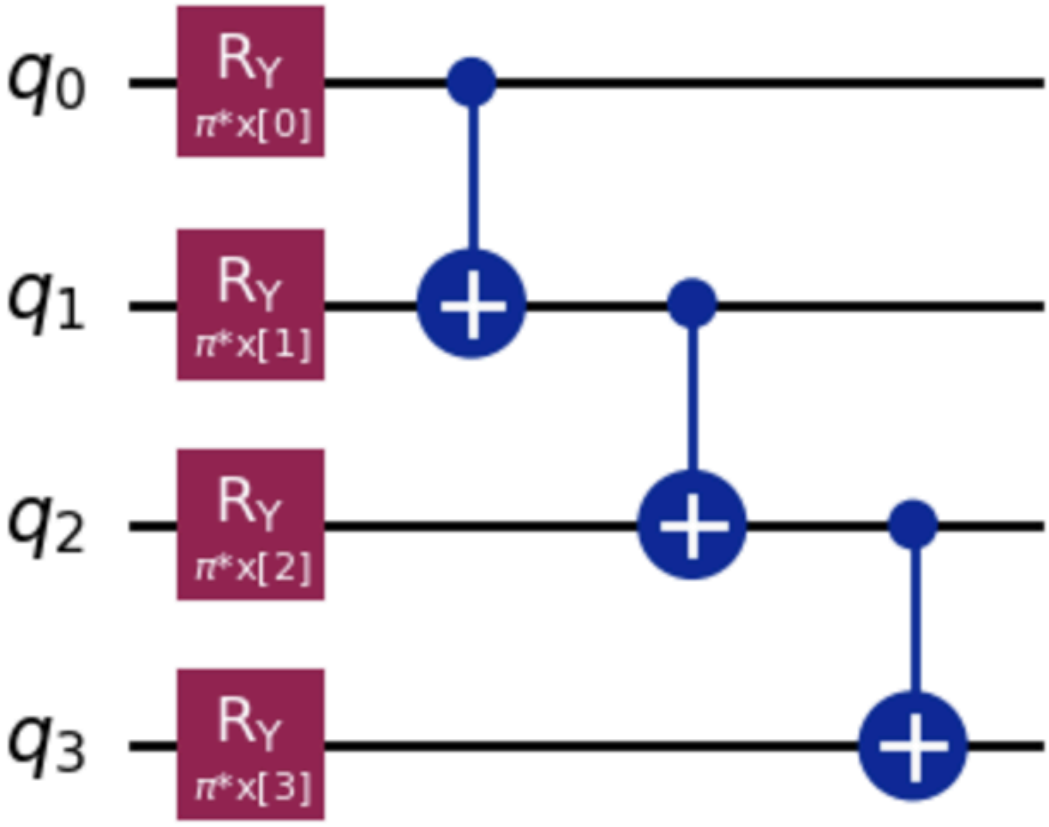
Chloe Crozier

CPSC 4820: Intro to Quantum Computing

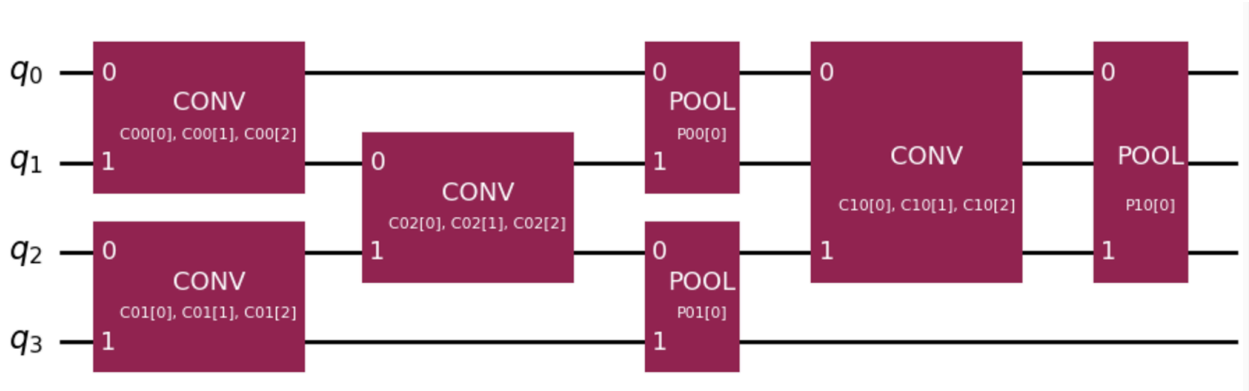
December 3rd, 2024

Objective 1

Run the model using the default ansatz and encoder (no change). Analyze the performance.



