Quantum Machine Learning

Chuong Nguyen

March 16, 2022

Contents

List of Figures

List of Tables

Preface

This lecture note focuses on three parts: introduce Machine Learning and Quantum Machine Learning, Quantum Neural Network in terms of Variational Quantum Eigensolver and Quantum Kernel. Then we give some examples about supervised learning with code in Pennylane

Introduce Machine Learning and Quantum Machine Learning

We will explain what is machine learning and give some intuitions about types of machine learning algorithms. Then we quantize machine learning into quantum machine learning.

Quantum Neural Network

In this part, we present Quantum Neural Network (QNN), one of the most popular structures of quantum machine learning. How to build and train it.

Quantum Kernel

The last part is about Quantum Kernel, which is another approach for quantum machine learning, then we build one Quantum Kernel for the real dataset to show its application

1

Introduce Machine Learning

"Machine learning is the science of getting computers to learn without being explicitly programmed."

– Sebastian Thrun

1.1 What is Machine Learning?

Technically, machine learning is the computational algorithms that can automatically execute some task without explicit instructions from humans just by learning how to do it from data. That means machine learning sometimes is a black box, giving an input (or a sample) and we have an output as a prediction or decision for our problem without knowing what is actually happening inside the model. There are many ways to categorize machine learning algorithms based on their learning style or their similarity in form or function.

1.1.1 Algorithms Grouped by Learning Style

The first is a grouping of algorithms by their learning style:

- **Supervised Learning**: Input data is called training data and has a known label or result such as spam/not-spam or a stock price at a time.
  - A model is prepared through a training process in which it is required to make predictions and is corrected when those predictions are wrong. The training process continues until the model achieves a desired level of accuracy on the training data.
  - Example problems are classification and regression.
  - Example algorithms include: Logistic Regression and the Back Propagation Neural Network.

- **Unsupervised Learning**: Input data is not labeled and does not have a known result.
  - A model is prepared by deducing structures present in the input data. This may be to extract general rules. It may be through a mathematical process to systematically reduce redundancy, or it may be to organize data by similarity.
  - Example problems are clustering, dimensionality reduction and association rule learning.
  - Example algorithms include: the Apriori algorithm and K-Means.
- **Semi-Supervised Learning**: Input data is a mixture of labeled and unlabelled examples.
  - There is a desired prediction problem but the model must learn the structures to organize the data as well as make predictions.
  - Example problems are classification and regression.
  - Example algorithms are extensions to other flexible methods that make assumptions about how to model the unlabeled data.
- **Reinforcement learning**: is a behavioral machine learning model that is similar to supervised learning, but the algorithm is not trained using sample data. This model learns as it goes by using trial and error. A sequence of successful outcomes will be reinforced to develop the best recommendation or policy for a given problem. This is currently being used in combination with deep learning to model more biologically plausible and powerful neural networks, that can for example maybe solve the Go game problem (see Google's DeepMind AlphaGo).

1.1.2 Algorithms Grouped By Similarity

Algorithms are often grouped by similarity in terms of their function (how they work). For example, tree-based methods, and neural network inspired methods.

- Regression Algorithms.
- Decision Tree Algorithms.
- Bayesian Algorithms.
- Clustering Algorithms.
- Artificial Neural Network Algorithms.
- Deep Learning Algorithms.
- etc.

1.2 Quantum Machine Learning

We can think of quantum machine learning (QML) as an intersection between machine learning and quantum computing. For example, we can use a quantum computer to speed up the training process of machine learning models or find new machine learning models that can execute on a quantum computer.

UC Berkeley[1] breaks out the learning system of a machine learning algorithm into three main parts.

- **A Decision Process (Representation)**: In general, machine learning algorithms are used to make a prediction or classification. Based on some input data, which can be labelled or unlabeled, your algorithm will produce an estimate about a pattern in the data.
- **An Error Function (Evaluation)**: An error function serves to evaluate the prediction of the model. If there are known examples, an error function can make a comparison to assess the accuracy of the model.
- **An Model Optimization Process (Optimization)**: If the model can fit better to the data points in the training set, then weights are adjusted to reduce the discrepancy between the known example and the model estimate. The algorithm will repeat this evaluate and optimize process, updating weights autonomously until a threshold of accuracy has been met.

We can take advantage of quantum computers in two of three parts above to make machine learning becoming more efficient:

- A Decision Process: using quantum computers like neural networks or making machine learning available on near-term quantum devices.
- An Model Optimization Process: using quantum computers as AI accelerators.

