In [1]:
```
import numpy as np
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
# For NN
from qiskit.circuit import Parameter
from qiskit.circuit.library import RealAmplitudes, ZZFeatureMap
from qiskit.opflow import StateFn, PauliSumOp, AerPauliExpectation, Lis
tOp, Gradient
from qiskit.utils import QuantumInstance
# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

In [2]:
```
# set method to calculcate expected values
expval = AerPauliExpectation()

# define gradient method
gradient = Gradient()

# define quantum instances (statevector and sample based)
qi_sv = QuantumInstance(Aer.get_backend('statevector_simulator'))

# we set shots to 10 as this will determine the number of samples later
on.
qi_qasm = QuantumInstance(Aer.get_backend('qasm_simulator'), shots=10)
```

In [3]:
```
from qiskit_machine_learning.neural_networks import OpflowQNN
```

In [4]:
```
# construct parametrized circuit
params1 = [Parameter('input1'), Parameter('weight1')]
qc1 = QuantumCircuit(1)
qc1.h(0)
qc1.ry(params1[0], 0)
qc1.rx(params1[1], 0)
qc_sfn1 = StateFn(qc1)

# construct cost operator
H1 = StateFn(PauliSumOp.from_list([('Z', 1.0), ('X', 1.0)]))

# combine operator and circuit to objective function
op1 = ~H1 @ qc_sfn1
print(op1)
```

```
ComposedOp([
  OperatorMeasurement(1.0 * Z
  + 1.0 * X),
  CircuitStateFn(
       ┌───┐┌─────────────┐┌──────────────┐
  q_0: ┤ H ├┤ RY(input1) ├┤ RX(weight1) ├
       └───┘└─────────────┘└──────────────┘
  )
])
```

In [5]:
```
# construct OpflowQNN with the operator, the input parameters, the weig
ht parameters,
# the expected value, gradient, and quantum instance.
qnn1 = OpflowQNN(op1, [params1[0]], [params1[1]], expval, gradient, qi_
sv)
```

In [6]:
```
# define (random) input and weights
input1 = np.random.rand(qnn1.num_inputs)
weights1 = np.random.rand(qnn1.num_weights)
```

In [7]:
```
# QNN forward pass
qnn1.forward(input1, weights1)
```

Out[7]:
```
array([[0.99334109]])
```

In [8]:
```
# QNN batched forward pass
qnn1.forward([input1, input1], weights1)
```

Out[8]:
```
array([[0.99334109],
       [0.99334109]])
```

In [9]:
```
# QNN backward pass
qnn1.backward(input1, weights1)
```

Out[9]:
```
(array([[[-0.93839794]]]), array([[[0.00259456]]]))
```

In [10]:
```
# QNN batched backward pass
qnn1.backward([input1, input1], weights1)
```

Out[10]:
```
(array([[[-0.93839794]],

        [[-0.93839794]]]),
 array([[[0.00259456]],

        [[0.00259456]]]))
```

In [11]:
```
op2 = ListOp([op1, op1])
qnn2 = OpflowQNN(op2, [params1[0]], [params1[1]], expval, gradient, qi_
sv)
```

In [12]:
```
# QNN forward pass
qnn2.forward(input1, weights1)
```

Out[12]:
```
array([[0.99334109, 0.99334109]])
```

In [13]:
```
# QNN backward pass
qnn2.backward(input1, weights1)
```

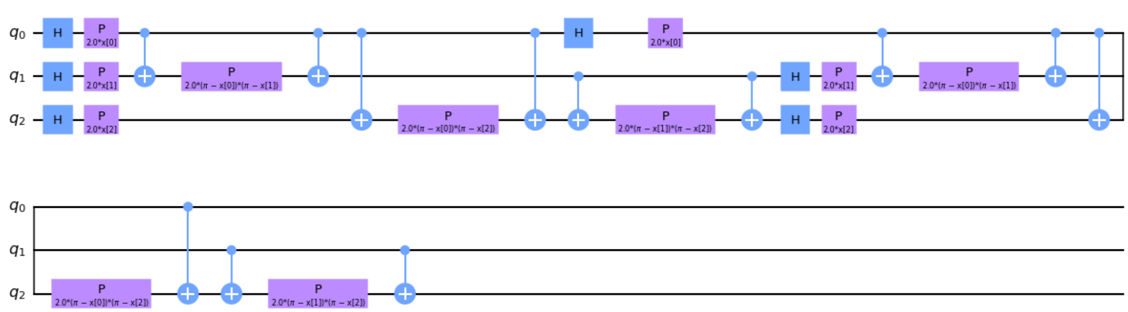
Out[13]:
```
(array([[[-0.93839794],
         [-0.93839794]]]),
 array([[[0.00259456],
         [0.00259456]]]))
```

In [14]:
```
#TwoLayerQNN
from qiskit_machine_learning.neural_networks import TwoLayerQNN
```

In [15]:
```
# specify the number of qubits
num_qubits = 3
```

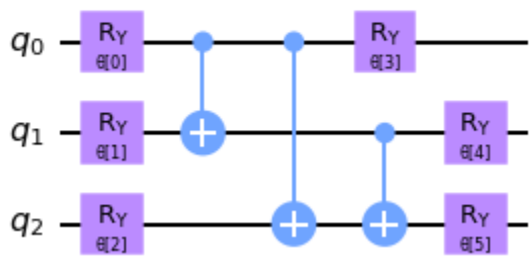
In [16]:
```
# specify the feature map
fm = ZZFeatureMap(num_qubits, reps=2)
fm.draw(output='mpl')
```

Out[16]:



In [17]:
```
# specify the ansatz
ansatz = RealAmplitudes(num_qubits, reps=1)
ansatz.draw(output='mpl')
```

Out[17]:



In [18]:
```
# specify the observable
observable = PauliSumOp.from_list([('Z'*num_qubits, 1)])
print(observable)
```

