QML and the Secrets of the Hilbert Space

Jacob L. Cybulski
Enquanted, Australia
and Deakin University, SIT

The dance of the blind puppeteer:

ML and QML
Parameterised circuits
Variational quantum algorithms
Data encoding and decoding
State measurement
Ansatz design and training
Model geometry and gradients
The dance - parameters optimisation
The curse of dimensionality
Qiskit / PennyLane quantum solution
Dealing with variance and errors
QML readings
Summary and Q&A

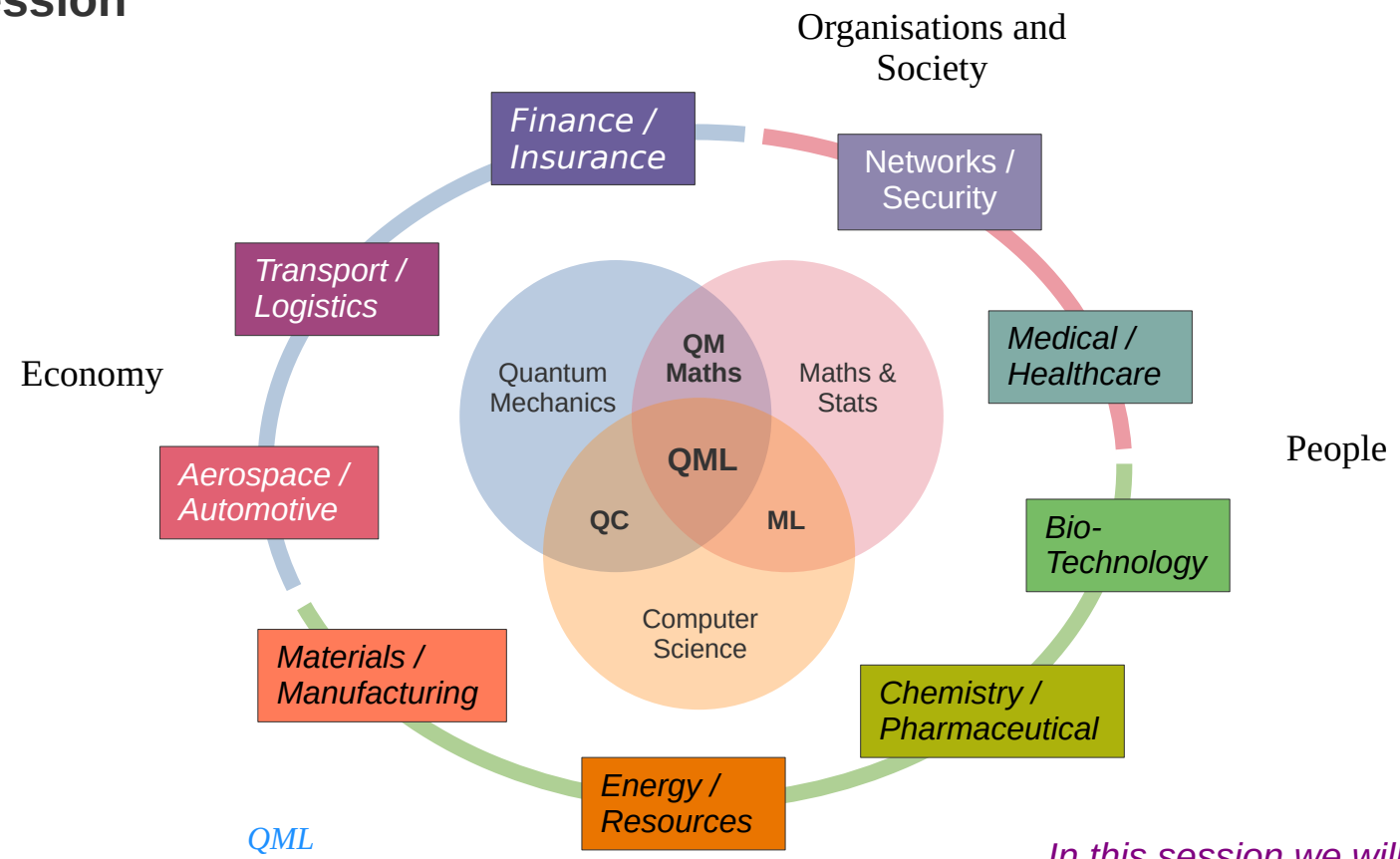
See: Ironfrown (Github)

Feb 26, 2026, Uttaranchal University, Dehradun, Uttarakhand, India

Creative Commons CC BY-NC-ND

Quantum ML
aims of this session

Jacob Cybulski, Founder
Enquanted, Australia



QML applications

Where is QAI?

In this session we will explain the very nature of QML models - the Hilbert space!

Who is doing what and where in QML?

Category Name	Company Name	Sample Project	Continent or Region
Business & Logistics	D-Wave Quantum	Real-time supply chain and fleet routing	North America
	Quanmatic	Semiconductor manufacturing workflow optimization	APAC
	Zapata Quantum	Enterprise generative AI and industrial optimization	North America
	Fujitsu	Digital annealing for traffic and logistics AI	APAC
	QuantumSouth	Cargo load and payload distribution optimization	South America
Finance	Multiverse Computing	Risk management, credit scoring, and stress testing	Europe
	IBM Quantum	Quantum fraud detection and options pricing	North America
	Scenario X	Real-time financial stress testing and risk modeling	Europe
	Horizon Quantum	Automated compilation of classical code into QAI	APAC
	Q-Africa	Financial inclusion and credit-scoring for unbanked	Africa
Chem & Pharma	Microsoft	AI agents for molecular screening and discovery	North America
	Qubit Pharmaceuticals	Physics-driven drug discovery foundation models	Europe
	Algorithmiq	Quantum noise mitigation for drug-target binding	Europe
	Quantum Intelligence	Neural-network based analysis of ADME/Tox	APAC
	G42	Genomic research and Arabic Large Language Models	Arab World
Engineering	SECQAI	Computational Fluid Dynamics (CFD) for aerospace	Europe
	GenMat	Generative materials science for high-perf alloys	North America
	BQP *(BosonQ Psi)*	Quantum-inspired engineering and CAD simulations	India
	Altransinnov	Predictive monitoring for energy infrastructure	Europe
	Senfio	Precision agriculture and yield prediction	South America
	Quantum Brilliance	Edge Quantum AI for robotics and satellites	Australia

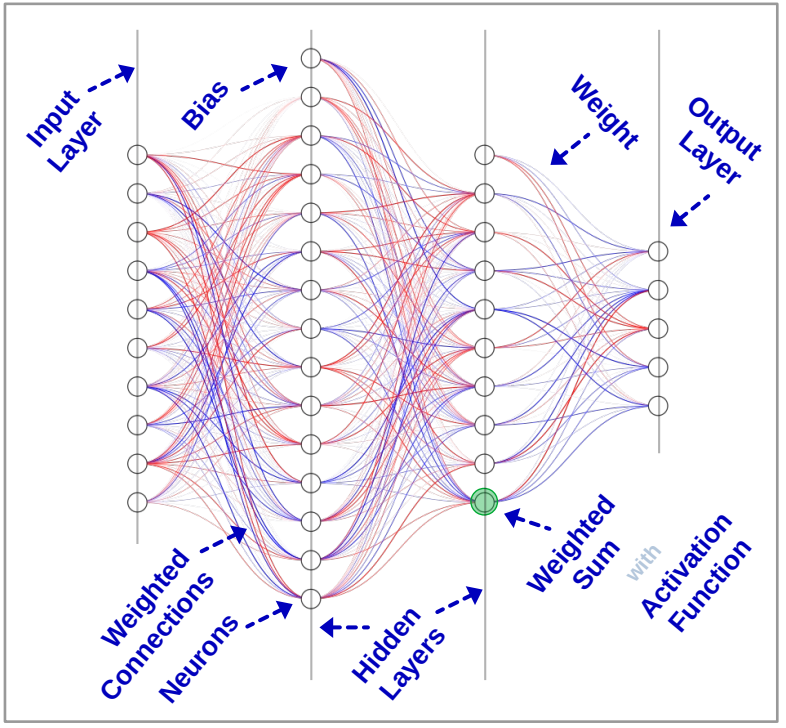
BQP: US company with Indian roots, with a major tech hub in Bengaluru.

