

# Embedding data into quantum circuits

황용수 @ ETRI

2023.07.17

# Quantum Information Tech. @ ETRI

The screenshot shows the ETRI website interface. On the left is a navigation menu under '연구·행정부서' (Research & Administration Department) with '인공지능컴퓨팅연구소' (AI Computing Research Institute) highlighted. The main content area is titled '부서 소개' (Department Introduction) for the '양자기술연구본부' (Quantum Technology Research Center). A horizontal navigation bar at the top of the main content area includes '초성능컴퓨팅연구본부', '지능형반도체연구본부', '사이버보안연구본부', and '양자기술연구본부' (the latter being highlighted with a blue box). Below this, there are tabs for '소개' (Introduction) and '수행업무' (Business Performance). The main text describes the center's focus on quantum sensing and communication technologies. A blue box highlights the first research team: '① 양자컴퓨팅연구실(방정호 실장 T. 042-860-6391)', which focuses on quantum computing and quantum information technology. A 'TOP' button is visible in the bottom right corner of the page content area.

# Quantum Information Tech. @ ETRI



**Dr. Jeongho Bang**

- Quantum information theory
- Quantum algorithm
- Quantum machine learning



**Dr. Kyunghyun Baek**

- Quantum information theory
- Quantum algorithm
- Quantum machine learning



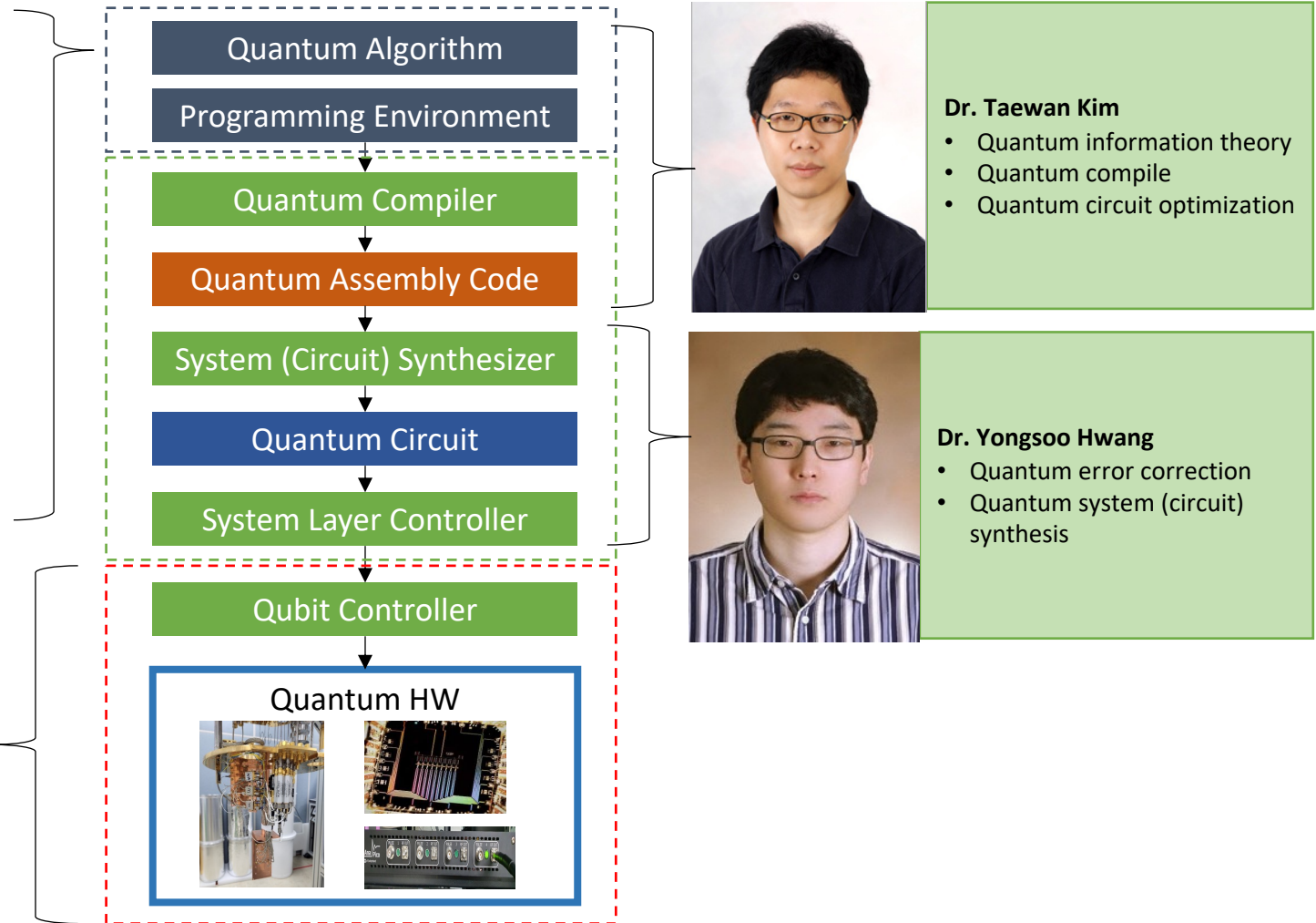
**Dr. Seungjin Lee**

- Quantum information theory
- Quantum algorithm
- Quantum machine learning



**Dr. Sungun Cho**

- Superconducting Qubit
- Cryostat
- Qubit Control



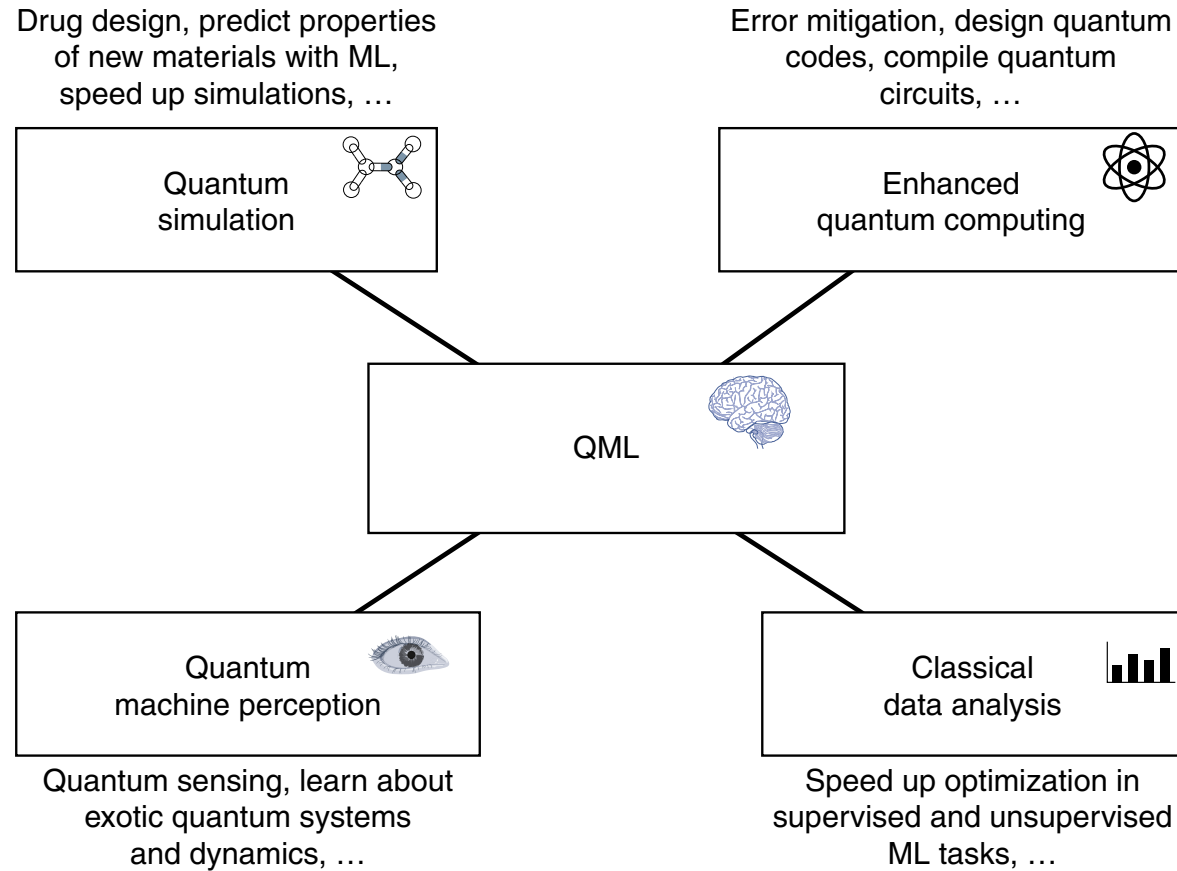
# Outline

- Quantum Data Embedding
- Ansatz, depth, layers
- How to embed data into a quantum state

