QUANTUM MACHINE LEARNING

Qiskit Fall Festival submission. November 4th, 2022

EL OUARDI Abdelghani

University Mohammed V

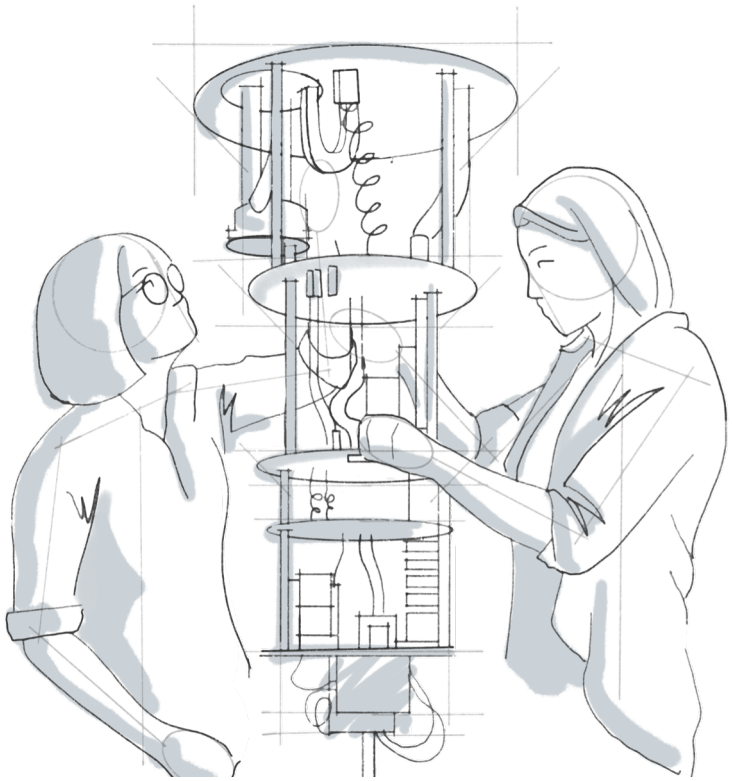


Table of Contents

1. Classical Machine Learning (HandWritten digits Recognition) 2

1.1. Neural Network: a general structure 2

1.2. The Gradient descent and the back-propagation 4

1.3. Implementing the handwritten Digits recognition neural network 6

2. Quantum Machine Learning (HandWritten Binary Digits Recognition) 9

2.1. Quantum Machine Learning: a general structure 9

2.2. Data encoding 10

2.2.1. Basis encoding 10

2.2.2. Amplitude encoding 11

2.2.3. angle encoding 11

2.3. Implementation 12

Quantum feature map 13

Quantum Kernel Estimation 16

3. Analog Machines: A new hope? 19

3.1. History 19

3.2. Machine Learning and Analog Computing 20

3.3. Summary: where does quantum finds its place 22

1. Classical Machine Learning (Handwritten digits Recognition)

1.1. Neural Network: a general structure

The handwritten digits recognition is somewhat of a classical example for introducing the topic of machine learning, or what some may call it the **"hello world"** of machine learning.

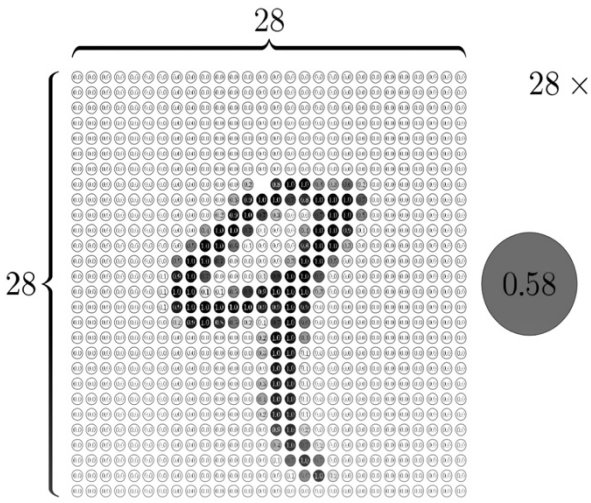
As the name suggest, Neural networks are inspired by the brain, but in what sense? what exactly do we mean by neurons? And in what way they are connected?

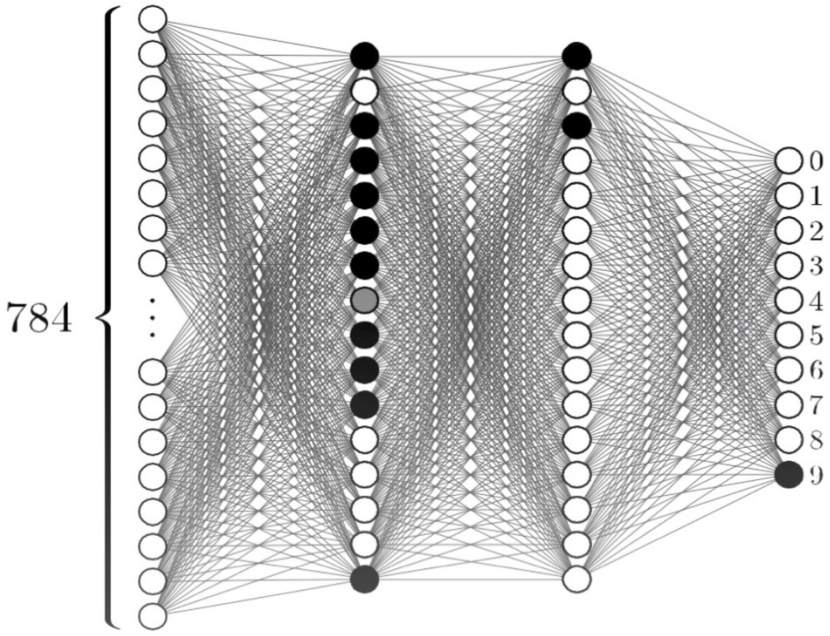
When we talk about neuron, what we should imagine is an object that hold a number, or what's called an activation number, that is between 0 and 1.

For example, the network starts with a bunch of neurons corresponding to each of the 28 X 28 pixels of the input image, each one of these holds a number, ranging from 0 for white pixels, up to one for black pixels.

These 784 neurons make up our first layer of our network.

And what we hope for is that the last layer (which will have 10 neurons corresponding to the 10 digits) will be activated in a way that correspond to the right digit. In our example that will be 9.



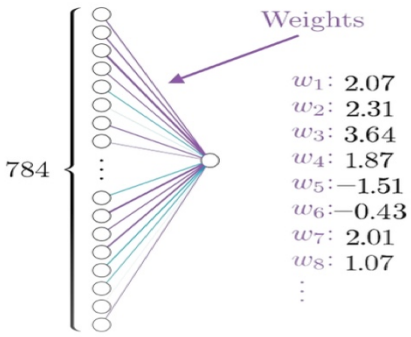


There are also a couple layers in between, called the hidden layers.

The way this network works is that the activation in one layer determine the activation in the next layer. And of course, the network as an information processing mechanism, comes down to exactly how those activation from one layer bring about activations in the next layer.

The sole purpose of this network is that when we feed it an image, the pattern at which those 784 neurons are activated, will determine how the neurons in the second layer are activated, and the pattern formed in the second layer will determine how the neurons in the third layer are activated, and finally how the

neurons in the output layer will be activated. The neuron activated in the output layer, represent the network choice for what digit this image represent.



But how exactly the activation in one layer determines the activation in the next layer. To explain this mechanism, we going to focus on one specific example at how the activation in the first layer influence the activation in one neuron in the second layer.

The way we do this is assigning a "weight" to the connection between the neurons of the first layer and the specific neuron in the second layer, these weights are just numbers.

And the activation in this neuron is just the weighted sum of the activation of the neurons in the first layer

$$w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n$$

But since the activation in a neuron Is restricted to the values between 0 and 1, what we will do is to pump this weighted sum into the sigmoid function, also known as the logistic curve

Which basically take the value 1 for verry positive input, and 0 for very negative input.

