Secrets revealed in this session:

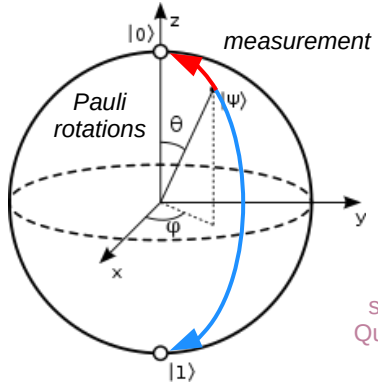
To gain experience to evaluate design and implementation options for the development of quantum estimation models

QML process
Quantum estimation
Linear models
Data preparation and partitioning
Fixing the random seed
Model creation and training
Chasing the performance targets
Scoring the model
Never trust your luck!
Don't get cocky
Avoiding barren plateaus
Model diagnostics -
residual analysis
Barriers to quantum estimation

Quantum Machine Learning

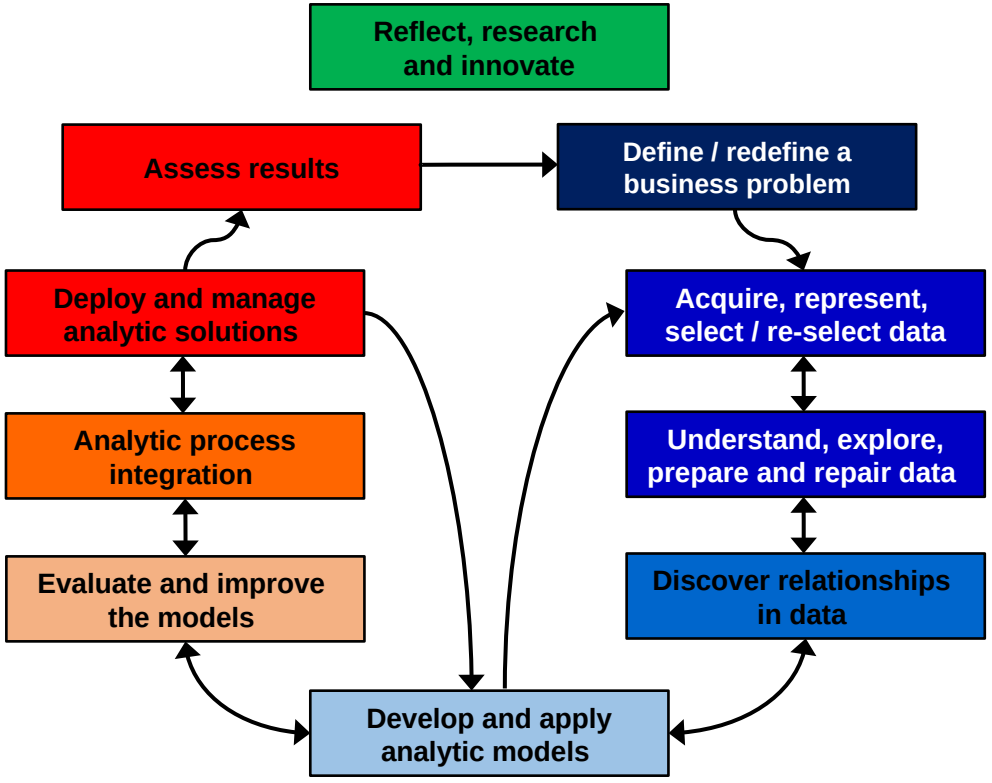
Quantum Estimators

Jacob L. Cybulski
Enquanted, Melbourne, Australia



We will assume
some knowledge of
Quantum Computing
ML and Python

Process of quantum model development



Define a business problem – Formulate a business problem and specify requirements for its solution in terms of decisions to be turned into business actions.

Understand and prepare data – Select a data sample. Explore and understand attributes characteristics. Deal with missing values and outliers. Clean, transform, convert and select attributes. Reduce data dimensionality if needed.

Discover relationships in data – Explore, visualise and understand relationships between data features. Determine targets, labels and their predictors.

Create analytic models – Evaluate alternative quantum models and algorithms to suit the problem and data. Study the models' characteristics, their strengths and weaknesses. Select and build the most promising.

Evaluate and improve the models – Validate and test the model for its ability to predict or explain. Evaluate the model training and test performance. Tune the model to optimise its performance. Interpret and report results.

Analytic process integration – Integrate pre-processing, exploratory and predictive analytic elements and visualisations into a complete analytic process.

Deploy, manage and assess analytic solutions – Embed the final quantum model and classical analytic process in a business application. Apply the process to live data. Use the results to support business decisions and actions. Measure and assess the model performance on real data. Reflect, research and innovate.

Classical estimation / Quantum estimation

Estimation models (or estimators) in machine learning are used to predict continuous numerical outputs (or targets), in contrast to classification models, which predict discrete labels. Such models are able to estimate quantities based on input features.

