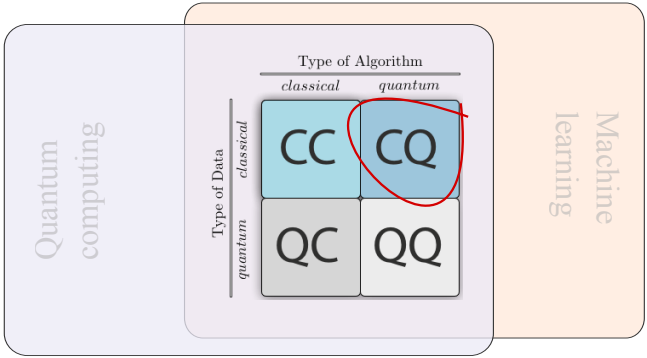
Quantum Machine Learning

Maria Schuld

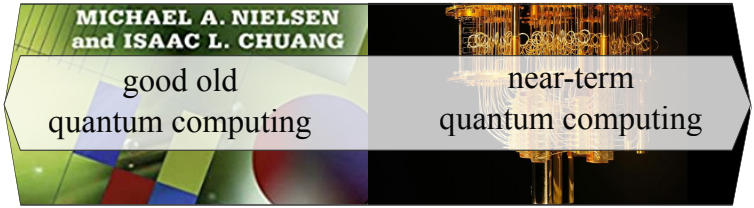
Xanadu and University of KwaZulu-Natal

QC Summer School, September 2020

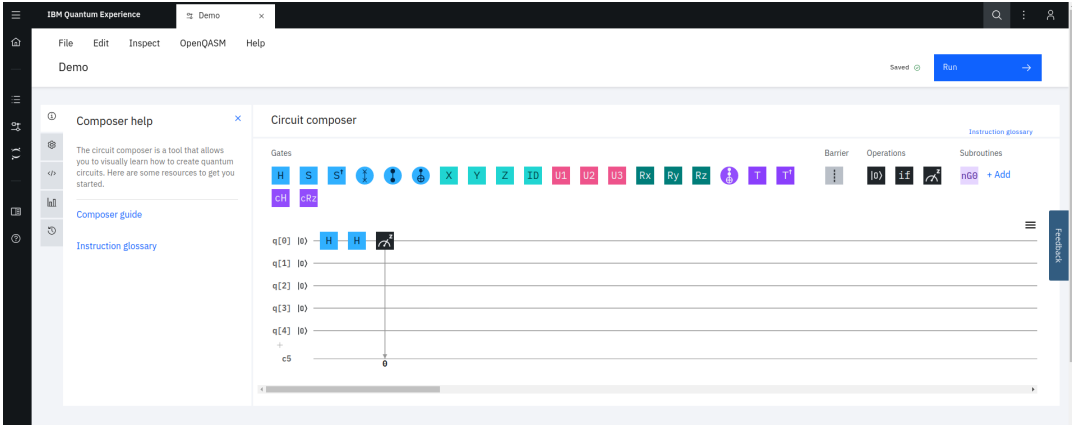
There are many intersections of QC and ML.



Quantum computing is an emerging technology.



Quantum computing is an emerging technology.



Quantum computing is an emerging technology.

An emerging technology needs applications.

WANTED

Application which

- doesn't care about noise
- gives us access to multi-billion $ markets
- attracts young researchers

An emerging technology needs applications.

WANTED

Application which

- doesn't care about noise
- gives us access to multi-billion $ markets
- attracts young researchers

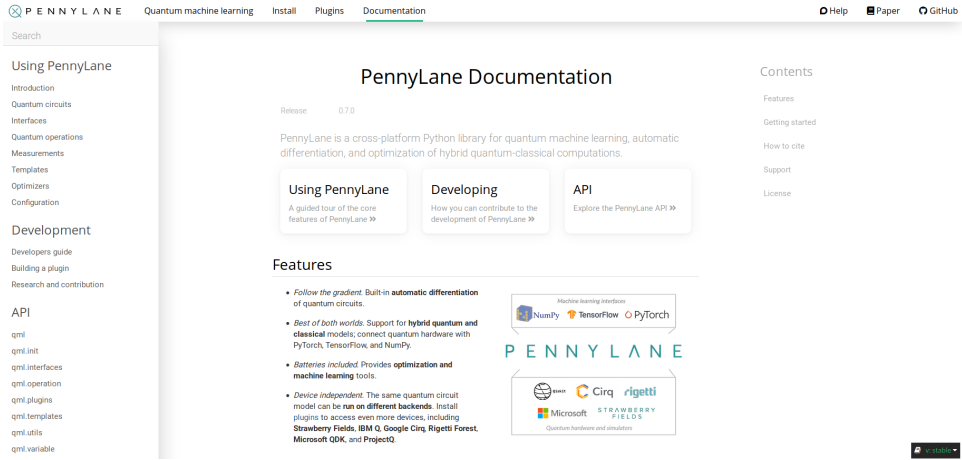
Machine Learning!

This lecture in a nutshell.

```python
import torch
from torch.autograd import Variable

data = torch.tensor([(0., 0.), (0.1, 0.1), (0.2, 0.2)])

def model(phi, x=None):
    return x*phi

def loss(a, b):
    return torch.abs(a - b) ** 2

def av_loss(phi):
    c = 0
    for x, y in data:
        c += loss(model(phi, x=x), y)
    return c

phi_ = Variable(torch.tensor(0.1), requires_grad=True)
opt = torch.optim.Adam([phi_], lr=0.02)

for i in range(5):
    l = av_loss(phi_)
    l.backward()
    opt.step()
```

```python
from pennylane import *
import torch
from torch.autograd import Variable

data = [(0., 0.), (0.1, 0.1), (0.2, 0.2)]

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

@qnode(dev, interface='torch')
def circuit(phi, x=None):
    templates.AngleEmbedding(features=[x], wires=[0])
    templates.BasicEntanglerLayers(weights=phi, wires=[0, 1])
    return expval(PauliZ(wires=[1]))

def loss(a, b):
    return torch.abs(a - b) ** 2

def av_loss(phi):
    c = 0
    for x, y in data:
        c += loss(circuit(phi, x=x), y)
    return c

phi_ = Variable(torch.tensor([[0.1, 0.2],[-0.5, 0.1]]), requires_grad=True)
opt = torch.optim.Adam([phi_], lr=0.02)

for i in range(5):
    l = av_loss(phi_)
    l.backward()
    opt.step()
```

This lecture in a nutshell.

