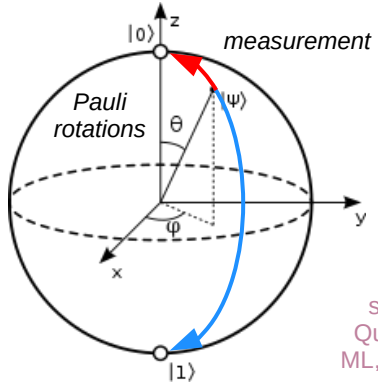
Secrets revealed in this session:

To explore the practical aspects of building quantum machine learning models and their optimisation

QML and its aims
Parameterised circuits
Variational quantum algorithms
Data encoding / angle encoding
The good, the bad and the ugly
State measurement
Ansatz design and training
Working with DL and QML models
Model geometry and gradients
Parameters optimisation
The curse of dimensionality
QML readings
Qiskit demo and tasks (TS curve fitting)
Summary and Q&A

Quantum Algorithms and Data Encoding for QML with Qiskit

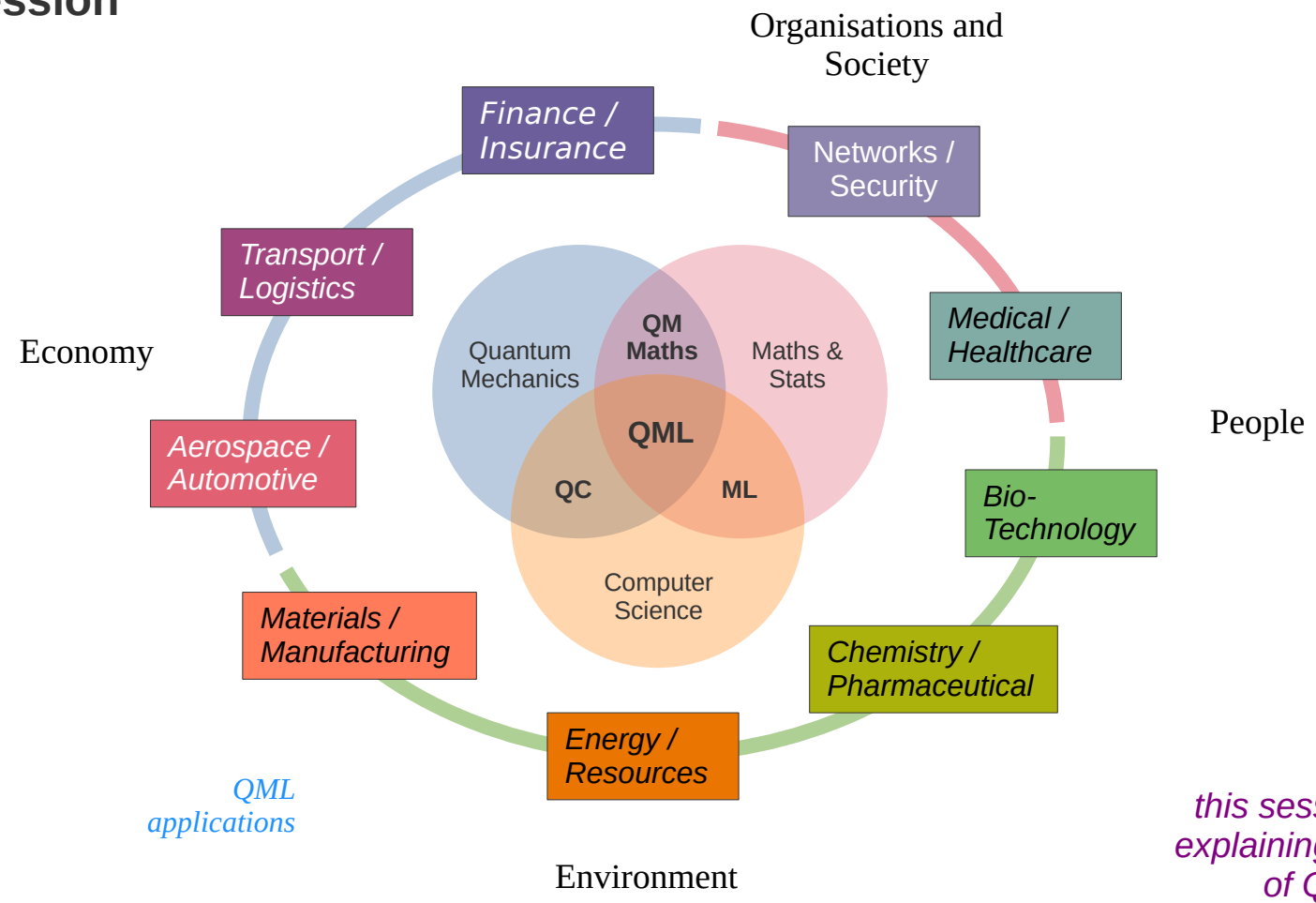
Jacob L. Cybulski
Enquanted, Melbourne, Australia



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

Quantum ML
aims of this session

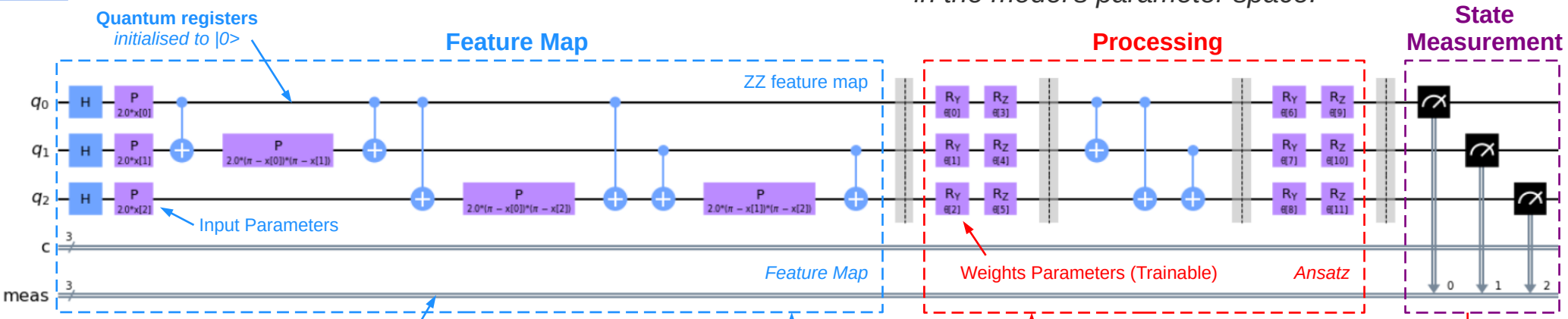
Jacob Cybulski, Founder
Enquanted, Australia



this session aims at explaining the nature of QML models

Variational Quantum Circuits
and Variational Quantum Algorithms

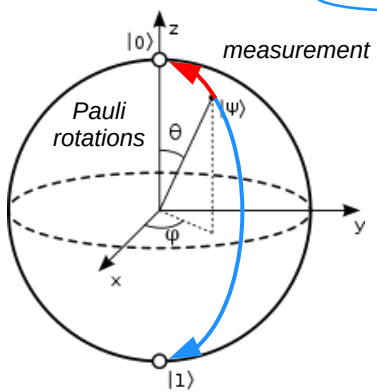
Ansatz parameters are trainable.
Each parameter defines a dimension in the model's parameter space.



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), the circuit consists 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.

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).

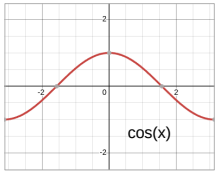
Rotations are *relative to a specific qubit state*, commonly starting at |0⟩ state, or (|0⟩+|1⟩)/sqrt(2), which require qubits to be initialised in these states.

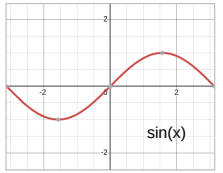
The encoded value could be represented either by the *angular rotation*, or the *amplitude* of the qubit projective measurement (Z).

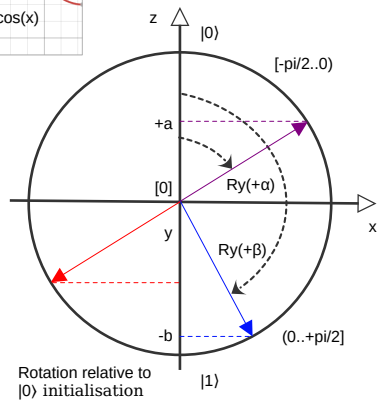
Input data can also be repeatedly encoded and spread around the circuit, which is called *data reuploading*, and which is known to improve the model performance.

