Lecture 2 - 25^{th} November 2024

Lecture 2 on Variational Quantum Algorithm

This notebook presentation is part of my lecture material for the Advanced Quantum Mechanics course at the University of Trieste. It contains a general introduction to Qiskit and Pennylane
Michele Grossi, Coordinator of Hybrid Quantum Computing Infrastracture and Algorithms @ CERN

Variational Quantum Algorithms

Variational Quantum Algorithms (VQA) are hybrid algorithm that use a classical optimizer to train a parameterized quantum circuit to approximate solutions for a given problem. You will find this family of architectures for a lot of application in different fields like: Chemistry, Physics, Finance, Machine Learning and more.

VQA typically need fewer gates and qubits. In turn, they are more resistant to noise and are well suited to handle near-term quantum computer constraints.

VQA's are typically iterative. Each iteration involves both quantum and classical processing.

Output (a measurement) from one iteration is sent to the classical optimizer which generates input (a parameter) for the next iteration:

No description has been provided for this image

VQE - Application to Physics to study static properties

The Variational Quantum Eigensolver (VQE) is a central algorithm in many applications, e.g. quantum chemistry or optimization. This tutorial shows you how to run the VQE. We'll start off by defining the algorithm settings, such as the Hamiltonian and ansatz, and then run a VQE.

In [8]:
```
# VQE with PennyLane: Ground State of a 5-Qubit Hamiltonian
import pennylane as qml
from pennylane import numpy as np
from scipy.linalg import eigh

# 1. Define the problem Hamiltonian
def define_hamiltonian():
```

```
    """Construct the Hamiltonian for the VQE problem."""
    coeffs = np.random.uniform(-1, 1, 15)  # Random coefficients
    obs = [
        qml.PauliZ(0) @ qml.PauliZ(1),
        qml.PauliZ(1) @ qml.PauliZ(2),
        qml.PauliZ(2) @ qml.PauliZ(3),
        qml.PauliZ(3) @ qml.PauliZ(4),
        qml.PauliZ(0) @ qml.PauliZ(4),
        qml.PauliX(0),
        qml.PauliX(1),
        qml.PauliX(2),
        qml.PauliX(3),
        qml.PauliX(4),
        qml.PauliY(0) @ qml.PauliY(1),
        qml.PauliY(2) @ qml.PauliY(3),
        qml.PauliY(3) @ qml.PauliY(4),
        qml.PauliY(0) @ qml.PauliY(4),
        qml.PauliZ(1) @ qml.PauliZ(3),
    ]
    return qml.Hamiltonian(coeffs, obs)

hamiltonian = define_hamiltonian()
n_qubits = 5
```

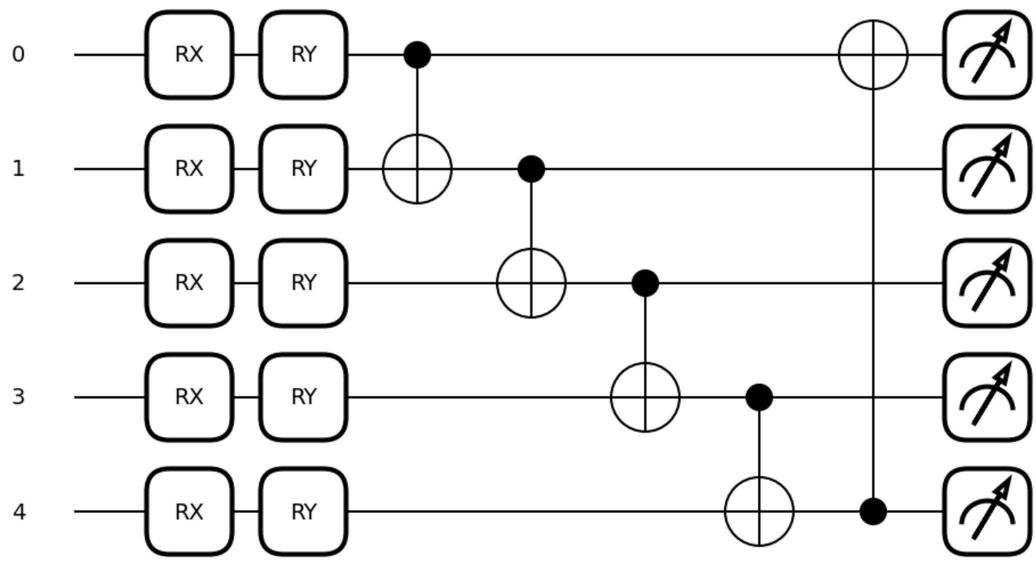
In [9]:
```
# Classical solution for comparison
def classical_ground_state(hamiltonian):
    """Compute the ground state energy of the Hamiltonian classical
    hamiltonian_matrix = qml.matrix(hamiltonian)  # Get the matrix
    eigenvalues, _ = eigh(hamiltonian_matrix)     # Diagonalize the
    return min(eigenvalues)

# Compute classical ground state
classical_solution = classical_ground_state(hamiltonian)
print(f"Classical ground state energy: {classical_solution:.6f}")
```

```
Classical ground state energy: -4.414483
```

