1 Methodology & implementation

In this section, we will give brief introductions on the implementation of the various algorithms and methods presented in the theory section. These introductions will give a general overview of the steps taken in our project, and how the different parts of the project are connected. We will also present some insights behind some of our choices in the implementation.

1.1 Data encoding

For our simple, 4 feature IRIS dataset, we will encode the data on qubits in the following way:

- For each feature, we assign a qubit
- For each feature value, we assign a rotation angle θ, scaled with the sample value, to the corresponding qubit

1.1.1 Pre-processing of the dataset

As explained in the theory section (**??**), we need to scale the feature values (the samples in the dataset) to the interval $[0, 4\pi]$. To do so, we will use the built in MinMaxScaler in the Scikit-learn library [**scikit-learn**], which scales the values to the interval $[0, 1]$. We will then multiply the scaled values with 4π to get the desired interval.
We will also split the dataset into training and testing sets, using the built-in method train_test_split in Scikit-learn. This method will split the dataset into two parts, one for training the model and one for testing the model. We will use 80% of the dataset for training and 20% for testing, and this split is necessary to prevent overfitting to our dataset, and to assess the generalization capabilities of our model.

1.1.2 Encoding the data on qubits

This is easily done using the Qiskit framework, see [**qiskit2024**], using the gates presented in section **??**. For each scaled feature value, first apply a Hadamard gate before we apply a rotation gate (with the corresponding angle θ) to the corresponding qubit.

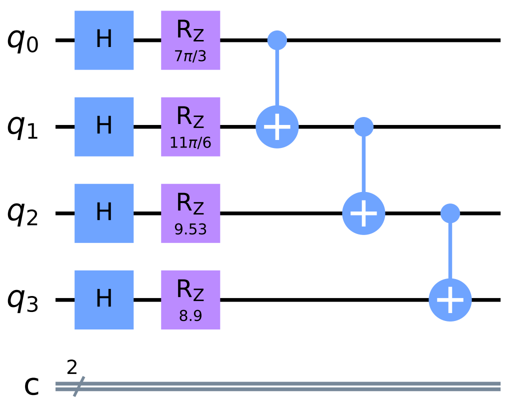

Figure 1: A quantum circuit encoding a single sample of the IRIS dataset. The rotation angles are scaled with the feature values.

A circuit of a single encoded sample is shown in figure **??**, where the Hadamard gate is used to create superposition in each qubit.

1.2 Quantum Circuit

As stated previously, we will be using the Qiskit framework to implement our quantum circuits. This framework allows for— easy implementation of quantum computing algorithms, and provides a wide range of tools for quantum computing. As an example, the following code snippet will reproduce the data encoding circuit shown in figure **??**:

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
qreg = QuantumRegister(4)
creg = ClassicalRegister(2)
circuit = QuantumCircuit(qreg, creg)
circuit.h(qreg)
circuit.ry(theta1, qreg[0])
circuit.ry(theta2, qreg[1])
circuit.ry(theta3, qreg[2])
circuit.ry(theta4, qreg[3])
circuit.barrier()
```

where the angles θ should be calculated from the sample values as explained in the previous section.

1.3 Variational ansatz

The variational ansatz is implemented in a similar way as the data encoding, using the Qiskit framework. The ansatz is a simple circuit, consisting of CNOT gates to create entanglement between the qubits, and rotation gates to introduce the variational parameters. The following code snippet shows how this can be done:

```
some_params = ...
circuit = QuantumCircuit(qreg, creg)
circuit.h(qreg)
for i in range(4):
    circuit.ry(some_params[i], qreg[i])
for c in range(4):
    for t in range(c, 4):
        if c != t:
            circuit.cx(qreg[c], qreg[t])
.circuit.ry(some_params[-1], 3)
```

where the variational parameters are stored in a list, and the circuit is constructed similarly to the data encoding circuit. Here we've put CNOT gates between each pair of qubits, but the entanglement can be constructed in any way that is deemed necessary for the problem at hand. We've also added a "bias" term at the end of the circuit, on the qubit that we are going to measure - just to add more complexity to our model. For our actual quantum circuit, we will initialize the parameters randomly in the domain $[-2\pi, 2\pi]$, before starting the optimization.
Running the above snippet yields the following circuit shown in figure **??**.

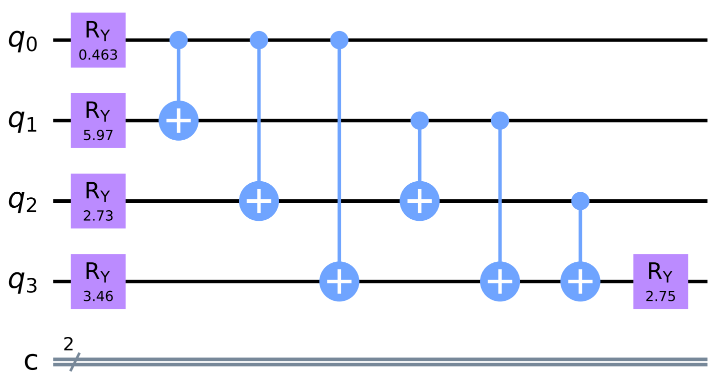

Figure 2: A simple variational ansatz circuit, with rotation gates and CNOT gates.

1.4 Making predictions

To use our quantum circuit for classification tasks, we will need to map the measurements done on the qubits to an output that mimics the target matrix of our dataset. The measurements themselves are easily performed in Qiskits framework - using a quantum simulator (since we currently lack access to a quantum computer) - and the results are stored in the classical registers. Using the mapping presented in section (**??**), we identify that we need only to measure two of the four qubits needed to encode the Iris dataset. This is done by applying a measurement gate to the two qubits we want to measure, and then store the result in a matrix to be used to evaluate the loss function. A simple example of how this can be done is shown in the following code snippet:

```
frorm qiskit import transpile
from qiskit_aer import AerSimulator
circuit.measure(qreg[-1], creg[0])
backend = AerSimulator(method="statevector")
circuit = transpile(circuit, backend)
result = backend.run(circuit, shots=10000).result()
counts = result.get_counts()
tmp = 0.0
for key in counts.keys():
    if key[0] == '1':
        tmp += 1 * counts[key]
prediction = tmp / 10000
```

where the measurement is here done on the last qubit. The result is then stored in the variable prediction, which can be used to evaluate the loss function. It is important that we use the AerSimulator with the statevector argument to get the correct probabilities, as the other noisy simulators will struggle with convergence for such few shots.

1.5 Optimization

1.5.1 Loss function

Implementing the binary cross-entropy loss function (**??**) is pretty straightforward in Python, and can be done using the following code snippet:

```
def cross_entropy_loss(y_true, y_pred):
    loss = 0
    for i in range(len(y_true)):
        loss += - (y_true[i]*np.log(y_pred[i]) + (1 - y_true[i])*np.log(1 - y_pred[i]))
    return loss
```

where y_{true} is the target vector and y_{pred} is the predicted vector. The loss function is then used to evaluate the performance of the model, and to update the variational parameters in the optimization process, as explained in section (**??**).

1.5.2 Gradient Descent

The simplest of optimization algorithms that we will use to update the variational parameters of our quantum circuit is the gradient descent algorithm. For our quantum circuit, this gradient descent optimization algorithm is as follows:

Algorithm 1 Gradient Descent

```
1: Initialize the variational parameters $\theta$
2: while not converged do
3:     Calculate the gradient of the loss function with respect to the variational parameters
4:     Update the variational parameters using the gradient
5:     Evaluate the loss function and check for convergence
6: end while
```

1.5.3 Scipy-minimize by gradient-free methods

Due to the barren plateu problem that often arises when optmizing quantum circuit, we will also implement a non-gradient based optimization algorithm, namely the COBYLA algorithm from the Scipy library. COBYLA (Constrained Optimization by Linear Approximations) is a derivative-free optimization algorithm that is well suited for optimization problems where the gradient is not known[**powell1994direct**].