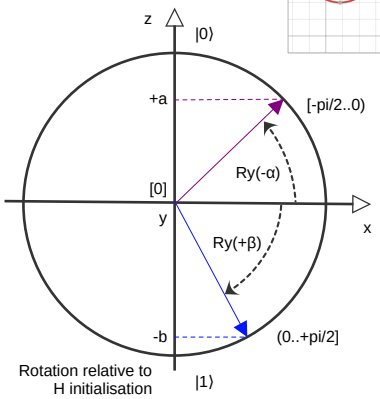
Maria Schuld and Francesco Petruccione.
Machine Learning with Quantum Computers. 2nd ed. Springer, 2021.

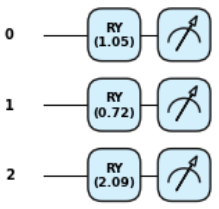
Note that training will place qubit states in areas x < 0 and arbitrarily around the z axis. Measurements of such states cannot distinguish them from "pure" x > 0 and z = 0.

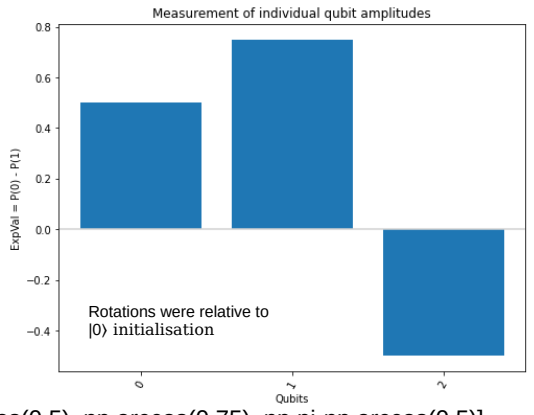












Input

Values entered: [np.arccos(0.5), np.arccos(0.75), np.pi-np.arccos(0.5)]
Ry angles used: [1.047, 0.723, 2.094]

Measurements

Probabilities: [[0.25, 0.75], [0.562, 0.438], [0.25, 0.75]]
Amplitudes: [0.5, 0.75, -0.5]

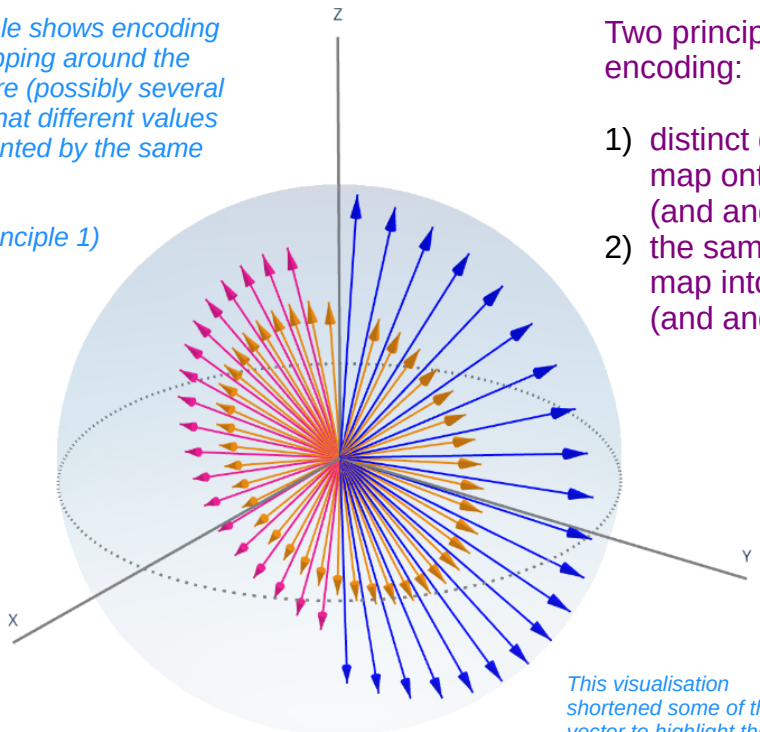
Angle encoding
The Good, the Bad and the Ugly

These two examples explain what happens at the very beginning of model training. Later during parameters optimisation, which relies on Rx, Ry and Rz state rotations and CNOT entanglements, the model qubits potentially assume all possible states on their Bloch sphere. It is important, however, that the measured outcomes are not in conflict.

Angle Range: -6 to 6
0..π = "blue" (long) | >π = "deeppink" (medium) | -π..0 = "darkorange" (short)

This example shows encoding values wrapping around the Bloch sphere (possibly several times), so that different values are represented by the same amplitude.

(violates principle 1)



This visualisation shortened some of the vector to highlight their difference, however, they are all of length 1.

Two principles of quantum data encoding:

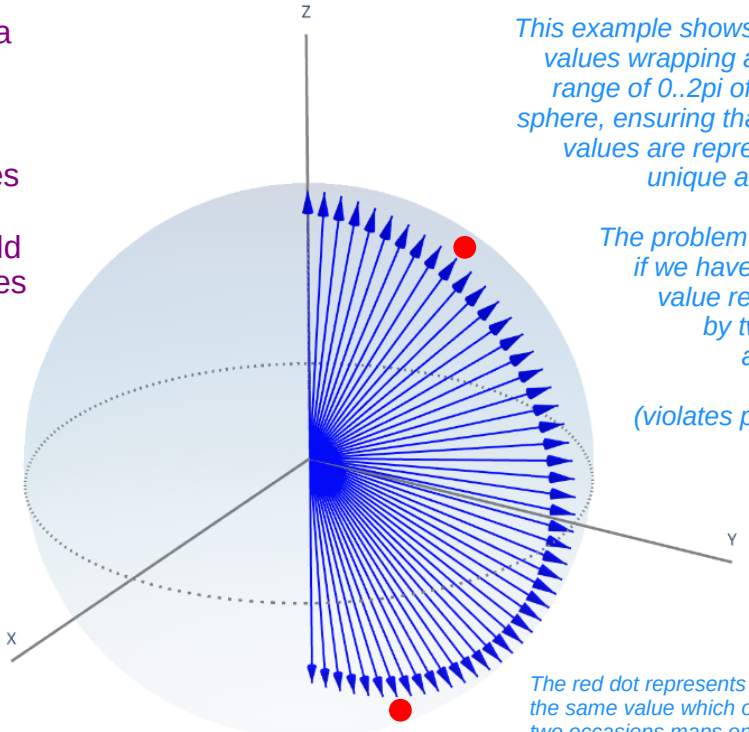
1) distinct data values should map onto distinct amplitudes (and angular codes)
2) the same data values should map into identical amplitudes (and angular codes)

Angle Range: 0 to 3.141592653589793
0..π = "blue" (long) | >π = "deeppink" (medium) | -π..0 = "darkorange" (short)

This example shows encoding values wrapping around the range of 0..2pi of the Bloch sphere, ensuring that different values are represented by unique amplitudes.

The problem may arise if we have the same value represented by two distinct amplitudes

(violates principle 2)



The red dot represents the same value which on two occasions maps onto two different angles and thus two amplitudes.

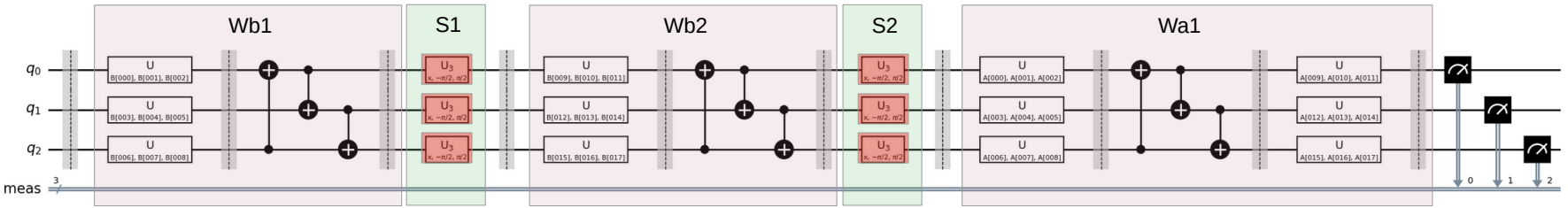
Ansatz design and training

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** !*



*Data **reuploading** across circuit's width and depth*

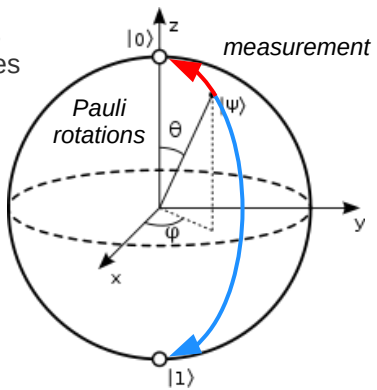
feature maps vary in:
structure and function (!!!)