In [10]:
```
# 3. Define the VQE quantum circuit
dev = qml.device("default.qubit", wires=n_qubits)

def variational_ansatz(params):
    """Define the variational ansatz."""
    for i in range(n_qubits):
        qml.RX(params[i], wires=i)
        qml.RY(params[i + n_qubits], wires=i)
    for i in range(n_qubits - 1):
        qml.CNOT(wires=[i, i + 1])
    qml.CNOT(wires=[n_qubits - 1, 0])  # Wrap around


# Define the QNode
@qml.qnode(dev)
def circuit(params):
    variational_ansatz(params)
    return qml.expval(hamiltonian)
```

```python
# Generate random parameters
np.random.seed(42)
params = np.random.uniform(0, 2 * np.pi, 2 * n_qubits)

# Draw the circuit
qml.draw_mpl(circuit)(params)
```

Out[10]: (<Figure size 1000x600 with 1 Axes>, <Axes: >)



In [11]:
```python
# 4. Optimize the VQE
np.random.seed(42)
init_params = np.random.uniform(0, 2 * np.pi, 2 * n_qubits)

optimizer = qml.GradientDescentOptimizer(stepsize=0.4)
params = init_params
max_iter = 100
convergence_tol = 1e-6

energy = []
for n in range(max_iter):
    params, cost = optimizer.step_and_cost(circuit, params)
    energy.append(cost)
    print(f"Iteration {n+1}: Energy = {cost:.6f}")
    if len(energy) > 2 and np.abs(energy[-1] - energy[-2]) < conver
        break

print(f"VQE estimated ground state energy: {cost:.6f}")
```

