

Secrets revealed in this session:

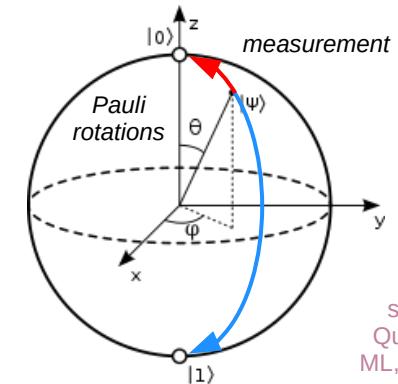
**To improve understanding of
VQAs and skills in building
quantum machine learning
models and their optimisation**



QML and its aims
Parameterised circuits
Variational quantum algorithms
Data encoding / angle encoding
State measurement
Ansatz design and training
Model geometry and gradients
Parameters optimisation
Curse of dimensionality
QML readings
Qiskit demo and tasks (TS forecasting)
Summary and Q&A

An introduction to Quantum Machine Learning in Qiskit

Jacob L. Cybulski
Enquanted, Australia



We will assume
some knowledge of
Quantum Computing
ML, Qiskit and Python

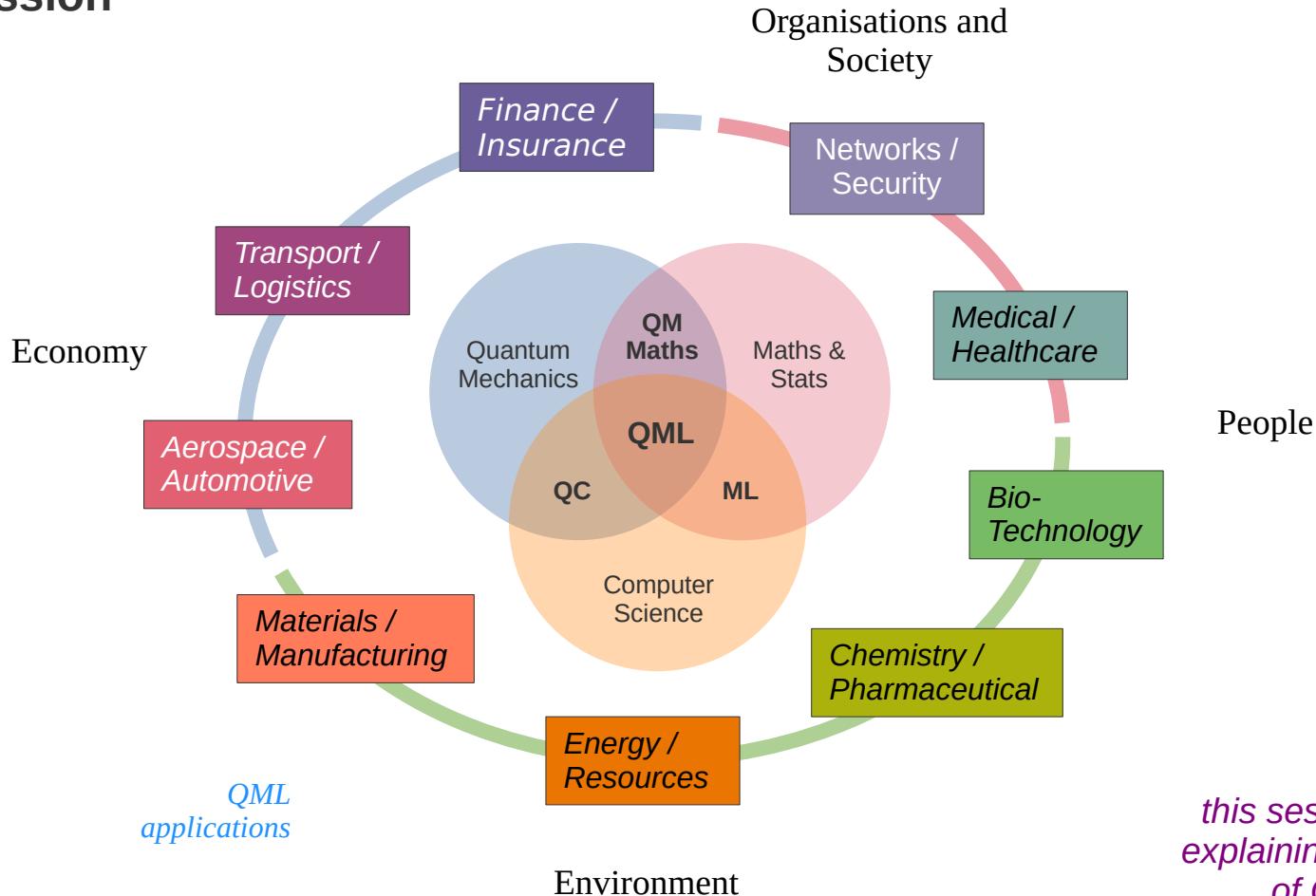
Quantum ML

aims of this session

Jacob L. Cybulski, Quantum Business Series (Deakin, RMIT, ACS, Warsaw School of Economics)
Jacob L. Cybulski, Quantum Computing Intro Series (SheQuantum, Assoc of Polish Profs in Australia)
2021-2025



Jacob Cybulski, Founder
Enquantum, Australia

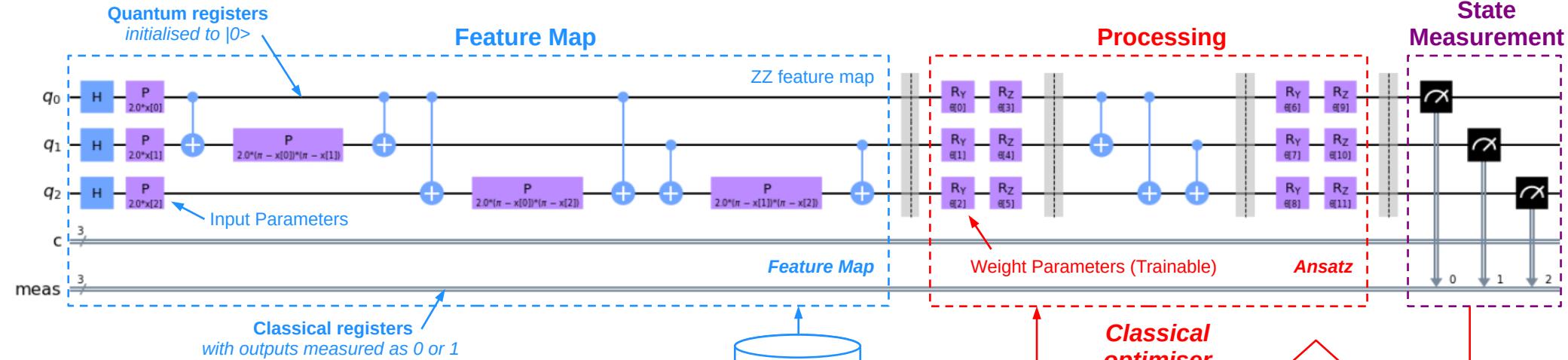


*this session aims at
explaining the nature
of QML models*

Parameterised Quantum Circuits and Variational Quantum Algorithms

Variational quantum circuits are not executable!

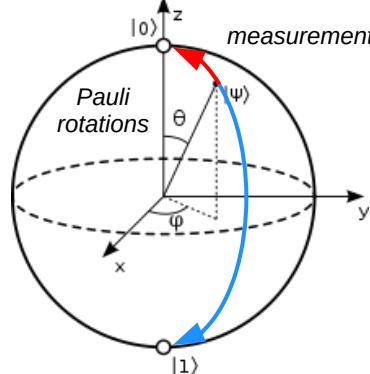
They must first be instantiated, i.e. all of their input and weight parameters must be assigned values!



