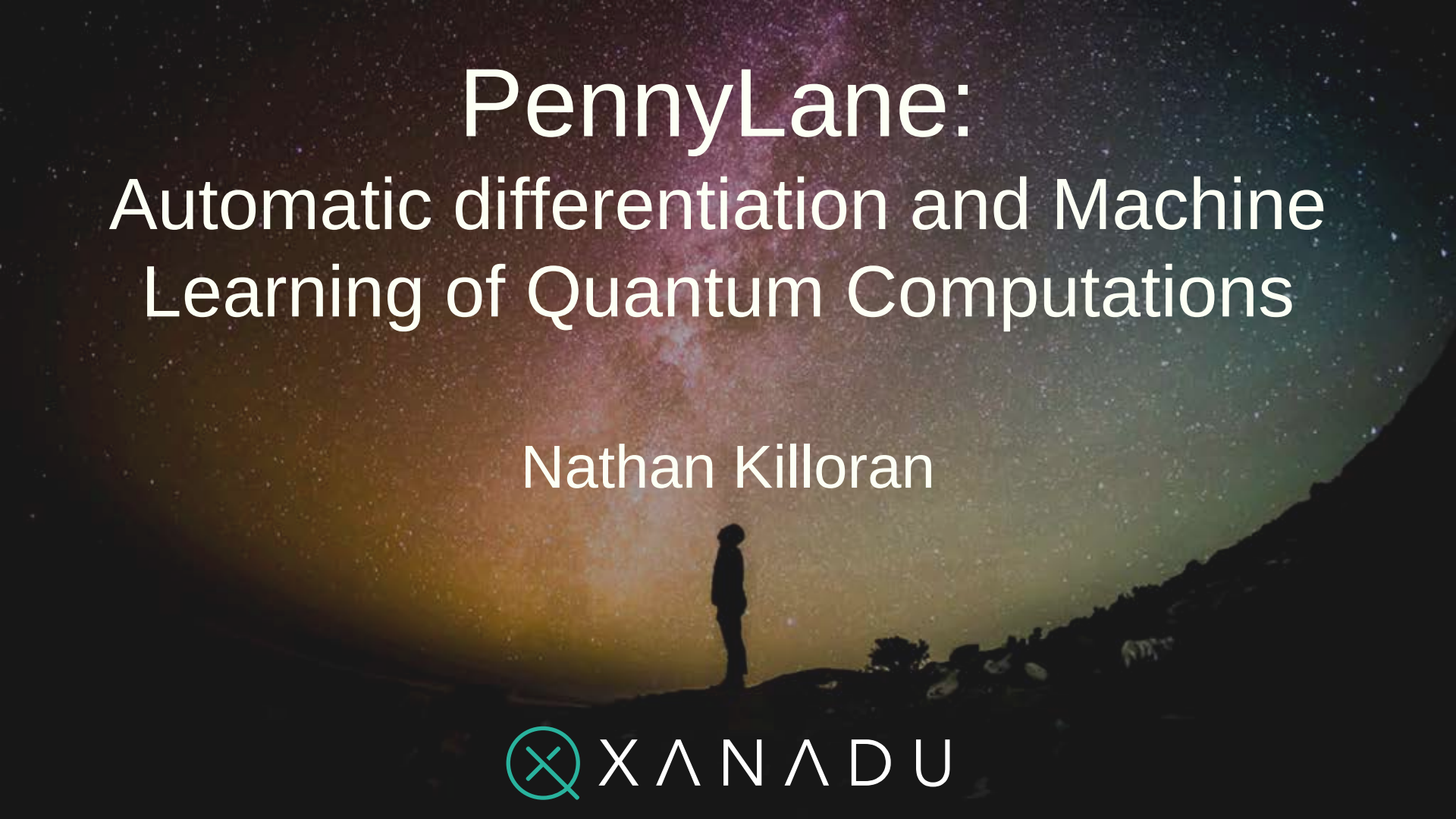


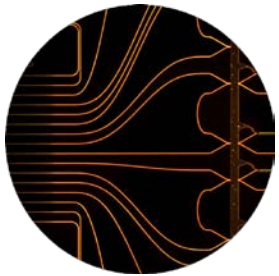
PennyLane:
Automatic differentiation and Machine Learning of Quantum Computations

Nathan Killoran

XANADU

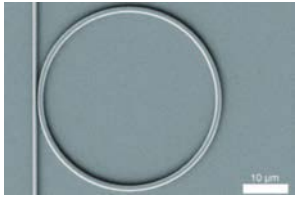
ABOUT XANADU

Hardware



What we're building:

- Photonic quantum technologies
- Integrated nanophotonics
- Continuous-variable (CV) model



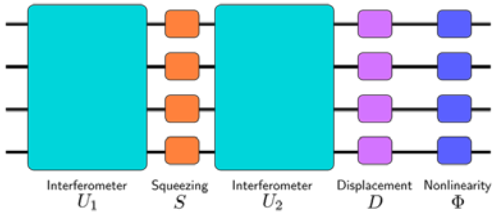
Building blocks:

- Waveguides
- Resonators
- Modulators
- Beam Splitters

Software

STRAWBERRY FIELDS

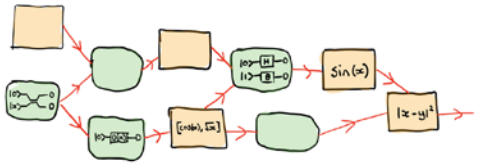
```
from strawberryfields.ops import *

prog = sf.Program(2)

with prog.context as q:
  Sgate(1.0)  | q[0]
```



- Only software platform dedicated to photonic quantum computing
- Three built-in simulators
- Hardware connectivity (currently private access only)

PENNYLANE

```
import pennylane as qml
import torch
from torch.autograd import Variable

qpu = qml.device('forest.qpu', device='Aspen-1-2Q-B')
```



- Dedicated platform for quantum machine learning
- Device agnostic: Xanadu, IBM, Rigetti, Microsoft
- Connects to standard ML libraries: PyTorch, TensorFlow

Xanadu

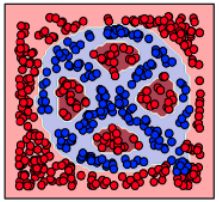
PhDs in:

Theoretical Physics | Experimental Physics | Computational Physics | Quantum Information | Quantum Optics | Physical Sciences | Mathematics | Electrical Engineering | Computer Engineering | Quantum Machine Learning

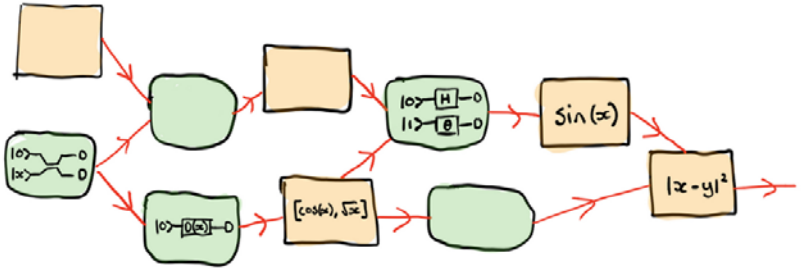
HARVARD UNIVERSITY · Massachusetts Institute of Technology · Stanford University · UNIVERSITY OF WATERLOO · UNIVERSITY OF TORONTO · PRINCETON UNIVERSITY

Outline

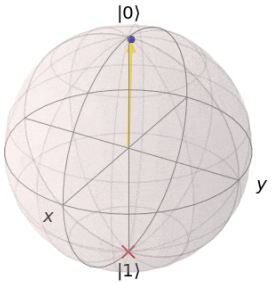
1. From machine learning to *quantum* machine learning



1. Training quantum circuits



1. PennyLane + examples

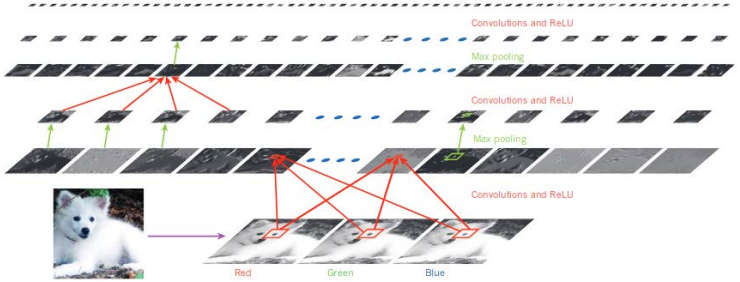



LESSONS FROM
DEEP LEARNING

Why is Deep Learning successful?

- Hardware advancements (GPUs, TPUs)

- Workhorse algorithms (backpropagation, stochastic gradient descent)



[*Nature* **521**, p 436–444 (2015)]

- Specialized, user-friendly software

PyTorch

TensorFlow

Why is Deep Learning successful?