[1] https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/

2

Quantum Neural Network

2.1 Variational Quantum Circuits

As we have discussed above, one of the two approaches of quantum machine learning is building and training a quantum circuit as a machine learning model. There are many ways to do that, such as Killoran et al. (2018), Iris Cong et al. (2018), Lorenzo Buffoni and Filippo Caruso (2021) or Yunseok Kwak et al. (2021). For the basic level, we introduce the Variational Quantum Circuits (VQC) to demonstrate how a basic quantum neural network(QNN) works with VQC.

A variational quantum circuit (VQC) is a quantum circuit using rotation operator gates with free parameters to perform various numerical tasks, such as approximation, optimization, classification. An algorithm using a variational quantum circuit is called variational quantum algorithm (VQA), which is a classical-quantum hybrid algorithm because its parameter optimization is often performed by a classical computer. Since its universal function approximating property, many algorithms using VQC are designed to solve various numerical problems. This flow led to many applications of VQA in machine learning and is also for replacing the artificial neural network of the existing model with VQC. VQC is similar to artificial neural networks in that it approximates functions through parameter learning, but has differences due to the several characteristics of quantum computing. Since all quantum gate operations are reversible linear operations, quantum circuits use entanglement layers instead of activation functions to have multilayer structures. These VQCs are called quantum neural networks (QNN).

The way a QNN processes data is as follows:

1. The input data is encoded into the corresponding qubit state of an appropriate number of qubits.

2. Then, the qubit state is transformed through the parameterized rotation gates and entangling gates for a given number of layers.

3. The transformed qubit state is then measured by obtaining expected value of a hamiltonian operator, such as Pauli gates.

4. These measurements are decoded back into the form of appropriate output data.

5. The parameters are then updated by an optimizer like Adam optimizer. A neural network constructed in the form of VQC can perform various roles in various forms, which will be explored as quantum neural networks.

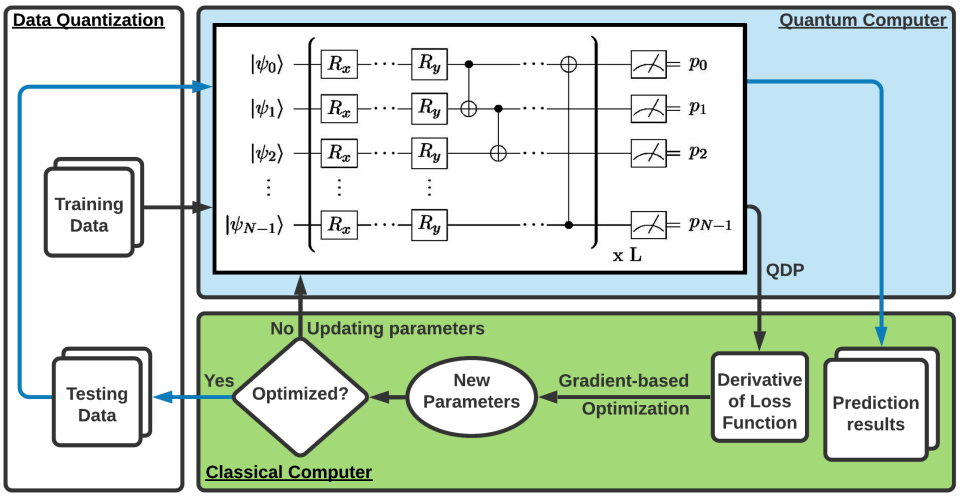


Figure 2.1: A Quantum Neural Network framework.

The blue box is the quantum part of the QNN, and the green box is the classical part of the QNN. At first, quantify and encode the dataset (training and testing) into quantum states of qubits $|\psi_i\rangle$. Each feature in the dataset encodes into one qubit. The employed rotation gates will parameterize the quantum circuit, and the CNOT gates cause entangled in the circuit. These gates repeat L times for L layers in the quantum neural network. After that, we measure the circuit and give the corresponding probabilities p_k. We employ a QDP scheme with the pentameter-shift rule to calculate the derivative of p_k and send the results to a classical computer to derive the derivative of the loss function. After that, we implement a gradient-based optimization to obtain new parameters. When the scheme is not optimal yet, we update the circuit with new parameters; when it is optimal, we turn it to the testing process.