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)

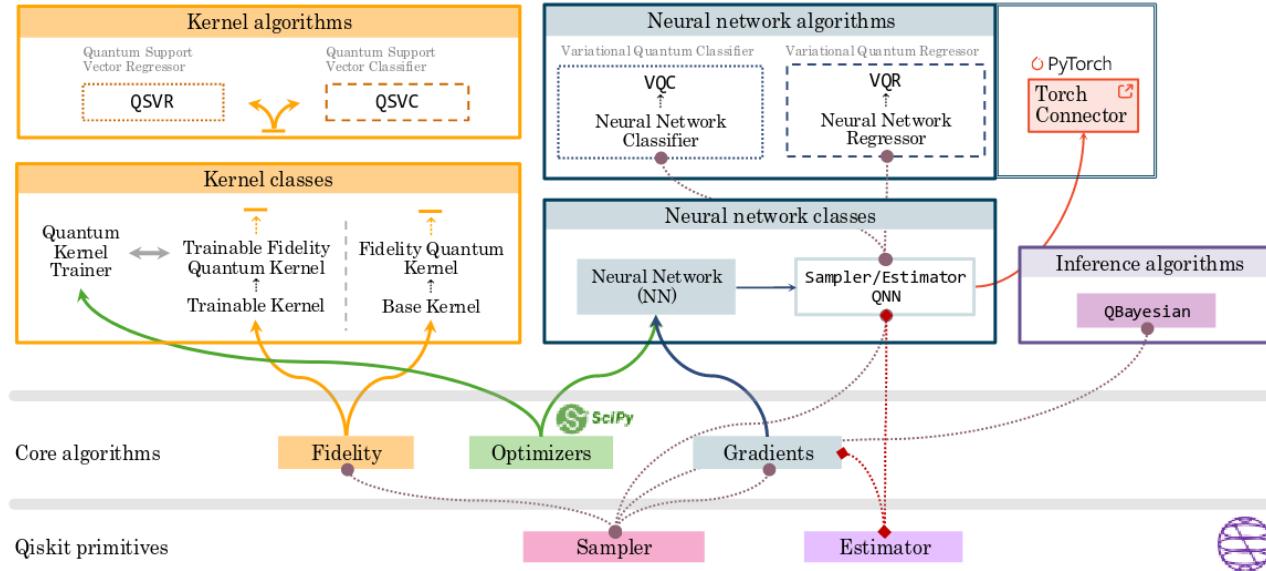


Classical input data is encoded into the feature map's parameters, setting the model's initial quantum state.

The quantum state is altered by an ansatz, of parameterised gates (operations), which are trained by an optimiser

The final quantum state of the circuit is then measured and interpreted as the model's output in the form of classical data.

Qiskit ML Resources



Qiskit ML models and related algorithms:

- Quantum Neural Networks (QNN, VQC/R, QCNN, qGAN)
- Quantum Kernel Methods (Feature Maps, Estimators)
- Quantum Support Vector Machines (QSVM, QSVC/R)
- Quantum Bayesian Modelling (Qbayesian)
- Quantum Kernel Principal Components Analysis (QKPCA)
- Quantum Clustering Algorithms (QCA k-NN, DQC)
- Quantum Optimisation Algorithms (QAOA, QUBO)

Sahin, M.E., Altamura, et al., 2025. Qiskit Machine Learning: an open-source library for quantum machine learning tasks at scale on quantum hardware and classical simulators. ArXiv.2505.17756.

Olivier Ezratty, Understanding Quantum Technologies (2024)

Other open source or published algorithms

- Quantum Fourier Analysis (QFT, QFFT)
- Quantum Sequence Models (QRNN, QLSTM, QGRU)
- Quantum Annealing / Quantum Adiabatic Algorithm (QAA)
- Quantum Boltzmann Machines (QBM, QRBM))
- Quantum Self-Attention and Transformers
- Quantum Random Forest (QRF)
- Quantum k-Nearest Neighbour (QkNN)
- Quantum Hopfield Associative Memory (QHAM)
- Quantum Reinforcement Learning (QRL)
- Quantum Genetic Algorithms (QGA)

Data encoding strategies

Data encoding

There are many methods of data embedding, such as:
the *basis*, *angle*, *amplitude*, *QRAM*, ... encoding,

In this workshop we will rely on *angle encoding* realised as qubit state rotation by the angle defined by the data.

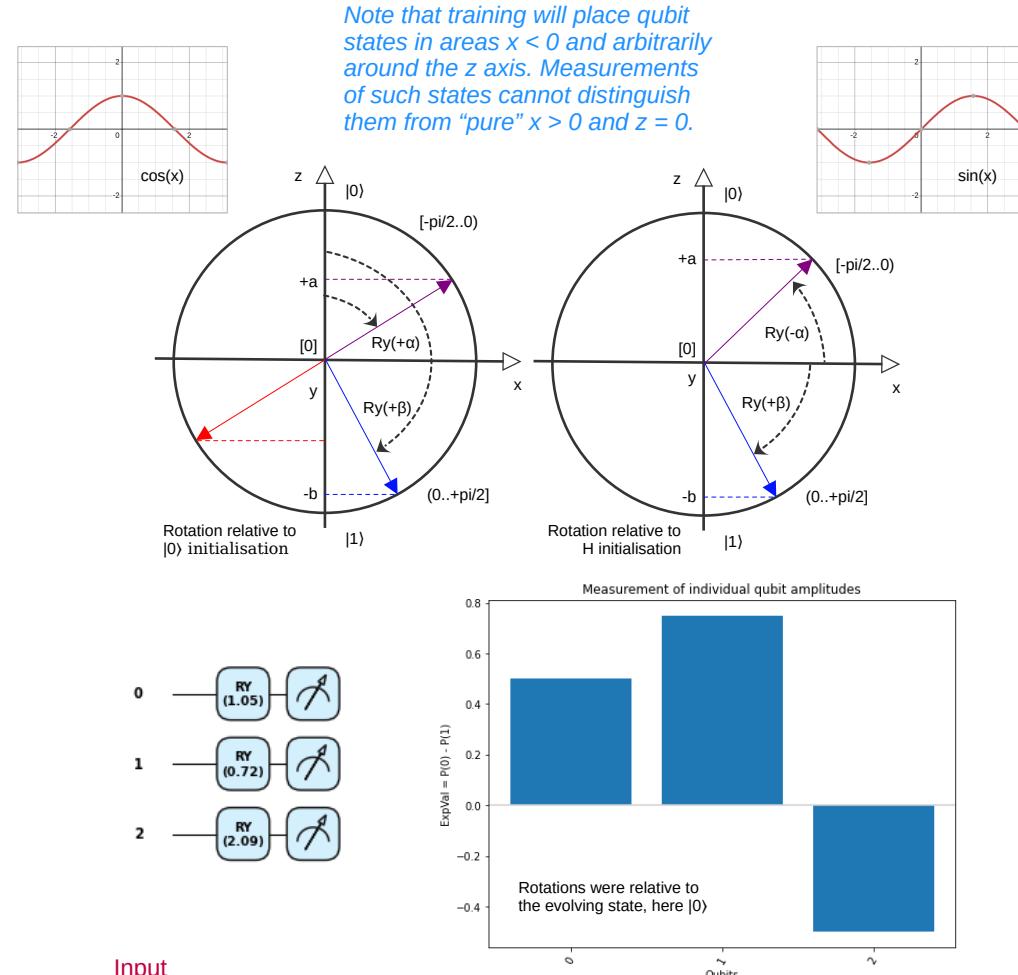
The rotation operators are always available in a quantum platform API, e.g. *Rx*, *Ry*, *Rz*, *P* or *U* (*xyz*).

Typically, the encoding rotation is performed around x or y axis, or both (allowing two values per qubit).

Rotations are *relative to a specific qubit state*, commonly starting at $|0\rangle$ state, or $(|0\rangle+|1\rangle)/\sqrt{2}$, which require qubits to be initialised in these states.

The encoded value could be represented either by the *angular rotation*, or the *amplitude* of the qubit projective measurement (Z).

Input data can also be repeatedly encoded and spread around the circuit, which is called *data reuploading*, and which is known to improve the model performance.