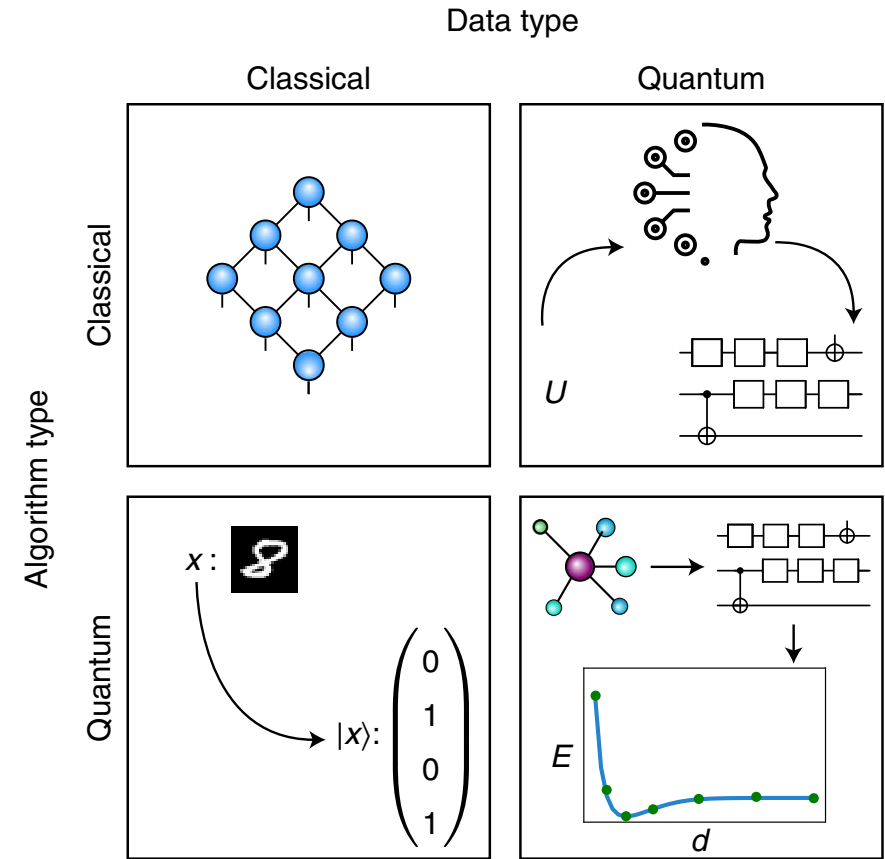
# Reference

- Jacob Biamonte et al., Quantum machine learning, *nature* 549, 195—202 (2017)
- M. Cerezo et al., Challenges and opportunities in quantum machine learning, *nature computational science* 2, 567—576 (2022)
- Maria Schuld and Nathan Killoran, Quantum Machine Learning in Feature Hilbert Spaces, *Phys. Rev. Lett.* 122, 040504 (2019)
- Seth Lloyd et al., Quantum embeddings for machine learning, *arXiv:2001.03622* (2020)
- Manuela Weigold et al., Encoding patterns for quantum algorithms, *IET Quantum Communication*, Vol. 2, Issue 4, pp.141—152 (2021)
- <https://pennylane.ai>