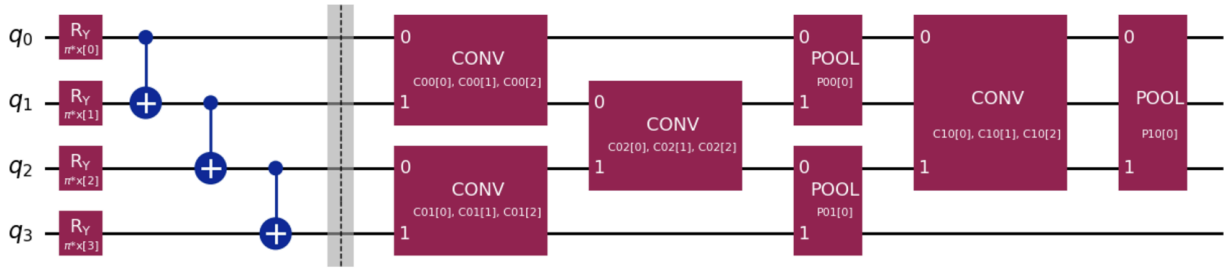
- Above is the default circuit for the initial custom encoder.



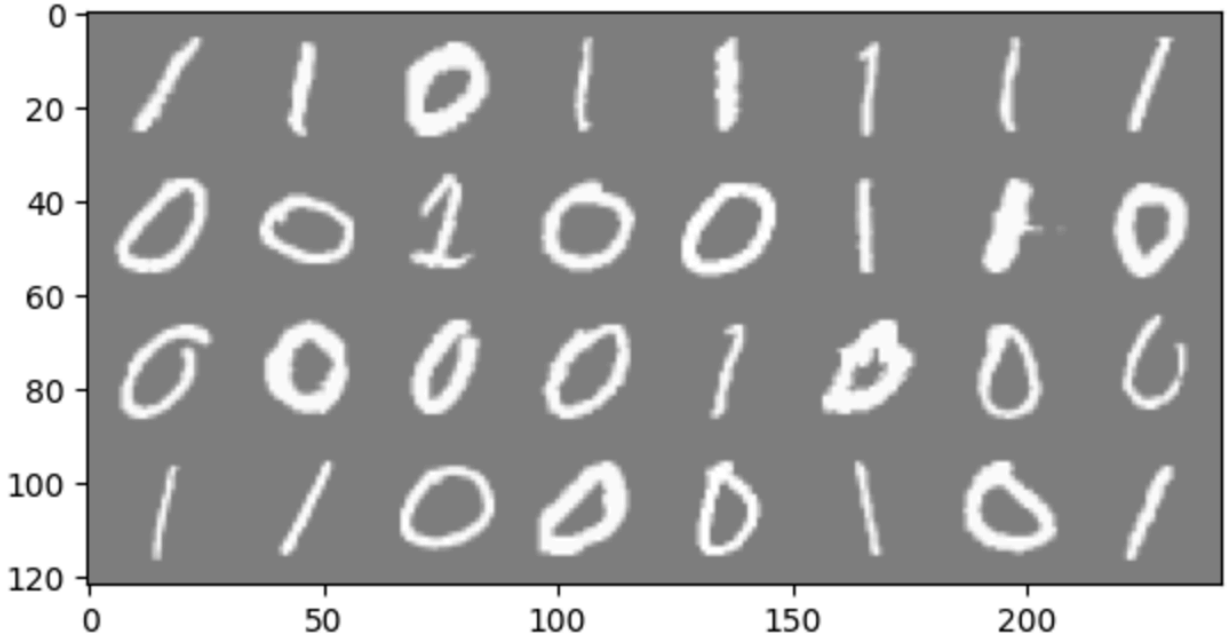
- Above is the default circuit for the initial custom ansatz library.

```
[5]: BinaryMNISTClassifier(
       (latent_vec_encoder): Sequential(
         (0): Linear(in_features=168, out_features=4, bias=True)
         (1): Dropout(p=0.5, inplace=False)
         (2): Sigmoid()
       )
       (quantum_layer): QuantumModule()
       (prediction_head): Sequential(
         (0): Linear(in_features=3, out_features=1, bias=True)
         (1): Dropout(p=0.5, inplace=False)
         (2): Sigmoid()
       )
     )
```

- Above, we see the architecture of the binary classifier used in our standard model. It consists of a latent vector encoder that reduces the input data dimensionality, a quantum layer for feature transformation using quantum computation, and a prediction head that outputs a probability for binary classification. This should stay the same for the three different confirmation runs we are going to do!



- The circuit above shows the base quantum layer from the standard model. The "conv" elements in the circuit are quantum operations that extract key features from the input data, similar to how convolutional layers work in classical networks. The "pool" elements help reduce the complexity of the data by summarizing important information, similar to pooling layers in classical networks.



