中国科学院高能物理研究所
Institute of High Energy Physics Chinese Academy of Sciences

Introduction to Quantum Machine learning

Qiyu Sha

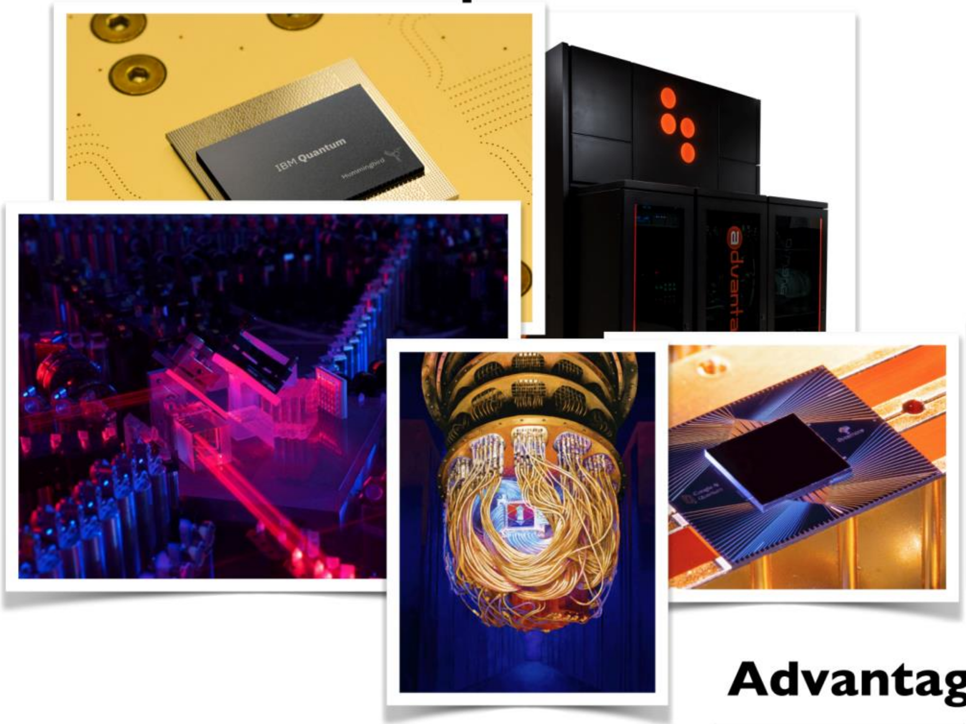
IHEP

Quantum Computing and Machine Learning Workshop 2024

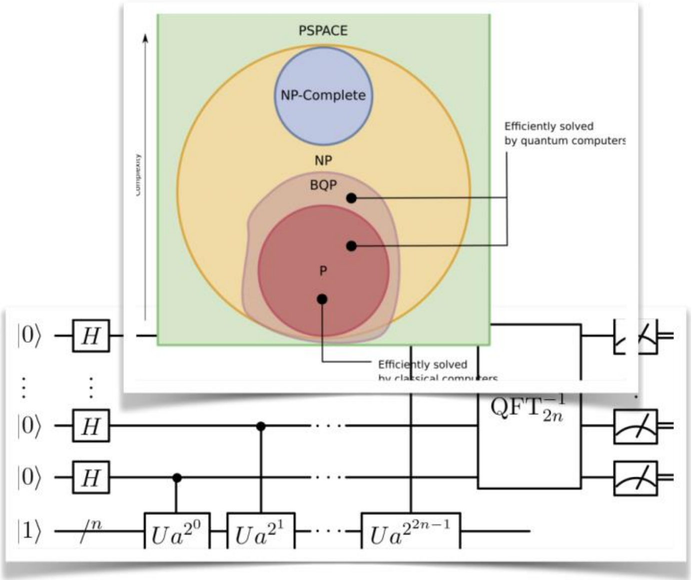
Overview

Qubit Type	Pros/Cons	Select Players
Superconducting	***Pros***: High gate speeds and fidelities. Can leverage standard lithographic processes. Among first qubit modalities so has a head start. ***Cons***: Requires cryogenic cooling; short coherence times; microwave interconnect frequencies still not well understood.	rigetti, Google, IBM Q, QuTech, OQC, IQM, qci Quantum Circuits, Inc, 本源量子 Origin Quantum
Trapped Ions	***Pros***: Extremely high gate fidelities and long coherence times. Extreme cryogenic cooling not required. Ions are perfect and consistent. ***Cons***: Slow gate times/ operations and low connectivity between qubits. Lasers hard to align and scale. Ultra-high vacuum required. Ion charges may restrict scalability.	IONQ, AQT, QUANTINUUM, oxford ionics, Universal Quantum
Photonics	***Pros***: Extremely fast gate speeds and promising fidelities. No cryogenics or vacuums required. Small overall footprint. Can leverage existing CMOS fabs. ***Cons***: Noise from photon loss; each program requires its own chip. Photons don't naturally interact so 2Q gate challenges.	PsiQuantum, XANADU, QuiX QUANTUM, ORCA Computing
Neutral Atoms	***Pros***: Long coherence times. Atoms are perfect and consistent. Strong connectivity, including more than 2Q. External cryogenics not required. ***Cons***: Requires ultra-high vacuums. Laser scaling challenging.	ColdQuanta, QuEra COMPUTING INC., atom computing, PASQAL
Silicon Spin/Quantum Dots	***Pros***: Leverages existing semiconductor technology. Strong gate fidelities and speeds. ***Cons***: Requires cryogenics. Only a few entangled gates to-date with low coherence times. Interference/cross-talk challenges.	intel, Silicon Quantum Computing, diraq, QUANTUM MOTION, QUANTUM BRILLIANCE