Classical estimation models include:

- ***Linear Regression Models***
  (Ordinary Least Squares Regression, Ridge and Lasso Regression)
- ***Non-Linear Regression Models***
  (Polynomial Regression, Support Vector Regression)
- ***Bayesian Models***
  (Bayesian Regression, Gaussian Processes)
- ***Neural Networks***
  (Feedforward Neural Networks, Recurrent Neural Networks)
- ***Time Series Forecasting Models***
- ***Ensemble Models***
- ***Etc.***

Quantum estimation models adopt the principles of quantum mechanics to enhance the precision, efficiency, or computational power of estimation tasks. These models are especially useful in situations where classical estimation methods face limitations, such as:

- *high-dimensional data*
- *noisy environments, or*
- *highly complex calculations.*

Applications of classical and quantum estimators:

- ***Finance:***
  Risk assessment, stock price prediction.
- ***Healthcare:***
  Disease progression estimation, drug effectiveness.
- ***Retail:***
  Demand forecasting, inventory optimization.
- ***Autonomous Vehicles:***
  Estimating distances, speeds.
- ***Energy:***
  Power consumption prediction.
- ***Etc.***

Quantum models capable of estimation tasks:

- ***Quantum Neural Networks (QNNs):***
  speedups for high-dimensional data.
- ***Quantum Kernel Methods:***
  encoding data into quantum Hilbert space.

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.

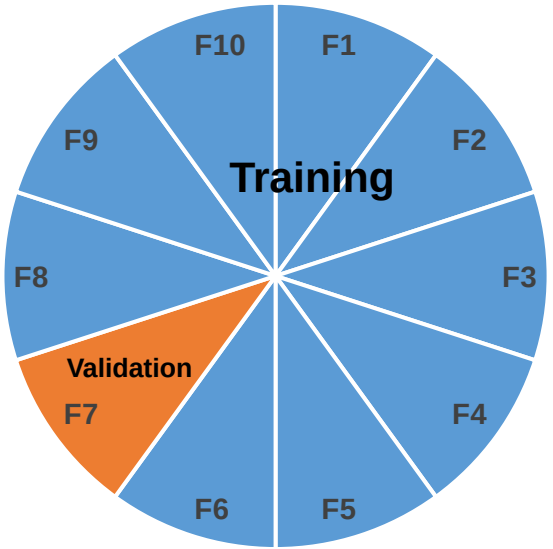
Commonly adopted approach is known as *holdout testing*, where we randomly split data into three partitions - one to be used for model *training*, one for its *validation* while improving the model, and one for *honest testing* on data never used before.

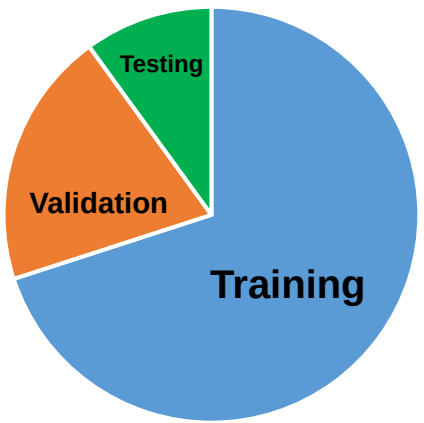
Holdout testing validates and test the model once only, assuming all data partitions as representative of the population, which may not be true.

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.

Linear pl Q. Models

They are not regression!

All quantum models are linear.
However, this one is also fully functional!

```python
### Define a quantum linear model functionally
### This is the PennyLane style of doing things
def qlinear(n_wires, scaler=1.0, basic=False):

    def _qlinear(weights, inputs):
        nonlocal n_wires, scaler, basic
        data_wires = list(range(n_wires))
        scaled_inputs = inputs * scaler

        # Standard angle encoding block
        qml.AngleEmbedding(scaled_inputs, wires=data_wires)

        # Ansatz with optional basic or strong entangling block
        if basic:
            qml.BasicEntanglerLayers(weights, wires=data_wires, rotation=qml.RY)
        else:
            qml.StronglyEntanglingLayers(weights, wires=data_wires)
        return qml.expval(qml.PauliZ(0))

    return _qlinear
```

quantum model

1

```python
### Check the model shape
### This is defined so that we could select and check the shape in one place
def qlinear_shape(n_wires, n_layers=1, basic=False):
    if basic:
        shape = qml.BasicEntanglerLayers.shape(n_layers=n_layers, n_wires=n_wires)
    else:
        shape = qml.StronglyEntanglingLayers.shape(n_layers=n_layers, n_wires=n_wires)
    return shape
```

shape calculation

2

```python
### Create a quantum linear model

# Prepare the quantum linear model and then its
qmodel = qlinear(n_wires, basic=basic_ent)
qlinear_model = qml.QNode(qmodel, dev)
shape = qlinear_shape(n_wires, n_layers=n_layers, basic=basic_ent)

# Plot the model circuit (needs weights and some input)
test_params = np.random.uniform(high=np.pi, size=shape, requires_grad=True)
draw_circuit(qlinear_model, scale=0.5,
             title=f'Quantum circuit "qlinear", shape={shape}, param#={np.prod(shape)}',
             level='device')(test_params, X_train_tens[0]) # level='device'/'gradient'
```

model creation

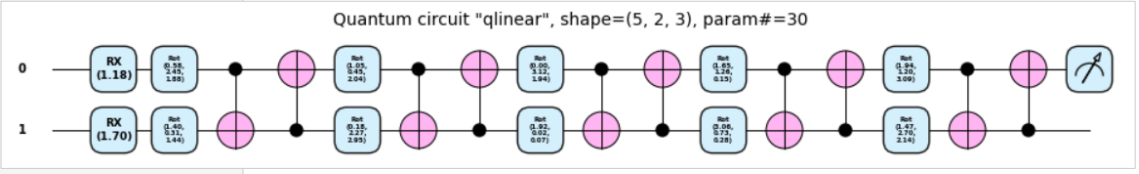
```python
### Define device, e.g.
#   default.qubit, lightning.qubit, lightning.gpu

sim = 'default.qubit'
dev = qml.device(sim, wires=n_wires, shots=shots)

print(f'Device to be used:  {sim}\n')
```

device spec

3



Here we present a simple linear quantum model developed in PennyLane.