Input

Values entered:
Ry angles used:

`[np.arccos(0.5), np.arccos(0.75), np.pi-np.arccos(0.5)]
[1.047, 0.723, 2.094]`

Measurements

Probabilities:
Amplitudes:

`[[0.25, 0.75], [0.562, 0.438], [0.25, 0.75]]
[0.5, 0.75, -0.5]`

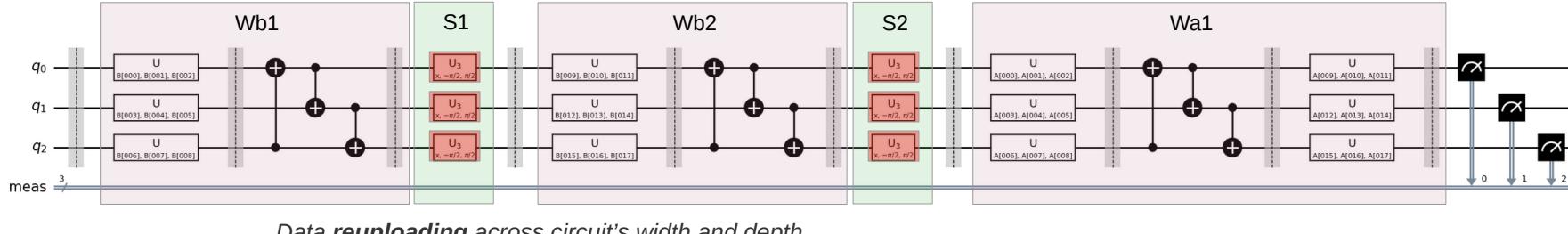
Ansatz design and training

A sample curve fitting model ...

Beware that
adding qubits adds
parameters and entanglements!

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

*Encoding of classical data in a quantum circuit is
not what our ML experience tells us about **inputs** !*



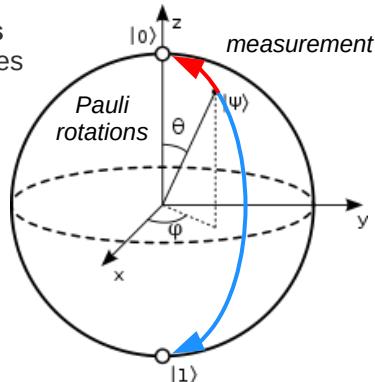
feature maps vary in:
structure and function (!!?)

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 $R(x, y, z)$ and CNOT gates
(rotation) (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-Leibnner 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)

Commonly used measurements and interpretation

Quantum circuits can be measured in many ways, e.g.

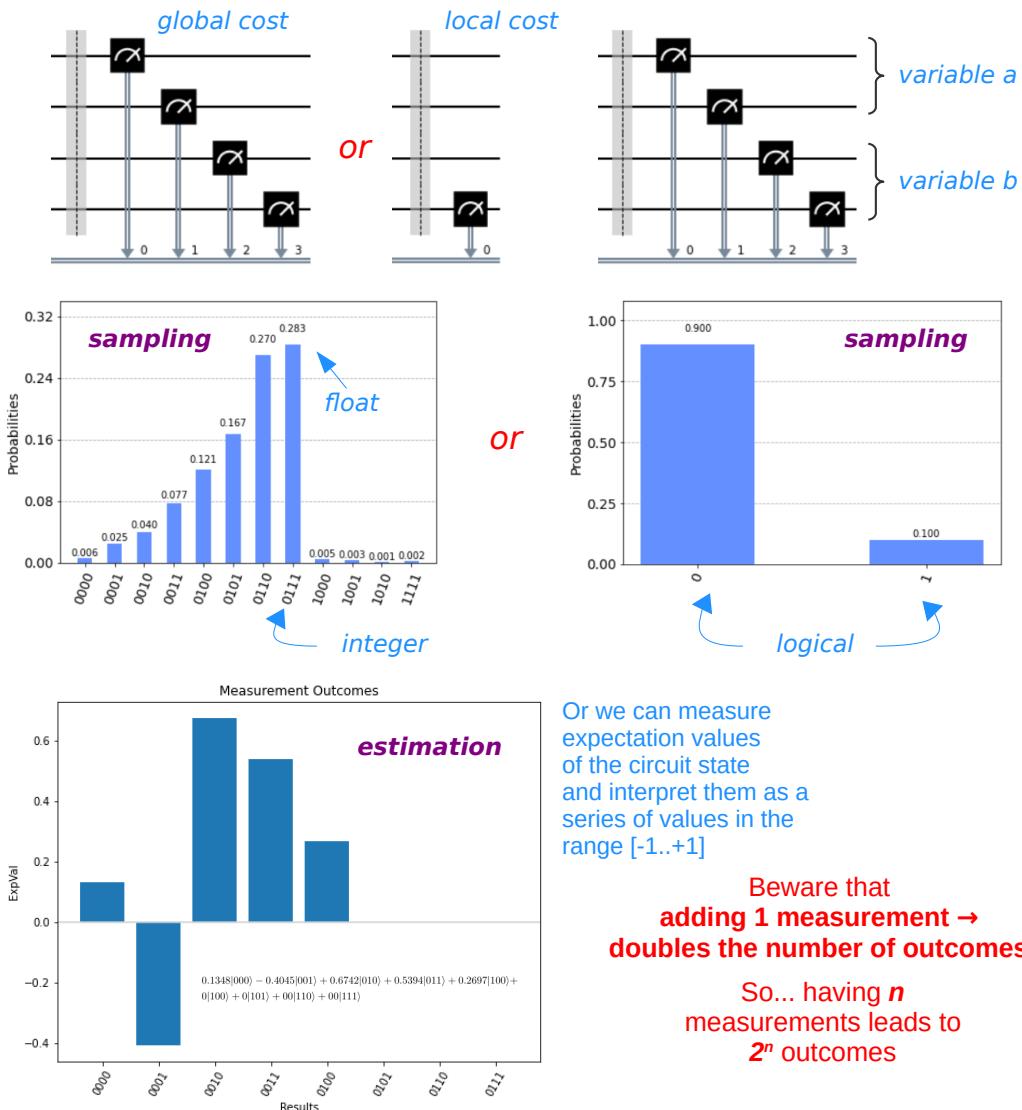
- all qubits (global cost / measurement)
- a few selected qubits (local cost / measurement)
- groups of qubits (each as a variable value)

And received in many different formats, e.g.

- as counts of outcomes (repeated measurements)
- as probabilities of outcomes (e.g. $P(|0111\rangle)$)
- as Pauli expectation values (i.e. of eigenvalues)
- as expectation of interpreted values (e.g. 0 to 15)
- as variance, etc.

Repeated measurement can be interpreted as outcomes of different types, e.g.