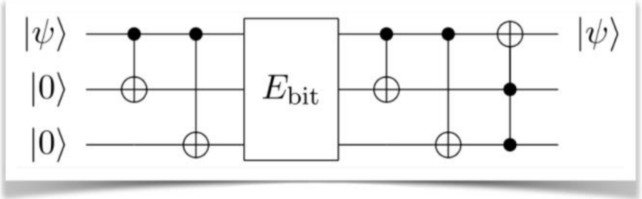
Computers



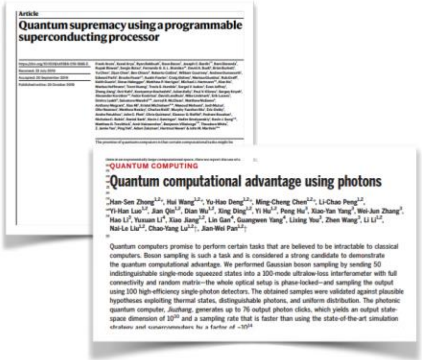
Algorithms



Error Correction

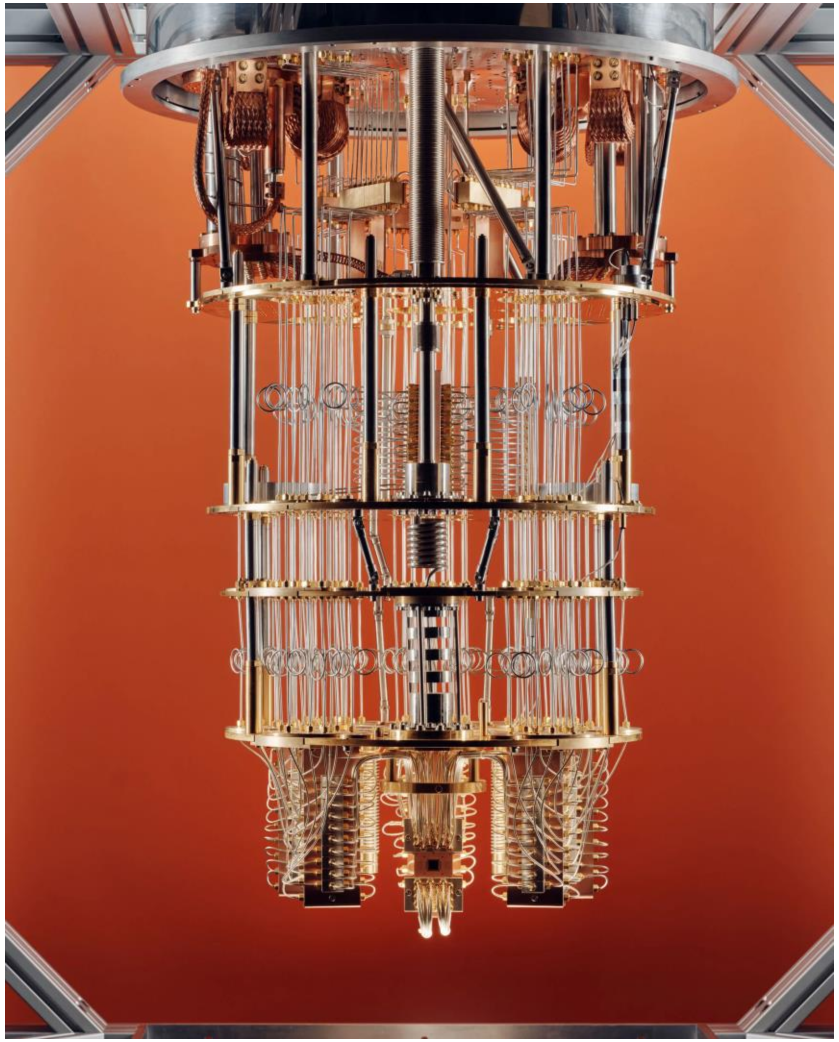


Advantage

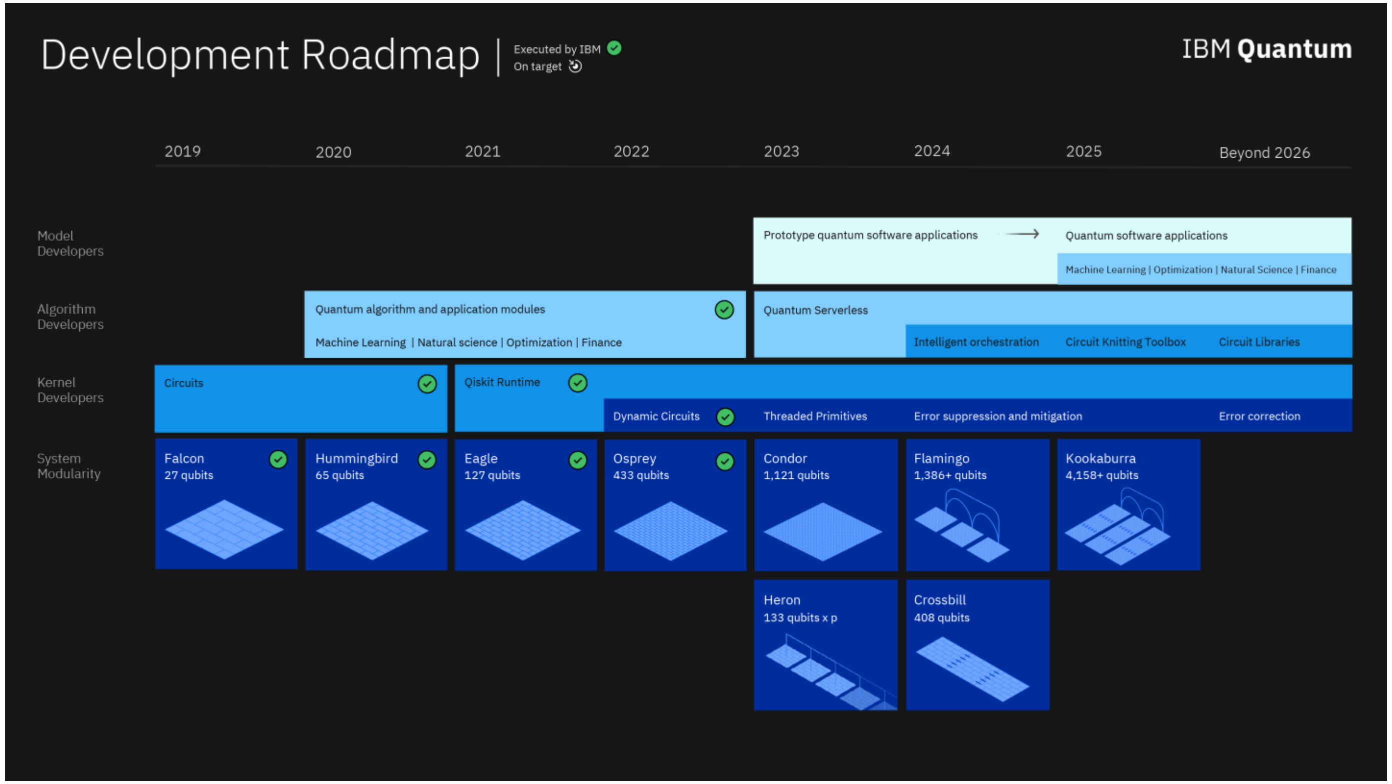


Programming

Introduction---IBM Quantum Computer



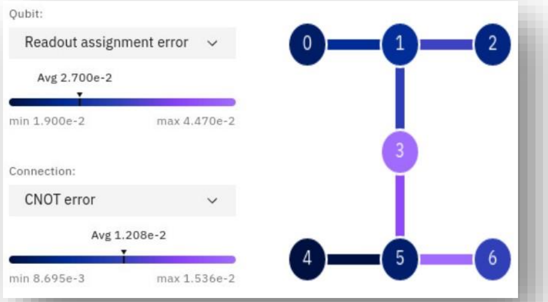
Credited to Thomas Prior for TIME



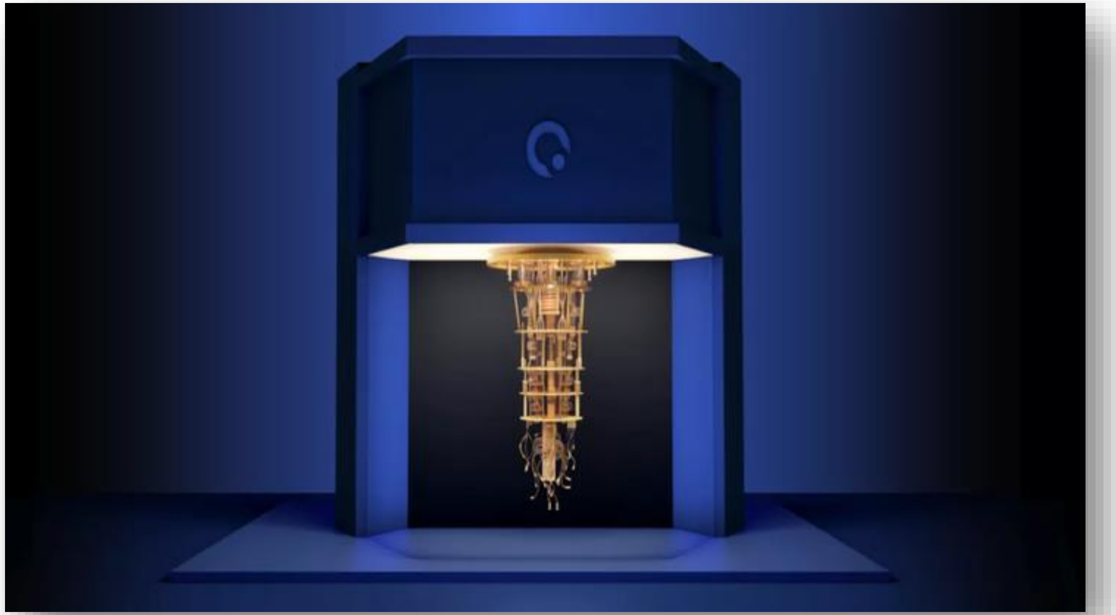
IBM has ambitious pursuits:

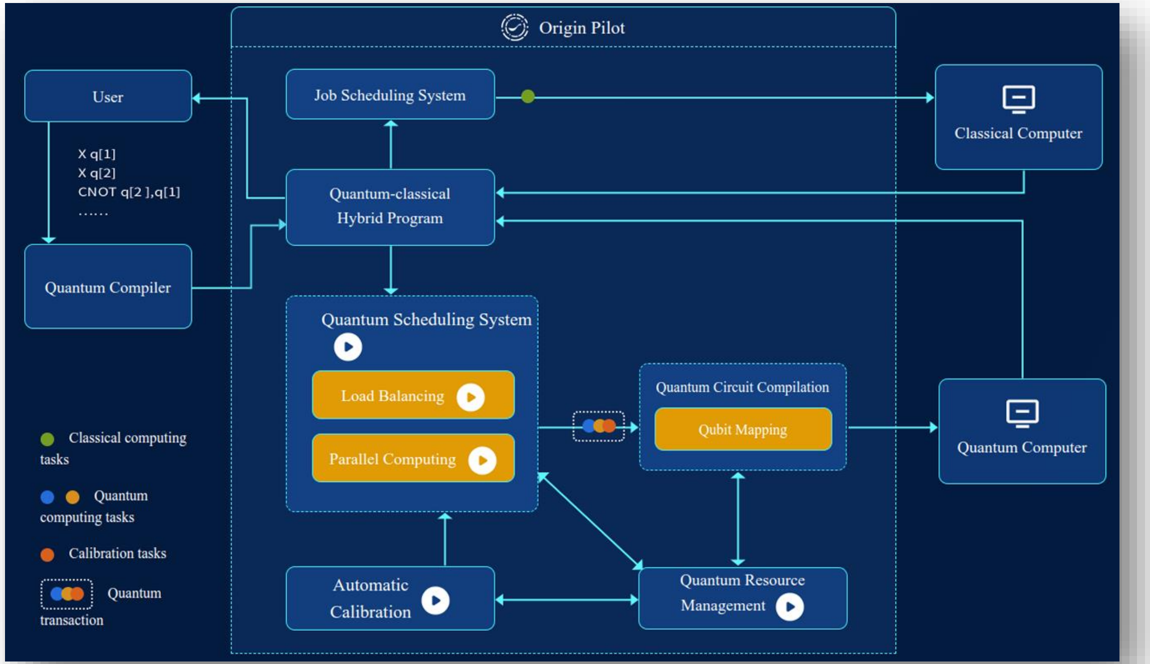
- 433-qubits IBM Quantum Osprey
- Three times larger than the Eagle processor (127-qubits)
- Going up to 10k-100k qubits.

Now, IBM provides up to 127 qubits for free.



Introduction---Origin Quantum Computer



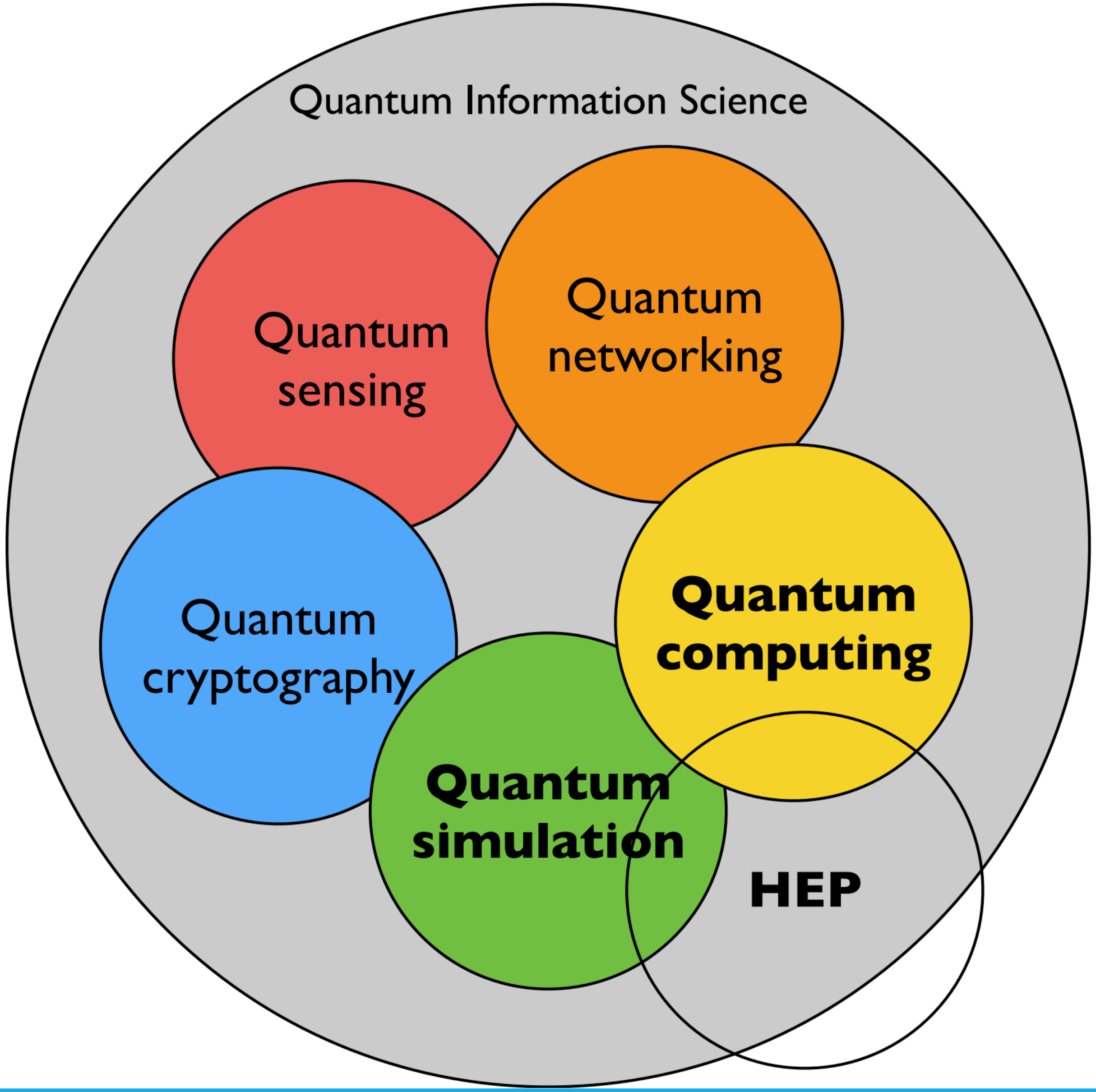


Origin wuyuan:

- The first “practical quantum computer” in China.
- 24-qubits with own control system.

Origin wuyuan provide up to 6 qubits for free.

Overview


Quantum Information Science
Quantum sensing
Quantum networking
Quantum cryptography
Quantum computing
Quantum simulation
HEP

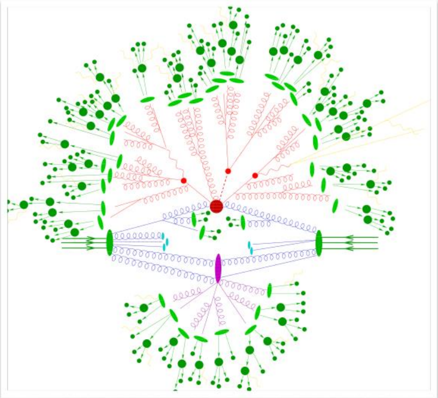
Outline of today

Applications of quantum computing in HEP

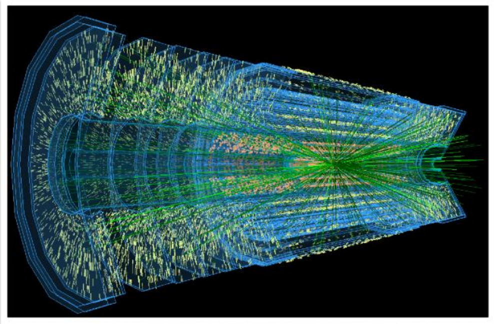
- Simulation
  - Parton shower correlations
  - Lattice QCD
- Reconstruction
  - Particle tracking
- Analysis
  - Higgs analyses
  - SUSY search
  - …

Progress has been very rapid here.