It is implemented as a *factory function* “qlinear”, which creates a *closure* “_qlinear” able to access its own arguments, as well as the variables of the function that created it. This is especially useful when we define a model of a fixed parameter structure, but which requires additional information for processing.

In this implementation the closure is required to only accept input and weights, but needs to use information about the number of wires, as well as, the flag “basic” which determines the type of entanglement to be used by the circuit.

We also defined an equally flexible function that determined the shape of weights used by the closure.

Training a
quantum “linear” model

```
### Set the random seed
np.random.seed(seed)

### Select a PennyLane optimiser
opt = qml.AdamOptimizer(stepsize=0.5)

### Define the cost function
cost_fun = cost_fun_gen(qlinear_model, mse_cost)

### Initialise the model weights / parameters
params = np.random.uniform(high=np.pi, size=shape, requires_grad=True)

### Training loop
hist_cost = []
hist_params = []
prompt_fract = 0.1
start_time = time.time()
for iter in range(epochs):
    params, cost = \
        opt.step_and_cost(lambda p: cost_fun(p, X_train_tens, y_train_tens), params)
    hist_cost.append(float(cost))
    hist_params.append(params)
    elapsed_time = time.time()-start_time
    if (prompt_fract == 0) or (iter % int(prompt_fract*epochs) == 0):
        print(f'Iter: {iter:03d} ({int(elapsed_time):03d} sec) '+ \
              f'cost={np.round(cost, 6)}')
```

training loop

training log

```
Iter: 000 (000 sec) cost=[0.60859]
Iter: 010 (009 sec) cost=[0.063793]
Iter: 020 (017 sec) cost=[0.008793]
Iter: 030 (026 sec) cost=[0.003654]
Iter: 040 (034 sec) cost=[0.00226]
Iter: 050 (043 sec) cost=[0.001536]
Iter: 060 (051 sec) cost=[0.001596]
Iter: 070 (059 sec) cost=[0.001055]
Iter: 080 (068 sec) cost=[0.000869]
Iter: 090 (076 sec) cost=[0.000874]
```

```
epochs=100, min cost=0.000799 @ 99, time=084 secs
```

```
### Our own gradient-friendly loss function
def mse_cost(targets, predictions):
    cost = 0
    for l, p in zip(targets, predictions):
        cost = cost + (l - p) ** 2
    cost = cost / len(targets)
    return cost

### The cost function generator
def cost_fun_gen(model, cost_fun):
    def _cost_fun(params, inputs, targets):
        nonlocal model, cost_fun
        preds = [model(params, x) for x in inputs]
        return cost_fun(targets, preds)
    return _cost_fun
```

cost function

Training of a PL quantum model is structured the same way as in any other environment (e.g. Torch or Tensorflow).

It requires an *optimiser* (selected from PL), a *cost function* (must be differentiable), and the model *initial parameters* (must have the shape of the model weights).

PL provides a utility function “step_and_cost” which folds three optimisation steps into one, i.e. a *forward* step which applies the model with its current parameters to training data, *cost calculation* for this step, and a *backward* optimisation step to improve the model parameters.

In the process, the optimiser will run the model’s QNode (the *circuit*), which associates the model and a device, and optionally defines an *interface* to a gradient package (e.g. “torch”) and a *differentiation method* (e.g. "parameter-shift"). The *device* identifies the simulator (e.g. “default.qubit”) to be used in all calculations.

Available interfaces: Autograd, Jax, Tensorflow, Torch

Quantum model performance:
Scoring your quantum model

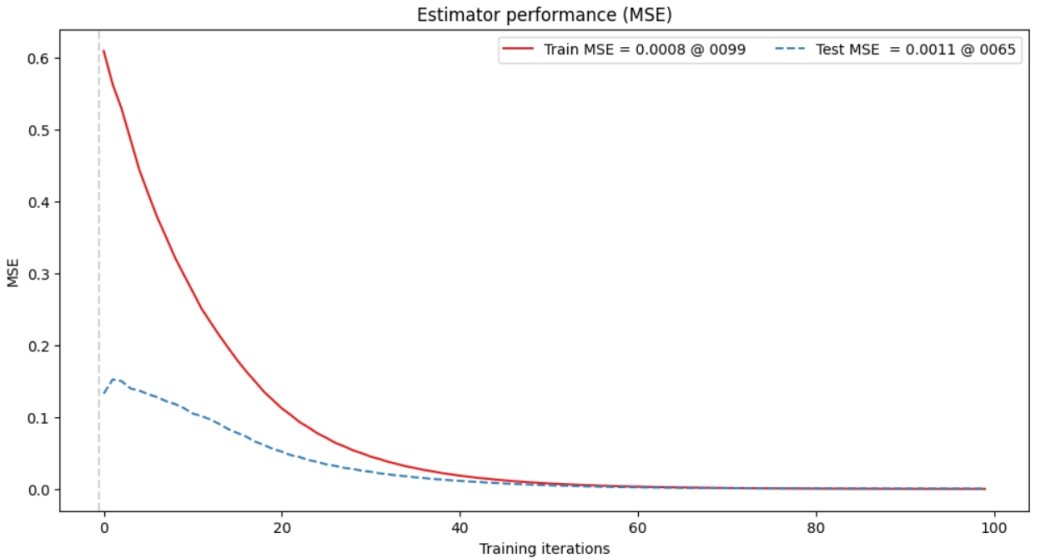
In model training we rely on the specific loss / cost function, e.g. MSE or MAE, to guide the optimiser.