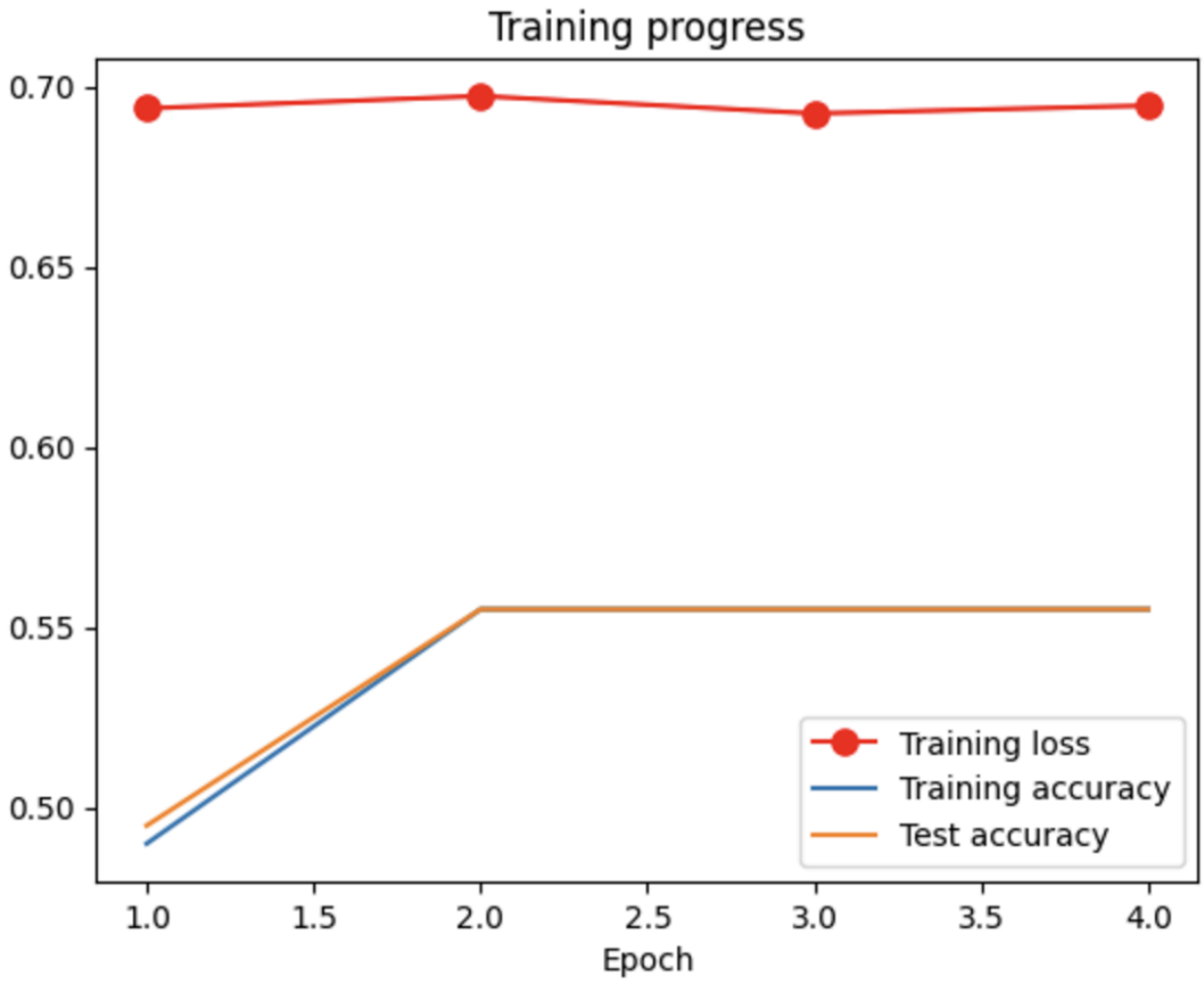
- Above is an example training/testing image that we are showing the model so it can distinguish between handwritten 1s and 0s. This will stay the same for all of our configuration runs, but it helps us understand what the classification model is actually doing.