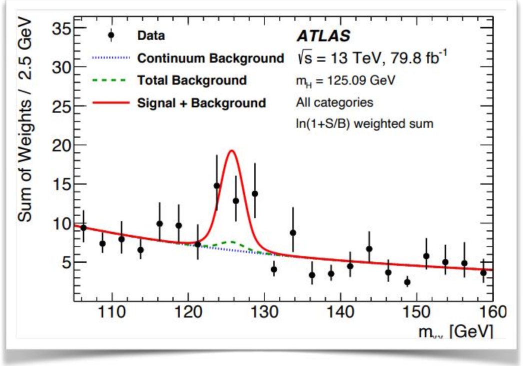
Simulation



Reconstruction

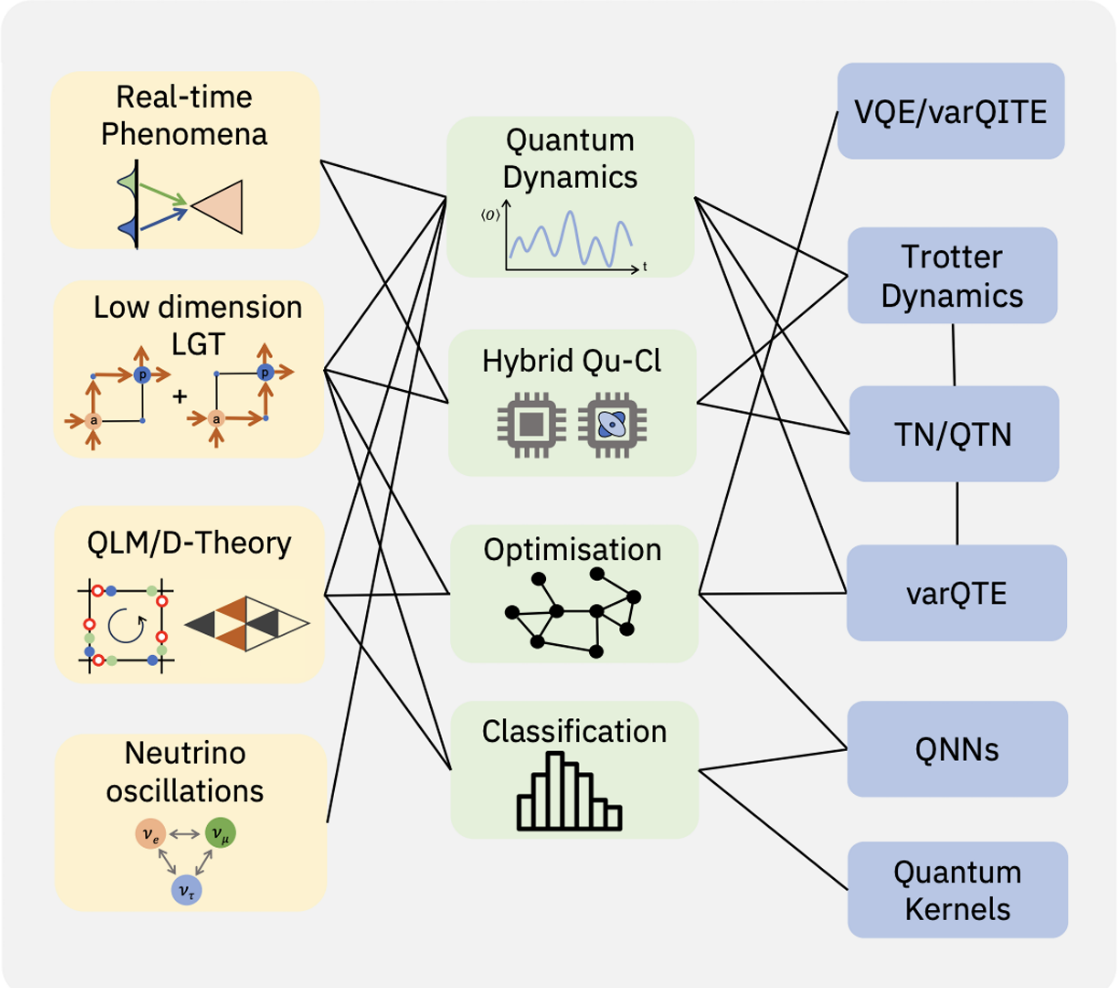


Analysis

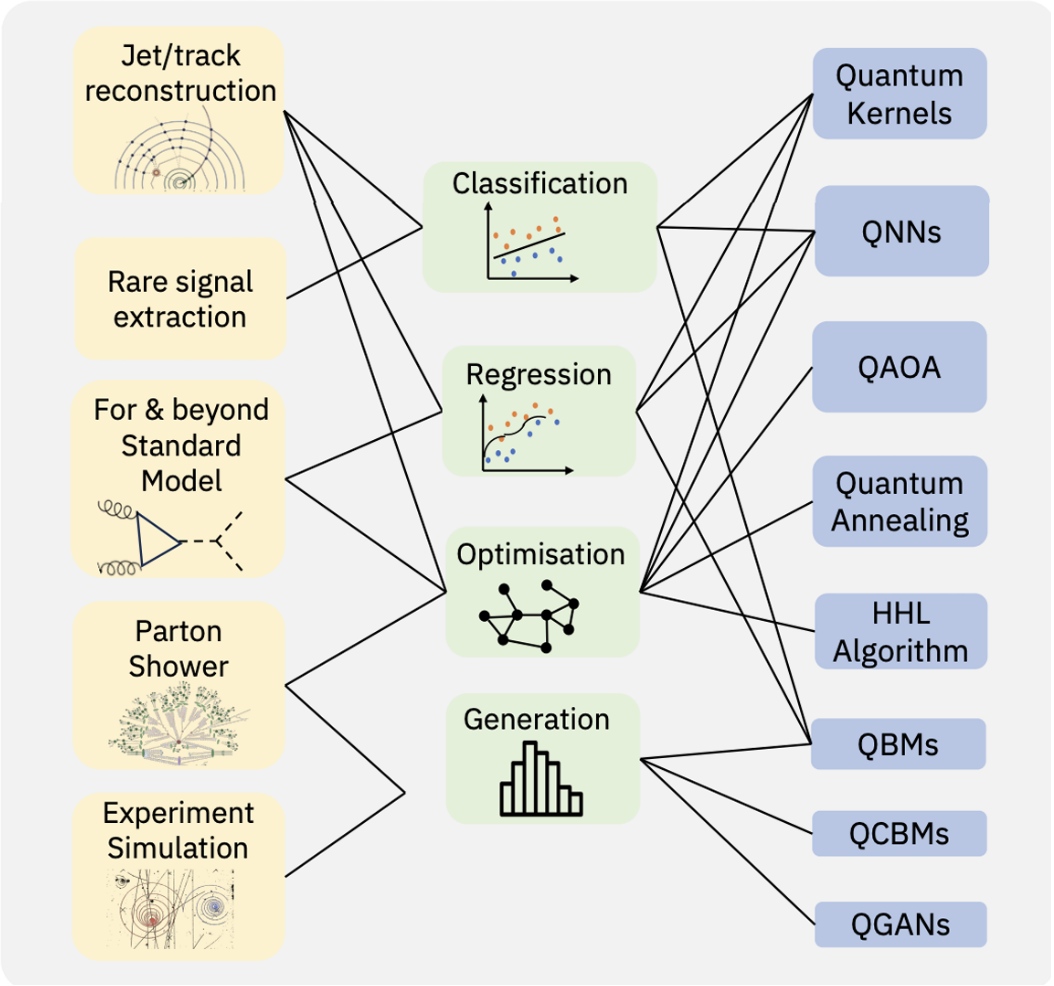


Outline of today

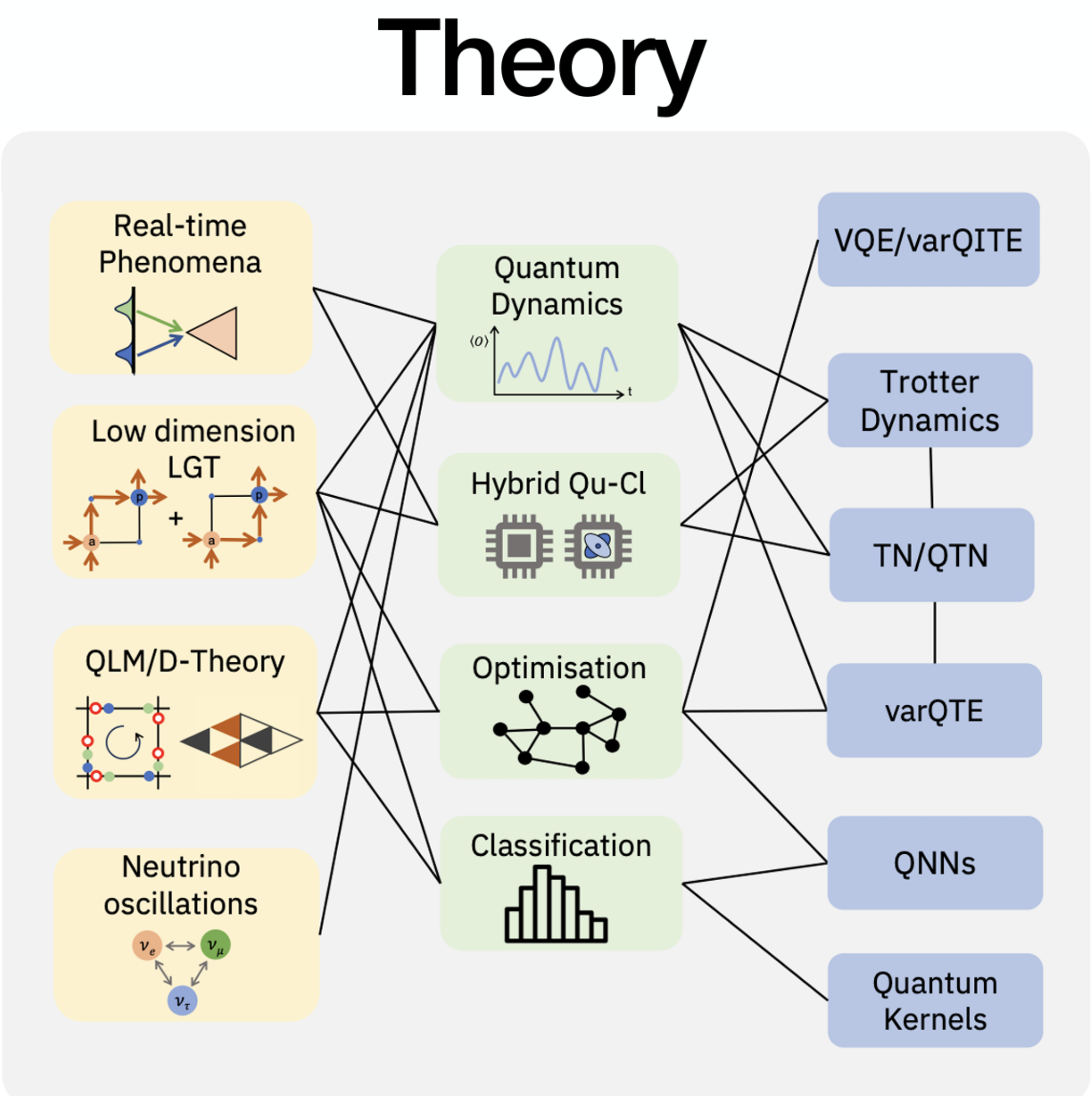
Theory

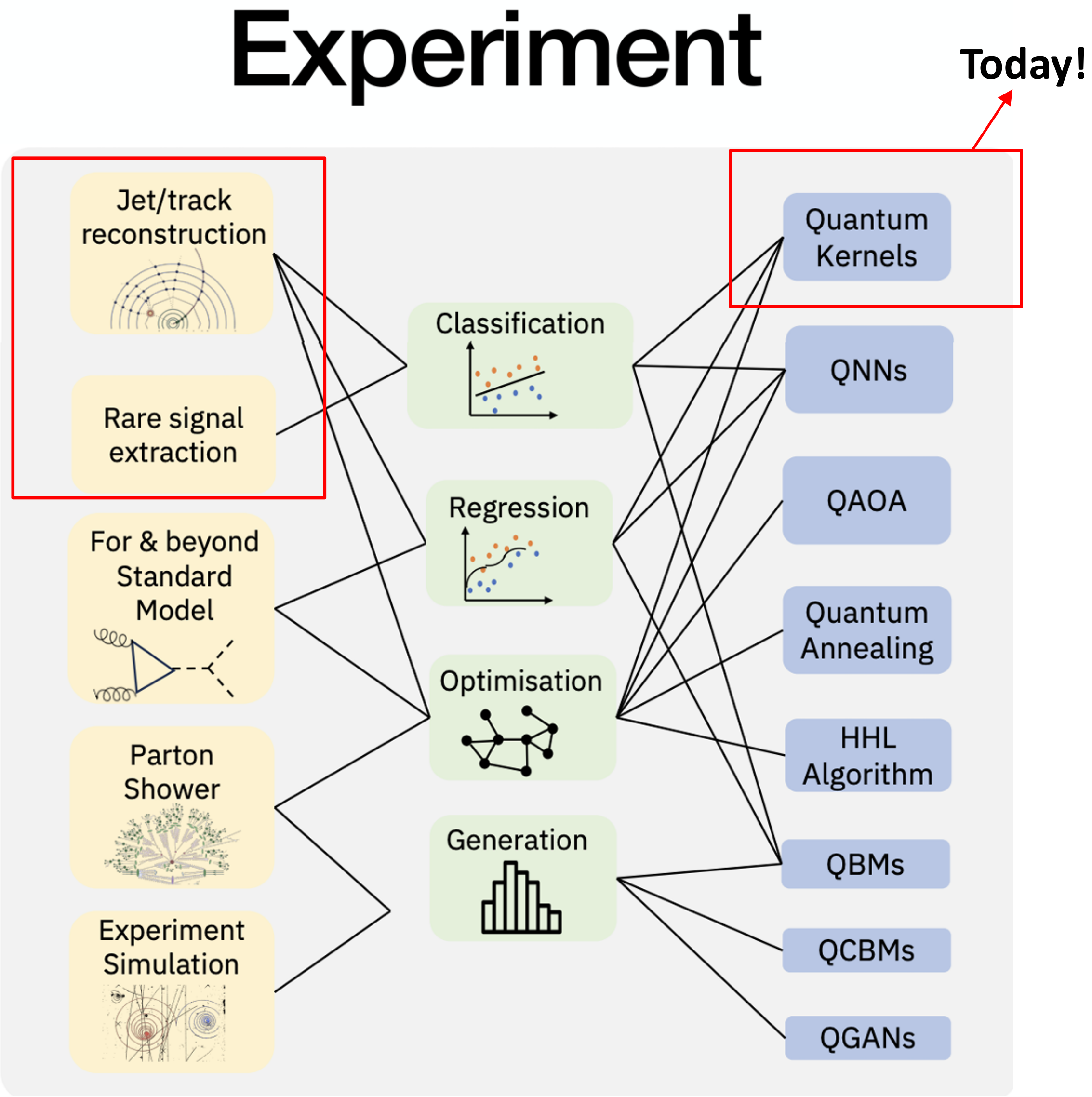


Experiment



Outline of today

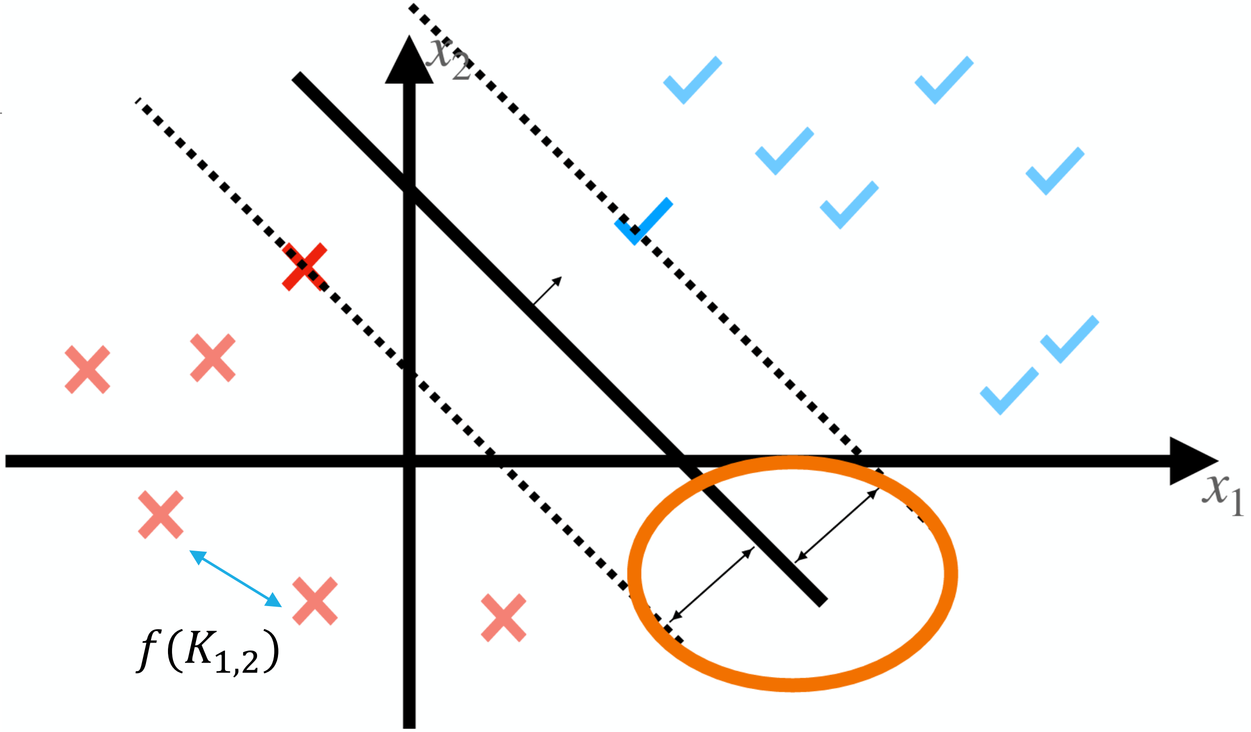




Introduction---SVM

SVM: Max margin

Usually done using the dual(think Lagrangian multipliers)



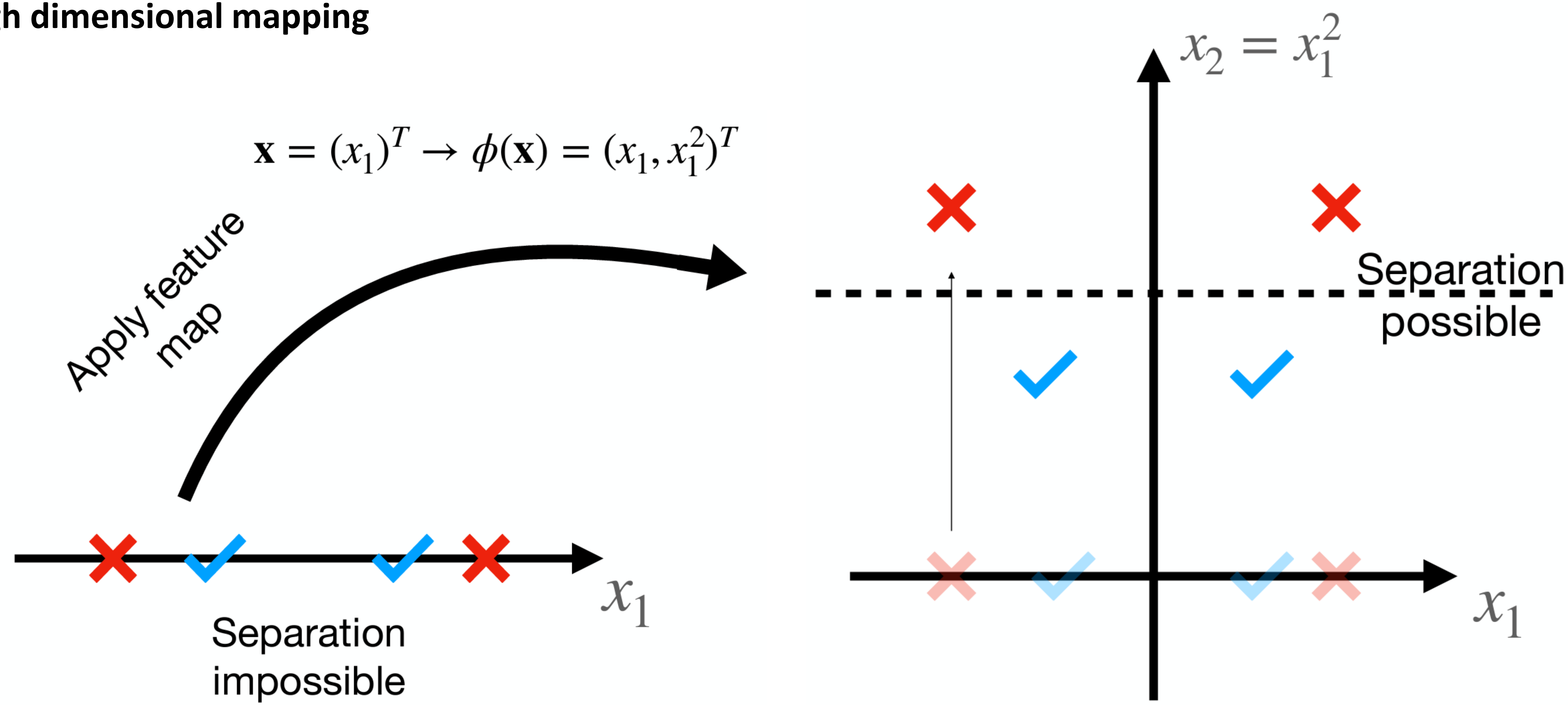
Maximise this

Results in building a kernel matrix.

$K_{1,1}$	$K_{1,2}$	$K_{1,3}$
$K_{2,1}$	$\ddots$	

Introduction---SVM feature map