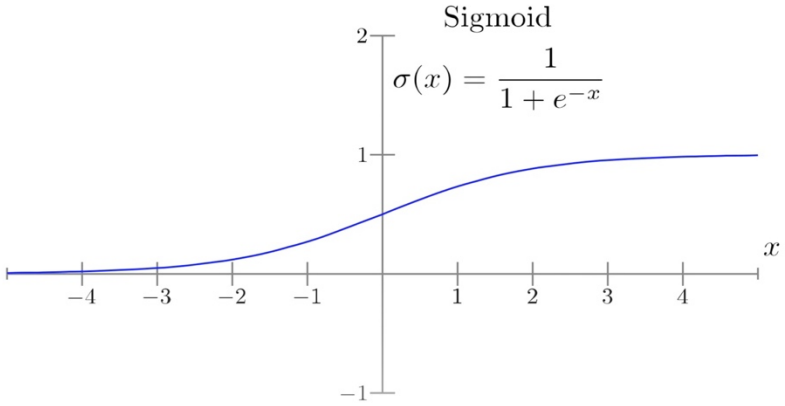
So, the activation in the neuron in the second layer is therefor

$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n)$$

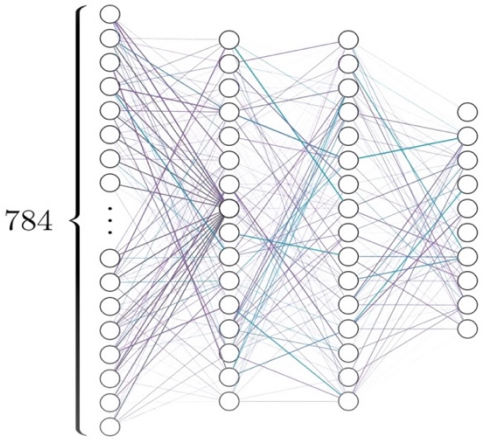
But we don't want the activation to equal to one just because the weighted sum is bigger than 0, but rather when it's bigger than a specific number b called the bias for inactivity. And so, we will plug this number into the weighted sum before pumping it into the sigmoid function

$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \cdots + w_na_n + b)$$

And this is just one neuron. All the other neurons will be activated according to their own weighted sum and their own bias.



And that's just the connection between the first layer and second layer, each of the connection between the other layers have their own weights and biases.



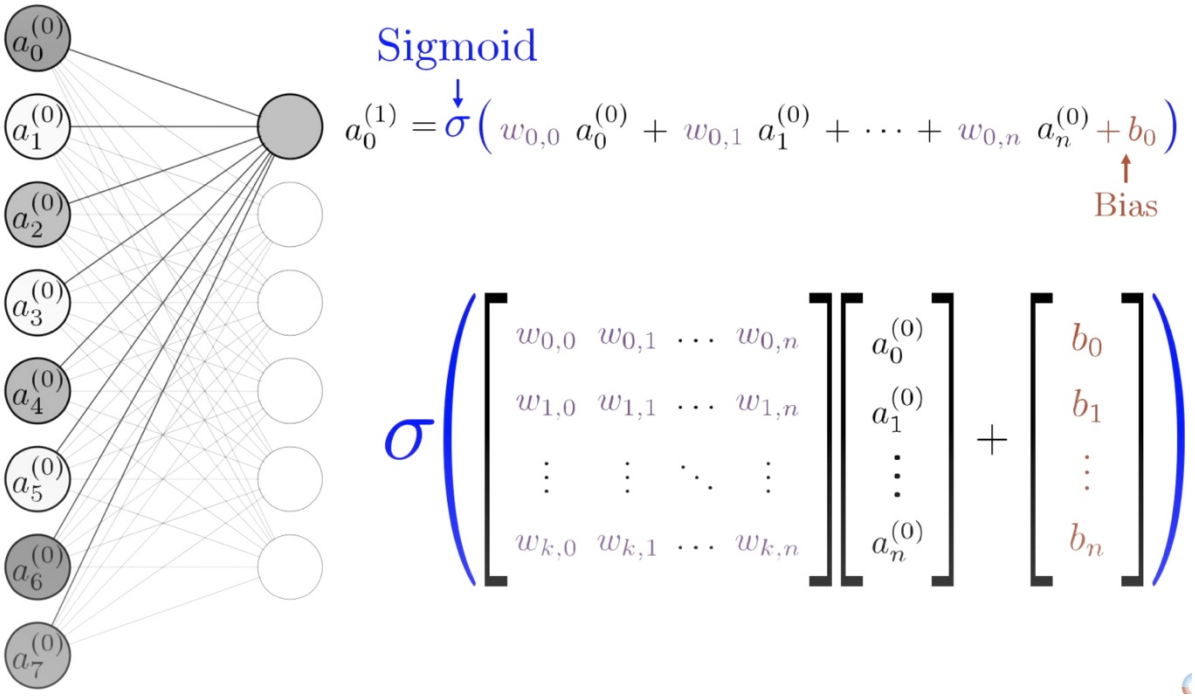
$784\times16 + 16\times16 + 16\times10$ weights

$16 + 16 + 10$ biases

13,002

In general, this weighted sum can be thought as a matrix transformation, where the set of activation in each layer is a vector that results from multiplying the previous layer by some matrix, then adding the bias

and then applying the sigmoid function



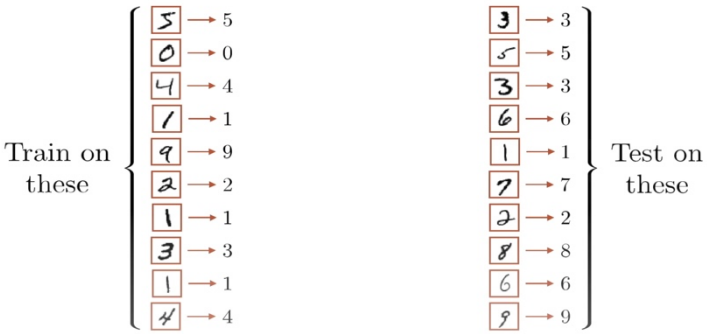
Which is summarized in the following expression

$$a_i^{(L)} = \sigma\left(\sum_j w_{ij}\, a_j^{(L-1)} + b_i\right)$$

But how is this weight values are determined? It's tempting to think that these values are pre-determined. But we won't have learning machine in that case. What actually happening is that these weights are constantly modified when exposing the machine to a training data set, the modification is achieved by a process called the **Back-propagation.**

1.2. The Gradient descent and the back-propagation

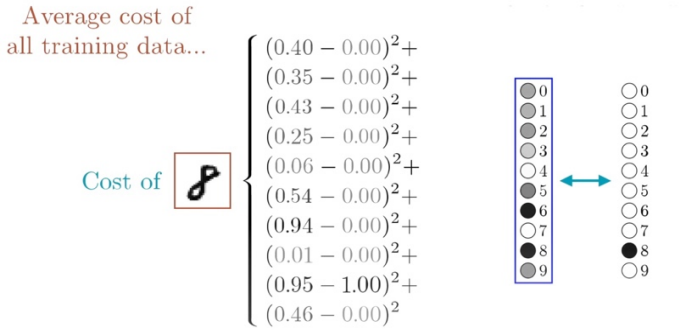
In order for the network to learn, we need an algorithm that feed this network a whole bunch of data which come in the form of a bunch of handwritten digits along with labels for what those digits supposed to be, and it will adjust those weight and biases accordingly, and hoping that the neural network will generalize what it learns to images beyond the training data.



And then see how accurately it classifies those images, luckily for us, the **MNIST** data base contain a collection of 10,000s of hand written digits images along with their label

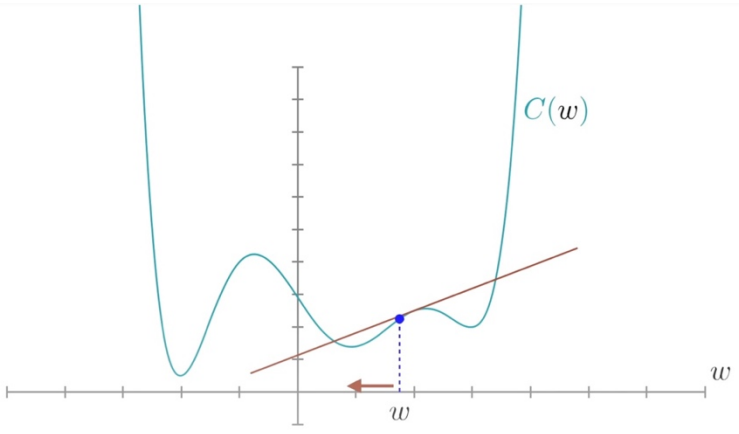
What we mean by learning, is a process at which the system tries to find the minimum of a certain function called the **cost function,** and the weights and biases an adjusted in a way that minimize this cost function. The cost function is a measure of the error i.e., how bad Is the machine at recognizing digits.