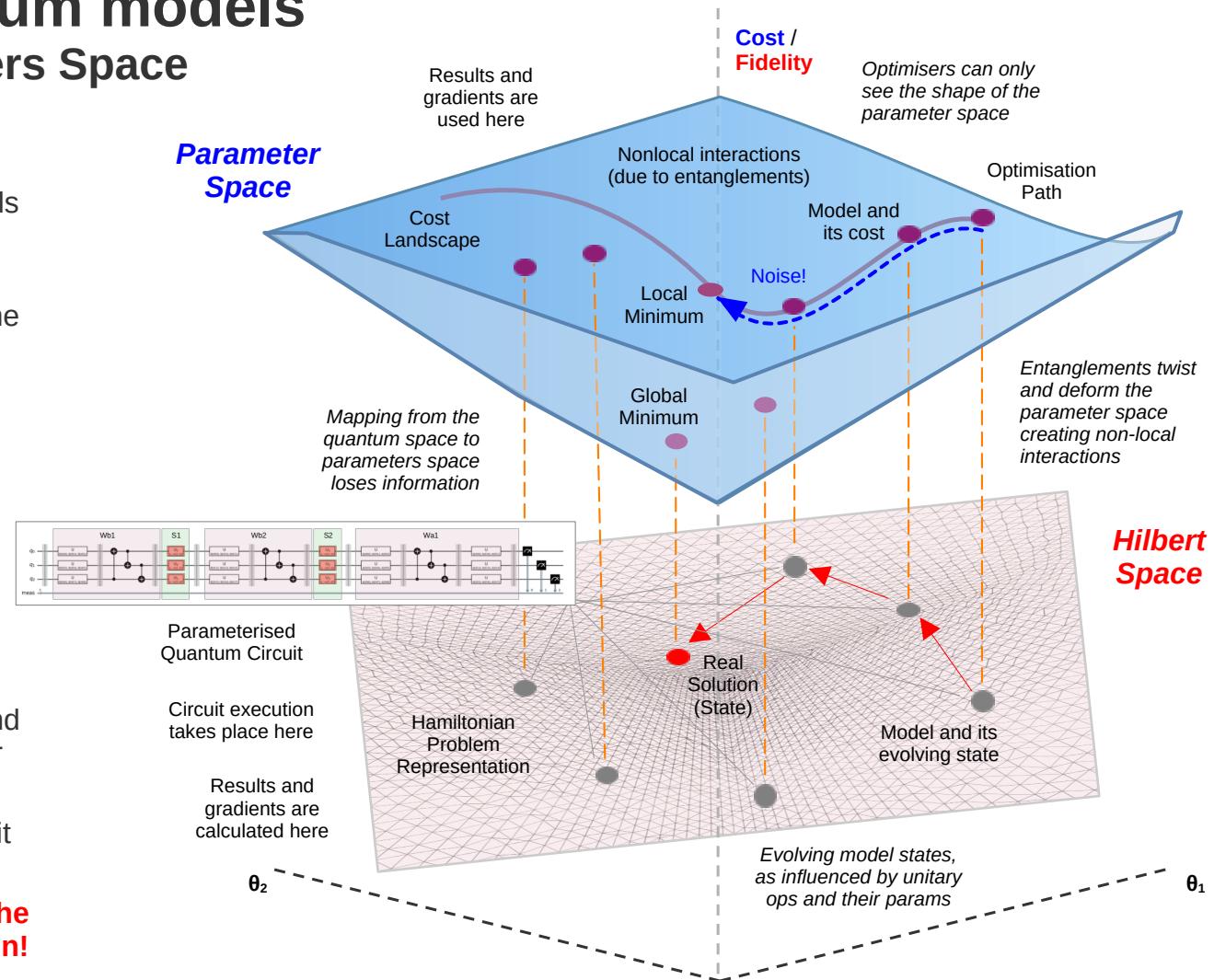
- as a probability distribution (as is)
- as a series of values (via expvals)
- as a binary outcome:
single qubit measurement or parity of kets
- as an integer:
most probable ket in multi-qubit measurement
- as a continuous variable:
probability of the selected ket (e.g. $|0^n\rangle$)



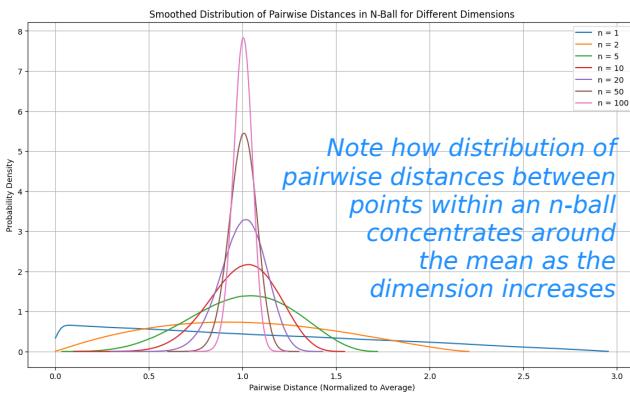
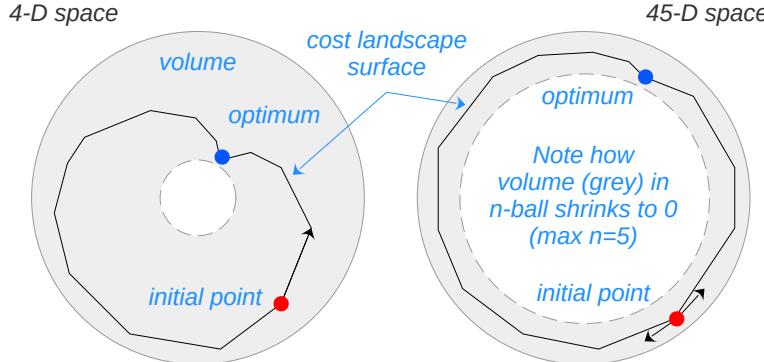
Working with quantum models

Hilbert Space vs Parameters Space

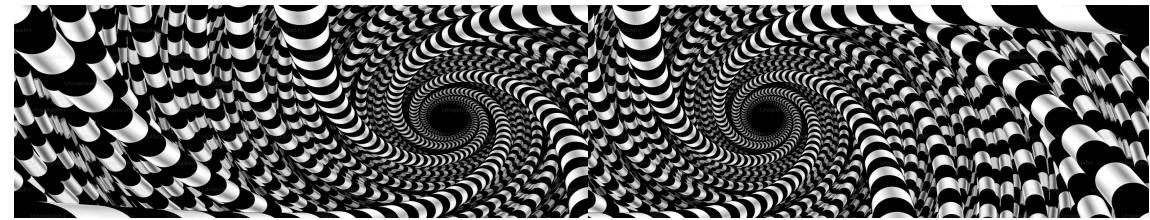
- **Hilbert state space** (dim = the number of qubits) is the quantum realm where the models and their states evolve in response to unitary operations as defined by the circuit gates
- **Data encoding** brings in classical data into the Hilbert space as unique and correlated quantum states during the model execution
- **Layers of circuit gates** determine the evolution of the quantum model's initial state into its final state during the circuit execution
- **Trainable parameter space** is a classical multi-dimensional space of circuit gate parameters, which the optimiser navigates
- **Entanglements** (defined by CNOTs) create and correlate non-separable qubit states, which alter the parameter space geometry, and also the cost landscape used by the optimiser
- **Measurement** of individual qubits collapses their states, consequently projecting the circuit state onto classical outcomes
- **The mapping from the quantum space to the classical parameter loses some information!**



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.



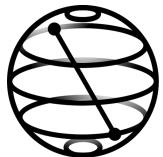
Barren Plateaus (too many dimensions)

- Pairwise distances between uniformly distributed points in high-dimensional space become (almost) identical, and the surface of such a space is almost flat (n -ball value is near its surface).
- 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 optimisation initial 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)

Qiskit QML Workshop



Why Qiskit?

- Accessible from *Python*, *Rust*, *C++* and more...
- Has a standard set of *quantum state operations*
- Supports creation of flexible QML *algorithms*
- Executes on *simulators* and *quantum hardware*
- Supports hardware *accelerators* (e.g. *GPUs*)
- Provides tools for *error mitigation*
- Utilises variety of *quantum gradients models*
- Supports *hybrid quantum-classical models*
- Provides many QML models, e.g. *QNNs*, *QCNN*, *QAE*, *QSVM* and *Bayesian models*
- Can be extended with *PyTorch* and *TensorFlow*
- Among quantum SDKs, it is *the best performer*
- It is largely *hardware agnostic via vendor backends*
- Supports *IBM quantum backend and runtime*
- It is *complex* and its *core design changes too often!*

Qiskit QML tasks (time series forecasting):

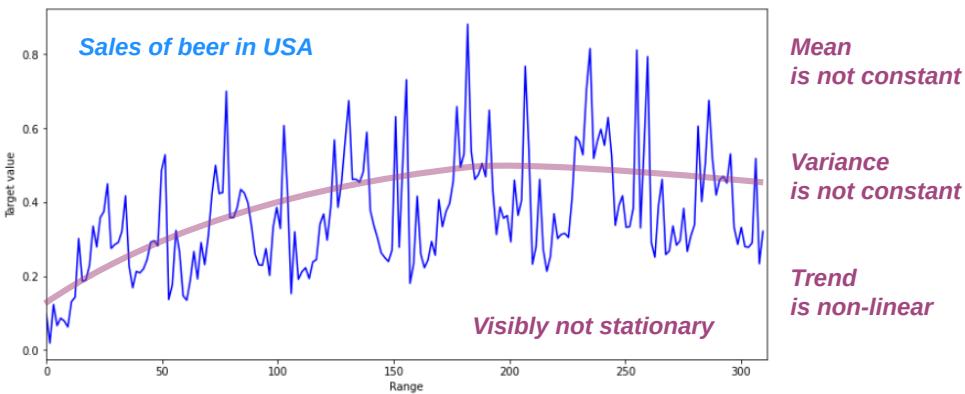
- Add ML 0.8.3 package to Qiskit 1.4.4 (Python 3.11)
- Create standard and custom models to fit simple data
- Learn the interaction b|n estimator and regressor
- Explore the impact of ansatz structure on performance
- Explore the impact of observables on performance
- Explore the impact of optimiser on performance
- **Challenge:** Apply your skills to chaotic data
- **Reflection:** Refine your QML development process