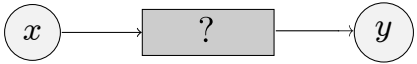


Agenda

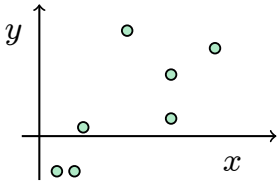
- Machine Learning Basics
- Quantum Computing in Pictures
- Quantum Machine Learning

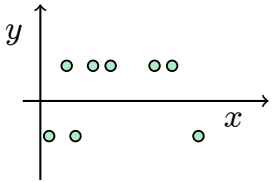
ML Basics

The first ingredient of ML is data.



$(x, y) \in \mathcal{X} \times \mathcal{Y}$

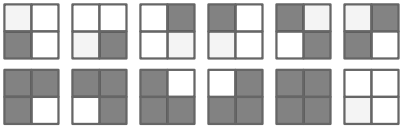




The first ingredient of ML is data.

$$(x, y) \in \mathcal{X} \times \mathcal{Y}$$

A

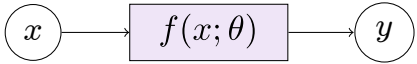


B

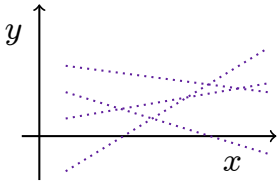
The first ingredient of ML is data.

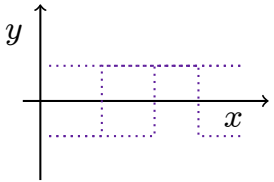
```
1    import torch
2    from torch.autograd import Variable
3
4    data = torch.tensor([(0., 0.), (0.1, 0.1), (0.2, 0.2)])
```

The second ingredient of ML is a model family.

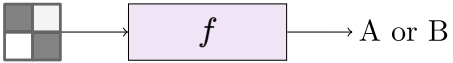


$f(x) \in \{\mathcal{F}\}$





The second ingredient of ML is a model family.



$f(x) \in \{\mathcal{F}\}$

f_1: It is A if a row or column is filled, else B

f_2: It is A if two blocks are filled, else B

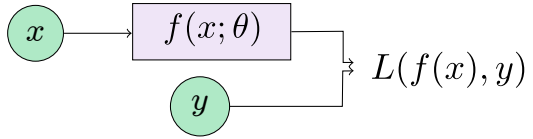
f_3: It is A if no block is filled, else B

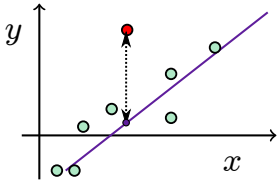
The second ingredient of ML is a model family.

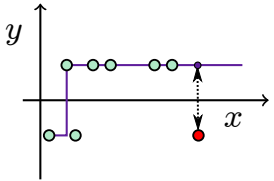
```python
import torch
from torch.autograd import Variable

data = torch.tensor([(0., 0.), (0.1, 0.1), (0.2, 0.2)])

def model(phi, x=None):
    return x*phi
```

The third ingredient of ML is a loss.

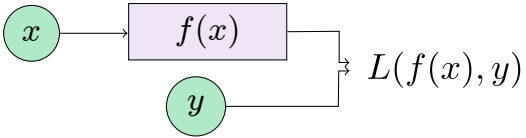
$L(f(x), y) \in \mathbb{R}$







The third ingredient of ML is a loss.



$L(f(x), y) \in \mathbb{R}$

$L(f(\square), A) = 1$

$L(f(\square), B) = 0$

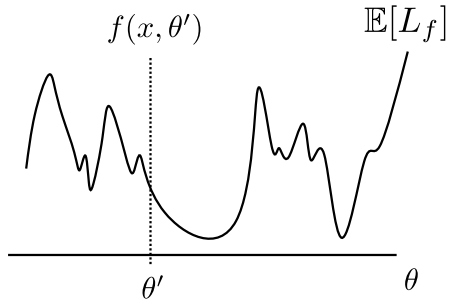
The third ingredient of ML is a loss.

```python
import torch
from torch.autograd import Variable

data = torch.tensor([(0., 0.), (0.1, 0.1), (0.2, 0.2)])

def model(phi, x=None):
    return x*phi

def loss(a, b):
    return torch.abs(a - b) ** 2
```

The goal of ML is to minimise the "average" loss of the model.

$$\mathbb{E}[L_f] = \int L(f(x), y)\, p(x, y)\, dxdy$$

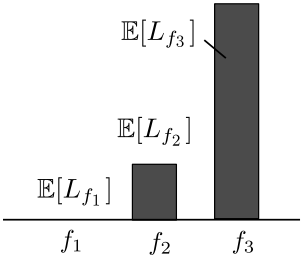
$$f^* = \min_{f \in \{\mathcal{F}\}} \mathbb{E}[L_f]$$



The goal of ML is to minimise the "average" loss of the model.

$$R(f) = \sum_{\text{squares}} L(f(\text{square}), y) \ \frac{1}{16}$$

$$f^* = \min_f R(f)$$

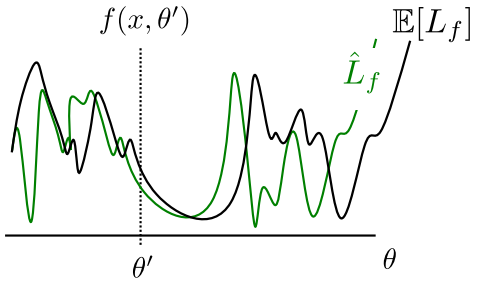


f_1: It is A if a row or column is filled, else B
f_2: It is A if two blocks are filled, else B
f_3: It is A if no block is filled, else B