The cost function is defined as the sum of the difference between the activation value of the output layer and the corresponding label squared, of the training data. The following is an example of the cost function of the digit 8.



And then the problem of training the machine is reduced to finding the weights that minimize the cost function, we can do this by the process of the gradient descent, which Is just modifying the weight in a matter that lowers the cost function.

This graph is an over simplification for one weight value, while in our case, the handwritten digits recognition neural network contains more than 13



000 connections, each associated with its own weight value.

The algorithm of minimizing this function is to compute the gradient ∇C and then taking a step in the $-\nabla C$ direction, and repeat that over and over.

Where each element of $-\nabla C(\vec{w})$ tells each element of $\vec{w}$ if it should increase or decrease and by how much, in order to minimize the cost function.

$$\vec{\mathbf{W}} = \begin{bmatrix} 2.43 \\ -1.12 \\ 1.47 \\ \vdots \\ -0.76 \\ 3.50 \\ 2.03 \end{bmatrix} \qquad -\nabla C(\vec{\mathbf{W}}) = \begin{bmatrix} 0.18 \\ 0.45 \\ -0.51 \\ \vdots \\ 0.40 \\ -0.32 \\ 0.82 \end{bmatrix}$$

we can describe this procedure mathematically by

$$w_k \longmapsto w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \longmapsto b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

Where η is a small positive parameter known as the learning rate.

And by repeatedly applying this update, we will be able to roll down to the nearest local minimal, this is the process at which the neural network learns.

the above cost function used in the updating process is not that of one training example but rather the average cost function of all the training data

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x$$

Where. $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for individual training examples, the problem is the number of training examples is very large, so this will take a large time and the learning will occur very slowly.

The way we solve this is by using an idea called the stochastic gradient, the idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs.

We will label this training samples by $X_1, X_2, X_3, \ldots, X_m$, we will refer to these as mini batch. We expect these sample sizes to be large enough for the average value ∇C_{x_j} to be roughly equal to the average over all ∇C_x

$$\frac{1}{m} \sum_{j=1}^{m} \nabla C_{x_j} \approx \frac{1}{n} \sum_x \nabla C_x = \nabla C$$

And then we plug this into the updating process

$$w_k \longmapsto w'_k = w_k - \frac{\eta}{m} \sum_{j=1}^{m} \frac{\partial C_{x_j}}{\partial w_k}$$

$$b_l \longmapsto b'_l = b_l - \frac{\eta}{m} \sum_{j=1}^{m} \frac{\partial C_{x_j}}{\partial b_l}$$

1.3. Implementing the handwritten Digits recognition neural network

Apart from the **MNIST** data, we also going to need a python library called **NumPy**, for doing fast linear algebra.

First of all, we have to initiate the Network object

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

Where the size contains the number of neurons in the respective layers. So, if we want to create a Network with 784 neurons in the first layer, 16 neurons in the second and third layer and 10 neurons in the last layer, we just have to write

```
net = Network([784, 16, 16, 10])
```

While all the weights and biases are initiated randomly using `np. random.randn.`

The set of weights and biases that connect the first to the second layer are stored as a list of NumPy matrices. As mention before the activation in the one layer is a function of the activation in the previous layer

$$a' = \sigma(wa + b)$$

Where w is the matrix representation of the weights connect one layer to the other, and a' a and b are vectors.

The next step is to define the sigmoid function

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

We then add a feed forward method to the network class, which apply the equation $a' = \sigma(wa + b$ to each layer

```
def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

Now we need to implement the stochastic gradient descent in order for our neural network to learn

```
  def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)
```

The training data is a list of tuples (x,y) representing the training input and the corresponding desired outputs.

The code starts by randomly shuffling the training data, and then partitions it into mini batches, and then for each mini batch we apply a single step of gradient descent which is done by

```
self.update_mini_batch(mini_batch, eta)
```

which update the weights and biases according to a single iteration, the update_mini_bach is defined as follows

```
def update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

most of the work is done by

```
delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

which for every example in the mini_bach it updates self.weights and self.biases appropriately.

The back propagation function is defined

```
def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
```

This code is just an implementation of all mathematics we have seen before

We also going to have to define a couple of other functions that we will need

```
def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

def cost_derivative(self, output_activations, y):
        return (output_activations-y)
```

The evaluating function Returns the number of test inputs for which the neural network outputs the correct result. Where the cost_derivative Returns the vector of partial derivatives $\frac{\partial C_x}{\partial a}$ for the output activations.

And of course, the derivative of the sigmoid function

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

No to test the program we going to start by loading the MNIST data

```
import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

After loading the data, we will set up a network with 30 hidden layers

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

And finally, we will use the stochastic gradient descent to learn from the MNIST training data over 30 epochs, with a mini -batch size of 10, and a learning rate of $\eta = 3.0$,

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

here is a partial transcript of the output of one training run of the neural network. The transcript shows the number of test images correctly recognized by the neural network after each epoch of training. As you can see, after just a single epoch this has reached 9,129 out of 10,000, and the number continues to grow,

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

That is, the trained network gives us a classification rate of about 95 percent.

2. Quantum Machine Learning (Handwritten Binary Digits Recognition)

Quantum computer are naturally able to solve complex correlations between inputs that can be incredibly hard for traditional or classical computers This suggests that learning models made on quantum computers may be dramatically more powerful for selected applications, potentially performing faster computation, better generalization on less data, or both. Hence it is of great interest to understand in what situations such a "quantum advantage" might be achieved.

"With just 275 qubits, we can already represent more states than the number of atoms in the universe"

Carlton Caves

Quantum computers are with no doubt far more capable than Classical Computers, or at least when used in certain application, however, sometimes we are required to make a hybrid machine in order to witness the full benefits of quantum computing.

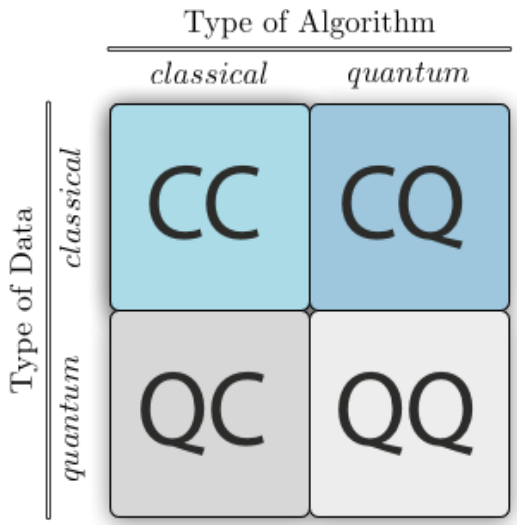
Can we see any these benefits in Quantum Machine Learning?

2.1. Quantum Machine Learning: a general structure

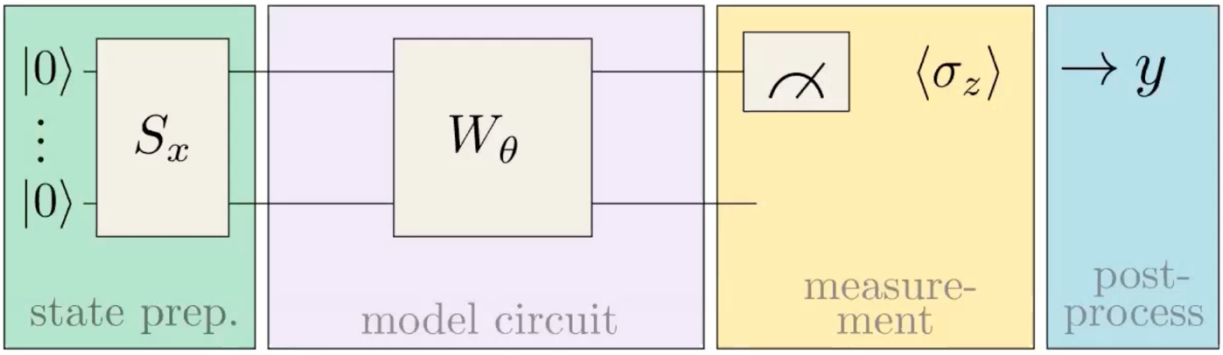
QML in general is casted into four paradigms, in each of these blocks we have an interplay of whether our data is Classical or Quantum, and whether the device or the computer that we using is Classical or Quantum

When we talk about QML in general we will be focusing on the idea of having classical data processed somehow or inter playing somehow with the quantum computer.