2.1.1 Quantum Encoding (or Embedding)

For almost all problems in machine learning, the data we all have are classical data. To make this data to be understood in the quantum machine, classical data need to be encoded into the corresponding quantum state in the Hilbert space. There are two standard methods are usually used to encode data. Given an input vector $\boldsymbol{x} \in \mathbb{R}^n$, the encoding process U performs one of two things below:

- $U(\boldsymbol{x}) = |x_1\rangle \otimes |x_2\rangle \otimes \cdots \otimes |x_n\rangle$, or
- $U(\boldsymbol{x}) = U(x_1)\,|1\rangle + U(x_2)\,|2\rangle + \cdots + U(x_n)\,|n\rangle$, where $U(x_i)$ is the amplitude of state $|i\rangle$

Based on the particular form of U, some quantum encoding is listed in Tak Hur et al. (2021) as shown in the following:

Amplitude encoding

The amplitude encoding represents input data of $\boldsymbol{x} = (x_1, ..., x_N)^T$ of dimension $N = 2^n$ as amplitudes of an n-qubit quantum state $|\phi(\boldsymbol{x})\rangle$ as

$$U_\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^N \rightarrow |\phi(\boldsymbol{x})\rangle = \frac{1}{||\boldsymbol{x}||} \sum_{i=1}^{N} x_i\,|i\rangle$$

Qubit encoding

The qubit encoding maps input data of $\boldsymbol{x} = (x_1, ..., x_N)^T$ to N qubits as

$$U_\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^N \rightarrow |\phi(\boldsymbol{x})\rangle = \bigotimes_{i=1}^{N} \left(\cos\left(\frac{x_i}{2}\right)|0\rangle + \sin\left(\frac{x_i}{2}\right)|1\rangle\right)$$

where $x_i \in [0, \pi)$ for all i.

Dense qubit encoding

The dense qubit encoding maps input data of $\boldsymbol{x} = (x_1, ..., x_N)^T$ to $N/2$ qubits as

$$U_\phi(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^N \rightarrow |\phi(\boldsymbol{x})\rangle = \bigotimes_{j=1}^{N/2} \left(e^{-i\frac{x_{N/2+j}}{2}\sigma_y} e^{-i\frac{x_j}{2}\sigma_x}\,|0\rangle\right)$$

2.1.2 Structure of quantum circuits

The structure of QNN consists of two components: rotation gates and entangling gates. Different combinations of two types of quantum gates gives different QNN as listed in Tobias Haug et al. (2021). The choice of structures is based on the characteristics of the data that our problem needs.

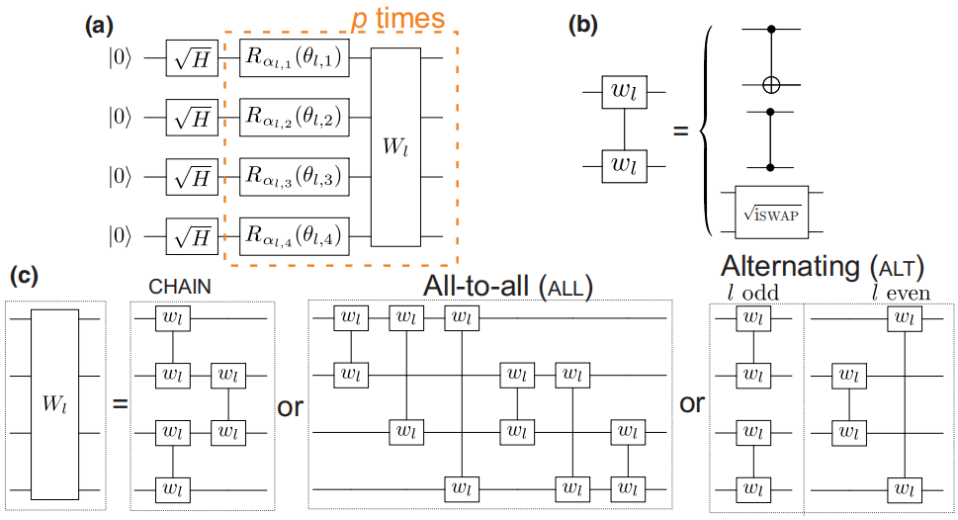


Figure 2.2: Some QNN structure designs in Tobias Haug et al. (2021).

