
Enhancing the Quantum Approximation Optimization Algorithm using Graph Convolutional Networks for Ising Models

Rebecca Tsekanovskiy, Micheal Halpern
Rensselaer Polytechnic Institute
Troy, NY
Tsekar@rpi.edu, Halpem2@rpi.edu

Abstract

1 The Ising Spin Model is a fundamental concept in the study of Spin Glass systems,
2 which explores particle interactions. Our research focuses on solving the Ising
3 Model using the Quantum Approximation Optimization Algorithm. The Quantum
4 Approximation Optimization Algorithm approximates Hamiltonian minimization,
5 allowing it to find the ground state efficiently. We implemented a graph convolu-
6 tional network to predict optimal γ and β values alongside a custom created loss
7 function to analyze the expected value for a max cut based off the predictions.
8 Traditional evaluation metrics do not account for expected value requirements and
9 using a custom loss function allows for a direct comparison between known γ and
10 β values. Ultimately, by utilizing a graph convolutional network, we improved the
11 calculation of the γ and β values by 99.9% compared to a classical approach, with
12 only a 1.2% reduction in expected value.

13 1 Background

14 1.1 Ground State of the Ising Model

15 The Ising Spin Model is a critical component in the study of Spin Glass systems, which are concerned
16 with the interactions of particles. In this model, each spin, denoted as σ_i , can take one of two values:
17 +1 or -1, where σ_i represents the i-th spin. Particles with a spin quantum number of -1 have a spin
18 orientation that is opposite to that of particles with a spin quantum number of +1 [5]. The term w_{ij}
19 signifies the interaction between neighboring spins and their strength. It holds a value of zero for
20 non-neighboring spins and -1 if and only if spins i and j are connected [5]. Each spin σ_i contributes
21 its energy to the Hamiltonian of the spin system. This contribution is defined as follows [3]:

$$H(\sigma) = - \sum_{i < j} w_{ij} \sigma_i \sigma_j \quad (1)$$

22 This equation represents the Hamiltonian of the system, which is a measure of the total energy of the
23 system. The ground state of the Ising model corresponds to the configuration of spins that minimizes
24 this Hamiltonian and thus minimizes the total energy of the system. Finding the ground state of an
25 Ising spin model can be mapped to a two-dimensional graph, where each particle can be represented
26 as a node. Particles with spin quantum numbers of +1 and -1, which are neighbors, share an edge in
27 the graph. Minimizing the energy of the physical system and finding it's ground state is equivalent to
28 finding the max-cut of this graph. An example of mapping the Ising spin model to a two-dimensional
29 graph and finding the graph's max-cut is 1.

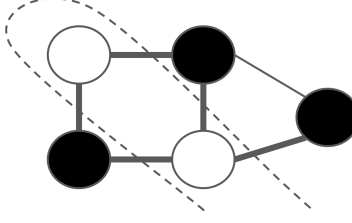


Figure 1: Example: A graph with five nodes and six edges. The dashed line represents the max-cut of this graph: the empty nodes are in set V_1 and the colored nodes are in set V_2 . There are other max-cuts, but in this graph the max-cut is five.

1.2 Quantum Approximation Optimization Algorithm

The Quantum Approximation Optimization Algorithm (QAOA) is used for many different types of optimization problems in Quantum Computing, more specifically, the QAOA can generate an approximate Hamiltonian minimization and thus in turn approximately minimize the system's energy and find the approximate ground state of the Ising Model [5]. This is represented as 1 where $\mathbf{b} \in \{0, 1\}^n$ [5].

$$C(\mathbf{b}) = \sum_{\alpha=1}^m C_{\alpha}(\mathbf{b}) \quad (2)$$

Equations 1 and 2 are closely related, as 1 represents the Hamiltonian that 2 is approximately minimizing. By mapping the Ising system model to the max-cut problem, we can define the QAOA by the following steps [1]:

1. First, a qubit is allocated for each node in the graph. All qubits are initialized to a state that is an equal superposition.
2. Then, by applying the $U(H_c, \gamma)$, otherwise known as the Hamiltonian cost 3, we are able to calculate the cut value for the given graph for an angle γ .
3. By applying the $U(H_c, \beta)$, otherwise known as the Hamiltonian mixer 4, we are able to modify the current states of the qubits based on the cost calculated in step 3 for an angle β .
4. We will repeat the previous two steps, for a total of p times with different parameters γ_i and β_i , ultimately, creating new states. p is defined as the depth chosen for the algorithm.
5. Based off the function of the depth, $F_p(\gamma, \beta)$, we calculate the expected Hamiltonian cost after running the QAOA algorithm for a given depth p .

1.2.1 Hamiltonian Cost Function and Hamiltonian Mixer Function

The Hamiltonian Cost Function 3 acts on qubits i, j which are the edges in our graph and w_{ij} . This function applies Pauli-z operators on both nodes i and j rotating the qubits about the Z-axis for a given rotation γ . This function changes the qubits states based on their edge connections to other qubits [3].

$$H_C = \frac{1}{2} \sum_{\langle i, j \rangle} w_{ij} \left(I - \sigma_z^{(i)} \sigma_z^{(j)} \right) \quad (3)$$

The Hamiltonian mixer function 4 acts on all qubits in the circuit. A Pauli-X gate is applied, rotating the qubits about the X-axis for a given rotation β . This rotation results in mixing the computational state, changing the probability of receiving a positive or negative state when measuring the qubit [3].

$$H_B = \sum_{j=1}^n \sigma_x^{(j)} \quad (4)$$

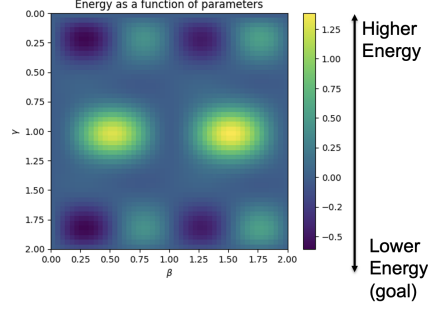


Figure 2: The visual representation of the γ and β value search space for a given graph. In the example, 25 beta and 25 gamma values are calculated making the size of the search space 2500. The darker regions represent γ and β values associated with lower energy states and therefore more promising γ and β values, while the lighter regions represent γ and β values associated with higher energy states [7].

57 1.2.2 Finding Gamma and Beta Values

58 To find suitable γ and β values, we can search the two-dimensional solution space of those values.
 59 To search the solution space, we can repeatedly calculate the expected value produced by running
 60 the QAOA algorithm at depth one for a given set of γ and β values 2. Searching the solution space
 61 this way allows us to find γ and β values which produce low amounts of energy after running QAOA
 62 at depth one. These low energy values are likely to lead to lower states of energy when the QAOA
 63 algorithm is run at a higher depth, producing a low energy state [3].

64 1.3 Graph Convolutional Networks

65 Graph Convolutional Networks (GCNs), a type of deep learning model operates specifically in a
 66 non-Euclidean space, meaning these models best understand data in a graph like structure, like trans-
 67 portation and social networks [8]. GCNs cause each node to continuously update its representation
 68 based off the neighboring nodes around. This iterative process makes a GCN an effective choice for
 69 capturing complex relationships in Ising spin models. We use GCNs to help find the most optimal γ
 70 and β values for the QAOA algorithm. By using a GCN model, hidden patterns in the graph structure
 71 are better understood and analyzed [9].

72 2 Related Works

73 This section disucssess two studies focused on utilizing graph neural networks (GNNs) with the
 74 QAOA algorithm.

75 2.1 Graph Neural Network Initialization of Quantum Approximate Optimization Algorithm

76 In this study, Jain et al. propose an innovative approach to initializing QAOA for solving the
 77 Max-Cut problem using GNNs The research addresses two key challenges: how to effectively
 78 initialize QAOA and how to optimize its parameters to achieve the best possible solution [2]. The
 79 authors demonstrate that integrating GNNs as a warm-start technique significantly enhances QAOA's
 80 performance compared to using QAOA or GNNs independently. Additionally, they show that GNNs
 81 provide warm-start generalization across various graph instances and sizes, an advantage not readily
 82 available with other methods [2].