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 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-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)

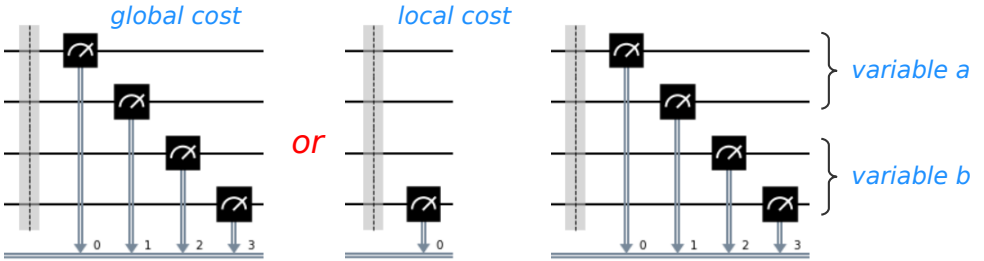
Commonly used
measurements and interpretation

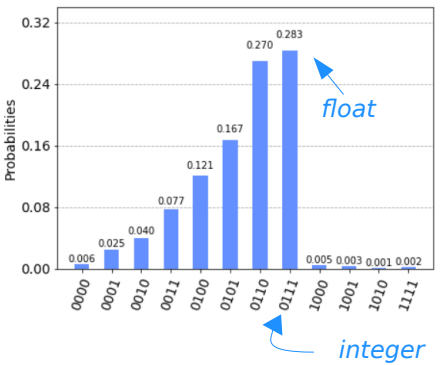
There are many ways of obtaining the outcome of a circuit execution, e.g. we can measure:

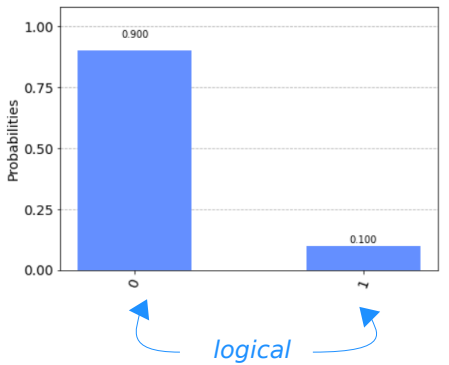
- all qubits (global cost / measurement)
- a few selected qubits (local cost / measurement)
- groups of qubits (each as a variable value)
- as counts of outcomes (repeated measurements)
- as probabilities of outcomes (e.g. P(|0111⟩))
- as Pauli expectation values (i.e. of eigenvalues)
- as expectation of interpreted values (e.g. 0 to 15)
- as variance, etc.

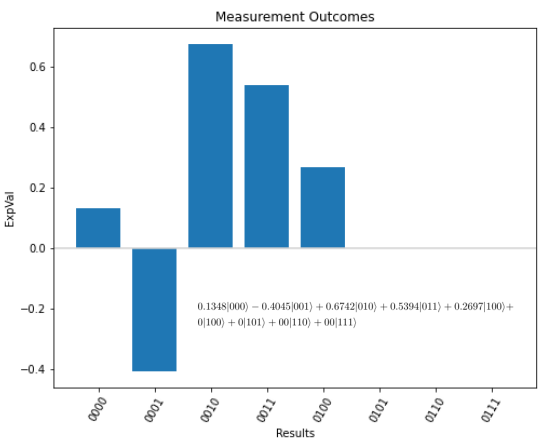
Repeated circuit measurement can be interpreted as outcomes of different types, e.g.

- 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. $|0^n\rangle$)









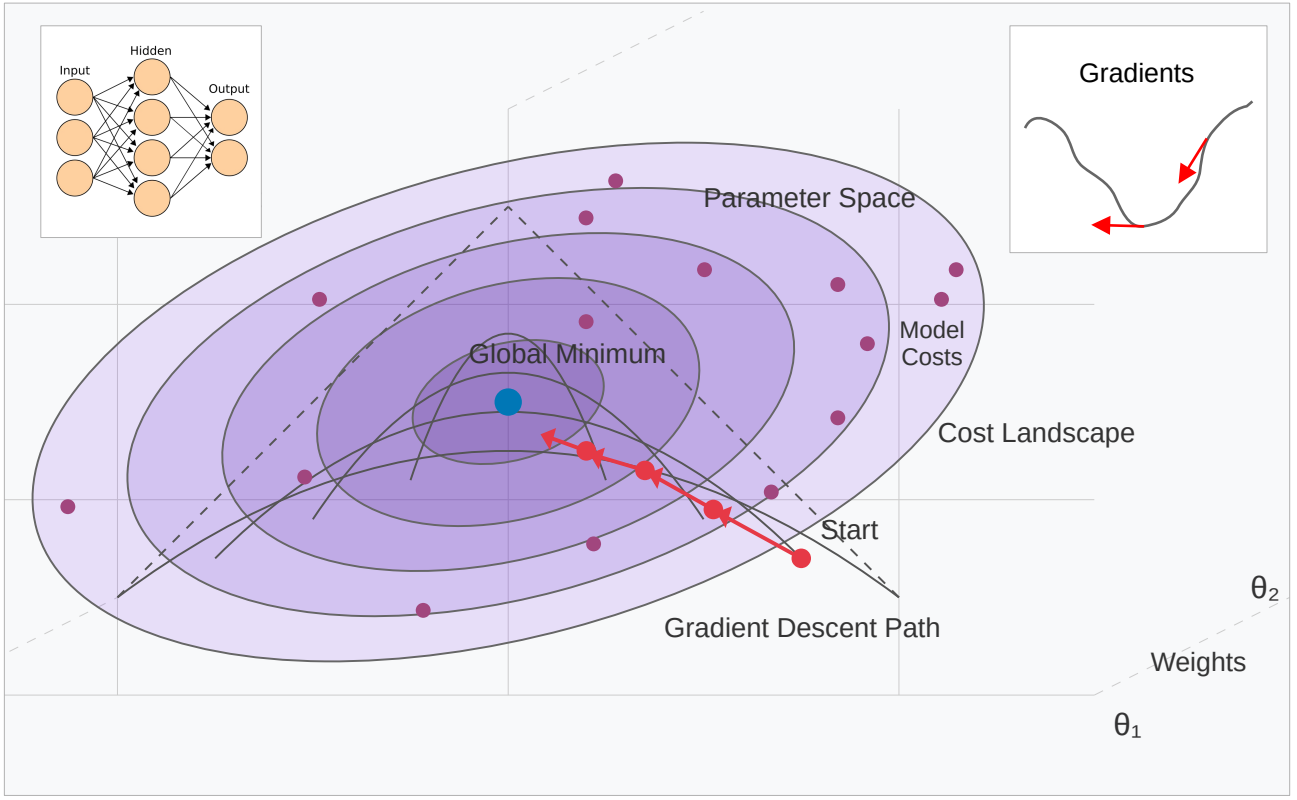
Or we can measure expectation values of the circuit state and interpret them as a series of values

Beware that adding 1 measurement → doubles the number of outcomes!