Key takeaways:

- Plan model development, tests and experiments
- Data encoding is crucial to model performance
- Carefully consider your quantum model initialisation
- More params and entanglements improve *expressivity*
- More params and entanglements reduce *trainability*
- Dealing with *the curse of dimensionality*
- High dimensional parameter space upsets even non-gradient optimisers due to *model sparsity*
- More training often does not eliminate problems!
- Selection of appropriate optimisers, observables and custom models, may be necessary to break the performance swamp

QML for time series analysis

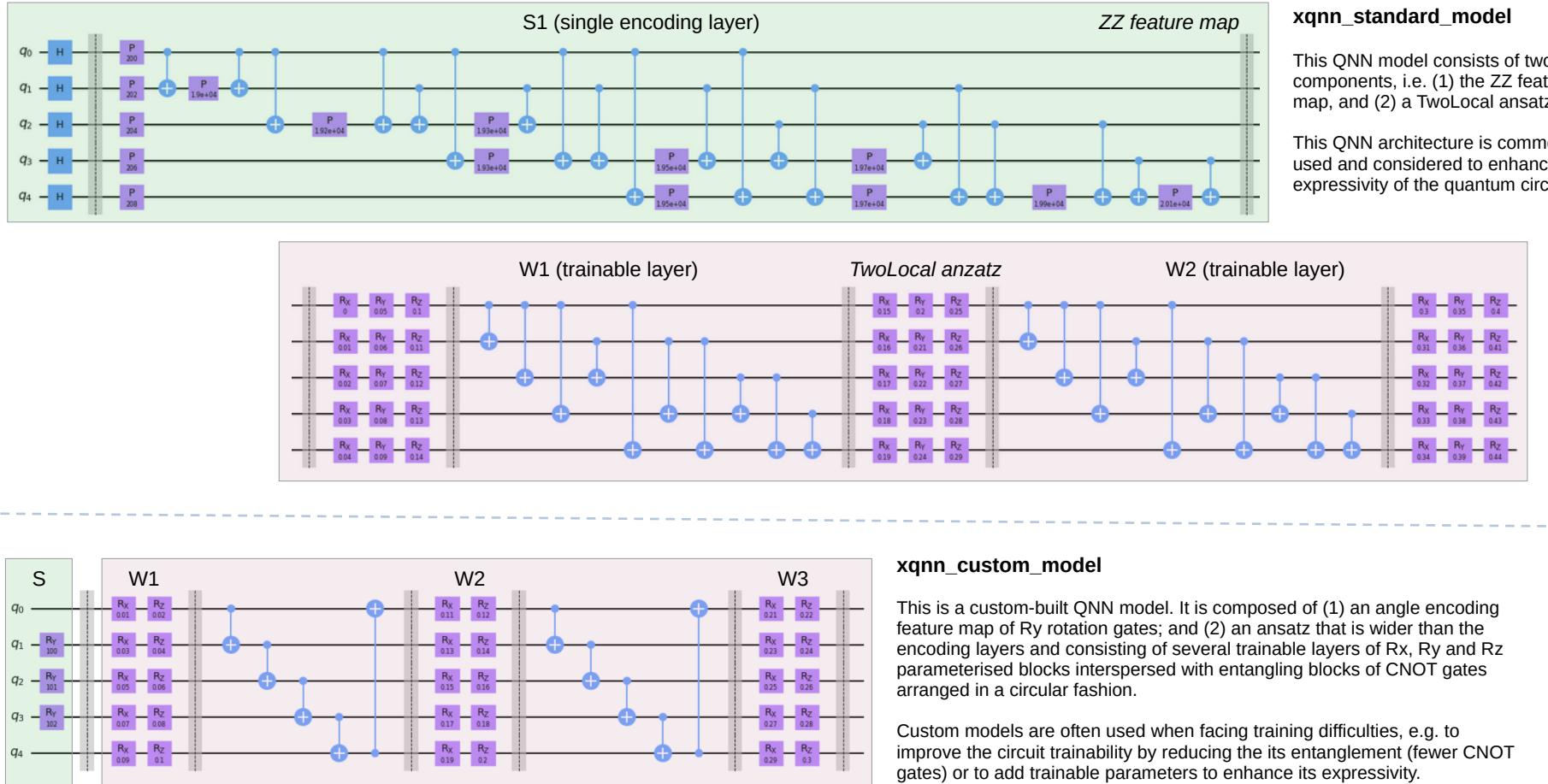
- Time series (TS) analysis aims to *identify patterns* in historical time data and to *create forecasts* of what data is likely to be collected in the future
- *Many TS applications*, including heart monitoring, weather forecasts, machine condition monitoring, etc.
- Time series can be *univariate* or *multivariate*
- Time series often show *seasonality* in data, i.e. some patterns repeating over time



Quantum time series analysis is hard!

- TS values are dependent on the preceding values!
- Distinction between consecutive TS values is small!
- There are several different types of TS models, e.g.
 - The first group are *curve-fitting models*, which are trained to fit a function to a sample of data points, to predict data values at specific points in time
 - The second group are *forecasting models*, which are trained to predict future data points from their preceding temporal context (a fixed-size window sliding over TS)
- Majority of statistical forecasting methods require *strict data preparation*, such as dimensionality reduction, TS aggregation, imputation of missing values, removal of noise and outliers, adherence to normality and homoskedasticity, they need to be stationary
- QML methods do not have such strict requirements, and are promising for effective time series analysis and forecasting!

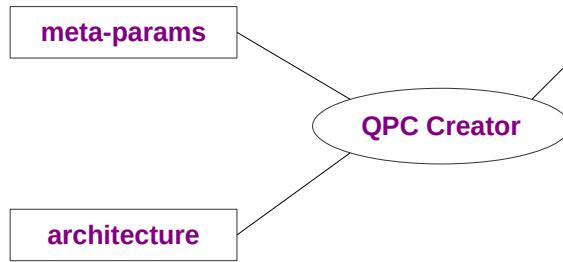
Forecasting models



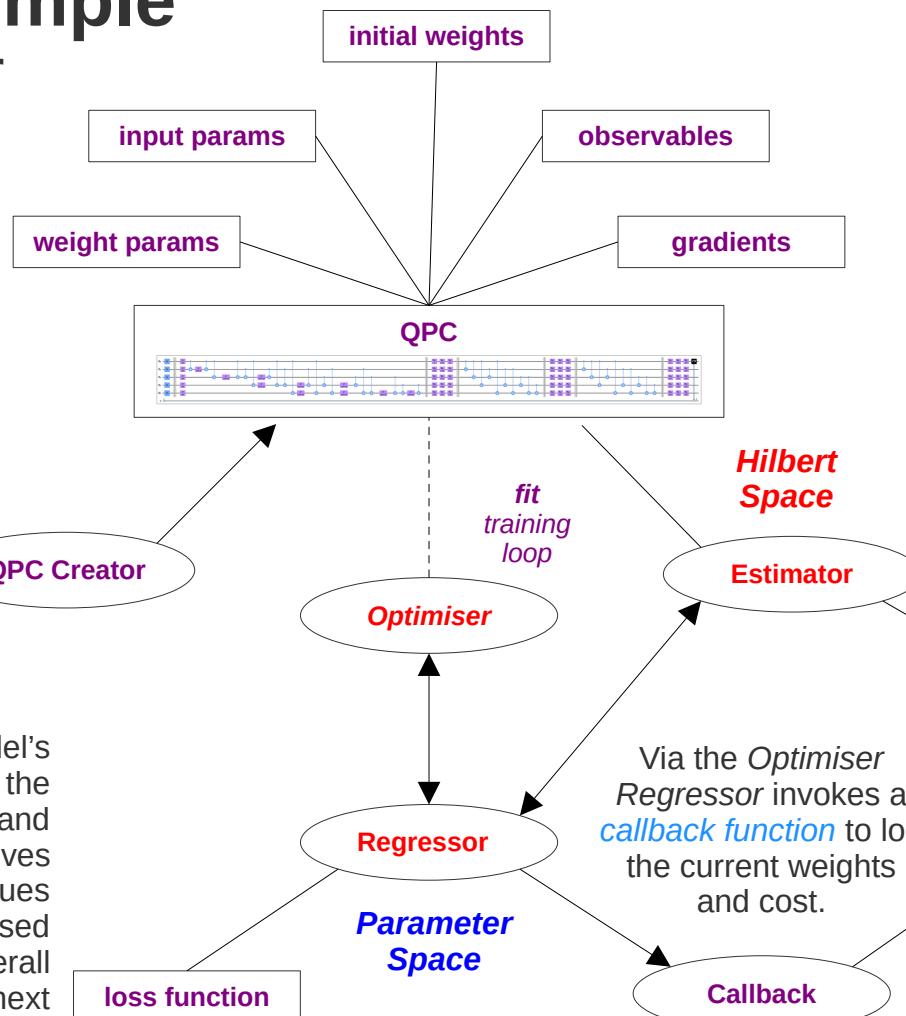
In this workshop we provide two alternative QNN models. The first features the commonly used circuit structure relying on Qiskit supplied parameterised circuits. The second is custom made and is created from the Qiskit basic building blocks (gates and parameters).