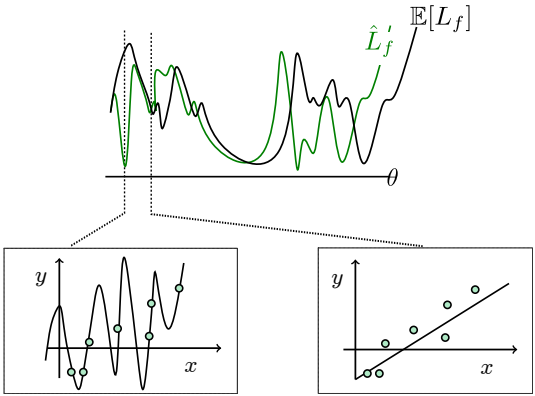
We can only minimise the average loss on training data.

$$\hat{L}_f = \sum_{(x,y)\in\mathcal{D}} L(f(x), y)$$

$$f^* = \min_{f\in\{\mathcal{F}\}} \hat{L}_f$$



We can only minimise the average loss on training data.

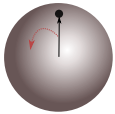


Putting it all together.

```python
import torch
from torch.autograd import Variable

data = torch.tensor([(0., 0.), (0.1, 0.1), (0.2, 0.2)])

def model(phi, x=None):
    return x*phi

def loss(a, b):
    return torch.abs(a - b) ** 2

def av_loss(phi):
    c = 0
    for x, y in data:
        c += loss(model(phi, x=x), y)
    return c

phi_ = Variable(torch.tensor(0.1), requires_grad=True)
opt = torch.optim.Adam([phi_], lr=0.02)

for i in range(5):
    l = av_loss(phi_)
    l.backward()
    opt.step()
```

Quantum computing

Quantum computers perform linear algebra.



PHYSICAL CIRCUIT

$$n \left[\begin{array}{c} |0\rangle \\ \vdots \\ |0\rangle \end{array} \right.$$

MATHEMATICAL DESCRIPTION

$$2^n \left[\begin{array}{c} 1+0i \\ 0+0i \\ \vdots \end{array} \right.$$

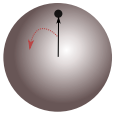
$|1|^2 = p(0...00)$

$|0|^2 = p(0...01)$

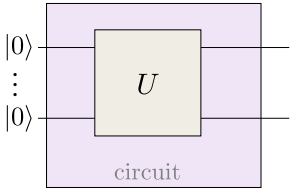
Quantum computers perform linear algebra.

```python
from pennylane import *

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

@qnode(dev)
def circuit():
    return probs(wires=[0, 1])

print(circuit()) # [1. 0. 0. 0.]
print(dev.state) # [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
```

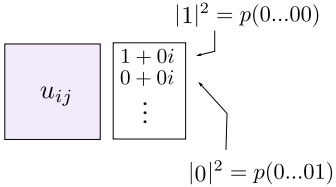
Quantum computers perform linear algebra.



PHYSICAL CIRCUIT

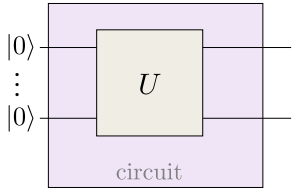


MATHEMATICAL DESCRIPTION



Quantum computers perform linear algebra.

PHYSICAL CIRCUIT



MATHEMATICAL DESCRIPTION

$|\psi_1|^2 = p(0...00)$

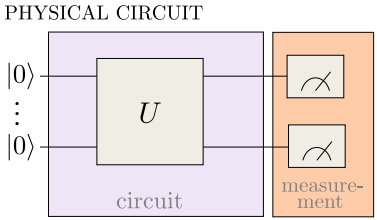
ψ_1
ψ_2
$\vdots$

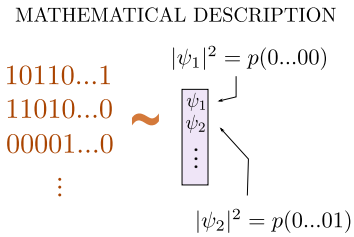
$|\psi_2|^2 = p(0...01)$

Quantum computers perform linear algebra.

```
from pennylane import *
import numpy as np

dev = device('default.qubit', wires=2)
U = np.array([[0.,   -0.70710678, 0.,    0.70710678],
              [0.70710678, 0.,   -0.70710678, 0.   ],
              [0.70710678, 0.,    0.70710678, 0.   ],
              [0.,   -0.70710678, 0.,   -0.70710678]])

@qnode(dev)
def circuit():
    QubitUnitary(U, wires=[0, 1])
    return probs(wires=[0, 1])

print(circuit()) # [0. 0.5 0.5 0.]
print(dev.state) # [0.+0.j 0.707+0.j 0.707+0.j 0.+0.j]
```

Quantum computers perform linear algebra.

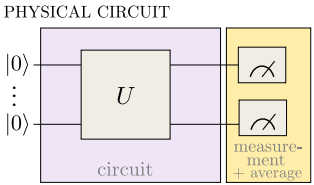


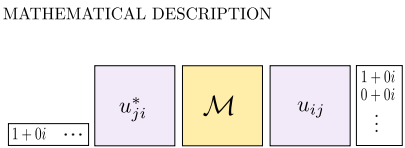


Quantum computers perform linear algebra.