- Hardware advancements

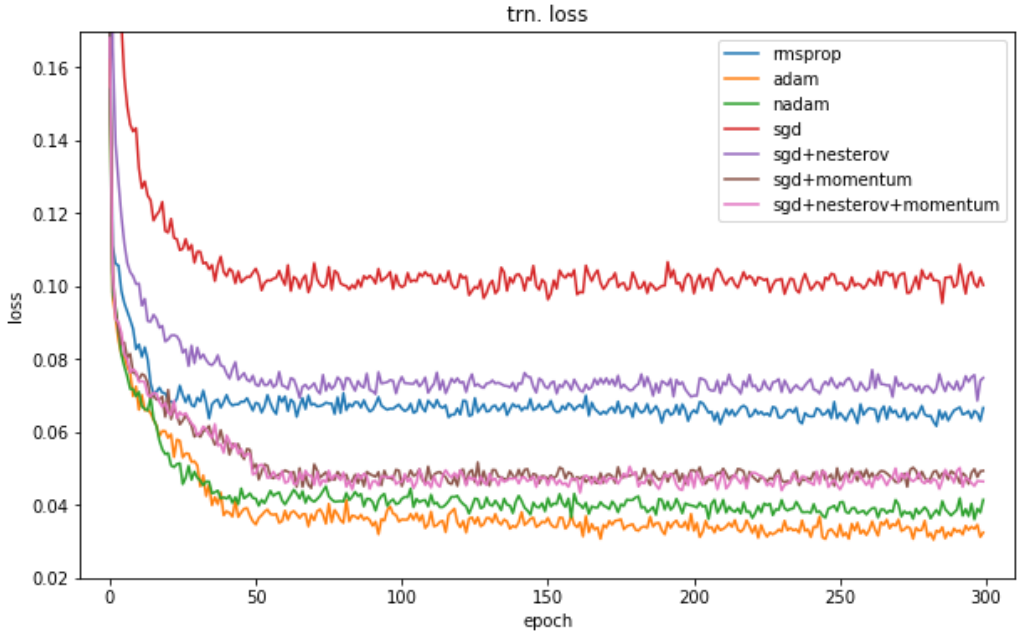
Large-scale Deep Unsupervised Learning using Graphics Processors

Rajat Raina RAJATR@CS.STANFORD.EDU
Anand Madhavan MANAND@STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Computer Science Department, Stanford University, Stanford CA 94305 USA

In this paper, we suggest massively parallel methods to help resolve these problems. We argue that modern graphics processors far surpass the computational capabilities of multicore CPUs, and have the potential to revolutionize the applicability of deep unsupervised learning methods. We develop general principles for massively parallelizing unsupervised learning tasks using graphics processors. We show that these principles can be applied to successfully scaling up learning algorithms for both DBNs and sparse coding. Our implementation of DBN learning is up to 70 times faster than a dual-core CPU implementation for large models. For example, we are able to reduce the time required to learn a four-layer DBN with 100 million free parameters from several weeks to around a single day. For sparse coding, we develop a simple, inherently parallel algorithm, that leads to a 5 to 15-fold speedup over previous methods.

Why is Deep Learning successful?

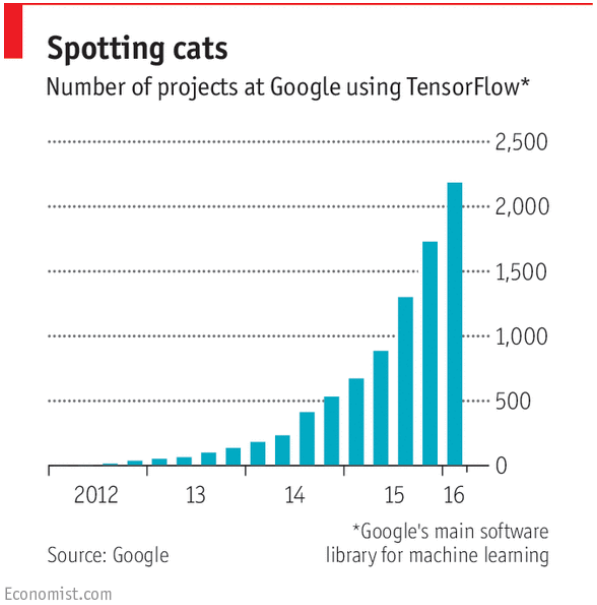
- Workhorse algorithms



[https://shaoanlu.files.wordpress.com/2017/05/trn_loss.png]

Why is Deep Learning successful?

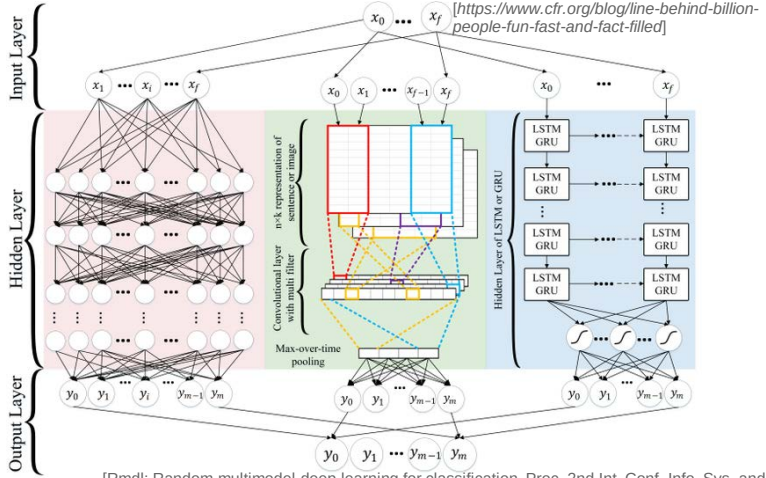
- Specialized, user-friendly software



Other takeaways from Deep Learning

- Crowdsourced innovation
- Shared code (including trained weights)
- Compose and reuse componer
- Rapid iteration
- New ways of thinking

[https://www.cfr.org/blog/line-behind-billion-people-fun-fast-and-fact-filled]

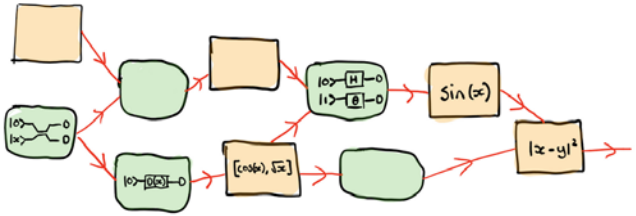


[Rmdl: Random multimodel deep learning for classification. Proc. 2nd Int. Conf. Info. Sys. and Data Min. (pp. 19-28). ACM.]

What can we leverage for Quantum?

- Hardware advancements (**QPUs**)

- Workhorse algorithms
  (**quantum-aware** backpropagation, stochastic gradient descent)



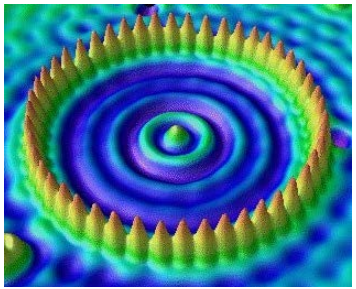
- Specialized, user-friendly software

PENNYLANE

QUANTUM MACHINE LEARNING

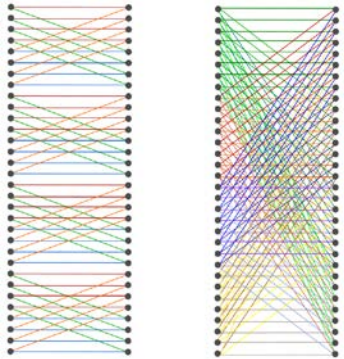
Quantum computers are good at:

Quantum physics

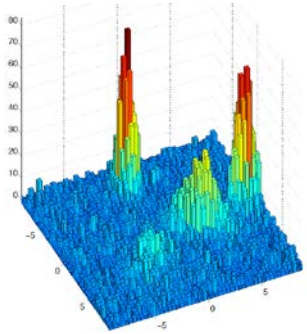


[*Science* 262, 218-220 (1993)]

Linear algebra

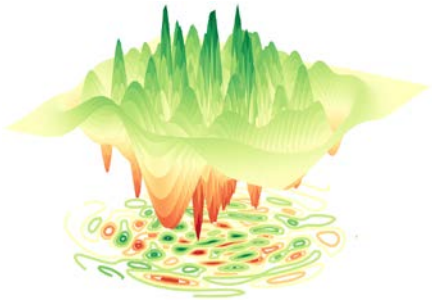


Sampling



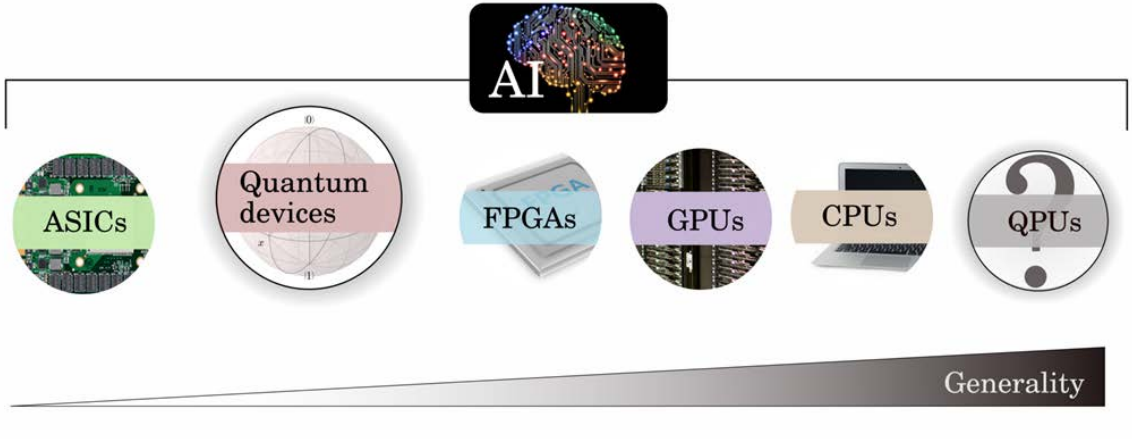
[*https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/46012/versions/1/screenshot.jpg*]