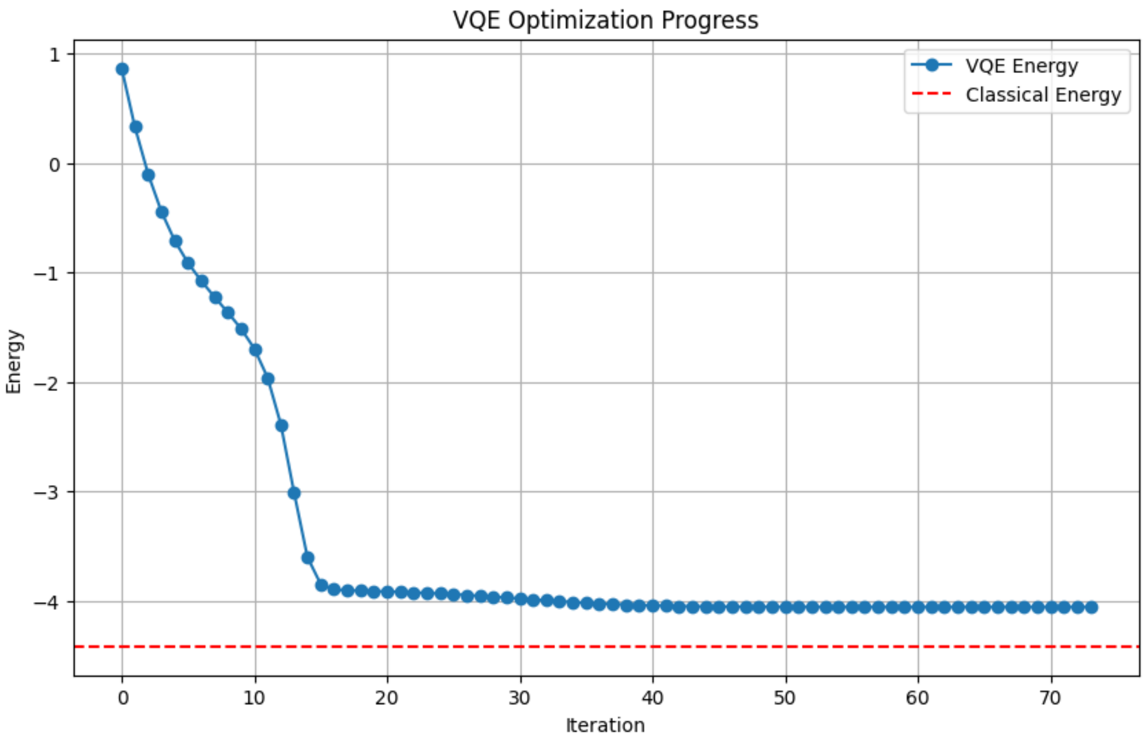
```
Iteration 1: Energy = 0.868477
Iteration 2: Energy = 0.330473
Iteration 3: Energy = -0.101060
Iteration 4: Energy = -0.443666
Iteration 5: Energy = -0.708333
Iteration 6: Energy = -0.911286
Iteration 7: Energy = -1.076336
Iteration 8: Energy = -1.222891
```

```
Iteration 9: Energy = -1.364007
Iteration 10: Energy = -1.513023
Iteration 11: Energy = -1.694888
Iteration 12: Energy = -1.960066
Iteration 13: Energy = -2.388614
Iteration 14: Energy = -3.013419
Iteration 15: Energy = -3.599598
Iteration 16: Energy = -3.847304
Iteration 17: Energy = -3.895453
Iteration 18: Energy = -3.903996
Iteration 19: Energy = -3.907475
Iteration 20: Energy = -3.910641
Iteration 21: Energy = -3.914156
Iteration 22: Energy = -3.918150
Iteration 23: Energy = -3.922676
Iteration 24: Energy = -3.927768
Iteration 25: Energy = -3.933448
Iteration 26: Energy = -3.939722
Iteration 27: Energy = -3.946573
Iteration 28: Energy = -3.953959
Iteration 29: Energy = -3.961805
Iteration 30: Energy = -3.970007
Iteration 31: Energy = -3.978429
Iteration 32: Energy = -3.986911
Iteration 33: Energy = -3.995278
Iteration 34: Energy = -4.003354
Iteration 35: Energy = -4.010978
Iteration 36: Energy = -4.018013
Iteration 37: Energy = -4.024364
Iteration 38: Energy = -4.029975
Iteration 39: Energy = -4.034833
Iteration 40: Energy = -4.038962
Iteration 41: Energy = -4.042414
Iteration 42: Energy = -4.045257
Iteration 43: Energy = -4.047569
Iteration 44: Energy = -4.049429
Iteration 45: Energy = -4.050912
Iteration 46: Energy = -4.052085
Iteration 47: Energy = -4.053009
Iteration 48: Energy = -4.053732
Iteration 49: Energy = -4.054295
Iteration 50: Energy = -4.054734
Iteration 51: Energy = -4.055074
Iteration 52: Energy = -4.055337
Iteration 53: Energy = -4.055541
Iteration 54: Energy = -4.055698
Iteration 55: Energy = -4.055819
Iteration 56: Energy = -4.055913
Iteration 57: Energy = -4.055985
Iteration 58: Energy = -4.056040
Iteration 59: Energy = -4.056083
Iteration 60: Energy = -4.056116
Iteration 61: Energy = -4.056141
Iteration 62: Energy = -4.056161
Iteration 63: Energy = -4.056176
Iteration 64: Energy = -4.056187
```

```
Iteration 65: Energy = -4.056196
Iteration 66: Energy = -4.056203
Iteration 67: Energy = -4.056208
Iteration 68: Energy = -4.056212
Iteration 69: Energy = -4.056215
Iteration 70: Energy = -4.056218
Iteration 71: Energy = -4.056220
Iteration 72: Energy = -4.056221
Iteration 73: Energy = -4.056222
Iteration 74: Energy = -4.056223
VQE estimated ground state energy: -4.056223
```

In [12]:

```
# 5. Compare classical and VQE results
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(energy, label="VQE Energy", marker="o")
plt.axhline(y=classical_solution, color="r", linestyle="--", label='
plt.xlabel("Iteration")
plt.ylabel("Energy")
plt.title("VQE Optimization Progress")
plt.legend()
plt.grid()
plt.show()
```



Discussion about Pennylane return functions

PennyLane offers various return types to extract different kinds of information from quantum circuits. The difference between `qml.state()` and `qml.expval(hamiltonian)` lies in the type of information returned and its intended use.

1. qml.state()

- Purpose: Returns the full quantum state (statevector) of the system after executing the circuit.
- Output: A complex-valued array representing the amplitudes of the quantum state. Useful for inspecting or debugging the complete state of the system.
- Usage: Typically used in simulations, where the statevector is accessible. When you need detailed information about the full quantum state. Ideal for debugging or verifying intermediate results in simulations.

