for TensorFlow fans with a strong focus on performance	Versatile QML framework with many plugins to various hardware vendors

*currently no publicly available QPUs **continuous-variable quantum computing

qosf

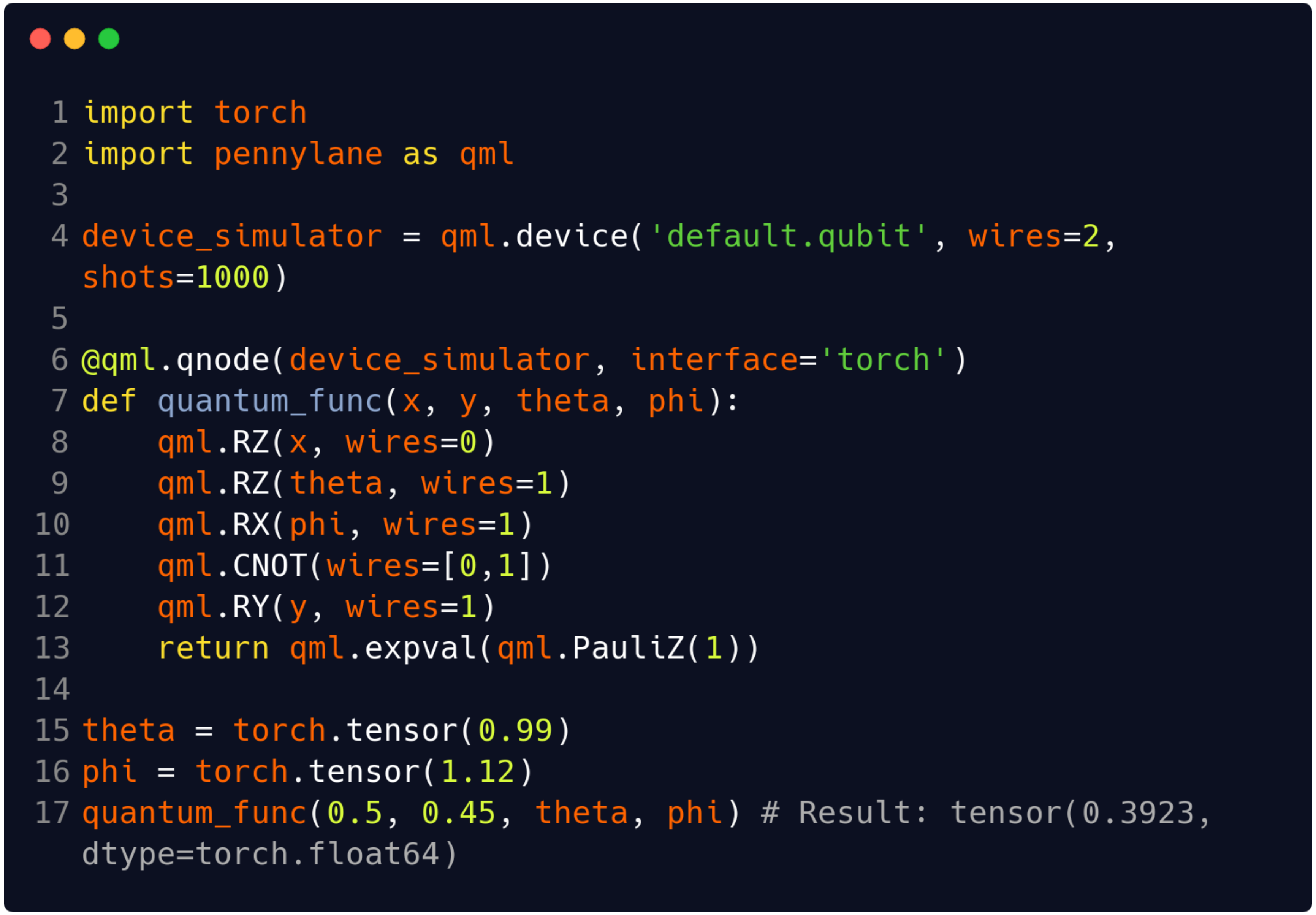
From QOS foundation

Defining a quantum node in PennyLane



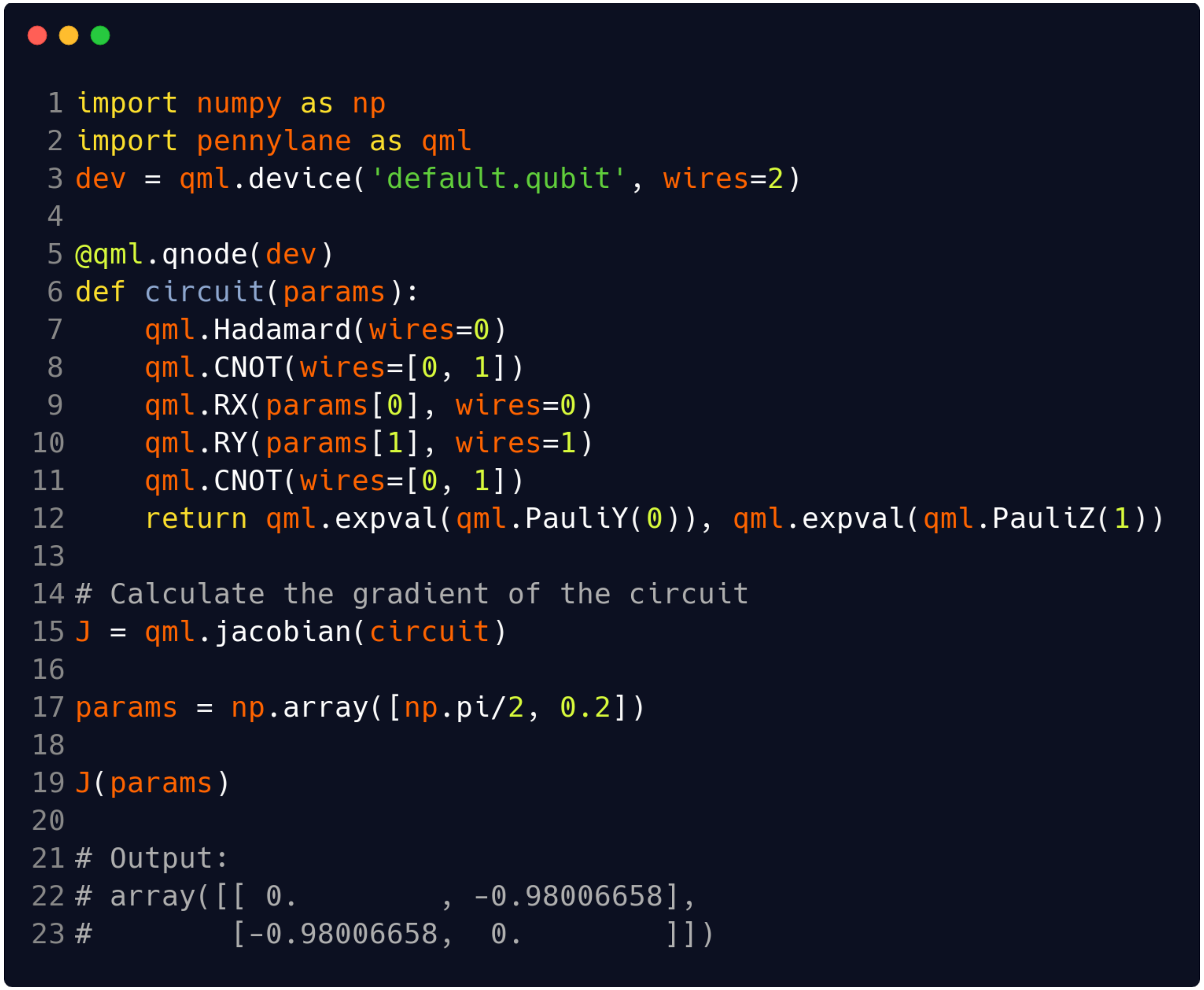
```python
import pennylane as qml

device_simulator = qml.device('default.qubit', wires=2, shots=1000)
device_hardware = qml.device('qiskit.ibmq', wires=2, backend='ibmq_16_melbourne')

@qml.qnode(device_simulator)
def quantum_func(x, y):
    qml.RZ(x, wires=0)
    qml.CNOT(wires=[0,1])
    qml.RY(y, wires=1)
    return qml.expval(qml.PauliZ(1))

quantum_func(0.5, 0.45) # Result: 0.9004471023526768
```