# Quantum Machine Learning

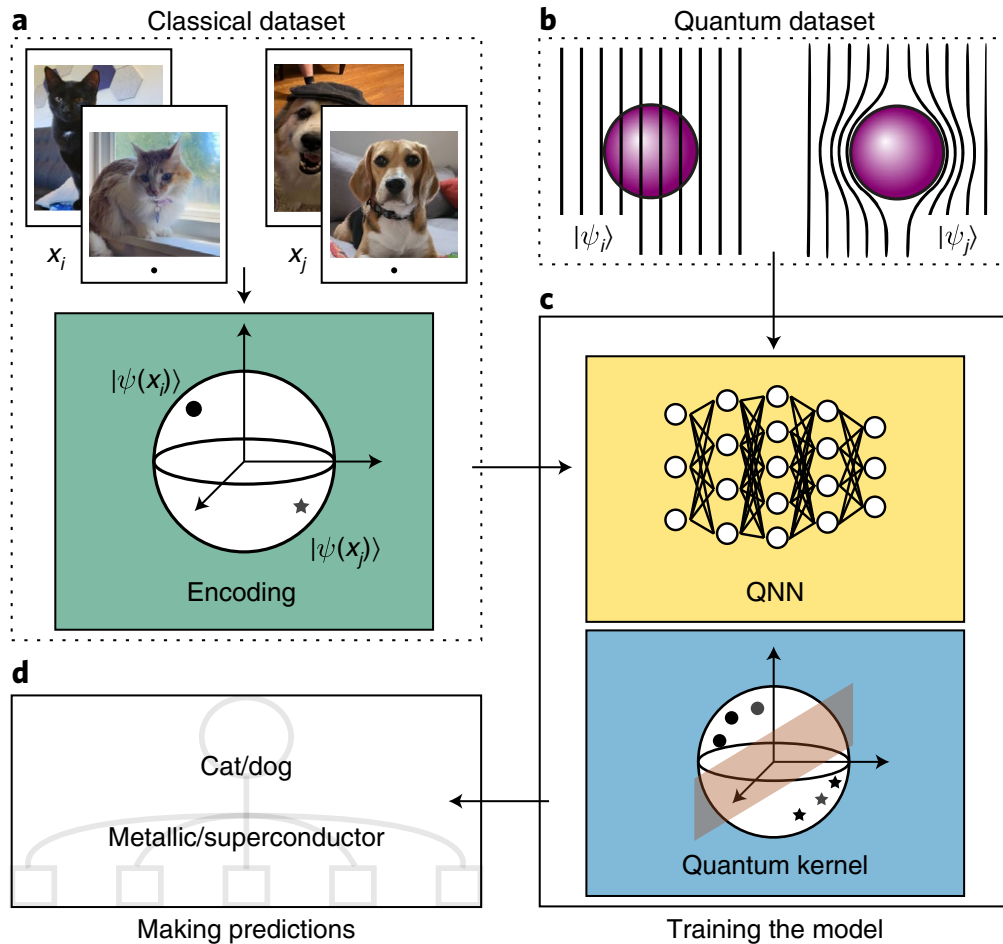


<Key applications for QML>



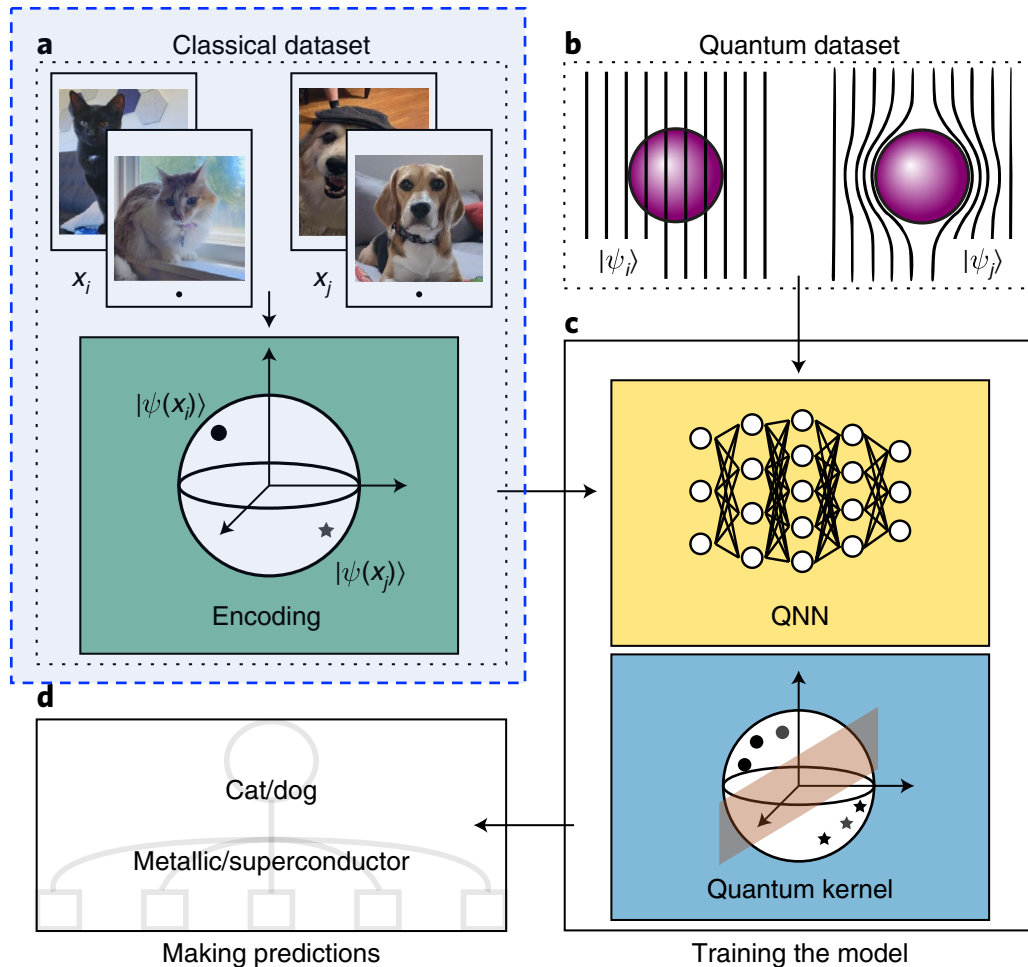
<QML Tasks : Types of algorithm and data>

# Quantum Machine Learning



**Fig. 3 | Classification with QML.** **a**, The classical data  $x$ , that is, images of cats and images of dogs, is encoded into a Hilbert space via some map  $x \rightarrow |\psi(x)\rangle$ . Ideally, data from different classes (here represented by dots and stars) are mapped to different regions of the Hilbert space. **b**, Quantum data  $|\psi\rangle$  can be directly analyzed on a quantum device. Here the dataset is composed of states representing metallic or superconducting systems. **c**, The dataset is used to train a QML model. Two common paradigms in QML are QNNs and quantum kernels, both of which allow for classification of either classical or quantum data. In kernel methods we fit a decision hyperplane that separates the classes. **d**, Once the model is trained, it can be used to make predictions.

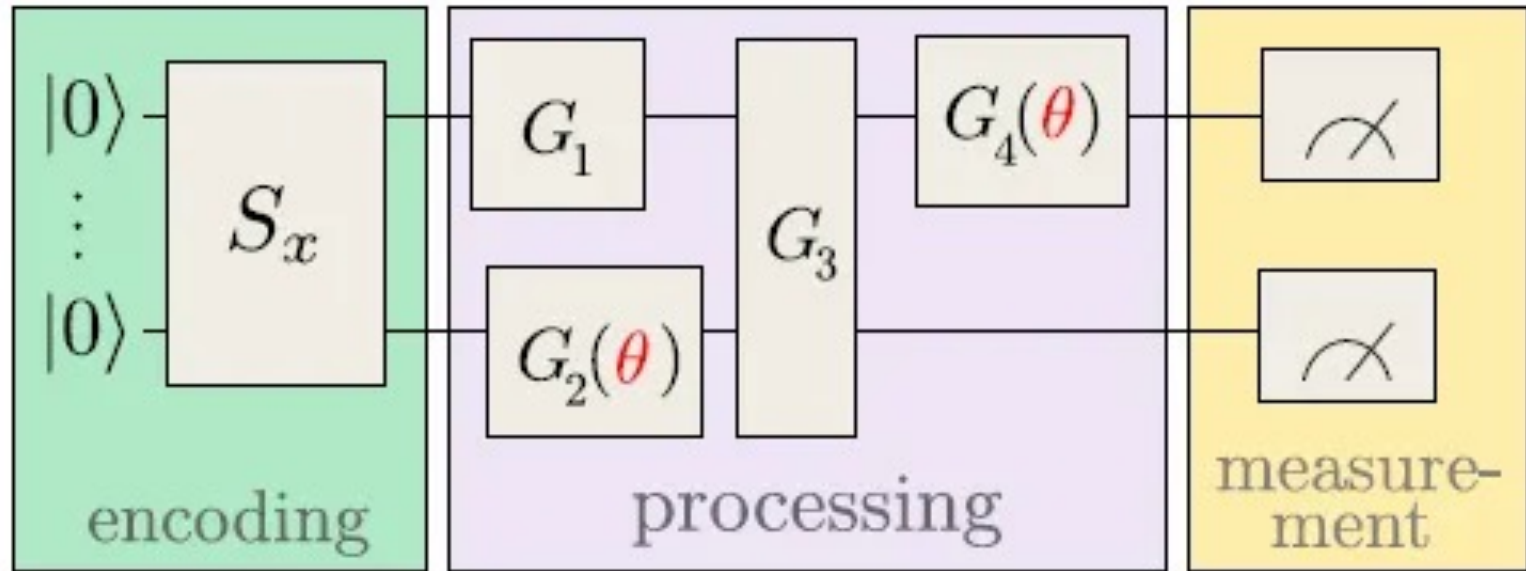
# Quantum Machine Learning



**Fig. 3 | Classification with QML.** **a**, The classical data  $x$ , that is, images of cats and images of dogs, is encoded into a Hilbert space via some map  $x \rightarrow |\psi(x)\rangle$ . Ideally, data from different classes (here represented by dots and stars) are mapped to different regions of the Hilbert space. **b**, Quantum data  $|\psi\rangle$  can be directly analyzed on a quantum device. Here the dataset is composed of states representing metallic or superconducting systems. **c**, The dataset is used to train a QML model. Two common paradigms in QML are QNNs and quantum kernels, both of which allow for classification of either classical or quantum data. In kernel methods we fit a decision hyperplane that separates the classes. **d**, Once the model is trained, it can be used to make predictions.



# 3 Steps of QML



# Quantum Data Encoding (or Embedding)

- Quantum Data Encoding (Embedding)
  - Process to load classical data onto a quantum system

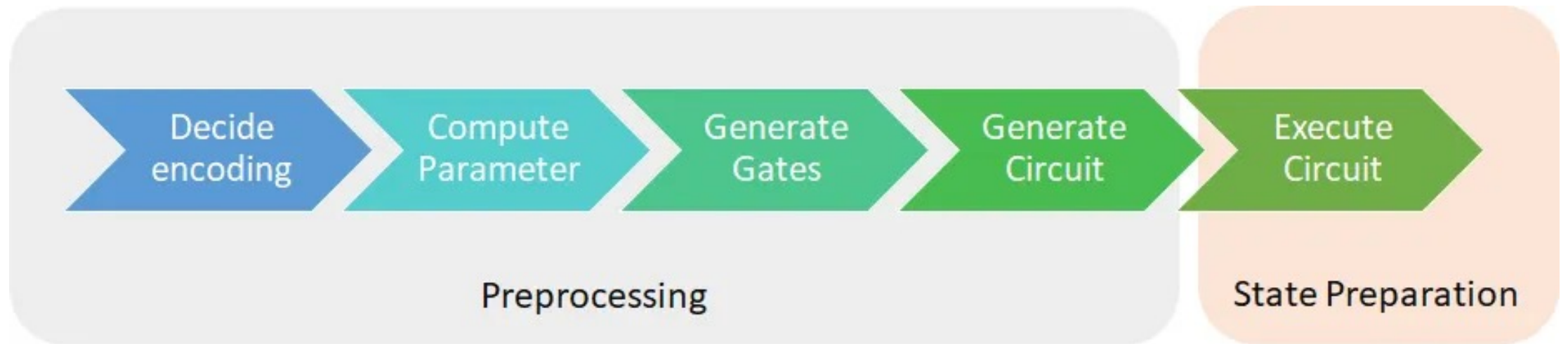
$$x \mapsto |\psi(x)\rangle$$

- Two categories:
  - **Digital** : data  $\rightarrow$  qubit strings,  $\frac{1}{\sqrt{M}} \sum_{m=1}^M |x(m)\rangle$
  - **Analogue** : data  $\rightarrow$  amplitude of a quantum state,  $\sum_{i=1}^N x_i |i\rangle$  with  $\sum |x_i|^2 = 1$