2.1.3 Quantum Differentiable Programming

Given a quantum state $|\psi(\vec{\theta})\rangle$ with $\vec{\theta}$ as variational parameters and an observable $\widehat{C}$, the task is to seek the global minimum of the expectation value $C(\vec{\theta}) = \langle\psi(\vec{\theta})|\widehat{C}|\psi(\vec{\theta})\rangle$ with respect to parameters $\vec{\theta}$. For example, if $\widehat{C}$ is a Hamiltonian, its global minimum is the ground state energy. In general, $C(\vec{\theta})$ is called as the cost function, and minimizing the cost function requires its derivative with respect to parameters $\vec{\theta}$, $\partial C(\vec{\theta})/\partial\theta$. In classical computing, if the analytical form of $\partial C(\vec{\theta})/\partial\theta$ is unknown, finite difference methods are often used to evaluate the derivative approximately. Although this approximation is fast and easy to implement, its accuracy depends on discretization steps. In contrast to the classical finite differentiation, quantum differentiable programming (QDP) is an automatic and exact method to compute the derivative of a function. QDP is thus essential for accurate gradient computation in multiple VQAs, including QML models.

The heart of QDP is the parameter-shift rule that is analytically computed using quantum circuits. The algorithm is outlined in Algorithm 1. Let us introduce a parameterized generator $\widehat{V}$ independent of $\vec{\theta}$ such that $|\psi(\vec{\theta})\rangle =$

Algorithm 1: QDP implementation in Qsun

Result: Derivative of a function
$f \leftarrow$ Function; $c \leftarrow$ Quantum Circuit;
$p \leftarrow$ Params; $s \leftarrow$ Shift;
$diff \leftarrow$ [0, ..., 0], size($diff$) = size(p);
for $i \leftarrow$ *0,* size($diff$) **do**
  $p_$plus $\leftarrow$ copy(p);
  $p_$subs $\leftarrow$ copy(p);
  $p_$plus[i] $\leftarrow p[i] + s$;
  $p_$subs[i] $\leftarrow p[i] - s$;
  diff[i] $\leftarrow (f(c, p\_\text{plus}) - f(c, p\_\text{subs}))/(2*\sin(s))$;
end

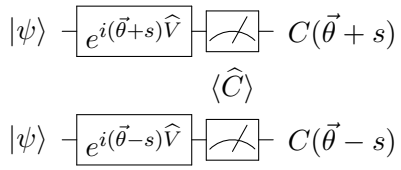
$e^{i\vec{\theta}\widehat{V}}|\psi\rangle$. The cost function is then rewritten as

$$C(\vec{\theta}) = \langle\psi|e^{-i\vec{\theta}\widehat{V}}\widehat{C}e^{i\vec{\theta}\widehat{V}}|\psi\rangle = \mathrm{Tr}(\widehat{C}e^{\mathcal{Z}}[\rho]), \tag{2.1}$$

with $\rho = |\psi\rangle\langle\psi|$; $\mathcal{Z} = i\vec{\theta}\widehat{V}$, and the superoperator $e^{\mathcal{Z}}[\rho] = e^{-\mathcal{Z}}\rho e^{\mathcal{Z}}$. The parameter-shift rule for each $\theta \in \vec{\theta}$ states that

$$\frac{\partial C(\vec{\theta})}{\partial\theta} = \mathrm{Tr}(\widehat{C}\partial_\theta e^{\mathcal{Z}}[\rho]) = c[C(\vec{\theta}+s) - C(\vec{\theta}-s)], \tag{2.2}$$

where $c = 1/\sin(s)$, and s is determined based on the superoperator and independent of $\vec{\theta}$. The values of the cost function C at $\vec{\theta} \pm s$ are measured on quantum computers by implementing two quantum circuits as follows



The derivative is finally obtained by subtracting measurement results from the two circuits. An advantage of the parameter-shift rule is that it can compute the derivative of the given function exactly while the shift s can be chosen arbitrarily large.

