

Secrets revealed in this session:

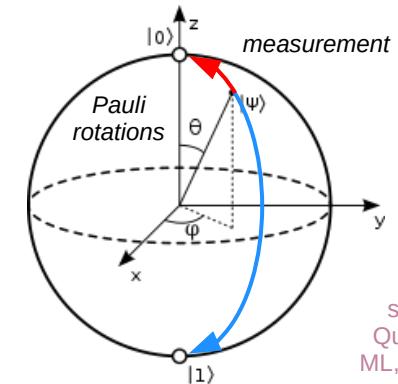
**To explore the practical aspects
of building quantum machine
learning models and their
optimisation**



QML and its aims
Parameterised circuits
Variational quantum algorithms
Data encoding / angle encoding
State measurement
Ansatz design and training
Model geometry and gradients
Parameters optimisation
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, Australia



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

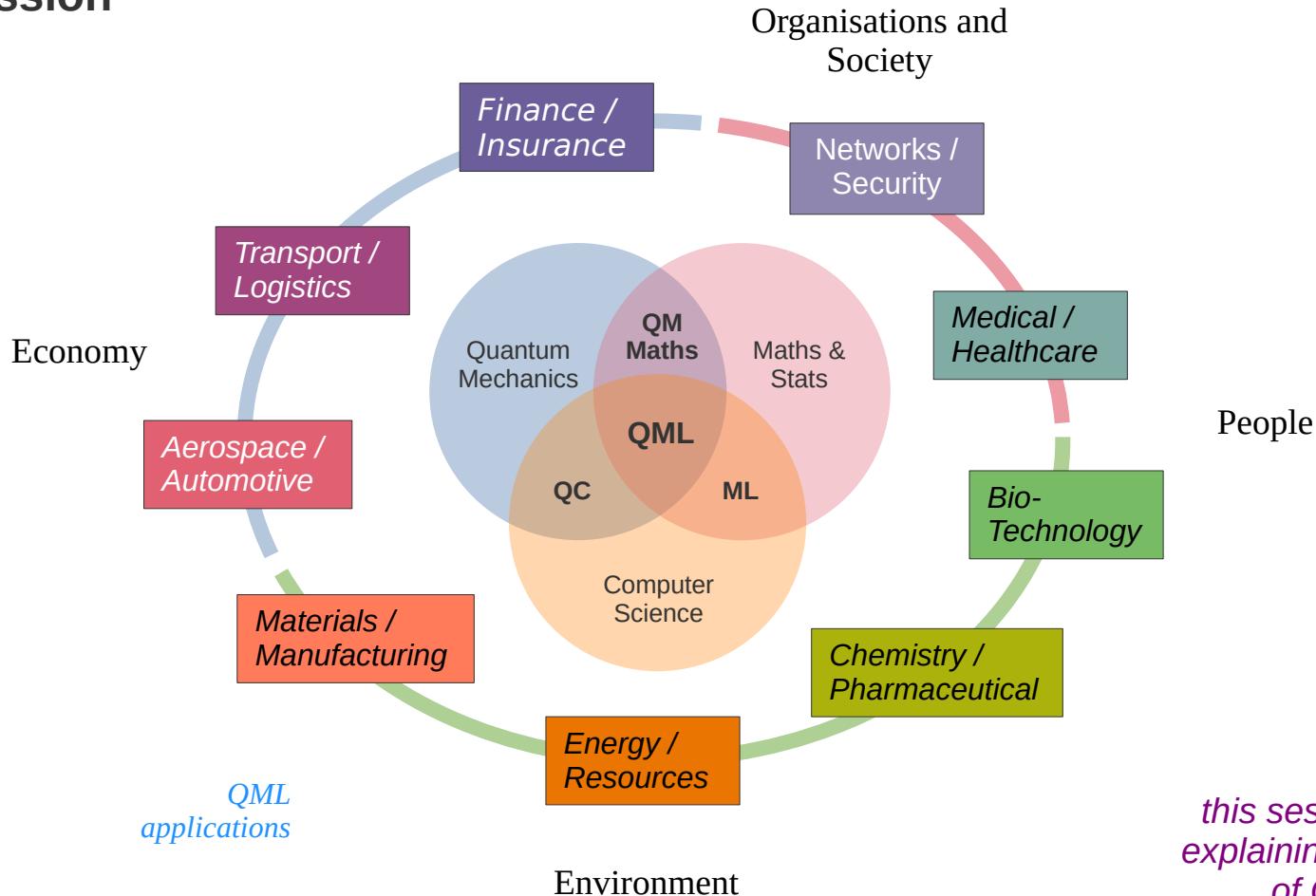
Quantum ML

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

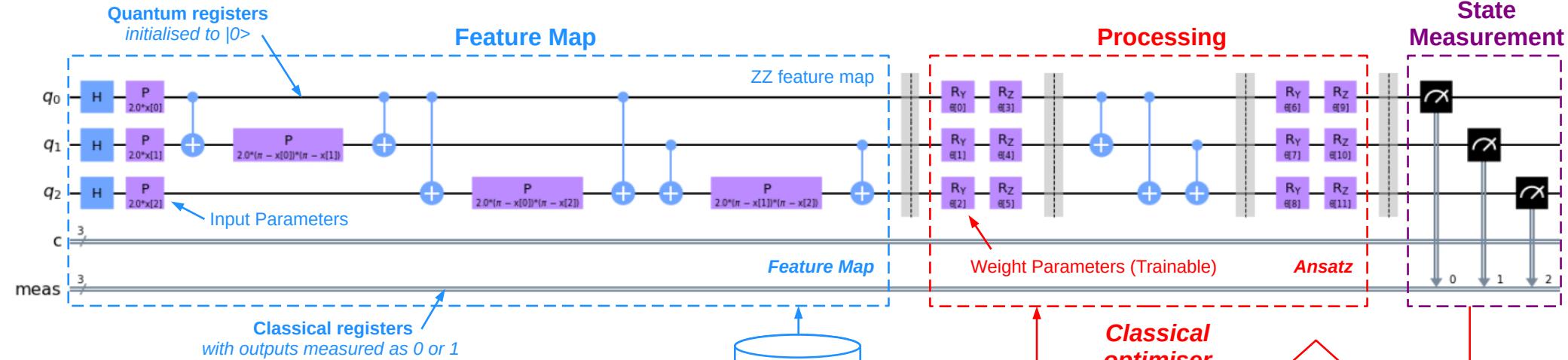


Jacob Cybulski, Founder
Enquantum, Australia



Variational Quantum Circuits and Variational Quantum Algorithms

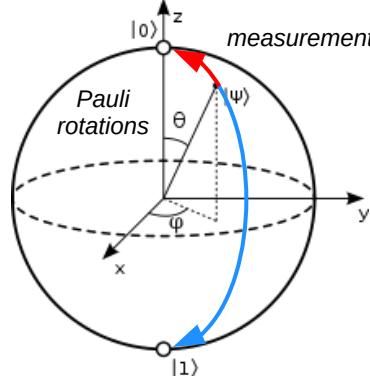
Variational quantum circuits are not executable!
They must first be instantiated, i.e. all of their input and weight parameters must be assigned values!