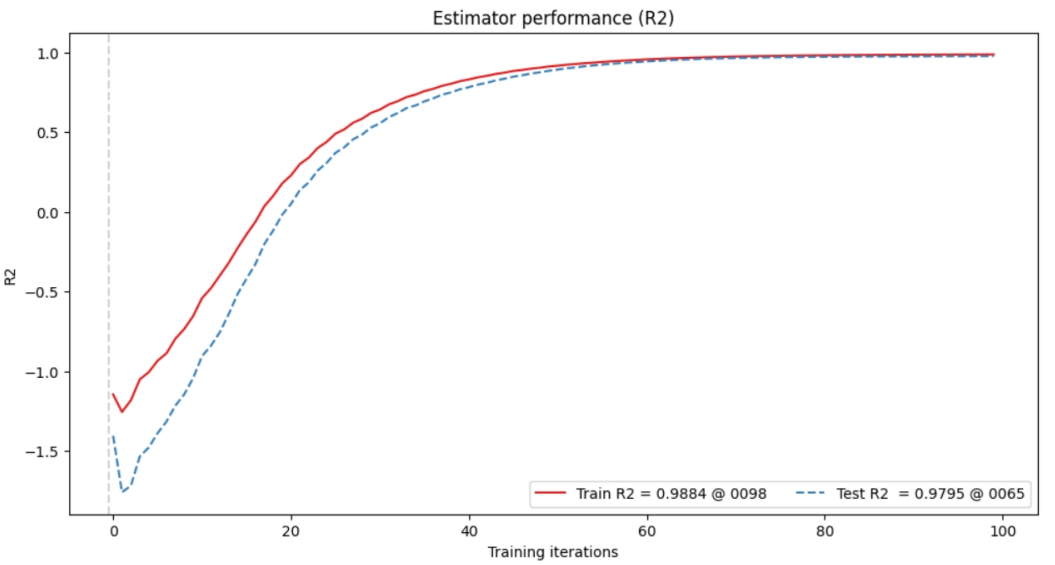
During training, the costs and the model parameters for all optimisation steps are saved for later use.

The lowest cost indicates the optimum model training parameters. However, the model performing best in training may not be the most generalisable.

So we can use all saved model parameters, to reconstruct the intermediate models and apply apply them to either training, validation or test data.

This allows calculation of other performance scores useful in further analysis to determine the parameters of the model that is optimum for the population and most suitable for its final deployment.





Never trust your luck: Example

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

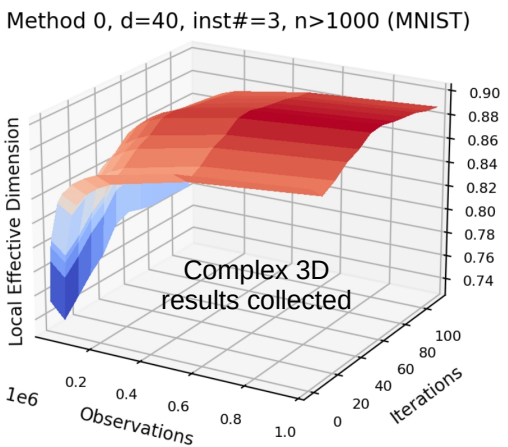
Global Effective Dimension (GED):

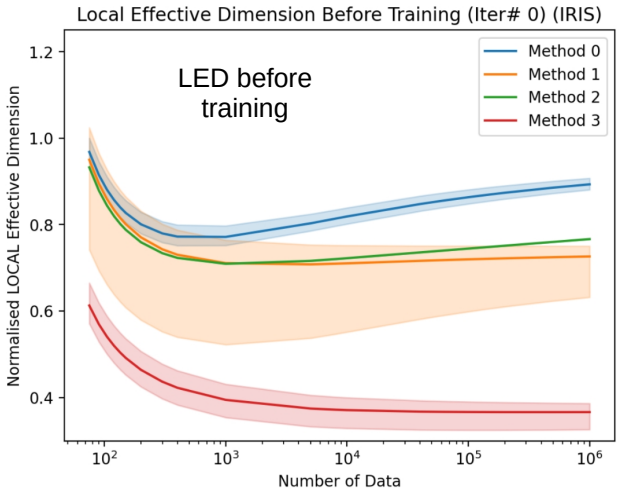
A static, probabilistic measure of the model complexity as the geometry of its entire parameter space.