# Quantum Data Encoding (or Embedding)

- Preference
  - #Qubits should be minimal
  - #Parallel Op. should be minimal to minimize the width of the circuit
  - Data must be represented appropriately for further calculations
    - Digital for arithmetic computation
    - Analogue as mapping data into large Hilbert space for QML

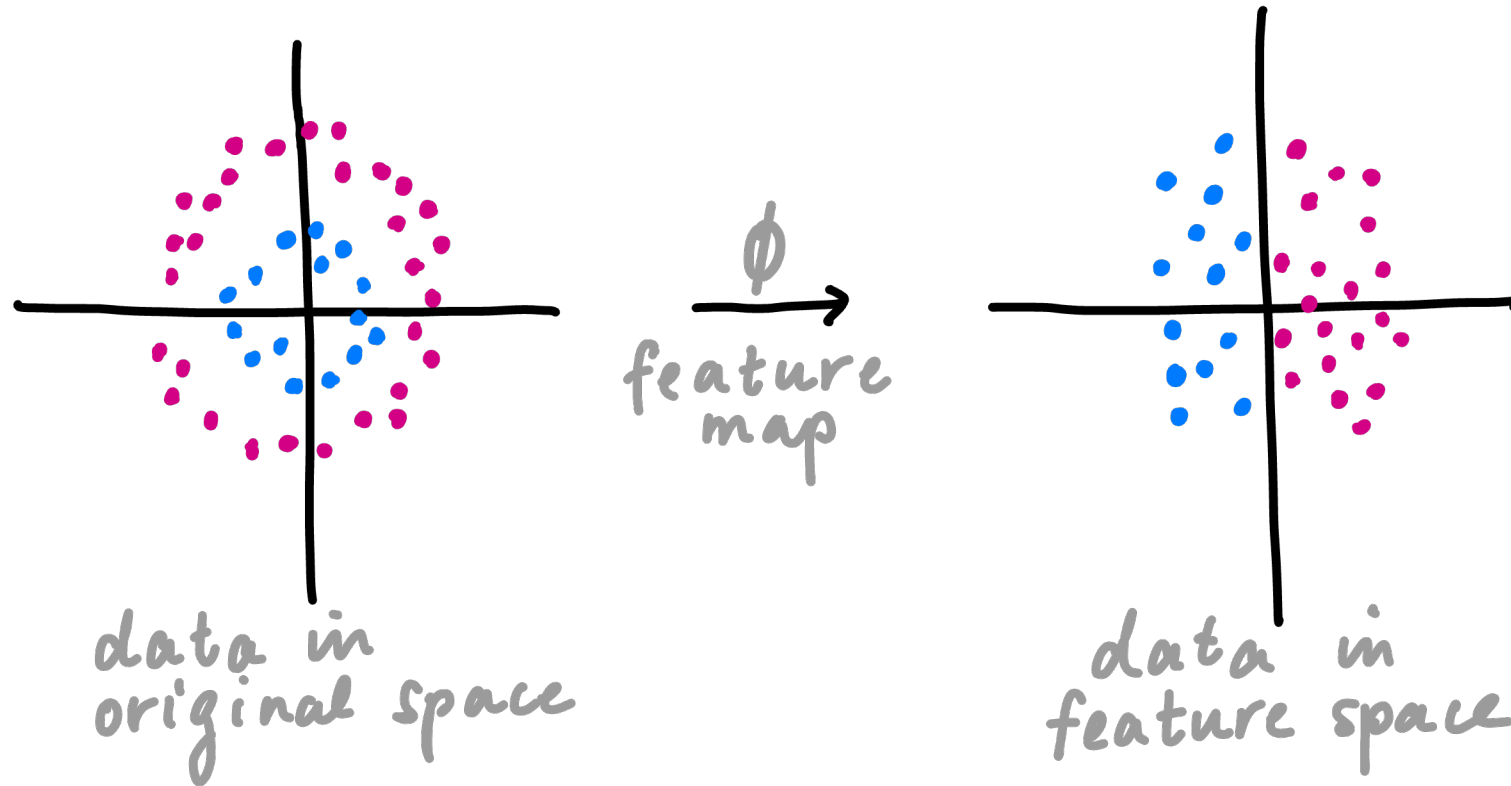
# Quantum Data Encoding (or Embedding)



# Feature Map

- Feature Mapping
  - Technique used in data analysis and machine learning to transform input data *from a lower-dimensional space* to *a higher-dimensional space*, where it can be more easily analyzed or classified  
(<https://www.geeksforgeeks.org/feature-mapping/>)

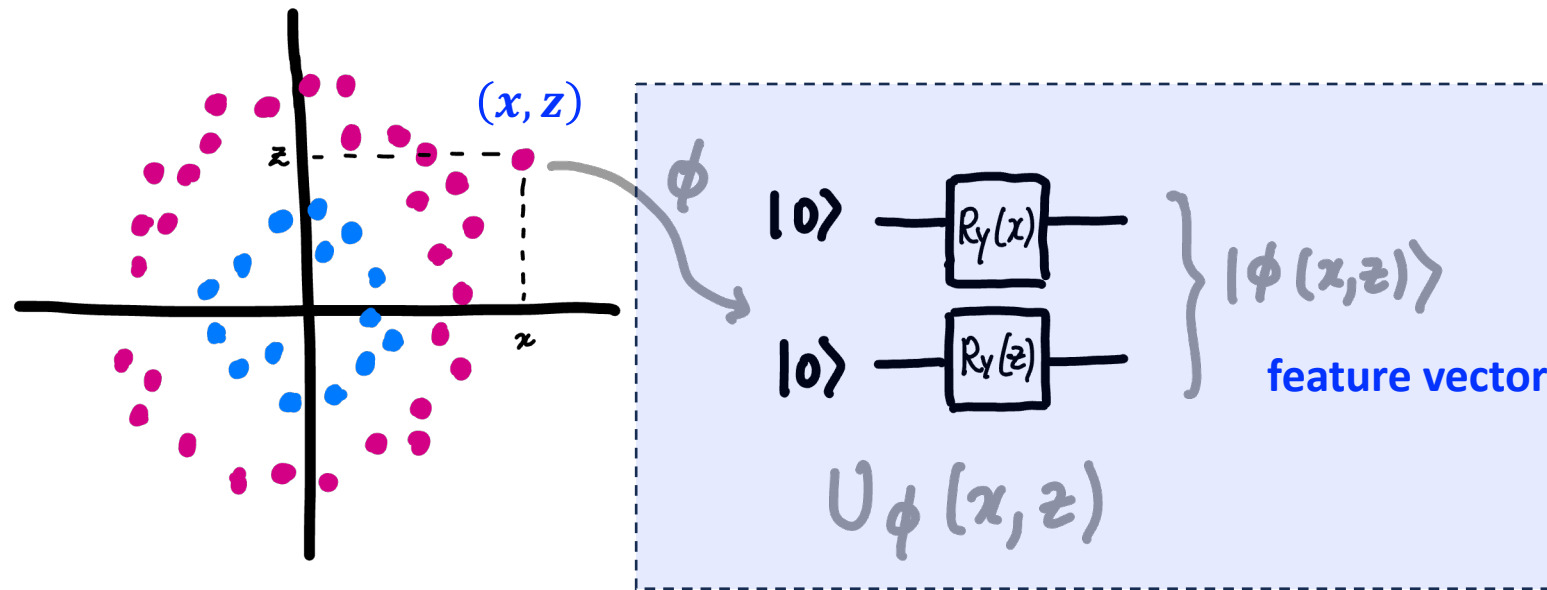
# Feature Map



$\mathcal{X}$  be a set of input data. A **feature map**  $\phi: \mathcal{X} \mapsto \mathcal{F}$  where  $\mathcal{F}$  is the **feature space**. The output of the map  $\phi(x)$  for all  $x \in \mathcal{X}$  are called **feature vectors**.