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).

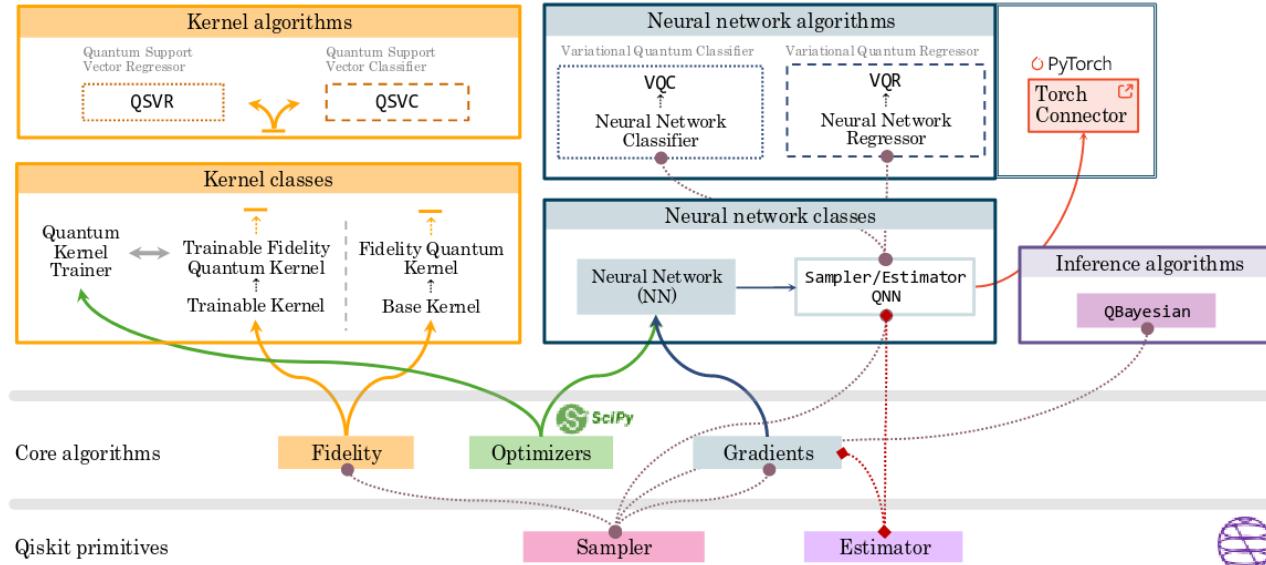
Typically (but now always), the circuit consists of three blocks:

- 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.

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.

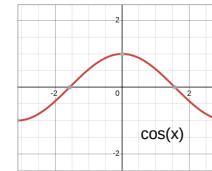
The rotation operators are always available in a quantum platform API, e.g. *Rx*, *Ry*, *Rz*, *P* or *U* (*xyz*).

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

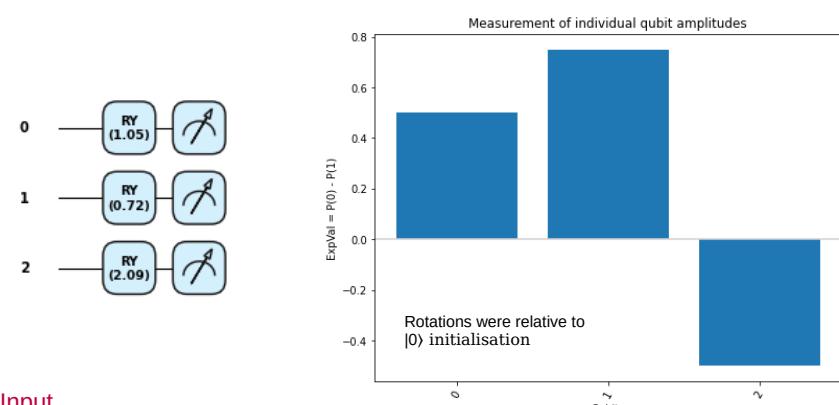
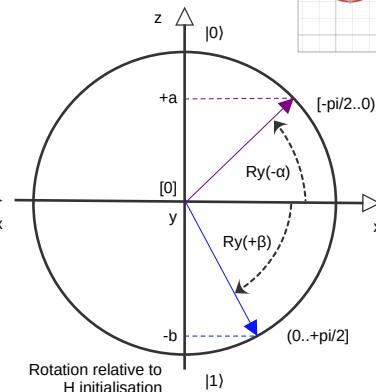
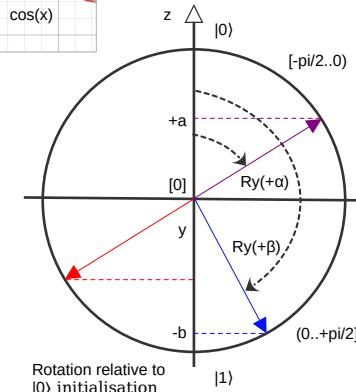
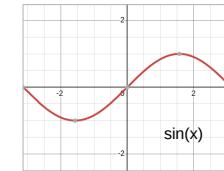
Rotations are *relative to a specific qubit state*, commonly starting at $|0\rangle$ state, or $(|0\rangle+|1\rangle)/\sqrt{2}$, which require qubits to be initialised in these states.

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.



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:
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]

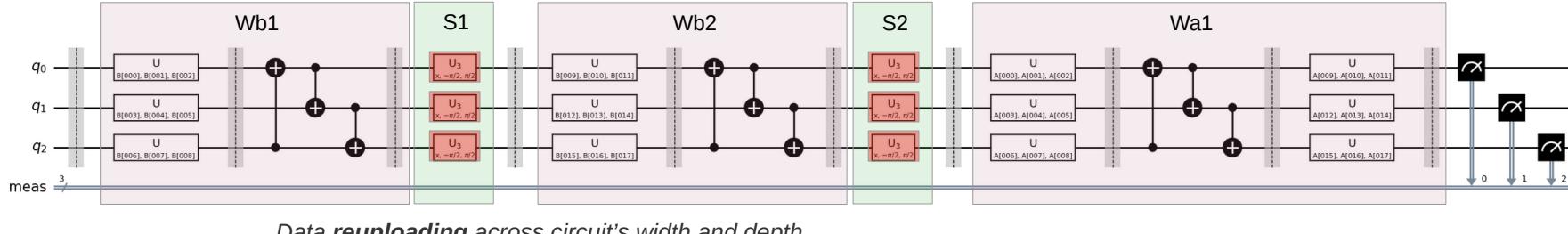
Ansatz design and training

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** !*



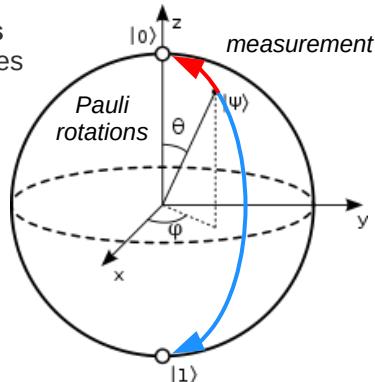
feature maps vary in:
structure and function (!!?)