83 The methodology proposed involves using GNNs to initialize the QAOA's parameter space close
 84 to an optimal solution, thereby reducing the quantum resources needed for optimization [2]. The
 85 authors compare their GNN-based initialization method to other approaches, such as those using
 86 semi-definite programming (SDP) relaxations and Trotterized quantum annealing. Through extensive
 87 numerical experiments, they demonstrate that the GNN-augmented QAOA consistently outperforms
 88 these traditional methods, particularly in scenarios where the depth of the quantum circuit is limited,
 89 a common constraint in noisy intermediate-scale quantum (NISQ) devices [2]. Further, the study

90 highlights the scalability and generalization capabilities of GNNs when used for QAOA initialization.
91 Unlike other methods that may require new problem instances for each new graph, the GNN approach
92 generalizes well across different graph sizes, making it a versatile tool for various combinatorial
93 optimization problems [2]. The research shows that even when trained on smaller graphs, the GNNs
94 can effectively warm-start QAOA for larger and more complex graphs, maintaining high solution
95 quality with reduced computational overhead [2].

96 In summary, Jain et al.’s work provides a compelling case for the integration of classical machine
97 learning techniques, such as GNNs, with quantum algorithms like QAOA [2]. Their findings
98 suggest that this hybrid approach can significantly enhance the efficiency and scalability of quantum
99 optimization algorithms, particularly in solving NP-hard problems like Max-Cut. The study paves
100 the way for future research into the combination of classical and quantum computing techniques to
101 tackle complex optimization challenges.

102 **2.2 Graph Learning for Parameter Prediction of Quantum Approximate Optimization** 103 **Algorithm**

104 The primary objective of this work is to enhance QAOA’s performance by optimizing its initialization
105 process, which is crucial for efficient execution on NISQ devices. These devices, while offering a
106 promising platform for quantum computations, are limited by their relatively small number of qubits
107 and susceptibility to noise. By leveraging the strengths of GNNs, the researchers aim to provide a
108 well-informed starting point for the QAOA parameters, thereby reducing the quantum computational
109 overhead and improving the algorithm’s convergence rates.

110 The researchers focus on training four distinct GNN architectures—Graph Isomorphism Networks
111 (GIN), Graph Convolutional Networks, Graph Attention Networks, and GraphSage, each selected for
112 their unique strengths in capturing and processing graph-structured data. First, a synthetic dataset
113 is created varying from sizes of 2 to 15. The methodology begins with data preprocessing, where
114 the node degrees and one-hot encoding of node IDs are computed to serve as node features [4].
115 Once the dataset is prepared, it is fed into the various GNN architectures. Each GNN model is
116 tasked with learning to predict the QAOA parameters, γ and β [4]. The GNNs perform inference
117 by iteratively aggregating feature information from each node’s local neighborhood. This involves a
118 message-passing framework where, at each layer, a node’s representation is updated based on the
119 aggregated information from its neighbors.

120 For example, in the GCN model, the node representations are updated by applying a combination of
121 linear transformations followed by a non-linear activation function. In contrast, GATs introduce an
122 attention mechanism that assigns different weights to different nodes in the neighborhood, allowing
123 the model to focus on the most relevant nodes during the aggregation process. The GraphSAGE
124 model, on the other hand, optimizes for neighborhood sampling, allowing the GNN to efficiently
125 handle graphs with large or varying node degrees [4]. After training, the GNNs output the initial
126 estimates for γ and β , which are then used to warm-start the QAOA. This means that the QAOA
127 does not begin its search from a random initialization but from a well-informed starting point, thus
128 potentially avoiding poor local minima and improving the overall optimization process [4].