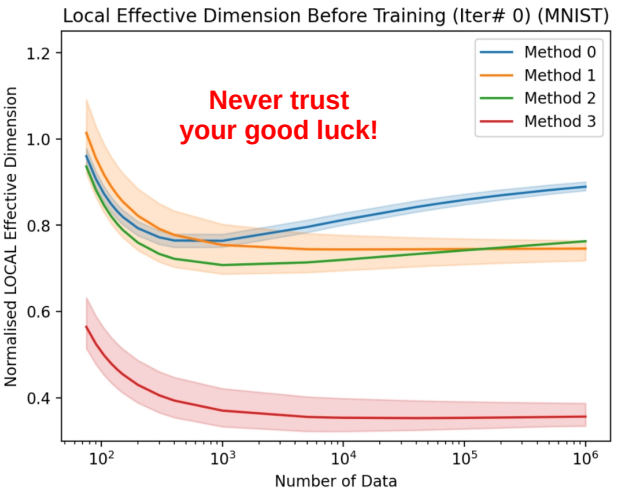
Local Effective Dimension (LED):

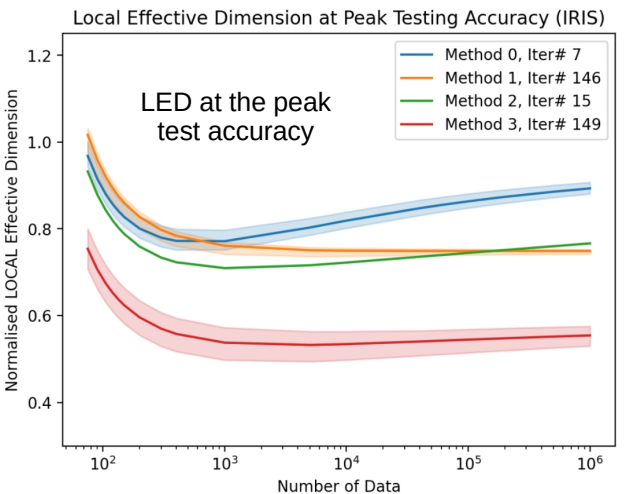
Dynamic geometric measure of the model complexity in training, derived from GED, but affected by data distribution and optimisation algorithm

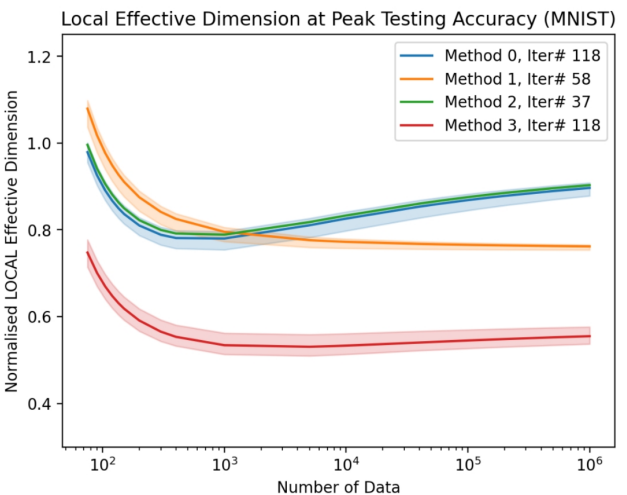
The study investigated how GED and LED correlated with **barren plateaus (BP)**.











This study used **two** different datasets - IRIS and MNIST, investigated **four** approaches to dealing with BPs, each required a **unique** quantum model, however, each model had **ten** runs **randomly initiated**.

Jacob L. Cybulski and Sebastian Zając (2024): "The Art of Data Encoding and Decoding for Quantum Time Series Analysis." 8th Intl Conf on *Quantum Techniques in Machine Learning,* University of Melbourne, Melbourne, Australia, 25-29 Nov 2024.

Don’t get cocky

Compare quantum vs classical model

QAE *CAE*

The experiments show:

The larger the QAE latent space, the better learning
(the accepted idea that reducing latent space helps abstraction is wrong)

There is an optimum depth for the QAE model.

PennyLane “minimum” hybrid models outperformed Qiskit models in training, but not in validation.

Within the limit of 1000 epochs, QAE matched CAE.

In general, QML models on simple tasks (such as DL AE) *do not outperform the classical models* – so to gain quantum advantage you need to pick the application very carefully.

USA beer sales (IRI) **Varying the latent space:** DL CAE model in PyTorch @ 1000 epochs

	Experiments		Training					Validation				
	Lat	Tr	TR2	TMSE	TRMSE	TMAE	TMAPE	VR2	VMSE	VRMSE	VMAE	VMAPE
Run												
0	8	0	0.9621	0.0087	0.0925	0.0716	0.0615	0.7475	0.0478	0.2105	0.1583	0.1444
1	7	1	0.9636	0.0086	0.0925	0.0683	0.0607	0.7491	0.0463	0.2016	0.1614	0.1431
2	6	2	0.9631	0.0081	0.0911	0.0708	0.0604	0.7547	0.0466	0.2133	0.1554	0.1443
3	5	3	0.9592	0.0085	0.0925	0.0710	0.0624	0.7468	0.0455	0.2133	0.1633	0.1467
4	4	4	0.9609	0.0088	0.0941	0.0713	0.0618	0.7668	0.0445	0.2120	0.1604	0.1445
5	3	5	0.9625	0.0092	0.0973	0.0701	0.0646	0.7461	0.0453	0.2146	0.1677	0.1464
6	2	6	0.9515	0.0121	0.1096	0.0815	0.0697	0.7144	0.0516	0.2264	0.1694	0.1504
7	1	7	0.8575	0.0321	0.1788	0.1274	0.1098	0.4706	0.0937	0.3012	0.2217	0.1859