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

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

Working with deep learning models
Classical model optimisation

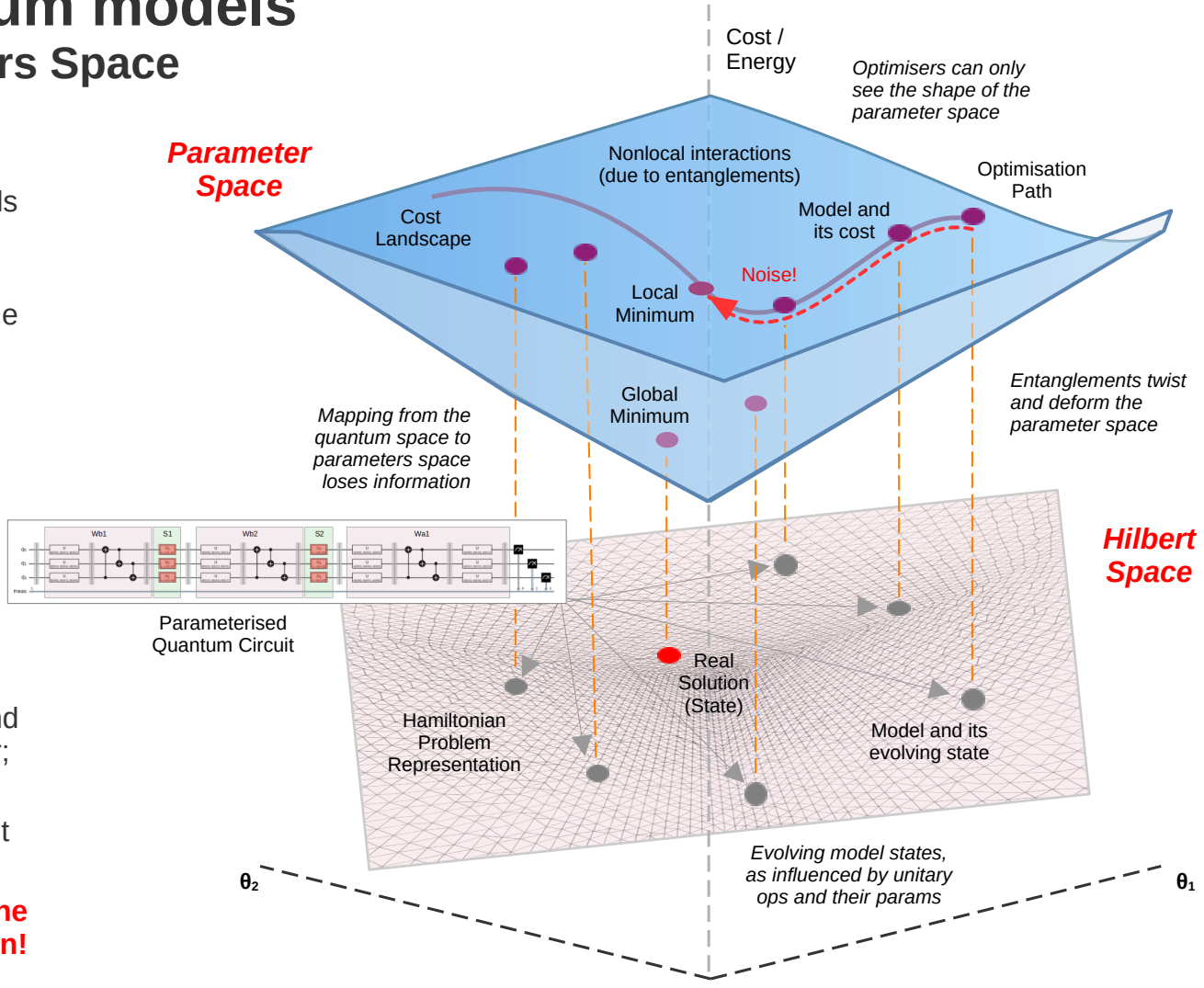


Gradients are local, i.e. their changes influence only their immediate neighbourhood

- Deep learning (DL) models evolves layers of consecutive *representations of data on input*
- *Weights* and *biases* parameterise DL models
- Model *parameters are independent*, with *local interaction* only, limited to neighbourhood
- *Cost function* evaluates the model quality, where the lower the cost, the better the model
- Differently parameterised models form a multi-dimensional geometry = *the parameter space*
- The costs of the all DL models form a surface of this geometry = *the cost landscape*
- *Optimisation algorithms* navigate the the cost landscape in search for the minimum cost
- This process may rely on the shape of the cost landscape and its *gradients*
- *Gradient descent* algorithms navigates gradients towards the *minimum cost* model
- *Backpropagation* implements gradient descent very efficiently by relying on the knowledge of local interactions between model parameters
- In the *forward step*, a gradient-based optimiser calculates the activation of DL model nodes, producing the model's output
- In the *backward step* it backpropagates its calculation error, and recalculates DL model weights and biases to correct the error

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;
- **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!**

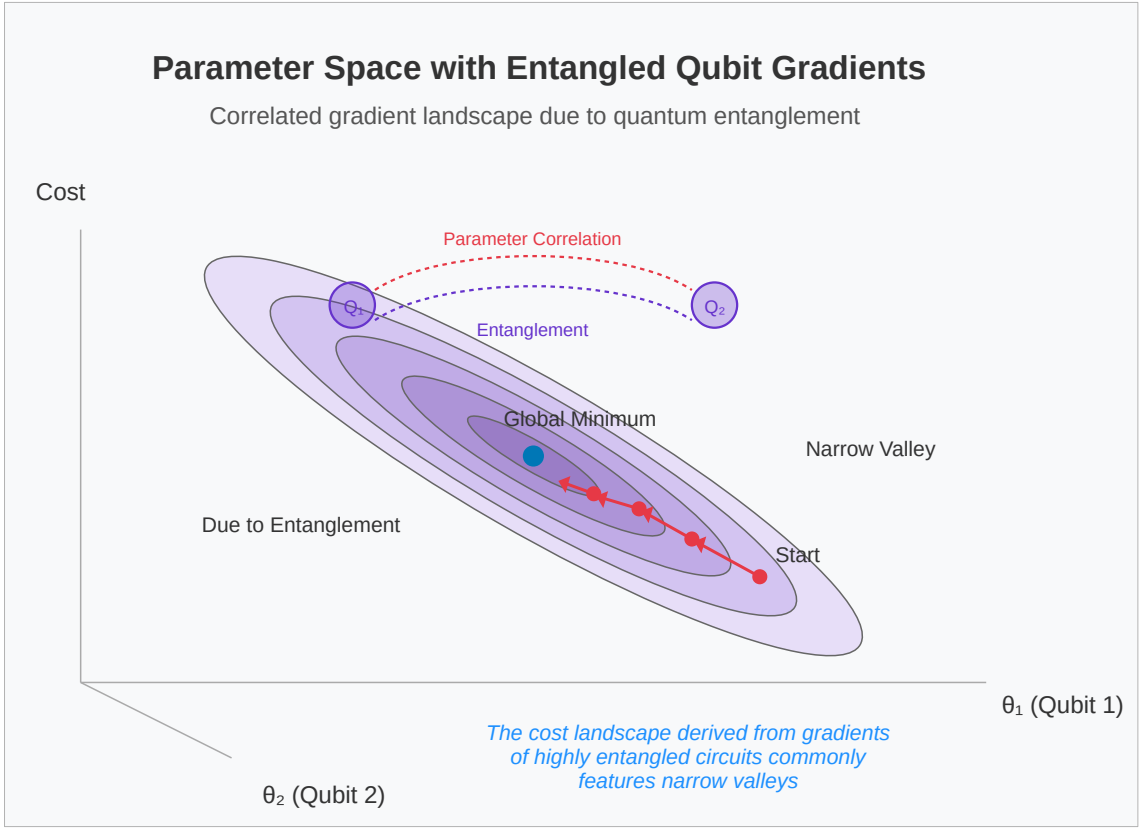


Quantum model optimisation

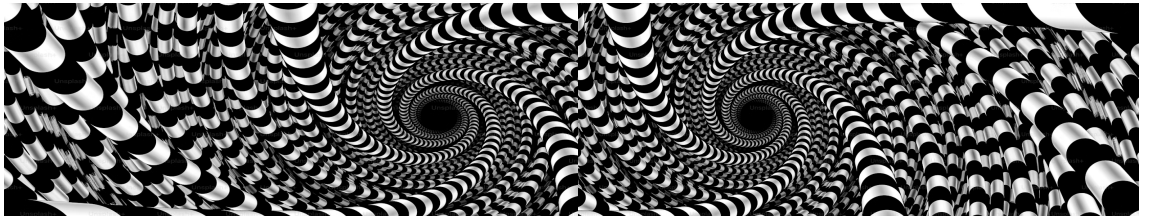
model geometry vs. classical optimisers

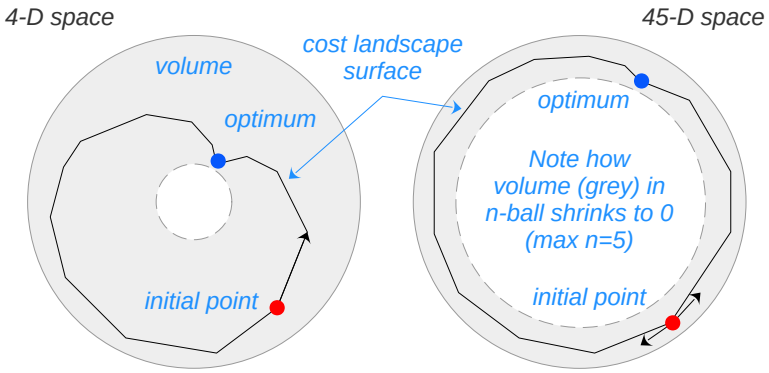
- All models generated from a PQC can be defined by the possible valid combination of its parameters
- A set of all valid models form some *geometry* in quantum state space and classic parameter space

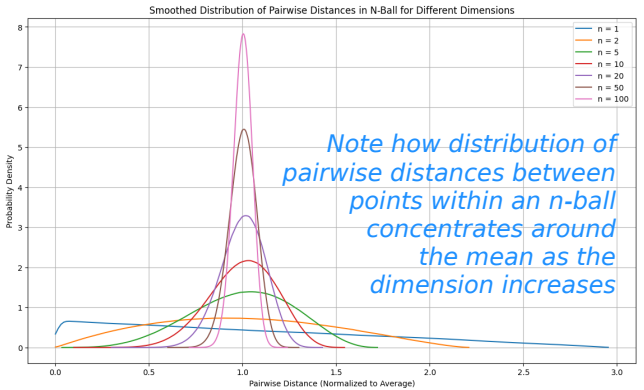
- This geometry is commonly combined with a set of model observables, which create a *navigable surface*, e.g. a quantum *energy landscape* or classical *cost landscape*
- *Classical optimisers* can then be used to navigate the classical parameter space in search for the model configuration associated with the *minimum cost*
- Such optimisation of quantum models needs unique approaches due to the presence of *non-local gradients*
- *Entangled qubits* result in *correlated parameters*, and indirectly gradients, so the changes to one are reflected in the changes to distant others
- *Backpropagation* cannot thus be used effectively in training quantum circuits, as the model states, and thus local gradients, are not directly accessible and their measurements collapse the state
- *Gradient descend* can still be used with *global gradients*, derived from the geometry of the cost landscape rather than from the relationships between parameters
- Other optimisation techniques can be used instead of gradient-based methods, e.g. *stochastic optimisation* or *particle swarm optimisation*



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 parameters)

- 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)

Quantum model dev.
training, validation and testing

While developing a quantum model we need to test its capacity to learn. This can be done by applying the model to data used in its training, with the objective to test its ability to recall what it learnt.