We will start by giving a rough frame work on how we can think about model optimization, function approximation and so on.



So, we always start with the notion of some data, and then we throw that data into a model, and what comes out of that model is a prediction that we can then score with a cost function and figure out how to update model's parameters based on gradient descent technics



Now the task becomes replacing our model with a piece of computation that runs on a quantum computer in a way that is beneficial and advantages

Variational models can be sort of Quantum circuit that contains some parameters in it that we want to train, optimize and tweak.

Let's start with a general circuit that start with some input state $|\psi\rangle$ and then we apply some unitary operation $U(\vec{\theta})$ that depend on some vector parameter $\vec{\theta}$, and then we do some sort of a measurement to our system. This vector parameter can be specified, trained and optimize.

When we measure a system, the output is stochastic by definition, so what we do is to repeat the measurement multiple times to get an expectation value, so we get a probability distribution on the possible basis stat when we do a measurement.

So how do we go about setting an architecture with a variational circuit as the classifier or the model that can take some classical data as an input, and gives us a prediction for a label or a target.

Our general Task will be Train a quantum circuit on labelled samples in order to predict labels for new data.

Step1: Encoding the classical data into a quantum state.

Step2: applying a parametrize model.

Step3: measuring the circuit to extract labels.

Step4: Optimize the model's parameters.

2.2. Data encoding

Encoding classical data into a quantum state is still at moment an open question, there's no hard and fast rule on how to do that, in fact it depends very much on the problem at hand, and non of the technics used have really been shown to be beneficial in a certain context.

A quantum embedding represent a classical data as a quantum state in Hilbert space via a quantum feature map, it takes a classical data point x and translates it into a set of gate parameters in a quantum circuit, creating a quantum state $|\psi_x\rangle$

2.2.1. Basis encoding

We are going to consider a classical input data consisting of M examples, with N features each,

$$\mathcal{D} = \left\{x^{(1)}, \dots, x^{(m)}, \dots, x^{(M)}\right\}$$

As the name suggest, basis encoding or basis embedding associates each input with a computational basis state of a qubit system, and to do that, classical data has to be in the form of binary strings, the encoded quantum state is the translation of the binary strings into the corresponding quantum state, for

In general

$$x^{(m)} = (b_1, \dots, b_N) \longrightarrow |x^{(m)}\rangle = |b_1, \dots, b_N\rangle \quad \text{with} \quad b_i \in \{0,1\}$$

The entire dataset can be represented by a superposition of the computational basis states

$$\mathcal{D} \longmapsto |\mathcal{D}\rangle = \frac{1}{\sqrt{M}} \sum_{m=1}^{M} |x^{(m)}\rangle$$

For example

$$\begin{cases} x^{(1)} = 01 \\ x^{(2)} = 11 \end{cases} \longmapsto \begin{cases} |x^{(1)}\rangle = |01\rangle \\ |x^{(2)}\rangle = |11\rangle \end{cases} \implies |\mathcal{D}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |11\rangle)$$

2.2.2. Amplitude encoding

In this technic the data is encoded into the amplitude of a quantum state, a classical N-dimensional datapoint x is represented by the amplitude of a n-qubit state $|\psi_x\rangle$ as

$$|\psi_x\rangle = \frac{1}{\sqrt{N}} \sum_{i=1}^{N} x_i |\varphi_i\rangle$$

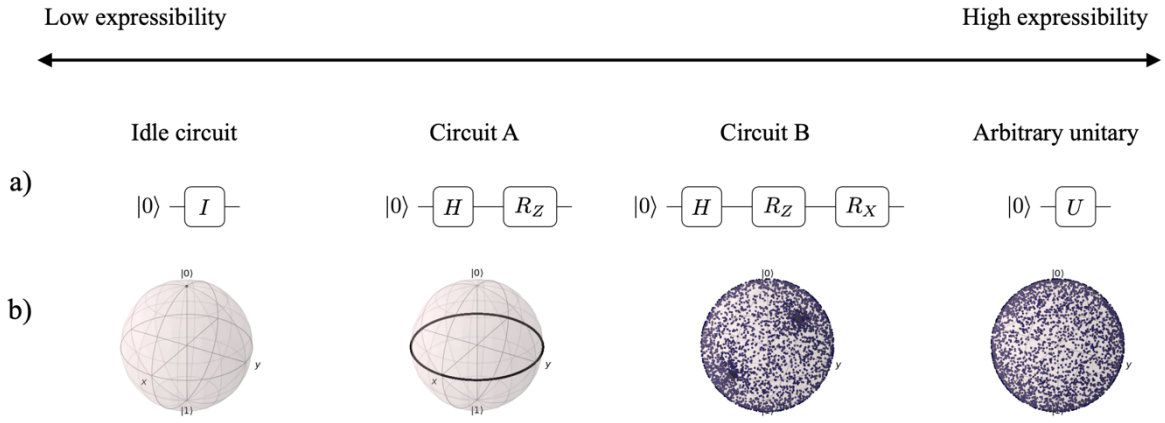
Where $N = 2^n$.

For example, $x = (1.0, 0.0, -5.5, 0.0)$ the state will take the form

$$|\psi_x\rangle = \frac{1}{\sqrt{31.25}}[|00\rangle - 5.5|10\rangle]$$

2.2.3. angle encoding

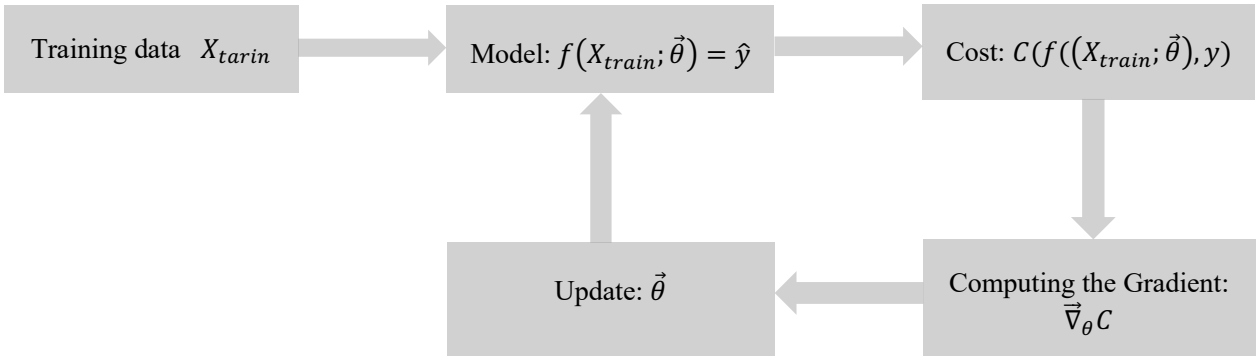
The idea of angle encoding is that we encode the classical data into rotating the qubit along a certain with a parameter that depend on the feature value, resulting state vector. For high accessibility to the Bloch sphere, we can apply multiple rotation along different axis,



So, for example, in the first setting (Idle circuit) we just apply the identity to the first qubit, then what we can access in the Bloch-sphere is nothing. However, in the second setting we put our qubit in a superposition using the Hadamard gate, then applying a rotation along the z-axis, this allows us to access to deferent theta values.

The more general is the gate we apply, the more accessibility we have to the Bloch-sphere. The arbitrary Unitary Gate gives us a complete accessibility to the Bloch -sphere.