Optimization



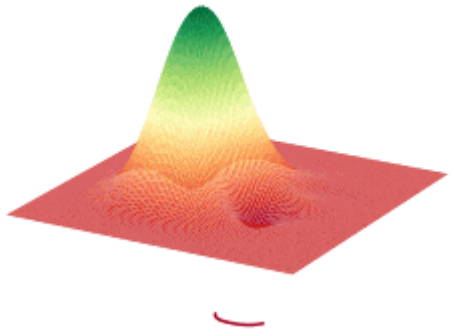
Quantum Machine Learning

- AI/ML already uses special-purpose processors: GPUs, TPUs, ASICs
- Quantum computers (QPUs) could be used as special-purpose AI accelerators
- May enable training of previously intractable models



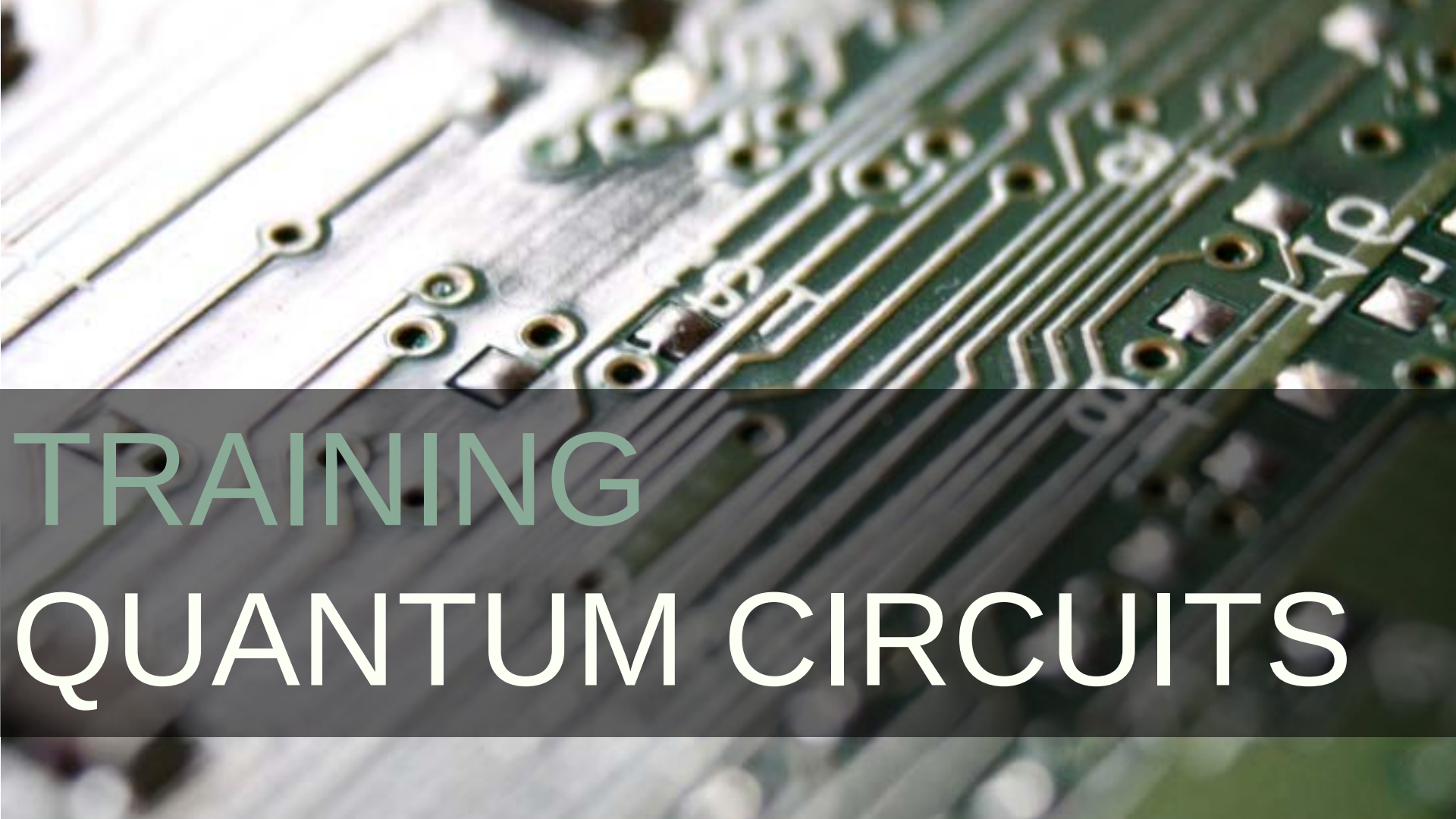
Machine Learning Quantum

- We can adapt machine learning tools to help understand and build quantum computers
- Use automatic differentiation to tune circuits
- Discover new algorithms and error-correction strategies
- Unearth new physics?



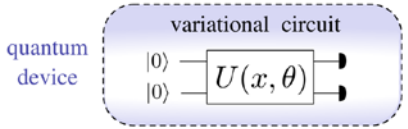
Vision for Quantum Machine Learning

- Everyone can easily explore and run QML algorithms
- Models are widely shared (or reimplemented by others)
- Reusable circuit blocks, embeddings, pre-/post-processing
- Exciting QML results *every single month*
- Feedback loop →new ideas we can’t predict today

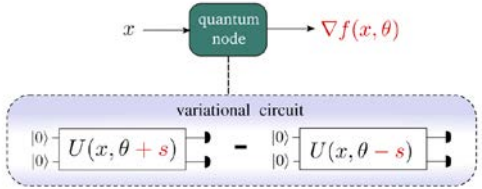

TRAINING
QUANTUM CIRCUITS

Key Concepts

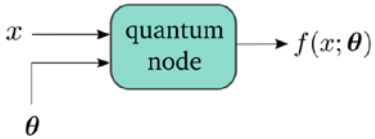
- Variational circuits



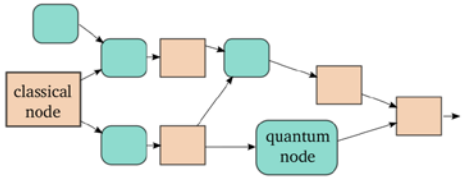
- Training quantum circuits



- Quantum nodes

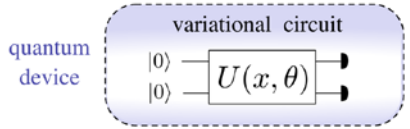


- Hybrid computation

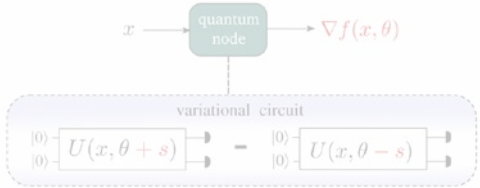


Key Concepts

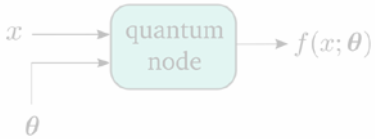
- Variational circuits



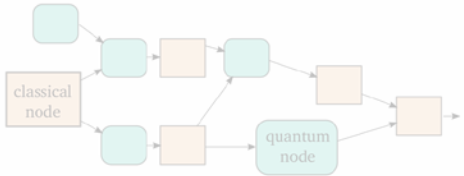
- Training quantum circuits



- Quantum nodes



- Hybrid computation

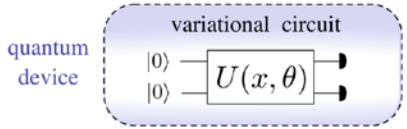


Variational Circuits

- Main QML method for near-term (NISQ) devices
- Ideas earlier began in simpler specialized forms:
    - Variational Quantum Eigensolver (VQE)
    - Quantum Alternating Operator Ansatz (QAOA)
- Natural extension to other circuits and tasks (e.g., quantum classifier)
- Nowadays many, many proposals

Variational Circuits

- Basic structure of a variational circuit:

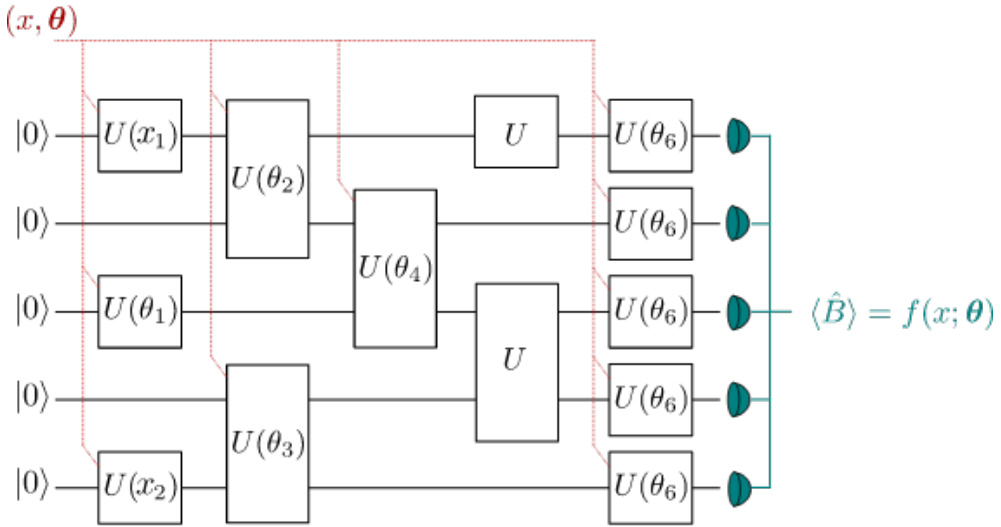


=

I. Preparation of a fixed initial state

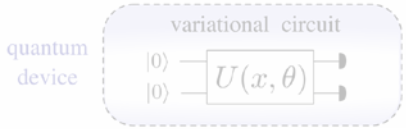
II. Quantum circuit; input data and free parameters are used as gate arguments