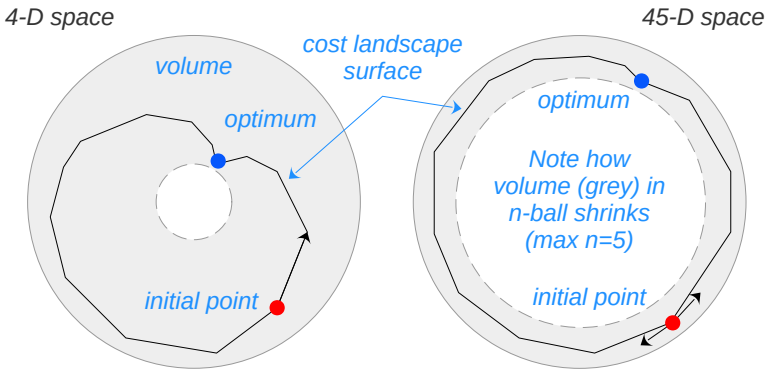
Varying the circuit depth: quantum model in PennyLane + PyTorch @ 1000 epochs

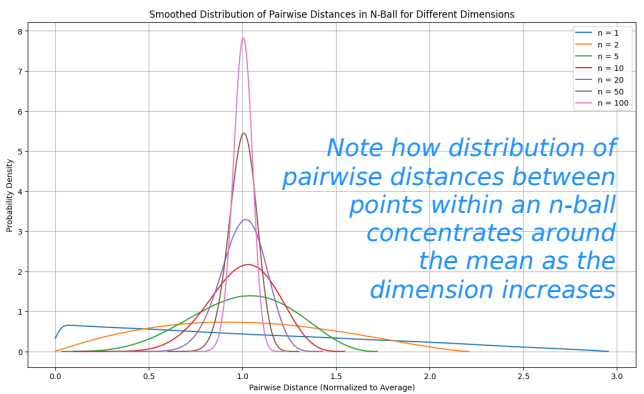
	Experiments				Training						Validation			
	Lay	Lat	Tr	Xtr	TR2	TMSE	TRMSE	TMAE	TMAPE	VR2	VMSE	VRMSE	VMAE	VMAPE
Run														
8	1	5	3	0	0.7663	0.0449	0.2112	0.1508	0.1285	0.1460	0.1019	0.3154	0.2192	0.2081
9	2	5	3	0	0.9635	0.0084	0.0910	0.0703	0.0652	0.6278	0.0475	0.2169	0.1656	0.1598
10	3	5	3	0	0.9589	0.0093	0.0953	0.0693	0.0631	0.6926	0.0400	0.1994	0.1470	0.1397
11	4	5	3	0	0.9644	0.0081	0.0885	0.0656	0.0592	0.6890	0.0413	0.2028	0.1545	0.1457
12	5	5	3	0	0.9572	0.0096	0.0971	0.0693	0.0624	0.7198	0.0386	0.1962	0.1474	0.1374
13	6	5	3	0	0.9528	0.0104	0.1015	0.0722	0.0642	0.6915	0.0408	0.2016	0.1531	0.1445
14	7	5	3	0	0.9499	0.0111	0.1052	0.0747	0.0659	0.6866	0.0412	0.2027	0.1502	0.1404
15	8	5	3	0	0.9525	0.0106	0.1027	0.0728	0.0649	0.7073	0.0400	0.1999	0.1503	0.1411

Varying the latent space: quantum model in PennyLane + PyTorch @ 1000 epochs

	Experiments				Training						Validation			
	Lay	Lat	Tr	Xtr	TR2	TMSE	TRMSE	TMAE	TMAPE	VR2	VMSE	VRMSE	VMAE	VMAPE
Run														
0	3	8	0	1	0.9732	0.0062	0.0770	0.0581	0.0541	0.7139	0.0417	0.2034	0.1545	0.1478
1	3	7	1	1	0.9736	0.0061	0.0764	0.0579	0.0545	0.7350	0.0373	0.1928	0.1467	0.1460
2	3	6	2	1	0.9667	0.0076	0.0864	0.0643	0.0602	0.6953	0.0438	0.2083	0.1518	0.1463
3	3	5	3	1	0.9540	0.0103	0.1003	0.0731	0.0653	0.6770	0.0455	0.2126	0.1620	0.1499
4	3	4	4	1	0.9244	0.0160	0.1221	0.0879	0.0765	0.6189	0.0499	0.2211	0.1688	0.1593
5	3	3	5	1	0.9056	0.0194	0.1346	0.0980	0.0866	0.6106	0.0553	0.2332	0.1765	0.1642
6	3	2	6	1	0.8435	0.0309	0.1703	0.1205	0.1035	0.4838	0.0653	0.2533	0.1814	0.1662
7	3	1	7	1	0.7197	0.0522	0.2284	0.1521	0.1263	0.2278	0.0895	0.2991	0.2136	0.1878

