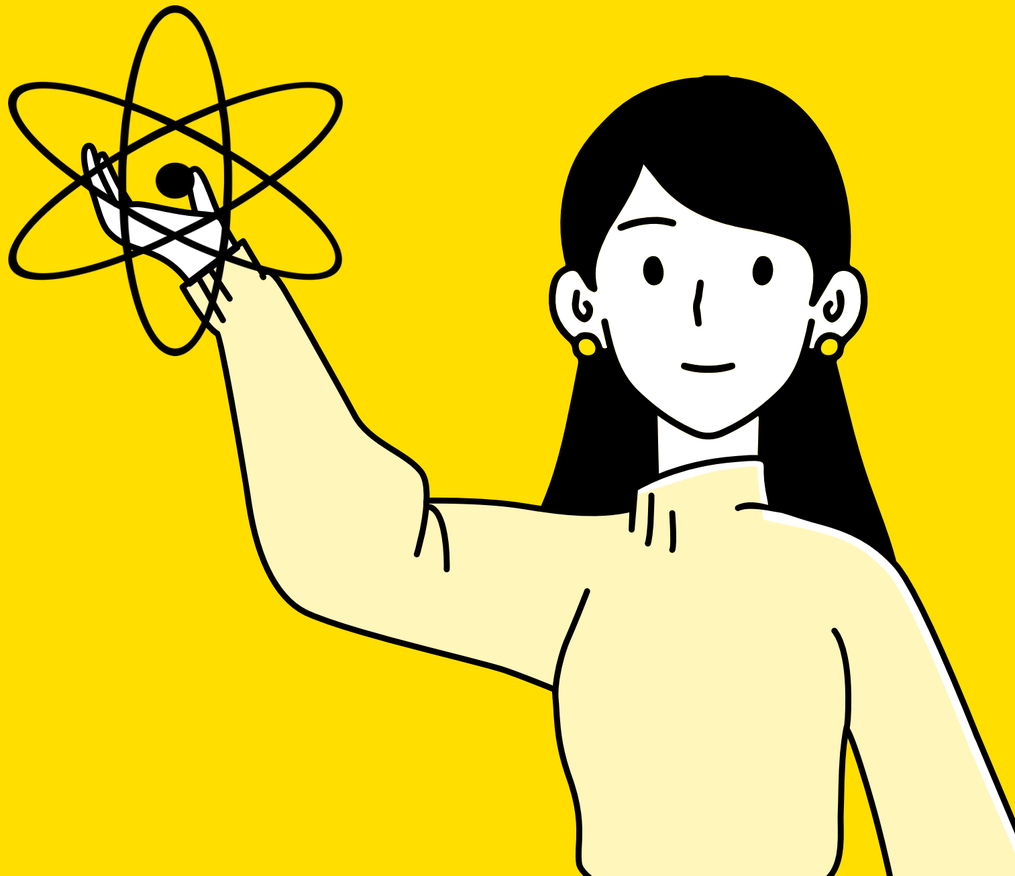
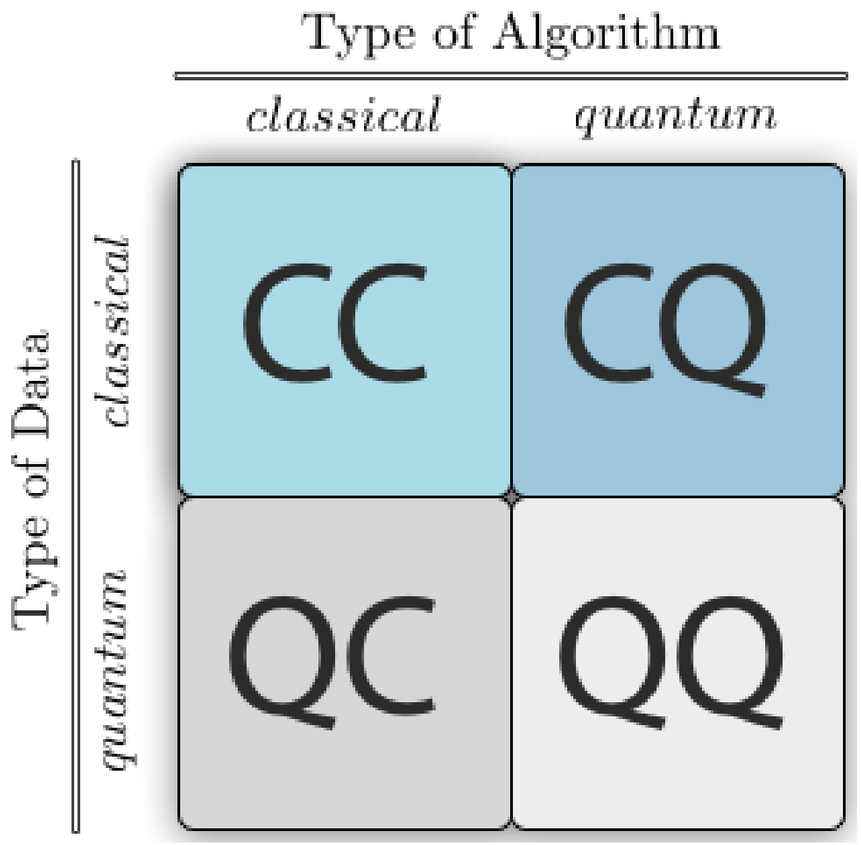
QUANTUM MACHINE LEARNING



A hybrid approach

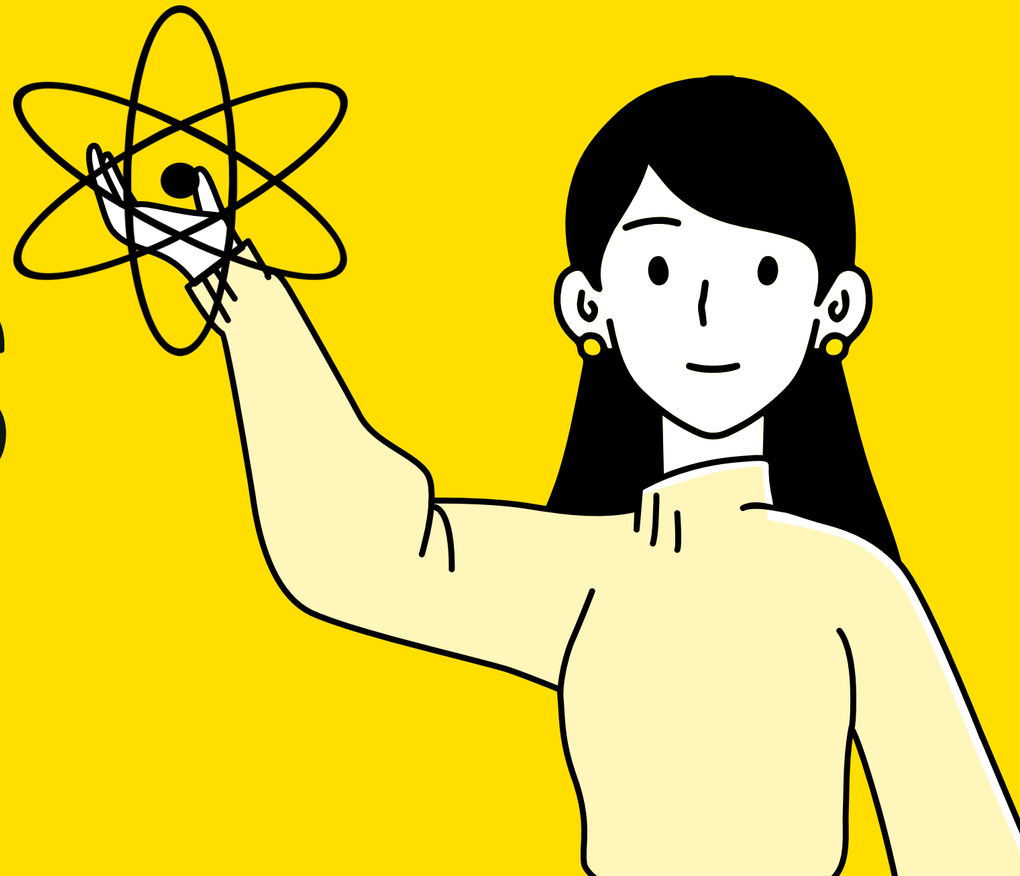
Quantum machine learning is the integration of quantum algorithms within machine learning programs.