III. Measurement of fixed observable

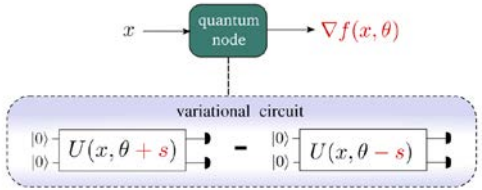


Key Concepts

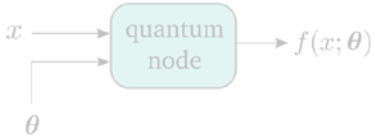
- Variational circuits



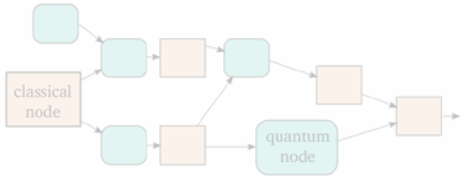
- Training quantum circuits



- Quantum nodes



- Hybrid computation

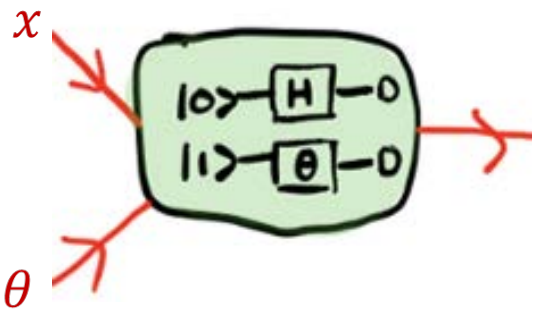


Differentiable computation

- Deep learning began with neural networks, but nowadays the models are *much richer*
    - Attention mechanisms, external memory, neural differential equations, etc.
- Key insight: computation is end-to-end *differentiable*
- Program only the structure of computation (*“build the model”*)
- Use optimization (e.g., gradient descent) to fine-tune parameters (“*train the model*”)

Differentiable computation

- Quantum computing *is also differentiable*
- Gates are controlled by parameters (e.g., rotation angle, squeezing amplitude)
- Expectation values depend smoothly on gate parameters
- Can we *train* quantum circuits?



$$f(x,\theta) = \langle \hat{B} \rangle$$
$$= \langle U^\dagger(\theta)\hat{B}\,U(\theta) \rangle$$

How to train quantum circuits?

Two approaches:

I. *Simulator-based*

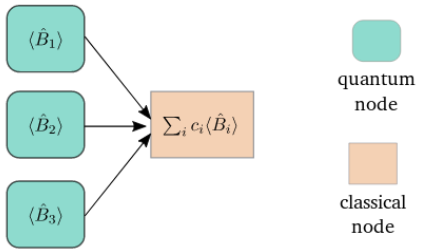
- Build simulation inside existing classical library
- Can leverage existing optimization & ML tools
- Great for small circuits, but *not scalable*

STRAWBERRY FIELDS

QuantumFlow

II. *Hardware-based*

- No access to quantum information; only have measurements & expectation values
- Needs to work as hardware becomes more powerful and *cannot be simulated*



Gradients of quantum circuit ∇f

- Training strategy: use gradient descent algorithms
- *Need to compute gradients* of variational circuit outputs with respect to their gate parameters
- How can we compute gradients of quantum circuits when even simulating their output is *classically intractable*?

The ‘parameter shift’ trick

$$f(\theta) = \sin\theta \Rightarrow \partial_\theta f(\theta) = \cos\theta$$

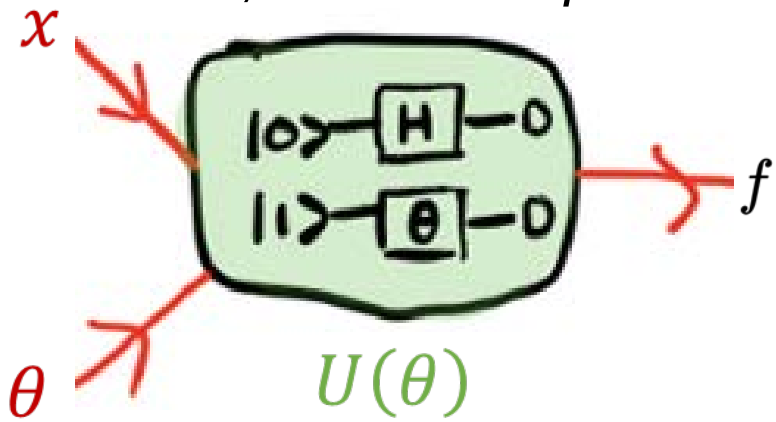
$$\cos\theta = \frac{\sin(\theta+\pi/4)-\sin(\theta-\pi/4)}{\sqrt{2}}$$

$$\partial_\theta f = \frac{1}{\sqrt{2}}\left[f(\theta+\pi/4) - f(\theta-\pi/4)\right]$$

Parameter shift method

- Main insight: Use the same quantum hardware to evaluate its own gradients.

- The gradient of a circuit can be computed by the *same circuit*, with *shifted parameters*



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

The parameters ***c*** and ***s*** depend on the specific function. Crucially, ***s*** is large.

This is not finite difference!

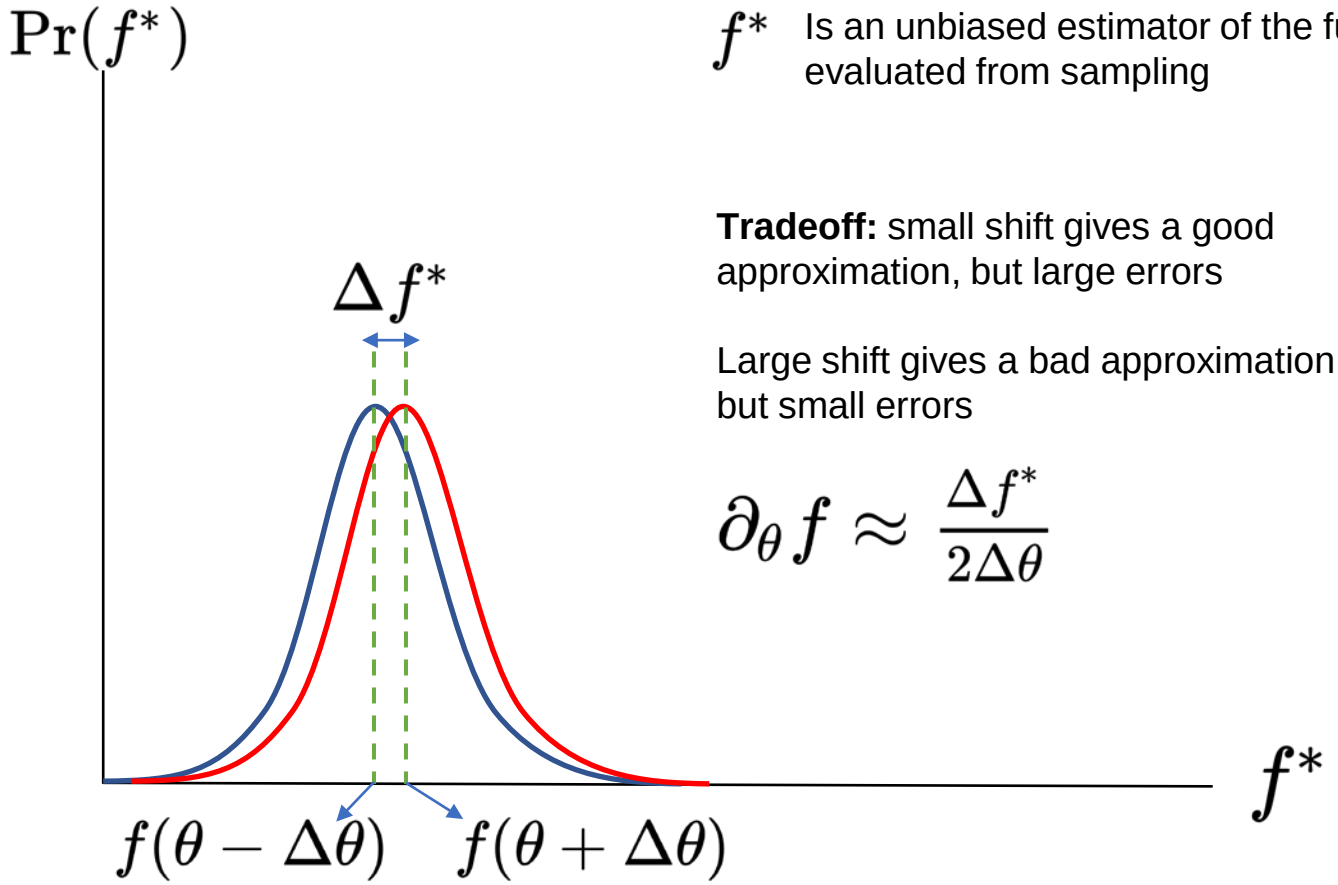
$$\partial_\theta f(\theta) = c[f(\theta + s) - f(\theta - s)]$$

- Exact
- Shift is specific to each gate – in general, we use a **large** shift

$$\partial_\theta f(\theta) = \frac{f(\theta+\Delta\theta) - f(\theta-\Delta\theta)}{2\Delta\theta}$$

- Only an **approximation**
- Requires that shift is small
- Known to give rise to numerical issues
- For NISQ devices, small shifts could lead to the resulting difference being swamped by noise