Now once we have our data encoded, we move our next task, which applying the variational model that depend on some parameter that we can optimize and tweak using the gradient descent



Where we can compute the gradient using the following method

$$\vec{\nabla}_\theta \; Quantum\; Circuit(\theta) = Quantum\; Circuit(\theta + \delta\theta) - Quantum\; Circuit(\theta - \delta\theta)$$

However, as we will see later on, the Feature Map give rise to the Quantum kernel, which we will use to train our model via Support Vector Machine

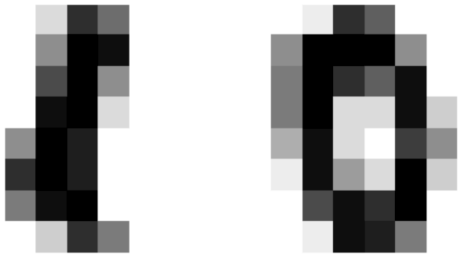
2.3. Implementation

First of all, we will start by loading the training data and testing data from the data base. Our focus will be aiming on a simple example of differentiating between 0 and 1

```
In [2]: digits = datasets.load_digits(n_class=2)
```

And now we're going to start by processing the data, there are a total of 360 datapoint in the dataset, each datapoint is an 8×8 image of a digits, collapsed into an array, where each element is an integer between 0 (white) and 16 (black).

```
In [10]: fig, axs = plt.subplots(1, 2, figsize=(6,3))
         axs[0].set_axis_off()
         axs[0].imshow(digits.images[70], cmap=plt.cm.gray_r, interpolation='nearest')
         axs[1].set_axis_off()
         axs[1].imshow(digits.images[15], cmap=plt.cm.gray_r, interpolation='nearest')
         plt.show()
```



As per classical classification, we need to split the dataset into training (100) and testing (20) samples, and normalize it. To use the dataset for quantum classification, we need to scale the range to between -1 and 1, and reduce the dimensionality to the number of qubits we want to use (4).

We start by splitting the data into training data and testing data

```
In [4]: sample_train, sample_test, label_train, label_test = train_test_split(
            digits.data, digits.target, test_size=0.2, random_state=22)
```

And since we are going to use 4 qubits, we have to use Principal component analysis (PCA). Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

```
In [5]: n_dim = 4
        pca = PCA(n_components=n_dim).fit(sample_train)
        sample_train = pca.transform(sample_train)
        sample_test = pca.transform(sample_test)
```

However, our data is still not normalized, we do this using the standard scaler

```
In [6]: std_scale = StandardScaler().fit(sample_train)
        sample_train = std_scale.transform(sample_train)
        sample_test = std_scale.transform(sample_test)
```

In order to be able to encode this datapoint, the feature values have to range between -1 and 1, we achieve this using the MinMaxScaler

```
In [11]: samples = np.append(sample_train, sample_test, axis=0)
         minmax_scale = MinMaxScaler((-1, 1)).fit(samples)
         sample_train = minmax_scale.transform(sample_train)
         sample_test = minmax_scale.transform(sample_test)
```

And now we take our 100 sample from the training data, and 20 samples from the testing data

```
In [12]: train_size = 100
         sample_train = sample_train[:train_size]
         label_train = label_train[:train_size]

         test_size = 20
         sample_test = sample_test[:test_size]
         label_test = label_test[:test_size]
```

For example

```
In [13]: print(sample_train[70], label_train[70])
         print(sample_test[1], label_test[1])
```

```
[-0.78949582 -0.35954741 -0.50614829 -0.22558562] 0
[-0.00468043  0.84820751  0.08392021  0.07491652] 1
```

Now we move to encoding our data into a quantum state space using quantum feature map.

Quantum feature map

In general, the way we encode our data into a quantum state Is by imposing a unitary operator on the qubit after putting in uniform superposition using the Hadamard gate. The unitary operator used, will give us access the whole Bloch sphere, by apply a rotation or a set of rotations that depend on the feature values, allowing us to encode our classical data.

A unitary operator can be written in the form

$$M = \sum_{i,j} \lambda_{ij} |\varphi_i\rangle\langle\varphi_j| = \lambda_{00}|0\rangle\langle 0| + \lambda_{10}|1\rangle\langle 0| + \lambda_{01}|0\rangle\langle 1| + \lambda_{11}|1\rangle\langle 1|$$

And following the properties of the Pauli matrices:

$$\frac{1+Z}{2} = \frac{1}{2}\left[\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} + \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\right] = |0\rangle\langle 0|$$

$$\frac{1-Z}{2} = \frac{1}{2}\left[\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} - \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\right] = |1\rangle\langle 1|$$

$$\frac{X+iY}{2} = \frac{1+Z}{2}X = |0\rangle\langle 0|X = |0\rangle\langle 1|$$
$$\frac{X-iY}{2} = \frac{1-Z}{2}X = |1\rangle\langle 1|X = |1\rangle\langle 0|$$

From this, it follows that we can decompose any rotation operator into a linear combination of Pauli matrices $\{I, X, Y, Z\}$For a single qubit:

$$R = \sum_{P\in\{I,X,Y,Z\}} C_P P$$

More generally, if we have n qubits

$$R = \sum_{P_{n-1},\ldots,P_0\in\{I,X,Y,Z\}} C_{p_{n-1},\ldots,P_0} P_{n-1}\otimes P_{n-2}\otimes \ldots \otimes P_0$$

In lie-algebra representation, the angular momentum is the rotation generator, in the case of a qubit, the rotation generator will be the spin, hence we can write our unitary operator

$$U_C = exp\left(i \sum_{P_{n-1},\ldots,P_0\in\{I,X,Y,Z\}} C_{p_{n-1},\ldots,P_0} P_{n-1}\otimes P_{n-2}\otimes \ldots \otimes P_0\right)$$

This is the operator that we are going to use to encode our classical data. By using a quantum feature map $\phi(\vec{x})$, our unitary operator will take the form

$$U_{\Phi(\vec{x})} = exp\left(i \sum_{S\subseteq[n]} \phi_S(\vec{x}) \prod_{k\in S} P_i\right)$$

the index S describes connectivities between different qubits or datapoints: $S \in \{\binom{n}{k}\ combinations,\ k = 1,\ldots,n\}$,

And finally, our Unitary operator of depth d

$$\mathcal{U}_{\Phi(\vec{x})} = \prod_d U_{\Phi(\vec{x})}\, H^{\otimes n}$$

which contains layers of Hadamard gates interleaved with entangling blocks, $U_{\Phi(\vec{x})}$, encoding the classical data.

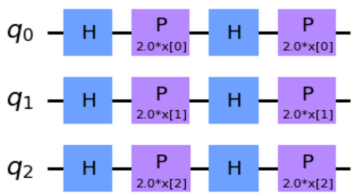
For $k = 1, P_0 = Z$, we get the ZFeatureMap

$$\mathcal{U}_{\Phi(\vec{x})} = \left(exp\left(i \sum_j \phi_j(\vec{x})\, Z_j\right) H^{\otimes n}\right)^d$$

For depth d=2

```
In [15]: map_z = ZFeatureMap(feature_dimension=3, reps=2)
         map_z.decompose(reps=1).draw('mpl')
```

Out[15]:



the lack of entanglement in this feature map, this means that this feature map is simple to simulate classically and will not provide quantum advantage.

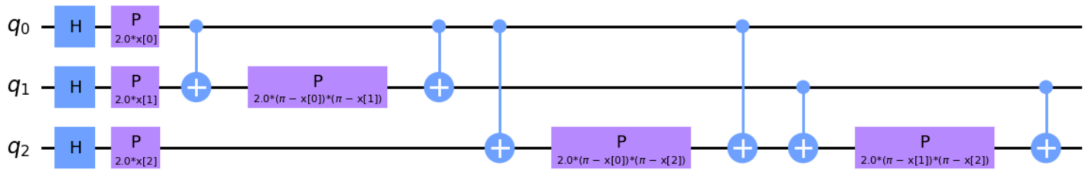
For $k = 2, P_0 = Z, P_1 = ZZ$ this is the ZZFeatur eMap

$$\mathcal{U}_{\Phi(\vec{x})} = \left(exp\left(i \sum_{jk} \phi_{\{j,k\}}(\vec{x})\, Z_j \otimes Z_k \right) exp\left(i \sum_{j} \phi_j(\vec{x})\, Z_j \right) H^{\otimes n} \right)^d$$

For depth $d = 1$

```
In [16]: map_zz = ZZFeatureMap(feature_dimension=3, reps=1)
         map_zz.decompose(reps=1).draw('mpl')
```