2.2 Example

In this example we show how a variational circuit can be used to learn a fit for a one-dimensional function when being trained with noisy samples from that function. The variational circuit we use is the continuous-variable

quantum neural network model described in Killoran et al. (2018) and is executed in the code below. You can read more about this example in here.

```
import pennylane as qml
from pennylane import numpy as np
from pennylane.optimize import AdamOptimizer

dev = qml.device("strawberryfields.fock", wires=1, cutoff_dim
    =10)

def layer(v):
    # Matrix multiplication of input layer
    qml.Rotation(v[0], wires=0)
    qml.Squeezing(v[1], 0.0, wires=0)
    qml.Rotation(v[2], wires=0)

    # Bias
    qml.Displacement(v[3], 0.0, wires=0)

    # Element-wise nonlinear transformation
    qml.Kerr(v[4], wires=0)

@qml.qnode(dev)
def quantum_neural_net(var, x=None):
    # Encode input x into quantum state
    qml.Displacement(x, 0.0, wires=0)

    # "layer" subcircuits
    for v in var:
        layer(v)

    return qml.expval(qml.X(0))

def square_loss(labels, predictions):
    loss = 0
    for l, p in zip(labels, predictions):
        loss = loss + (l - p) ** 2

    loss = loss / len(labels)
    return loss

def cost(var, features, labels):
    preds = [quantum_neural_net(var, x=x) for x in features]
    return square_loss(labels, preds)

# source: https://raw.githubusercontent.com/XanaduAI/pennylane/
    v0.3.0/examples/data/sine.txt
data = np.loadtxt("data/sine.txt")
X = data[:, 0]
Y = data[:, 1]

np.random.seed(0)
num_layers = 4
var_init = 0.05 * np.random.randn(num_layers, 5)
```

```
50
51 opt = AdamOptimizer(0.01, beta1=0.9, beta2=0.999)
52
53 var = var_init
54 for it in range(500):
55     var = opt.step(lambda v: cost(v, X, Y), var)
56     print("Iter: {:5d} | Cost: {:0.7f} ".format(it + 1, cost(
       var, X, Y)))
57
58 x_pred = np.linspace(-1, 1, 50)
59 predictions = [quantum_neural_net(var, x=x_) for x_ in x_pred]
60
61 plt.figure()
62 plt.scatter(X, Y)
63 plt.scatter(x_pred, predictions, color="green")
64 plt.xlabel("x")
65 plt.ylabel("f(x)")
66 plt.tick_params(axis="both", which="major")
67 plt.tick_params(axis="both", which="minor")
68 plt.show()
```

Listing 2.1: Example of Quantum Neural Network.

3

Quantum Kernel

3.1 Feature Map

There are many classification problems that can be transformed into regression problems by introducing a term: decision boundary. While training a classifier on a dataset, using a specific classification algorithm, it is required to define a set of hyper-planes, called Decision Boundary, that separates the data points into specific classes, where the algorithm switches from one class to another. On one side a decision boundary, a data point is more likely to be called class A - on the other side of the boundary, it's more likely to be called class B. For instance, a logistic regression is defined as:

$$\hat{y}(\boldsymbol{x}) = sign(F(\boldsymbol{w}, \boldsymbol{x})) = \begin{cases} 1, F(\boldsymbol{w}, \boldsymbol{x})) > 0.5 \\ 0, F(\boldsymbol{w}, \boldsymbol{x})) \leq 0.5 \end{cases} \tag{3.1}$$

where $\boldsymbol{w}$ is a trained-parameter vector and $\boldsymbol{x}$ is an input.

In the real-life, datasets that are not separable by hyper-planes can't be classified without error, this means that our classification model of those datasets is non-linear and is not easy to construct and train it. That is when feature maps appear to make things easier. A feature map $\phi(\boldsymbol{x})$ is a function which maps a data vector to feature space. The main logic in machine learning for doing so is to present your learning algorithm with data that it is better able to regress or classify. Then instead of training models on the original non-linear space, now we train our model on the linear feature space, which can have better results and be easy to train. A quantum version of the feature map is called quantum feature map, and it is exactly the quantum encoding that we have discussed above.

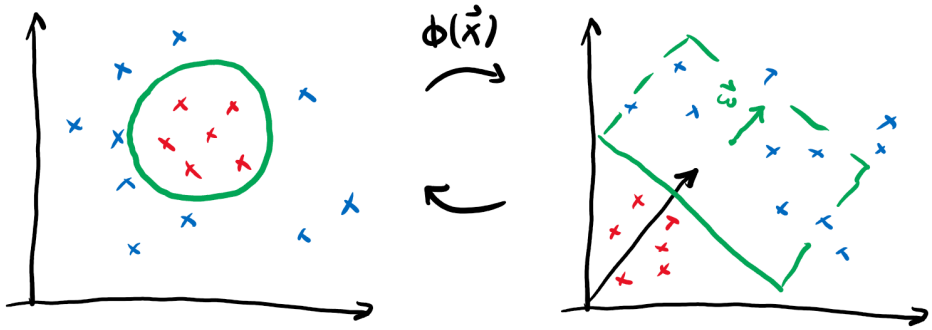


Figure 3.1: Example about feature map, source: pennylane.