Finite difference



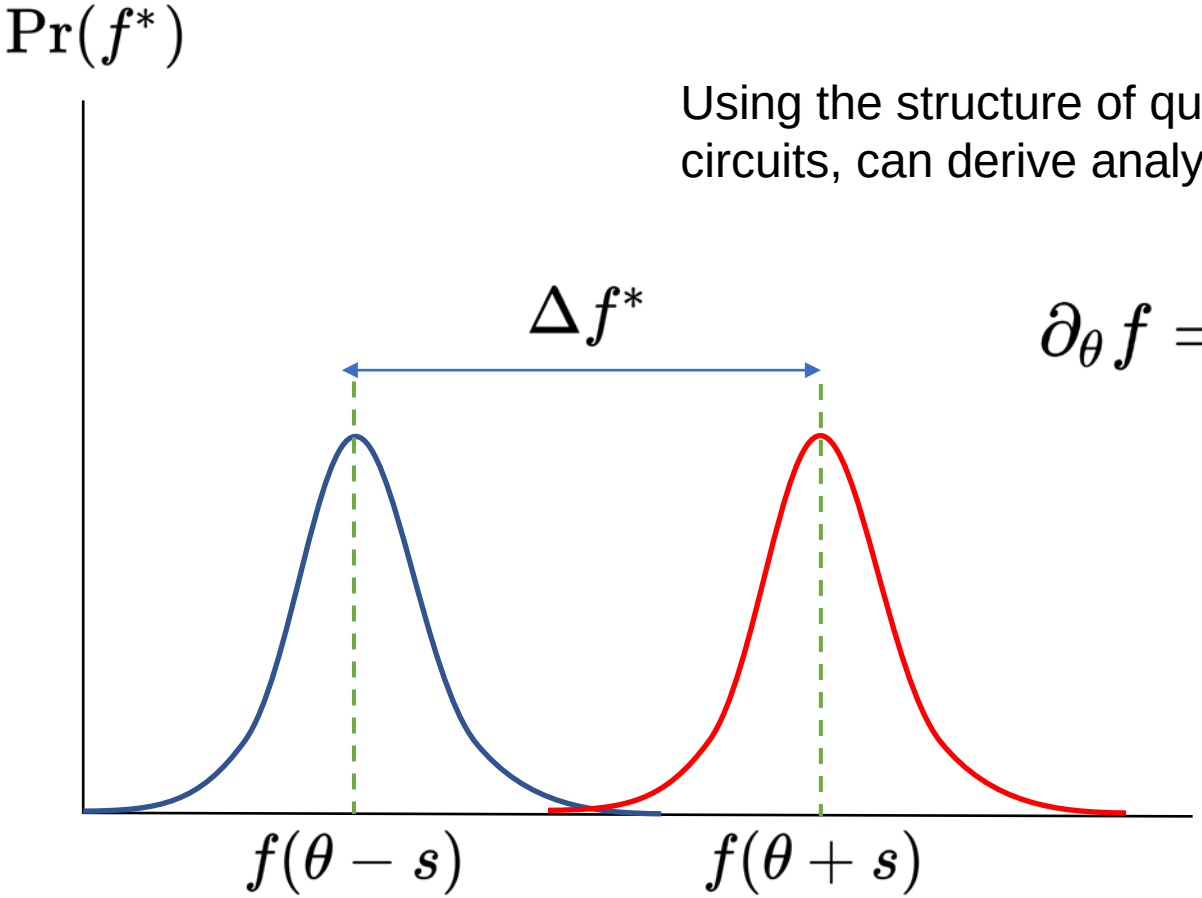
f^* Is an unbiased estimator of the function, evaluated from sampling

Tradeoff: small shift gives a good approximation, but large errors

Large shift gives a bad approximation, but small errors

$$\partial_\theta f \approx \frac{\Delta f^*}{2\Delta\theta}$$

Parameter shift method



Using the structure of quantum circuits, can derive analytic recipes

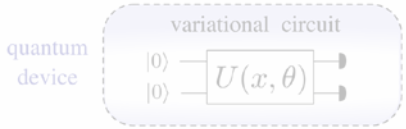
$$\partial_\theta f = c\Delta f^*$$

Gradient recipes for photonic QC

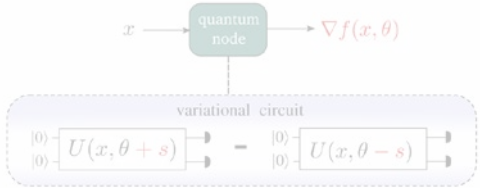
Gate $\mathcal{G}$	Heisenberg representation $M^{\mathcal{G}}$	Partial derivatives of $M^{\mathcal{G}}$
Phase rotation $R(\phi)$	$M^R(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix}$	$\partial_\phi\, M^R(\phi) = \frac{1}{2}\big(M^R(\phi+\frac{\pi}{2}) - M^R(\phi-\frac{\pi}{2})\big)$
Displacement $D(r,\phi)$	$M^D(r,\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 2r\cos\phi & 1 & 0 \\ 2r\sin\phi & 0 & 1 \end{pmatrix}$	$\partial_r M^D(r,\phi) = \frac{1}{2s}\big(M^D(r+s,\phi) - M^D(r-s,\phi)\big),\ s\in\mathbb{R}$ $\partial_\phi M^D(r,\phi) = \frac{1}{2}\big(M^D(r,\phi+\frac{\pi}{2}) - M^D(r,\phi-\frac{\pi}{2})\big)$
Squeezing[a] $S(r)$	$M^S(r) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \mathrm{e}^{-r} & 0 \\ 0 & 0 & \mathrm{e}^{r} \end{pmatrix}$	$\partial_r M^S(r) = \frac{1}{2\sinh(s)}\big(M^S(r+s) - M^S(r-s)\big),\ s\in\mathbb{R}$
Beamsplitter $B(\theta,\phi)$	$M^B(\theta,\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & -\alpha & -\beta \\ 0 & 0 & \cos\theta & \beta & -\alpha \\ 0 & \alpha & -\beta & \cos\theta & 0 \\ 0 & \beta & \alpha & 0 & \cos\theta \end{pmatrix}$	$\partial_\theta M^B(\theta,\phi) = \frac{1}{2}\big(M^B(\theta+\frac{\pi}{2},\phi) - M^B(\theta-\frac{\pi}{2},\phi)\big)$ $\partial_\phi M^B(\theta,\phi) = \frac{1}{2}\big(M^B(\theta,\phi+\frac{\pi}{2}) - M^B(\theta,\phi-\frac{\pi}{2})\big)$ $\alpha = \cos\phi\sin\theta,\quad \beta = \sin\phi\sin\theta$

Key Concepts

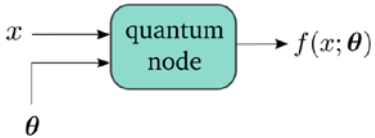
- Variational circuits



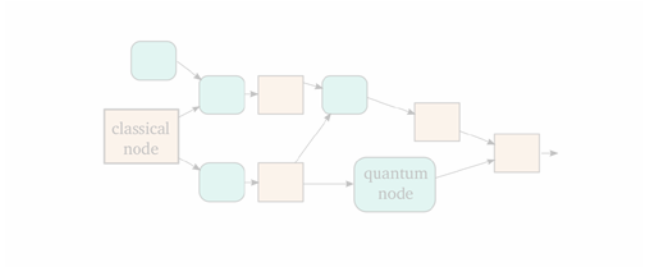
- Training quantum circuits



- Quantum nodes

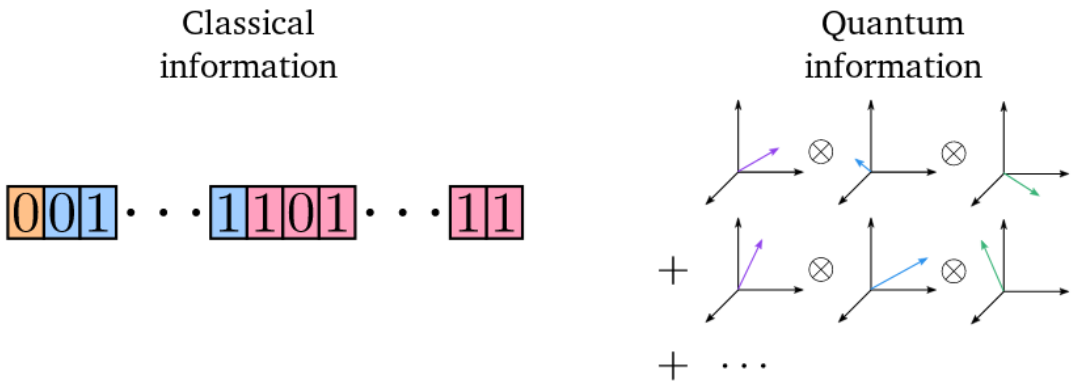


- Hybrid computation



Quantum Nodes

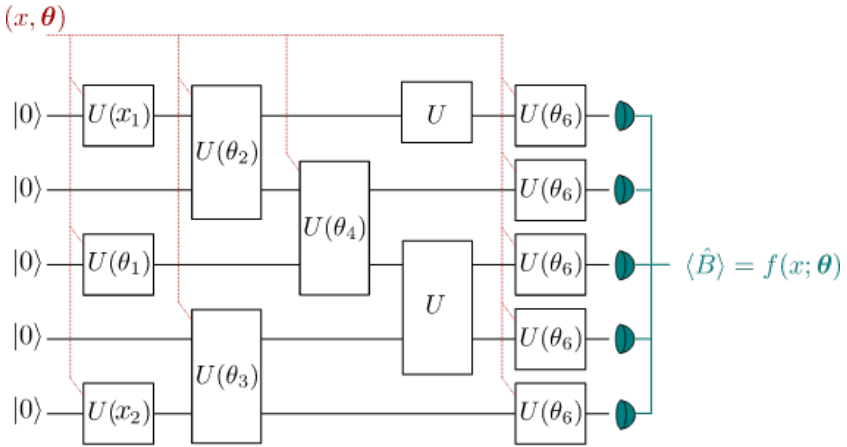
- Classical and quantum information are distinct
- A classical processor can’t access quantum information inside a circuit