Focus

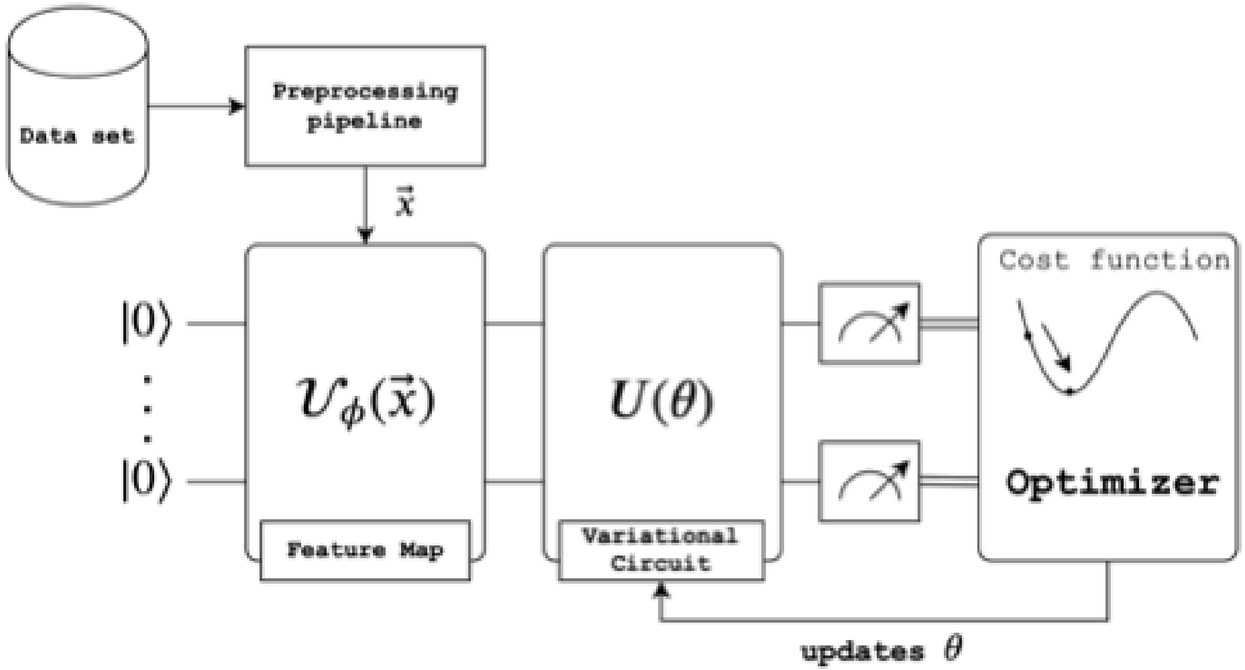
The most common use of the term refers to machine learning algorithms for the analysis of classical data executed on a quantum computer, i.e. **quantum-enhanced machine learning**.

VARIATIONAL QUANTUM CLASSIFIERS



A four-step process

1. Feature Map
2. Variational Circuit
3. Measurement & Label Assignment
4. Classical Optimization Loop



Feature Map

A quantum feature map $\Phi(\vec{x})$ takes classical feature vector $\vec{x}$ to $|\Phi(\vec{x})\rangle\langle\Phi(\vec{x})|$, a vector in Hilbert space