3.2 Quantum Kernel method

Kernel methods (or kernel tricks) is a technique that uses to improve the performance of supervised learning models with associated learning algorithms that analyze data for classification called Support Vector Machine, or SVM for short. Formally, kernel methods define a kernel function which has a form $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. A kernel function returns the inner product between two points in a space, which means you can kernel a kernel function will tell you the "distance" between two data points, and SVMs require $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for all possible couples of two data points (i, j) for N data points in the dataset (a kernel matrix). When giving a feature map $\phi(\boldsymbol{x}_i)$, the kernel function becomes $K(\phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j))$. By this definition, we can work directly with kernel function and bypass the explicit expression of the feature map.

Name	Formula
linear	$(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$
polynomial	$(r + \gamma \boldsymbol{x}_i \cdot \boldsymbol{x}_j)^d$
sigmoid	$\tanh(r + \gamma \boldsymbol{x}_i \cdot \boldsymbol{x}_j)$
rbf	$\exp(-\gamma \lvert\lvert \boldsymbol{x}_i - \boldsymbol{x}_i \rvert\rvert_2^2)$

Table 3.1: Classical kernel functions $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$

A quantum version of quantum kernel is defined as:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = |\langle \phi(\boldsymbol{x}_j) | \phi(\boldsymbol{x}_i) \rangle|^2$$

where $|\phi(\boldsymbol{x}_i)\rangle = S(\boldsymbol{x}_i) |0 \cdots 0\rangle$, $S(\boldsymbol{x}_i)$ is a quantum encoding operator. And its quantum circuit is shown in Fig. 3.2.

Using measure operator $M = |0 \cdots 0\rangle \langle 0 \cdots 0|$, we obtain

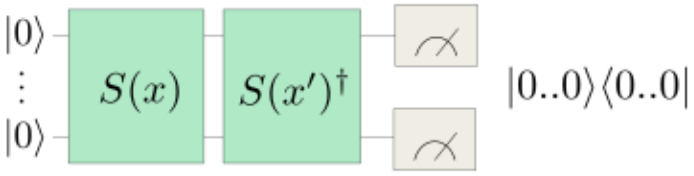


Figure 3.2: Quantum kernel circuit, source: pennylane.

$$
\begin{aligned}
&\langle 0\cdots 0|\, S(\boldsymbol{x}_j)S^\dagger(\boldsymbol{x}_i)MS^\dagger(\boldsymbol{x}_j)S(\boldsymbol{x}_i)\,|0\cdots 0\rangle \\
&= \langle 0\cdots 0|\, S(\boldsymbol{x}_j)S^\dagger(\boldsymbol{x}_i)\,|0\cdots 0\rangle\, \langle 0\cdots 0|\, S^\dagger(\boldsymbol{x}_j)S(\boldsymbol{x}_i)\,|0\cdots 0\rangle \\
&= \left|\langle 0\cdots 0|\, S^\dagger(\boldsymbol{x}_j)S(\boldsymbol{x}_i)\,|0\cdots 0\rangle\right|^2 \\
&= |\,\langle\phi(\boldsymbol{x}_j)|\phi(\boldsymbol{x}_i)\rangle\,|^2 \\
&= K(\boldsymbol{x}_i, \boldsymbol{x}_j)
\end{aligned}
\tag{3.2}
$$

That is a quantum kernel that we have defined above. As Maria Schuld has shown in her paper Maria Schuld (2021), each quantum kernel corresponds to one quantum encoding. After obtaining a quantum kernel by measuring a corresponding quantum circuit. We replace it with a classical kernel in SVM and train a new SVM on a classical computer as shown in Fig 3.3.

```
import numpy as np

from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import pennylane as qml
from pennylane.templates import AngleEmbedding

np.random.seed(42)

X, y = load_iris(return_X_y=True)

# pick inputs and labels from the first two classes only,
# corresponding to the first 100 samples
X = X[:100]
y = y[:100]