Training a simple TS Qiskit estimator

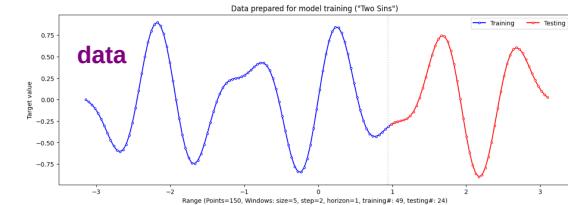
Qiskit **Optimiser** provides function **fit** which executes a training loop, performing: a *forward* pass which applies the model with its current parameters to training data, *loss function*, and a *backward* pass to improve the model parameters.



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.



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



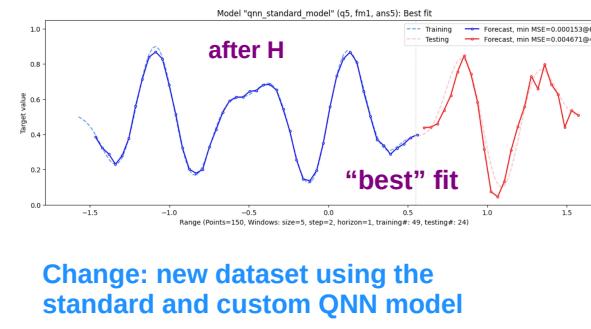
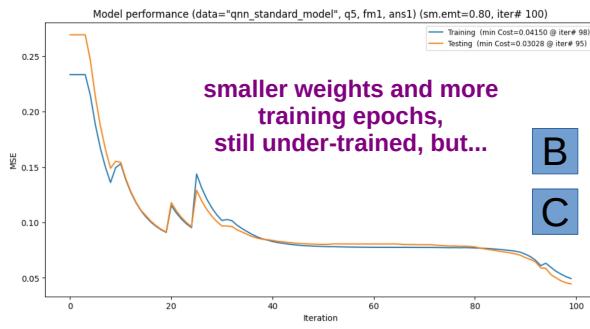
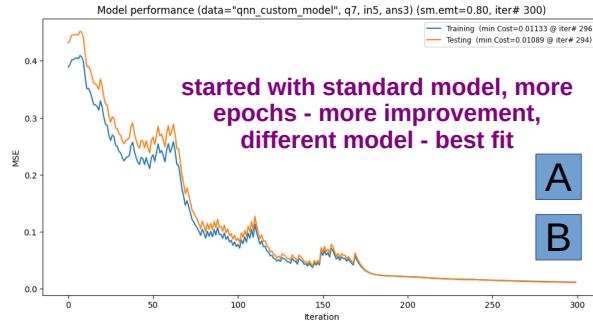
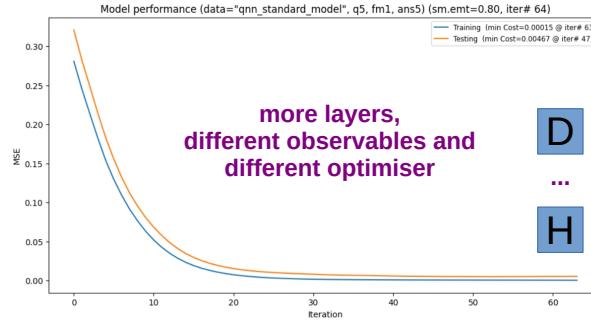
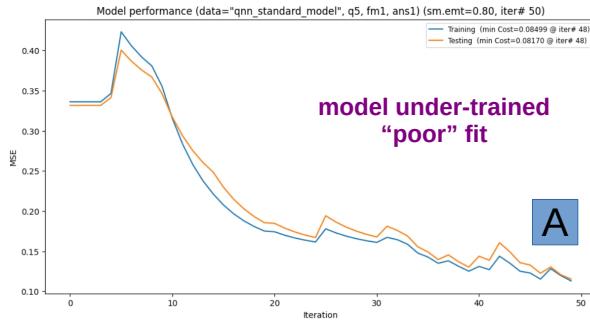
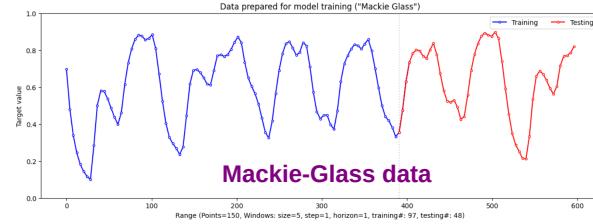
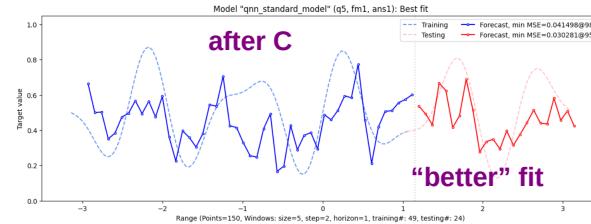
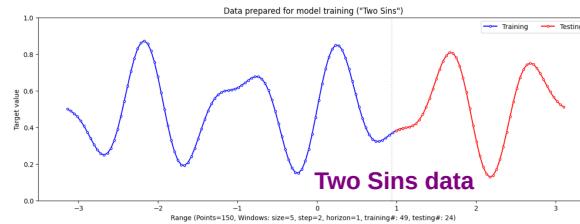
Estimator creates the physical circuit using the *observables*, *input parameters* and *weight parameters*, and the *gradient method* used in the calculation of expectation values. It then executes the circuit by relying on a hardware specific *estimator primitive*. It returns the calculated expectation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

estimator primitive

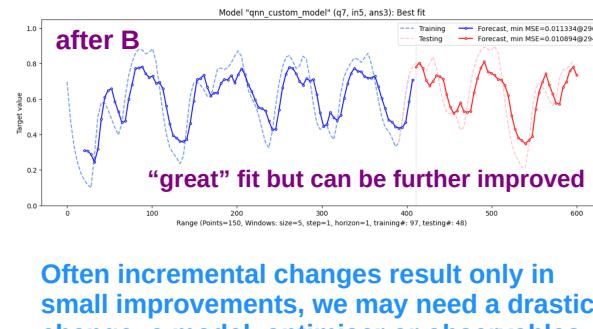
Model training started		training log
(00:00:00)	- Iter#:	0 / 500, Cost: 0.238564
(00:00:07)	- Iter#:	50 / 500, Cost: 0.162685
(00:00:14)	- Iter#:	100 / 500, Cost: 0.126066
(00:00:21)	- Iter#:	150 / 500, Cost: 0.073866
(00:00:29)	- Iter#:	200 / 500, Cost: 0.053152
(00:00:36)	- Iter#:	250 / 500, Cost: 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

In search of a solution!