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

dev = device('default.qubit', wires=2, shots=1)
U = np.array([[0.,   -0.70710678, 0.,    0.70710678],
              [0.70710678, 0.,   -0.70710678, 0.   ],
              [0.70710678, 0.,    0.70710678, 0.   ],
              [0.,   -0.70710678, 0.,   -0.70710678]])

@qnode(dev)
def circuit():
    QubitUnitary(U, wires=[0, 1])
    return sample(PauliZ(wires=0)), sample(PauliZ(wires=1))

print(circuit()) # [[-1], [ 1]]
print(dev.state) # [0.+0.j 0.707+0.j 0.707+0.j 0.+0.j]
```

Quantum computers perform linear algebra.



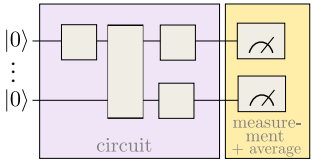


Quantum computers perform linear algebra.

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

dev = device('default.qubit', wires=2, shots=1)
U = np.array([[0.,   -0.70710678, 0.,    0.70710678],
              [0.70710678, 0.,   -0.70710678, 0.   ],
              [0.70710678, 0.,    0.70710678, 0.   ],
              [0.,   -0.70710678, 0.,   -0.70710678]])

@qnode(dev)
def circuit():
    QubitUnitary(U, wires=[0, 1])
    return expval(PauliZ(wires=0)), expval(PauliZ(wires=1))

print(circuit()) # [0., 0.]
print(dev.state) # [0.+0.j 0.707+0.j 0.707+0.j 0.+0.j]
```

Quantum computers perform linear algebra.

PHYSICAL CIRCUIT



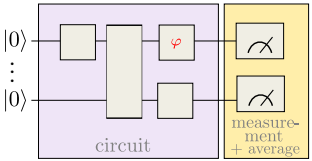
MATHEMATICAL DESCRIPTION

Quantum computers perform linear algebra.

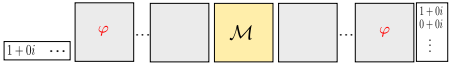
```python
from pennylane import *

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

@qnode(dev)
def circuit():
    PauliX(wires=0)
    CNOT(wires=[0, 1])
    Hadamard(wires=0)
    PauliZ(wires=1)
    return expval(PauliZ(wires=0)), expval(PauliZ(wires=1))

print(circuit()) # [0., -1.]
print(dev.state) # [ 0.+0.j -0.70710678+0.j  0.+0.j  0.70710678+0.j]
```

Quantum computers perform linear algebra.

PHYSICAL CIRCUIT



MATHEMATICAL DESCRIPTION

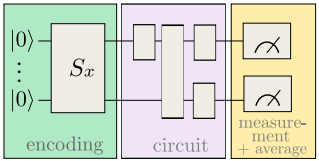


Quantum computers perform linear algebra.

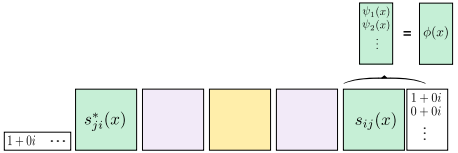
```python
from pennylane import *

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

@qnode(dev)
def circuit(phi):
    RX(phi, wires=0)
    CNOT(wires=[0, 1])
    Hadamard(wires=0)
    PauliZ(wires=1)
    return expval(PauliZ(wires=0)), expval(PauliZ(wires=1))

print(circuit(0.2)) # [0. 0.98006658]
print(dev.state) # [0.70+0.j 0.+0.07j 0.70+0.j 0.-0.070j]
```

We can feed data into quantum computers.

PHYSICAL CIRCUIT



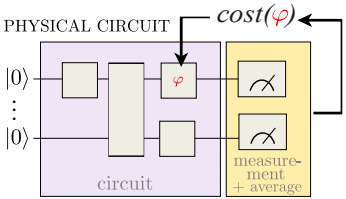
MATHEMATICAL DESCRIPTION

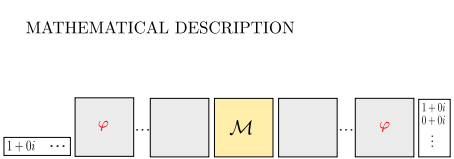


We can feed data into quantum computers.

```
from pennylane import *

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

@qnode(dev)
def circuit(phi, x=None):
    RX(x, wires=[0])
    CNOT(wires=[0, 1])
    RY(phi, wires=[1])
    return expval(PauliZ(wires=[1]))

print(circuit(0.2, x=0.1)) # 0.975
print(circuit(0.2, x=0.5)) # 0.860
```

We can train quantum computers.

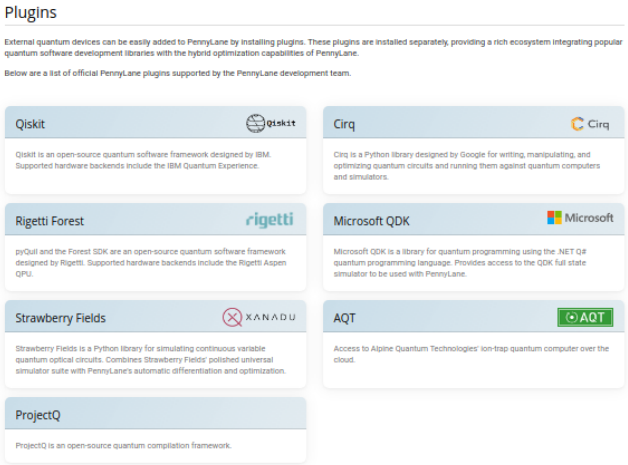




We can train quantum computers.

```
from pennylane import *

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

@qnode(dev)
def circuit(phi):
    Hadamard(wires=0)
    RY(phi, wires=0)
    return expval(PauliZ(wires=0))

phi = 0.2
opt = GradientDescentOptimizer(stepsize=0.2)

for i in range(5):
    phi = opt.step(circuit, phi)

    print(phi)
    # 0.39601331556824826
    # 0.5805345472544579
    # 0.7477684644009802
    # 0.8944100937922911
    # 1.0196058853338432
```

There are many different devices.



Plugins

External quantum devices can be easily added to PennyLane by installing plugins. These plugins are installed separately, providing a rich ecosystem integrating popular quantum software development libraries with the hybrid optimization capabilities of PennyLane.

Below are a list of official PennyLane plugins supported by the PennyLane development team.

Qiskit

Qiskit is an open-source quantum software framework designed by IBM. Supported hardware backends include the IBM Quantum Experience.

Cirq

Cirq is a Python library designed by Google for writing, manipulating, and optimizing quantum circuits and running them against quantum computers and simulators.

Rigetti Forest

pyQuil and the Forest SDK are an open-source quantum software framework designed by Rigetti. Supported hardware backends include the Rigetti Aspen QPU.