Quantum Nodes

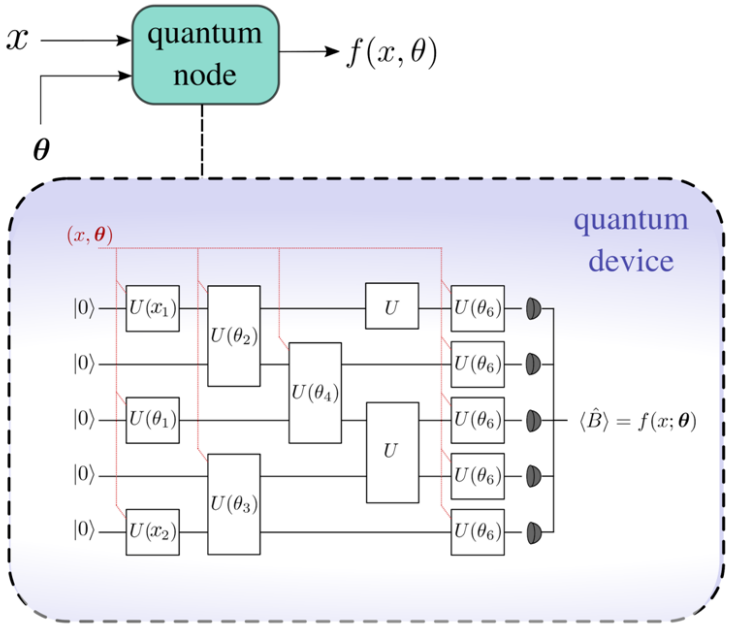
- However, a variational circuit:
  - takes classical information as input (gate parameters)
  - produces classical information as output (expectation values)

- Transforms *classical data* to *classical data*
  - Function itself may be classically intractable



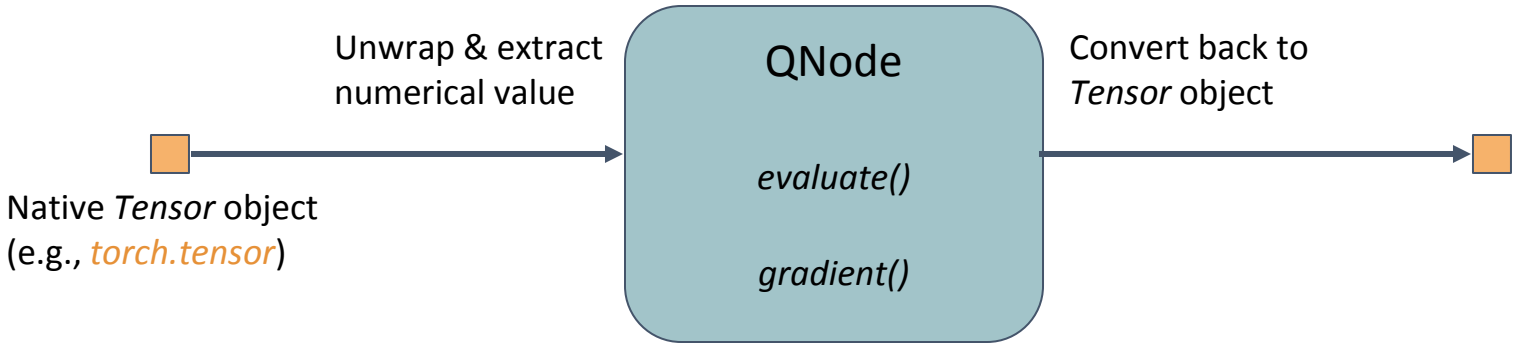
Quantum Nodes

- QNode: common interface for quantum and classical devices
  - Classical device sees a callable parameterized function
  - Quantum device sees fine-grained circuit details



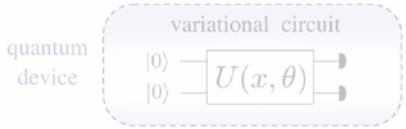
Quantum Nodes

- QNode enables interface between quantum computers and classical ML libraries
  - NumPy Autograd
  - TensorFlow
  - PyTorch

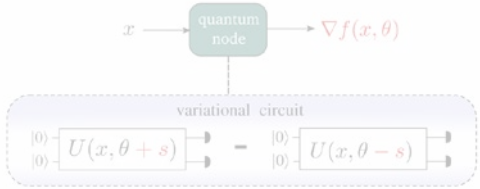


Key Concepts

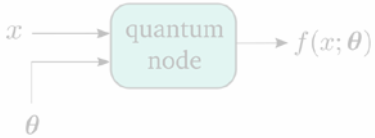
- Variational circuits



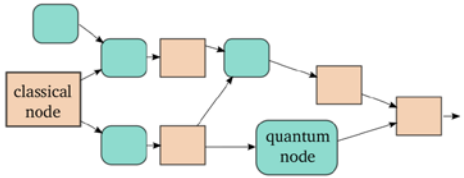
- Training quantum circuits



- Quantum nodes

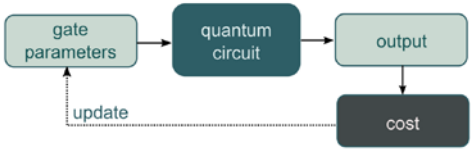


- Hybrid computation

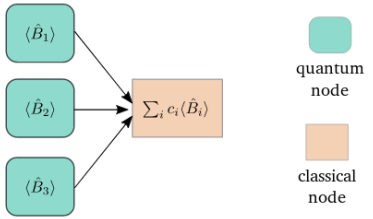


Hybrid Computation

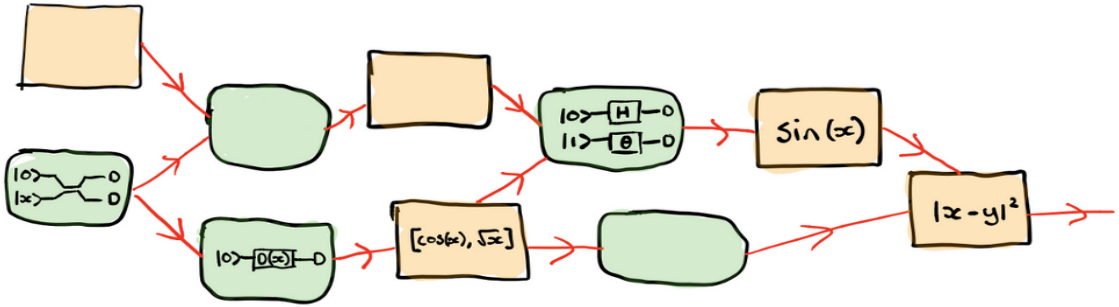
- Quantum circuit can be just one step of a larger computation
  - Classical optimization loop



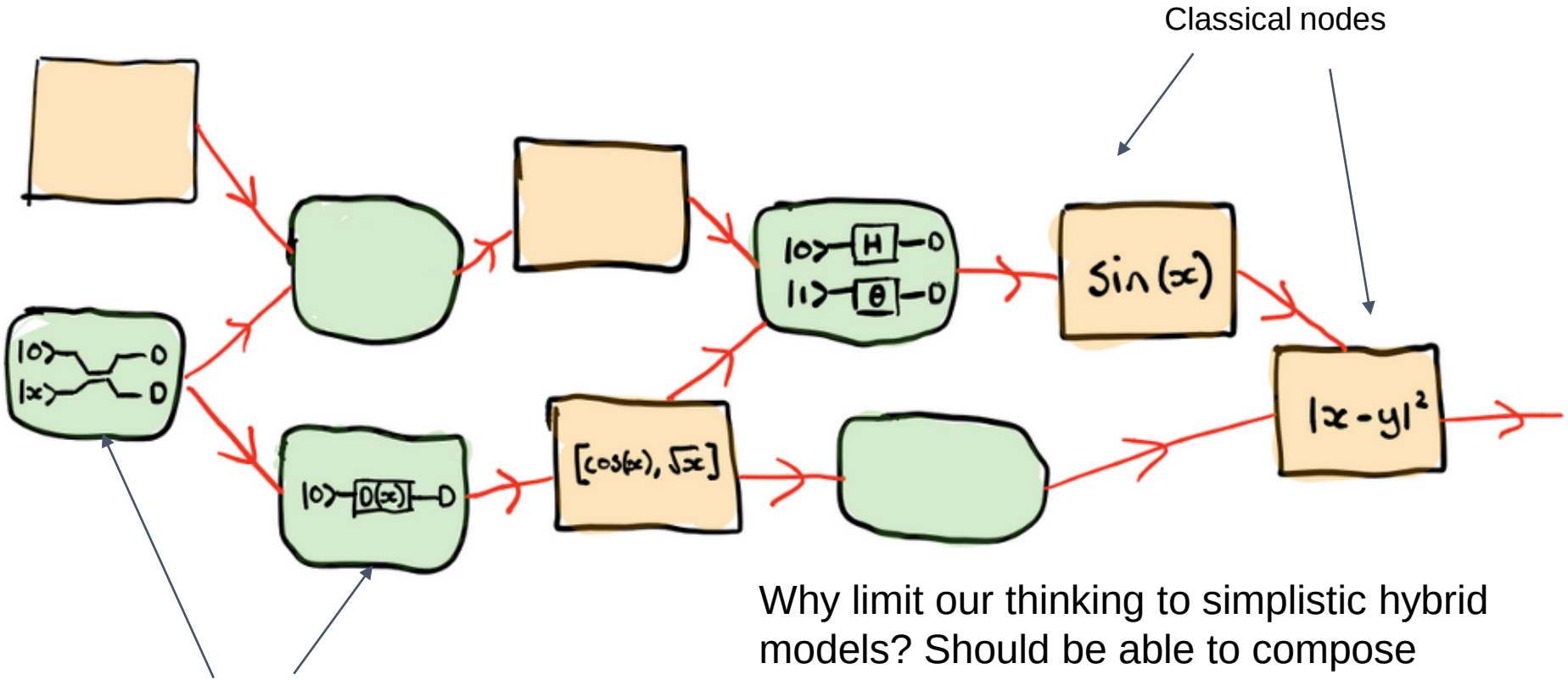
  - Pre-/post-process quantum circuit outputs