129 Liang et al. demonstrate that their GNN-based initialization not only reduces the computational burden
130 on quantum devices but also exhibits strong generalization capabilities, successfully applying the
131 model to larger, unseen graphs [4]. This scalability is particularly important for practical applications
132 of quantum algorithms on NISQ devices.

133 **3 Motivation**

134 Quantum computing deals with optimizing problems like Ising Model, a known NP-complete problem.
135 Solving the Ising model enables the solution of all other NP-complete problems [6]. A solution to
136 The Ising Model can be found by using QAOA algorithms in linear time, but finding optimal β and
137 γ parameters requires sampling a solution space. This research aims to optimize solving the Ising
138 Model on the Rensselaer Polytechnic Institute (RPI)-IBM Quantum System One, to understand the
139 challenges and limitations, and use machine learning algorithms to increase the efficiency of finding
140 β and γ parameters for a given graph.

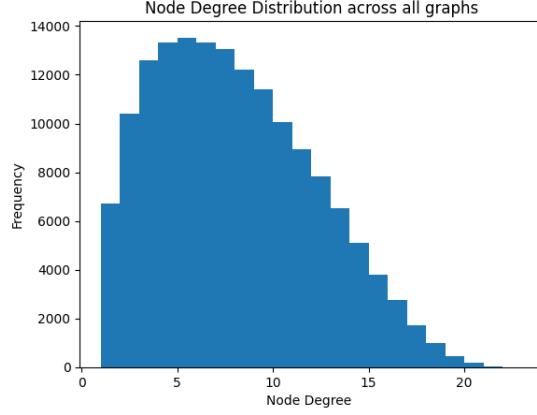


Figure 3: The histogram demonstrates the diversity of node degrees resulting from the predefined minimum and maximum fill rates and vertex counts.

Calculating the starting Gamma and Beta values usually require running the QAOA algorithm at depth one for a range of β and γ parameters. This process either requires a large scale simulation, which is very computationally expensive on classical hardware. This simulation scales linearly with the number of edges and nodes for a given graph. On the other hand, running on real quantum hardware is computationally very slow and expensive due to the repeated transpilation and long wait times for API calls, which are both required to search the β and γ parameter space. Both options are computationally expensive and slow when compared to a neural network, which should give results in a faster time.

4 Methodology

4.1 Data Creation

We created our own dataset containing a graph ID, adjacency list, and optimal γ and β values, which were used to train the graph convolutional network model. To generate this data, we defined four input parameters: minimum number of vertices, maximum number of vertices, minimum fill rate of a graph, and maximum fill rate of a graph. We define the minimum and maximum fill rates as the percentages of edges that are filled in when compared to a complete graph. A fill rate of 1 represents a complete graph. We selected the maximum number of nodes to be 24, as this is the limit of the Qiskit simulator, and the minimum number of nodes to be 2, as a graph requires at least two nodes to have a max cut greater than 0. To represent a broad spectrum of graphs, we set the minimum fill rate to 0.1 and the maximum fill rate to 0.9. This range of fill rates strikes a balance between sparsity and density in graph structures, enabling the generation of a diverse set of graphs with varying complexities. As shown in figure 3, the dataset generated includes graphs with varying densities and structures, as evidenced by the wide range of node degrees.

The number of vertices, defined as V , was randomly selected between the predefined minimum and maximum vertices. We then calculated the number of edges for V vertices. We set the minimum number of edges to be equal to the minimum fill rate times the number of edges in a complete graph. Similarly, we set the maximum number of edges to be equal to the maximum fill rate times the number of edges in a complete graph. We then chose a random number between the minimum and maximum number of edges, which is how the adjacency list was formed.

To calculate the optimal gamma and beta values, we ran the QAOA at depth one. We used COBYLA, a SciPy method for constrained optimization, to find the best gamma and beta values that provide a good expected value from the QAOA function. This computation was run on RPI's Aimos's DSC Cluster.