Microsoft QDK

Microsoft QDK is a library for quantum programming using the .NET Q# quantum programming language. Provides access to the QDK full state simulator to be used with PennyLane.

Strawberry Fields

Strawberry Fields is a Python library for simulating continuous variable quantum optical circuits. Combines Strawberry Fields' polished universal simulator suite with PennyLane's automatic differentiation and optimization.

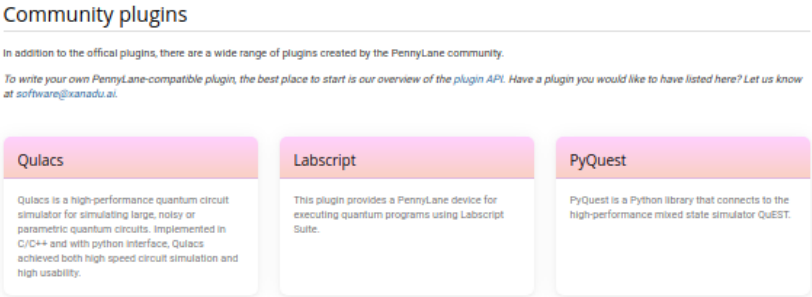
AQT

Access to Alpine Quantum Technologies' ion-trap quantum computer over the cloud.

ProjectQ

ProjectQ is an open-source quantum compilation framework.

There are many different devices.



Community plugins

In addition to the offical plugins, there are a wide range of plugins created by the PennyLane community.

To write your own PennyLane-compatible plugin, the best place to start is our overview of the plugin API. Have a plugin you would like to have listed here? Let us know at software@xanadu.ai.

Qulacs

Qulacs is a high-performance quantum circuit simulator for simulating large, noisy or parametric quantum circuits. Implemented in C/C++ and with python interface, Qulacs achieved both high speed circuit simulation and high usability.

Labscript

This plugin provides a PennyLane device for executing quantum programs using Labscript Suite.

PyQuest

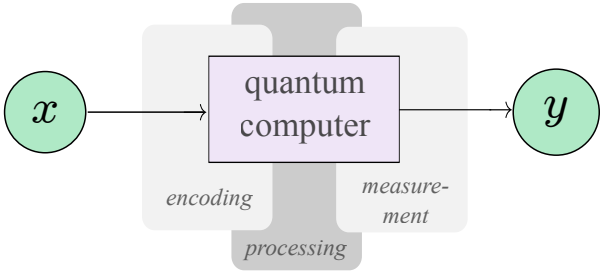
PyQuest is a Python library that connects to the high-performance mixed state simulator QuEST.

There are many different devices.

```
from pennylane import *

dev1 = device('qiskit.ibmq', wires=1)
dev2 = device('qiskit.aer', wires=1)
dev3 = device('cirq.simulator', wires=1)
dev4 = device('forest.wavefunction', wires=1)
dev5 = device('microsoft.QuantumSimulator', wires=1)
dev6 = device('default.qubit.tf', wires=1)
```

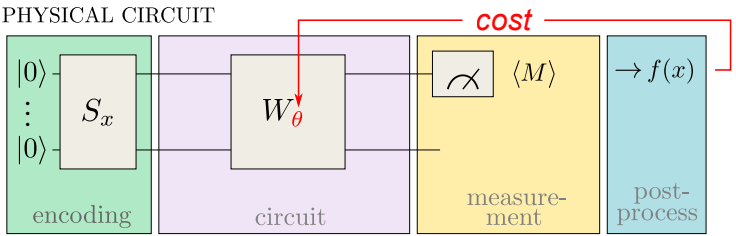
Quantum Machine Learning

Quantum circuits can be used as machine learning models.



Farhi & Neven 1802.06002, Schuld et al. 1804.00633, Benedetti et al. 1906.07682

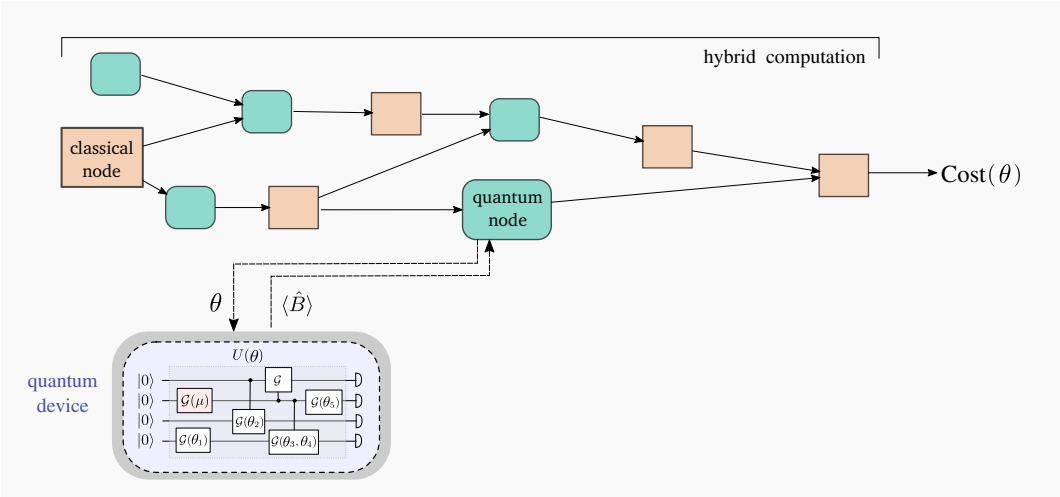
Quantum circuits can be used as machine learning models.