```
1.0 * ZZZ
```

In [19]:
```
# define two layer QNN
qnn3 = TwoLayerQNN(num_qubits,
                   feature_map=fm,
                   ansatz=ansatz,
                   observable=observable, quantum_instance=qi_sv)
```

In [20]:
```
# define (random) input and weights
input3 = np.random.rand(qnn3.num_inputs)
weights3 = np.random.rand(qnn3.num_weights)
```

In [21]:
```
# QNN forward pass
qnn3.forward(input3, weights3)
```

Out[21]:
```
array([[0.51661076]])
```

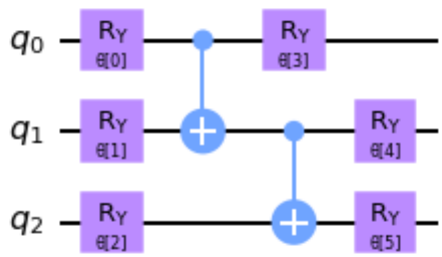
In [22]:
```
# QNN backward pass
qnn3.backward(input3, weights3)
```

Out[22]:
```
(array([[[-0.2911684 , -1.84712948, -0.43124692]]]),
 array([[[-0.22940634, -0.03568436, -0.22217109, -0.00993838,
          0.55941921, -0.34746975]]]))
```

In [23]:
```
#CircuitQNN
from qiskit_machine_learning.neural_networks import CircuitQNN
```

In [24]:
```
qc = RealAmplitudes(num_qubits, entanglement='linear', reps=1)
qc.draw(output='mpl')
```

Out[24]:



In [25]:
```
# specify circuit QNN
qnn4 = CircuitQNN(qc, [], qc.parameters, sparse=True, quantum_instance=
qi_qasm)
```

In [26]:
```
# define (random) input and weights
input4 = np.random.rand(qnn4.num_inputs)
weights4 = np.random.rand(qnn4.num_weights)
```

In [27]:
```
# QNN forward pass
qnn4.forward(input4, weights4).todense()  # returned as a sparse matrix
```

Out[27]:
```
array([[0.9, 0. , 0.1, 0. , 0. , 0. , 0. , 0. ]])
```

In [28]:
```
# QNN backward pass, returns a tuple of sparse matrices
qnn4.backward(input4, weights4)
```

Out[28]:
```
(<COO: shape=(1, 8, 0), dtype=float64, nnz=0, fill_value=0.0>,
 <COO: shape=(1, 8, 6), dtype=float64, nnz=24, fill_value=0.0>)
```

In [29]:
```
#dense parity probabilities
# specify circuit QNN
parity = lambda x: '{:b}'.format(x).count('1') % 2
output_shape = 2  # this is required in case of a callable with dense o
utput
qnn6 = CircuitQNN(qc, [], qc.parameters, sparse=False, interpret=parit
y, output_shape=output_shape,
                  quantum_instance=qi_qasm)
```

In [30]:
```
# define (random) input and weights
input6 = np.random.rand(qnn6.num_inputs)
weights6 = np.random.rand(qnn6.num_weights)
```

In [31]:
```
# QNN forward pass
qnn6.forward(input6, weights6)
```

Out[31]:
```
array([[0.8, 0.2]])
```

In [32]:
```
# QNN backward pass
qnn6.backward(input6, weights6)
```

Out[32]:
```
(array([], shape=(1, 2, 0), dtype=float64),
 array([[[-0.25, -0.05, -0.1 , -0.3 , -0.3 ,  0.2 ],
         [ 0.25,  0.05,  0.1 ,  0.3 ,  0.3 , -0.2 ]]]))
```

In [33]:
```
#Samples
# specify circuit QNN
qnn7 = CircuitQNN(qc, [], qc.parameters, sampling=True,
                  quantum_instance=qi_qasm)
```

In [34]:
```
# define (random) input and weights
input7 = np.random.rand(qnn7.num_inputs)
weights7 = np.random.rand(qnn7.num_weights)
```

In [35]:
```
# QNN forward pass, results in samples of measured bit strings mapped t
o integers
qnn7.forward(input7, weights7)
```

Out[35]:
```
array([[[6.],
        [7.],
        [2.],
        [2.],
        [0.],
        [2.],
        [2.],
        [0.],
        [7.],
        [6.]]])
```

In [36]:
```
# QNN backward pass
qnn7.backward(input7, weights7)
```

Out[36]:
```
(None, None)
```

In [37]:
```
#Parity Samples
# specify circuit QNN
qnn8 = CircuitQNN(qc, [], qc.parameters, sampling=True, interpret=parit
y,
                  quantum_instance=qi_qasm)
```

In [38]:
```
# define (random) input and weights
input8 = np.random.rand(qnn8.num_inputs)
weights8 = np.random.rand(qnn8.num_weights)
```

In [39]:
```
# QNN forward pass, results in samples of measured bit strings
qnn8.forward(input8, weights8)
```

Out[39]:
```
array([[[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [1.],
        [0.],
        [0.],
        [1.]]])
```

In [40]:
```
# QNN backward pass
qnn8.backward(input8, weights8)
```

Out[40]:
```
(None, None)
```

In [ ]:
```
# Executed by Bhadale IT in IBM Quantum Lab, demo of QNN
```