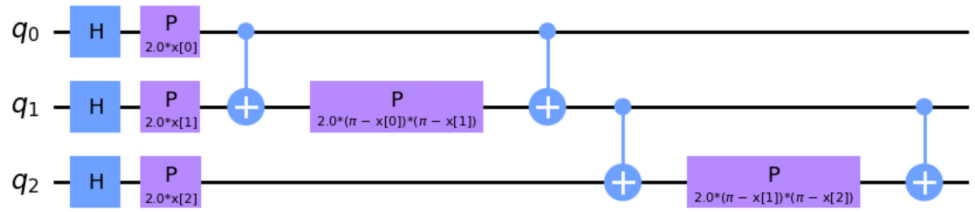
Out[16]:



now that there is entanglement in the feature map, we can define the entanglement map:

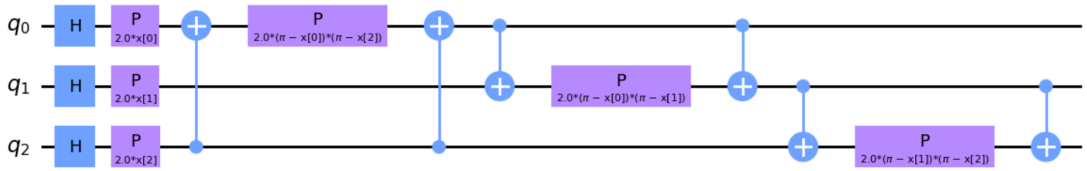
```
In [17]: map_zz = ZZFeatureMap(feature_dimension=3, reps=1, entanglement='linear')
         map_zz.decompose(reps=1).draw('mpl')
```

Out[17]:




```
In [18]: map_zz = ZZFeatureMap(feature_dimension=3, reps=1, entanglement='circular')
         map_zz.decompose(reps=1).draw('mpl')
```

Out[18]:



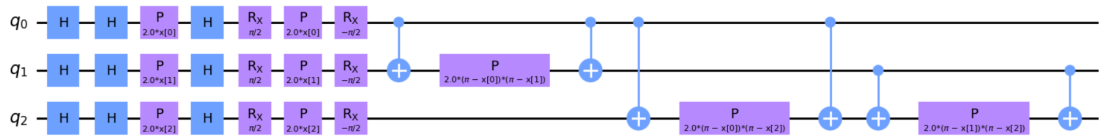
We can also customize the Pauli gates in the feature map, for example, $P_0 = X, P_1 = Y, P_2 = ZZ$

$$\mathcal{U}_{\Phi(\vec{x})} = \left(exp\left(i \sum_{jk} \phi_{\{j,k\}}(\vec{x})\, Z_j \otimes Z_k \right) exp\left(i \sum_{j} \phi_j(\vec{x})\, Y_j \right) exp\left(i \sum_{j} \phi_j(\vec{x})\, X_j \right) H^{\otimes n} \right)^d$$

For depth d=1

In [19]:
```
map_pauli = PauliFeatureMap(feature_dimension=3, reps=1, paulis = ['X', 'Y', 'ZZ'])
map_pauli.decompose(reps=1).draw('mpl')
```

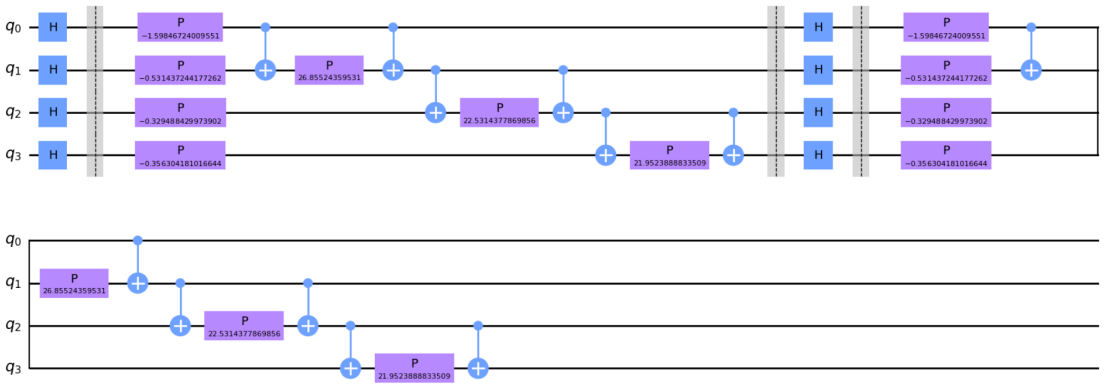
Out[19]:



Now we will encode our first sample using the ZZFeatureMap

In [21]:
```
encode_map = ZZFeatureMap(feature_dimension=4, reps=2, entanglement='linear', insert_barriers=True)
encode_circuit = encode_map.bind_parameters(sample_train[0])
encode_circuit.decompose(reps=1).draw(output='mpl')
```

Out[21]:



Quantum Kernel Estimation

A quantum future map $\phi(\vec{x})$, give rise to the Quantum Kernel $k(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i)^\dagger \phi(\vec{x}_j)$, which can be seen as a measure of similarity: $k(\vec{x}_i, \vec{x}_j)$ is large when $\vec{x}_i$ and $\vec{x}_j$ are close.

When considering a finite data, we can represent the quantum Kernel by the matrix $K_{ij} = \left|\langle \phi^\dagger(\vec{x}_i) | \phi(\vec{x}_j) \rangle\right|^2$. We can calculate each element of the Kernel matrix on a Quantum Computer by calculating the transition amplitude:

$$\left|\langle \phi^\dagger(\vec{x}_i) | \phi(\vec{x}_j) \rangle\right|^2 = \left|\left\langle 0^{\otimes n} \middle| \mathbf{U}_\phi^\dagger(\vec{x}_i) \mathbf{U}_\phi(\vec{x}_j) \middle| 0^{\otimes n} \right\rangle\right|^2$$

Assuming the feature map is a parametrized Quantum Circuit, which can be described as a unitary transformation $\mathbf{U}_\phi(\vec{x}_j)$ on n qubits.

This provides us with an estimate of the quantum kernel matrix, which we can then use in a kernel machine learning algorithm, such as support vector classification.

With our training and testing datasets ready, we set up the QuantumKernel class with the ZZFeatureMap and use the BasicAer statevector_simulatorto estimate the training and testing kernel matrices.

In [24]:
```
zz_map = ZZFeatureMap(feature_dimension=4, reps=2, entanglement='linear', insert_barriers=True)
zz_kernel = QuantumKernel(feature_map=zz_map, quantum_instance=Aer.get_backend('statevector_simulator'))
```

Let's calculate the transition amplitude between the first and second training data samples, one of the entries in the training kernel matrix.

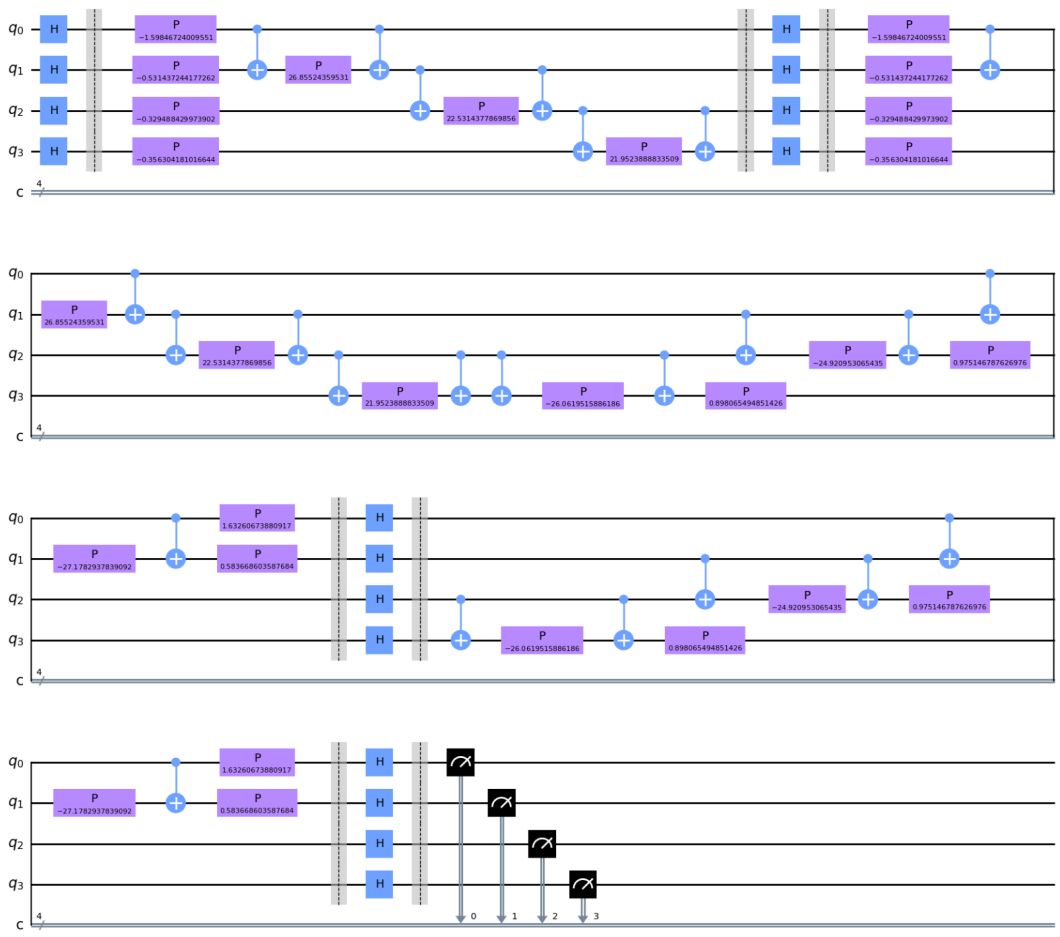
In [22]:
```
print(sample_train[0])
print(sample_train[1])
```

```
[-0.79923362 -0.26571862 -0.16474421 -0.17815209]
[-0.81630337 -0.2918343  -0.48757339 -0.44903275]
```