173 4.2 Evaluation Metrics

174 We introduce a custom loss function specifically designed for evaluating the γ_i and β_i prediction
175 values in our model. Traditional evaluation metrics such as mean absolute error, root mean squared
176 error, training loss, and testing loss are inadequate for our purposes because they do not account for
177 the requirements focusing on the expected value.

178 Our custom loss function addresses these requirements by focusing on the expected values produced
179 from a given γ and β value. Here, we define expected value as the average cut of a graph after
180 QAOA at depth 1 for the given γ and β value. This approach allows us to directly compare the
181 expected value calculated with the predicted γ and β values against the expected value calculated
182 from known good γ and β values from our created dataset. This will help ensure a more relevant and
183 precise assessment of prediction accuracy. Traditional metrics result in high error rates because they
184 compare our predicted values with dataset values we assume to be good, which may not reflect the
185 best possible γ and β values. We need a custom loss function because the model could predict a better
186 γ and β value, but because we provided the model with our known values, a normal loss function will
187 indicate bad model performance. This will cause a high error rate if the γ and β values do not match
188 our dataset. To encourage our model to outperform the expected γ and β values, we introduced an
189 offset of 10 to the total loss for each epoch. This offset allows the model to reward negative loss
190 values, rather than penalize them, thereby incentivizing the model to exceed expectations. By directly
191 comparing to the expected value, our custom loss function provides a more accurate and meaningful
192 evaluation of our model's performance.

```
193  
194 1 def custom_loss(actual_params, predicted_params, graphs, batch_size):  
195 2     if len(actual_params) < batch_size:  
196 3         print(f"Skipping batch as its size {len(actual_params)} is  
197         smaller than {batch_size}.")  
198 4         return torch.tensor(0.0, device='cuda', requires_grad=True)  
199 5  
200 6     randomIndices = random.sample(range(0, batch_size), 5)  
201 7     totalLoss = 0  
202 8     for i in randomIndices:  
203 9         gammaPredicted = predicted_params[0][i].item()  
204 10        betaPredicted = predicted_params[1][i].item()  
205 11        gammaKnown = actual_params[i][0].item()  
206 12        betaKnown = actual_params[i][1].item()  
207 13        graph = graphs[i].tolist()  
208 14        expected_cost = cost_function([betaKnown, gammaKnown], True,  
209        1, graph)  
210 15        predicted_cost = cost_function([betaPredicted,  
211        gammaPredicted], True, 1, graph)  
212 16        totalLoss += (predicted_cost - expected_cost)  
213 17    totalLoss += 10  
214 18    if totalLoss < 0:  
215 19        totalLoss = 0.0  
216 20    return torch.tensor(totalLoss, device='cuda', requires_grad=True)
```

218 4.3 Model Architecture

219 The model is a Graph Convolutional Network (GCN) implemented using PyTorch to predict optimal
220 γ and β values for graph structures. The architecture is designed to capture complex relationships
221 between nodes in a graph by leveraging multiple graph convolutional layers, each followed by batch
222 normalization and ReLU activation functions to stabilize training and improve generalization.

223 We begin by initializing every node in the graph to have one singular feature, as we were more
224 focused on how the structure of the graph rather than the individual node characteristics. The model
225 consists of four graph convolutional layers, which are used to learn node representations based on the
226 neighbors in the graph. The first convolutional layer looks at each node in the graph and all other
227 neighbors of that node. By combining all the features it is now learned about the neighbors of the
228 original node, the node then contains 128 features compared to the original one. By looking at the

previous layer, this process repeats ultimately for the next four convolutional layers, going from 128 features to 256 and finally outputting 512 features per node. By the time all the convolutional layers have been passed through, each node has gone from being represented as just a singular number to be represented by 512 feature vector.

The next part of the process is going through global pooling layer, which ultimately causes each nodes feature vector to be condensed into a single vector that summarizes the entire graph. As such, for each of the 512 features in the vector, the average is computed across all the nodes in the graph, ultimately creating a vector that contains the overall structure and feature distribution of the graph. More so, it reduces the dimensionality of the node features to a fixed size, regardless of the number of nodes in the graph. This layer is crucial for reducing the computational complexity. This layer will then output a singular 512 dimensional vector that represents the entire graph.