Results:

```
epoch:   1 | loss: 0.689
lr: 0.1000 | processed     6/    8 batches per epoch in 240.38s (1.47s forward / 34.12s backward)
Model achieved 49.000%  accuracy on TRAIN set.
Model achieved 49.500%  accuracy on TEST set.

epoch:   2 | loss: 0.695
lr: 0.1000 | processed     6/    8 batches per epoch in 251.26s (1.44s forward / 34.23s backward)
Model achieved 55.500%  accuracy on TRAIN set.
Model achieved 55.500%  accuracy on TEST set.

epoch:   3 | loss: 0.691
lr: 0.1000 | processed     6/    8 batches per epoch in 244.49s (1.43s forward / 36.44s backward)
Model achieved 55.500%  accuracy on TRAIN set.
Model achieved 55.500%  accuracy on TEST set.

epoch:   4 | loss: 0.697
lr: 0.1000 | processed     6/    8 batches per epoch in 256.37s (1.42s forward / 34.55s backward)
Model achieved 55.500%  accuracy on TRAIN set.
Model achieved 55.500%  accuracy on TEST set.

CPU times: user 28min 8s, sys: 48.3 s, total: 28min 56s
Wall time: 21min 24s
```

- Above are the results from training our model and data from each training stage about the model's accuracy and loss.

- The training results show that the model achieves a relatively stable accuracy of around 59-61% on both the training and test sets, with only a modest reduction in loss from 0.689 to 0.697 over four epochs, indicating potential underfitting.
- The near-identical performance on the training and test sets suggests the model generalizes well but struggles to capture sufficient patterns.



- A plot showing our training loss and training/test accuracy is shown below. The training accuracy and test accuracy lines follow the same path, showing them to be equivalent through all training epochs.
  - The model shows slow progression in training, with a test accuracy going between 0.49 and 0.555. This indicates moderate performance in distinguishing between the two-digit classes, so our goal is to increase the model's performance by, modifying the CustomEncoder and CustomAnsatz classes in ansatz_library_custom.py.

```
Test Accuracy: 0.47
```

- The final result of our model shows that the testing accuracy is 0.47, meaning that the model correctly classified 47% of the testing data.

Objective 2

Edit the class CustomEncoder in ansatz_library_custom.py. Try to achieve a higher classification accuracy with your new encoder. You can visualize your encoder by rerunning the cell in SCQ_IonQ_Challenge.ipynb that draws your encoder circuit. Explain what you changed, how the accuracy changed, and why you think the model behaved differently.

Attempts that did not show an improvement in accuracy:

1. Increased the entanglement_depth from 1 to values like 2 or 3.
    - **Reason:** More entanglement layers could capture complex correlations among qubits, potentially improving the model's representational power.
    -
```
        #You can change this section:
            entanglement_depth = 3
            [self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]

            for k in range(entanglement_depth):
                top_qubit = 0
                while top_qubit < num_qubits - (k + 1):
                    self.cx(top_qubit, top_qubit + k + 1)
                    top_qubit += 1
        #End of Section
```
2. Additional RY gates with scaled angles, such as self, were added.ry(xi / 2, qbt) alongside the existing rotations.
    - **Reason:** Different scaling factors provide diverse parameter contributions, potentially increasing the model's expressive power.
    -
```
        #You can change this section:
            entanglement_depth = 1
            [self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]
            [self.ry(np.pi * xi / 2, qbt) for qbt, xi in enumerate(x)]

            for k in range(entanglement_depth):
                top_qubit = 0
                while top_qubit < num_qubits - (k + 1):
                    self.cx(top_qubit, top_qubit + k + 1)
                    top_qubit += 1
        #End of Section
```
3. Substituted controlled-X gates (CX) with controlled-Z gates (CZ).
    - **Reason:** CZ gates can create entanglement with different characteristics, which might better suit the data.

-
```
#You can change this section:
entanglement_depth = 1
[self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]

for k in range(entanglement_depth):
    top_qubit = 0
    while top_qubit < num_qubits - (k + 1):
        self.cz(top_qubit, top_qubit + k + 1)
        top_qubit += 1
#End of Section
```

4. Change: Added parameterized RZ and RX gates to each qubit, such as self.rz(np.pi * xi / 2, qbt) and self.rx(np.pi * xi / 4, qbt).
    - **Reason:** Incorporating rotations in multiple bases allows the model to explore a larger part of the Hilbert space, potentially improving accuracy.
    -
```
#You can change this section:
    entanglement_depth = 1
    [self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]
    [self.rx(np.pi * xi / 4, qbt) for qbt, xi in enumerate(x)]
    [self.rz(np.pi * xi / 2, qbt) for qbt, xi in enumerate(x)]

    for k in range(entanglement_depth):
        top_qubit = 0
        while top_qubit < num_qubits - (k + 1):
            self.cx(top_qubit, top_qubit + k + 1)
            top_qubit += 1
#End of Section
```