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.

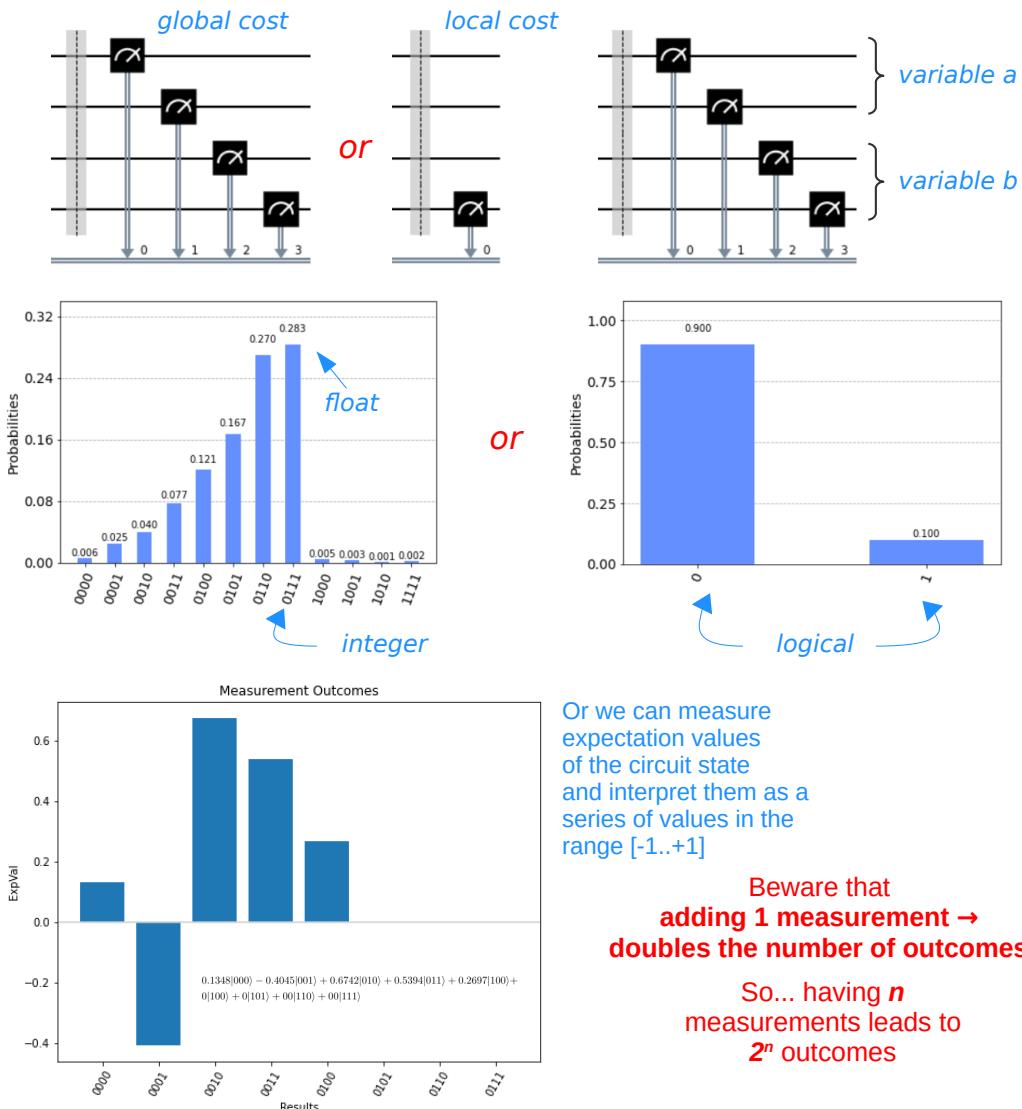
- all qubits (global cost / measurement)
- a few selected qubits (local cost / measurement)
- groups of qubits (each as a variable value)

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. $|0^n\rangle$)



Or we can measure expectation values of the circuit state and interpret them as a series of values in the range [-1..+1]

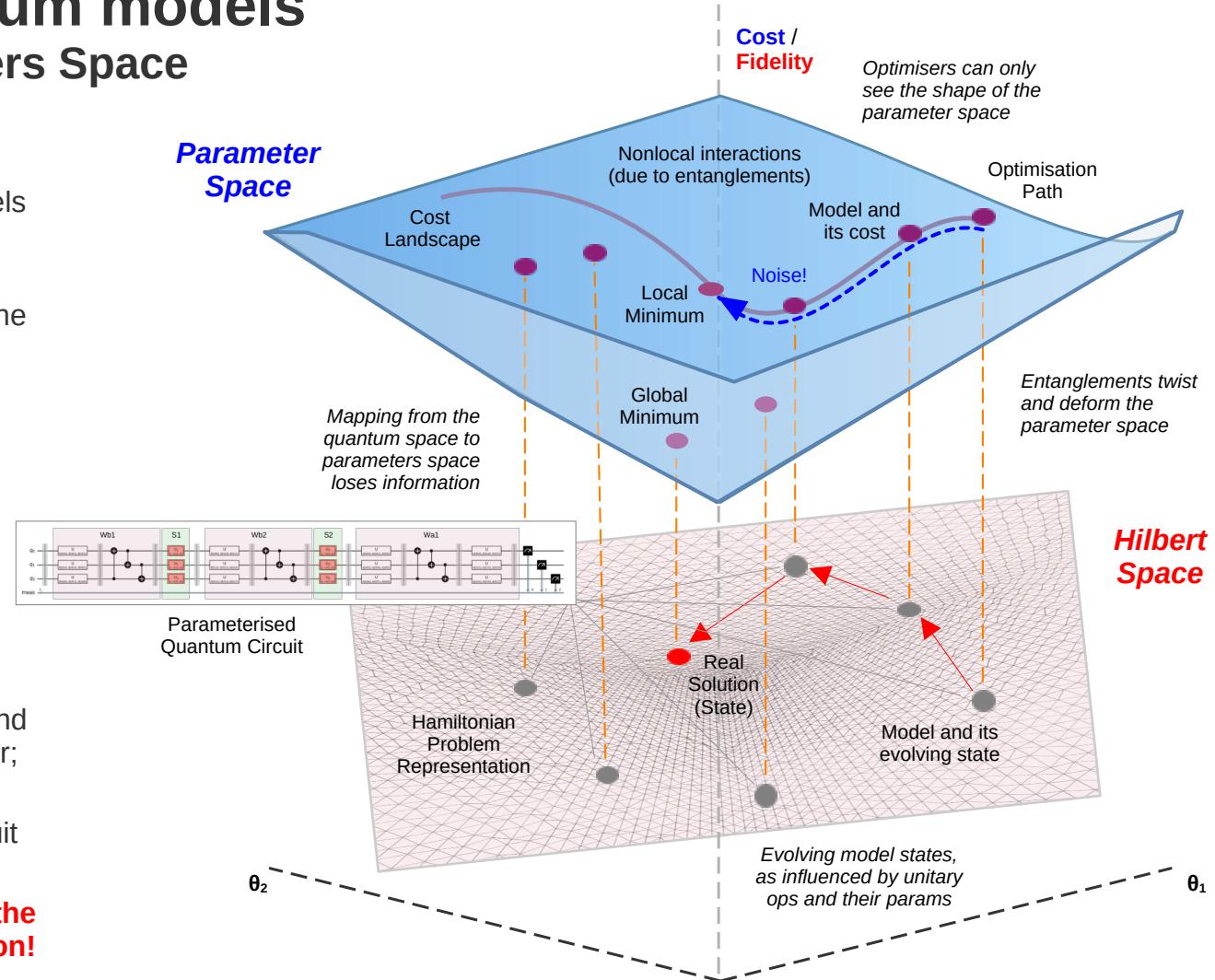
Beware that
adding 1 measurement →
doubles the number of outcomes!