Founded in 2020 by Abhishek Chopra, Rut Lineswala, Jash Minocha and Aditya Singh.

The company name pays homage to the Indian physicist Satyendra Nath Bose.

Multilayer Perceptron (MLP)

A class of Neural Network (NN) models



The cost of each NN model is a point in a multi-D space of weights

w1 x **w2** x ... X **MAE**

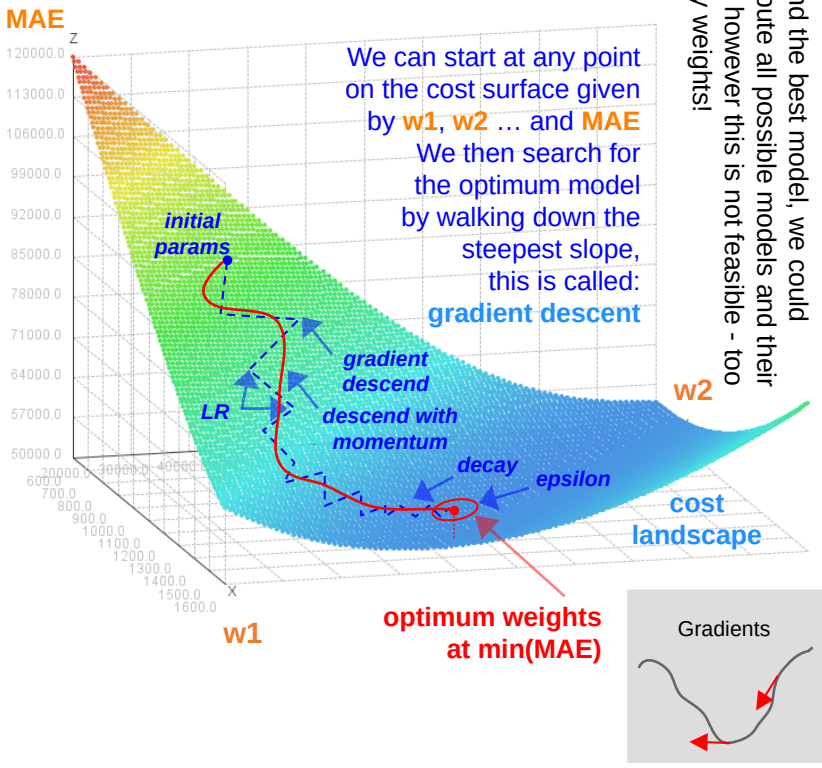
All such points form a “cost” surface.

The shape of such a surface we call the **cost landscape**.

When a model has many weights, the cost surface is multi-dimensional and called a manifold.

Now we will look for the best NN model, i.e. the model which generates the lowest MAE

To find the best model, we could compute all possible models and their cost, however this is not feasible - too many weights!



- MLP is capable of learning any “smooth” function by associating inputs with outputs
- Other NNs: CNN, AE, GAN, LSTM + QNNs

The optimizer controls this process via hyper-parameters, i.e. parameters of the gradient descent itself:

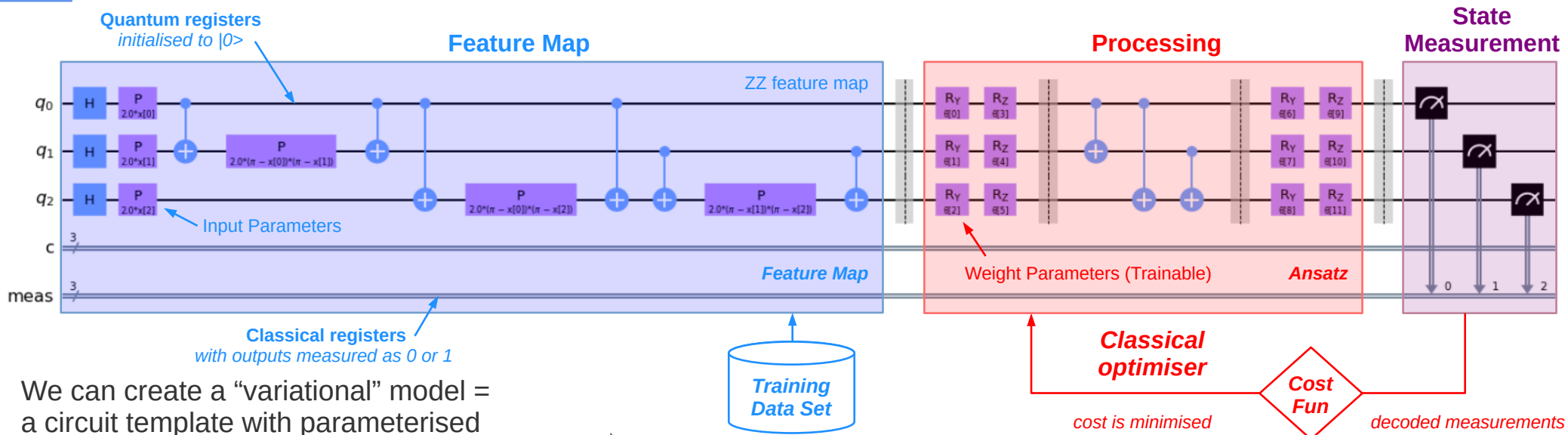
learning rate
momentum
decay
epsilon

By using gradient descent, the optimum cost (and thus the model), was found at:

A=1060 (Lot_Frontage)
B=90000 (Intercept)
MAE=53473.097 (Error)

Parameterized Quantum Circuits (PQC) & Variational Quantum Algorithms (VQA)

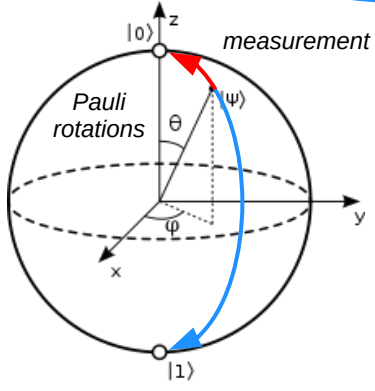
Variational quantum circuits are not executable!
Their input and weight params must be assigned values!
Backpropagation cannot work on quantum machines!



We can create a “variational” model = a circuit template with parameterised gates, e.g. P(a), Ry(a) or Rz(a), each allowing rotation of a qubit state in x, y or z axis (as per Bloch sphere).

Typically (but now always), such circuits consist of three blocks:

- a feature map (input)
- an ansatz (processing)
- measurements (output)



Classical input data is encoded (embedded) into the feature map’s parameters, setting the model’s initial quantum state.

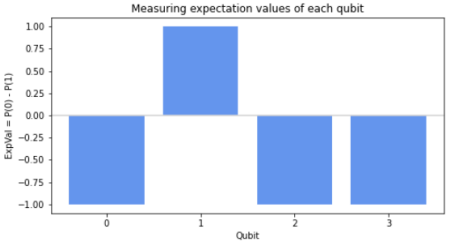
Ansatz evolves the quantum state, it consists of paramete-rised quantum gates, trained by a classic optimiser

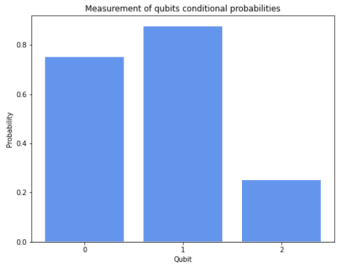
The circuit final state is measured and decoded (interpreted) as the model’s output in the form of classical data.

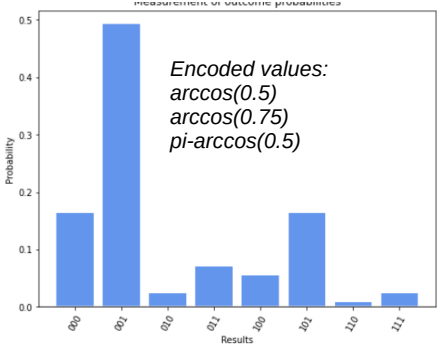
Data Encoding

Maria Schuld and Francesco Petruccione
Machine Learning with Quantum Computers. 2nd ed. Springer, 2021.