Farhi & Neven 1802.06002, Schuld et al. 1804.00633, Benedetti et al. 1906.07682

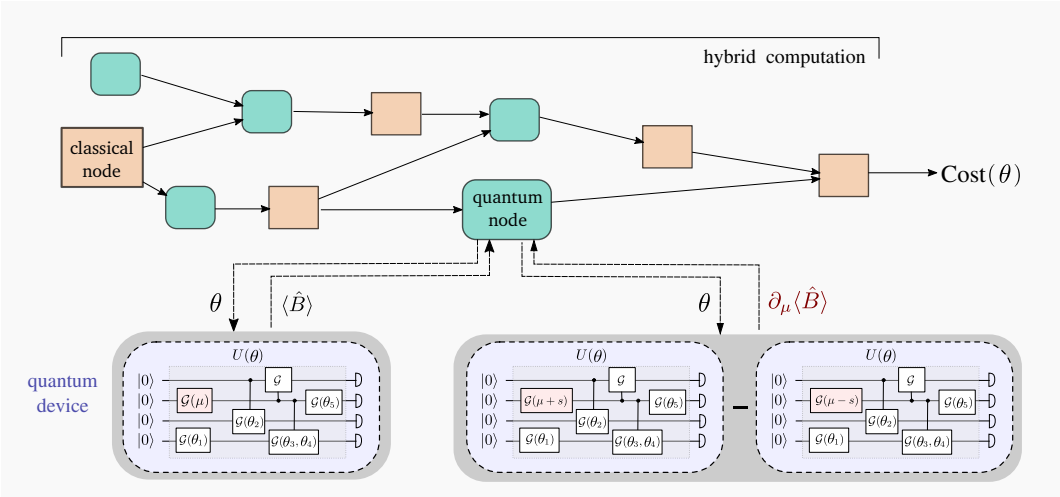
1. How can we optimize quantum models?

We can compute gradients of quantum computations.



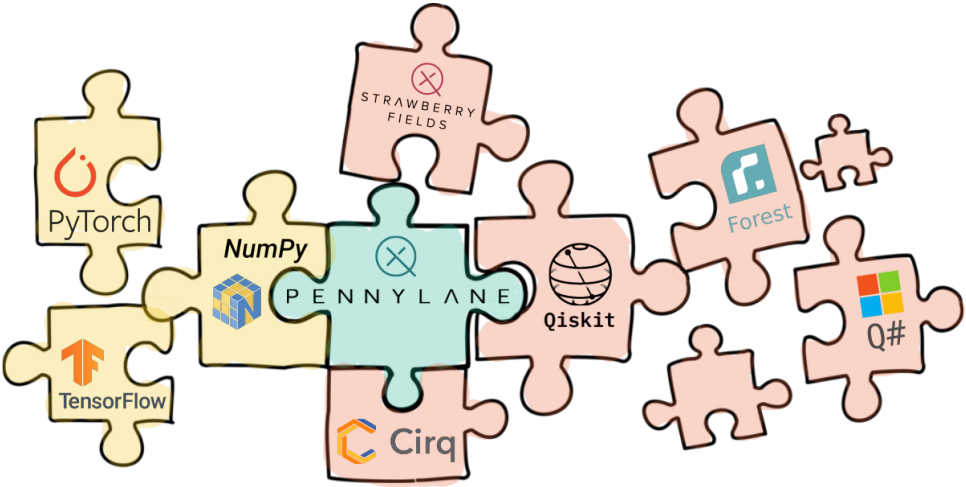
Guerreschi & Smelyanskiy 1701.01450, Mitarai et al. 1803.00745, Schuld et al. 1811.11184

We can compute gradients of quantum computations.

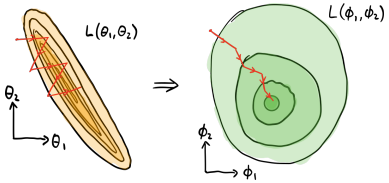


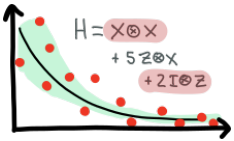
Guerreschi & Smelyanskiy 1701.01450, Mitarai et al. 1803.00745, Schuld et al. 1811.11184

We can compute gradients of quantum computations.



We can compute gradients of quantum computations.





Stokes et al. 1909.02108, Kübler et al. 1909.09083, Sweke et al. 1910.01155, Ostaszewski et al. 1905.09692, ...

We can compute gradients of quantum computations.

Barren plateaus in quantum neural network training landscapes

Jarrod R. McClean,[1,*] Sergio Boixo,[1,†] Vadim N. Smelyanskiy,[1,‡] Ryan Babbush,[1] and Hartmut Neven[1]
[1] *Google Inc., 340 Main Street, Venice, CA 90291, USA*
(Dated: March 30, 2018)

Many experimental proposals for noisy intermediate scale quantum devices involve training a parameterized quantum circuit with a classical optimization loop. Such hybrid quantum-classical algorithms are popular for applications in quantum simulation, optimization, and machine learning. Due to its simplicity and hardware efficiency, random circuits are often proposed as initial guesses for exploring the space of quantum states. We show that the exponential dimension of Hilbert space and the gradient estimation complexity make this choice unsuitable for hybrid quantum-classical algorithms run on more than a few qubits. Specifically, we show that for a wide class of reasonable parameterized quantum circuits, the probability that the gradient along any reasonable direction is non-zero to some fixed precision is exponentially small as a function of the number of qubits. We argue that this is related to the 2-design characteristic of random circuits, and that solutions to this problem must be studied.

quant-ph] 29 Mar 2018

Rapid developments in quantum hardware have motivated advances in algorithms to run in the so-called noisy intermediate scale quantum (NISQ) regime [1]. Many of the most promising application-oriented approaches are hybrid quantum-classical algorithms that rely on optimization of a parameterized quantum circuit [2–8]. The resilience of these approaches to certain types of errors and high flexibility with respect to coherence time and gate requirements make them especially attractive for NISQ implementations [3, 9–11].

The first implementation of such algorithms was de-

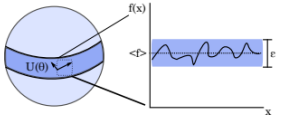


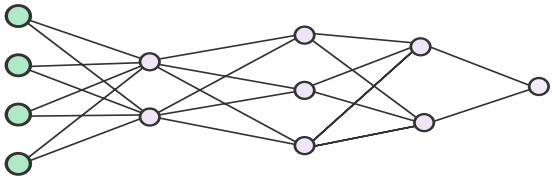
FIG. 1. A cartoon of the general geometric results from this work. The sphere depicts the phenomenon of concentration of

McClean et al. 1803.11173

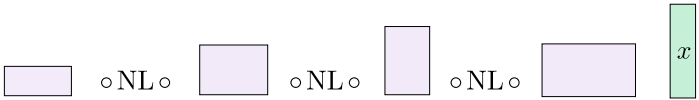
2. What *are* these models?