Lastly, the 512-dimensional vector passes through a series of fully connected layers, where it is first reduced to 256 dimensions. A ReLU activation is applied, allowing the model to capture complex patterns that might not have been recognized yet. This process repeats, with the vector being reduced to 128 dimensions, followed by another ReLU activation, and so forth until the vector is finally reduced to a size of two. These two dimensions correspond to the predicted γ and β values. The non-linear ReLU function is crucial for the model structure because most real-world relationships are non-linear. Without this, the model would only be able to capture linear relationships, which could lead to underfitting and limit its ability to predict complex relationships.

```

248
249 1 class GCNRegressor(nn.Module):
250 2     def __init__(self):
251 3         super(GCNRegressor, self).__init__()
252 4         self.conv1 = GCNConv(in_channels=1, out_channels=128)
253 5         self.conv2 = GCNConv(in_channels=128, out_channels=256)
254 6         self.conv3 = GCNConv(in_channels=256, out_channels=256)
255 7         self.conv4 = GCNConv(in_channels=256, out_channels=512)
256 8         self.bn1 = nn.BatchNorm1d(128)
257 9         self.bn2 = nn.BatchNorm1d(256)
258 10        self.bn3 = nn.BatchNorm1d(256)
259 11        self.bn4 = nn.BatchNorm1d(512)
260 12        self.fc1 = nn.Linear(512, 256)
261 13        self.fc2 = nn.Linear(256, 128)
262 14        self.fc3 = nn.Linear(128, 2) # Predicting gamma and beta
263 15
264 16    def forward(self, batch):
265 17        adj_matrices, params = batch
266 18        batch_size, num_nodes, _ = adj_matrices.size()
267 19
268 20        outputs = []
269 21        for i in range(batch_size):
270 22            adj_matrix = adj_matrices[i]
271 23            edge_index = adj_to_edge_index(adj_matrix)
272 24            node_features = torch.ones((num_nodes, 1)).to(device)
273 25
274 26            x = F.relu(self.bn1(self.conv1(node_features,
275            edge_index)))
276 27            x = F.relu(self.bn2(self.conv2(x, edge_index)))
277 28            x = F.relu(self.bn3(self.conv3(x, edge_index)))
278 29            x = F.relu(self.bn4(self.conv4(x, edge_index)))
279 30            x = F.dropout(x, p=0.5, training=self.training)
280 31            x = F.relu(self.fc1(x))
281 32            x = F.relu(self.fc2(x))
282 33            out = self.fc3(x.mean(dim=0))
283 34            outputs.append(out)
284 35        return torch.stack(outputs)
285

```

286 4.4 Implementation

287 The model training was implemented in PyTorch, focusing on ensuring compatibility with RPI's
288 AiMos NPL Cluster and the associated graphics processing unit (GPU). An instance of the GCN
289 model was instantiated and moved to the correct GPU device. Using a learning rate of 0.01 and the
290 Adam optimizer, we began to train our model. We had a total of 20 epochs and a batch size of 32.
291 For each batch, the model would make its predictions, and the loss was calculated using a custom
292 loss function specific to the batch. We trained the model using a single node on the RPI Center for
293 Computational Innovation's NPL Cluster. This node was equipped with two 20-core 2.5 GHz Intel
294 Xeon Gold 6248 processors, eight NVIDIA Tesla V100 GPUs (each with 32 GiB HBM), and 768
295 GiB of RAM. However, our PyTorch model only utilized one GPU for training.