Examples

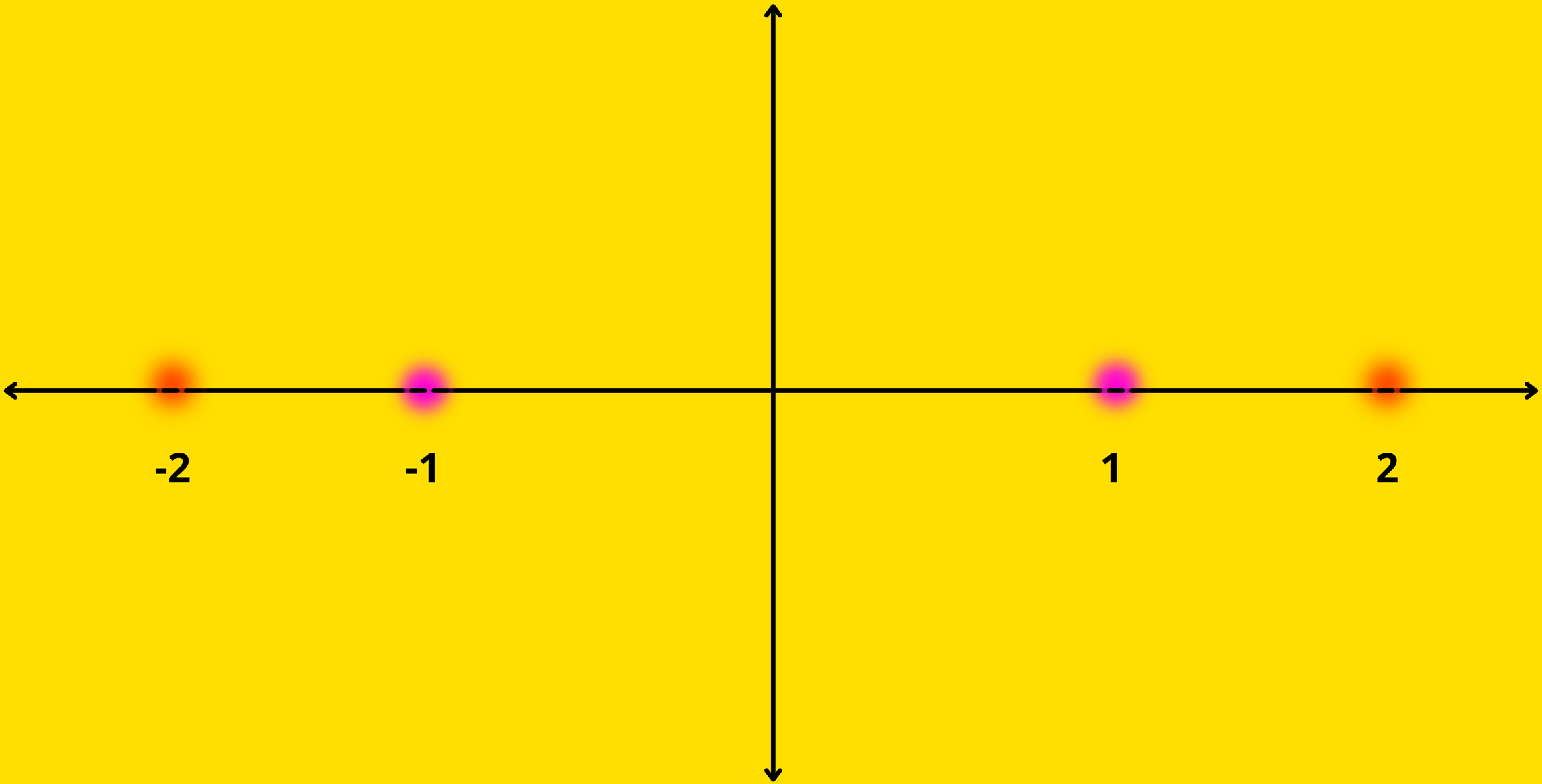
ZFeatureMap

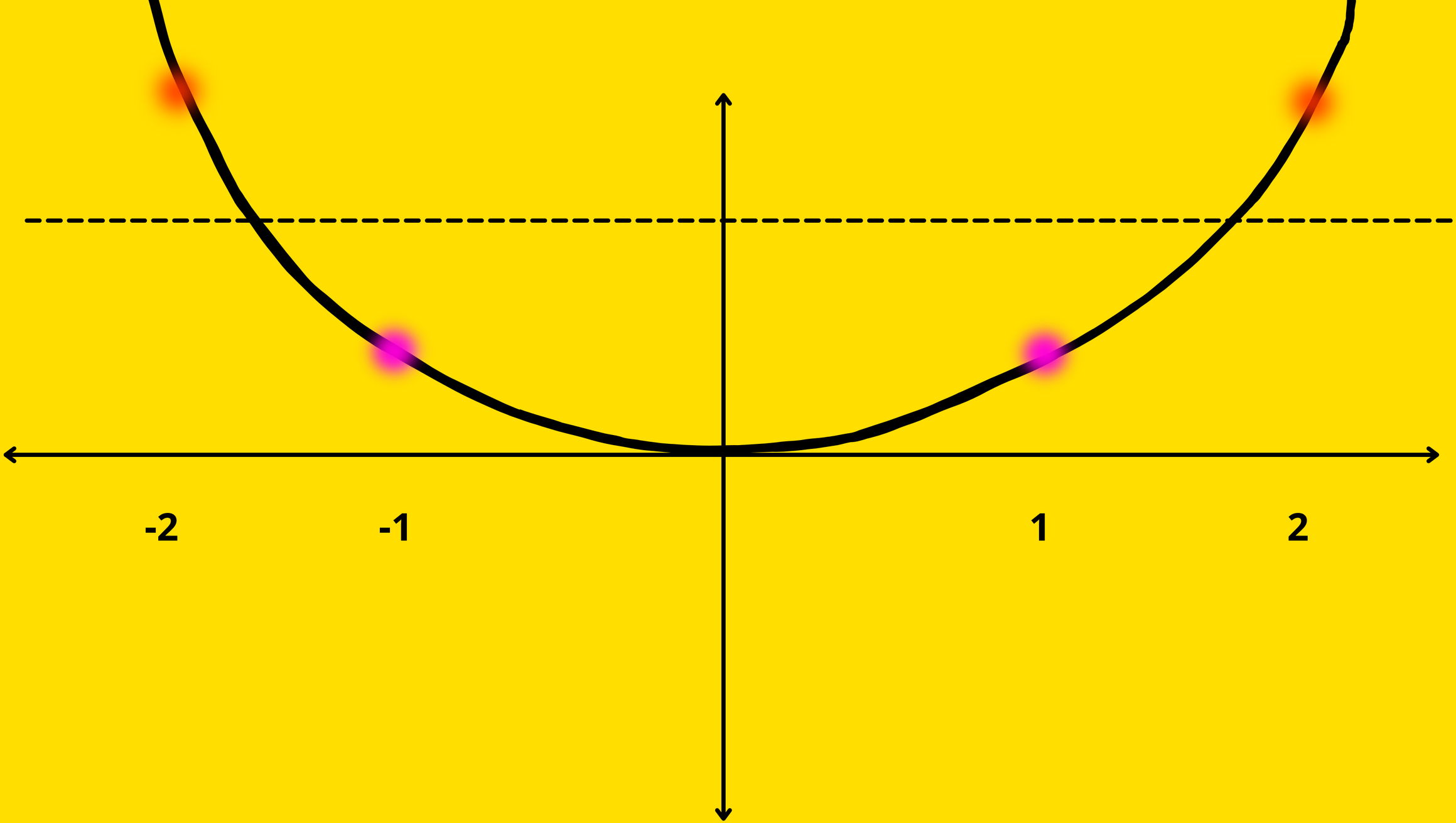
ZZFeatureMap

PauliFeatureMap

Underneath the hood

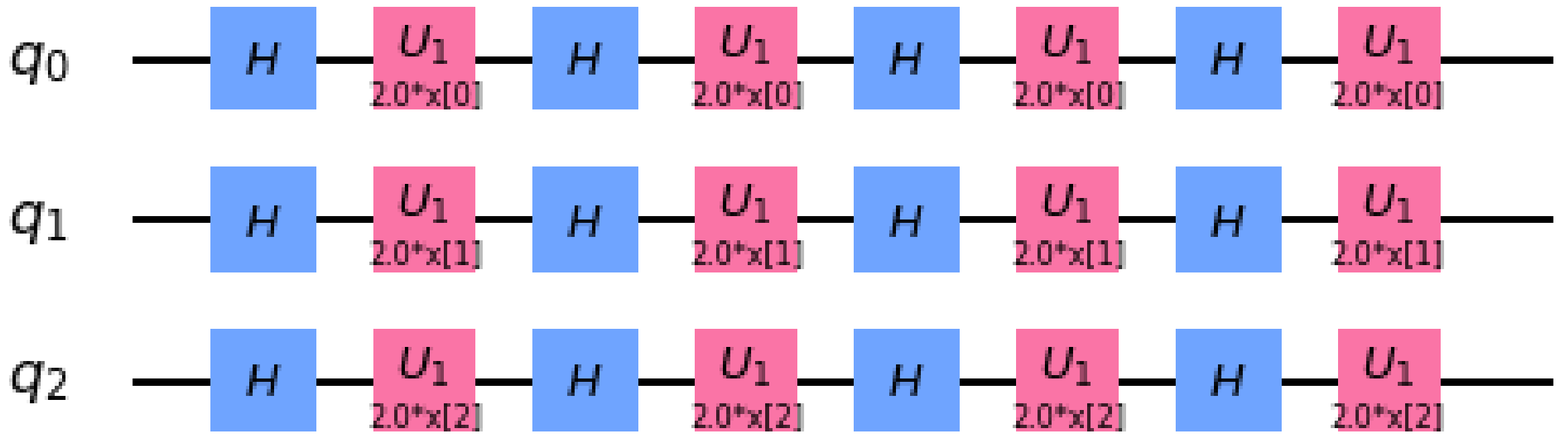
It maps the dataset non-linearly onto a higher dimension where a hyperplane can be found to classify it.


-2
-1
1
2


-2
-1
1
2

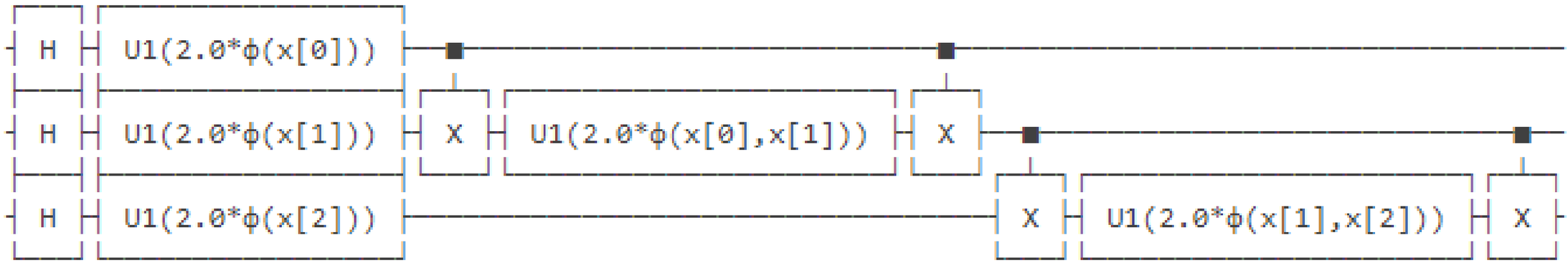
ZFeatureMap

A first order diagonal expansion is implemented using the ZFeatureMap feature map. The resulting circuit contains no interactions between features of the encoded data, and therefore no entanglement.



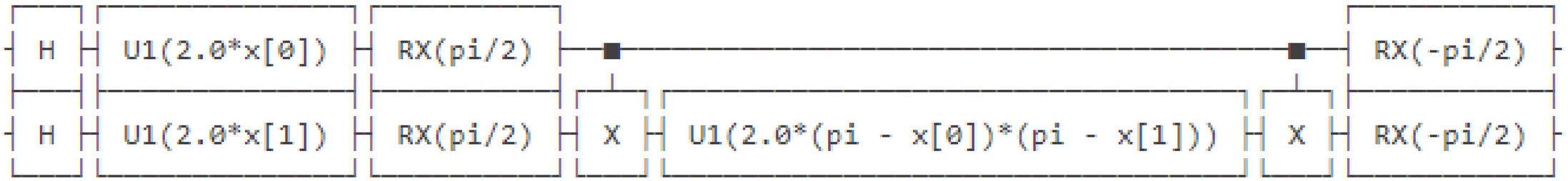
ZZFeatureMap

The ZZFeatureMap allows interactions in the data to be encoded in the feature map according to the connectivity graph and the classical data map and takes an additional input of entaglement .



PauliFeatureMap

This feature map has the same parameters as ZFeatureMap and ZZFeatureMap such as reps and data_map_func along with an additional paulis parameter to change the gate set.



Variational Circuit

This is the learning stage of the algorithm. A parameterized unitary operator U(w) is created such that $|\psi(x; \theta)\rangle = U(w)|\psi(x)\rangle$