In [13]:
```
@qml.qnode(dev)
def circuit(params):
    variational_ansatz(params)
    return qml.state()
```

If the system is in a state $|\psi >= \alpha|0 > +\beta|1 >$ calling `qml.state()` would return the vector $[\alpha, \beta]$

If the system is in a state $|\psi >= \alpha|0 > +\beta|1 >$ calling `qml.state()` would return the vector $[\alpha, \beta]$

2. qml.expval(hamiltonian)

- Purpose: Computes the expectation value of a given observable (e.g., a Hamiltonian) with respect to the final quantum state.
- Output: A scalar value representing the expected measurement outcome of the observable. This is the average value you would obtain if you repeatedly measured the observable on the prepared state.
- Usage: Used in tasks like Variational Quantum Eigensolvers (VQE) to estimate the ground state energy of a Hamiltonian. Suitable for tasks involving real quantum hardware or noisy simulators.

In [14]:
```
@qml.qnode(dev)
def circuit(params):
    variational_ansatz(params)
    return qml.expval(hamiltonian)
```

If the system is in state $|\psi >$ and the Hamiltonian is H the expectation value is computed as $< \psi|H|\psi >$

Qiskit VQE

Note: You can find tutorials on solving more comprehensive problems, such as finding the ground state of the lithium hydride molecule, using the VQE within the tutorials of Qiskit Nature.

we want to use a variational algorithm to find the eigenvalue of the following observable:

$$O_1 = 2II - 2XX + 3YY - 3ZZ$$

This observable has the following eigenvalues:
$\lambda_0 = -6, \lambda_1 = 4, \lambda_2 = 4, \lambda_3 = 6$

In [15]:
```
from qiskit.quantum_info import SparsePauliOp

observable_1 = SparsePauliOp.from_list([("II", 2), ("XX", -2), ("YY'
```

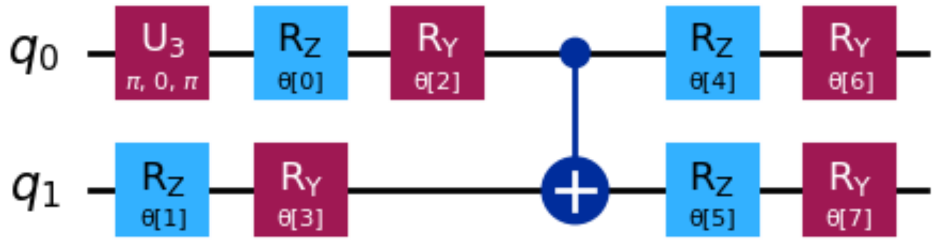
Building VQE from scratch

We'll first explore how to construct a VQE instance manually to find the lowest eigenvalue for O_1.

In [16]:
```
def cost_func_vqe(params, ansatz, hamiltonian, estimator):
    """Return estimate of energy from estimator

    Parameters:
        params (ndarray): Array of ansatz parameters
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        hamiltonian (SparsePauliOp): Operator representation of Ham
        estimator (Estimator): Estimator primitive instance

    Returns:
        float: Energy estimate
    """
    pub = (ansatz, hamiltonian, params)
    cost = estimator.run([pub]).result()[0].data.evs

    return cost
```

In [17]:
```
from qiskit.circuit.library import TwoLocal
from qiskit import QuantumCircuit

import numpy as np

reference_circuit = QuantumCircuit(2)
reference_circuit.x(0)

variational_form = TwoLocal(
    2,
    rotation_blocks=["rz", "ry"],
    entanglement_blocks="cx",
    entanglement="linear",
    reps=1,
)
raw_ansatz = reference_circuit.compose(variational_form)

raw_ansatz.decompose().draw('mpl')
```

Out[17]:



In [18]:
```
## Let's start on local simulators

from qiskit.primitives import StatevectorEstimator as Estimator
from qiskit.primitives import StatevectorSampler as Sampler
estimator = Estimator()
sampler = Sampler()
```

In [19]:
```
## initial parameters
import numpy as np

x0 = np.ones(raw_ansatz.num_parameters)
print(x0)
```

```
[1. 1. 1. 1. 1. 1. 1. 1.]
```

We can minimize this cost function to calculate optimal parameters

In [20]:
```
# SciPy minimizer routine
from scipy.optimize import minimize
import time

start_time = time.time()

result = minimize(cost_func_vqe, x0, args=(raw_ansatz, observable_1

end_time = time.time()
execution_time = end_time - start_time
```

```
   Normal return from subroutine COBYLA

   NFVALS =   123   F =-6.000000E+00    MAXCV = 0.000000E+00
   X = 1.281322E+00   9.423508E-01   1.570825E+00   3.631211E-05
1.917020E+00
         1.224584E+00   6.217758E-01   6.217610E-01
   NFVALS =   123   F =-6.000000E+00    MAXCV = 0.000000E+00
   X = 1.281322E+00   9.423508E-01   1.570825E+00   3.631211E-05
1.917020E+00
         1.224584E+00   6.217758E-01   6.217610E-01
```

In [21]:
```
result
```

Out[21]:
```
 message: Optimization terminated successfully.
 success: True
  status: 1
     fun: -5.999999993368837
       x: [ 1.281e+00  9.424e-01  1.571e+00  3.631e-05  1.917e+00
            1.225e+00  6.218e-01  6.218e-01]
    nfev: 123
   maxcv: 0.0
```

we can check this by using NumPy's linear algebra eigensolver due to the small dimension of the problem.