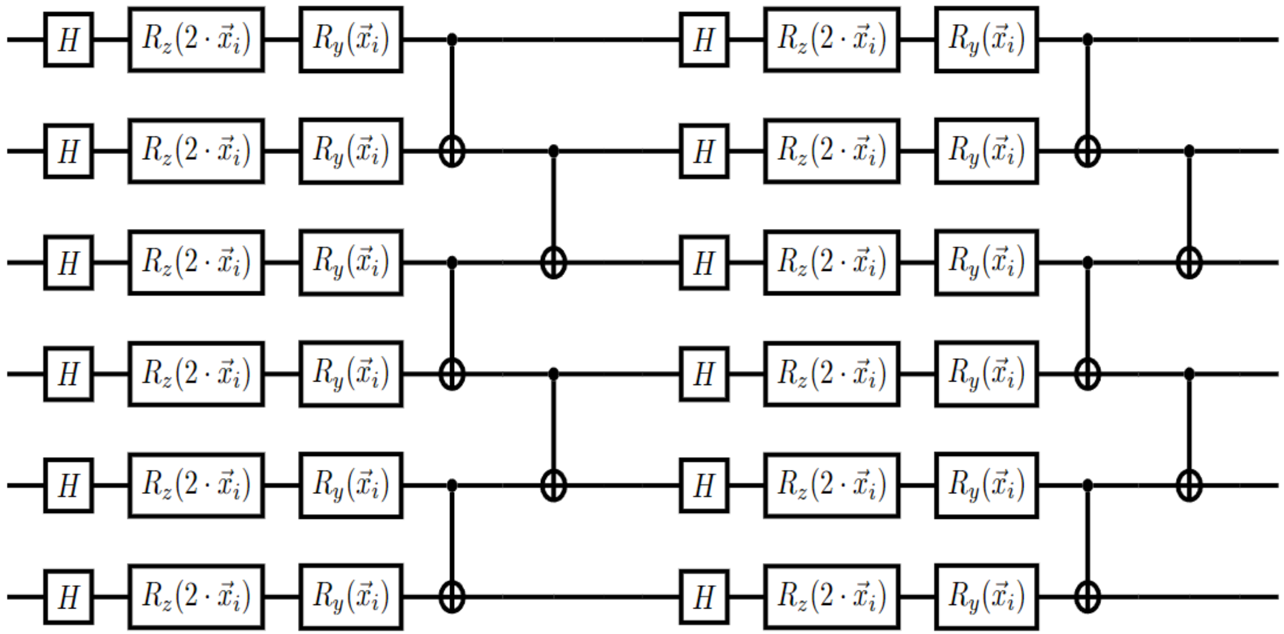
High dimensional mapping

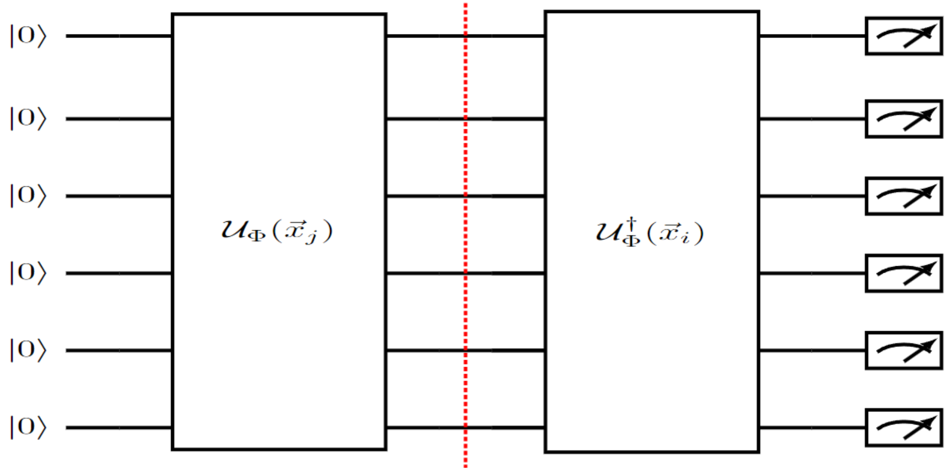


Introduction---SVM code

```python
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# 加载数据集
iris = datasets.load_iris()
X = iris.data
y = iris.target

# 数据集拆分
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# 数据标准化
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 训练SVM模型
model = SVC(kernel='linear', C=1.0)
model.fit(X_train, y_train)

# 预测
y_pred = model.predict(X_test)

# 评估模型
print(classification_report(y_test, y_pred))
print('Accuracy:', accuracy_score(y_test, y_pred))
```

What we can do?