Many encoding methods, e.g. basis, angle, amplitude, QRAM, ...

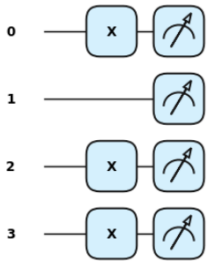






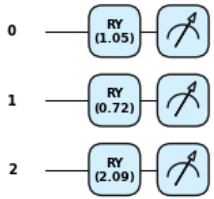
Encoding can be repeated across the circuit, which is called data reuploading

Basis encoding



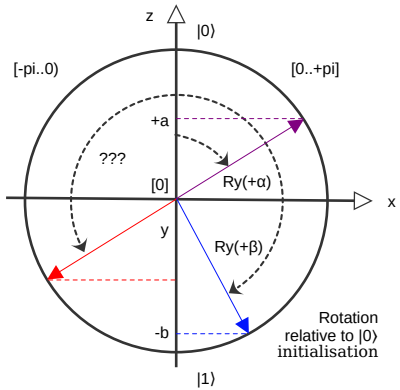
The simplest data encoding and very popular, however, little data can be encoded at a time, you can encode only binary values per each qubit.

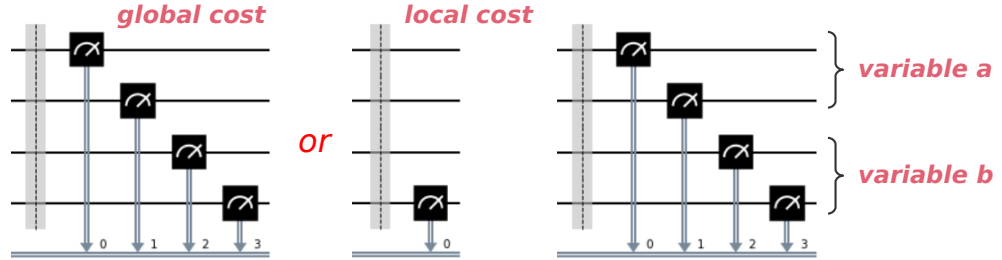
Angle encoding

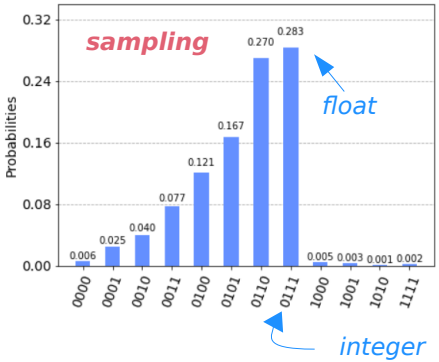


One of the most flexible and very effective, you can encode floating point numbers.

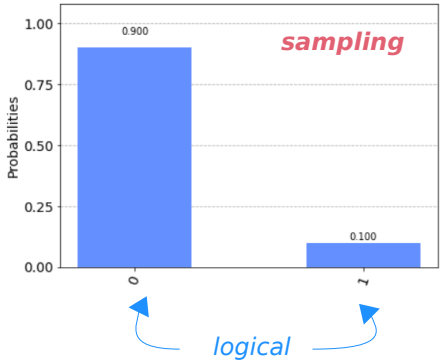
What you encode depends on your intention!

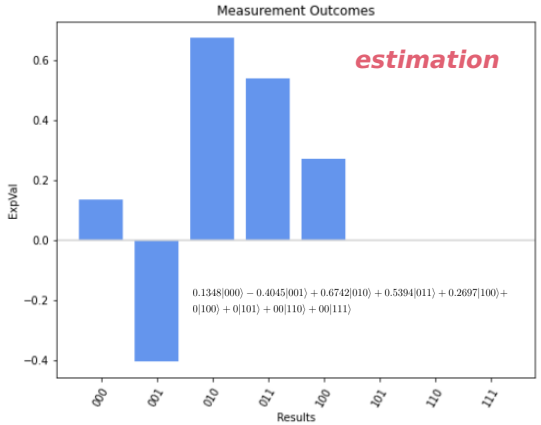






or





Measurements must be repeated, collected and then can be interpreted in many different ways

It is also possible to measure mid-circuit, however, beware as the circuit is no longer unitary and not reversible!

State Measurement

Ansatz design

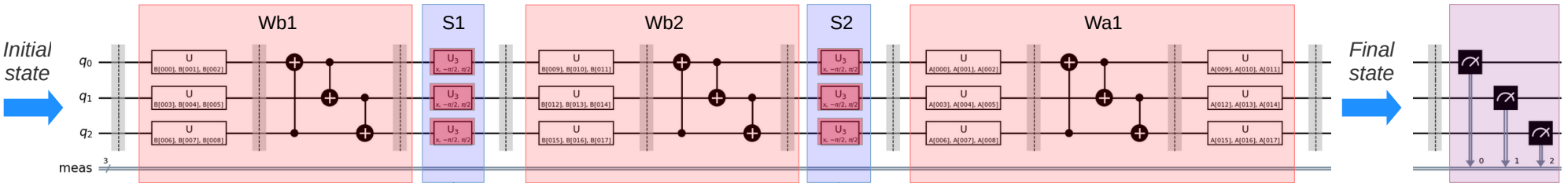
Beware that
adding qubits adds parameters and entanglements!

The number of states represented by the circuit **grows exponentially** with the number of qubits!

Beware that
adding 1 measurement doubles the number of outcomes!

So... having ***n*** measurements leads to 2^n outcomes

State evolution across ansatz layers



*Data can be **reuploaded** across circuit's width and depth*

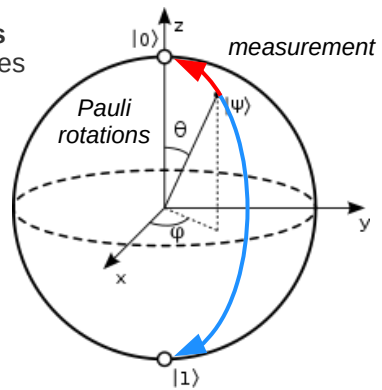
feature maps vary in:
structure and function

ansatze vary in:

- width (qubits #)
- depth (layers #)
- dimensions (param #)
- structure (e.g. funnelling)
- entangling (circular, linear, sca)

ansatz layers consist of:
rotation blocks and entangling blocks of U(z, y, z) and CNOT gates
(rotations) (entanglement)

rotation gates
alter qubit states around x, y, z axes



To execute a circuit we just apply it to input data and the optimum parameters

different cost functions:
R2, MAE, MSE, Huber, Poisson, cross-entropy, hinge-embedding, Kullback-Leibner divergence

different optimisers:
gradient based (Adam, NAdam and SPSA)
linear approximation methods (COBYLA)
non-linear approximation methods (BFGS)
quantum natural gradient optimiser (QNG)

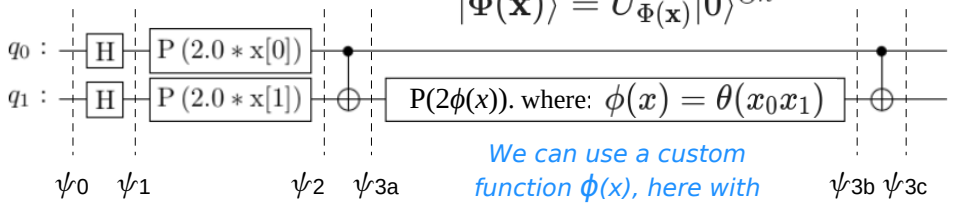
circuit execution on:
simulators (CPUs), accelerators (GPUs) and real quantum machines (QPUs)

What is the Hilbert space?