In [22]:
```
from numpy.linalg import eigvalsh

solution_eigenvalue = min(eigvalsh(observable_1.to_matrix()))

print(f"""Number of iterations: {result.nfev}""")
print(f"""Time (s): {execution_time}""")

print(
    f"Percent error: {abs((result.fun - solution_eigenvalue)/soluti
)
```

```
Number of iterations: 123
Time (s): 0.5258231163024902
Percent error: 1.11e-09
```

Experimenting with different optimizers

We can adjust the optimizer using SciPy `minimize`'s method argument, with more options found here. We originally used a constrained minimizer (`COBYLA`). In this example, we'll explore using an unconstrained minimizer (`BFGS`) instead

In [23]:
```
import time
start_time = time.time()

result = minimize(cost_func_vqe, x0, args=(raw_ansatz, observable_1

end_time = time.time()
execution_time = end_time - start_time
```

In [24]:
```
print("CHANGED TO BFGS OPTIMIZER:")
print(f"""Number of iterations: {result.nfev}""")
print(f"""Time (s): {execution_time}""")
```

```
CHANGED TO BFGS OPTIMIZER:
Number of iterations: 117
Time (s): 0.504896879196167
```

VQE with Qiskit Primitives

In [25]:
```
from qiskit.primitives import Estimator
from qiskit_algorithms import VQE
```

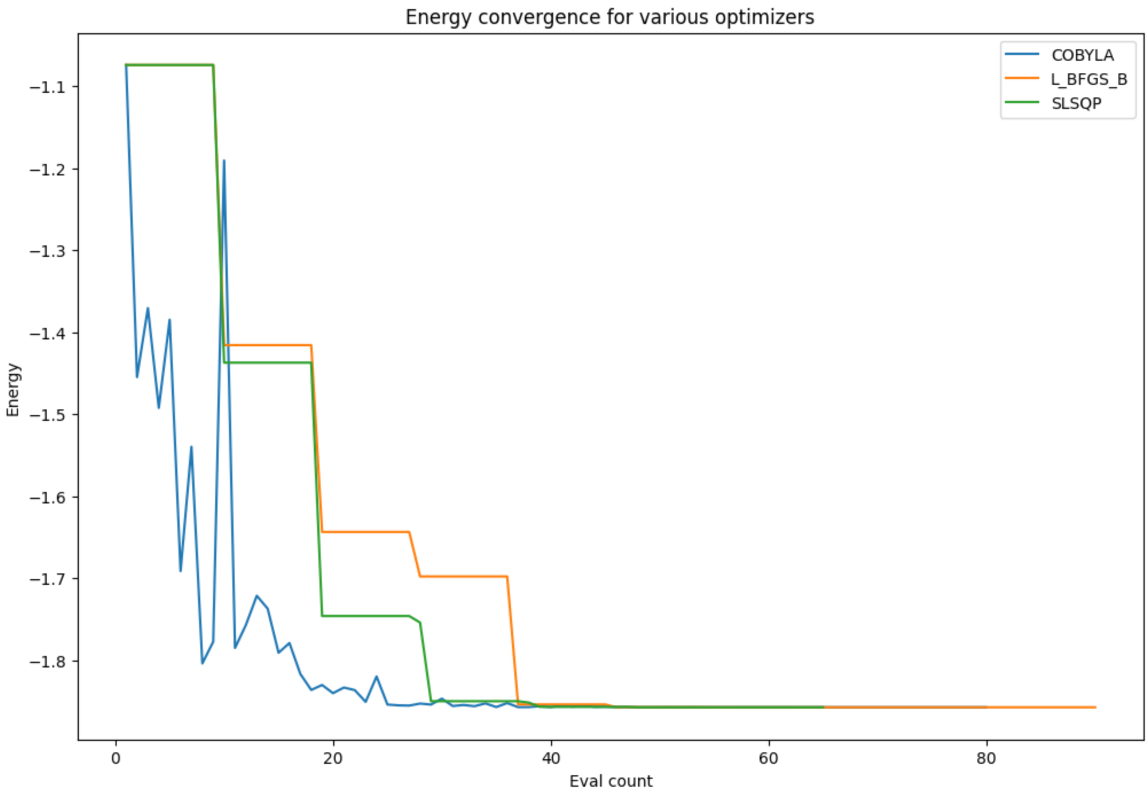
```
from qiskit_algorithms.optimizers import COBYLA, L_BFGS_B, SLSQP
from qiskit_algorithms.utils import algorithm_globals
from qiskit.circuit.library import TwoLocal
from qiskit.quantum_info import SparsePauliOp

H2_op = SparsePauliOp.from_list(
    [
        ("II", -1.052373245772859),
        ("IZ", 0.39793742484318045),
        ("ZI", -0.39793742484318045),
        ("ZZ", -0.01128010425623538),
        ("XX", 0.18093119978423156),
    ]
)
```