Change: new dataset using the standard and custom QNN model



Often incremental changes result only in small improvements, we may need a drastic change: a model, optimiser or observables

Task: improve a forecasting model for two datasets

Let's look at the code

The screenshot shows a GitHub repository page for 'qml_bcd_lab'. The repository is public and has 1 branch and 0 tags. The main branch contains 5 commits from 'ironfrown' (BCD V9.14) made 2 days ago. The commits include changes to 'dev', 'examples', 'runs', 'utils', '.gitignore', 'LICENSE', and 'README.md'. The 'README' file is the active tab. The page also lists the 'About' section, which describes the repository as a workshop session introducing quantum machine learning for those already familiar with Quantum Computing algorithms and Qiskit. It includes links to 'Readme', 'GPL-3.0 license', 'Activity', '0 stars', '0 watching', and '0 forks'. The 'Releases' section indicates no releases have been published, with a link to 'Create a new release'. The 'Packages' section shows no packages published, with a link to 'Publish your first package'. The 'Languages' section shows Jupyter Notebook at 99.5% and Python at 0.5%. The 'Suggested workflows' section is based on the tech stack and includes a 'Python package' button with a 'Configure' link.

qml_bcd_lab Public

main 1 Branch 0 Tags Go to file Add file Code About

ironfrown BCD V9.14 66aefb5 · 2 days ago 5 Commits

dev BCD V9.14 2 days ago

examples BCD V9.14 2 days ago

runs BCD V9.14 2 days ago

utils BCD V9.14 2 days ago

.gitignore Initial commit 5 days ago

LICENSE Initial commit 5 days ago

README.md Update README 4 days ago

README GPL-3.0 license

Quantum Machine Learning B-C-D in Qiskit

- Author: [Jacob Cybulski \(LinkedIn\)](#), Enquanted
- Associated with: [QPoland](#)
- Aims: This is a workshop session introducing quantum machine learning for those already familiar with Quantum Computing algorithms and Qiskit.
- Prerequisites: This GitHub assumes good knowledge of quantum computing and machines learning, as well as previous experience with Python and Qiskit.
- Description: This QML BCD lab explores the process of developing a simple quantum machine learning model in Qiskit. The lab includes a practical session that covers the QML concepts, models, and techniques. The initial lab tasks will be demonstrated by the presenter. The following tasks are designed to be completed by the participants and discussed on Discord.

About

This is a workshop session introducing quantum machine learning for those already familiar with Quantum Computing algorithms and Qiskit.

Readme

GPL-3.0 license

Activity

0 stars

0 watching

0 forks

Releases

No releases published

Create a new release

Packages

No packages published

Publish your first package

Languages

Jupyter Notebook 99.5%

Python 0.5%

Suggested workflows

Based on your tech stack

Python package Configure

Create and test a Python package on multiple Python versions.

Resources for this session, see:
ironfrown (Jacob L. Cybulski, Enquanted)
https://github.com/ironfrown/qml_bcd_lab

Quantum model performance:

Scoring a quantum model (different example)

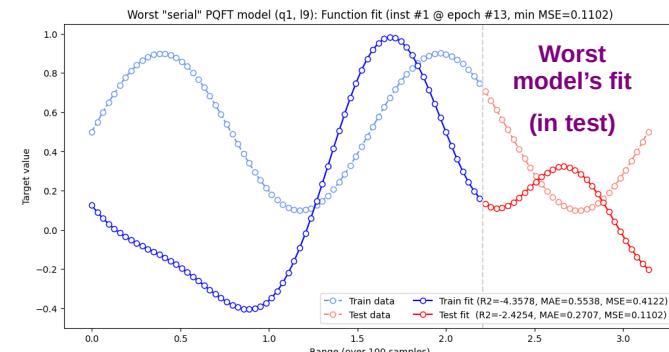
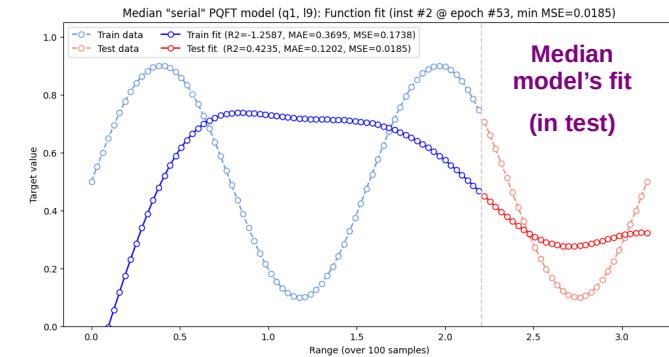
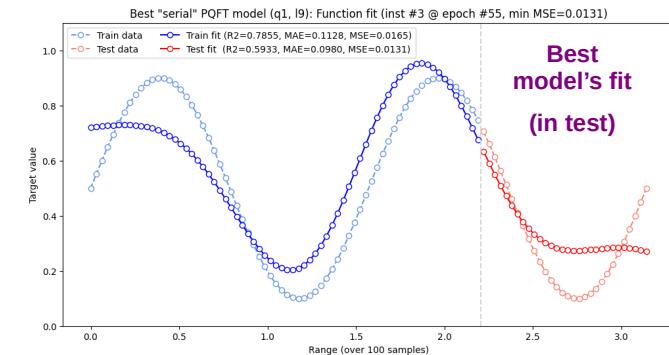
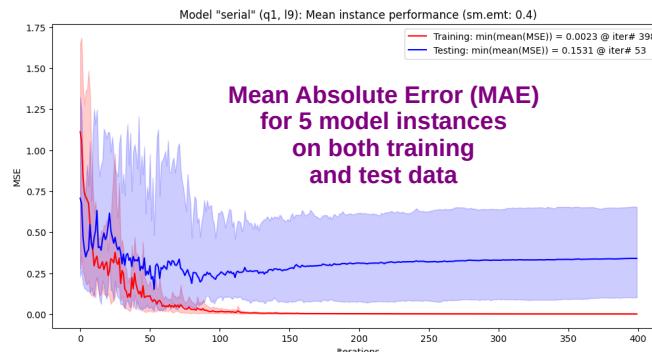
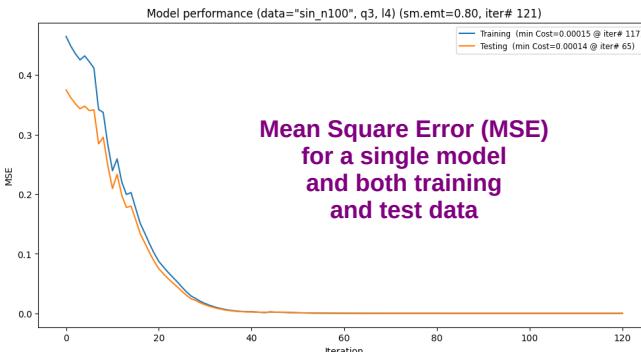
Quantum model training relies on the training data and a loss function to guide the optimiser, e.g. L2Loss (MSE cost), however, other performance metrics may also be needed, e.g. MSE, MAE or R^2 , calculated for training, validation and test data.

Therefore, at each optimisation step, the model parameters are saved for later use. These parameters values can be assigned to the weights of the model circuit, which can then be scored using all data partitions, against the expected values (figure bottom-left).