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

Working with quantum models

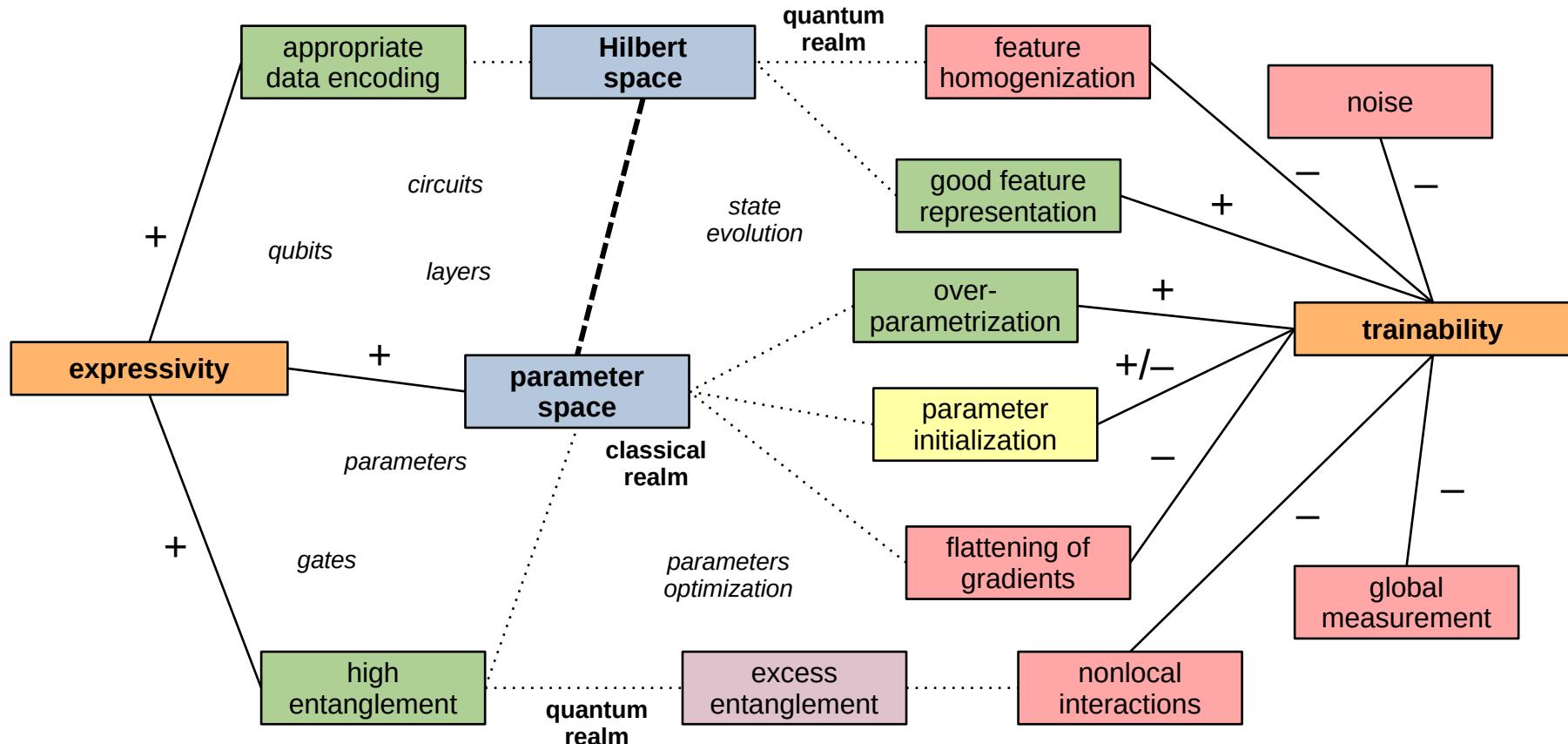
Hilbert Space vs Parameters Space

- **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!**



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.



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 curve fitting):

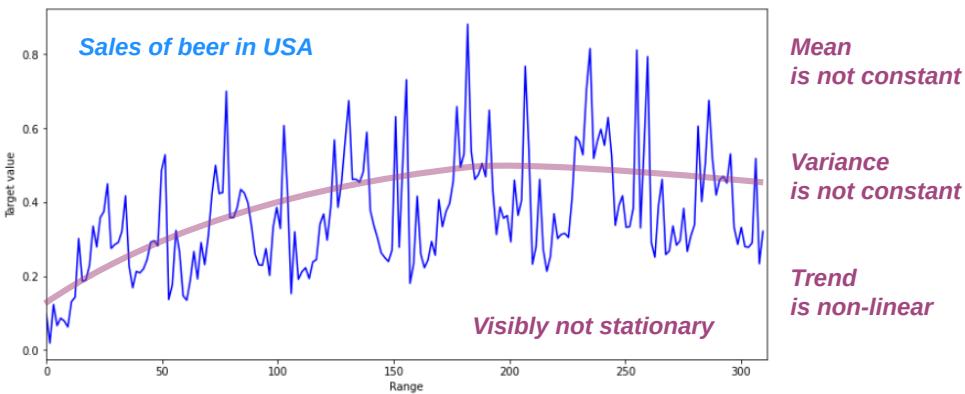
- Add ML 0.8.3 package to Qiskit 1.4.4 (Python 3.11)
- Create a few simple models to fit samples of data
- Learn to initialise model weights
- Learn the interaction b|n data encoding and ansatz
- Understand how observables / measurements work
- Explore the impact of ansatz structure on performance
- **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!