Example Classes

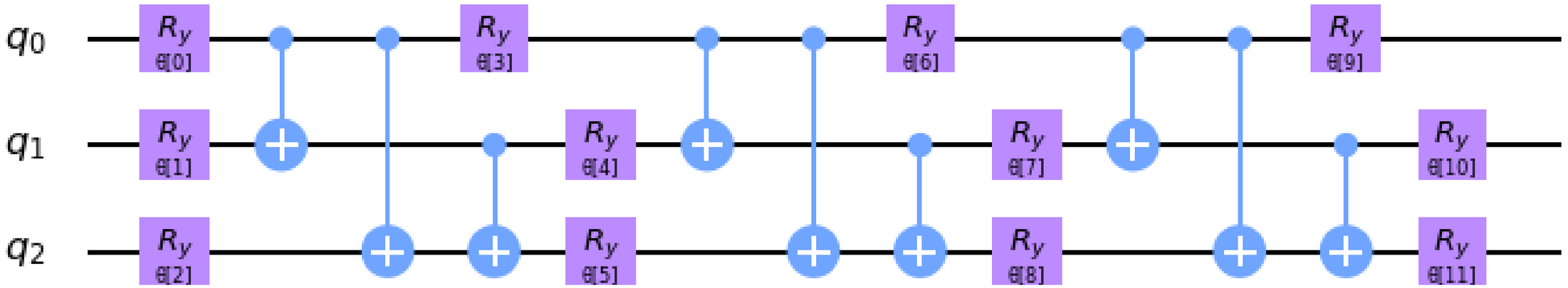
RealAmplitudes

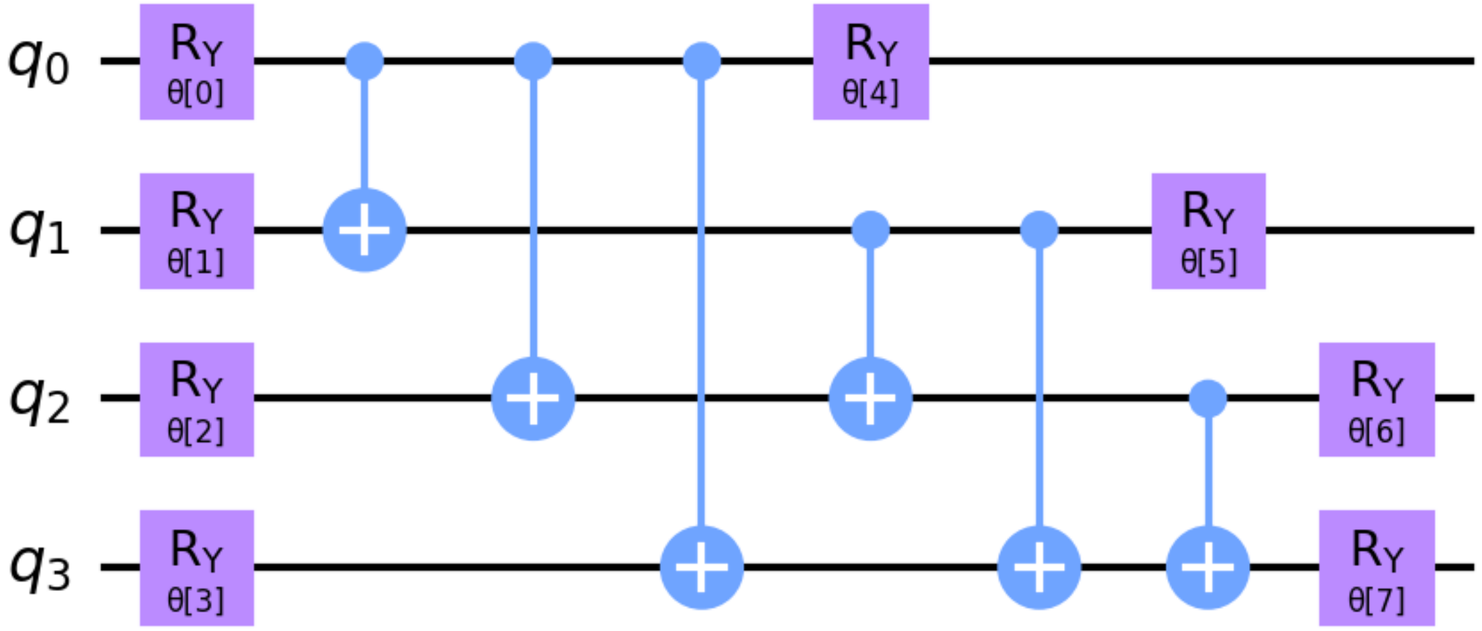
EfficientSU2

NLocal

Important point

It is essential to choose a variational circuit of shorter depth (making it especially viable for implementations on real hardware), lesser number of parameters to train (faster training process)


q_0
q_1
q_2
R_y θ[0]
R_y θ[1]
R_y θ[2]
R_y θ[3]
R_y θ[4]
R_y θ[5]
R_y θ[6]
R_y θ[7]
R_y θ[8]
R_y θ[9]
R_y θ[10]
R_y θ[11]


q_0
q_1
q_2
q_3
RY θ[0]
RY θ[1]
RY θ[2]
RY θ[3]
RY θ[4]
RY θ[5]
RY θ[6]
RY θ[7]

Measurement & Assignment

This step estimates the probability of belonging to a class by performing certain measurements. It’s the equivalent of sampling multiple times and obtaining an expectation value.

Examples

Parity function

Choosing the k-th digit

Important point

We get an n-bit classification string on measurement which is assigned values with a boolean function $f : \{0, 1\}^n \rightarrow 0, 1$

Classical Optimization Loop

This step aims to minimize the cost function classically.

Examples

COBYLA

SPSA

SLSQP

ADAM

Important point

The parameters of the quantum variational circuit are updated using a classical optimization routine once the measurements are ready

MNIST Classification with VQC

Dimensionality Reduction

```
from sklearn.decomposition import TruncatedSVD
from sklearn.manifold import TSNE

# Using SVD to bring down the dimension to 10
tsvd = TruncatedSVD(n_components=10)
X_SVD = tsvd.fit_transform(train_data_features)

# Further using t-SNE to bring the dimension down to 2
np.random.seed(0)
tsne = TSNE(n_components=2)
train_data_features_reduced = tsne.fit_transform(X_SVD)
```

```
print("Dimension of the reduced dataset: {}".format(train_data_features_reduced.shape[1]))
```

```
Dimension of the reduced dataset: 2
```

MNIST Classification with VQC

Extraction and normalisation

```
zero_datapoints = []
one_datapoints = []
for i in range(10000):
    if train_data_labels[i] == 0:                    # extracting zeros
        zero_datapoints.append(train_data_features_reduced[i])

for i in range(10000):
    if train_data_labels[i] == 1:                    # extracting ones
        one_datapoints.append(train_data_features_reduced[i])

zero_datapoints = np.array(zero_datapoints)
one_datapoints = np.array(one_datapoints)

def normalize(arr, max_val, n):
    a = np.divide(arr, max_val)
    return a + n

zero_datapoints_normalized = normalize(zero_datapoints, 100, 1)
one_datapoints_normalized = normalize(one_datapoints, 100, 1)
```

MNIST Classification with VQC

Defining training and testing datapoints

```
train_size = 20
test_size = 10
dp_size_zero = 5
dp_size_one = 5

zero_train = zero_datapoints_normalized[:train_size]
one_train = one_datapoints_normalized[:train_size]

zero_test = zero_datapoints_normalized[train_size + 1:train_size + test_size + 1]
one_test = one_datapoints_normalized[train_size + 1:train_size + test_size + 1]

training_input = {'A':zero_train, 'B':one_train}
test_input = {'A':zero_test, 'B':one_test}

# datapoints is our validation set
datapoints = []
dp_zero = zero_datapoints_normalized[train_size + test_size + 2:train_size + test_size + 2 + dp_size_zero]
dp_one = one_datapoints_normalized[train_size + test_size + 2:train_size + test_size + 2 + dp_size_one]
datapoints.append(np.concatenate((dp_zero, dp_one)))
dp_y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
datapoints.append(dp_y)

class_to_label = {'A': 0, 'B': 1}
```

MNIST Classification with VQC

Instantiate FeatureMap, Variational Circuit and Classical Optimizer

Instantiate the Feature map to use:

```
seed = 10598
feature_dim = zero_train.shape[1]

feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='linear')
feature_map.draw()
```

Instantiate the Classical Optimizer to use:

```
from qiskit.aqua.components.optimizers import COBYLA

cobyla = COBYLA(maxiter=500, tol=0.001)
```

Instantiate the variational Circuit to use:

```
from qiskit.circuit.library import EfficientSU2, RealAmplitudes

var = EfficientSU2(feature_dim, reps=2)
var.draw()
```

MNIST Classification with VQC

Running the classifier

```
# initilizing our backend
backend = BasicAer.get_backend('qasm_simulator')
backend_options = {"method": "statevector"}

# creating a quantum instance
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed, backend_options=backend_opt
ions)

#initilizing our VQC object
vqc = VQC(optimizer=cobyla, feature_map=feature_map, var_form=var, callback=call_back_vqc, training_dataset=training_input,
          test_dataset=test_input, datapoints=datapoints[0])
```

```
start = time.process_time()

result = vqc.run(quantum_instance)

print("time taken: ")
print(time.process_time() - start)


print("testing success ratio: {}".format(result['testing_accuracy']))
```

MNIST Classification with VQC

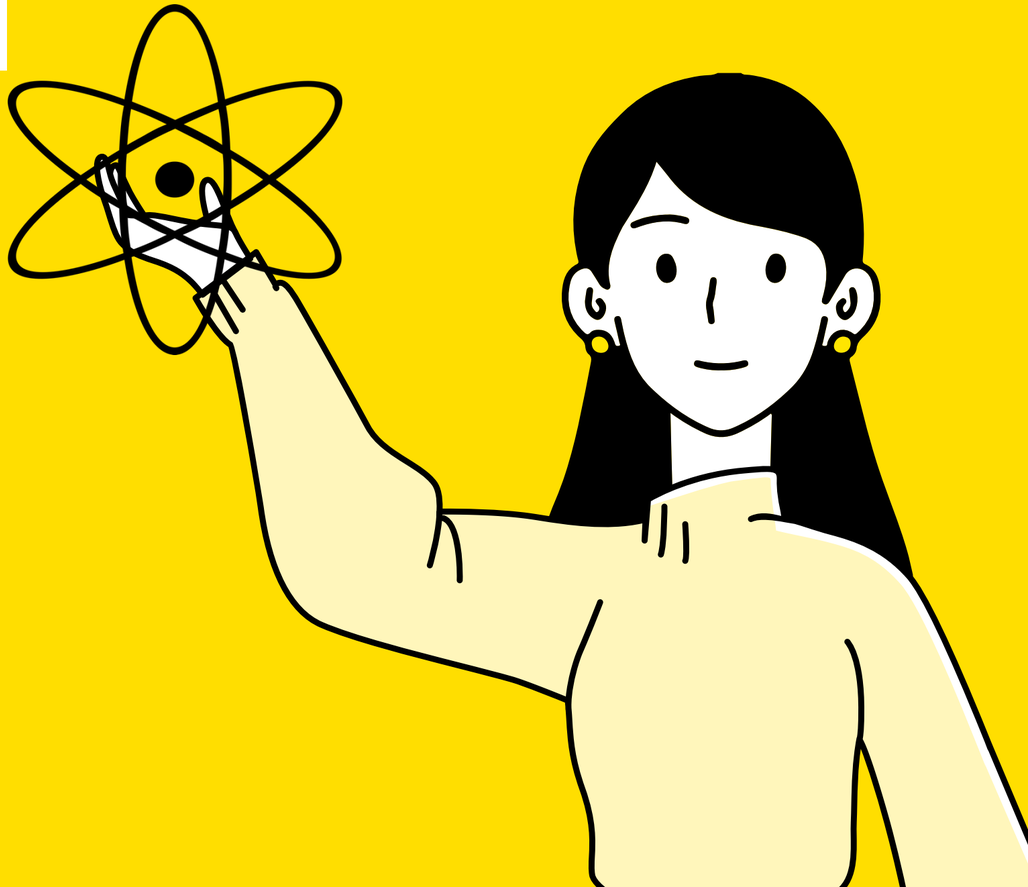
Results

```
time taken:
48.947100999999975
testing success ratio: 1.0
```

```
print("prediction of datapoints:")
print("ground truth: {}".format(map_label_to_class_name(datapoints[1], vqc.label_to_class)))
print("prediction:   {}".format(result['predicted_classes']))
```

```
preduction of datapoints:
ground truth: ['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B']
prediction:   ['A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B']
```