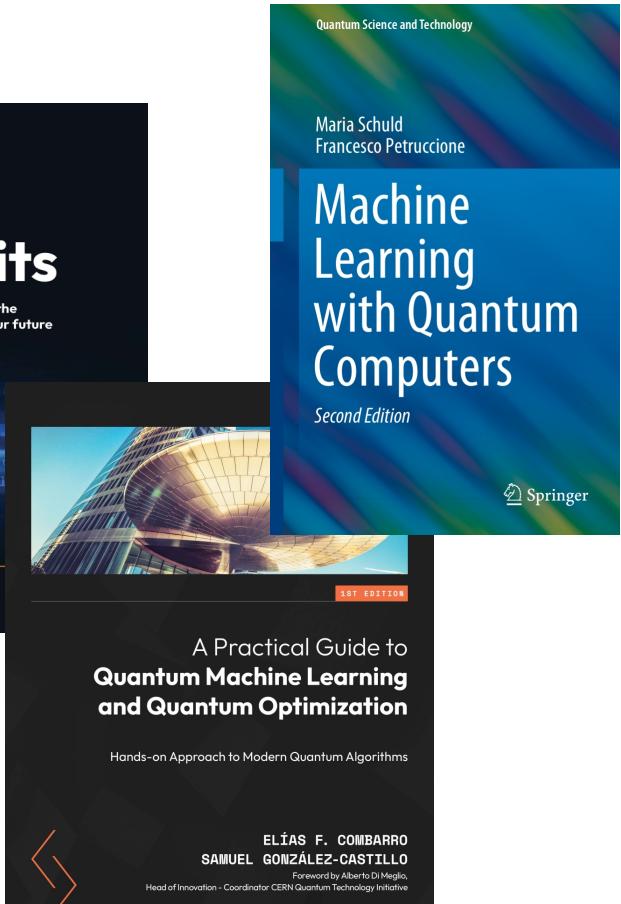
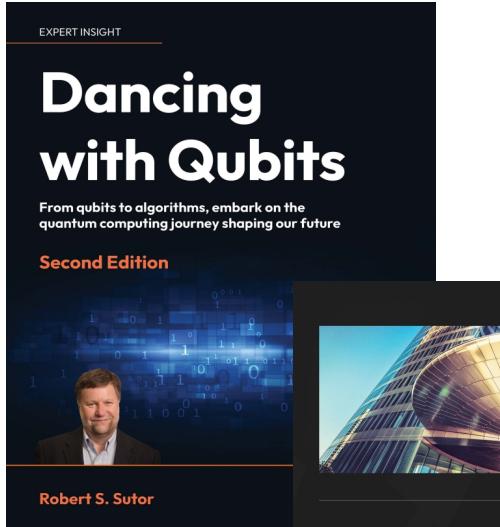
However, as a quantum model performance is highly sensitive to its initialisation, it is also advisable to run multiple, differently initialised, instances of the same model. Subsequently we can analyse a distribution of their performance results, e.g. here we present 5 instances of the same model with identical configurations (figure bottom-middle).

When doing so, it is also possible to present the level of model's fit to data, depending on it best, median or worst instance performance (figures right).

In doing so, our performance assessment can be reported in honest and unbiased way.



Recommended reading on QML with Qiskit



Quantum computing with Qiskit

Ali Javadi-Abhari,¹ Matthew Trinastic,¹ Kevin Krissian,¹ Christopher J. Wood,¹ Jake Lishman,¹ Julian Gacon,³ Simon Martis,⁴ Paul D. Nation,¹ Lev S. Bishop,¹ Andrew W. Cross,¹ Blake R. Johnson,¹ and Jay M. Gambetta¹

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 runtime system. We also highlight the quantum circuit abstraction, which is the core of Qiskit, and its use in 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 reusability 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.

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, one of the first applications of Qiskit was the optimization of qubit mapping [51], a bit-tolerant identity [42], inspired of qiling gates [101, 104], or the circuit editor toolbox after [1]. The packt is a collection of books have been created by whom the Python as well as human [3, 4]. More have used and used Qiskit is the most [10]. A milestone in philosophy drive quantum circuit form capability a quantum leverages a virtual world and pressio system t

The transpiler translates circuit 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, the ISA is likely to be a sequence of low-level assembly-like instructions, such as `qreg`, `ctrl`, `target`, `ctrl`, `target`, etc.

Release News: Qiskit SDK v2.1 is here!

Technical release summary for Qiskit SDK v2.1, including updates on top new features, breaking changes, and our ongoing efforts to make Qiskit the world's most performant quantum SDK.

Qiskit v2.1 Release Notes

2.1

Thank you!

Any questions?

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



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

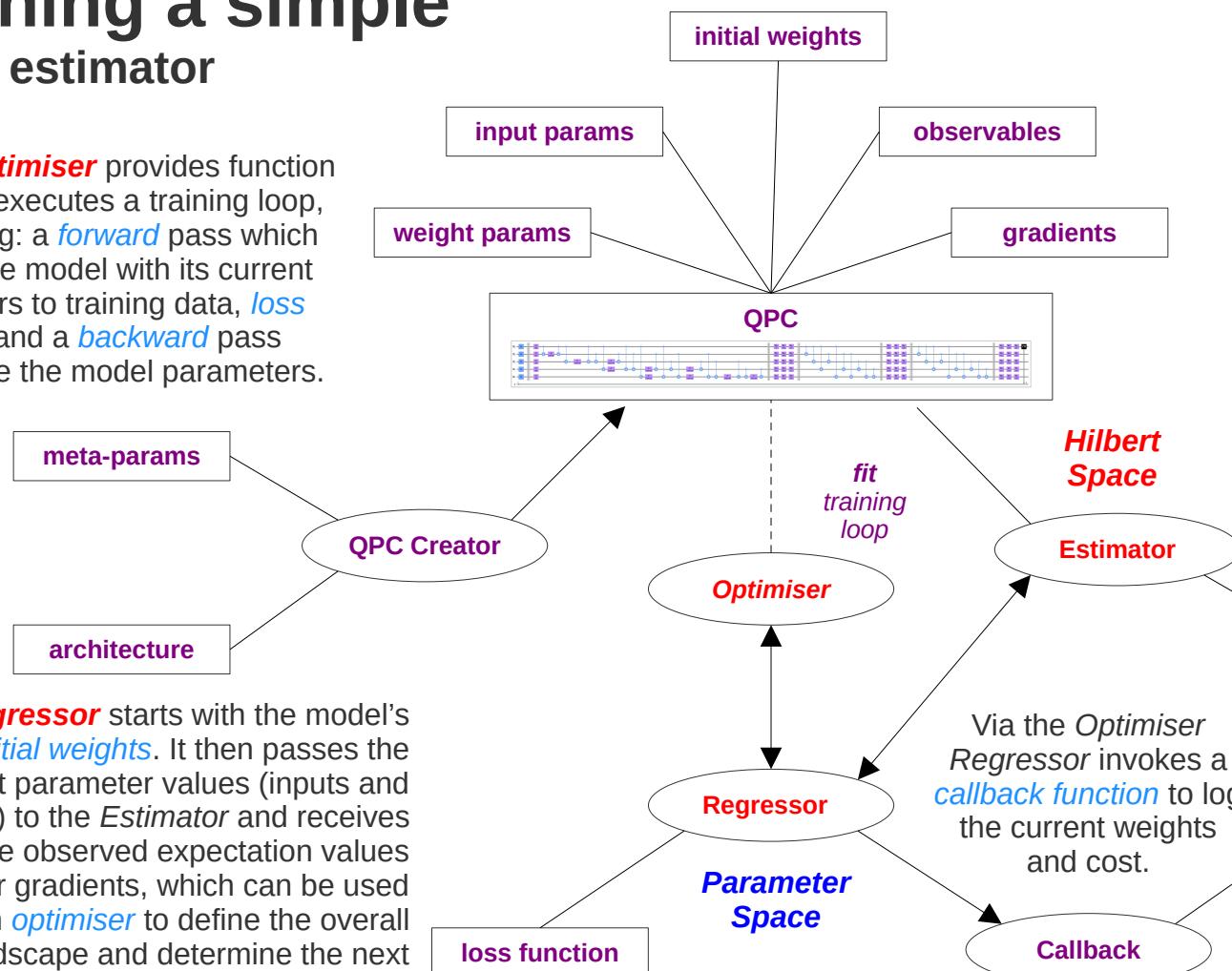
BY: credit must be given to the creator.

NC: Only noncommercial uses of the work are permitted.

ND: No derivatives or adaptations of the work are permitted.

Training a simple Qiskit estimator

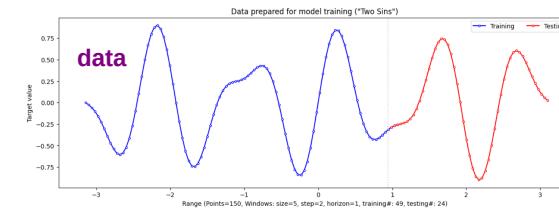
Qiskit **Optimiser** provides function ***fit*** which executes a training loop, performing: a ***forward*** pass which applies the model with its current parameters to training data, ***loss function***, and a ***backward*** pass to improve the model parameters.



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.

estimate primitiv

```
Model training started                                training log
(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
```