  - Arbitrarily structured hybrid computations



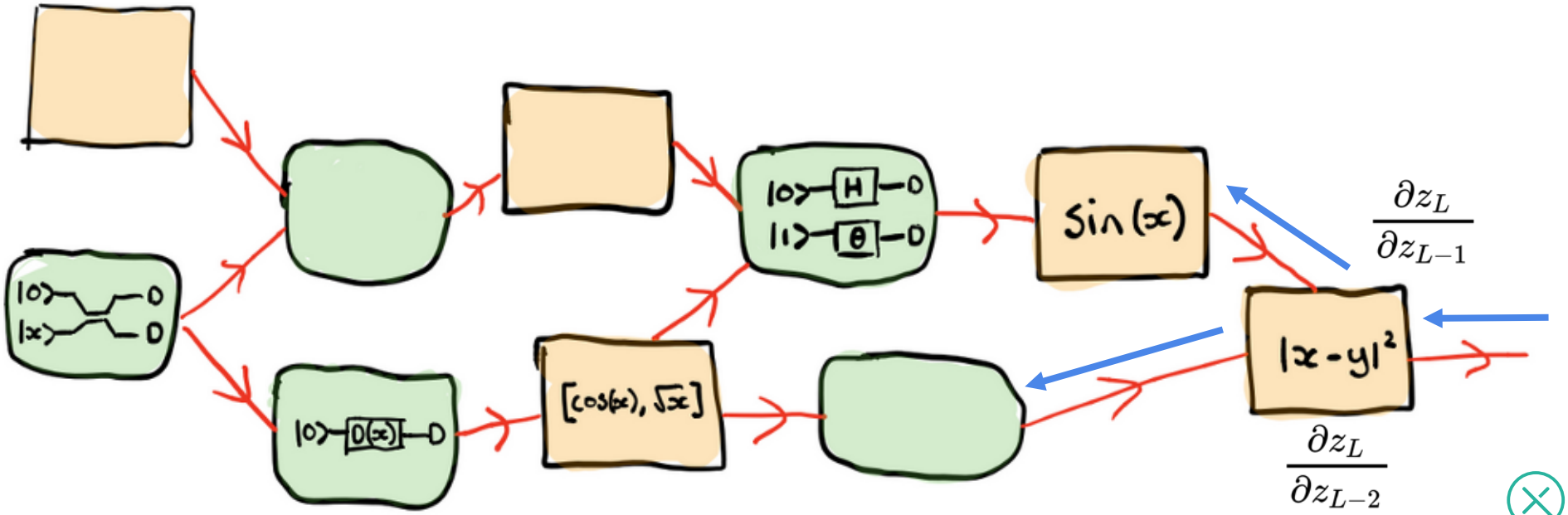
Hybrid Computation



Why limit our thinking to simplistic hybrid models? Should be able to compose arbitrarily

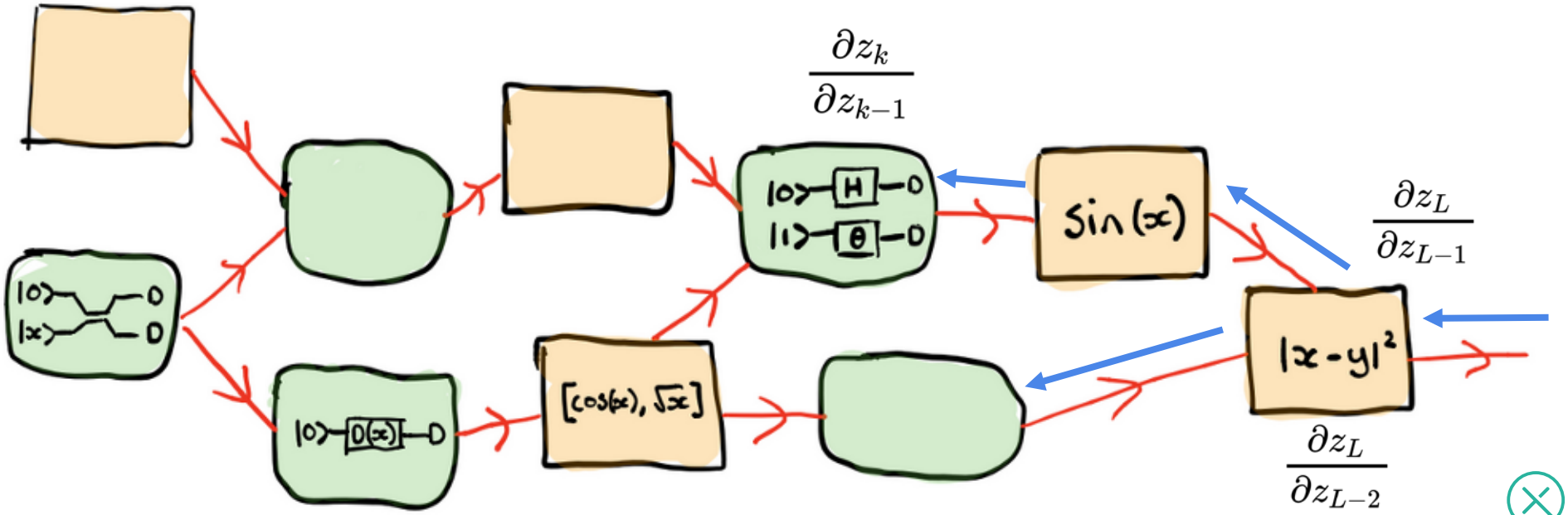
Compatibility with backpropagation

- Backprop steps backwards through computational graph
- Computes gradient at each step and aggregates (chain rule)



Compatibility with backpropagation

- When we hit a quantum node, use parameter shift method
- Can’t backprop *inside* a QNode, but can backprop *through* it
- End-to-end differentiable

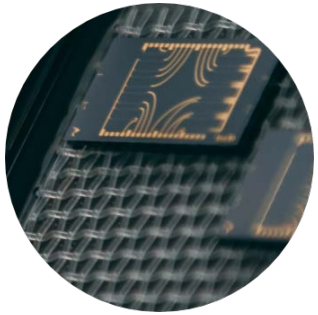


Hybrid Computation

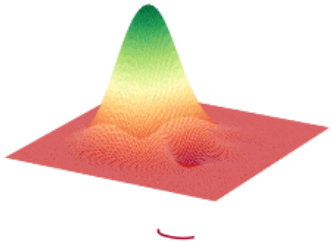
- Deep learning began with neural networks, but nowadays the models are *much richer*
- Similarly, quantum machine learning can be much richer than current models
- All the ingredients are now available. How can we get started?


PENNYLANE

Philosophy



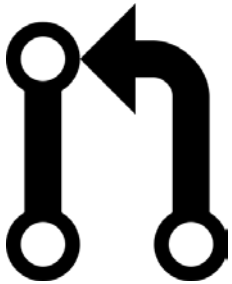
Champion QML



Push new ideas

Best-practices

Python interface

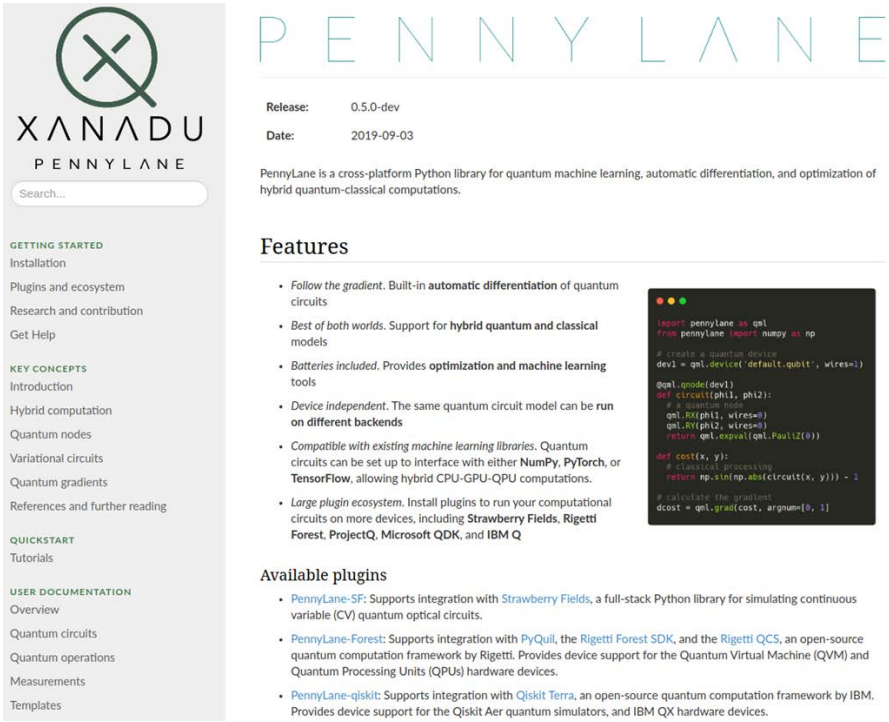


Open-source

PennyLane

“The TensorFlow of quantum

- Train a quantum computer the same way as a neural network
- Designed to scale as quantum computers grow in power
- Compatible with multiple quantum platforms