# Quantum Feature Map

A **quantum feature map**  $\phi: \mathcal{X} \mapsto \mathcal{F}$  is a feature map where the vector space  $\mathcal{F}$  is a Hilbert space and the feature vectors are quantum states. The map transforms  $x \mapsto |\phi(x)\rangle$  by way of a unitary transformation  $U_\phi(x)$ , which is typically a **variational circuit** whose parameters depend on the input data.



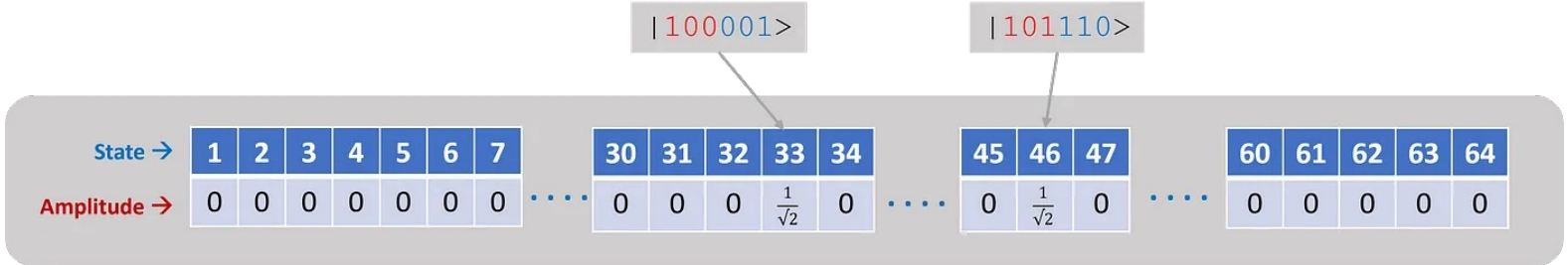
Variational circuit depends on *parameters* and *encoding*

# Basis Encoding

- Basis encoding is primarily used when real numbers have to be arithmetically manipulated in a quantum algorithm. Such an encoding represents **real numbers** as **binary numbers** and then transforms them into a **quantum state** on a computational basis.

Input Data			Pre-processing to convert to Binary form		Basis encoded state	
Input Sample	Feature x1	Feature x2	Binary (x1)	Binary (x2)	Basis encoded Quantum state	Amplitude vector
X <sup>1</sup>	5	6	101	110	101110>	$\frac{1}{\sqrt{2}}$  101110>
X <sup>2</sup>	4	1	100	001	100001>	$\frac{1}{\sqrt{2}}$  100001>

6 qubit = 2<sup>6</sup> = 64 States



Amplitude vector is very sparse , only 2 state has non-zero amplitude out of 64 states



# Real number as bit string (Binary fraction representation)

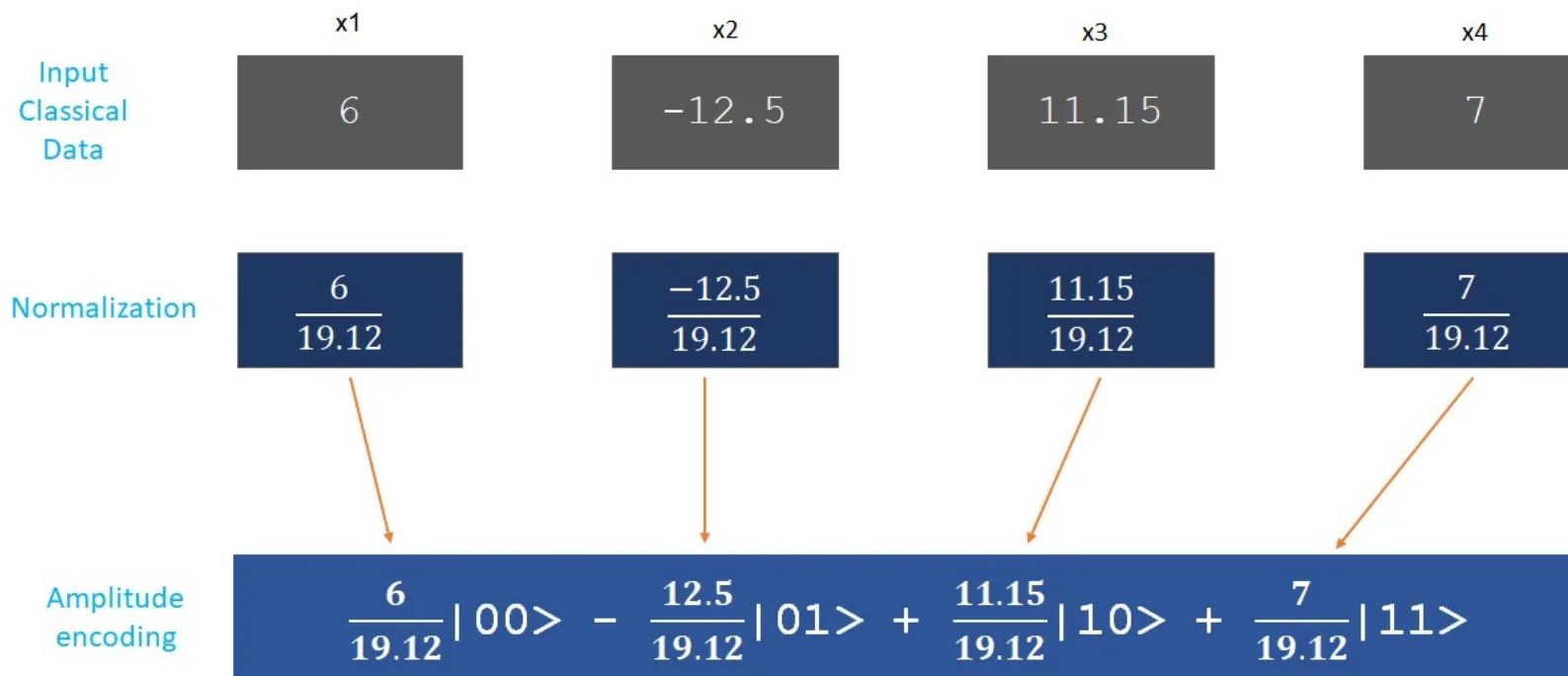
**Definition 3.1** (Fixed-point encoding of real numbers (Rebentrost et al., 2021)). Let  $c_1, c_2$  be positive integers, and  $a \in \{0, 1\}^{c_1}$ ,  $b \in \{0, 1\}^{c_2}$ , and  $s \in \{0, 1\}$  be bit strings. Define the rational number as:

$$\mathcal{Q}(a, b, s) := (-1)^s \left( 2^{c_1-1} a_{c_1} + \dots + 2a_2 + a_1 + \frac{1}{2}b_1 + \dots + \frac{1}{2^{c_2}}b_{c_2} \right) \in [-R, R], \quad (3.2)$$

where  $R := 2^{c_1} - 2^{-c_2}$ . If  $c_1, c_2$  are clear from the context, we can use the shorthand notation for a number  $z := (a, b, s)$  and write  $\mathcal{Q}(z)$  instead of  $\mathcal{Q}(a, b, s)$ . Given an  $n$ -dimensional vector  $v \in (\{0, 1\}^{c_1} \times \{0, 1\}^{c_2} \times \{0, 1\})^n$  the notation  $\mathcal{Q}(v)$  means an  $n$ -dimensional vector whose  $j$ -th component is  $\mathcal{Q}(v_j)$ , for  $j \in [n]$ .