296 4.4.1 QAOA Qiskit

297 Later in our code implementation, after finding optimal γ and β values, we generate a circuit that can
298 be run on the RPI-IBM Quantum System One. In this context, the GCN model predicts the γ and β
299 parameters, which are then used by the QAOA circuit function to construct, optimize, and run the
300 quantum circuit. Our circuit generation is defined as follows:

```
301 def QAOA_circuit(beta: float, gamma: float, depth: int,  
302 adjacencyList: list[list[int]]):  
303     num_qubits = len(adjacencyList)  
304     qc = QuantumCircuit(num_qubits, num_qubits)  
305     qc.h(range(num_qubits))  
306     qc.barrier()  
307     for _ in range(depth):  
308         for edgeIndex, neighbors in enumerate(adjacencyList):  
309             for edge in neighbors:  
310                 if edge > edgeIndex:  
311                     qc.cz(edgeIndex, edge)  
312                     qc.rz(2 * gamma, edgeIndex)  
313                     qc.cz(edge, edgeIndex)  
314             qc.barrier()  
315             for qubit in range(num_qubits):  
316                 qc.rx(2 * beta, qubit)  
317             qc.barrier()  
318     qc.measure(range(num_qubits), range(num_qubits))  
319     return qc
```

```
322 def QAOA(beta: float, gamma: float, fakeBackend: bool, depth: int,  
323 adjacencyList: list[list[int]],) -> dict:  
324     if fakeBackend:  
325         backend = GenericBackendV2(len(adjacencyList))  
326     else:  
327         backend = renslearBackend  
328     qc = QAOA_circuit(beta, gamma, depth, adjacencyList)  
329     job = backend.run(qc, shots=1024)  
330     result = job.result()  
331     return result.get_counts()
```

334 5 Results

335 Based off the total of ten graphs, which were all run on the RPI-IBM Quantum System One, the
336 classical expected value (EV) was 341.5713, and the GCN EV was 337.4512. The total time to find
337 the γ and β values using the classical approach was 854.4607 seconds, while the total time to find the
338 γ and β with the GCN was 0.029002 seconds.

Graph	Classical Beta	Classical Gamma	Classical EV	Classical Time in Seconds	GCN Beta	GCN Gamma	GCN EV	GCN Time in Seconds
[[1, 12, 13, 8, 6, 4, 3], ...]	0.6722	0.5703	28.42	19.45	-0.0655	0.1148	27.98	0.0038
[[12, 16, 19, 11, 2, 1], ...]	0.7546	0.8523	39.09	47.93	-0.0340	0.0758	38.14	0.0030
[[6, 7, 3, 12, 9, 10], ...]	2.2413	2.1869	11.87	17.36	-0.0320	0.1051	12.04	0.0029
[[11, 12, 6, 3], ...]	0.6112	0.6382	17.14	20.15	-0.0630	0.1202	17.20	0.0027
[[4, 8, 9, 1], ...]	3.9368	1.1151	8.86	8.82	-0.0235	0.1159	8.94	0.0027
[[13, 9, 7, 15, 6, 11, 2, 16, 1, 3, 4, 12], ...]	2.4383	1.7958	57.18	58.56	-0.1189	0.2139	56.29	0.0029
[[6, 1, 22], ...]	1.5488	0.5639	33.91	260.22	-0.0436	0.1134	33.60	0.0027
[[18, 3, 17, 14, 11, 15, 7, 5, 8, 4, 19], ...]	1.4853	1.2505	64.99	62.35	-0.1073	0.1549	64.77	0.0027
[[5, 2, 6, 1], ...]	0.4996	0.5010	15.05	11.77	-0.0376	0.1118	14.80	0.0027
[[17, 1, 7, 2, 12, 6, 4, 9, 18, 10, 5], ...]	0.7479	0.4673	65.07	347.86	-0.1058	0.1216	63.71	0.0028

Table 1: Comparison of Classical and GCN Methods Across Different Graphs

$$\text{Reduction} = \left(\frac{854.4607 - 0.029002}{854.4607} \right) \times 100 \approx 99.997\%$$

$$\text{Expected Value} = \left(\frac{341.5713 - 337.4512}{341.5713} \right) \times 100 \approx 1.2\%$$

Therefore, we achieved a 99.99% reduction time with a 1.2% expected value reduction. This suggests that the GCN method is not only computationally efficient but also maintains a high degree of accuracy, making it a viable alternative to classical approaches in practical scenarios. Although the GCN model demonstrates a significant reduction in prediction time, it is important to consider the training phase, which required approximately 521 minutes. This training time is a one-time cost, and once the model is trained, the benefits of rapid prediction times are realized across all future predictions. This trade-off between initial training time and subsequent prediction efficiency makes the GCN model a highly practical choice for large-scale applications.