https://github.com/XanaduAI/pennylane

https://pennylane.readthedocs.io

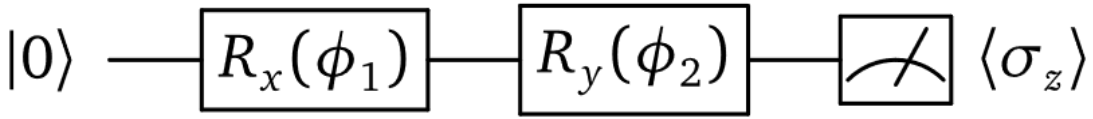
https://pennylane.ai

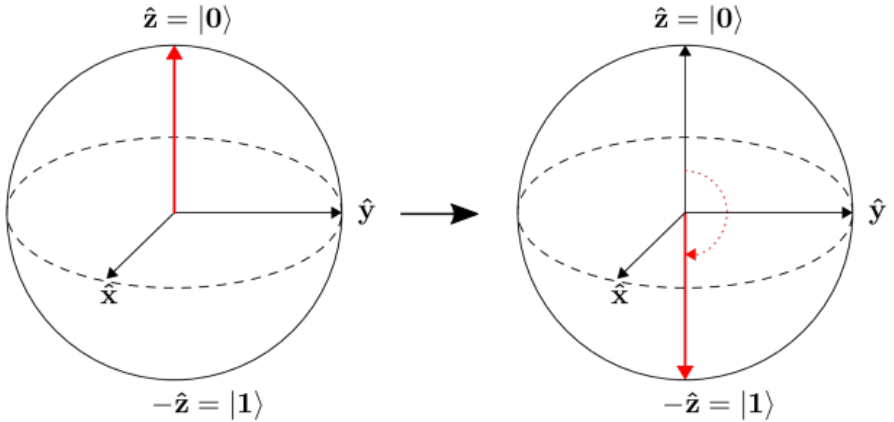
Comes with a growing plugin ecosystem, supporting a wide range of quantum hardware and classical software

Qubit Rotation Tutorial

The PennyLane version of “hello world”.

Goal is to build a single-qubit circuit that rotates a qubit to a desired pure state.





Import libraries

```
import pennylane as qml
from pennylane import numpy as np
```

Important: NumPy must be imported from PennyLane to ensure compatibility with automatic differentiation.

Basically, this allows you to use NumPy as usual.

You can also use **PyTorch** or **TensorFlow** instead of NumPy.

Create device

Device name

Wires are subsystems (because they are represented as wires in a circuit diagram)

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

Any computational object that can apply quantum operations and return a measurement result is called a quantum **device**.

In PennyLane, a device could be a hardware device (such as the Rigetti QPU, via the PennyLane-Forest plugin), or a software simulator (such as Strawberry Fields, via the PennyLane-SF plugin).

Create a qnode

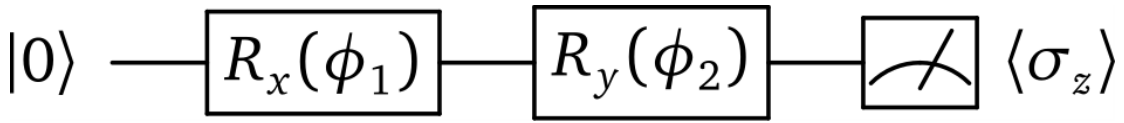
Gate parameters

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

Python decorator

Wire the gate acts on

Expectation value output



QNodes are quantum functions, described by a quantum circuit. They are bound to a particular quantum device, which is used to evaluate expectation values of this circuit.

Cost function

```
def cost(params):
    expval = circuit(params)
    return np.abs(expval - (-1)) ** 2
```

Can define any differentiable NumPy function from the output of the qnode.

In this case, we want the expectation value of the circuit to be -1.

Initial parameters

```
params = np.random.normal(size=(2,))
circuit(params)
```

Dimension of params

We can evaluate the circuit at any value of params

In this case, there are two rotation angles, which we initialize randomly from the standard normal distribution.

When any qnode is evaluated, PennyLane calls the device itself to obtain the result.

Optimize the circuit

```
opt = qml.AdamOptimizer()
steps = 300

for i in range(steps):
    params = opt.step(cost, params)

print('Circuit output:', circuit(params))
print('Final parameters:', params)
```

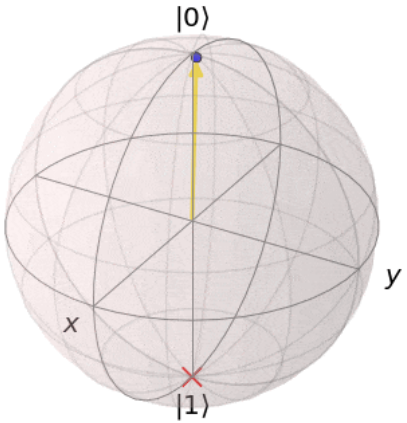
Improves parameters by gradient descent

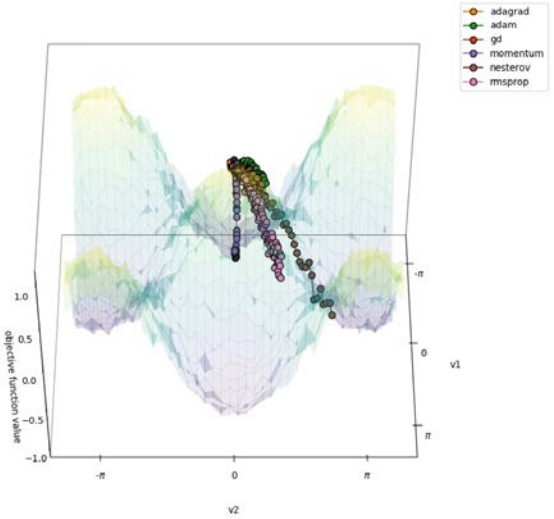
We can choose from a wide variety of gradient-based optimizers. In this case we select the Adam optimizer.

The parameters are optimized one step at a time for a total of 300 steps, then printed.

Putting it all together

```
import pennylane as qml
from pennylane import numpy as np

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

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

def cost(params):
  expval = circuit(params)
  return np.abs(expval - (-1)) ** 2

params = np.random.normal(size=(2,))

opt = qml.AdamOptimizer()
steps = 300

for i in range(steps):
    params = opt.step(cost, params)

print('Circuit output:', circuit(params))
print('Final parameters:', params)
```





Classical interfaces

```python
import pennylane as qml
from pennylane import numpy as np

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

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

def cost(params):
  expval = circuit(params)
  return np.abs(expval - (-1)) ** 2

params = np.random.normal(size=(2,))

opt = qml.AdamOptimizer()
steps = 300

for i in range(steps):
  params = opt.step(cost, params)
```

NumPy

```python
import pennylane as qml
import torch
from torch.autograd import Variable

qpu = qml.device('forest.qpu', device='Aspen-1-2Q-B')

@qml.qnode(dev, interface='torch')
def circuit(phi1, phi2):
  qml.RX(phi1, wires=0)
  qml.RY(phi2, wires=0)
  return qml.expval(qml.PauliZ(0))

def cost(phi1, phi2):
  expval = circuit(phi1, phi2)
  return torch.abs(expval - (-1)) ** 2

phi1 = Variable(torch.tensor(1.), requires_grad=True)
phi2 = Variable(torch.tensor(0.05), requiers_grad=True)
opt = torch.optim.Adam([phi1, phi2], lr=0.1)

steps = 300

for i in range(steps):
  opt.zero_grad()
  loss = cost(phi, theta)
  loss.backward()
  opt.step()
```

PyTorch

PennyLane Summary

- Run and optimize directly on quantum hardware (GPU→QPU)
- “Quantum-aware” implementation of gradient descent optimization
- Hardware agnostic and extensible via plugins
- Open-source and extensively documented
- Use-cases:
  - Train quantum circuits as ML models
  - Machine learning of quantum computations
  - Hybrid quantum-classical machine learning

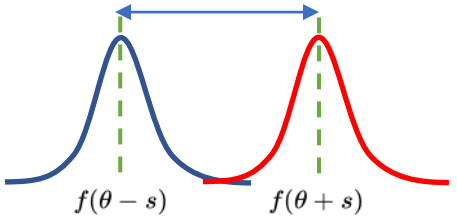
Source code: github.com/XanaduAI/pennylane

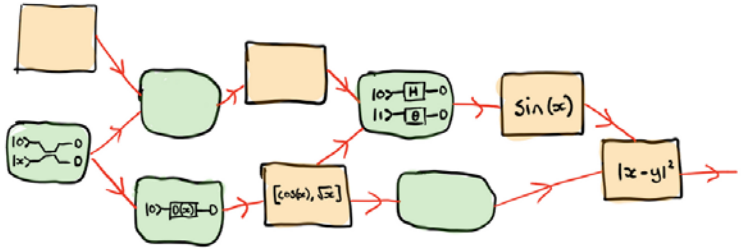
Documentation:
pennylane.readthedocs.io

Landing page:
pennylane.ai

Summary

- Variational circuits: strong foundation for near-term QML
- Compute gradients of quantum circuits using "parameter shift" method
- Train quantum circuits the same as neural networks
- QNode abstraction enables highly flexible hybrid classical-quantum computation
- Speed up progress via ease-of-implementation, reusability, sharing code/models, rapid iteration





https://github.com/XanaduAI/pennylane

https://pennylane.readthedocs.io

https://pennylane.ai

Thank You

XANADU