Example: Consider the following ZZ feature map

The story of the Hilbert space, the circuit states, the state vector paths and manifolds – and this is just the beginning...

$$|\Phi(\mathbf{x})\rangle = U_{\Phi(\mathbf{x})}|0\rangle^{\otimes n}$$



We can use a custom function ϕ(x), here with θ trainable parameter

the final state:

$$|\Phi(\mathbf{x})\rangle = \frac{1}{2}\begin{pmatrix} 1 \\ e^{i(2x_1+2\phi(x))} \\ e^{i(2x_0+2\phi(x))} \\ e^{i(2x_0+2x_1)} \end{pmatrix} \begin{matrix} \leftarrow |00\rangle \\ \leftarrow |01\rangle \\ \leftarrow |10\rangle \\ \leftarrow |11\rangle \end{matrix}$$

Step 0: Initial State

$$|\psi_0\rangle = |0\rangle \otimes |0\rangle = |00\rangle$$

Step 1: Superposition (Hadamard Gates $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$)

$$|\psi_1\rangle = (H \otimes H)|00\rangle = \frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$$

Step 2: Phase Gates (adds phase when qubit is in state |1>)

$$|\psi_2\rangle = \frac{1}{2}\left(|00\rangle + e^{i2x_1}|01\rangle + e^{i2x_0}|10\rangle + e^{i(2x_0+2x_1)}|11\rangle\right)$$

Step 3: Entanglement (CNOT – Phase – CNOT)

3a. First CNOT (Control q0, Target q1)

$$|\psi_{3a}\rangle = \frac{1}{2}\left(|00\rangle + e^{i2x_1}|01\rangle + e^{i2x_0}|11\rangle + e^{i(2x_0+2x_1)}|10\rangle\right)$$

3b. Middle Phase Gate P(2ϕ) on q1

$$|\psi_{3b}\rangle = \frac{1}{2}\left(|00\rangle + e^{i2x_1}e^{i2\phi}|01\rangle + e^{i2x_0}e^{i2\phi}|11\rangle + e^{i(2x_0+2x_1)}|10\rangle\right)$$

3c. Second CNOT (Control q0, Target q1)

$$|\psi_{3c}\rangle = \frac{1}{2}\left(|00\rangle + e^{i(2x_1+2\phi)}|01\rangle + e^{i(2x_0+2\phi)}|10\rangle + e^{i(2x_0+2x_1)}|11\rangle\right)$$

The Hilbert space is a coordinate system and a space of directions that the state of your model could take during its execution. As it executes it leaves a path, a “trail” its state vector follows. Because of entanglement, the state vector is a “complex” arrow and the trail becomes “knotted”. Because of Heisenberg uncertainty and noise, each time the model executes, it takes a slightly different path. Trainable parameters allow adjusting the direction we take. By varying encoded data and model parameters, all possible paths create a smooth curved surface – a manifold.

The dance of the blind puppeteer

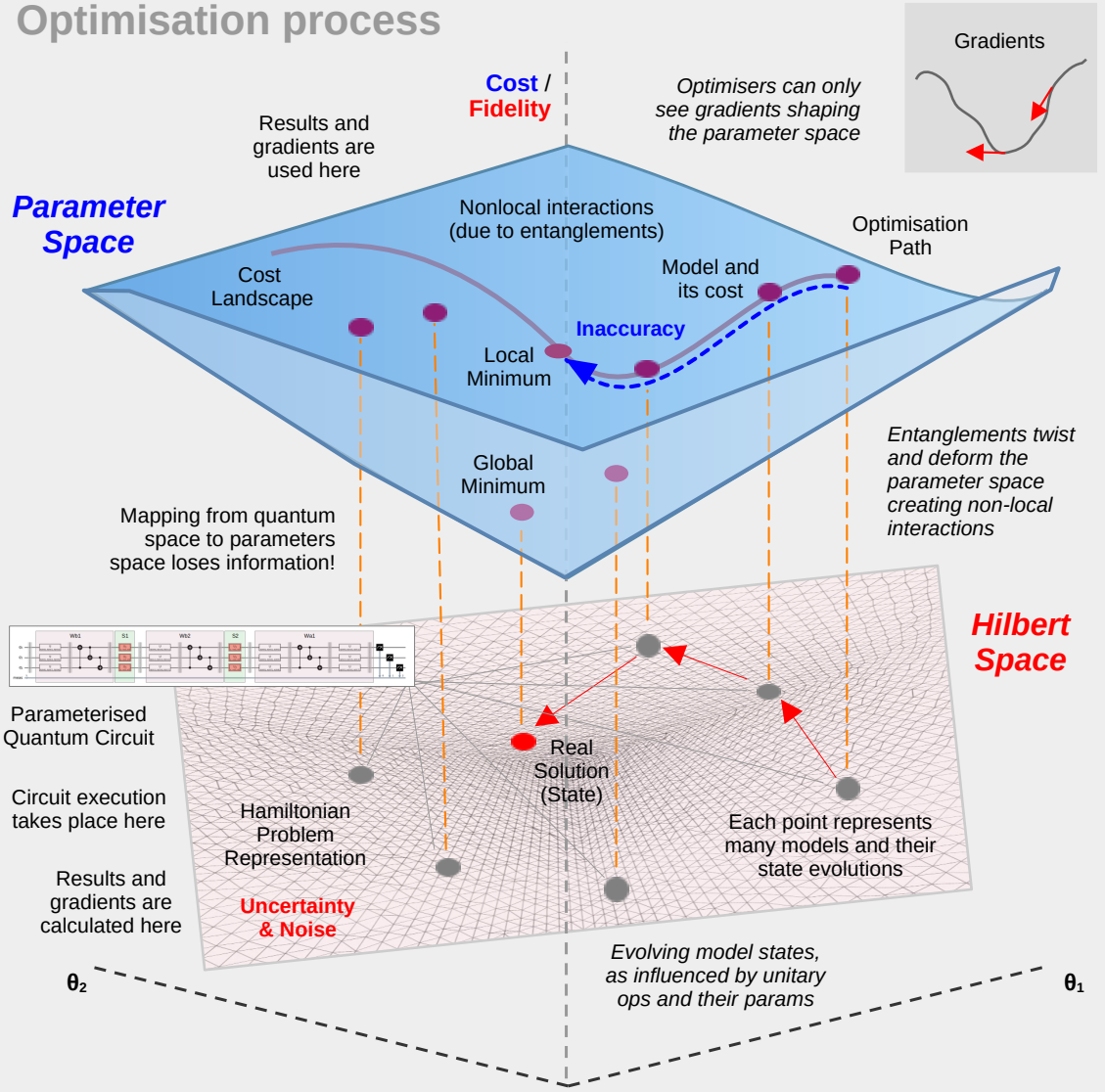
Parameter space
(dim = the number of params) is a classical multi-dim space of trainable gate parameters, which the optimiser navigates

Classical parameters
The optimiser can only manipulate the gate parameters but is not aware of their action on qubits states or their correlations via qubit entanglements.

Measurement
of individual qubits collapses their states, consequently projecting the circuit state onto classical outcomes, the only information reaching the optimiser. In the process we lose much of quantum information (e.g. phase).

Taking the blindfold from the puppeteer's eyes

Optimisation process



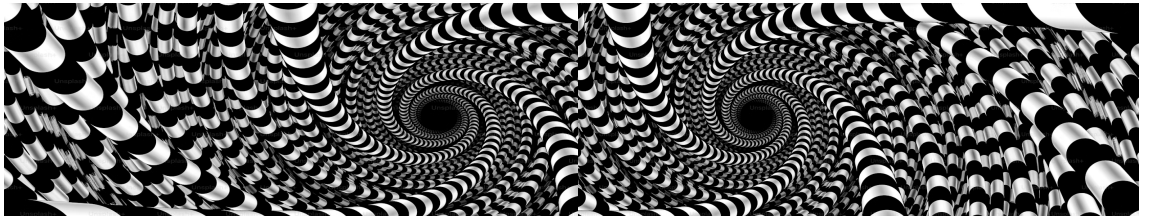
Hilbert state space
(dim ≈ $2^{\text{the number of qubits}}$) is the quantum realm where the models and their states evolve in response to unitary operations as defined by the circuit gates

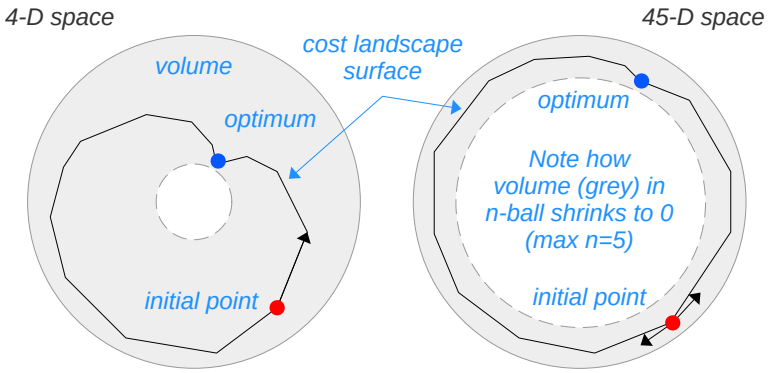
Data encoding
brings in classical data into the Hilbert space as unique and correlated quantum states during the model execution