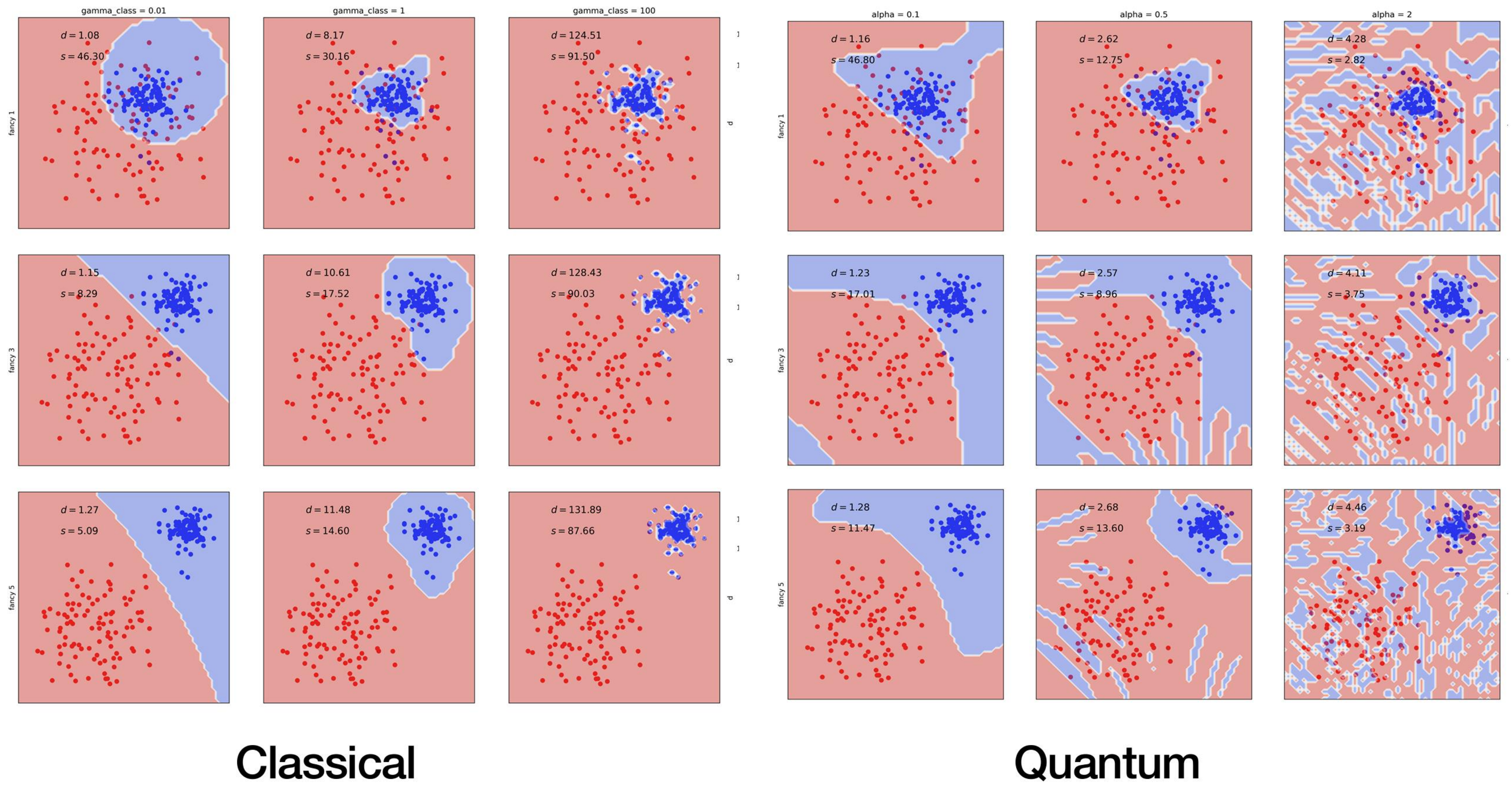
$K_{1,1}$	$K_{1,2}$	$K_{1,3}$
$K_{2,1}$	$\ddots$	

The only Quantum part in the QSVM!

$$k(\vec{x}_i,\vec{x}_j)=\left|\langle 0^{\otimes N}|\,\mathcal{U}^{\dagger}_{\Phi(\vec{x}_i)}\mathcal{U}_{\Phi(\vec{x}_j)}\,|0^{\otimes N}\rangle\right|^2$$

What we can do?

https://arxiv.org/abs/2212.07279



QSVM

- Code path: /cefs/higgs/shaqy/Quantum/QC_HEP/For_tutorial/QSVM
- Env: source /hpcfs/cepc/higgsgpu/shaqy/miniconda/etc/profile.d/conda.sh
  Conda activate Qiskit

SVC:

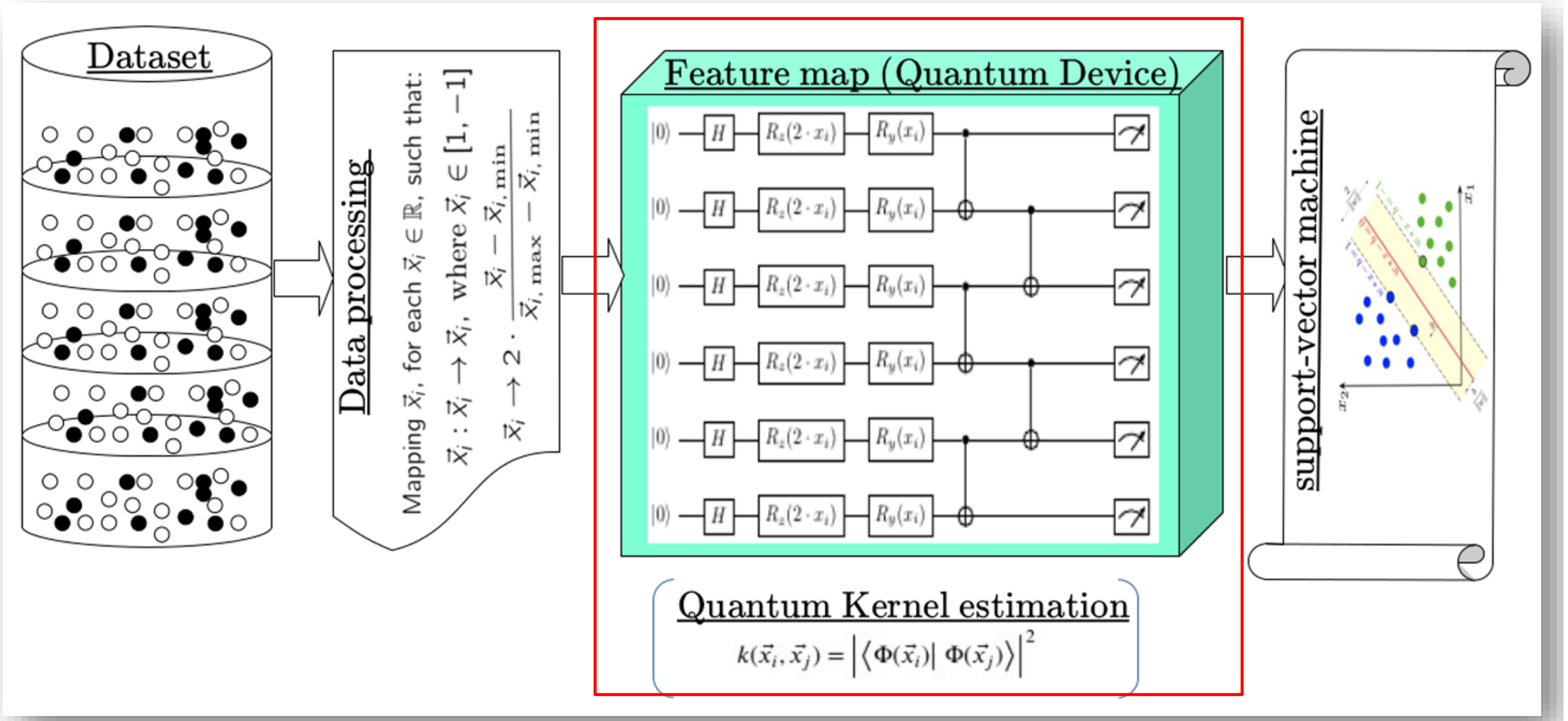
1. use your own defined kernels by passing a function to the kernel parameter.

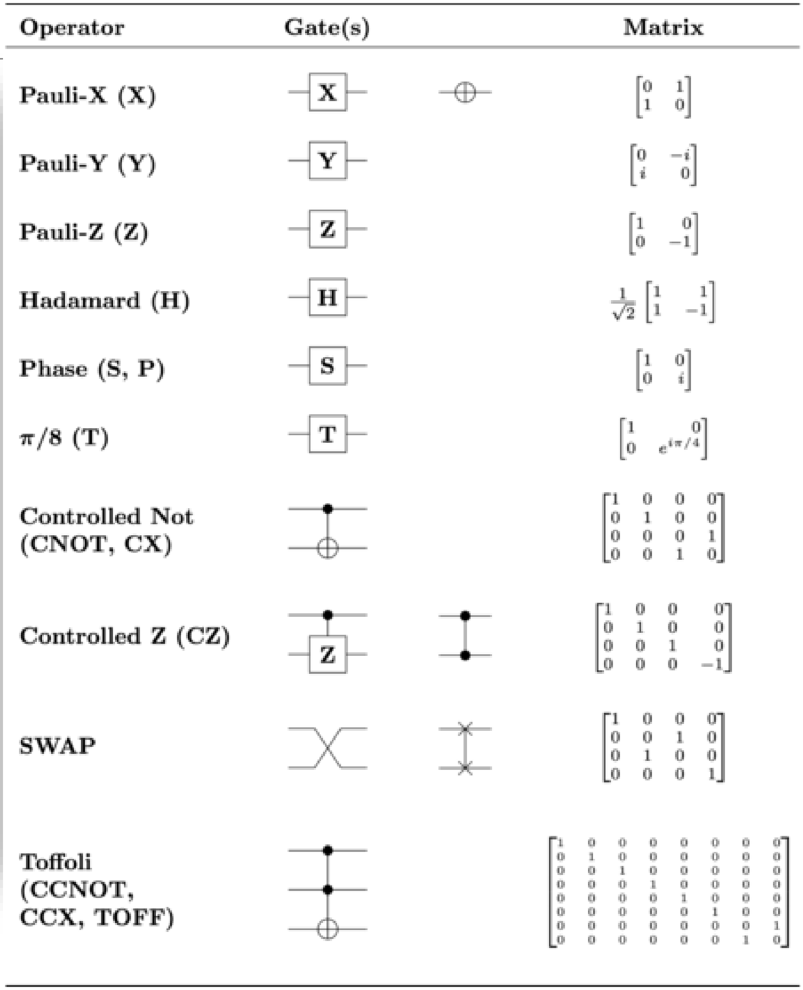
```
>>> import numpy as np
>>> from sklearn import svm
>>> def my_kernel(X, Y):
...     return np.dot(X, Y.T)
...
>>> clf = svm.SVC(kernel=my_kernel)
```