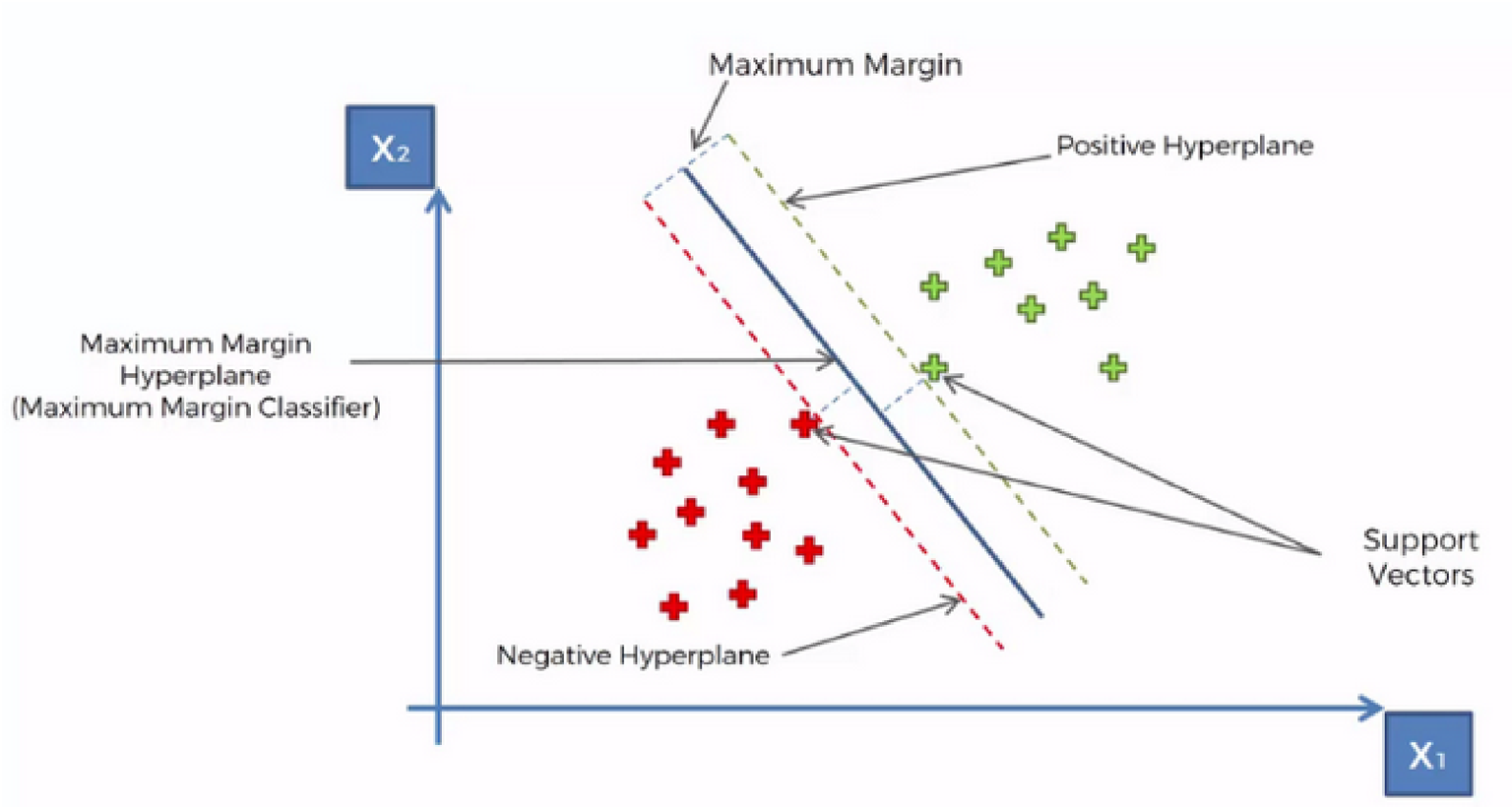
QUANTUM SUPPORT VECTOR MACHINES



Classical SVMs

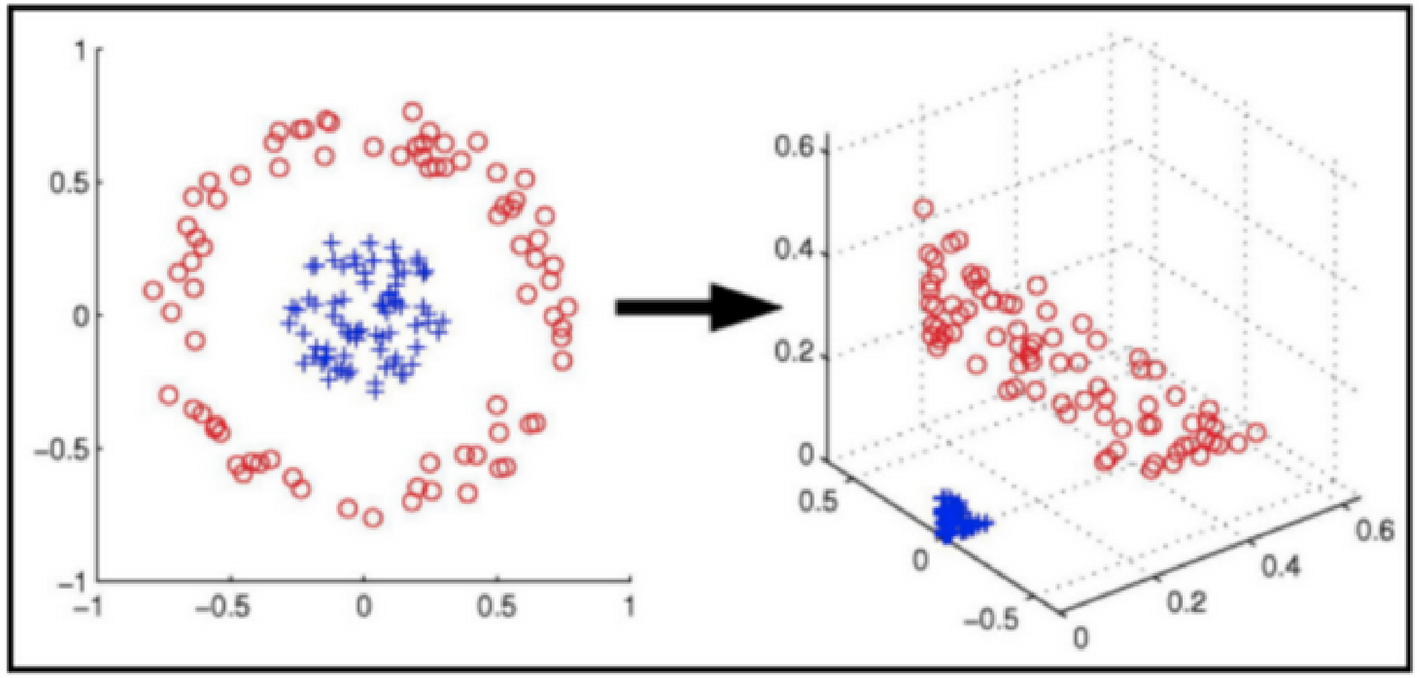
Hyperplanes

SVMs, when given labeled training data output a hyperplane that categorizes new samples. ex. Here in 2D data, it outputs a line that divides the plane into 2 parts, with classes on respective sides.



An example of a 2D case, the hyperplane clearly separates the green data points from the red data points.

The Kernel Trick



The Kernel Trick. The left side is the original scenario, and the right side is the result of the mapping after the kernel trick is applied.