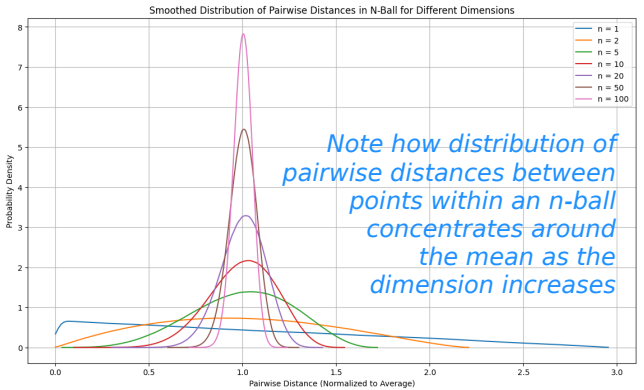
Entanglements
(CNOTs) create non-separable qubit states, which cause non-local interactions, and which alter the state space geometry, and as a side-effect, also the cost landscape

Circuit layers
determine the evolution of the quantum model's initial state into its final state during the circuit execution

The curse
of dimensionality







Cybulski, J.L., Nguyen, T., 2023. “Impact of barren plateaus countermeasures on the quantum neural network capacity to learn”, Quantum Inf Processing 22, 442.

Barren Plateaus (flattening of the cost landscape)

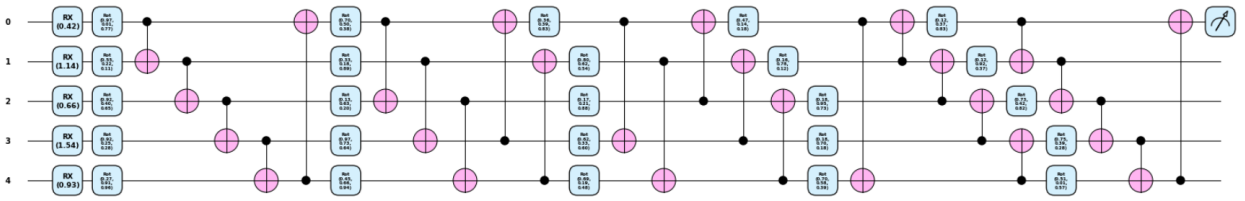
- All points within a high-D n-ball are near its surface, and distances between uniformly distributed points in such an n-ball become almost identical.
- In a quantum model with a high-D parameter space, the cost landscape is nearly flat, the situation called *barren plateau (BP)*.
- In high-D parameter space, models sampled by the optimiser are very sparse in both Hilbert space and parameter space.
- When BPs emerge, the optimiser struggles finding the optimum.
- Selecting the initial optimisation point far from the optimum (e.g. random) makes it even more difficult !

There are some well-known BP countermeasures

- use fewer qubits / layers / parameters
- use local cost functions (do not measure all qubits)
- use non-Euclidean metrics (e.g. Fisher Information Metric)
- beware of random params initialisation (and keep them small)
- use BP-resistant model design (e.g. layer-by-layer dev)
- use BP-resistant models (e.g. QCNNs)

Sample Model Training:

Estimate diabetes progression (data from scikit-learn)



Which estimator is better?
Which could still improve?

Would this change if we were running the model training on a quantum machine?

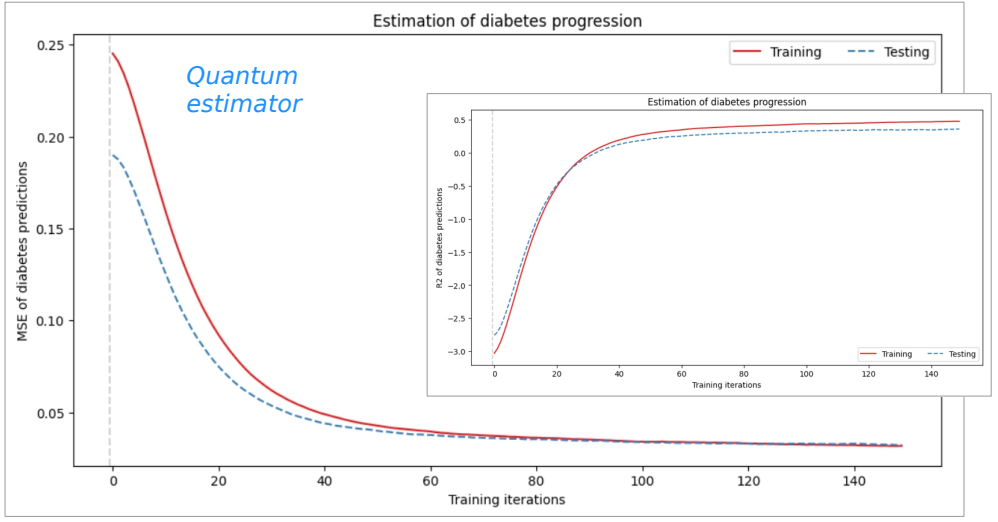
Quantum model training

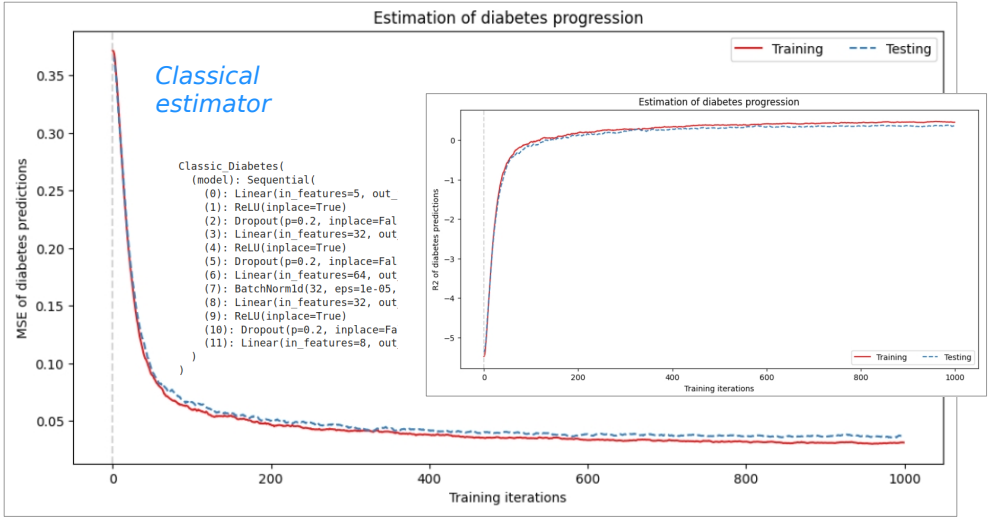
```
Model training started

    0 (000024 sec): Loss 0.2452   R2 -3.0295
    7 (000189 sec): Loss 0.0971   R2 -0.5967
   14 (000354 sec): Loss 0.0596   R2 0.0204
   21 (000519 sec): Loss 0.0499   R2 0.1802
   28 (000684 sec): Loss 0.0455   R2 0.2517
   35 (000848 sec): Loss 0.0421   R2 0.3077
   42 (001013 sec): Loss 0.0404   R2 0.3354
   49 (001178 sec): Loss 0.0388   R2 0.3618
   56 (001343 sec): Loss 0.0385   R2 0.3669
   63 (001507 sec): Loss 0.0371   R2 0.3904
   70 (001671 sec): Loss 0.0359   R2 0.4102
   77 (001835 sec): Loss 0.0347   R2 0.4293
   84 (002000 sec): Loss 0.0349   R2 0.4261
   91 (002164 sec): Loss 0.0343   R2 0.4368
   98 (002329 sec): Loss 0.0329   R2 0.4586
  105 (002493 sec): Loss 0.0324   R2 0.4673
  112 (002657 sec): Loss 0.0333   R2 0.4525
  119 (002822 sec): Loss 0.0313   R2 0.4859
  126 (002986 sec): Loss 0.0312   R2 0.4870
  133 (003151 sec): Loss 0.0316   R2 0.4811
  140 (003315 sec): Loss 0.0321   R2 0.4727
  147 (003479 sec): Loss 0.0308   R2 0.4935

Total training time: 3526s (00:58:46)
```

```
devices = cpu + lightning.qubit
samples = 296, features = 5, params = 75, epochs = 150
training: cost = 0.0306 @ 0141, r2 = 0.4977 @ 0141
testing:  cost = 0.0309 @ 0148, r2 = 0.3891 @ 0148
elapsed time = 3526sec (00:58:46)
```