```

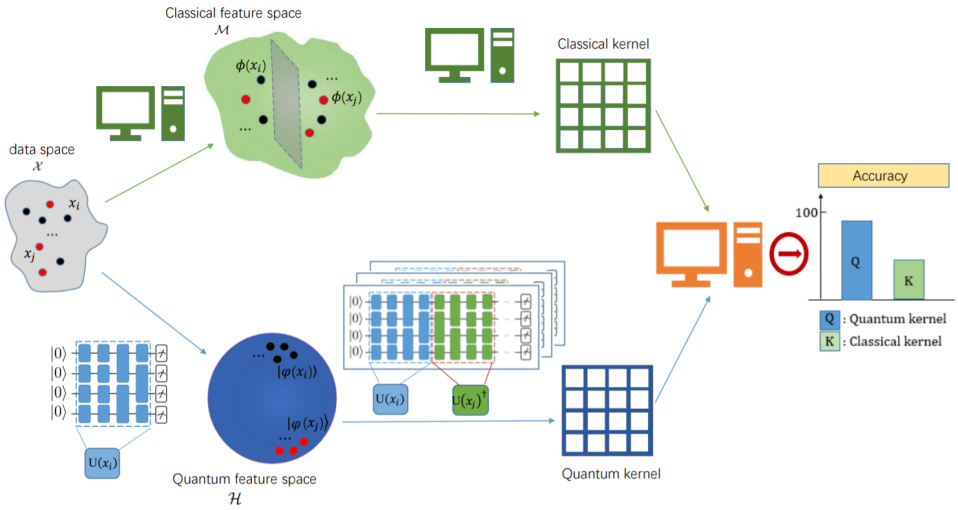


Figure 3.3: Quantum kernel circuit, source: Xinbiao Wang et al. (2021).

```
# scaling the inputs is important since the embedding we use is
    periodic
scaler = StandardScaler().fit(X)
X_scaled = scaler.transform(X)

# scaling the labels to -1, 1 is important for the SVM and the
# definition of a hinge loss
y_scaled = 2 * (y - 0.5)

X_train, X_test, y_train, y_test = train_test_split(X_scaled,
    y_scaled)

n_qubits = len(X_train[0])

dev_kernel = qml.device("default.qubit", wires=n_qubits)

projector = np.zeros((2**n_qubits, 2**n_qubits))
projector[0, 0] = 1

@qml.qnode(dev_kernel)
def kernel(x1, x2):
    """The quantum kernel."""
    AngleEmbedding(x1, wires=range(n_qubits))
    qml.adjoint(AngleEmbedding)(x2, wires=range(n_qubits))
    return qml.expval(qml.Hermitian(projector, wires=range(
    n_qubits)))

def kernel_matrix(A, B):
    """Compute the matrix whose entries are the kernel
       evaluated on pairwise data from sets A and B."""
    return np.array([[kernel(a, b) for b in B] for a in A])

```

```
svm = SVC(kernel=kernel_matrix).fit(X_train, y_train)
predictions = svm.predict(X_test)
print(accuracy_score(predictions, y_test))
```

Listing 3.1: Example of Quantum Kernel: source

3.3 Compare with VQC

The difference between the quantum kernel and VQE can be shown in Fig 3.4.