To assess the model's performance on new data we need to apply it to data not used in training.

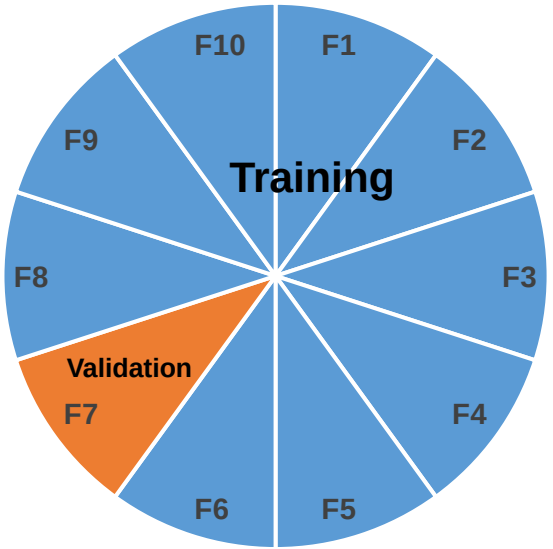
Commonly adopted approach is known as *holdout testing*, where we randomly split data into three partitions - one to be used for model *training*, one for its *validation* while improving the model, and one for *honest testing* on data never used before.

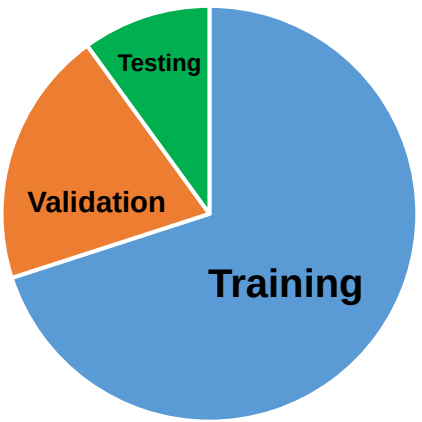
Holdout testing validates and test the model once only, assuming all data partitions as representative of the population, which may not be true.

Cross-validation (CV) is a more appropriate testing method, which repeatedly trains and validates a model using different data samples (folds), then averaging the performance obtained from all runs.

As training a quantum model may be extremely slow, cross-validation may not be feasible.

k-fold CV





As quantum models are sensitive to their initialisation, once the model is developed, we retest it with differently initialised weights, in the process also resampling data partitions.

Note: as we develop the model we freeze data partitions by setting a *random seed*, to ensure performance changes are due to our actions and not differences in data partitions.

Qiskit QML
Workshop

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 curve fitting):

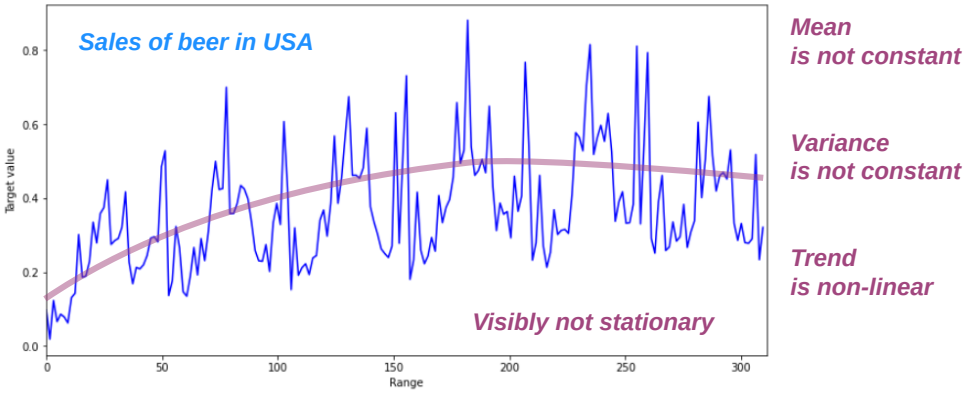
- Add ML 0.8.3 package to Qiskit 1.4.4 (Python 3.11)
- Create a few simple models to fit samples of data
- Learn to initialise model weights
- Learn the interaction b|n data encoding and ansatz
- Understand how observables / measurements work
- Explore the impact of ansatz structure on performance
- **Challenge:** Apply your skills to complex data
- **Reflection:** Refine your QML development process

Key takeaways:

- Plan model development, tests and experiments
- More params and entanglements improve *expressivity*
- Data reuploading makes a huge difference!
- More width / depth / params reduce *trainability* = *the curse of dimensionality*
- More entanglements reduce *trainability*
- High dimensional param space upsets even non-gradient optimisers due to *model sparsity*
- Bad data encoding spoils the bunch!
- Carefully consider your quantum model initialisation
- Surprise - a single qubit model still works! (and well)
- More training often does not eliminate problems!

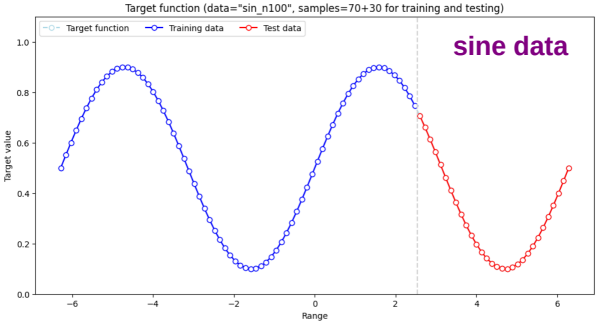
Key concepts in time series analysis

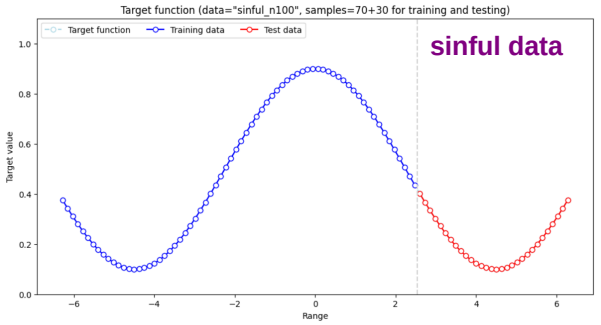
- Time series (TS) analysis aims to *identify patterns* in historical time data and to *create forecasts* of what data is likely to be collected in the future
- *Many TS applications*, including heart monitoring, weather forecasts, machine condition monitoring, etc.
- *Many excellent tools* and *efficient methods* of TS analysis, yet improvements are still desirable
- Time series must have an *unique temporal index* e.g. a time-stamp sequencing the series
- Time series must also have some *time-dependent attributes* suitable for modelling

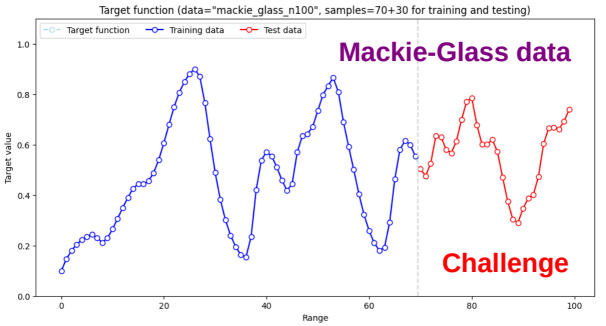


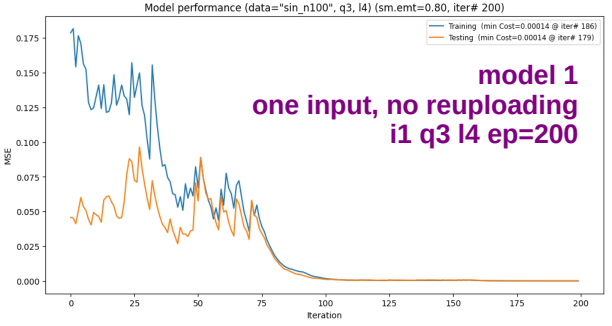
- Time series can be *univariate* or *multivariate*
- Time series often show *seasonality* in data, i.e. some patterns repeating over time
- TS values are dependent on the preceding values!
- Distinction between consecutive TS values is small!
- There are several different types of TS models, e.g.
  - 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*, i.e. dimensionality reduction, TS aggregation, imputation of missing values, removal of noise and outliers, adherence to normality and homoskedasticity (e.g. they need to be stationary)
- QML methods do not have such strict requirements, and are promising for effective time series analysis and forecasting!