Now we create and draw the circuit

```
In [25]: zz_circuit = zz_kernel.construct_circuit(sample_train[0], sample_train[1])
         zz_circuit.decompose().decompose().draw(output='mpl')
```

Out[25]:



the circuit is symmetrical, with one half encoding one of the data samples, the other half encoding the other.

We then simulate the circuit. We will use the qasm_simulator since the circuit contains measurements, but increase the number of shots to reduce the effect of sampling noise.

```
In [26]: backend = Aer.get_backend('qasm_simulator')
         job = execute(zz_circuit, backend, shots=8192,
                       seed_simulator=1024, seed_transpiler=1024)
         counts = job.result().get_counts(zz_circuit)
```

The transition amplitude is the proportion of counts in the zero state:

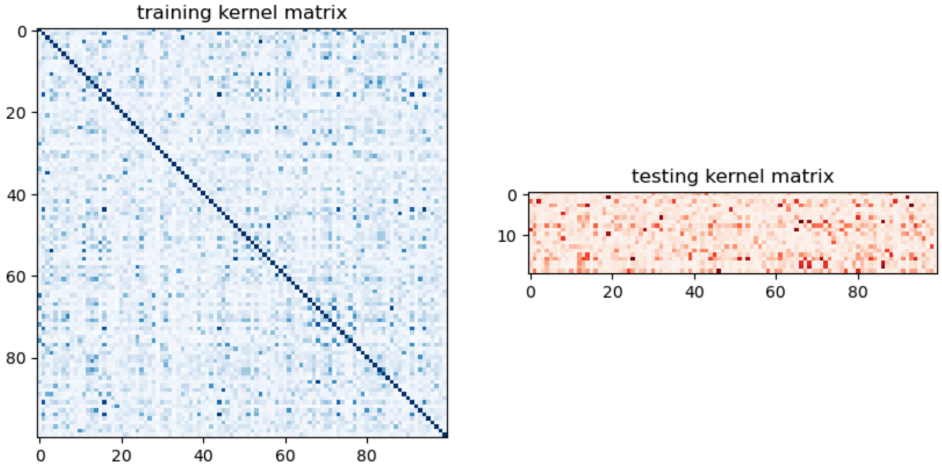
```
In [27]: counts['0000']/sum(counts.values())
```

Out[27]: 0.001953125

This process is then repeated for each pair of training data samples to fill in the training kernel matrix, and between each training and testing data sample to fill in the testing kernel matrix. Note that each matrix is symmetric, so to reduce computation time, only half the entries are calculated explictly.

Here we compute and plot the training and testing kernel matrices:

In [28]:
```
matrix_train = zz_kernel.evaluate(x_vec=sample_train)
matrix_test = zz_kernel.evaluate(x_vec=sample_test, y_vec=sample_train)

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(np.asmatrix(matrix_train),
              interpolation='nearest', origin='upper', cmap='Blues')
axs[0].set_title("training kernel matrix")
axs[1].imshow(np.asmatrix(matrix_test),
              interpolation='nearest', origin='upper', cmap='Reds')
axs[1].set_title("testing kernel matrix")
plt.show()
```



the quantum kernel support vector classification algorithm consists of these steps:

1. Build the train and test quantum kernel matrices.
   A. For each pair of datapoints in the training dataset $\vec{x}_i$ and $\vec{x}_j$ apply the feature map and measure the transition probability $K_{ij} = \left|\left\langle 0^{\otimes n}\middle|\mathbf{U}_\phi^\dagger(\vec{x}_i)\mathbf{U}_\phi(\vec{x}_j)\middle|0^{\otimes n}\right\rangle\right|^2$.
   B. For each training datapoint $\vec{x}_i$ and testing datapoint $\vec{y}_j$, apply the feature map and measure the transition probability $K_{ij} = \left|\left\langle 0^{\otimes n}\middle|\mathbf{U}_\phi^\dagger(\vec{x}_i)\mathbf{U}_\phi(\vec{x}_j)\middle|0^{\otimes n}\right\rangle\right|^2$.
2. Use the train and test quantum kernel matrices in a classical support vector machine classification algorithm.

The following code takes the training and testing kernel matrices we calculated earlier and provides them to the scikit_learn svc algorithm:

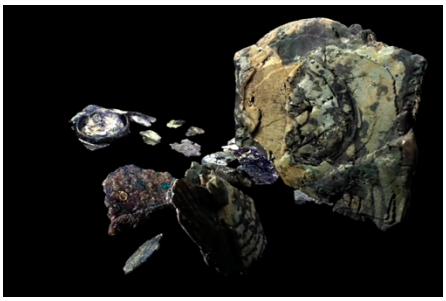
In [29]:
```
zzpc_svc = SVC(kernel='precomputed')
zzpc_svc.fit(matrix_train, label_train)
zzpc_score = zzpc_svc.score(matrix_test, label_test)

print(f'Precomputed kernel classification test score: {zzpc_score}')
```

```
Precomputed kernel classification test score: 0.95
```

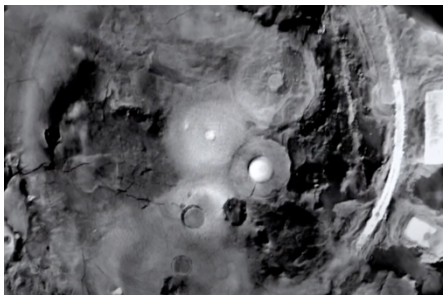
3. Analog Machines: A new hope?

3.1. History

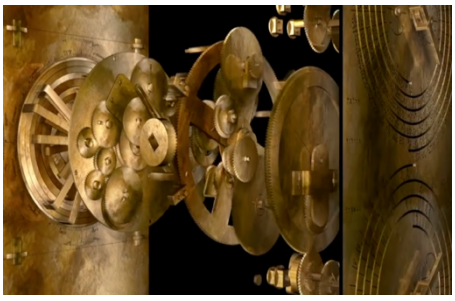
In 1901 A.D an ancient Greek artifact was discovered in a shipwreck of the island of Antikythera. 3D X-ray scans have revealed it contains 37 interlocking bronze gears, allowing it to model the motion of the sun and moon. And predict eclipses decades in advance.



the discovered Antikythera.



3D X-ray scan.



Visualization of the Antikythera

Constructed around a 100 B.C or 200 B.C, the Antikythera mechanism represent a sophisticated early computer.

However, this computer didn't work like the way modern digital computers work, it worked by analogy.

The gears were constructed in such a way that the motion of certain dials are analogues to the motion of the sun and moon. It is an analog computer.

the difference between analog computers and digital computer, is that in analog computers work on continuous rage of inputs and outputs, and the quantities of interest are represented by a physical parameter.