challenge problem:
do this in your own time
in Qiskit or PennyLane

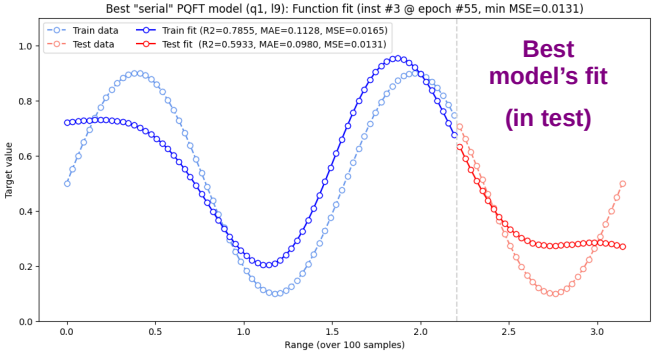
```
device = cpu
samples = 296, features = 5, params = 4721, epochs = 1000
training: cost = 0.0278 @ 0852, r2 = 0.5147 @ 0852
testing:  cost = 0.0304 @ 0980, r2 = 0.4708 @ 0980
elapsed time = 3sec (00:00:03)
```

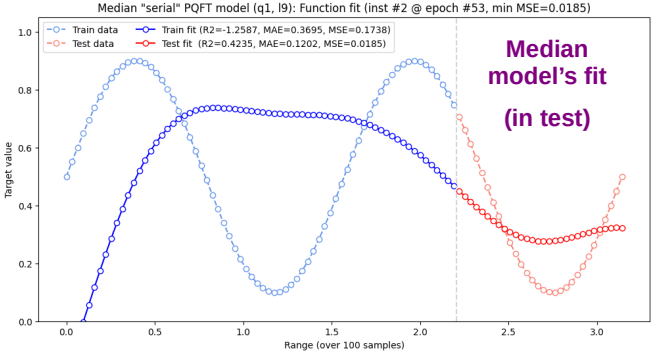


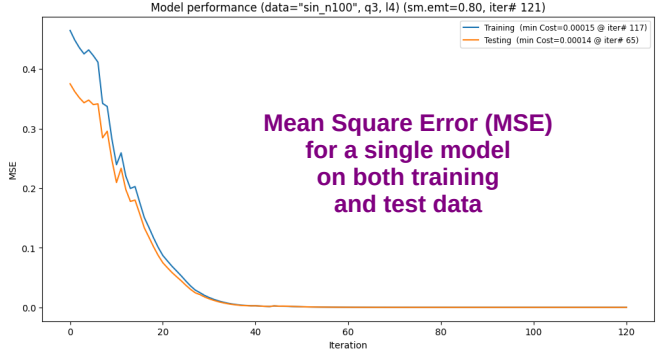


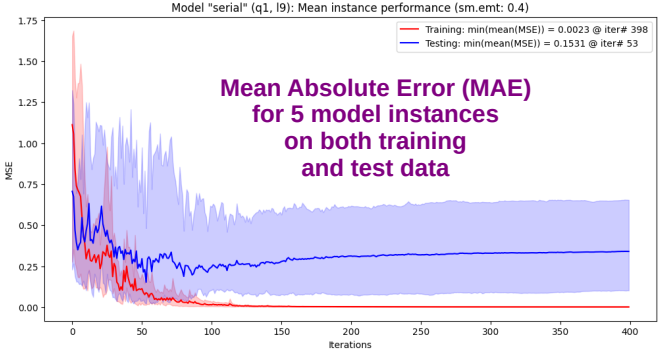
Quantum model performance:
Scoring a quantum model

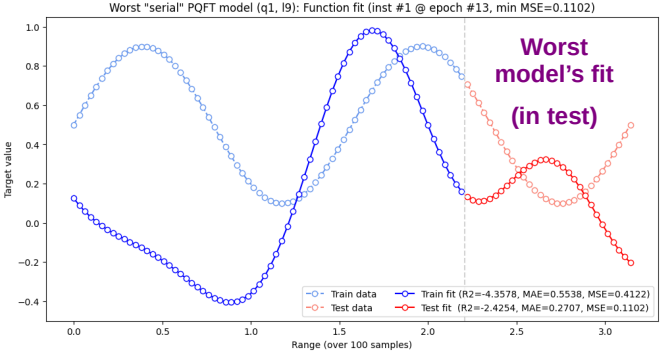
- Model training involves an optimizer, training data and a loss function, e.g. L2Loss (MSE).
- However, *several metrics may be needed to assess the model performance*, e.g. MSE, MAE or R^2, *to be calculated for training, validation and test data partitions*.
- At each optimisation step, the *model parameters should be saved for model scoring* on all data partitions (e.g. figure bottom-left).
- However, *quantum models are highly sensitive to their parameters initialisation*, therefore *performance of a single model run is not reliable*!
- So, we should *run multiple, differently initialised, instances of the same model* and analyse a distribution of their performance results.
- Here we present several (5) instances of the same model identically configured but differently initialised (figure bottom-middle).
- Set the model performance expectations by *indicating the model's fit to data*, depending on it best, median and worst instance performance (figures right).











Why are we getting errors?

The reasons we are getting errors (residuals) …

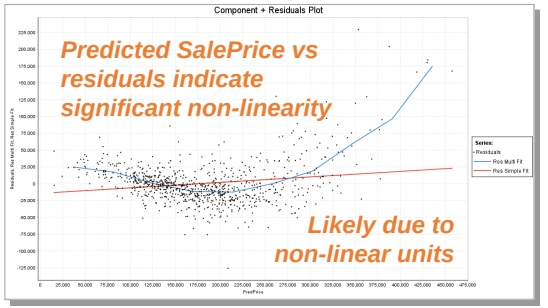
The quantum reasons:

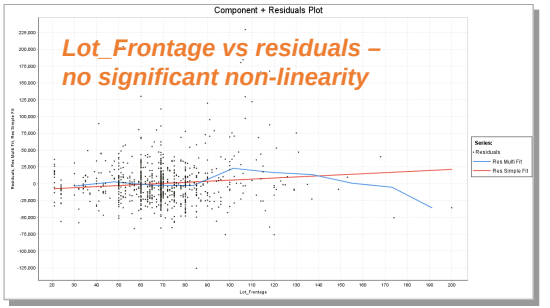
- There are problems with the model/ansatz design
  - not expressive enough / too simple, e.g. too few params
  - too complex, e.g. too many qubits / layers / params
  - does not fit data, e.g. temporal / spacial data
- There are problems with model optimization
  - Barren plateaus, i.e. vanishing gradients
  - Under / over training, i.e. too few / too many epochs
  - Bad initialisation, e.g. random (far from optimum)
  - Poor data encoding / lack of reuploading
  - Poor observables / measurement strategy
  - Poorly selected optimiser / cost function
  - Continuous cost (MSE) / nominal score (e.g. accuracy)
  - We lose precision / phase on measurement

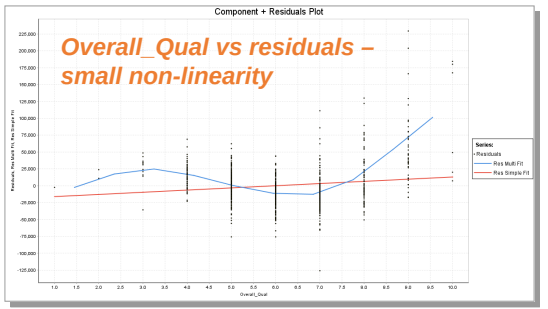
Classical / other reasons:

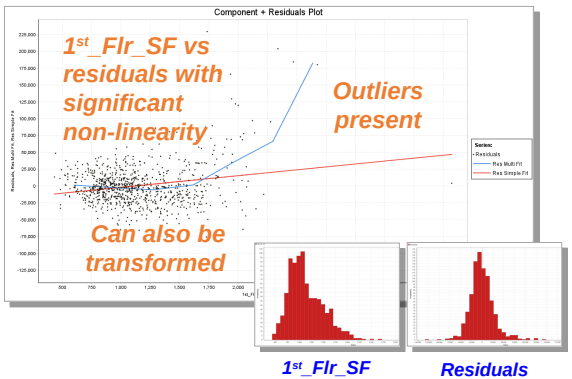
- Training data not representative (bad test results)
- Lack of cross-validation (you were lucky / or not)
- Poor data preparation (myth: prep not needed in Q)
- Presence of outliers / anomalies
- Other reasons

Understand your errors - so, experiment and measure!