Interfacing with Pytorch



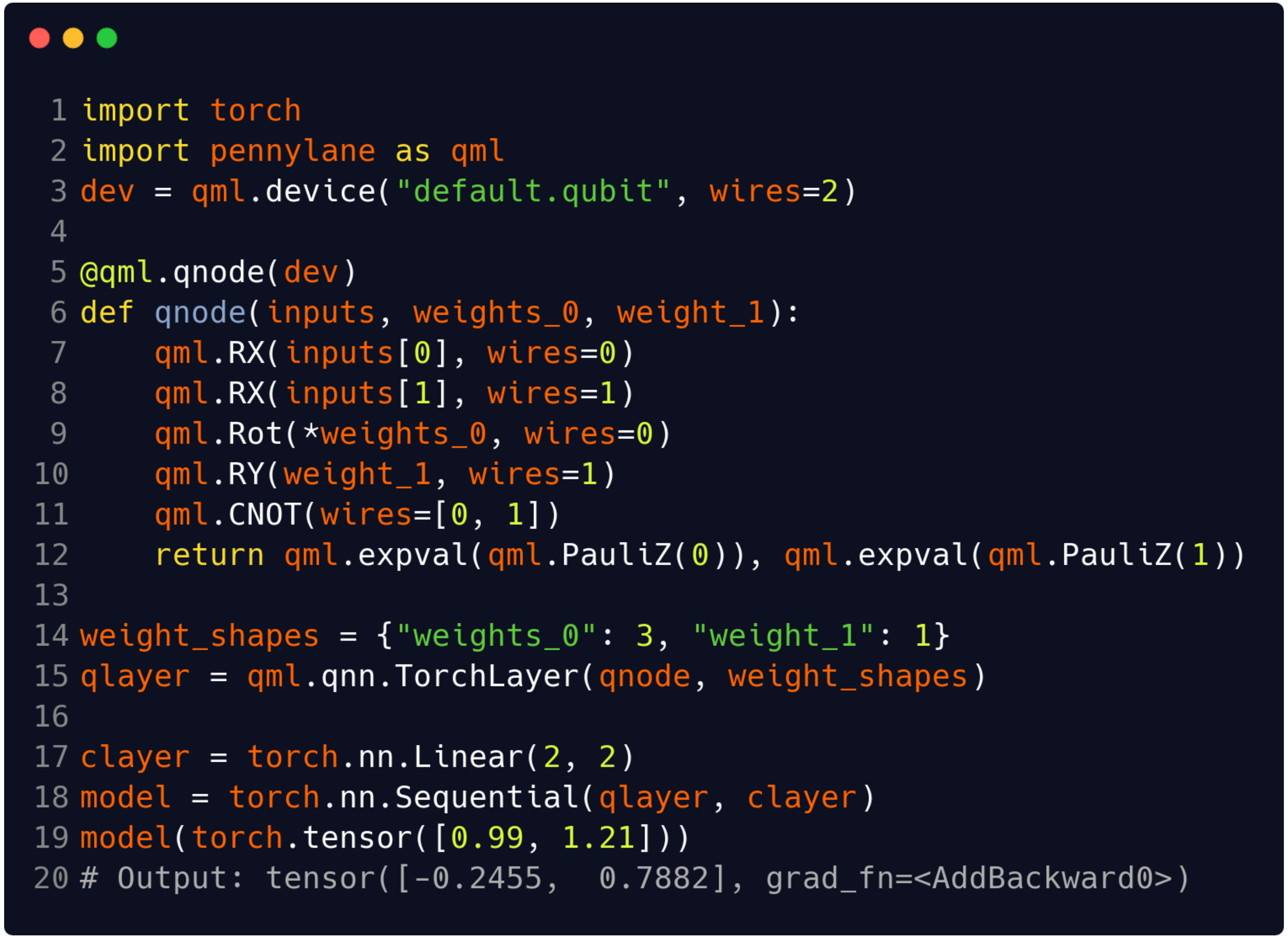
```python
import torch
import pennylane as qml

device_simulator = qml.device('default.qubit', wires=2,
shots=1000)

@qml.qnode(device_simulator, interface='torch')
def quantum_func(x, y, theta, phi):
    qml.RZ(x, wires=0)
    qml.RZ(theta, wires=1)
    qml.RX(phi, wires=1)
    qml.CNOT(wires=[0,1])
    qml.RY(y, wires=1)
    return qml.expval(qml.PauliZ(1))

theta = torch.tensor(0.99)
phi = torch.tensor(1.12)
quantum_func(0.5, 0.45, theta, phi) # Result: tensor(0.3923,
dtype=torch.float64)
```