QML for time series analysis

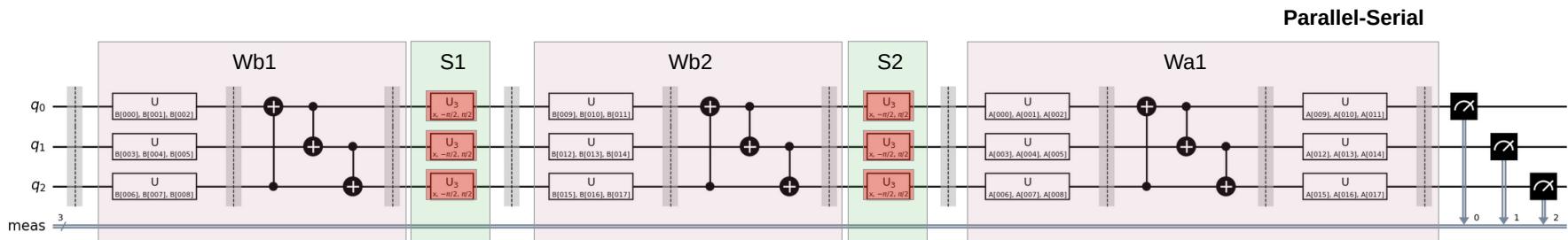
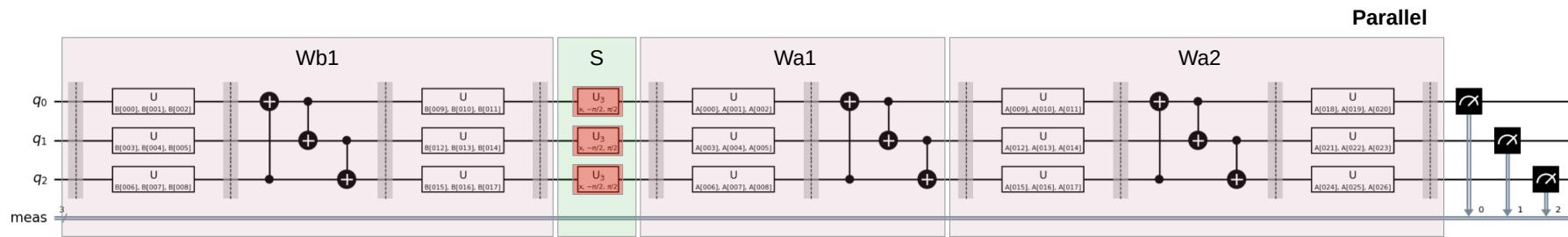
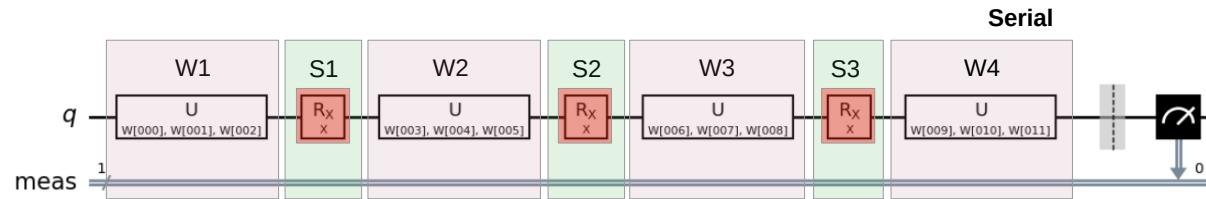
- Time series (TS) analysis aims to *identify patterns* in historical time data and to *create forecasts* of what data is likely to be collected in the future
- *Many TS applications*, including heart monitoring, weather forecasts, machine condition monitoring, etc.
- Time series can be *univariate*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