In [26]:
```
estimator = Estimator()


# we will iterate over these different optimizers
optimizers = [COBYLA(maxiter=80), L_BFGS_B(maxiter=60), SLSQP(maxit
converge_counts = np.empty([len(optimizers)], dtype=object)
converge_vals = np.empty([len(optimizers)], dtype=object)

for i, optimizer in enumerate(optimizers):
    print("\rOptimizer: {}        ".format(type(optimizer).__name__
    algorithm_globals.random_seed = 50
    ansatz = TwoLocal(rotation_blocks="ry", entanglement_blocks="cz"

    counts = []
    values = []

    def store_intermediate_result(eval_count, parameters, mean, std
        counts.append(eval_count)
        values.append(mean)

    vqe = VQE(estimator, ansatz, optimizer, callback=store_intermed
    result = vqe.compute_minimum_eigenvalue(operator=H2_op)
    #result = vqe.compute_minimum_eigenvalue(operator=observable_1)
    converge_counts[i] = np.asarray(counts)
    converge_vals[i] = np.asarray(values)

print("\rOptimization complete      ")
```

```
/var/folders/b6/5cpxpg110g9cj088x4tmqb1h0000gn/T/ipykernel_16762/245
694284.py:1: DeprecationWarning: The class ``qiskit.primitives.estim
ator.Estimator`` is deprecated as of qiskit 1.2. It will be removed
no earlier than 3 months after the release date. All implementations
of the `BaseEstimatorV1` interface have been deprecated in favor of
their V2 counterparts. The V2 alternative for the `Estimator` class
is `StatevectorEstimator`.
  estimator = Estimator()
Optimization complete
```

Now, from the callback data you stored, you can plot the energy value at each objective function call each optimizer makes. An optimizer using a finite difference method for computing gradient has that characteristic step-like plot where for a number of evaluations it is computing the value for close by points to establish a gradient

In [27]:
```
import pylab

pylab.rcParams["figure.figsize"] = (12, 8)
for i, optimizer in enumerate(optimizers):
    pylab.plot(converge_counts[i], converge_vals[i], label=type(opt
pylab.xlabel("Eval count")
pylab.ylabel("Energy")
pylab.title("Energy convergence for various optimizers")
pylab.legend(loc="upper right");
```



Discussion on the ansatz choice

To iteratively optimize from a reference state ρ to a target state $\psi(\theta)$, we need to define a variational form $U_V(\theta)$ that represents a collection of parametrized states for our variational algorithm to explore:

$$U_V(\theta)U_R|0>= U_V(\theta)\rho = U_A(\theta) = |\psi(\theta)>$$

Note that the parametrized state depends on both the reference state ρ, which does not depend on any parameters, and the variational form $U_V(\theta)$, which always depends on parameters. We refer to the combination of these two halves as an ansatz $U_A(\theta)$. As we construct our ansatz to represent a collection of parametrized states for our variational algorithm to explore, we realize an important issue: dimensionality. An n-qubit system (i.e., Hilbert space) has a vast number of distinct quantum states in the configuration space. We would require an unwieldy number of parameters to fully explore it. In addition, the runtime complexity of search algorithms, and others alike, grows exponentially with this dimensionality, a phenomenon often referred to in the literature as the curse of dimensionality.

To counter this setback, it is common practice to impose some reasonable constraints on the variational form such that only the most relevant states are explored. Finding an efficient truncated ansatz is an active area of research, but we'll cover two common designs.

Heuristic ansatze and trade-offs

If you do not have any information about your particular problem that can help restrict the dimensionality, you can try an arbitrary family of parameterized circuits with fewer than 2^{2n} parameters. However, there are some trade-offs to consider:

- Speed: By reducing the search space, the algorithm can run faster.
- Accuracy: However, reducing the space could risk excluding the actual solution to the problem, leading to suboptimal solutions.
- Noise: Deeper circuits are affected by noise, so we need to experiment with our ansatz's connectivity, gates, and gate fidelity.