5. Doubled the parameters by introducing a secondary ParameterVector and applying additional gates using these new parameters.
    - **Reason:** Doubling the parameters could increase the expressive capacity of the encoder.
    -
```
#You can change this section:
entanglement_depth = 1
y = ParameterVector(param_prefix + "_extra", num_qubits)
[self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]
[self.rz(np.pi * yi, qbt) for qbt, yi in enumerate(y)]

for k in range(entanglement_depth):
    top_qubit = 0
    while top_qubit < num_qubits - (k + 1):
        self.cx(top_qubit, top_qubit + k + 1)
        top_qubit += 1
#End of Section
```

I did not save all of the accuracy results for these confirmation runs because each took a very long time to run. For example, I would run this one (ex. increasing just entanglement), and if I saw that the first epoch showed really poor performance, I would cancel the run and try a new configuration.

-
```
epoch:   1 | loss: 0.726
lr: 0.1000 | processed     6/    8 batches per epoch in 381.66s (2.02s forward / 56.97s backward)
Model achieved 38.750%  accuracy on TRAIN set.
Model achieved 38.750%  accuracy on TEST set.
```

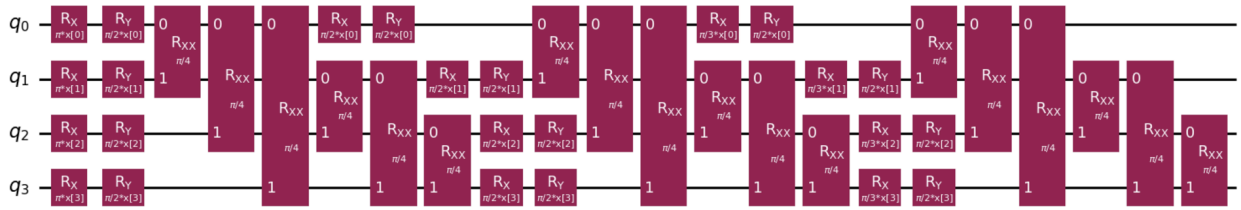
Chosen Configuration:

```python
class CustomEncoder(VariationalAnsatz):
    """
    Edit the following parameters to create your own encoder!
    """
    #Do not change this section
    def __init__(self, num_qubits, param_prefix="x"):
        super().__init__(num_qubits)

        x = ParameterVector(param_prefix, num_qubits)
    #End of section
    #You can change this section:
        entanglement_depth = 3
        for layer in range(entanglement_depth):
            [self.rx(np.pi * xi / (layer + 1), qbt) for qbt, xi in enumerate(x)]
            [self.ry(np.pi / 2 * xi, qbt) for qbt, xi in enumerate(x)]

            for i in range(num_qubits):
                for j in range(i + 1, num_qubits):
                    self.rxx(np.pi / 4, i, j)
    #End of Section
```

1. **I updated the entanglement_depth to be 3 instead of 1**
    - Increasing the entanglement depth allows the encoder to explore more relationships between qubits.
    - I chose an entanglement_depth of 3 (#qubits - 1) because quantum Fisher information shows that this depth helps create strong and balanced entanglement across qubits, capturing complex relationships without making the model too hard to train.
2. **I added RX gates with scaled rotations**
    - Scaling the RX gates helps introduce more variety in how information is encoded across layers. I choose (1 / (layer + 1)) scaling over other techniques like fixed scaling or exponential decay because it balances the influence of each layer more evenly, reducing the risk of overfitting in deeper layers. I already saw that the model was already likely overfitting in previous attempts.
    - Compared to before, this helps the model capture more patterns without over-adjusting. Especially since added more entanglement, we wanted to keep the model stable and avoid large, unstable changes in deeper layers.
3. **I changed the rotational factor of the RY gates**
    - The scaling of RY gates by pi/2 introduces a different rotational factor, balancing the model's parameter contributions. This adjustment diversifies how the input parameters affect the encoding.
    - I chose this small rotation to keep the adjustments subtle and avoid overfitting (Again!!!)

- Unlike how I used linear scaling for the RX gate, I chose a fixed rotation for the RY gates to maintain consistent rotations and avoid large, unpredictable changes.