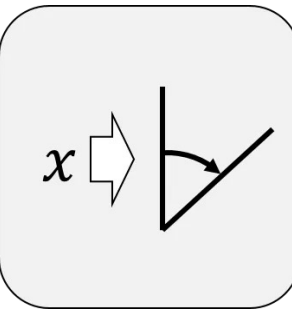
# Amplitude Encoding

- Data is encoded into the *amplitudes* of a quantum state. This encoding requires  $\log_2(n)$  qubits to represent an *n-dimensional data points*.

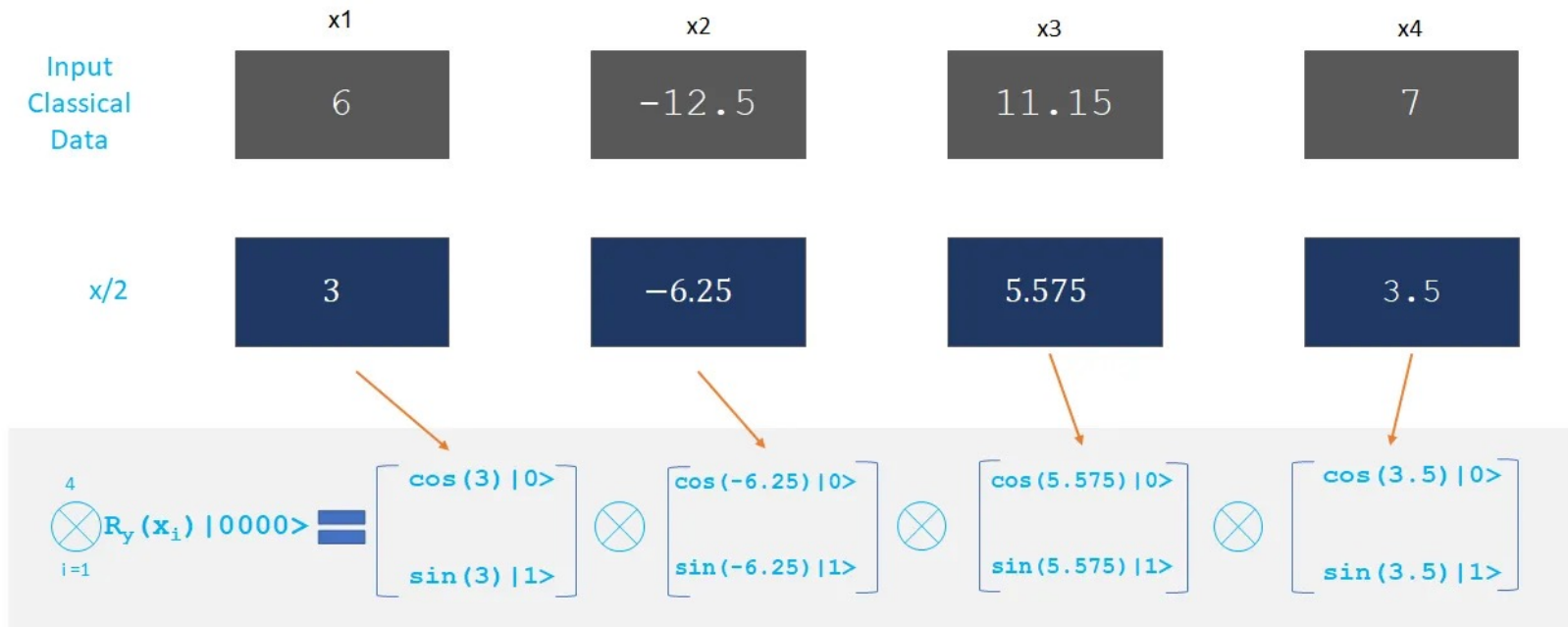


norm factor:  $\sqrt{6^2 + (-12.5)^2 + 11.15^2 + 7^2}$

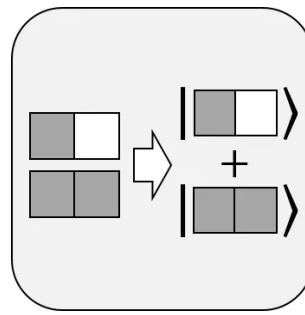
# Angle Encoding (a.k.a tensor product encoding)



- Angle encoding is essentially the most basic form of encoding classical data into a quantum state. The  $n$  classical features are encoded into the rotation angle of the  $n$  qubits. This encoding requires  $n$  qubits to represent  $n$ -dimensional data but is cheaper to prepare in complexity (**constant** circuit depth): it requires one rotation on each qubit,  $R_x(v)$  or  $R_y(v)$  for the value  $v$  to encode.

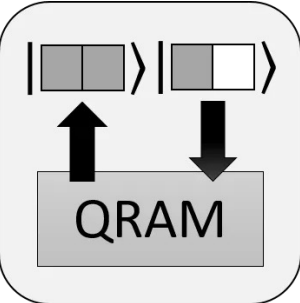


# QuAM (Q- Associative Memory)



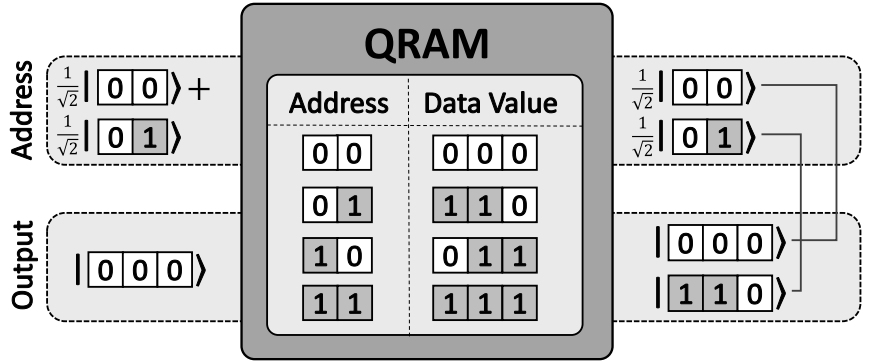
- This encoding is based on superposition to encode a set of data points in a qubit register of the same length. This requires a binary representation of all equally long values, or we need to pad with zeros. We need to use a quantum associative memory (QuAM) to prepare a superposition of basis encoded values in the same qubit register format. Note that the quantum register is an equally weighted superposition of the basis encoded values.

Input variable	Input Classical Data	Binary Number	Basis encoded Quantum Data	QUAM encoded value
X1	10	1010	$ 1010\rangle$	$\frac{1}{\sqrt{3}} 1010\rangle + \frac{1}{\sqrt{3}} 1111\rangle + \frac{1}{\sqrt{3}} 1000\rangle$
X2	15	1111	$ 1111\rangle$	
X3	8	1000	$ 1000\rangle$	



# QRAM (Q-Random Access Memory)

- QRAM is used to access a superposition of data values at once. A **classical RAM** that receives an address with a memory index loads the data stored at the address into an output register. **QRAM** provides the same functionality, but the address and the output register are quantum register. Both the address and the output register can be the superposition of multiple values. For this encoding,  $l$  qubits are needed to encode the data values using Basis encoding. The address register requires  $\log(n)$  additional qubits for a maximum of  $n$  addresses.



$$\frac{1}{\sqrt{m}} \sum_{i=0}^{m-1} |a_i\rangle |0\rangle \mapsto \frac{1}{\sqrt{m}} \sum_{i=0}^{m-1} |a_i\rangle |x_{a_i}\rangle$$

address register (pointing to  $|a_i\rangle$ )  
data register (pointing to  $|x_{a_i}\rangle$ )

**FIGURE 6** Basic functionality of a quantum random access memory (QRAM) is based on [8]. Given an address register that is in a superposition of addresses ( $|00\rangle$  and  $|01\rangle$ ), QRAM creates a superposition of addresses and their data values:  $\frac{1}{\sqrt{2}}|00\rangle|000\rangle + \frac{1}{\sqrt{2}}|01\rangle|110\rangle$