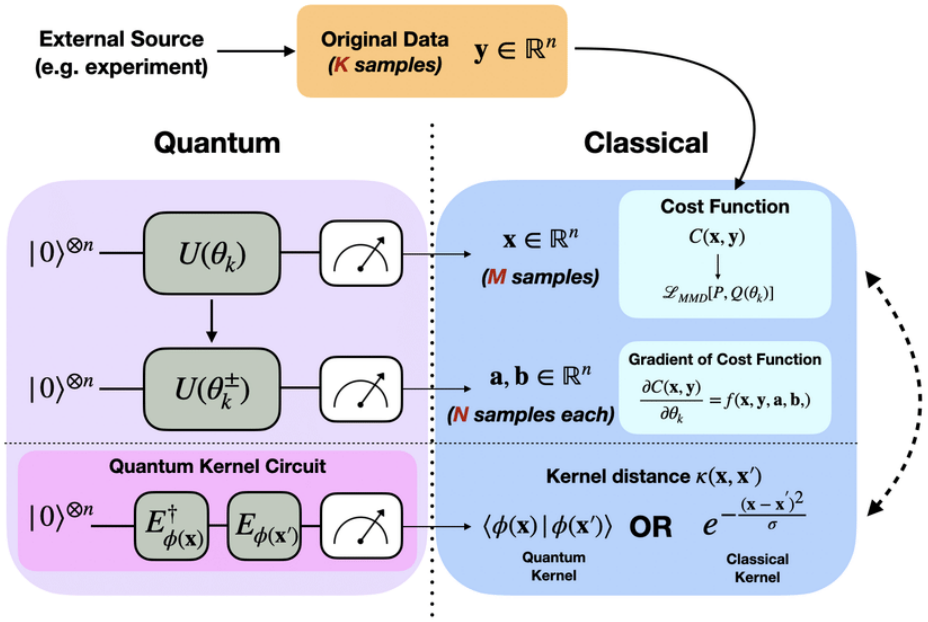


Figure 3.4: Compare quantum kernel and VQC, source: Ieva Čepaitė et al. (2020).

The answer to the question when we use quantum kernel instead of VQC depends on the characteristic of the dataset. Experimentally, the quantum kernel shows a better performance than VQC because the number of measurements (evaluations) on the quantum kernel is much less than VQC with specific datasets, as shown in Fig 3.5. Besides that, with appropriate quantum encoding, the parameter space of quantum kernel becomes convex. This means that we can gain a global minimum, which is very hard with VQC (where we only obtain local minimum), as shown in Fig. 3.6.

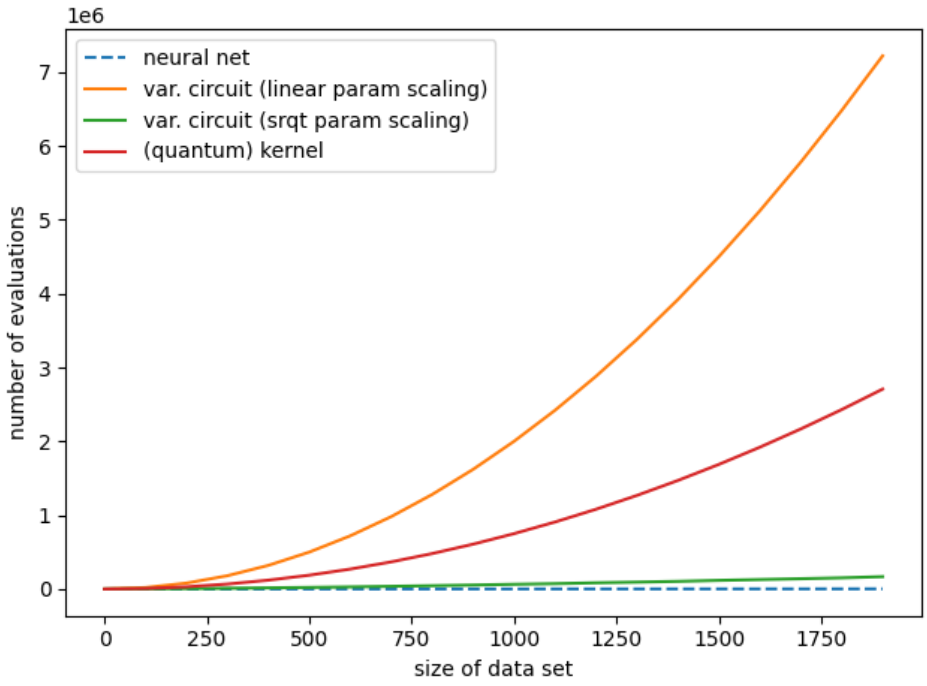


Figure 3.5: Compare the number of measurements on each machine learning model, source: pennylane.

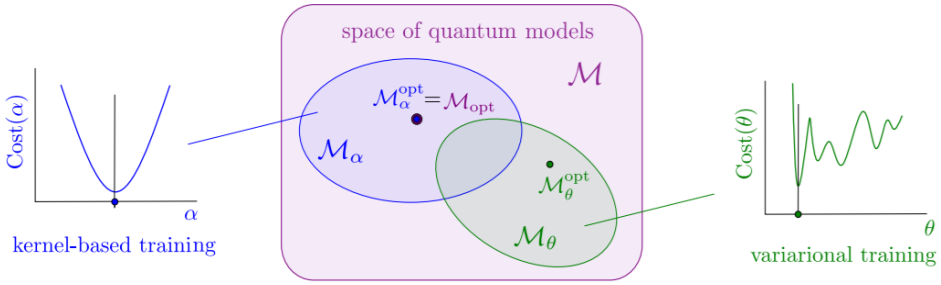


Figure 3.6: the parameter space, source: Maria Schuld (2021).