N-Local circuit

One of the most widely used examples of heuristic ansatzes is the N-local circuits, for a few reasons:

- Efficient implementation: The N-local ansatz is typically composed of simple, local gates that can be implemented efficiently on a quantum computer, using a small number of physical qubits. This makes it easier to construct and optimize quantum circuits.
- Captures important correlations: The N-local ansatz can capture important correlations between the qubits in a quantum system, even with a small number of gates. This is because the local gates can act on neighboring qubits and create entanglement between them, which can be important for simulating complex quantum systems.

These circuits consist of rotation and entanglement layers that are repeated alternatively one or more times as follows:

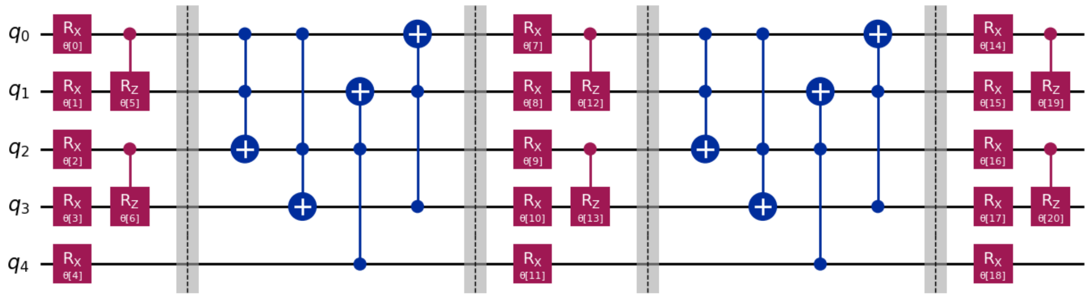
- Each layer is formed by gates of size at most N, where N has to be lower than the number of qubits.
- For a rotation layer, the gates are stacked on top of each other. We can use standard rotation operations (RX.RZ)
- For an entanglement layer, we can use gates like Toffoligates or CX with

an entanglement strategy.

Example: 5 qubits with Toffoli gate, which acts on three qubits, making the circuit `3-local`

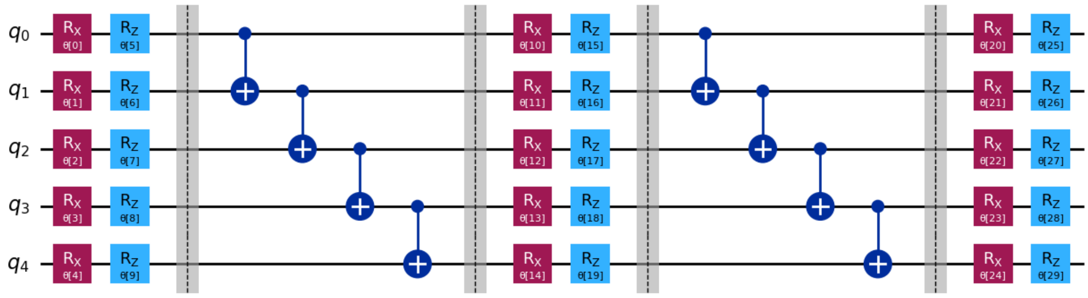
In [28]:
```
from qiskit.circuit.library import NLocal, CCXGate, CRZGate, RXGate
from qiskit.circuit import Parameter

theta = Parameter("θ")
ansatz = NLocal(
    num_qubits=5,
    rotation_blocks=[RXGate(theta), CRZGate(theta)],
    entanglement_blocks=CCXGate(),
    entanglement=[[0, 1, 2], [0, 2, 3], [4, 2, 1], [3, 1, 0]],
    reps=2,
    insert_barriers=True,
)
ansatz.decompose().draw("mpl")
```

Out[28]:



In [29]:
```
## Example using the standard 2-local circuits with single-qubit ro
from qiskit.circuit.library import TwoLocal

ansatz = TwoLocal(
    num_qubits=5,
    rotation_blocks=["rx", "rz"],
    entanglement_blocks="cx",
    entanglement="linear",
    reps=2,
    insert_barriers=True,
)
ansatz.decompose().draw("mpl")
```

Out[29]:

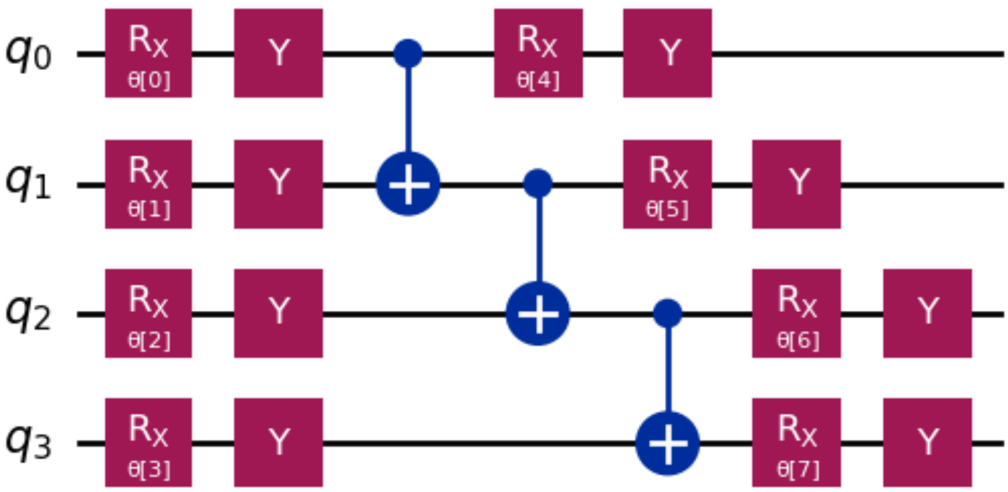


EfficientSU2

EfficientSU2 is a hardware-efficient circuit that consists of layers of single-qubit operations spanning $SU(2)$ and CX entanglements. This is a heuristic pattern that can be used to prepare trial wave functions for variational quantum algorithms or as a classification circuit for machine learning.

In [30]:
```
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(4, su2_gates=["rx", "y"], entanglement="linea
ansatz.decompose().draw("mpl")
```

Out[30]:



Extra on SU(2) ansatz

The `SU(2) ansatz` in quantum computing is deeply rooted in **group theory**, particularly in the representation theory of the special unitary group SU(2). Understanding this connection helps explain the mathematical structure behind the ansatz and why it's commonly used in variational quantum algorithms. Here's an expanded explanation of the connection:

SU(2) Ansatz and Its Connection to Group Theory

What is SU(2)?

- **Definition**:
  - $SU(2)$ is the group of all (2×2) unitary matrices with determinant (1).
  - Formally: $SU(2) = \{U \in \mathbb{C}^{2\times 2} : U^\dagger U = I, \det(U) = 1\}$.
  - Examples of $(SU(2))$ matrices include quantum gates such as $(R_X(\theta))$, $(R_Y(\theta))$, and $(R_Z(\theta))$, which represent rotations around the $(x$-, $y)$ -, and (z)-axes of the Bloch sphere.
- **Lie Algebra**:

- The generators of $(SU(2))$ are the **Pauli matrices** (X, Y, Z) scaled by $i/2$.
- Any element of $SU(2)$ can be expressed as: $U = e^{-i(\alpha X + \beta Y + \gamma Z)}$, where α, β, γ are real coefficients.

SU(2) Ansatz in Variational Algorithms

Structure

- The SU(2) ansatz leverages the fact that any $SU(2)$ operation can be expressed using parameterized rotations: $U = R_X(\theta_1)R_Y(\theta_2)R_Z(\theta_3)$.
- By stacking layers of these gates, the ansatz explores a parameterized subset of $SU(2)$, enabling optimization for specific problems.

Key Properties

1. **Universal Approximation**:

    - A combination of local $SU(2)$ operations (rotations) and entanglement gates is **universal**, meaning it can approximate any quantum operation on n-qubits to arbitrary precision.

2. **Symmetry-Constrained Problems**:

    - Many physical systems exhibit $SU(2)$-type symmetries (e.g., spin systems).
    - The SU(2) ansatz naturally respects these symmetries.

In [31]:
```
import pennylane as qml
from pennylane import numpy as np

n_qubits = 3
layers = 2
dev = qml.device("default.qubit", wires=n_qubits)

def su2_ansatz(params):
    for i in range(n_qubits):
        qml.RX(params[0][i], wires=i)
        qml.RY(params[1][i], wires=i)
        qml.RZ(params[2][i], wires=i)
    for i in range(n_qubits - 1):
        qml.CNOT(wires=[i, i + 1])

@qml.qnode(dev)
def circuit(params):
    for layer in range(layers):
        su2_ansatz(params[layer])
    return qml.state()
```

```python
params = np.random.uniform(0, 2 * np.pi, (layers, 3, n_qubits))
print("Final Quantum State:", circuit(params))
```

```
Final Quantum State: [-0.04330939+0.00197066j -0.00828894+0.08524238
j -0.05249925+0.06222108j
  0.14709074-0.0832011j  -0.33261814-0.37904314j  0.09459899+0.47519
876j
 -0.03768399+0.17113517j  0.65691275-0.06547249j]
```

In [ ]:

```python

```