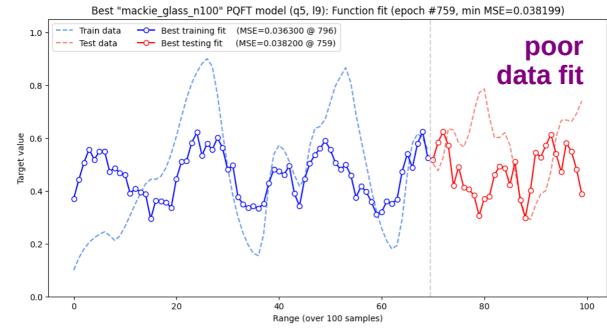
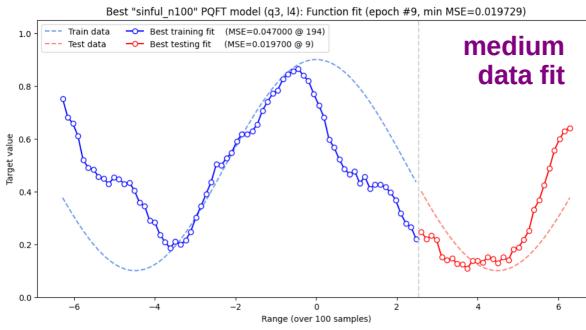
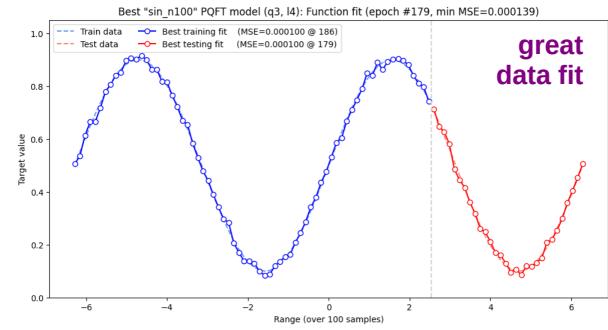
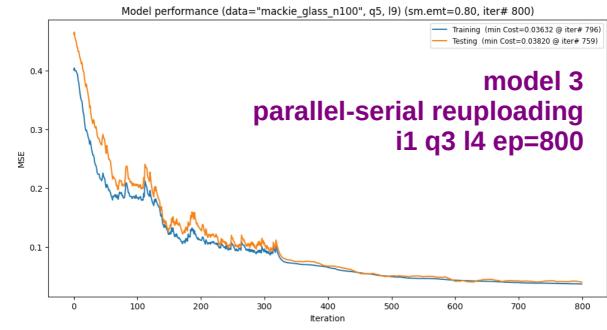
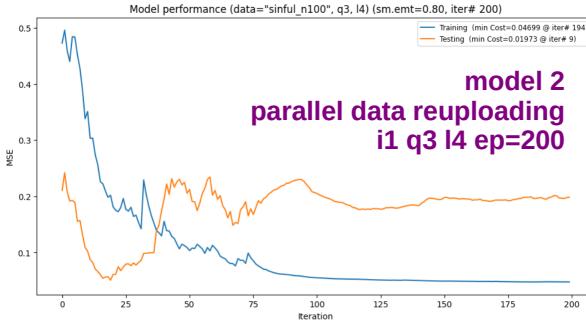
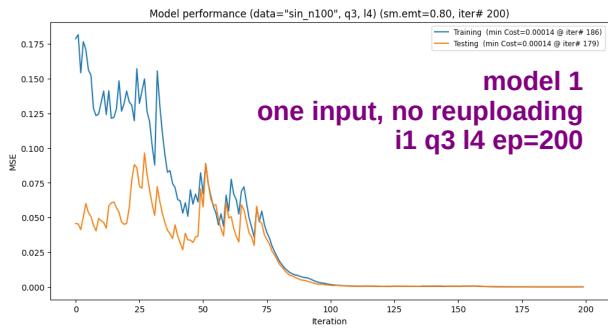
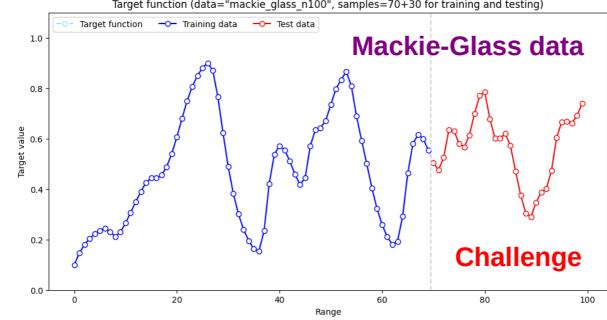
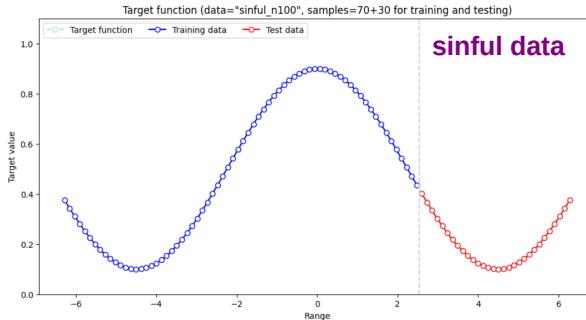
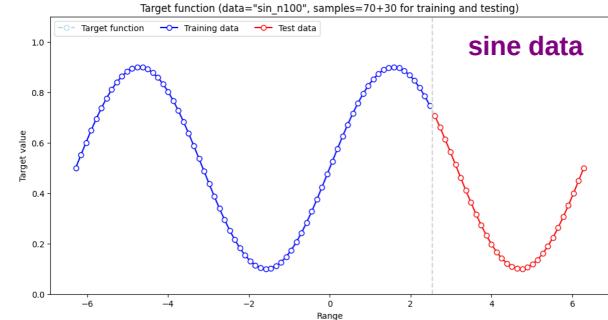
Quantum time series analysis is hard!

- 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!

Curve-fitting models



Data and some results



Approach to quantum TS modelling is data dependent!

Qiskit circuit creation

```
##### 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
```

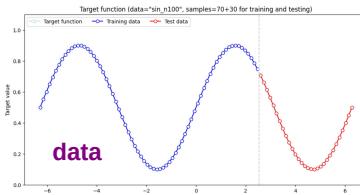
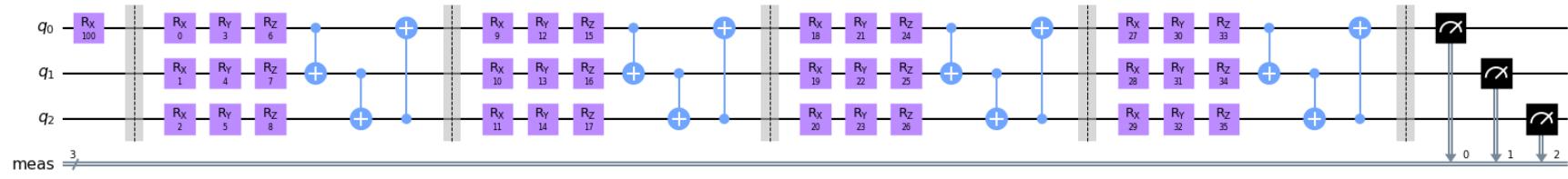
Model settings for models with 1 input

```
inputs_no = 1      # Number of inputs
qubits_no = 3       # Number of qubits
layers_no = 4       # Number of layers
epochs = 200        # Training epochs
shots = 1000        # Ignored
print_fract = 0.1   # Frequency of printing
weights_scaler = 1.00 # Scale factor for weights
```

model circuit

and its
meta-params

2



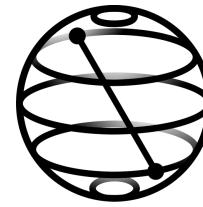
Sample data needs to be prepared, cleaned and split into training and test partitions.

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.



Qiskit

```
mfun = qnn_model_1
loss_fun = L2Loss()
optimizer = COBYLA(maxiter=epochs)
```

Select model, loss
function and optimiser

3

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

create model circuit
calculate initial weights

4

Training a simple Qiskit estimator

```
### 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),
)
```

```
### 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
)

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

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

```
regressor.fit(x_train, y_train)
```

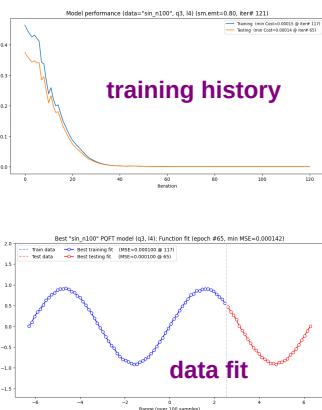
```
Model training started
training log
(00:00:00) - Iter#: 0 / 200, Cost: 0.178608
(00:00:01) - Iter#: 20 / 200, Cost: 0.176996
(00:00:02) - Iter#: 40 / 200, Cost: 0.028761
(00:00:03) - Iter#: 60 / 200, Cost: 0.008521
(00:00:05) - Iter#: 80 / 200, Cost: 0.004425
(00:00:06) - Iter#: 100 / 200, Cost: 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
```

5 estimator

6 regressor

7 training loop



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

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}, +\n'
                  f'Cost: {obj_func_eval:.6f}')

    return callback_func
```

Estimator [5] which computes the the *circuit* expectation values, with respect to given *observables*, and considering the circuit's *input parameters* and *weight parameters*, a *gradient method*, and an *estimator primitive* (hardware specific)

Regressor [6] which predicts continuous values using 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* pass which applies the model with its current parameters to training data, *cost calculation*, and a *backward* pass to improve the model parameters.