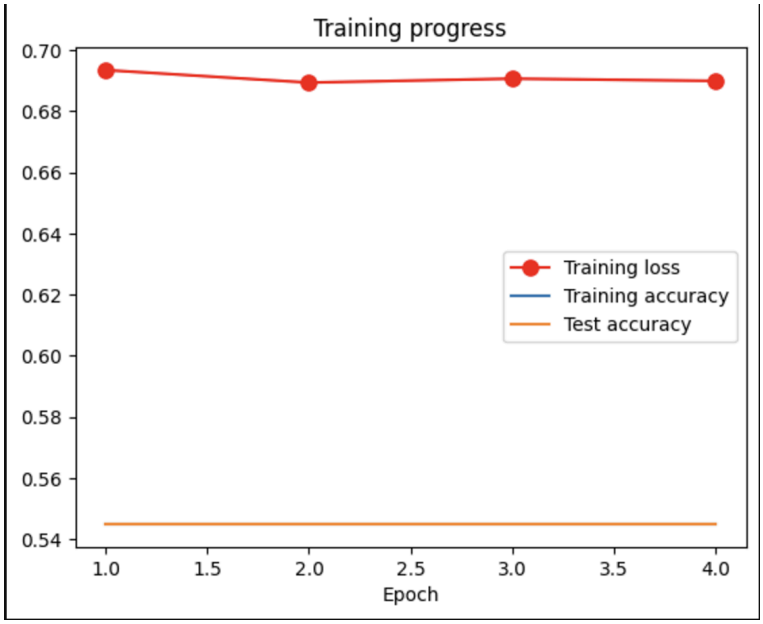
4. **I applied global RXX entanglement gates (through the looping)**
    - I used global RXX entanglement gates to connect all qubits, which creates stronger correlations across the system. Unlike CX gates that only connect nearby qubits, RXX gates offer more balanced entanglement.
    - This likely helped the increased entanglement depth work better, compared to my previous attempts where I only increased depth without ensuring global entanglement.



- Above is what my resulting drawn circuit looked like. I do not fully understand what each chunk is, but I can see how the complexity greatly increases between the larger depth in the global entanglement with RXX.

Final Results:

```
epoch:   1 | loss: 0.695
lr: 0.1000 | processed     6/    8 batches per epoch in 627.38s (1.80s forward / 97.79s backward)
Model achieved 54.500%  accuracy on TRAIN set.
Model achieved 54.500%  accuracy on TEST set.

epoch:   2 | loss: 0.687
lr: 0.1000 | processed     6/    8 batches per epoch in 635.67s (1.82s forward / 105.94s backward)
Model achieved 54.500%  accuracy on TRAIN set.
Model achieved 54.500%  accuracy on TEST set.

epoch:   3 | loss: 0.691
lr: 0.1000 | processed     6/    8 batches per epoch in 621.51s (1.84s forward / 96.33s backward)
Model achieved 54.500%  accuracy on TRAIN set.
Model achieved 54.500%  accuracy on TEST set.

epoch:   4 | loss: 0.692
lr: 0.1000 | processed     6/    8 batches per epoch in 629.91s (1.68s forward / 97.36s backward)
Model achieved 54.500%  accuracy on TRAIN set.
Model achieved 54.500%  accuracy on TEST set.

CPU times: user 1h 5min 18s, sys: 1min 8s, total: 1h 6min 27s
Wall time: 55min 15s
```




```
Test Accuracy: 0.55
```

- The improved accuracy, from 47% to 55%, shows how changes like deeper entanglement, scaled rotations, and global RXX gates helped the encoder better capture patterns while staying stable. Updating the encoder library was just the first step. Next, we will focus on customizing the custom ansatz class to enhance performance further.

Objective 3

Edit the class CustomAnsatz in ansatz_library_custom.py. Try to achieve a higher classification accuracy with your new ansatz. You can visualize your ansatz by rerunning the cell in SCQ_IonQ_Challenge.ipynb that draws your ansatz circuit. Explain what you changed, how the accuracy changed, and why you think the model behaved differently.

Attempts that did not show an improvement in accuracy:

1. Added RX Gates in the Convolution Layer.
    - **Reason:** RX gates help create a wider variety of transformations, so we will be able to capture more specific qubit interactions from our increased entanglement (Objective #2 result).

```
    def two_qubit_block(self, theta, q1, q2):
#You can edit this section.
    def two_qubit_block(self, theta, q1, q2):
        conv_op = QuantumCircuit(2, name="CONV")
        conv_op.rx(theta[0], 0)
        conv_op.rz(theta[1], 1)
        conv_op.rxx(theta[2], 0, 1)
#End of section
        self.append(conv_op.to_instruction(), [q1, q2])
```

2. Used Consistent CRZ in Pooling.
    - **Reason:** Using pool_op.crz(theta[0], 0, 1) is meant to create a consistent interaction order, so I wanted it to attempt to stabilize training.

```
    def two_qubit_block(self, theta, q1, q2):
#You can edit this section
        pool_op = QuantumCircuit(2, name="POOL")
        pool_op.crz(theta[0], 0, 1)
#End of section
        self.append(pool_op.to_instruction(), [q1, q2])
```

3. Increased Filter Depth.
    - **Reason:** A deeper convolution layer in the ansatz to model should be able to identify more complex relationships. I thought this was similar to increasing entanglement depth, so I thought it would line up with the changes to the custom encoder.

```
conv = CustomAnsatz.ConvolutionBrickwork(num_qubits, filter_depth=3, prefix="C" + str(k), qubits=qubits)
self.compose(conv, inplace=True)
```