Dot Products

1. Say the new space we want is:
   $z = x^2 + y^2$
2. Figure out what the dot product in that space looks like:

$a \cdot b = xa \cdot xb + ya \cdot yb + za \cdot zb$

$a \cdot b = xa \cdot xb + ya \cdot yb + (xa^2 + ya^2) \cdot (xb^2 + yb^2)$

The SVM can now use the new dot product, called a **kernel function** to calculate the hyperplane

The need for Quantum

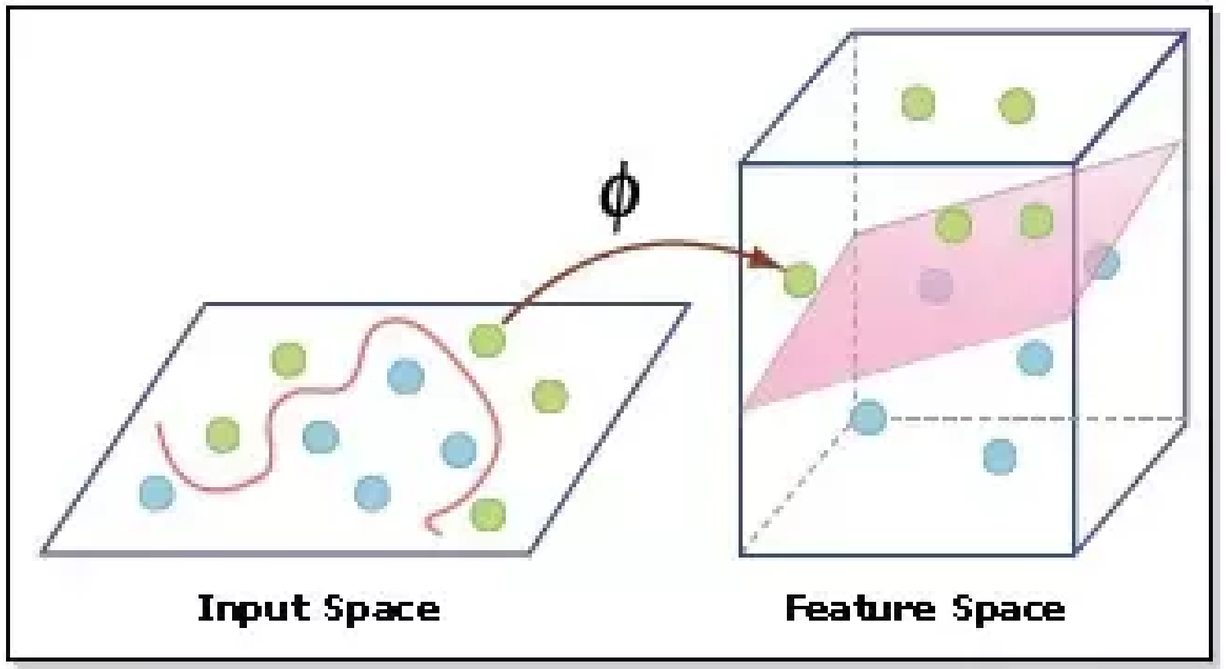


Image by MIT OpenCourseWare.

Higher Dimensions

As the number of dimensions increase, it becomes difficult for classical SVMs to calculate complex computations. Even if they were capable of doing so, they may take weeks to train, depending on the number of examples used, features, and regularization parameters.

Implementing a QSVM

1. Preprocessing (Scaling, normalization and principal component analysis)
2. Generation of Kernel matrix
3. Estimation of the kernel for new set of data points (test dataset) for QSVM classification

How does the Quantum algorithm actually work?

First, we create superposition using the quantum circuit. After manipulating and encoding the information using superposition we apply interference on the superpositioned states to get the final result.

Least squares reformulation of the SVM

We change the quadratic programming problem of SVM into a problem solving the linear equation system:

where,

$$F\begin{pmatrix} b \\ \vec{\alpha} \end{pmatrix} \equiv \begin{pmatrix} 0 & \vec{1}^T \\ \vec{1} & K + \gamma^{-1} I \end{pmatrix} \begin{pmatrix} b \\ \vec{\alpha} \end{pmatrix} = \begin{pmatrix} 0 \\ \vec{y} \end{pmatrix}$$

K is m×m kernel matrix and its elements can be calculated by

$$K = K\left(x_j\ ,\ x_k\right) = \ \phi\left(x_j\right).\phi\left(x_k\right)$$

γ is a user-defined value to control the trade-off between training error and SVM objective, **y** is a vector storing labels of training data.

The only unknown is the equation is the vector: $\begin{pmatrix} b \\ \vec{\alpha} \end{pmatrix}$

Training the Quantum model

Once the Kernel matrix is calculated, and the parameters of the hyperplane determined, a new datapoint x can be classified as:

$$y(x_0) = sgn\left(\sum_{i=0}^{m} \alpha_i K(x_i\ ,\ x_0) + b\right)$$

where,

vector **x** with i ∈ {1,2,....m} is the training data,

and the i-th dimension of the parameter vector **a :** ai

the sign function can be defined as:

$$sgn(x) = \begin{cases} 1, \text{ if } x \geqslant 0 \\ -1, \text{ if } x < 0 \end{cases}$$

Parameters:

- feature_dimension: number of features,
- depth: the number of repeated circuits,
- entangler_map: describe the connectivity of qubits [source, target],
- entanglement: generate the qubit connectivity {‘full’- entangles each qubit with all the subsequent ones and ‘linear’ -entangles each qubit with the next}
- feature_map: feature map module to transform the data to feature space,
- Datapoints: prediction dataset,
- quantum_instance: quantum backend with all execution settings,
- shots: number of repetitions of each circuit,
- seed_simulator: random seed for simulators,
- seed_transpiler: the random seed for circuit mapper
- QSVM: Quantum SVM method that will run the classification algorithm (binary or multiclass)

Algorithmic Complexity Comparision

The SVM can be formulated as a quadratic programming problem, which can be solved in time proportional to $O(\log(\varepsilon^{-1})\text{poly}(N, M))$, with N the dimension of the feature space, M the number of training vectors, and ε the accuracy.

Quantum support vector machine can be implemented with $O(\log NM)$ run time in both training and classification stages using the HLL algorithm in situations where classically O(M) training examples and O(N) samples for the inner product are required. The performance in N arises due to a fast quantum evaluation of inner products. For the performance in M, we can reexpress the SVM as an approximate least-squares problem that allows for a quantum solution with the matrix inversion algorithm.

ref.
https://dspace.mit.edu/bitstream/handle/1721.1/90391/PhysRevLet

QSVM using qiskit

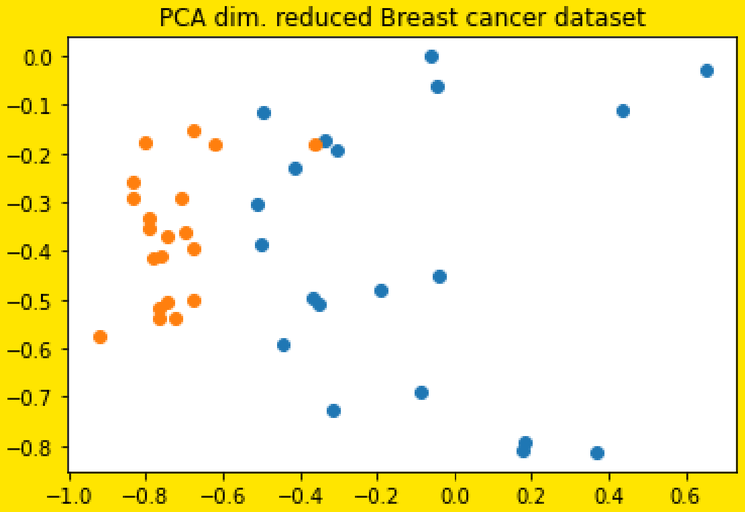
1. Here an example of classification problem is provided for which computation of kernel is not efficient
2. This means required resources are too less infront of the size of the problem
3. QSVM uses a Quantum processor to solve this problem by a direct estimation of the kernel in the feature space

```
import matplotlib.pyplot as plt
import numpy as np

from qiskit import BasicAer
from qiskit.circuit.library import ZZFeatureMap
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.aqua.algorithms import QSVM
from qiskit.aqua.utils import split_dataset_to_data_and_labels, map_label_to_class_name
```

Real-world Dataset

Now we run our algorithm with a real-world dataset: the breast cancer dataset, we use the first two principal components as features

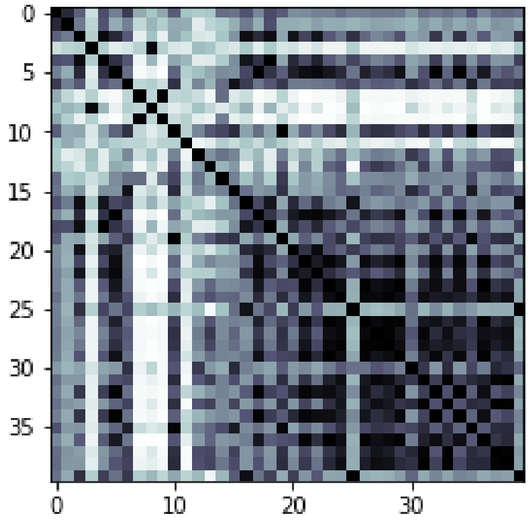
```
from qiskit.ml.datasets import breast_cancer

feature_dim = 2
sample_total, training_input, test_input, class_labels = breast_cancer(
    training_size=20,
    test_size=10,
    n=feature_dim,
    plot_data=True
)
```