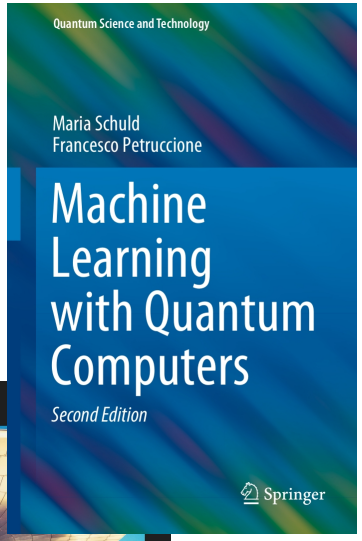


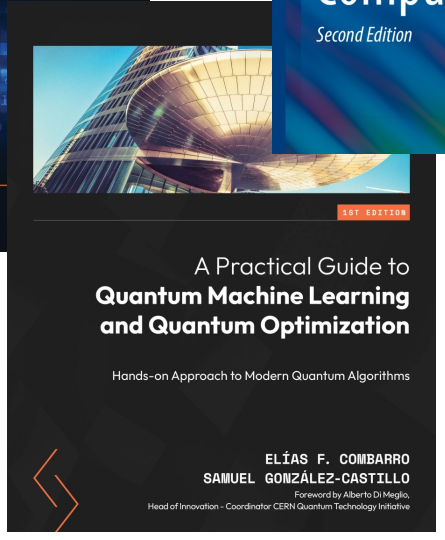






Recommended reading
on QML with Qiskit





Quantum computing with Qiskit

Ali Javadi-Abhari,[1] Matthew Treinish,[1] Kevin Krsulich,[1] Christopher J. Wood,[1] Jake Lishman,[2] Julien Gacon,[3] Simon Martiel,[4] Paul D. Nation,[1] Lev S. Bishop,[1] Andrew W. Cross,[1] Blake R. Johnson,[1] and Jay M. Gambetta[1]
[1]IBM Quantum, IBM T. J. Watson Research Center, Yorktown Heights, NY, 10598
[2]IBM Quantum, IBM Research Europe, Hursley, United Kingdom
[3]IBM Quantum, IBM Research Europe, Zürich, Switzerland
[4]IBM Quantum, IBM France Lab, Orsay, France

We describe Qiskit, a software development kit for quantum information science. We discuss the key design decisions that have shaped its development, and examine the software architecture and its core components. We demonstrate an end-to-end workflow for solving a problem in condensed matter physics on a quantum computer that serves to highlight some of Qiskit's capabilities, for example the representation and optimization of circuits at various abstraction levels, its scalability and retargetability to new gates, and the use of quantum-classical computations via dynamic circuits. Lastly, we discuss some of the ecosystem of tools and plugins that extend Qiskit for various tasks, and the future ahead.

arXiv:2405.08810v2 [quant-ph] 19 Jun 2024

I. INTRODUCTION

Quantum computing is progressing at a rapid pace, and robust software tools such as Qiskit are becoming increasingly important as a means of facilitating research, education, and to run computationally interesting problems on quantum computers. For example, Qiskit was

II. DESIGN PHILOSOPHY

We begin by discussing Qiskit's scope within the broader quantum computing software stack, as illustrated in Figure 1. Starting from a computational problem, a quantum algorithm specifies how the problem may be solved with quantum circuits. This step involves translating the classical problem to the quantum domain, for example Fermion to qubit mapping [34, 62]. Circuits at this level can be quite abstract, for example only specifying a set of Pauli rotations, some unitaries, or other high-level mathematical operators. Importantly, these abstract circuits are representable in Qiskit, which contains synthesis methods to generate concrete circuits from them. Such concrete circuits are formed using a standard library of gates, representable using intermediate quantum languages such as OpenQASM [31].

The transpiler rewrites circuits in multiple rounds of passes, in order to optimize and translate it to the target instruction set architecture (ISA). The word "transpiler" is used within Qiskit to emphasize its nature as a circuit-to-circuit rewriting tool, distinct from a full compilation down to controller binaries which is necessary for executing circuits. But the transpiler can also be thought of as an optimizing compiler for quantum programs.

The ISA is the key abstraction layer separating the hardware from the software, and depends heavily on the quantum computer architecture beneath. For example for a physical quantum computer based on superconducting qubits, this can include `CNOT`, $\sqrt{\text{X}}$ and `RZ`(θ) rotations. For a logical quantum computer, it can include joint Pauli measurements, magic state distillation, or other operations specific to the error correcting code [25]. Note that the ISA is often more than just a universal set of quantum gates, and can include `measure`, `reset` or `delay` operations, or classical control-flow such as `if/else`

Qiskit Machine Learning: an open-source library for quantum machine learning tasks at scale on quantum hardware and classical simulators

M. Emre Sahin,[1] Edoardo Altamura,[1] Oscar Wallis,[1] Stephen P. Wood,[2] Anton Dekusar,[3] Declan A. Millar,[4] Takashi Imamichi,[5] Atsushi Matsuo,[5] Stefano Mensa,[1,*] and Code contributors
[1]The Hartree Centre, STFC, Sci-Tech Daresbury, Warrington, WA4 4AD, United Kingdom
[2]IBM Quantum, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
[3]IBM Quantum, IBM Research Europe – Dublin, Ireland
[4]IBM Research – UK
[5]IBM Quantum, IBM Research – Tokyo, Tokyo 103-8510, Japan
(Dated: Friday 13th June, 2025)

We present Qiskit Machine Learning (ML), a high-level Python library that combines elements of quantum computing with traditional machine learning. The API abstracts Qiskit's primitives to facilitate interactions with classical simulators and quantum hardware. Qiskit ML started as a proof-of-concept code in 2019 and has since been developed to be a modular, intuitive tool for non-specialist users while allowing extensibility and fine-tuning controls for quantum computational scientists and developers. The library is available as a public, open-source tool and is distributed under the Apache version 2.0 license.

arXiv:2505.17756v1 [quant-ph] 23 May 2025

I. INTRODUCTION

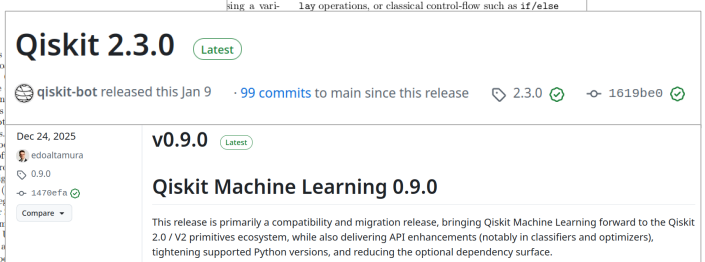
The convergence of quantum computing and machine learning promises a prospective shift in both research and industry. Quantum machine learning (QML) leverages the principles of quantum mechanics to potentially enhance or accelerate classical machine learning algorithms, opening new frontiers in fields ranging from materials science to finance. As the field of QML matures, there is a growing need for accessible and powerful software tools that bridge the gap between theoretical QML algorithms and their practical implementation on emerging quantum hardware and simulators.