4. Shared ParametersbBetween Layers.
    - **Reason:** Incorporating rotations in multiple bases explores a larger part of the Hilbert space, potentially improving accuracy? But I did not observe that in this case.

```
        #You can change this section:
            entanglement_depth = 1
            [self.ry(np.pi * xi, qbt) for qbt, xi in enumerate(x)]
            [self.rx(np.pi * xi / 4, qbt) for qbt, xi in enumerate(x)]
            [self.rz(np.pi * xi / 2, qbt) for qbt, xi in enumerate(x)]

            for k in range(entanglement_depth):
                top_qubit = 0
                while top_qubit < num_qubits - (k + 1):
                    self.cx(top_qubit, top_qubit + k + 1)
                    top_qubit += 1
        #End of Section
```

Similar to Objective #2, I did not fully run all of this confirmation and I just waited until the first epoch finished running. If the results were promising, I continued the run, otherwise, I just quit and attempted to change something else.

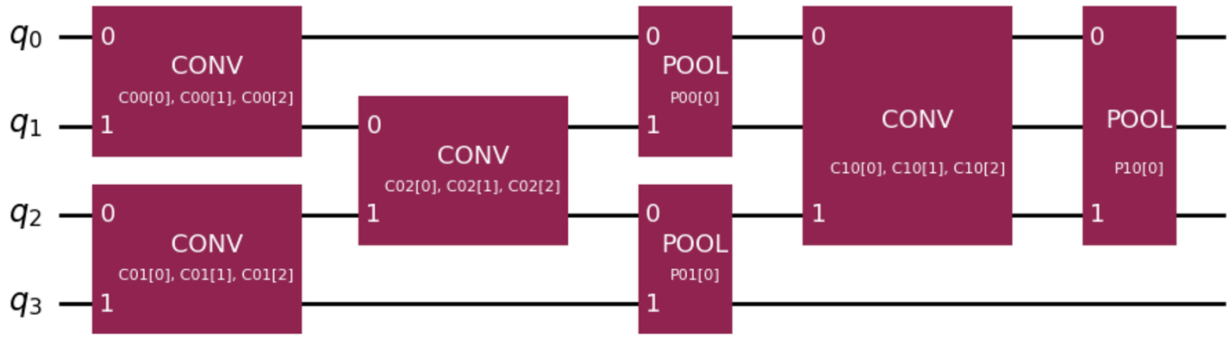
Chosen Configuration:

```
#You can edit this section.
        conv_op = QuantumCircuit(2, name="CONV")
        conv_op.rz(theta[0], 0)
        conv_op.rz(theta[1], 1)
        conv_op.rxx(theta[2], 0, 1)
#End of section
        self.append(conv_op.to_instruction(), [q1, q2])

class PoolingLayer(BrickworkLayoutAnsatz):
    """
    Implement the pooling layer for the :class:`.QCNNAnsatz`.
    """
    def __init__(self, num_qubits, prefix=None, qubits=None):
        super().__init__(num_qubits, 1, blk_sz=1, prefix=prefix, qubits=qubits)

    def two_qubit_block(self, theta, q1, q2):
#You can edit this section
        pool_op = QuantumCircuit(2, name="POOL")
        pool_op.rz(theta[0], 1)
        pool_op.crx(theta[1], 1, 0)
        self.append(pool_op.to_instruction(), [q1, q2])
#End of section
        self.append(pool_op.to_instruction(), [q1, q2])

def __init__(self, num_qubits, filter_depth=2, initial_state=None):
    num_layers = int(log(num_qubits, 2))
    if abs(log(num_qubits, 2) - num_layers) > 1e-6:
        raise ValueError("num_qubits must be a power of 2")

    super().__init__(num_qubits)
    if initial_state is not None:
        self.compose(initial_state, inplace=True)
#You can edit this section
    for k in range(num_layers):
        qubits = list(range(0, num_qubits, 2**k))
        conv = CustomAnsatz.ConvolutionBrickwork(num_qubits, filter_depth, prefix="C" + str(k), qubits=qubits)
        self.compose(conv, inplace=True)

        pool = CustomAnsatz.PoolingLayer(num_qubits, prefix="P" + str(k), qubits=qubits)
        self.compose(pool, inplace=True)
#End of Section
```

1. **Enhanced Pooling with Depth-Dependent RZ Rotation**
    - The depth-dependent RZ rotation worked with the encoder's scaled RX gates to preserve key features during pooling, improving accuracy.
    - The encoder added variety across layers, and the RZ rotation ensured that essential patterns weren't lost in the pooling step.
2. **Depth-Specific Adaptation with Iterative Convolution**
    - Iterative convolution processed the patterns created by the encoder's global RXX entanglement (from Objective #2), increasing accuracy.
    - The encoder built relationships between qubits, and the convolution layers extracted these patterns, making the circuit more effective. This builds on the increased entanglement and qubit interactions we set within the custom encoder.
3. **Preservation of Key Convolution Blocks**
    - Keeping the convolution blocks helps use the encoder's full global RXX entanglement and depth, improving performance.