The curse of dimensionality





Barren Plateaus (too many parameters)

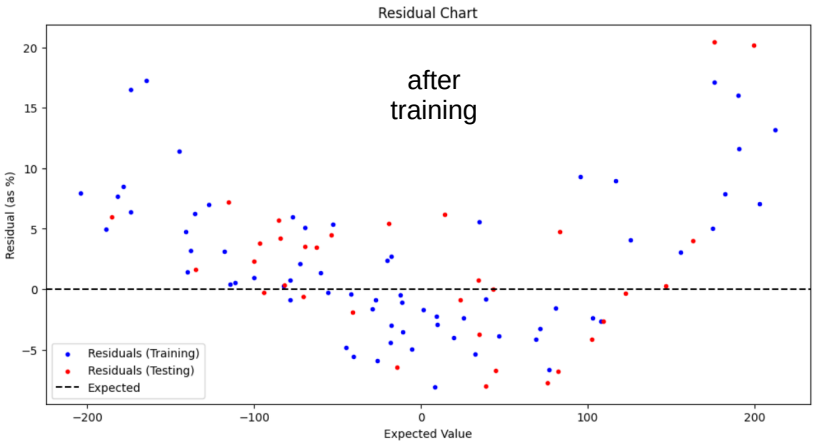
- Pairwise distances between uniformly distributed points in high-dimensional space become (almost) identical, and the surface of such a space is almost flat.
- In a quantum model with a high-D parameter space, the cost landscape is also flat, the situation called *barren plateau (BP)*.
- When BPs emerge, the optimiser struggles finding optimum.
- Selecting the optimisation initial point far from the optimum (e.g. random) makes it even more difficult!

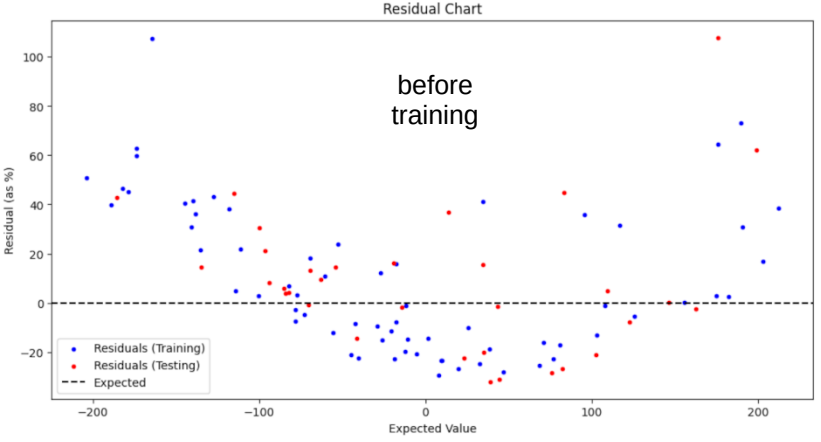
There exist well-known causes of BPs and there are well-known BP countermeasures

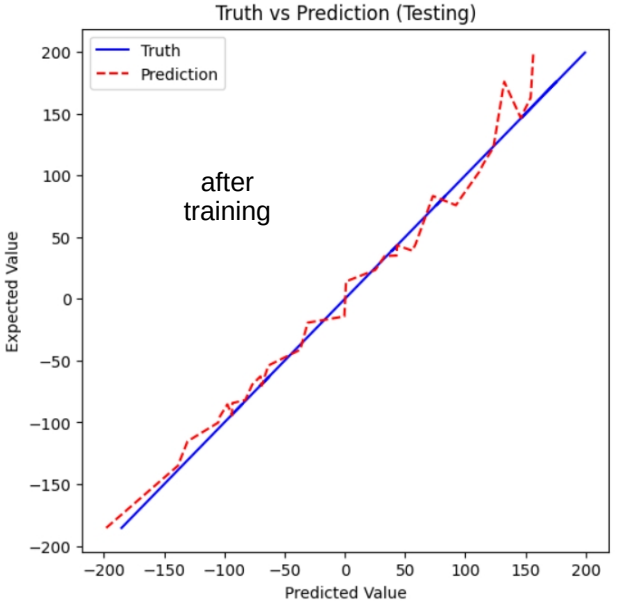
- use fewer qubits / layers / parameters
- use local cost functions (do not measure all qubits)
- beware of random params initialisation (and keep them small)
- use BP-resistant model design (e.g. layer-by-layer dev)
- use BP-resistant models (e.g. QCNNs)

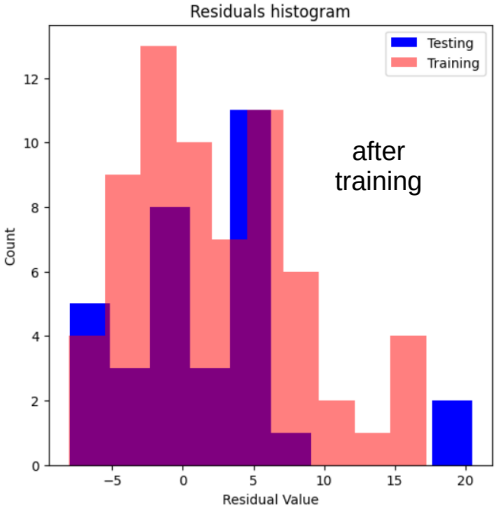
Cybulski, J.L., Nguyen, T., 2023. “Impact of barren plateaus countermeasures on the quantum neural network capacity to learn”, Quantum Inf Processing 22, 442.