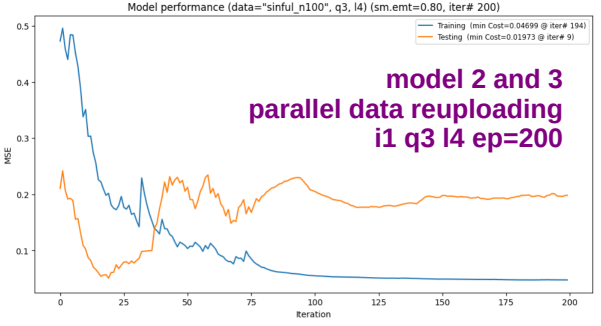
Data and some results

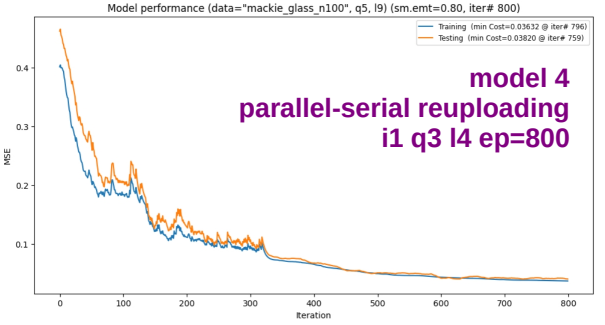


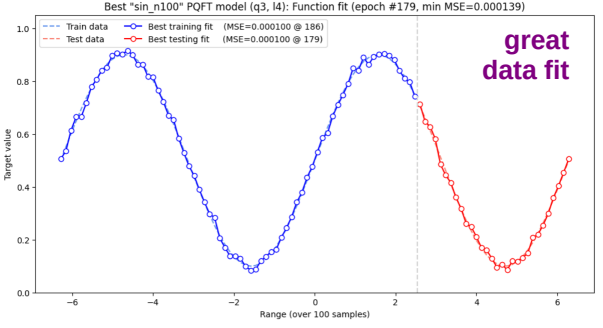


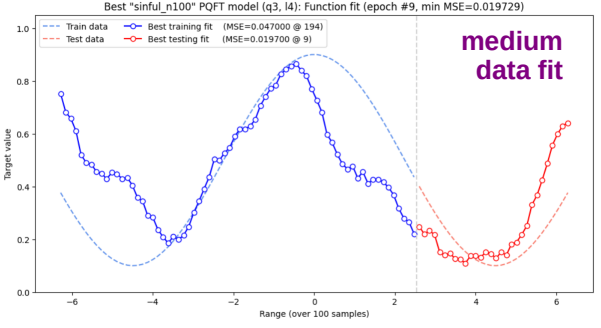


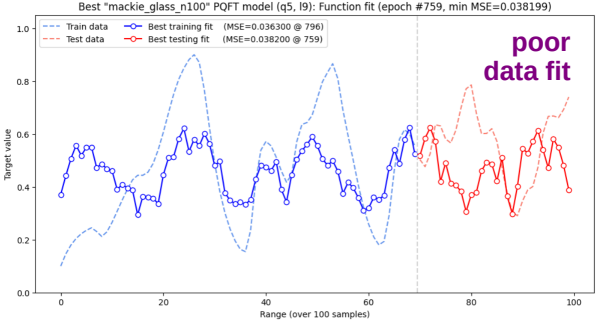








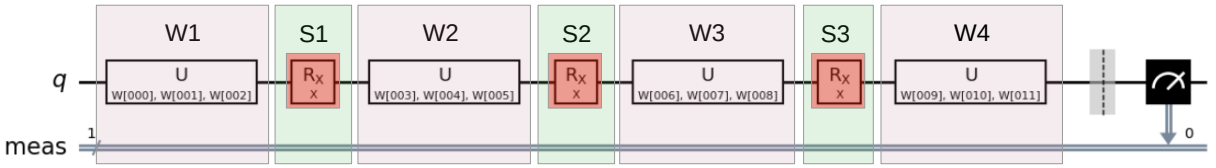




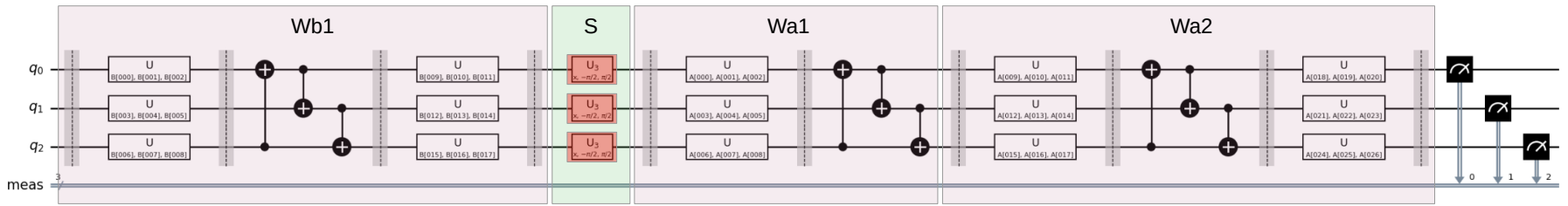
Tasks: improve curve-fitting of these models

Curve-fitting models

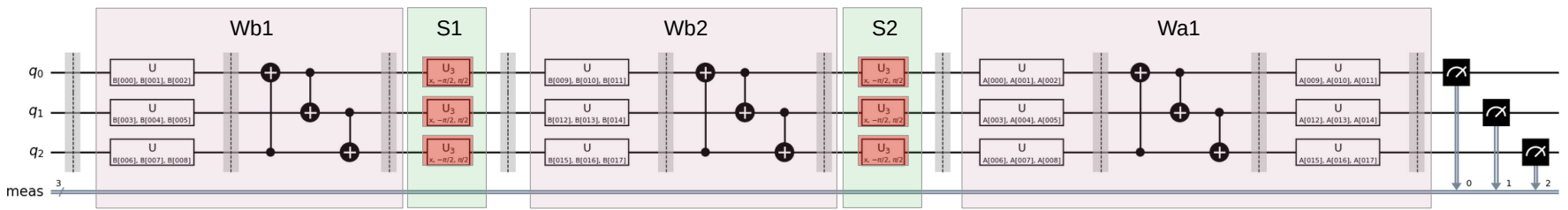
Serial



Parallel



Parallel-Serial



Qiskit models
Custom circuit creation

Here we present a simple curve-fitting quantum model developed in Qiskit.

The model's architecture consists of a feature map of a single Rx gate and several layers (4), each layer providing trainable parameters (Rx, Ry, Rz) spanning all qubits (3 x 3), as well as, an entangling block of CNOT gates arranged in a circular fashion (3).

The following code illustrates model creation and its preparation for training:

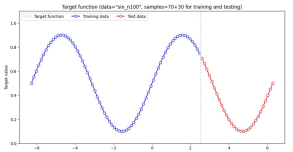
[1] Function creating parameterised model circuit (inputs_no, qubits_no, layers_no).
[2] Meta-parameters defining the model and its training (epochs, shots, weight_scaler).
[3] Selection of the model function, loss function and optimiser.
[4] Creation of the circuit and a vector of initial weight values.

```
##### Model 1 - Single input
def qnn_model_1(qubits_no, layers_no, add_meas=False):
    qr = QuantumRegister(qubits_no, 'q')
    ansatz = QuantumCircuit(qr, name="ansatz")
    param_x = Parameter('X')
    ansatz.rx(param_x, 0)

    for l in range(layers_no):
        ansatz.barrier()
        for q in range(qubits_no):
            ansatz.rx(Parameter(f'P{l:02d}_A{q:02d}'), q)
            ansatz.ry(Parameter(f'P{l:02d}_B{q:02d}'), q)
            ansatz.rz(Parameter(f'P{l:02d}_C{q:02d}'), q)
        for q in range(qubits_no-1):
            ansatz.cx(q, q+1)
        if qubits_no > 1:
            ansatz.cx(qubits_no-1, 0)
    if add_meas:
        ansatz.measure_all()
    return ansatz
```

define a quantum model **1**

```
##### Model settings for models with 1 input

inputs_no = 1          # Number of inputs
qubits_no = 3          # Number of qubits
layers_no = 4          # Number of layers
epochs = 200           # Tr
shots = 1000           # Ig
print_fract = 0.1      # Fr
weights_scaler = 1.00  # Sc
```

2 and its settings



Sample data needs to be prepared, cleaned and split into training and test partitions.

```
### Select which model to use
mfun = qnn_model_1

### Select the loss function (here, MSE cost)
loss_fun = L2Loss()

### Select the optimiser (here, non-gradient based)
optimizer = COBYLA(maxiter=epochs)
```