# Qsample encoding

- Qsample encoding is a hybrid case of basis and amplitude encoding. Qsample associates a real amplitude vector with classical discrete probability distributions. We use amplitude, but at the same time, all features are encoded in the qubit.
- Assume there is a pmf random variable  $X$  as  $\Pr(X=i)$ . Any discrete random variable could be represented like it just by indexing the events. For them, we can define qubit state  $|\psi\rangle = \sum_{i=1}^N p_i |i\rangle$ .
- For states of two random variables  $|x\rangle = \sum_{i=1}^{2^n} \sqrt{\Pr(x=i)} |i\rangle$  and  $|y\rangle = \sum_{j=1}^{2^n} \sqrt{\Pr(y=j)} |j\rangle$ . Then, the joint state of both is  $|x, y\rangle = \sum_{i,j=1}^{2^n} \sqrt{\Pr(x=i)\Pr(y=j)} |i\rangle|j\rangle$

# Hamiltonian encoding

- Hamiltonian encoding method encodes the data into the operator. To make some matrix  $A$  to Hamiltonian, we have to make it Hermitian first. Because the definition of Hermitian is  $H = H^*$ , we can make it by  $H = A^*A$ . Then to make it as a unitary matrix, we can use a matrix exponential as  $e^{-iAt}$ . Then, we can develop a unitary evolution as follows,

$$e^{iAt} = e^{i(\alpha + \beta X + \delta Y + \gamma Z)t} = e^{i\alpha t} e^{i\beta t X} e^{i\delta t Y} e^{i\gamma t Z} = R_x(\beta t) R_y(\delta t) R_z(\gamma t).$$

- Note that for any unitary matrix  $A$ , there is a real vector  $(\alpha, \beta, \delta, \gamma)$  such that  $A = (\alpha + \beta X + \delta Y + \gamma Z)$ .
- However, in reality, the above relation is not established because the commutative condition for matrices does not hold  $AB \neq BA$  ( $e^{A+B} \neq e^A e^B$ ).

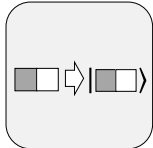
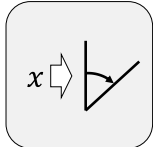
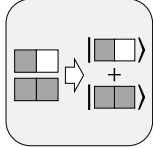
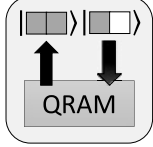
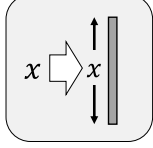
- We can overcome this by using the **Trotter Suzuki formula**

$$\text{For large enough } r, \text{ the following equation holds } e^{-i(H_1+H_2)t} \approx \left( e^{-iH_A t/r} e^{-iH_B t/r} \right)^r$$

- That is, we can implement some hamiltonian unitary matrices by finding the rotating magnitudes and rotation the state with them gradually.

# Summary

**TABLE 1** Comparison of data encoding patterns. For the QUANTUM RANDOM ACCESS MEMORY (QRAM) ENCODING, we assume that all  $n$  data points are loaded

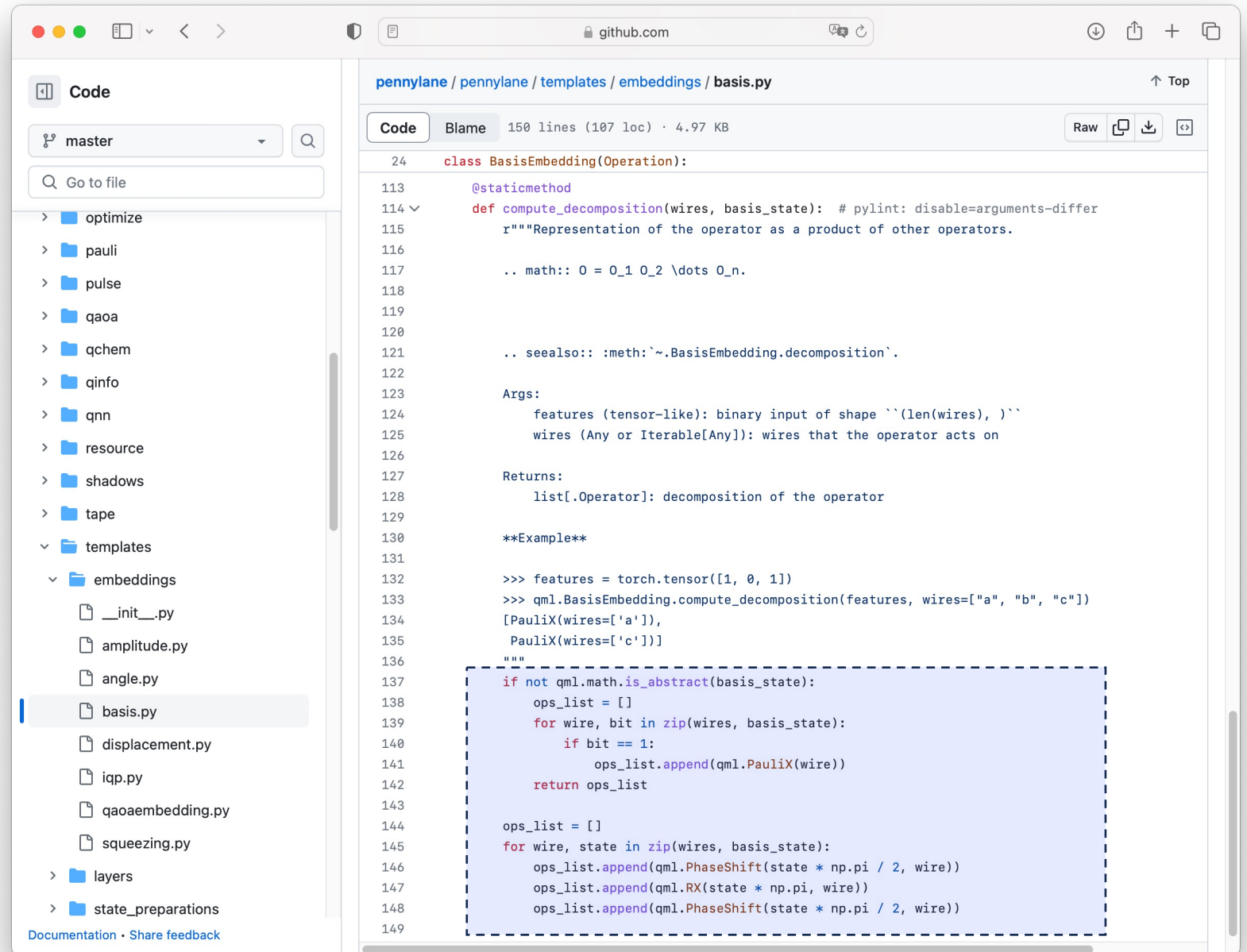
Encoding pattern	Encoding	Req. qubits
 BASIS ENCODING [13]	$x_i \approx \sum_{i=-k}^m b_i 2^i \mapsto  b_m \dots b_{-k}\rangle$	$l = k + m$ per data-point
 ANGLE ENCODING	$x_i \mapsto \cos(x_i)  0\rangle + \sin(x_i)  1\rangle$	1 per data-point
 QUAM ENCODING [13]	$X \mapsto \sum_{i=0}^{n-1} \frac{1}{\sqrt{n}}  x_i\rangle$	$l$
 QRAM ENCODING	$X \mapsto \sum_{i=0}^{n-1} \frac{1}{\sqrt{n}}  i\rangle  x_i\rangle$	$\lceil \log n \rceil + l$
 AMPLITUDE ENCODING [13]	$X \mapsto \sum_{i=0}^{n-1} x_i  i\rangle$	$\lceil \log n \rceil$