2. You can pass pre-computed kernels by using the kernel='precomputed' option. You should then pass Gram matrix instead of X to the fit and predict methods. The kernel values between all training vectors and the test vectors must be provided:

```
>>> import numpy as np
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import svm
>>> X, y = make_classification(n_samples=10, random_state=0)
>>> X_train , X_test , y_train, y_test = train_test_split(X, y, random_state=0)
>>> clf = svm.SVC(kernel='precomputed')
>>> # linear kernel computation
>>> gram_train = np.dot(X_train, X_train.T)
>>> clf.fit(gram_train, y_train)
SVC(kernel='precomputed')
>>> # predict on training examples
>>> gram_test = np.dot(X_test, X_train.T)
>>> clf.predict(gram_test)
array([0, 1, 0])
```

Data encoding and processing





Operator	Gate(s)	Matrix
Pauli-X (X)	X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)	Y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)	Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)	H	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)	S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)	T	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	Z	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \end{bmatrix}$

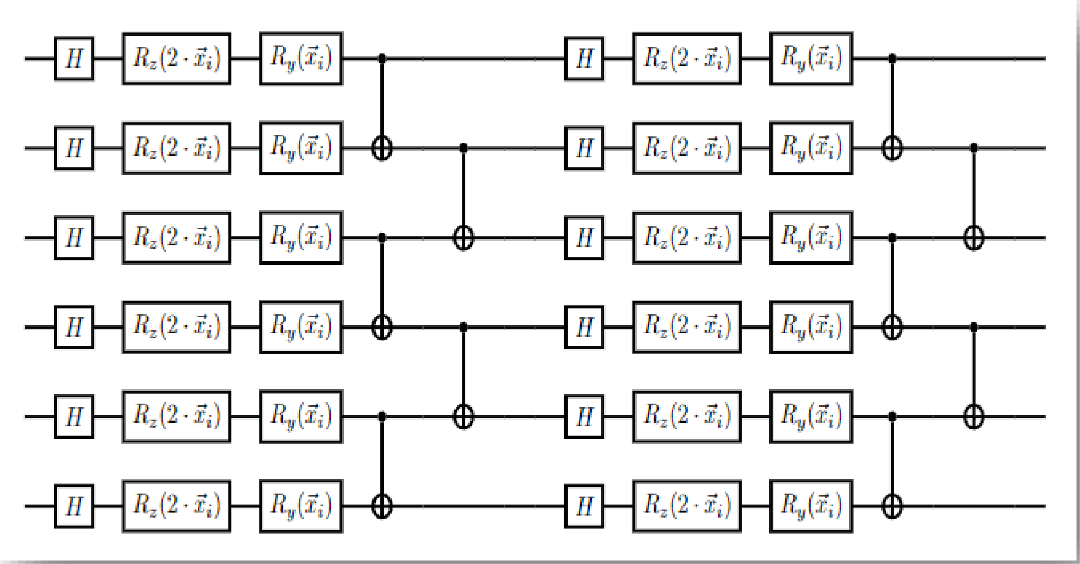
- Encoding the $e^+e^- \to ZH \to q\bar{q}\gamma\gamma$ (signal) and $e^+e^- \to (Z/\gamma^*)\gamma\gamma$ (background)
- Six variables are passed through preliminary mapping and then passed to a quantum circuit for evaluation.
- The Quantum support-vector machines kernel (QSVM-Kernel) is evaluated for each data point and the rest.

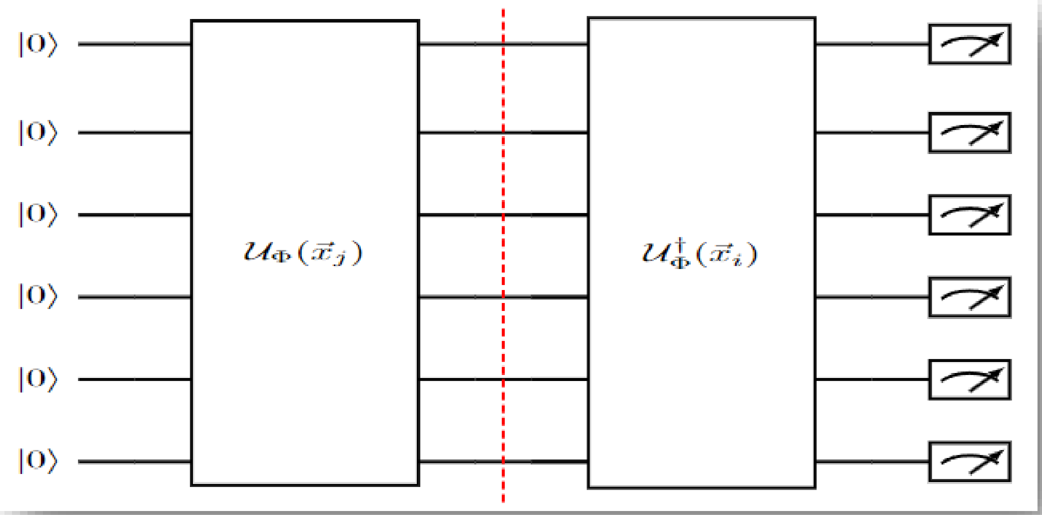
Feature map and quantum kernel estimation

Quantum feature map determines the QSVM-Kernel:

- Identical layers
- Single-qubit rotation gates
- Two-qubits CNOT entangling gates

Rotation	Depth	Events	Best AUC	Variation
$R_z(2\cdot\vec{x_i}) + R_y(\vec{x_i})$	2	5000	0.935	0.009
$R_z(\vec{x_i}) + R_y(\vec{x_i})$			0.933	0.015
$R_y(\vec{x_i}) + R_x(\vec{x_i})$			0.932	0.015
$R_z(\vec{x_i}) + R_z(\vec{x_i})$			0.932	0.014
$R_y(\vec{x_i})$			0.928	0.008
$R_z(\vec{x_i})$			0.928	0.008





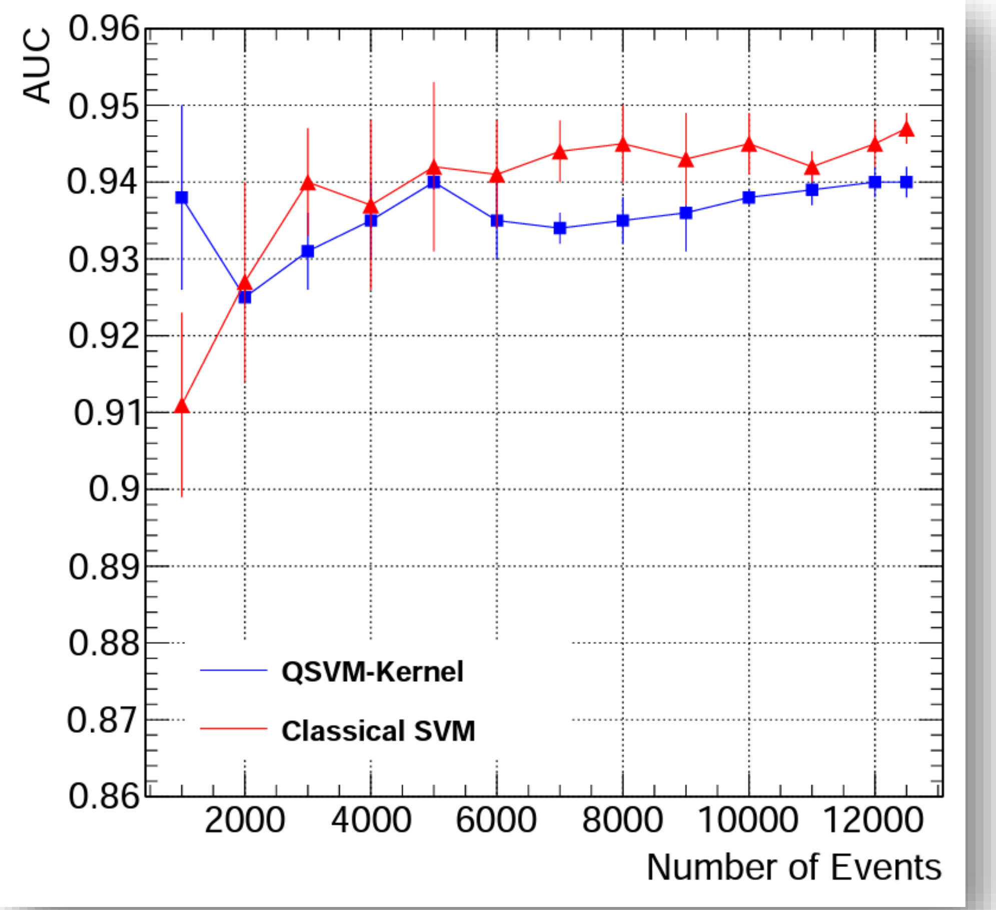
QSVM-Kernel estimation:

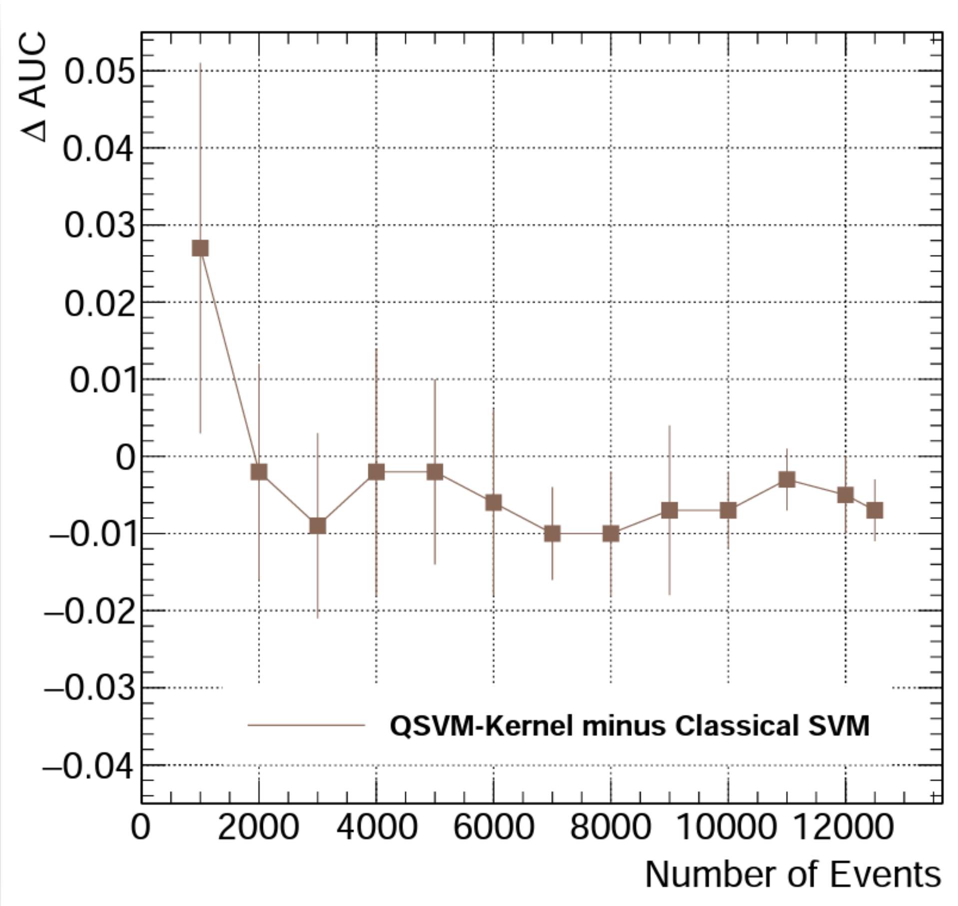
- Using 6 variables mapped to 6-qubit
- The expectation of each data point

$$k(\vec{x_i}, \vec{x_j}) = \left|\langle 0^{\otimes N}\right| \mathcal{U}^\dagger_{\Phi(\vec{x_i})}\mathcal{U}_{\Phi(\vec{x_j})}\left|0^{\otimes N}\rangle\right|^2$$