The Kernel Matrix

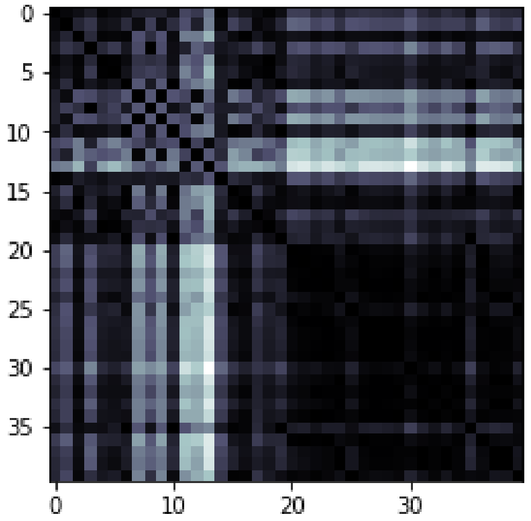
The following shows the kernel matrix that was built from the training sample of the dataset.

```
kernel_matrix = result['kernel_matrix_training']
img = plt.imshow(np.asmatrix(kernel_matrix),interpolation='nearest',origin='upper',cmap='bone_r')
```



Quantum SVM Kernel Matrix

Success ratio: 0.9



SVM Kernel Matrix

Success ratio: 0.85

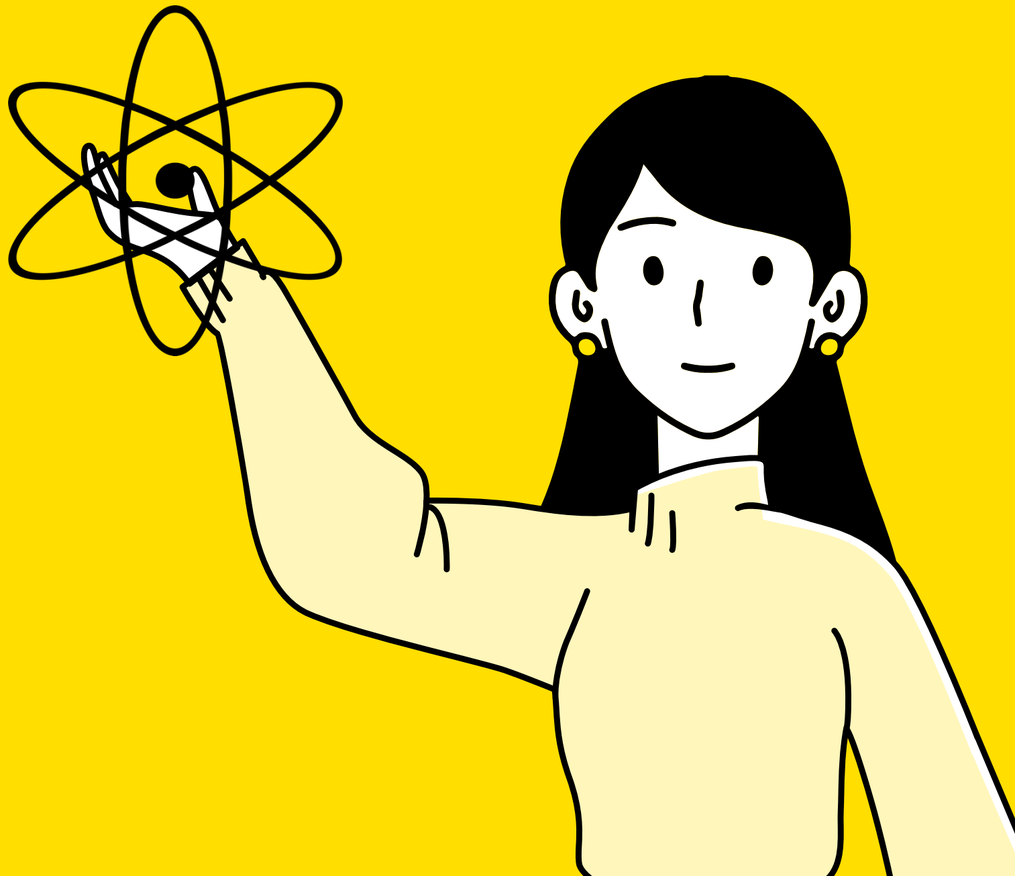
Testing Success Ratio

For the testing, the result includes the details and the success ratio.
Here new data without labels is classified according to the solution found in the training phase

```
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='linear')
qsvm = QSVM(feature_map, training_input, test_input)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)

result = qsvm.run(quantum_instance)

print(f'Testing success ratio: {result["testing_accuracy"]}')
```

Testing success ratio: 0.9

QUANTUM NEURAL NETWORKS



Neural Networks

The components

Neurons or nodes

Perceptrons and transformations

Cost Function

Forward Propagation

Backward Propagation

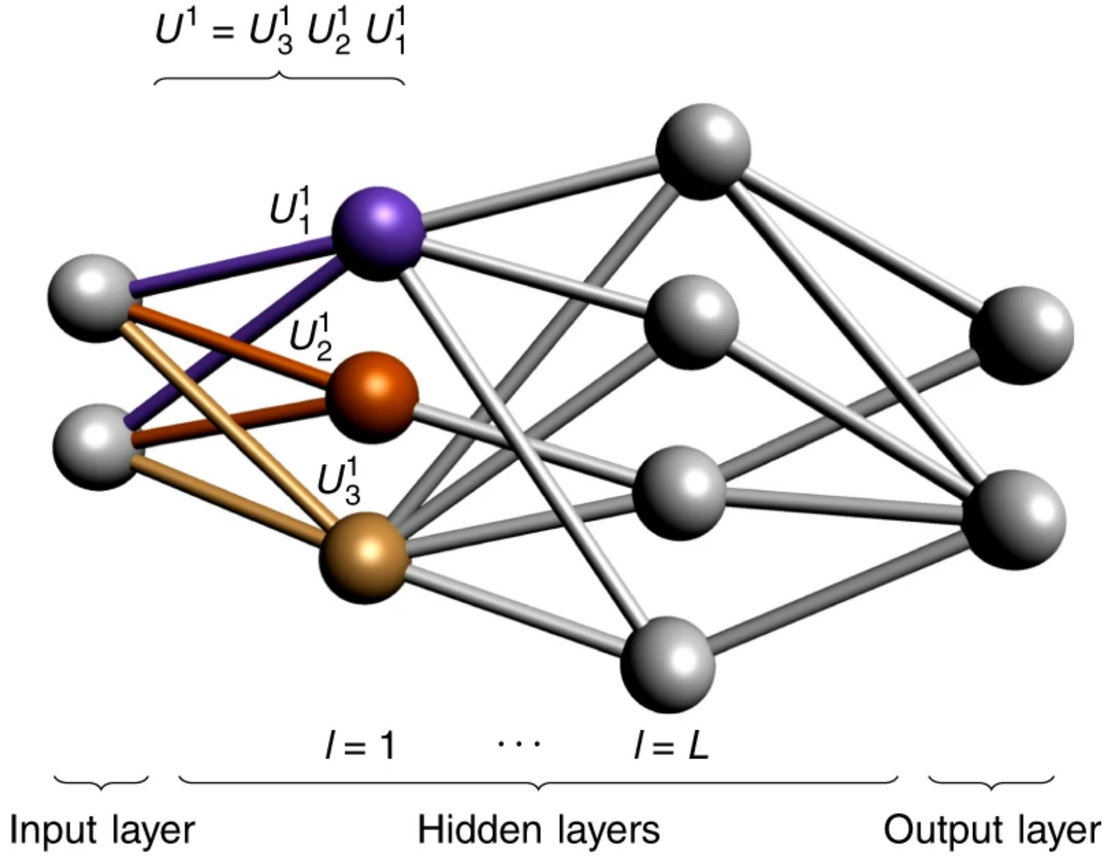
Various approaches

Perceptron Approach

Associative Memory Approach

Perceptron Approach

- Input encoding
- Maps from 1 layer of qubits to next
- No Cloning Theorem
- 'Fan out' output to the next layer
- Usage of 'Ancilla bits'
- Layer of Perceptrons per network layer
- Fidelity based cost function



Memory Associative Approach

- Many universes (many-world) interpretation of Everett
- Multimodular approach and Hopfield networks
- Superposition and parallelism
- Multilayer perceptrons
- Memorization and recall
- Recall - Grover Search Algorithm