Where in digital computers work on discrete inputs and outputs, and the quantities of interest are represented by symbols, like 0 and 1.

Up until the 1960s, the most powerful computers on the planet, were actually analog. Which was used in WWII, called THE NORDEN BOMBSIGHT.

Designed by the Dutch engineer CARL NORDEN, the Norden bombsight was built to enable high precision airborne bombing. it implemented 64 different simultaneous algorithms, including one that compensated the rotation of earth as the bomb fell.

But despite the enormous precision it was built on, and over 2000 fine parts, the Norden bombsight didn't work as expected.

The problem with analog computers, is that the physical device is a model of the real world, so any inaccuracy in the components, translates into inaccuracy of the computation. And since there will always be some slop in the connections between parts, if you run the same calculation twice, you won't get the same result.

Digital computer exploded into the scene with the advent of solid-state transistors. Now almost everything is digital.

What really opened the door to digital revolution, was the discovery made by Claude Shannon in his 1936 master's thesis. He showed that any numerical operator can be carried out using the basic building blocks of Boolean algebra.

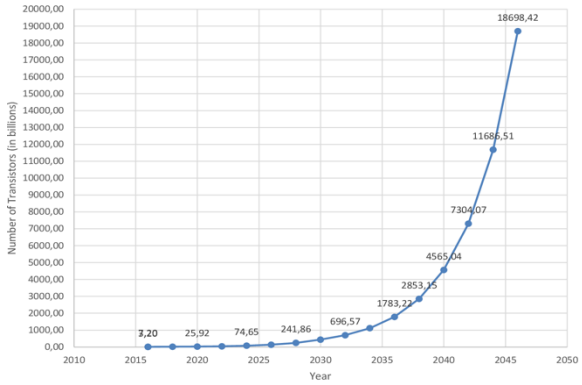
This made digital computers the ideal versatile computing machines.in contrast, each analog computer is an analog for only one type of problem.

furthermore, since digital computers operate on 0s and 1s, they are more resilient in the face of noise. Whereas a small error in analog computer can grow to a large error in the result.

But that may not be for too long.

Moore's law, the idea that you can double the number of transistors on a chip every two years. It's reaching its limit, because transistors are nearly the size of atoms.

Simultaneously, advancement in machine learning, are straining the capability of digital computers. The solution to these challenges, may well be a new generation of analog computers.



3.2. Machine Learning and Analog Computing

By the mid 2000s, most A.I researchers were focusing on improving algorithms. But despite all the advances, artificial neural network still struggled with seemingly simple tasks, and no one knew whether hardware or software was the weak link.

One researcher Fei-Fei Li thought maybe there was a different problem. Maybe these artificial neural networks just needed more data to train on. So, she planned to map out the entire world of object, from 2006 to 2009, she created ImageNet. A data base of 1.2 Million human labeled images, and from 2010 to 2017, ImageNet ran an annual contest:

The ImageNet Large Scale Visual Recognition Challenge. Where software programmers competed to correctly detect and classify images.

In 2012 an artificial neural network from the university of Toronto, blew away the competition, with a Top-5 error rate of just 16.4%. what set his neural network apart was its size and depth. The neural consisted of eight layers, and in total of 500,000 neurons. And to train this network, 60,000,000 weight and biases has to be carefully adjusted using the training database. Because of all the matrix multiplication, processing a single image required 700,000,000 individual math operations. So, the training was computationally intensive. Team managed it by the use of GPUs.

So, the future is clear we will see an ever-increasing demand for ever larger neural networks, and this a problem for several reasons:

- Energy consumption: training a neural network requires an amount of electricity similar to the yearly consumption of three households
- Von Neumann bottleneck: virtually every modern digital computer stores data in memory, and then access it as needed over a bus. When preforming a huge matrix multiplication required be deep neural network, most of the time and energy goes into fetching those weight values rather than actually doing the computation.
- Moore's law, imposing another problem for the digital computers

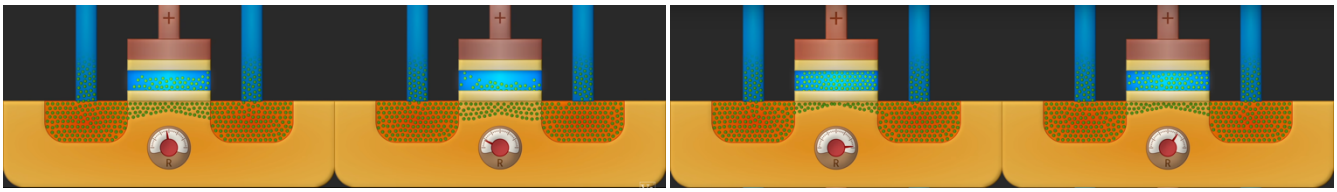
So digital computers are reaching their limits, meanwhile, neural network are exploding in popularity, and most of what they do, boils doing to a single task: matrix multiplication.

An analog computing startup called Mythic A.I, they're creating analog chips to run neural networks

the use cases for these chips goes from augmented reality to depth estimation.

To make this possible, they have repurposed digital flash storage cells. Normally these are used as memory to store either 1 or 0. If we apply a large positive voltage to the control gate, electrons tunnel up through an insulating barrier and become trapped on the floating gate.

After removing the voltage, the electrons remain on the floating gate for decades, preventing the cell from conducting current. We can read out the stored value by applying a small voltage, if there are electrons in the floating gates, no current flows, so that's a zero. If there aren't any electrons in the floating gate, the current does flow, and that's a one.



The Mythic A.I idea is to use these cells not as on/off switches but as variable resistors, by putting a specific number of electrons on each floating gate instead of all or nothing.

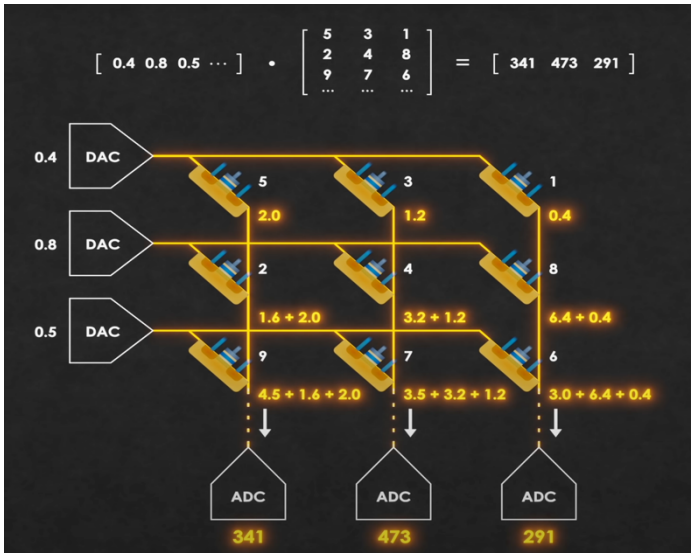
The greater the number of electrons the higher the resistance of the channel

Later when we apply a current of voltage, the current that flows is $I = GV$, where G is the conductance. So, a single flash cell can be used to multiply two values together, voltage times conductance.

So, to use this to run an artificial neural network, they associate all the weights to the flash cell's conductance, then we input the activation value as the voltage on the cells and the resulting product is the voltage × conductance, these cells are wired together in such a way that the current from each multiplication adds together, completing the matrix multiplication.

A single Mythic A.I chip can do 25 Trillion math operation per second, burning about 3Watts of power.

Whereas the newest digital chip can do about 25 Trillion operation per second. Burning out 100Watts of power, and a size of 100 times of that of the Mythic A.I chip.



Which make analogs chip perfect for single purpose A.I, Like security cameras, autonomous systems …

We can't say whether analog computers will take off the way digital did in the last century, but they do seem to be better suited for a lot of the tasks that we want our computers to perform today.

3.3. Summary: where does quantum finds its place

The reason I'm making an argument for analog computing, because it opens the way into enhancing physical laws into the way we do our computation, the simple multiplication of voltage and conductance is just a simple example to illustrate how we can reduce the number of steps required by simply making the system analogues to the real world.

The next step will be enhancing the quantum properties into building computation models, in a way that is dedicated to a specific use case. We definitely lose generality of our model to perform any giving computational task, but we increase the efficiency of our model, as well as the performance under those specific tasks that is designed for.