AUCs as function of the event

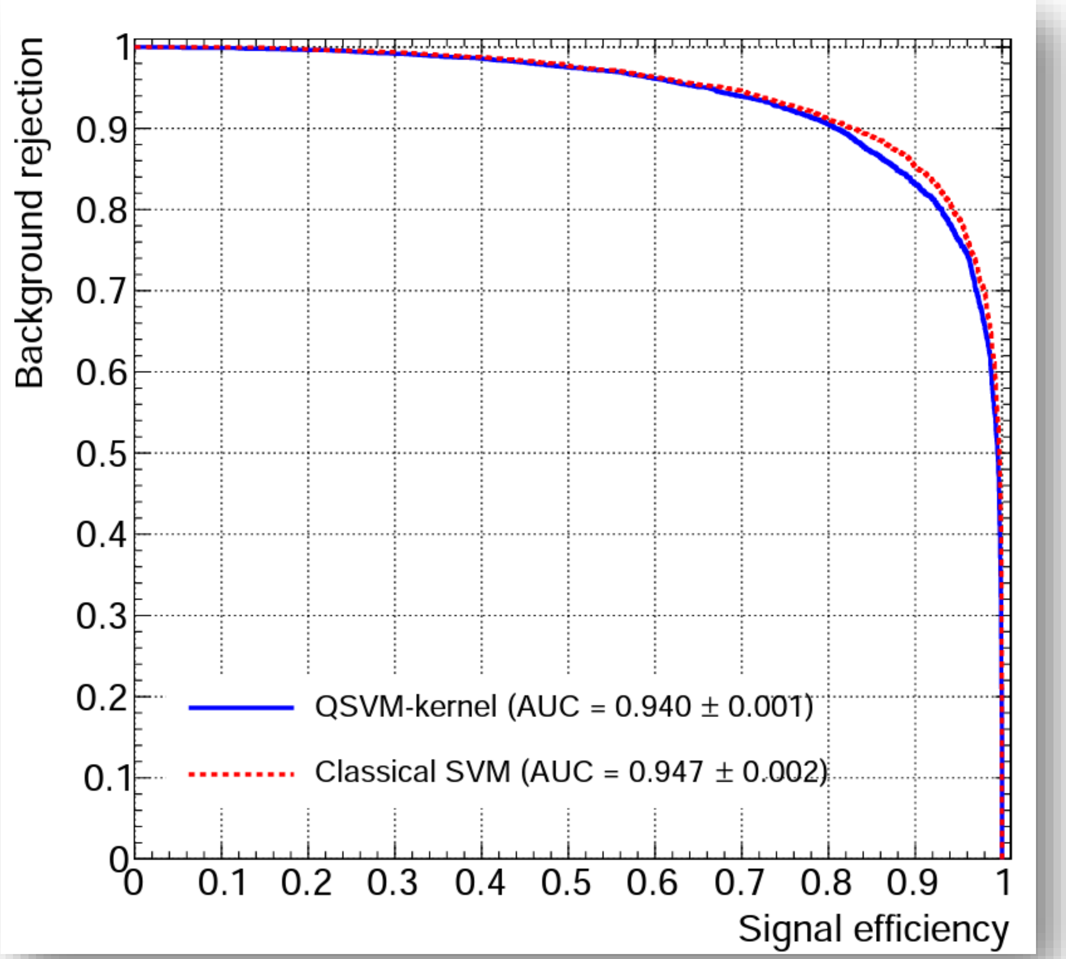
- The QSVM-Kernel and classical SVM classifiers with different dataset size from 1000 to 12500 events.
- The quoted errors are the standard deviations for AUCs calculated from several shuffles of the dataset.





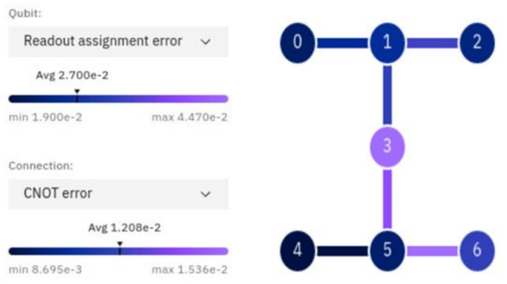
Performance of the quantum simulator

- The performance of the QSVM-Kernel using State-vector-simulator from IBM and the classical SVM.
- Use 12500 events for both signal and backgrounds.



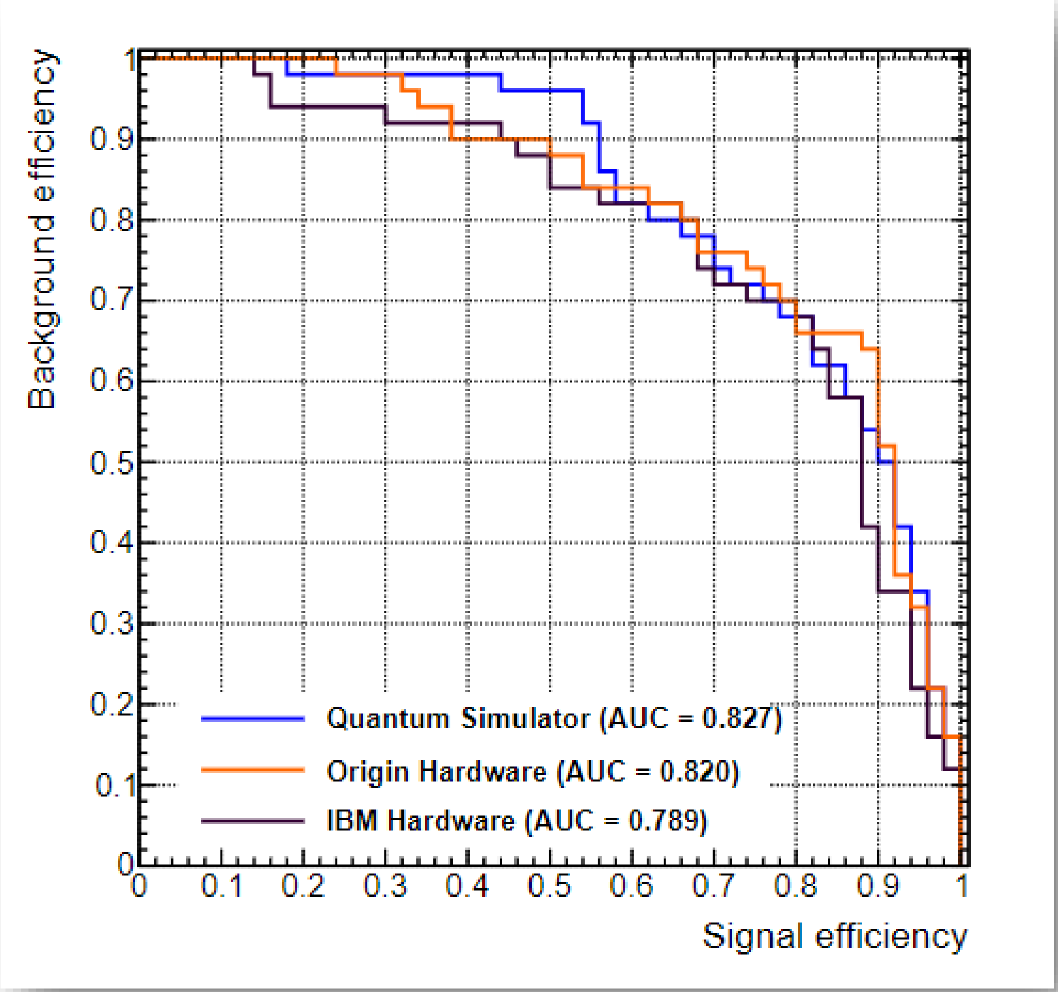
Performance of the real quantum computers

- IBM Nairobi & Origin Wuyuan quantum computer hardware
- Use 100 events for both signal and backgrounds.
- Use 6 qubits.



IBM Nairobi quantum

Origin Wuyuan quantum



How to do it

Set up:

- pip install pyqpanda

Requirements:

Linux

software	version
GCC	>= 5.4.0
Python	>= 3.7.0 && <= 3.9.0

- Official Tutorial: pyQPanda
- Code path: /cefs/higgs/shaqy/Quantum/QC_HEP/For_tutorial
- Env: source /hpcfs/cepc/higgsgpu/shaqy/miniconda/etc/profile.d/conda.sh
  conda activate QT_re (conda deactivate)

Everything is similar as IBM:

Different: Wuyuan don't have enough built-in functions like QSVM. We need to do by ourself.

- Only three steps: Use IBM Qiskit to generate QSVM kernel (gen_qasm.py shown in the next page.)

QASM(Quantum Assembly Language):

Convert QASM to program in wuyuan (page 21)
Get the kernel matrix.

The same way as IBM:
Use classical way to get results.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
creg c[6];
u2(0,pi) q[0];
rz(0.522150994) q[0];
u3(0.261075497,0.0,0.0) q[0];
u3(0.658964753,0.0,0.0) q[0];
rz(1.317929506) q[0];
u2(0,pi) q[0];
u2(0,pi) q[1];
rz(0.845885038) q[1];
u3(0.422942519,0.0,0.0) q[1];
u3(0.0224528909,0.0,0.0) q[1];
rz(0.0449057818) q[1];
u2(0,pi) q[1];
```

How to do it

Same as IBM tutorial:

- Create a feature map and kernel using Qiskit

```
rng = np.random.RandomState(0)
def gen_qasm():
  seed=2022
  backend = BasicAer.get_backend("statevector_simulator")
  feature_map_cus = customised_feature_maps.FeatureMap(num_qubits=6, depth=1, degree=1, entanglement='full', inverse=False)


  for i in range(1):
    seed = 2022
    algorithm_globals.random_seed = seed

    data = pd.read_csv("sample_%d.csv" % i)
    train = data[0:100]
    test = data[100:200]
    train_label = train.pop('tag')
    test_label = test.pop('tag')

    train_data = train.to_numpy()
    test_data = test.to_numpy()
    X_train = train_data
    X_test = test_data

    q_backend = QuantumInstance(backend, shots=10, seed_simulator=None, seed_transpiler=None)
    q_kernel = QuantumKernel(feature_map=feature_map_cus, quantum_instance=q_backend)

    qsvm_kernel_matrix_train = q_kernel.evaluate(x_vec=X_train)
    qsvm_kernel_matrix_test = q_kernel.evaluate(x_vec=X_test,y_vec=X_train)
    kernel_train_IBM = np.asmatrix(qsvm_kernel_matrix_train)
    kernel_test_IBM = np.asmatrix(qsvm_kernel_matrix_test)

    x = ParameterVector('x', length=6)
    y = ParameterVector('y', length=6)
    circuit = q_kernel.construct_circuit(x,y)
    circuit = q_backend.transpile(circuit)[0]

    for i in range(100):
      for j in range(100):
        f = open("all_qasm_train/qasm_%d_%d.txt"%(i,j),'w')
        cirtemp = circuit.assign_parameters({x:X_train[i], y:X_train[j]}, inplace=False)
        f.write(cirtemp.qasm())
        f.close()
```

Different part: (In red frame.)

- Save the dataset into a file(txt or .csv)
- For different (i,j) in train/test dataset:
  - Generate a QASM file.
  - One file represent a feature map for one point in kernel matrix.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
creg c[6];
u2(0,pi) q[0];
rz(0.522150994) q[0];
u3(0.261075497,0.0,0.0) q[0];
u3(0.658964753,0.0,0.0) q[0];
rz(1.317929506) q[0];
u2(0,pi) q[0];
u2(0,pi) q[1];
rz(0.845885038) q[1];
u3(0.422942519,0.0,0.0) q[1];
u3(0.0224528909,0.0,0.0) q[1];
rz(0.0449057818) q[1];
u2(0,pi) q[1];
```