callback function

8

Quantum model performance: Scoring your quantum model

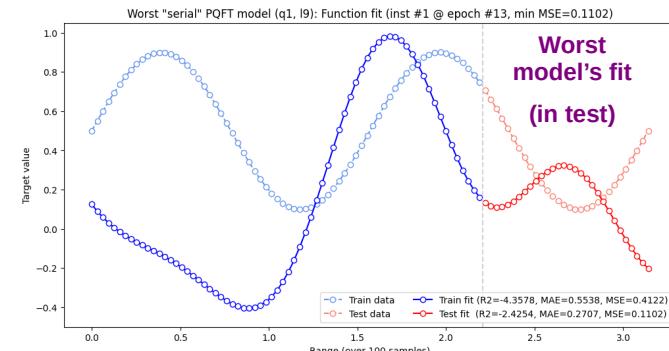
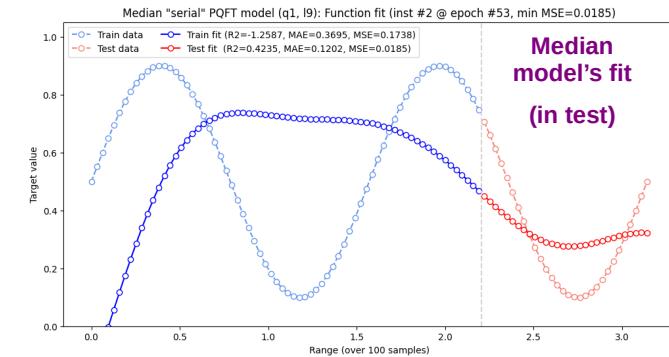
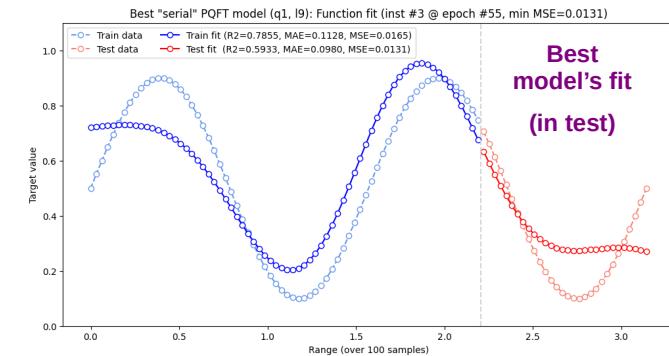
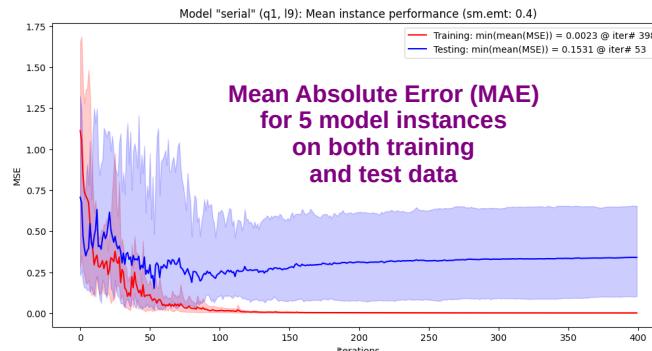
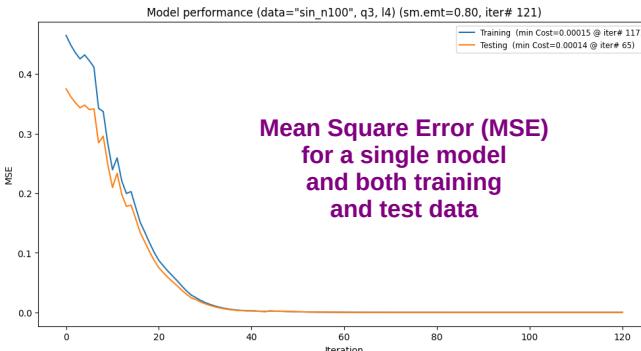
Quantum model training relies on the training data and a loss function to guide the optimiser, e.g. L2Loss (MSE cost), however, other performance metrics may also be needed, e.g. MSE, MAE or R^2 , calculated for training, validation and test data.

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).