Calculating Jacobians



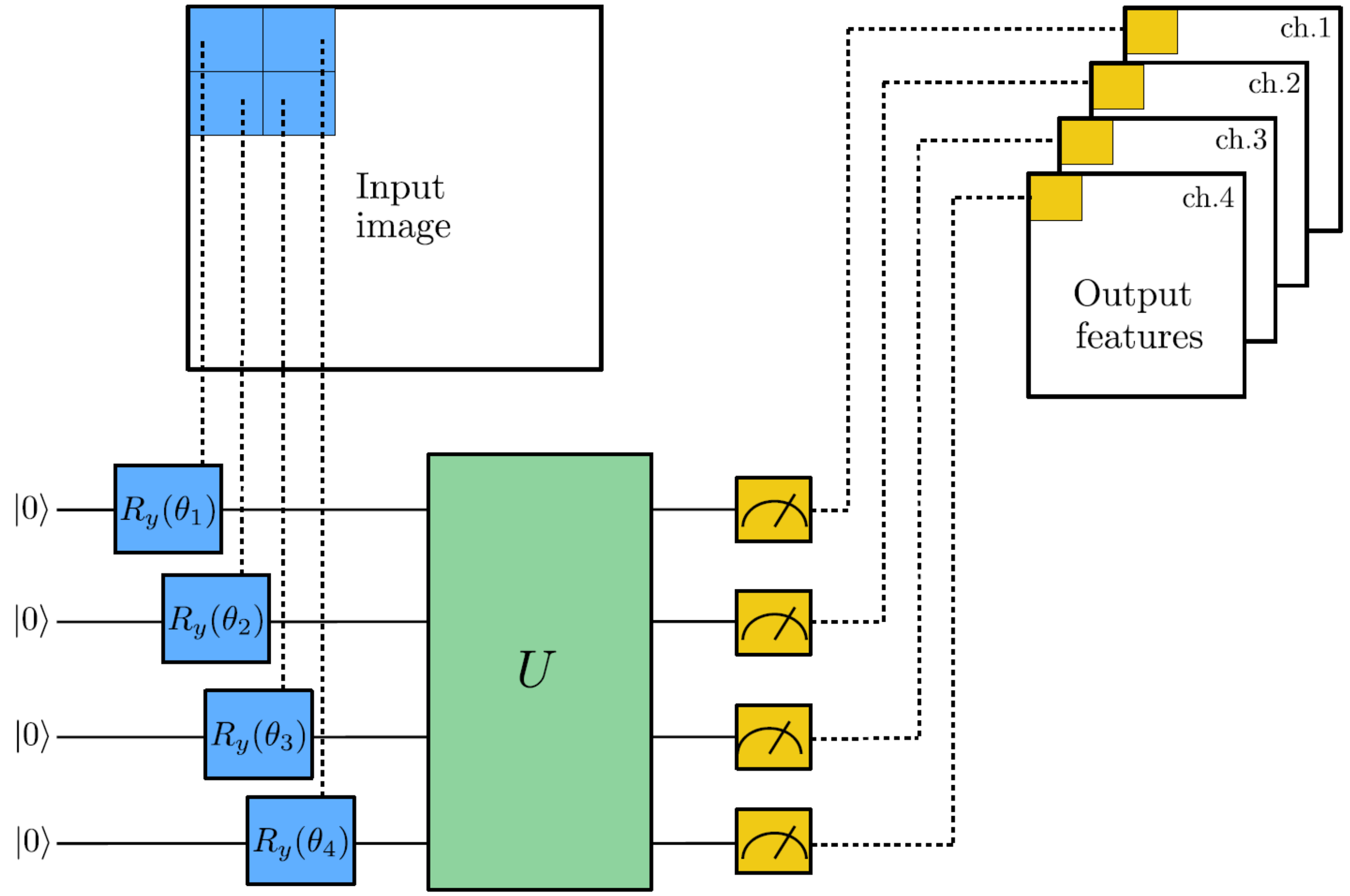
```python
import numpy as np
import pennylane as qml
dev = qml.device('default.qubit', wires=2)

@qml.qnode(dev)
def circuit(params):
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    qml.RX(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliY(0)), qml.expval(qml.PauliZ(1))

# Calculate the gradient of the circuit
J = qml.jacobian(circuit)

params = np.array([np.pi/2, 0.2])

J(params)

# Output:
# array([[ 0.        , -0.98006658],
#        [-0.98006658,  0.        ]])
```