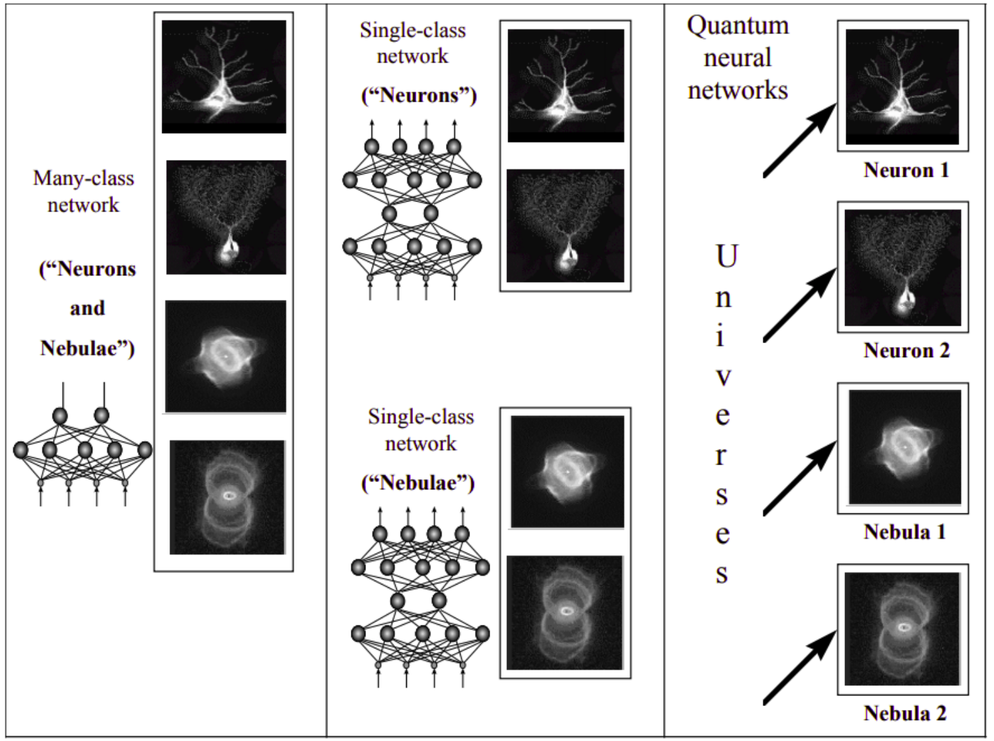


Fig. 1. Many-class networks are trained using the examples from different classes (here "Neurons" and "Nebulae" together) – left; A set of modular single-class neural networks use for training only the objects belonging to one class (two networks for two classes: "Neurons" and "Nebulae" separately) – center; Quantum neural networks may be trained using only pattern each! (four networks for four examples in many universes) – right.

Advantages

- General
  - Role of quantum in the human brain
  - Could solve classically intractable problems
- Perceptron Approach
  - Lesser memory requirements
  - Lack of cost function plateau
- Associative Memory Approach
  - Exponential memory

Encoding

```
THRESHOLD = 0.5

x_train_bin = np.array(x_train_nocon > THRESHOLD, dtype=np.float32)
x_test_bin = np.array(x_test_small > THRESHOLD, dtype=np.float32)

def convert_to_circuit(image):
    """Encode truncated classical image into quantum datapoint."""
    values = np.ndarray.flatten(image)
    qubits = cirq.GridQubit.rect(4, 4)
    circuit = cirq.Circuit()
    for i, value in enumerate(values):
        if value:
            circuit.append(cirq.X(qubits[i]))
    return circuit



x_train_circ = [convert_to_circuit(x) for x in x_train_bin]
x_test_circ = [convert_to_circuit(x) for x in x_test_bin]
```

Main Circuit

```
class CircuitLayerBuilder():
    def __init__(self, data_qubits, readout):
        self.data_qubits = data_qubits
        self.readout = readout

    def add_layer(self, circuit, gate, prefix):
        for i, qubit in enumerate(self.data_qubits):
            symbol = sympy.Symbol(prefix + '-' + str(i))
            circuit.append(gate(qubit, self.readout)**symbol)
```

Main Circuit

```
def create_quantum_model():
    """Create a QNN model circuit and readout operation to go along with it."""
    data_qubits = cirq.GridQubit.rect(4, 4)  # a 4x4 grid.
    readout = cirq.GridQubit(-1, -1)         # a single qubit at [-1,-1]
    circuit = cirq.Circuit()

    # Prepare the readout qubit.
    circuit.append(cirq.X(readout))
    circuit.append(cirq.H(readout))

    builder = CircuitLayerBuilder(
        data_qubits = data_qubits,
        readout=readout)

    # Then add layers (experiment by adding more).
    builder.add_layer(circuit, cirq.XX, "xx1")
    builder.add_layer(circuit, cirq.ZZ, "zz1")

    # Finally, prepare the readout qubit.
    circuit.append(cirq.H(readout))

    return circuit, cirq.Z(readout)
```

Training Procedure

```
model = tf.keras.Sequential([
    # The input is the data-circuit, encoded as a tf.string
    tf.keras.layers.Input(shape=(), dtype=tf.string),
    # The PQC layer returns the expected value of the readout gate, range [-1,1].
    tfq.layers.PQC(model_circuit, model_readout),
])

y_train_hinge = 2.0*y_train_nocon-1.0
y_test_hinge = 2.0*y_test-1.0

def hinge_accuracy(y_true, y_pred):
    y_true = tf.squeeze(y_true) > 0.0
    y_pred = tf.squeeze(y_pred) > 0.0
    result = tf.cast(y_true == y_pred, tf.float32)

    return tf.reduce_mean(result)

model.compile(
    loss=tf.keras.losses.Hinge(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=[hinge_accuracy])
```

Model Training

EPOCHS = 3

BATCH_SIZE = 32

NUM_EXAMPLES = len(x_train_tfcirc)

x_train_tfcirc_sub = x_train_tfcirc[:NUM_EXAMPLES]

y_train_hinge_sub = y_train_hinge[:NUM_EXAMPLES]

Training this model to convergence should achieve >85% accuracy on the test set.

```
qnn_history = model.fit(
      x_train_tfcirc_sub, y_train_hinge_sub,
      batch_size=32,
      epochs=EPOCHS,
      verbose=1,
      validation_data=(x_test_tfcirc, y_test_hinge))

qnn_results = model.evaluate(x_test_tfcirc, y_test)
```

References

1. https://en.wikipedia.org/wiki/Quantum_machine_learning
2. https://iiitaphyd.sharepoint.com/sites/msteams_7bc406/Shared%20Documents/TA%20--%20Aditya%20Morolia/quantum-machine-learning.pdf
3. https://born-2learn.github.io/posts/2020/12/variational-quantum-classifier/
4. https://medium.com/quantumcomputingindia/quantum-machine-learning-102-qsvm-using-qiskit-731956231a54
5. https://medium.com/swlh/qa2-explaining-variational-quantum-classifiers-b584c3bd7849
6. https://github.com/Qiskit-Challenge-India/2020/blob/master/Day%206%2C%207%2C8/VQC_notebook.ipynb
7. https://www.qmunity.tech/tutorials/building-a-variational-quantum-classifier
8. https://pennylane.ai/qml/demos/tutorial_variational_classifier.html
9. https://arxiv.org/pdf/1803.00745.pdf
10. https://qiskit.org/documentation/tutorials.html

References

1. https://github.com/Qiskit/qiskit-tutorials/tree/master/tutorials/machine_learning
2. https://github.com/qiskit-community/qiskit-community-tutorials/tree/master/machine_learning
3. https://qiskit.org/documentation/machine-learning/
4. https://github.com/Qiskit/qiskit-machine-learning/tree/master/docs/tutorials
5. https://www.nature.com/articles/s41467-020-14454-2
6. https://en.wikipedia.org/wiki/Quantum_neural_network#:~:text=Quantum%20neural%20networks%20are%20computational,the%20principles%20of%20quantum%20mechanics.&text=The%20hope%20is%20that%20features,can%20be%20used%20as%20resources.
7. https://axon.cs.byu.edu/papers/ezhov.fdisis00.pdf
8. https://arxiv.org/pdf/1804.07633.pdf
9. https://esobimpe.medium.com/quantum-neural-networks-9fce2566315d
10. https://qiskit.org/textbook/ch-machine-learning/machine-learning-qiskit-pytorch.html
11. https://arxiv.org/pdf/1802.06002.pdf
12. https://www.tensorflow.org/quantum/tutorials/mnist

References

1. https://www.sigmoid.com/blogs/quantum-computing-blog-3-how-to-implement-qsvm-in-the-ibm-q-environment/
2. https://towardsdatascience.com/support-vector-machines-for-classification-fc7c1565e3
3. https://arxiv.org/pdf/1803.07128.pdf
4. https://dspace.mit.edu/bitstream/handle/1721.1/90391/PhysRevLett.113.130503.pdf?sequence=1&isAllowed=y
5. https://arxiv.org/pdf/1909.11988.pdf
6. https://quantumcomputinguk.org/tutorials/how-to-implement-qsvm-algorithm-on-ibms-quantum-computers-with-qiskit
7. https://www.ijitee.org/wp-content/uploads/papers/v8i6/F3441048619.pdf
8. https://www.youtube.com/watch?v=OKbcJCUx6xA&t=583s
9. https://docs.microsoft.com/en-us/azure/quantum/user-guide/libraries/machine-learning/basic-classification?tabs=tabid-python
10. https://github.com/Qiskit/qiskit-tutorials/tree/master/tutorials/machine_learning

Thank you!