3

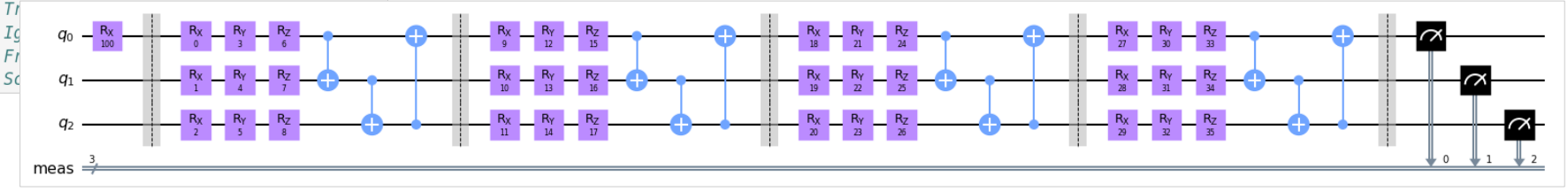
select:
- **model**
- **loss function**
- **optimiser**

```
algorithm_globals.random_seed = seed
np.random.seed(seed)

### Create and initialise a quantum model
model = mfun(qubits_no, layers_no)
init_weights = weights_scaler * 2 * np.pi * \
    algorithm_globals.random.random(model.num_parameters-inputs_no)

### Define a QNN estimator

obs = SparsePauliOp.from_list([("Z" * model.num_qubits, 1)])
```

create model circuit
calculate initial weights

4

model circuit



Training a simple
Qiskit estimator

```
model = mfun(qubits_no, layers_no)
init_weights = weights_scaler * 2 * np.pi * \
    algorithm_globals.random.random(model.num_parameters-inputs_no

### Define a QNN estimator

obs = SparsePauliOp.from_list([("Z" * model.num_qubits, 1)])
estimator = StatevectorEstimator(seed=seed)
regr_qnn = EstimatorQNN(
    circuit=model,
    input_params=[model.parameters[-inputs_no]],
    weight_params=model.parameters[:-inputs_no],
    observables=obs,
    estimator=estimator,
    gradient=ParamShiftEstimatorGradient(estimator),
)
```

5

estimator

```
### Define and fit regressor

regr_callback = create_callback(epochs, print_fract=print_fract)

regressor = NeuralNetworkRegressor(
    neural_network=regr_qnn,
    loss=loss_fun,
    optimizer=optimizer,
    initial_point=init_weights,
    callback=regr_callback
)

### Train the model
print(f'\nModel training started\n')
```

regressor

6

```
### Select the loss function (here, MSE cost)
loss_fun = L2Loss()

### Select the optimiser (here, non-gradient based)
optimizer = COBYLA(maxiter=epochs)
```

```
    callback=regr_callback
)

### Train the model
print(f'\nModel training started\n')

regressor.fit(X_train, y_train)
elapsed = time.time() - start
time_str = time.strftime("%H:%M:%S", time.gmtime(elapsed))
```

training loop

7

```
Model training started

(00:00:00) - Iter#:   0 / 200, Cost: 0.178608
(00:00:01) - Iter#:  20 / 200, Cost: 0.176996
(00:00:02) - Iter#:  40 / 200, Cost: 0.028761
(00:00:03) - Iter#:  60 / 200, Cost: 0.008521
(00:00:05) - Iter#:  80 / 200, Cost: 0.004425
(00:00:06) - Iter#: 100 / 200, Cost: 0.001005
(00:00:07) - Iter#: 120 / 200, Cost: 0.000322
(00:00:09) - Iter#: 140 / 200, Cost: 0.000784
(00:00:10) - Iter#: 160 / 200, Cost: 0.000176
(00:00:11) - Iter#: 180 / 200, Cost: 0.000145

Total time 00:00:12, min Cost=0.000144
```

training log

In Qiskit it is possible to create variety of different quantum models.

Here we will show to to build a custom circuit using a Qiskit state vector estimator and a neural network regressor.

```
##### Callback function use to collect training data

objfun_vals = []   # To store objective function values
params_vals = []   # To store parameter values

def create_callback(epochs, print_fract=0.1):

    global objfun_vals, params_vals
    objfun_vals = []
    params_vals = []
    elapsed = 0
    start_time = time.time()

    def callback_func(weights, obj_func_eval):
        nonlocal epochs, print_fract, start_time, elapsed
        global objfun_vals, params_vals

        iters = len(objfun_vals)
        objfun_vals.append(obj_func_eval)
        params_vals.append(weights)
        elapsed = time.time() - start_time
        time_str = time.strftime("%H:%M:%S", time.gmtime(elapsed))
        if (print_fract == 0) or (iters % int(print_fract*epochs) == 0):
            print(f"({time_str}) - Iter#: {iters:3d} / {epochs:3d}, "+\
                  f"Cost: {obj_func_eval:.6f}")

    return callback_func
```

callback function

8

```
regressor.fit(X_train, y_train)
elapsed = time.time() - start
time_str = time.strftime("%H:%M:%S", time.gmtime(elapsed))

### Retrieve training cost and params as arrays
objfun_vals = np.array(objfun_vals)
params_vals = np.array(params_vals)
```

callback info

Estimator [5] is a class collecting information about the model, i.e. its *circuit*, *input parameters* and *weight parameters*, definition of *observables* used to measure the circuit, a *gradient method*, and a specific *estimator primitive*.

Regressor [6] is an optimisation environment, which specifies an *optimiser*, a *loss function*, and the model *initial parameters*. We can access, save and print or plot, all intermediate optimisation steps via a *callback function* [8].

Qiskit ML provides a utility *function "fit"* which executes a *training loop* [7], folding into one: a *forward* step which applies the model with its current parameters to training data, *cost calculation*, and a *backward* step to improve the model parameters.

Quantum model performance:
Scoring your quantum model

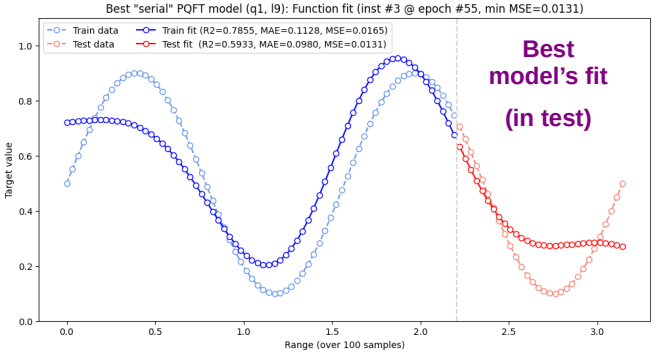
In model training we rely on the specific loss function to guide the optimiser, e.g. L2Loss (MSE cost).

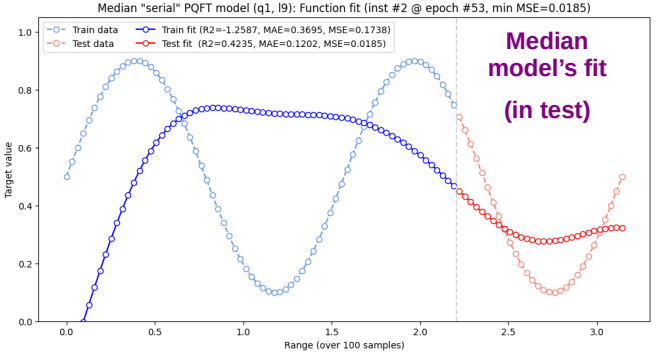
In training, the costs and the model parameters for all optimisation steps are saved. Later, the parameters values are assigned to the weights of the circuit, which is then applied to both training and test data, and results scored using variety of metrics (e.g. MAE and R^2).

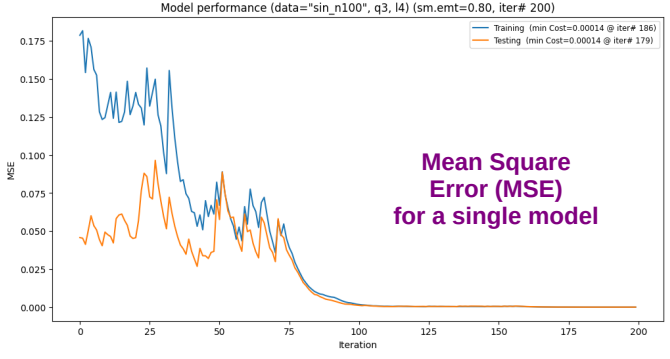
However, as a quantum model performance is highly sensitive to its initialisation, it is also advisable to run multiple, differently initialised, instances of the same model. Subsequently analyse distribution of their performance results (here we present only 5 instances of the same model in identical meta-parameter configurations).

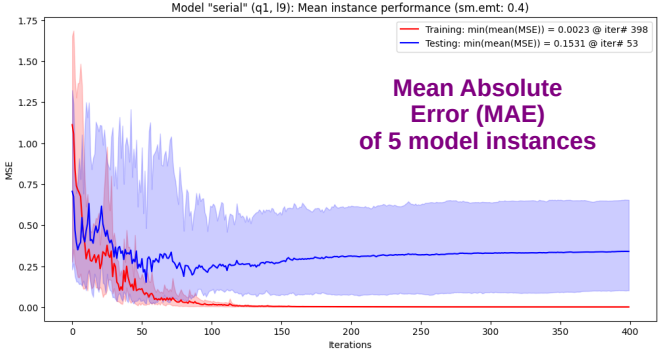
When doing so, it is also possible to present the level of model's fit to data, depending on it best, median or worst instance performance.