Qiskit Machine Learning (ML)[1], an open-source module within the Qiskit framework [1], addresses this need by providing a comprehensive and user-friendly platform for exploring the exciting landscape of QML. Built on core Qiskit elements such as primitives, it combines quantum circuit design, simulation, and execution to deliver cutting-edge QML algorithms. Users can experiment with quantum enhancements to established methods, such as quantum kernels for Support Vector Machines, or explore new, fully quantum approaches. Its tight integration with Python and reliance on widely used libraries like NumPy [2] and scikit-learn [3] make it accessible to practitioners in diverse fields, from engineering to the life sciences. It also includes a dedicated API connector to PyTorch [4] for neural network-based algorithms, seamlessly bridging quantum circuits with modern deep learning frameworks.

Qiskit ML is freely distributed under the Apache 2.0 license, encouraging community participation and open collaboration. Moreover, it sets itself apart from other platforms like PennyLane [5] in its approach to quantum hardware usage. Specifically, Qiskit ML's architecture is deliberately designed to handle quantum hardware workloads, while also allowing experimentation with state-of-the-art classical simulators and models of emulated hardware noise from near-term devices. Moreover, it is designed to be modular and extensible, making the addition of new quantum algorithms or building upon existing ones straightforward. Supported by extensive educational resources and tutorials, Qiskit ML stands at the forefront of QML research, helping students, scientists and developers worldwide investigate the applications of quantum computing for machine learning.

ML predominantly depends on Qiskit's primitives. It also interfaces with classical machine learning frameworks such as scikit-learn and Python numerical-core libraries like NumPy, enabling a continuous integration of classical and quantum machine learning techniques. Additionally, the models follow SciPy's structural foundation, and there is functionality for integrating neural networks with PyTorch to support the design, training, and inference of hybrid quantum-classical models.

* stefano.mensa@stfc.ac.uk
[1] github.com/qiskit-community/qiskit-machine-learning



Qiskit 2.3.0 Latest

qiskit-bot released this Jan 9 · 99 commits to main since this release · 2.3.0 · 1619be0

Dec 24, 2025
edoaltamura
0.9.0
1470efa
Compare

v0.9.0 Latest

Qiskit Machine Learning 0.9.0

This release is primarily a compatibility and migration release, bringing Qiskit Machine Learning forward to the Qiskit 2.0 / V2 primitives ecosystem, while also delivering API enhancements (notably in classifiers and optimizers), tightening supported Python versions, and reducing the optional dependency surface.

Summary and thank you!

- QML is an intersection of QC x ML x Maths
- The most common approach to PQC training are VQAs
- Quantum encoding is the key to success (but full of traps)
- Measurement of circuits requires interpretation of results
- Quantum circuit design needs to consider what happens in Hilbert space and what the optimizer does in classical parameter space, both are in conflict
- Training of the hybrid quantum-classical circuit relies on a classical optimizer, and its execution on a quantum machine
- Backpropagation does not work on quantum machines, due to: measurement collapse and no-cloning theorem
- Quantum models are highly sensitive to initialisation, so their performance needs to be assessed across different model instances
- Dimensionality of Hilbert space and parameter space promotes the circuit expressivity, yet, hampers the circuit trainability
- Qiskit QML models utilize PQCs
- Qiskit provides tools for data encoding, ansatz design and measurement
- Qiskit provides powerful runtime framework for training sampling (classification) and estimation models, equipped with noise suppression and mitigation tools
- In VQA an optimiser is totally blind to what happens in the Hilbert space
- Still we are able to create and train quantum models!

Q&A

Available resources, see:
ironfrown (Jacob L. Cybulski, Enquanted)
https://github.com/ironfrown/

This presentation has been released under the Creative Commons CC BY-NC-ND license, i.e.

BY: credit must be given to the creator.
NC: Only noncommercial uses of the work are permitted.
ND: No derivatives or adaptations of the work are permitted.

Images from Unsplash and Wikipedia
Enquanted is being somewhere in-between Enchanted and Entangled

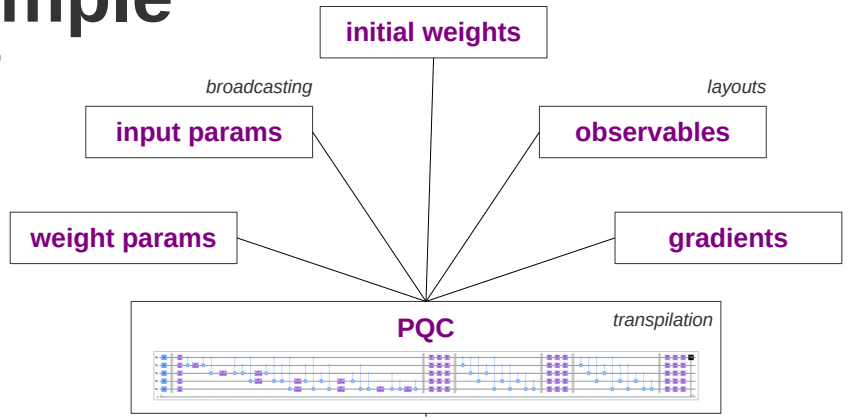
Appendix

More of who and what ...

Category Name	Company Name	Sample Project	Continent or Region
Defense & Security	SandboxAQ	GPS-denied navigation (MagNav) and PQC	North America
	Infleqtion	Edge QAI for RF receivers and signal intelligence	North America
	ID Quantique	Quantum Key Distribution and secure backbones	Europe
	QuSecure	Post-Quantum Cryptography (PQC) orchestration	North America
	TII	Sovereign quantum cryptography and security	Arab World
	Q-CTRL	Quantum-enhanced navigation and mission planning	Australia
Science & Research	Xanadu	Differentiable programming and QNN frameworks	North America
	Pasqal	Graph Neural Networks (GNN) for climate and physics	Europe
	NVIDIA	GPU-accelerated QML simulation for field theory	North America
	QuantX Labs	High-precision timing and quantum sensor	Australia
	Google Quantum AI	Quantum Reinforcement Learning and Field Echoes	North America
	SAQuTI	Environmental monitoring and biodiversity analysis	Africa
Sovereign AI Hubs	QpiAI	QAI platform for life sciences and finance	India
	TCS	Hybrid QAI algorithms for global retail chains	India
	Terra Quantum	Hybrid QML for enterprise-grade applications	Europe
	NEC Corporation	5G/6G network traffic optimization	APAC
	QuantumNexis	QAI-powered healthcare analytics platforms	Arab World
	CSIRO	Health, space, mining and defense science	Australia

Training a simple
TS Qiskit estimator

Qiskit ***Optimiser*** provides function ***fit*** which executes a training loop, performing: a *forward* pass which applies the model with its current parameters to training data, *loss function*, and a *backward* pass to improve the model parameters.



Regressor starts with the model's *initial weights*. It then passes the current parameter values (inputs and weights) to the *Estimator* and receives back the observed expectation values and their gradients, which can be used by an *optimiser* to define the overall cost landscape and determine the next step in the circuit weights optimisation.

Via the *Optimiser* *Regressor* invokes a *callback function* to log the current weights and cost.

Dataset is to be prepared, cleaned and partitioned for training and testing.

Estimator creates the physical circuit using the *observables*, *input parameters* and *weight parameters*, and the *gradient method* *used in the calculation of expectation values.* It then executes the circuit by relying on a hardware specific *estimator primitive*. It returns the calculated expectation values.

training log

```
Model training started
(00:00:00) - Iter#:   0 / 500, Cost: 0.238564
(00:00:07) - Iter#:  50 / 500, Cost: 0.162685
(00:00:14) - Iter#: 100 / 500, Cost: 0.126066
(00:00:21) - Iter#: 150 / 500, Cost: 0.073866
(00:00:29) - Iter#: 200 / 500, Cost: 0.053152
(00:00:36) - Iter#: 250 / 500, Cost: 0.038513
(00:00:43) - Iter#: 300 / 500, Cost: 0.033054
(00:00:50) - Iter#: 350 / 500, Cost: 0.029146
(00:00:58) - Iter#: 400 / 500, Cost: 0.027865
(00:01:05) - Iter#: 450 / 500, Cost: 0.026759

Total time 00:01:12, min Cost=0.026013
```