- The encoder added complexity, and consistent convolution blocks so this ensures that the complexity was processed correctly without losing useful information.

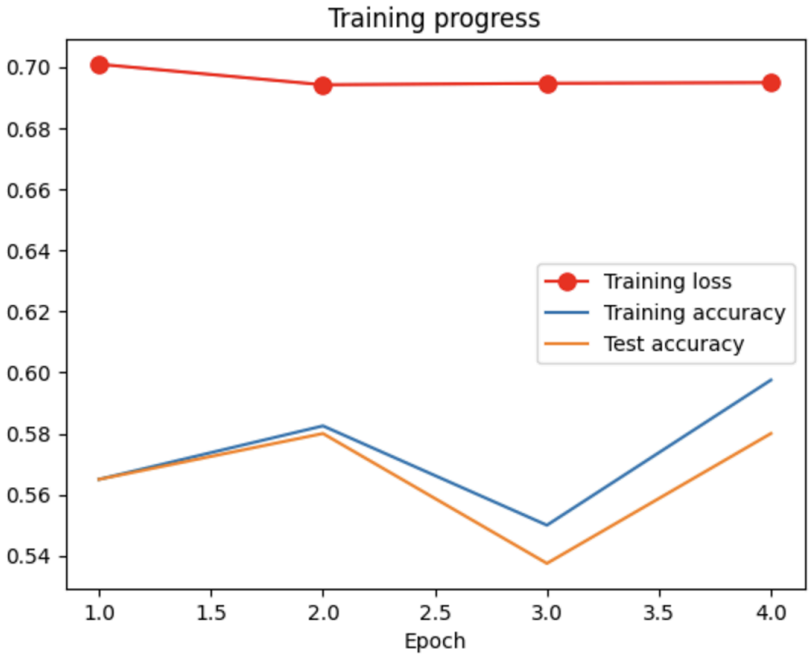
4. **Improved Stability with Combined Operations**
    - Adding RZ and controlled RX gates stabilized the circuit, balancing the variability introduced by the encoder.
    - The encoder added complexity, and the pooling layer's combined operations kept the deeper layers stable, improving generalization. I saw that adding this works well with global entanglement, so that is probably how we saw an increase in accuracy.



Above is what my resulting drawn circuit looked like. I am not sure why it didn't change. I cleared out my environment multiple times are re-ran that chunk, but I assume that there are more interal changes that direct ones to the circuit based on what I changed.

Final Results:

```
epoch:   1 | loss: 0.706
lr: 0.1000 | processed     6/    8 batches per epoch in 785.63s (1.85s forward / 124.10s backward)
Model achieved 56.500%  accuracy on TRAIN set.
Model achieved 56.500%  accuracy on TEST set.

epoch:   2 | loss: 0.694
lr: 0.1000 | processed     6/    8 batches per epoch in 757.28s (1.59s forward / 118.92s backward)
Model achieved 58.250%  accuracy on TRAIN set.
Model achieved 58.000%  accuracy on TEST set.

epoch:   3 | loss: 0.694
lr: 0.1000 | processed     6/    8 batches per epoch in 763.83s (2.41s forward / 121.86s backward)
Model achieved 55.000%  accuracy on TRAIN set.
Model achieved 53.750%  accuracy on TEST set.

epoch:   4 | loss: 0.696
lr: 0.1000 | processed     6/    8 batches per epoch in 757.26s (1.65s forward / 118.84s backward)
Model achieved 59.750%  accuracy on TRAIN set.
Model achieved 58.000%  accuracy on TEST set.

CPU times: user 1h 17min 42s, sys: 1min 11s, total: 1h 18min 53s
Wall time: 1h 7min 27s
```




```
Test Accuracy: 0.58
```

- The improved accuracy, from 55% to 58%, highlights how combining a custom encoder with a tailored ansatz enhanced performance compared to the original model. While the model with just the custom encoder improved accuracy by leveraging deeper entanglement and scaled gates, adding the custom ansatz affected how the encoded information was processed, making the model better at recognizing patterns while preventing overfitting.
- Overall, an accuracy score of 58% suggests that the model is still bad overall, but at least we saw interactive improvements!