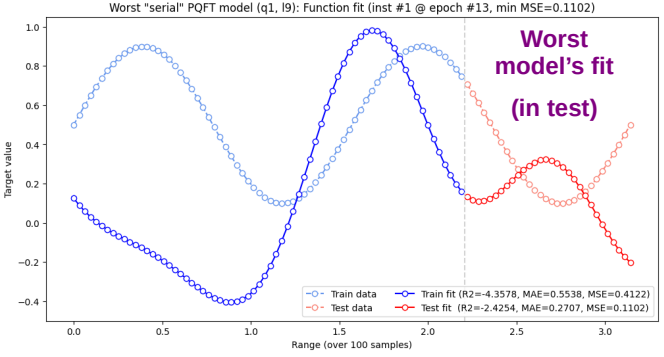
In this way, our performance assessment is not subject to our luck or the lack thereof, and most importantly we can provide an honest and unbiased reporting on the model's fitness for its practical deployment.







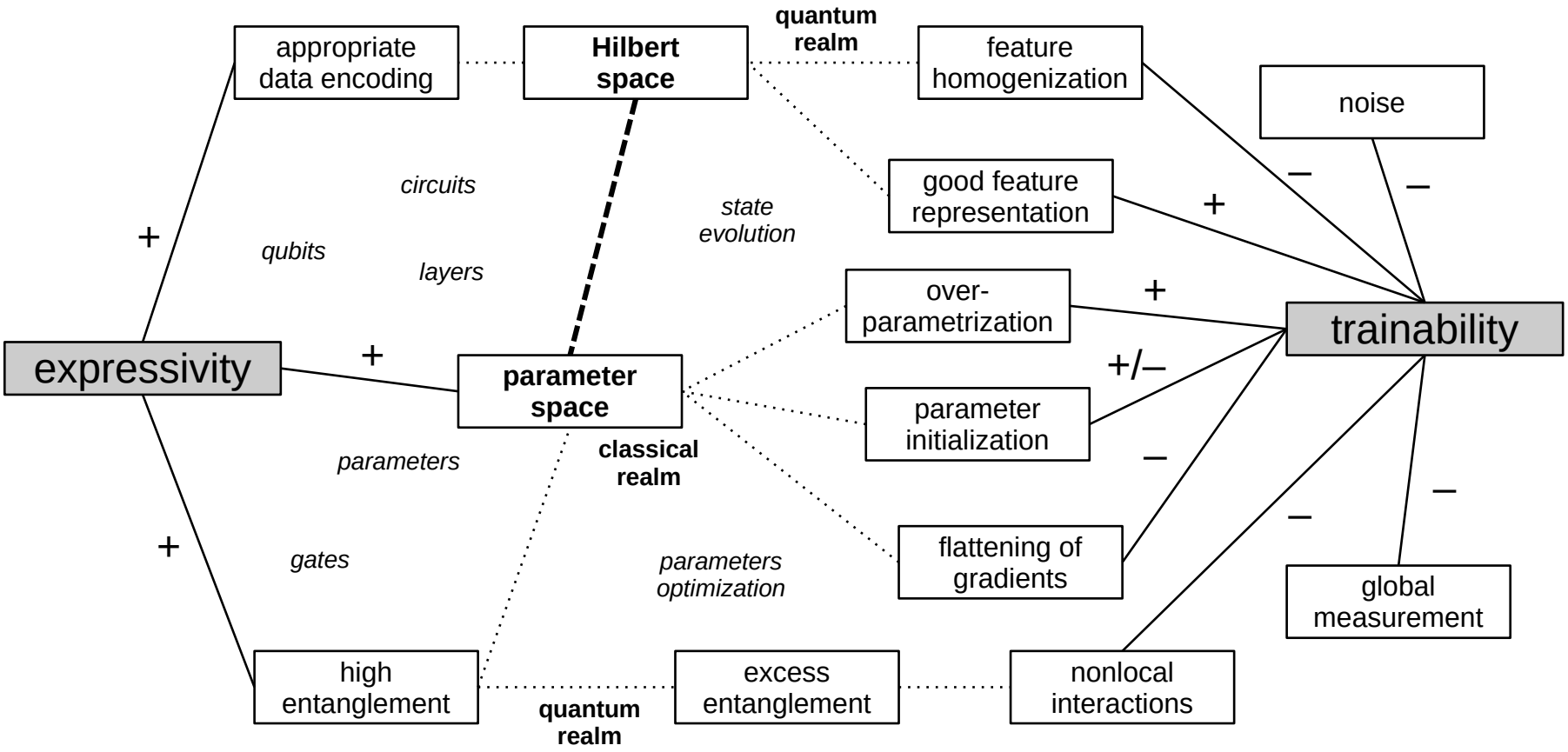




Expressivity vs. Trainability

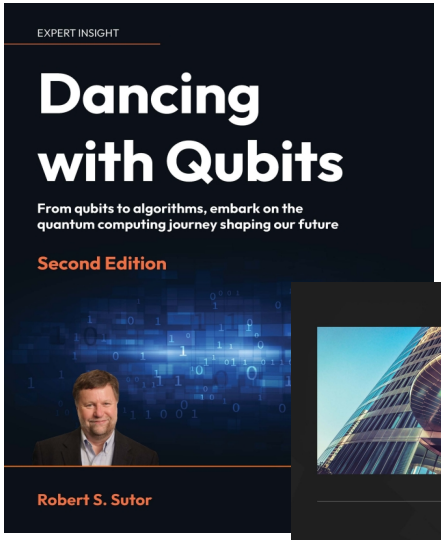
Model expressivity: ability to effectively represent time series data in quantum space.

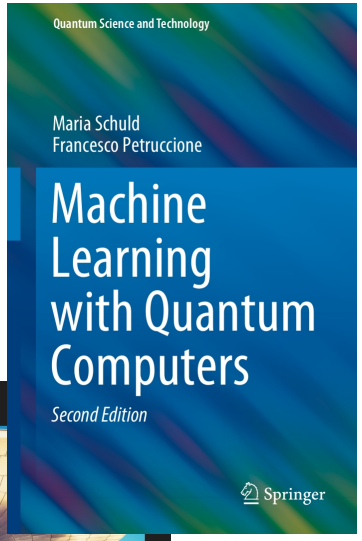
Model trainability: capacity to learn and generalise for predictive accuracy and efficiency in the process of model optimisation.

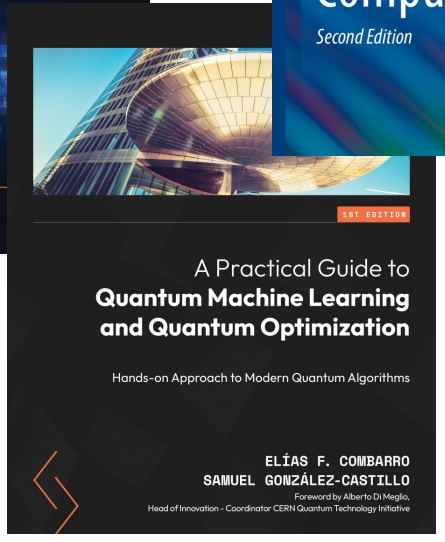


Recommended reading

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.

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 superconduct-

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.

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.

II. DESIGN STRUCTURE

This section outlines the role of Qiskit ML within the broader computing software stack, as illustrated in Fig. 1. Qiskit ML sits at the application level, providing a suite of tools and algorithms that leverage the power of quantum computation for machine learning tasks. It acts as a bridge between high-level machine learning concepts and the underlying quantum hardware or simulators. The design of Qiskit ML prioritises extensibility, modularity and an intuitive user interface, leading to a software package primed for rapid experimentation and prototyping of new and existing QML protocols. The high-level structure includes Quantum Neural Networks (QNNs), Variational Quantum Classifier (VQC) and Regressor (VQR) and Quantum Kernel (QK) Methods for Support Vector Machines, among other QML algorithms. These are provided through workflows shown in the Unified Modelling Language (UML) diagram in Fig. 1 and summarised in Table I.

Embedded within the wider Qiskit ecosystem, Qiskit 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

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.

2.1

Qiskit Team

Today, we're excited to announce the release of Qiskit SDK v2.1! The first minor release of the Qiskit SDK v2.x series brings performance improvements and exciting new capabilities designed to enable near-term demonstrations of quantum advantage.

The dawn of quantum advantage is fast approaching, and we predict the world will have incontrovertible proof of practical quantum advantage by the end of 2026. Continued collaboration between the quantum and high-performance computing (HPC) communities will play an essential role in making that happen, which is why we've focused so much of our recent development efforts on **extending the Qiskit SDK's C API support.**

Thank you!

Any questions?

Images from Unsplash and Wikipedia

Enquanted is being somewhere in-between Enchanted and Entangled