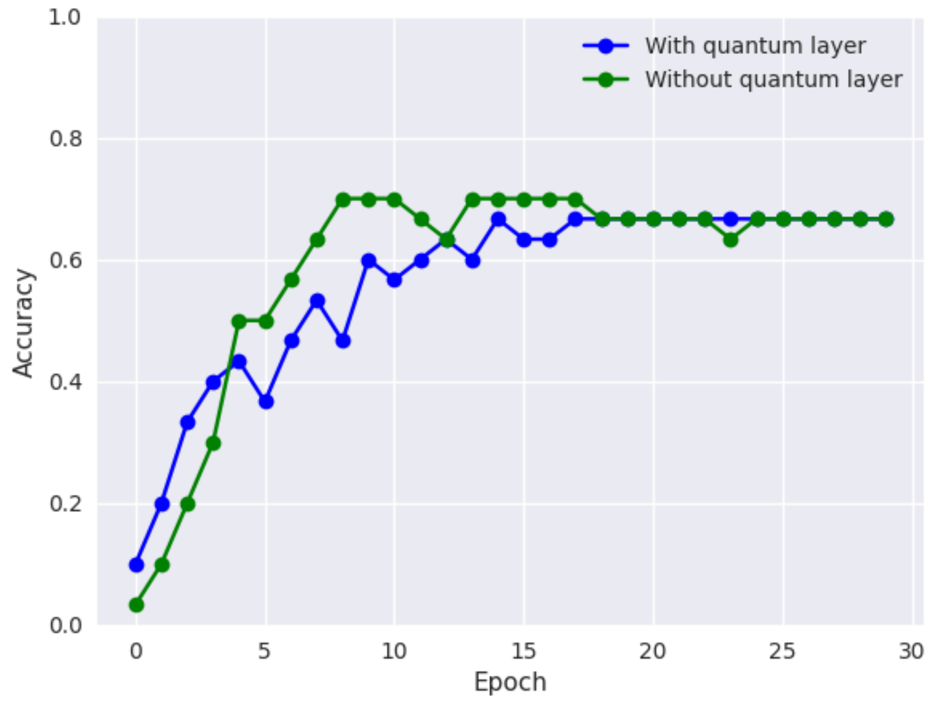
Using quantum layers in Pytorch models

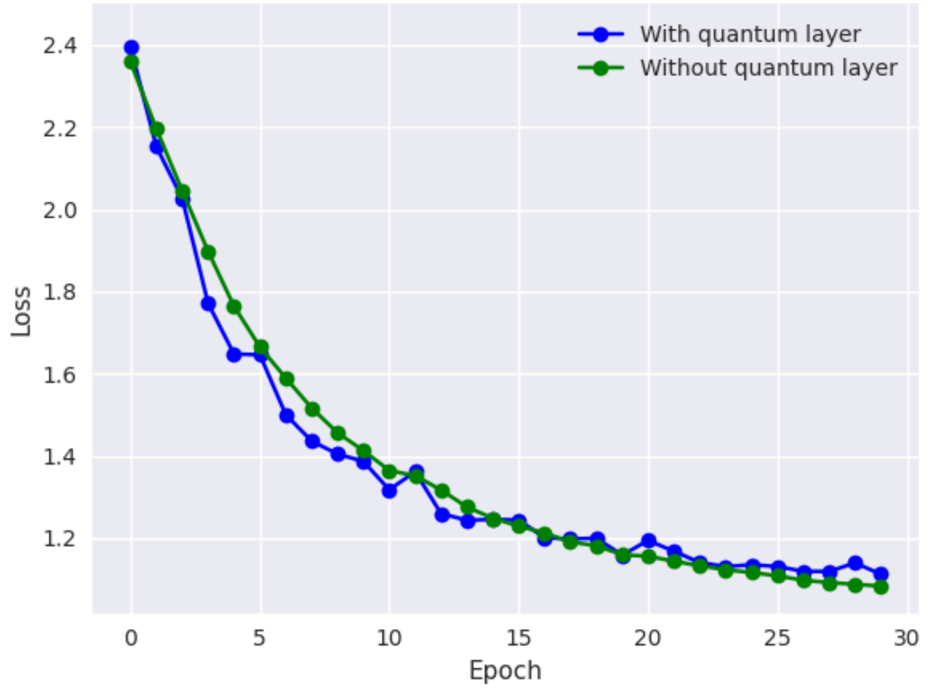


```python
import torch
import pennylane as qml
dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev)
def qnode(inputs, weights_0, weight_1):
    qml.RX(inputs[0], wires=0)
    qml.RX(inputs[1], wires=1)
    qml.Rot(*weights_0, wires=0)
    qml.RY(weight_1, wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(0)), qml.expval(qml.PauliZ(1))

weight_shapes = {"weights_0": 3, "weight_1": 1}
qlayer = qml.qnn.TorchLayer(qnode, weight_shapes)

clayer = torch.nn.Linear(2, 2)
model = torch.nn.Sequential(qlayer, clayer)
model(torch.tensor([0.99, 1.21]))
# Output: tensor([-0.2455,  0.7882], grad_fn=<AddBackward0>)
```

Example: Quantum Convolutional networks on the MNIST dataset







Challenges and open questions

Available hardware devices are small and noisy, simulation is slow and computationally heavy

Strategies for guessing the right ansatz circuit?

Can we prove that QML is better than ML?

The power of ML relies on nonlinearities between linear layers, buy quantum layers are always linear only (even if the Hilbert space is very high dimensional). Should we find a way to add nonlinearites?

Thank You for Your attention!