How to do it

- Apply a key in the OriginQ website.(One key can run 1000 jobs in one day.)
- Apply X qubits and classical bits which used to save the result.
- Convert QASM to program.
- Use real_chip_type:origin_wuyuan_d5 to run. (Only d4 and d5 can use, d5 is better)
- The value with ['00000'] is the point of kernel matrix. Save it.

```python
from pyqpanda import *

def run(i,j):
    QCM = QCloud()
    QCM.init_qvm("58DCD70A14814A4DB73ACF5C8F854FA2"

    qlist = QCM.qAlloc_many(5)
    clist = QCM.cAlloc_many(5)

    qvm = init_quantum_machine(QMachineType.CPU)
    qvm.init_qvm()

    prog_trans, qv, cv = convert_qasm_to_qprog("all_qasm_train/qasm_%d_%d.txt"%(i,j), qvm)
    try:
        result = QCM.real_chip_measure(prog_trans, 10000, real_chip_type.origin_wuyuan_d5)
    except:
        print("job %d %d failed" % (i,j))
        return
    value = result['00000']
    f = open("results_train_try/result_%d_%d.txt" %(i,j),'w')
    f.write(str(value))
    f.close()

if __name__=="__main__":

  for i in range(19,20):
    for j in range(i+1,100):
      run(i,j)
```

How to do it

- Import the kernel matrix.

```
score = []
for i in range(1):

  data = pd.read_csv("sample_%d.csv" % i)
  train = data[0:100]
  test = data[100:200]
  train_label = train.pop('tag')
  test_label = test.pop('tag')
  train_label_oh = label_binarize(train_label, classes=[1,-1])
  test_label_oh = label_binarize(test_label, classes=[1,-1])

  test_kernel = []
  for i in range(100):
    test_kernel_line = []
    for j in range(100):
      value = get_kernel(i,j,'test')
      test_kernel_line.append(value)
    test_kernel.append(test_kernel_line)
  test_kernel_array = np.array(test_kernel, np.float32)

  train_kernel = []
  for i in range(100):
    train_kernel_line = []
    for j in range(100):
      if i == j:
        train_kernel_line.append(1.0)
      else:
        if i>j:
          # the matrix is symmetric
          # switch i and j as j is always greater than i in data
          value = get_kernel(j,i,'train')
          train_kernel_line.append(value)
        else:
          value = get_kernel(i,j,'train')
          train_kernel_line.append(value)
    train_kernel.append(train_kernel_line)
  train_kernel_array = np.array(train_kernel, np.float32)
```

- Then use the classical svc to get the final results, generate AUC value and draw plots.

```
#QSVM
    svc = SVC(C=30, probability=True, kernel="precomputed")
    #svc = QSVC(C=5, probability=True, quantum_kernel="precomputed")
    csvc = svc.fit(train_kernel_array, train_label)
    predictions = csvc.predict_proba(test_kernel)
    fpr, tpr, _ = sklearn.metrics.roc_curve(test_label_oh, predictions[:, 0])
```

Quantum transformer

- Quantum Transformer:
  - At the core of any Transformer sits the so-called ***Multi-Headed Attention.***
  - We apply three different linear transformations $W_Q, W_K,$ and $W_V,$ to each element of the input sequence to transform each element embedding into some other internal representation states called Query (Q), Key (K) and Value (V). These states are then passed to the function that calculates the attention weights, which is simply defined as:

$$Attention(Q, K, V) = softmax_k\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

  - To promote the Transformer from the classical to quantum real, one can simply replace the linear transformations $W_Q, W_K,$ and W_V with variational quantum circuits.

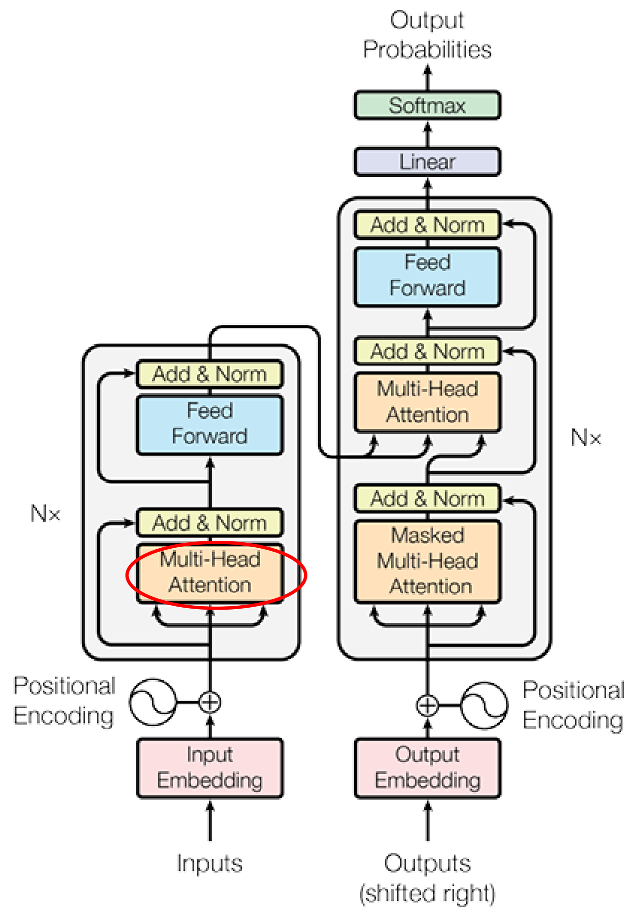


Figure 1: The Transformer - model architecture.

Code detail

Quantum transformer: (This code following here)

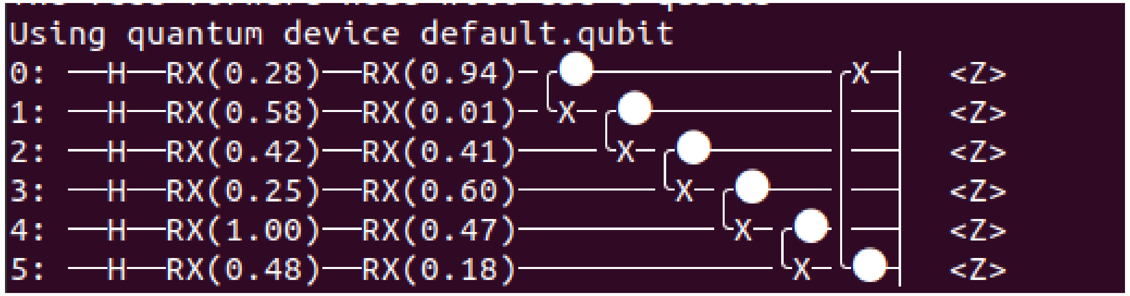
- The MultiHeadAttentionQuantum block
- Change the linear transformations.

```python
class MultiHeadAttentionClassical(MultiHeadAttentionBase):
    def __init__(self,
                 embed_dim: int,
                 num_heads: int,
                 dropout=0.1,
                 mask=None,
                 use_bias=False):
        super(MultiHeadAttentionClassical, self).__init__(embed_dim=embed_dim, num_heads=num_heads, dropout=dropout, mask=mask, use_bias=use_bias)

        self.k_linear = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.q_linear = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.v_linear = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.combine_heads = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.head_dim = embed_dim // num_heads

    def forward(self, x, mask=None):
        batch_size, seq_len, embed_dim = x.size()
        assert embed_dim == self.embed_dim, f"Input embedding ({embed_dim}) does not match layer embedding size ({self.embed_dim})"

        K = self.k_linear(x)
        Q = self.q_linear(x)
        V = self.v_linear(x)

        x = self.downstream(Q, K, V, batch_size, mask)
        output = self.combine_heads(x)
        return output
```

```python
self.n_qubits = n_qubits
self.n_qlayers = n_qlayers
self.q_device = q_device
self.head_dim = embed_dim // num_heads
if 'qulacs' in q_device:
    self.dev = qml.device(q_device, wires=self.n_qubits, gpu=True)
elif 'braket' in q_device:
    self.dev = qml.device(q_device, wires=self.n_qubits, parallel=True)
else:
    self.dev = qml.device(q_device, wires=self.n_qubits)
def _circuit(inputs, weights):
    for i in range(n_qubits):
        qml.Hadamard(wires=i)
    qml.AngleEmbedding(inputs, wires=range(n_qubits))
    qml.BasicEntanglerLayers(weights, wires=range(n_qubits))
    return [qml.expval(qml.PauliZ(wires=i)) for i in range(n_qubits)]

self.qlayer = qml.QNode(_circuit, self.dev, interface="torch")
self.weight_shapes = {"weights": (n_qlayers, n_qubits)}

#draw plots
weights = np.random.random([n_qlayers, n_qubits])
X = np.random.rand(6)
print(qml.draw(self.qlayer, expansion_strategy="device")(X,weights))

print(f"weight_shapes = (n_qlayers, n_qubits) = ({n_qlayers}, {self.n_qubits})")

self.k_linear = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
self.q_linear = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
self.v_linear = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
self.combine_heads = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
```

Quantum transformer model

- We make use of Xanadu's PennyLane quantum machine learning library to add quantum layers.
  - Add a function to the class and performs the quantum calculation (with a circuit) .
  - Wrap this circuit with a "QNode" to tell TensorFlow how to calculate the gradient with the parameter-shift rule.



  - Finally, we create a KerasLayer to handle the I/O within the hybrid neural network.($W_Q, W_K,$ and W_V)
  - Other part is same as the classical transformer.
  - https://github.com/shaqiyu/Quantum_transformer

Qiyu Sha qsha@cern.ch

How to work

- Download miniconda
- conda create -n env_name(Change the name as you want) root==6.24.00 python=3.8.6 -c conda-forge
- pip install -r requirment/*

Or:

source /hpcfs/cepc/higgsgpu/shaqy/miniconda/etc/profile.d/conda.sh

conda activate QT_re

- Python train_test.py (Change the option)

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-D', '--q_device', default='default.qubit', type=str)      #Fix
    parser.add_argument('-B', '--batch_size', default=32, type=int)                 #Fix for now.
    parser.add_argument('-E', '--n_epochs', default=10, type=int)                   #Fix for now, change after fix all hyperparameters
    parser.add_argument('-C', '--n_classes', default=2, type=int)                   #Fix
    parser.add_argument('-l', '--lr', default=0.001, type=float)                    #Changeable
    parser.add_argument('-v', '--vocab_size', default=6, type=int)
    parser.add_argument('-e', '--embed_dim', default=6, type=int)
    parser.add_argument('-f', '--ffn_dim', default=2048, type=int)                    #hidden layer dimension of feedforward networks. Changeable
    parser.add_argument('-t', '--n_transformer_blocks', default=2, type=int)       #Changeable
    parser.add_argument('-H', '--n_heads', default=2, type=int)
    parser.add_argument('-q', '--n_qubits_transformer', default=0, type=int) #6 for Quantum
    parser.add_argument('-Q', '--n_qubits_ffn', default=0, type=int)         #6 for Quantum
    parser.add_argument('-L', '--n_qlayers', default=1, type=int)            # For Quantum
    parser.add_argument('-d', '--dropout_rate', default=0.1, type=float)            #Changeable, but for few events, 0.1 is good
    args = parser.parse_args()


    Num_dataset = 20000
    if args.n_qubits_transformer > 0:
        Type_model = "Quantum"
    else:
        Type_model = "Classical"
```

Quantum transformer

- The dataset we used now is the CEPC MC sample
  - $e^+e^- \to ZH \to q\bar{q}\gamma\gamma$ (signal) and $e^+e^- \to (Z/\gamma^*)\gamma\gamma$ (background)
- Simulator: Pennlylane default device.
  - Time consuming: O(n), ~80 mins in CPU for 10k dataset with one epoch and one block(Q_layer).
- Current results (Use CPU): ~76% acc on validation dataset both in Quantum transformer and classical transformer in 20k dataset (10k train, 10k val).
  - Quantum:

```
Epoch 8/30
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_train.pdf has been created
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_val.pdf has been created
Epoch: 08 | Epoch Time: 140m 18s
        Train Loss: 0.510 | Train Acc: 76.24%
         Val. Loss: 0.500 |  Val. Acc: 76.52%
```

  - Classical:

```
Epoch 10/10
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_train_Classical.pdf has been created
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_val_Classical.pdf has been created
Epoch: 10 | Epoch Time: 0m 50s
        Train Loss: 0.485 | Train Acc: 76.39%
         Val. Loss: 0.467 |  Val. Acc: 77.66%
```