Quantum circuits are unitary neural nets in feature space.

MODEL

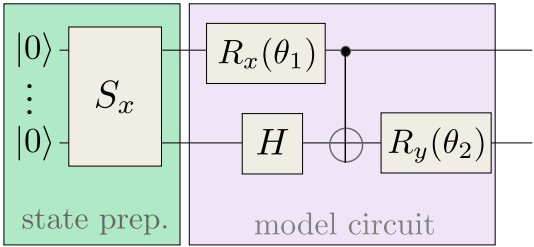


MATHEMATICAL DESCRIPTION



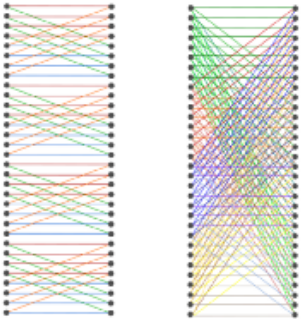
Quantum circuits are unitary neural nets in feature space.

PHYSICAL CIRCUIT



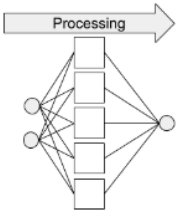
MATHEMATICAL DESCRIPTION

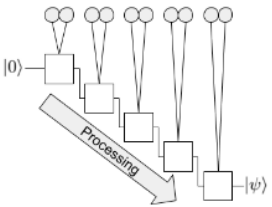
Quantum circuits are unitary neural nets in feature space.

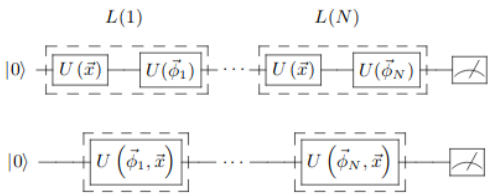


Schuld & Petruccione, Springer 2018

Quantum circuits are unitary neural nets in feature space.

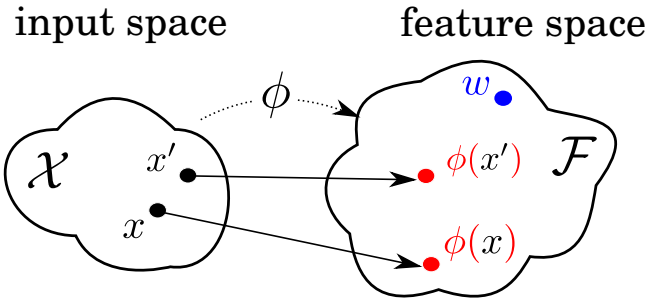






Pérez-Salinas et al. 1907.02085

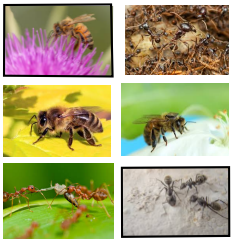
Quantum circuits are kernel methods.

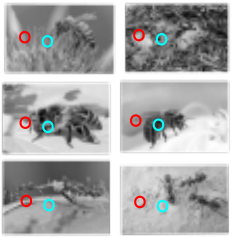


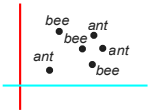
$$f(x) = \langle \phi(x), w \rangle$$

$$\kappa(x, x') = \langle \phi(x), \phi(x') \rangle$$

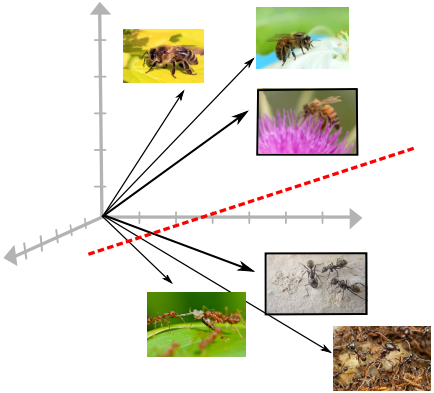
Quantum circuits are kernel methods.



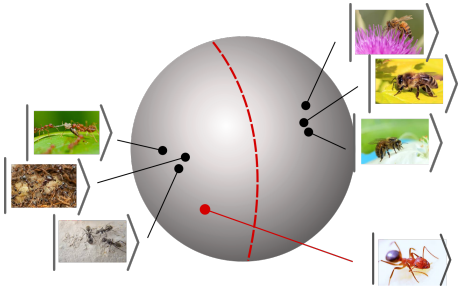




Quantum circuits are kernel methods.



Quantum circuits are kernel methods.



Quantum circuits are kernel methods.

Quantum feature map:

$$x \to \phi(x) \in \mathbb{C}^{2^{n_{\text{qubits}}}}$$

Quantum circuits are kernel methods.

Quantum feature map:

$$x \to \phi(x) \in \mathbb{C}^{2^{n_{\text{qubits}}}}$$

Measurement:

$$\phi(x)^\dagger M \phi(x)$$

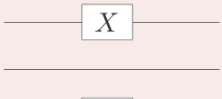
Quantum circuits are kernel methods.