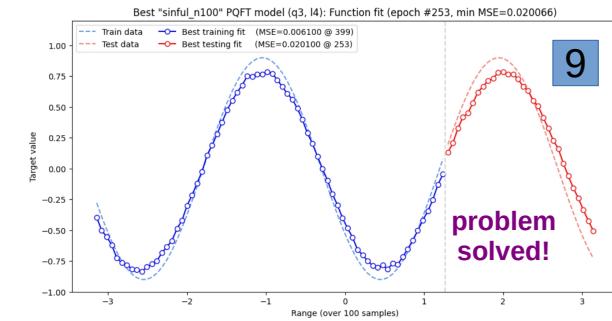
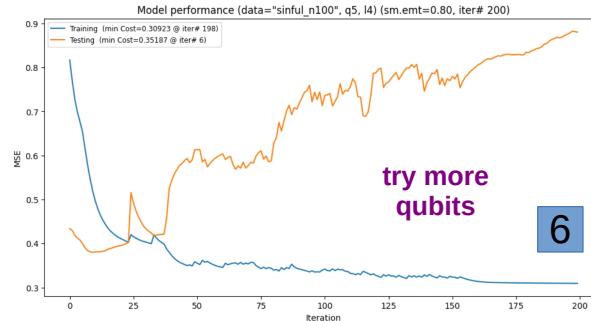
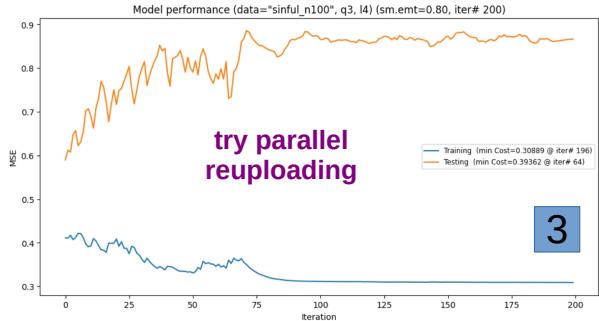
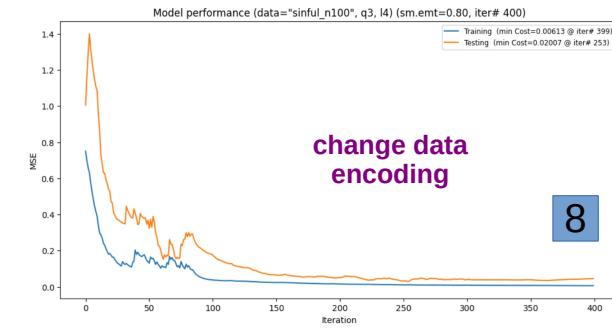
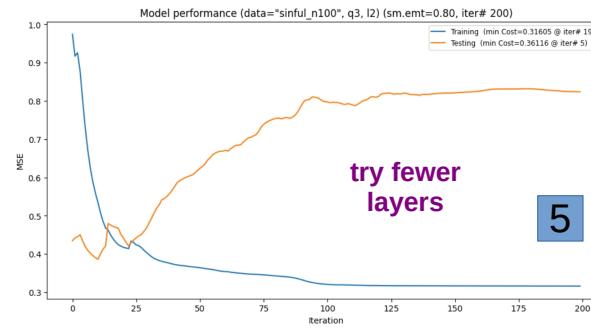
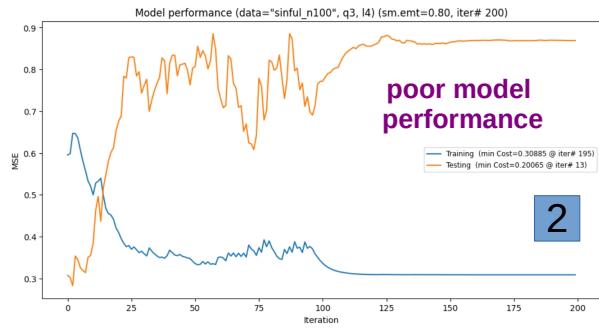
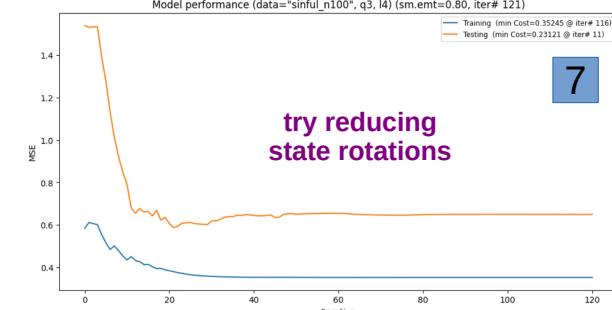
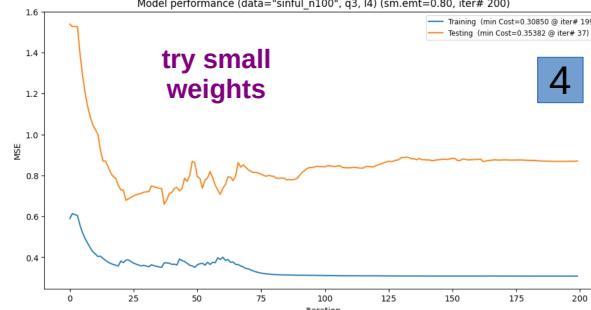
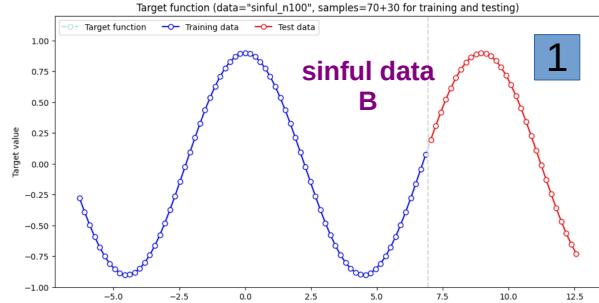
However, as a quantum model performance is highly sensitive to its initialisation, it is also advisable to run multiple, differently initialised, instances of the same model. Subsequently we can analyse a distribution of their performance results, e.g. here we present 5 instances of the same model with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

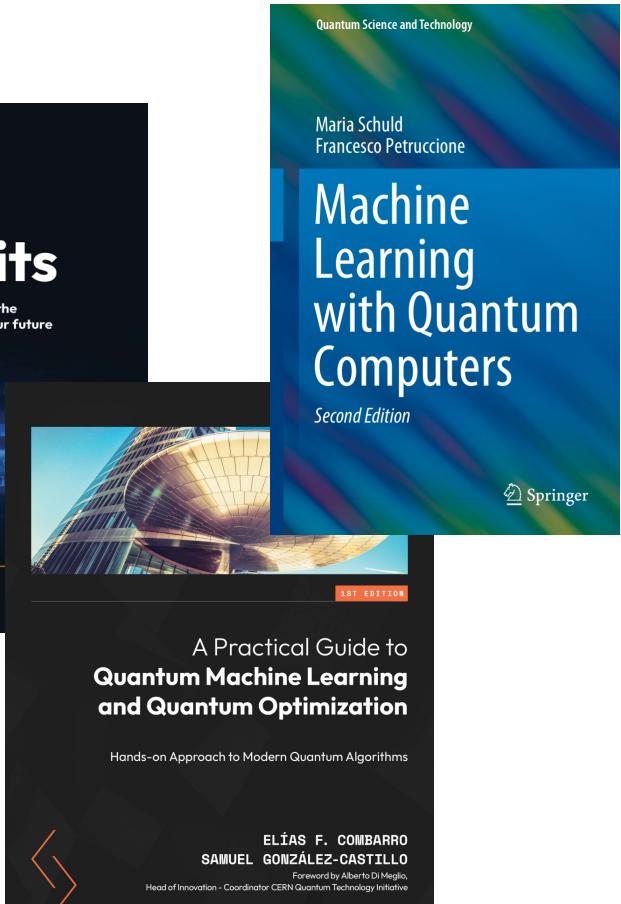
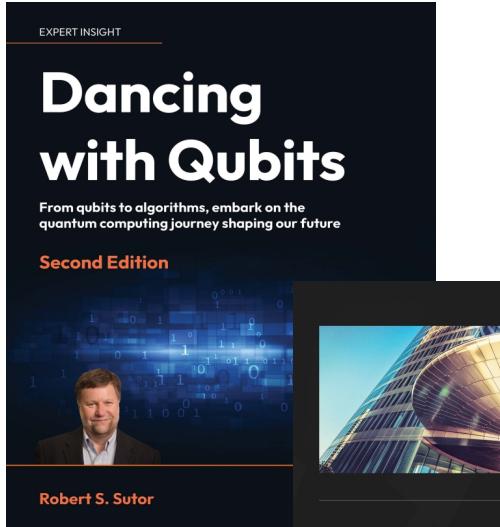


In search of solution!



Task: improve a curve-fitting model for the new dataset

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 building block of 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 function a quantum lever a very work and pressio system end.

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 must be solved on a quantum computer. This step involves translating the classical problem to the quantum domain, for example Fermion to qubit mapping [31, 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 generated using a standard library of gates, represented using intermediate language constructs such as OpenQASM [2].

The transpiler translates 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 technology, the ISA may include controls, clock signals, and memory access.

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

Today, we're excited to announce the release of Qiskit SDK v2.1! The first major 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 expect the world will need a powerful proof of practical quantum advantage by the end of 2024. 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.

2.1

Thank you!

Any questions?

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