# Appendix : Circuit for encoding (Pennylane API)

# Basis encoding

Apply **Pauli-X** for “1”



```
24 class BasisEmbedding(Operation):
113     @staticmethod
114     def compute_decomposition(wires, basis_state): # pylint: disable=arguments-differ
115         r"""Representation of the operator as a product of other operators.
116
117         .. math:: O = O_{1,0_2} \dots O_n.
118
119
120
121         .. seealso:: :meth:`~.BasisEmbedding.decomposition`.
122
123     Args:
124         features (tensor-like): binary input of shape ``(len(wires),)``
125         wires (Any or Iterable[Any]): wires that the operator acts on
126
127     Returns:
128         list[Operator]: decomposition of the operator
129
130     **Example**
131
132     >>> features = torch.tensor([1, 0, 1])
133     >>> qml.BasisEmbedding.compute_decomposition(features, wires=["a", "b", "c"])
134     [PauliX(wires=['a']),
135      PauliX(wires=['c'])]
136
137     """
138     if not qml.math.is_abstract(basis_state):
139         ops_list = []
140         for wire, bit in zip(wires, basis_state):
141             if bit == 1:
142                 ops_list.append(qml.PauliX(wire))
143         return ops_list
144
145     ops_list = []
146     for wire, state in zip(wires, basis_state):
147         ops_list.append(qml.PhaseShift(state * np.pi / 2, wire))
148         ops_list.append(qml.RX(state * np.pi, wire))
149         ops_list.append(qml.PhaseShift(state * np.pi / 2, wire))
```

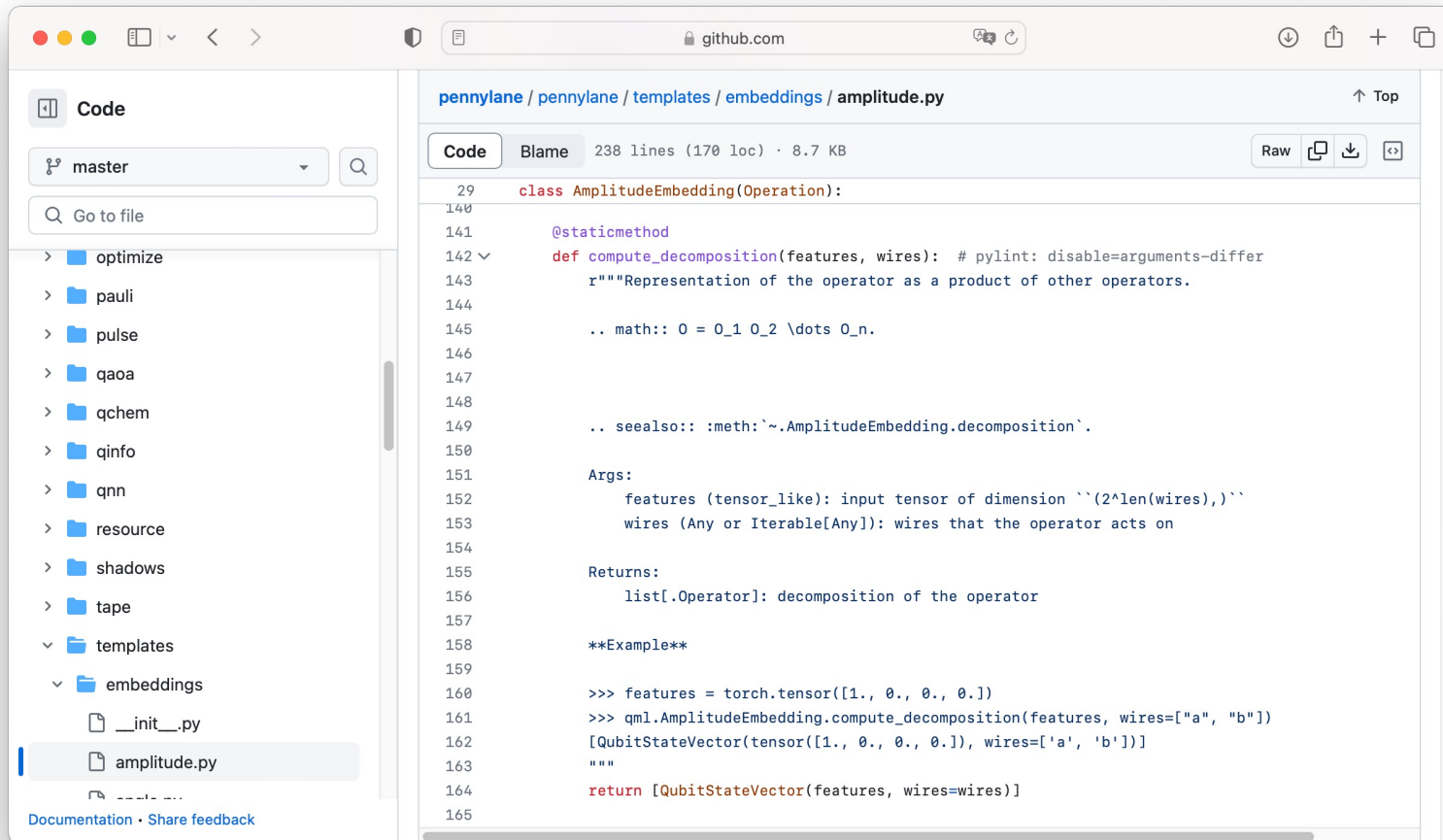
# Basis encoding

```
1 import pennylane as qml
2 from pennylane import numpy as np
3
4 # import the template
5 from pennylane.templates.embeddings import BasisEncoding
6
7 # quantum device where you want to run and how many Qubits
8 dev = qml.device('default.qubit', wires=6)
9
10 @qml.qnode(dev)
11 def circuit(data):
12     for i in range(6):
13         qml.Hadamard(i)
14     for i in range(len(data)):
15         BasisEncoding(features=data[i], wires=range(6), do_queue=True)
16     return qml.state()
17
18 data=[[1,0,1,1,1,0],
19       [1,0,0,0,0,1]]
20
21 circuit(data)
22
23 print(circuit.draw(show_all_wires=True))
24
25 #print output
26
27 0: —H—X—X—| State
28 1: —H————| State
29 2: —H—X————| State
30 3: —H—X————| State
31 4: —H—X————| State
32 5: —H—X————| State
```

pl-basisencoding.py hosted with by GitHub [view raw](#)

```
DataEmbedding — zsh — 97x18
(base) yongsoo@yongsooui-MacBookPro DataEmbedding % python pennylane_embedding.py
/Users/yongsoo/anaconda3/lib/python3.10/site-packages/pennylane/operation.py:1034: UserWarning: The do_queue keyword argument is deprecated. Instead of setting it to False, use qml.queueing.QueueingManager.stop_recording()
  warnings.warn(do_queue_deprecation_warning, UserWarning)
ic| basis_state: [1, 0, 1, 1, 1, 0]
ic| ops_list: [PauliX(wires=[0]), PauliX(wires=[2]), PauliX(wires=[3]), PauliX(wires=[4])]
ic| basis_state: [1, 0, 0, 0, 0, 1]
ic| ops_list: [PauliX(wires=[0]), PauliX(wires=[5])]
(base) yongsoo@yongsooui-MacBookPro DataEmbedding %
```

# Amplitude encoding



The screenshot shows a GitHub repository for pennylane, specifically the file `templates/embeddings/amplitude.py`. The code defines a class `AmplitudeEmbedding` with a static method `compute_decomposition`. The method takes `features` (a tensor) and `wires` (a list) as input and returns a list of operators representing the decomposition of the operator.

```
29 class AmplitudeEmbedding(Operation):
140
141     @staticmethod
142     def compute_decomposition(features, wires): # pylint: disable=arguments-differ
143         r"""Representation of the operator as a product of other operators.
144
145         .. math:: O = O_{1,0_2} \dots O_{0,n}.
146
147
148         .. seealso:: :meth:`~.AmplitudeEmbedding.decomposition`.
149
150         Args:
151             features (tensor_like): input tensor of dimension ``(2^len(wires),)``
152             wires (Any or Iterable[Any]): wires that the operator acts on
153
154         Returns:
155             list[Operator]: decomposition of the operator
156
157         **Example**
158
159         >>> features = torch.tensor([1., 0., 0., 0.])
160         >>> qml.AmplitudeEmbedding.compute_decomposition(features, wires=["a", "b"])
161         [QubitStateVector(tensor([1., 0., 0., 0.]), wires=['a', 'b'])]
162         """
163
164         return [QubitStateVector(features, wires=wires)]
165
```

??