Measurement:

$$\begin{aligned} & \phi(x)^T M \phi(x) \\ &= \phi(x)^\dagger \; w w^\dagger \; \phi(x) \\ &= |\phi(x)^\dagger w|^2 \end{aligned}$$

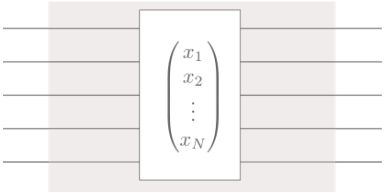
Data encoding defines a “quantum feature map”.

$$x \to \phi(x) = \begin{pmatrix} \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix}$$



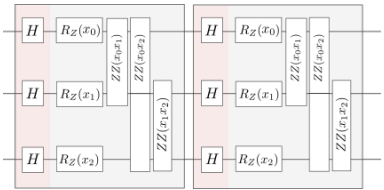
Data encoding defines a “quantum feature map”.

$$x \to \phi(x) = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix}$$

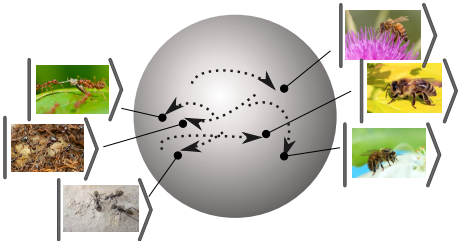


Data encoding defines a “quantum feature map”.

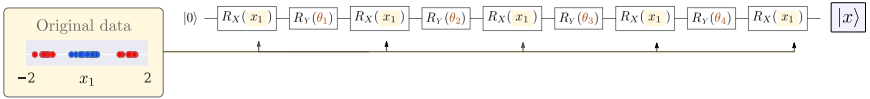
$$x \to S(x) \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{pmatrix}$$



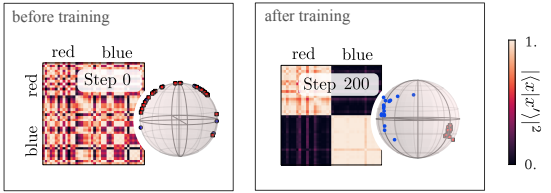
We can engineer/train our features.



We can engineer/train our features.

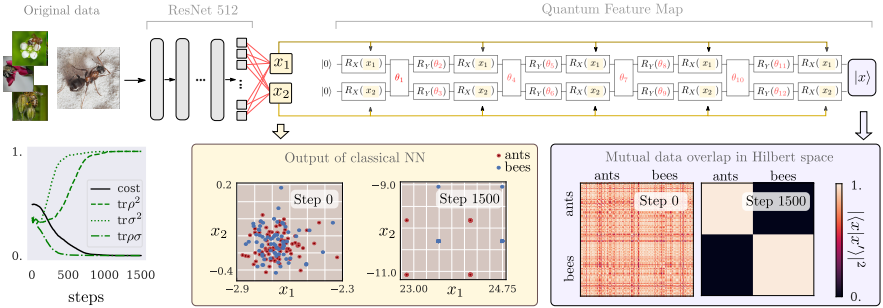


Mutual data overlap in Hilbert space



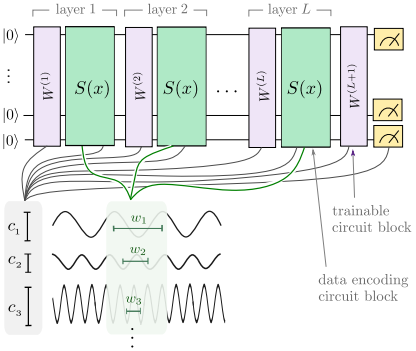
Lloyd et al. 2001.03622

We can engineer/train our features.



Lloyd et al. 2001.03622

Quantum circuits are partial Fourier series.



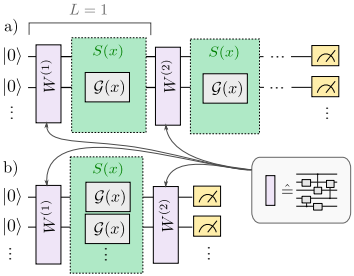
Input x encoded by gate e^{-ixH}:

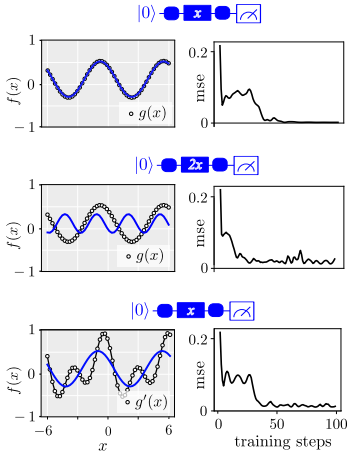
$$f(x) = \sum_{\omega} c_\omega e^{i\omega x}$$

$$f(x) = \sum_{n=-K}^{K} c_n e^{inx}$$

Schuld, Sweke and Meyer 2008.08605

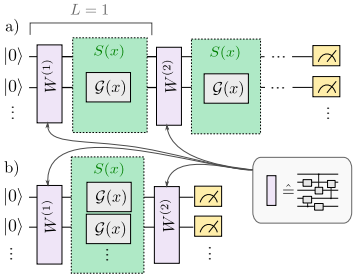
Quantum circuits are partial Fourier series.

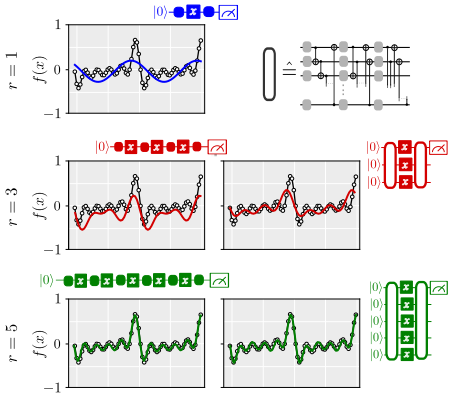




Schuld, Sweke and Meyer 2008.08605

Quantum circuits are partial Fourier series.





Schuld, Sweke and Meyer 2008.08605

3. How good are these models at generalisation?

CONCLUSION

Summary,...

machine intelligence = data/distributions + algorithm/hardware + models

... some open questions,...

- What models are quantum circuits?
- Are they actually useful?
- Will they perform well on larger problem instances?
- Will they perform well under noise?
- What problems are they good for?
- Is there a practically relevant problem for which QC are exponentially faster?
- Can QC accelerate machine learning?
- Can QC push the boundaries of what is learnable?

...and some advice.

- Don't compare quantum models blindly to classical ML.
- Understand the features and models you use.
- Understand what feature map your model performs.

Thank you!

www.pennylane.ai
www.xanadu.ai
@XanaduAI