Diagnostic charts









- The purpose of diagnostic charts is to better understand the accuracy (or precision) of the model.
- The residual scatter plot (left) gives an overview of the errors produced before and after training. However, the plot does not give an intuition of the severity of the problem.
- The severity of the errors (and their location) is better reflected in a histogram of residuals (upper-right), where the residual values are aggregated.
- The truth vs prediction chart (middle-up) allows to see how and where the prediction deviates from the expectation.

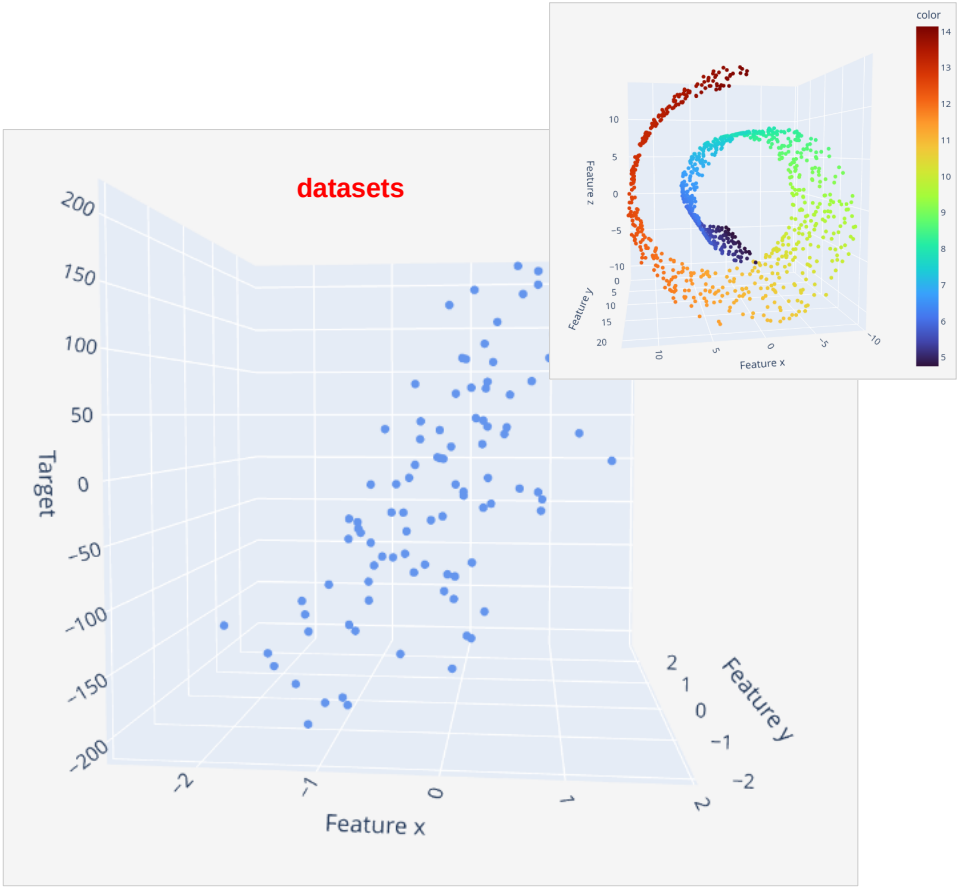
Barriers to quantum estimation

Problems	Possible Solutions
In VQA training • Quick forward step (quantum) • Slow backward step (classical)	• Use parameter-shift rules (quantum native gradients) • Use hardware-efficient differentiation • Use quantum kernels, where training is one-shot linear algebra • Use quantum annealing for training • Train using quantum Gibbs sampling (Bolzmann Machines) • Adopt time-evolution based gradients (with Trotterisation) • Use adaptive Trotterisation (to balance precision vs circuit depth)
Precision of continuous results = number of shots	• More shots, more precision (more slow) • Continuous-variable quantum computing (e.g. photonics) • Analogue quantum computing (e.g. neutral atoms, photonic systems, quantum annealing) • Superconducting resonators • Hadamard test / Iterative phase estimation
Quantum noise	• Train models with simulated noise • Apply error correction and error mitigation techniques

Isaac De Vlugt. *Differentiation on quantum hardware, PennyLane Tutorial*. Sep 12, 2023.

PennyLane Demo

Functions within functions...



PennyLane Demo:

- Explore the synthetic data
- Investigate data preprocessing tasks
- Learn how to encode data
- Check the clever cost function enclosure
- Look at the model as a function factory
- Learn how to calculate the model shape
- Produce the circuit and its plot
- Train the model
- Score the model and plot scores
- Perform analysis of residuals

Key takeaways:

- Quantum modelling is an engineering task
- There is more to success than a clever model
- Data encoding is (again) crucial to performance
- Experiment with the ansatz parameters
- A default optimiser may be the best afterall, however, design experiments to test it!
- More training does not give better results
- When the model converged it is time for its scoring
- Estimation needs residual analysis

Thank you!

Any questions?

Photos from Unsplash

Enquanted is being somewhere in-between Enchanted and Entangled