6 Conclusion

In this research, we successfully leveraged a Graph Convolutional Network to predict optimal β and γ for the QAOA algorithm applied to the Ising Spin Model. Our approach demonstrated a significant reduction in computation time, achieving a 99.99% decrease compared to classical methods, with only a 1.2% reduction in expected value.

The solving of the Ising Spin System will map over to any other NP-complete problem. Ultimately, by improving the computation of this problem will help the scientific community by improving the solutions to problems like polymer folding, memory, and decision-making in social sciences and economics[6].

In conclusion, our work not only advances the field of quantum optimization but also holds promise for broader scientific applications. Our work with QAOA and GCNs aims to develop a model that not only can calculate β and γ values much faster than traditional methods, but also serves as a benchmark for future research in quantum optimization.

7 Future Work

7.1 Monte Carlo k-folding

In our current approach, the custom loss function operates by randomly selecting 5 points from the dataset to evaluate the model's predictions. While this method provides a quick and straightforward way to approximate the model's performance, it has limitations. Specifically, because only a small, random subset of points is considered in each iteration, there's a risk that the model might not fully capture the underlying patterns in the data, leading to less reliable results.

To address this limitation, we propose introducing a Monte Carlo k-folding technique in our future work. Unlike the current method, this technique systematically partitions the entire dataset into k subsets (or "folds"). In each iteration, one of these folds is used as the test set, while the remaining k-1 folds are used for training the model. This process is repeated k times, with each fold serving as the test set exactly once.

However, instead of just performing standard k-fold cross-validation, we will integrate Monte Carlo sampling into the process. This means that within each fold, we will randomly sample points multiple

times across different iterations, ensuring that over the course of training, every point in the dataset is eventually selected and tested multiple times, rather than the current implementation where it can not be ensured that every point is selected at least once. This repeated random sampling within the k-fold framework allows us to estimate the expected value of the loss more accurately for each point.

By employing Monte Carlo k-folding, we can better assess the model’s performance across the entire dataset, rather than relying on a small, potentially unrepresentative subset of points. This approach will lead to a more thorough evaluation, ensuring that our model is tested on a more diverse and representative sample of the data, ultimately resulting in a more accurate and generalized model. This refinement is expected to yield a more robust understanding of the model’s predictive capabilities and improve its reliability in practical applications.

7.2 Implementing Noise

We currently calculate our known β and γ values by simulating the QAOA quantum circuits on a Qiskit simulator without noise. While this simulation is efficient, it lacks accuracy in representing the noisy testing environment of the RPI-IBM Quantum System One.

To address this issue, we propose calculating the known β and γ by simulating the QAOA quantum circuits on a Qiskit simulator with noise. Although this approach does not perfectly replicate the noisy testing environment of the RPI-IBM Quantum System One, it would provide a more accurate and realistic simulation when classically calculating β and γ values.

7.3 Classical Optimization after using GCN

Currently, the β and γ values generated using the GCN underperform those generated using the classical approach by 1.2%. To address this underperformance, we propose classically optimizing the β and γ values produced by the GCN. This approach is less computationally expensive than classically generating β and γ values from scratch, as the GCN has already identified relatively good β and γ values that can be further refined through local classical optimization.

8 References

- [1] Hidary, Jack. 2019. Quantum Computing : An Applied Approach. Springer Nature.
- [2] Jain, Nishant, Brian Coyle, Elham Kashefi, and Niraj Kumar. 2022. “Graph Neural Network Initialisation of Quantum Approximate Optimisation.” Quantum 6 (November): 861. <https://doi.org/10.22331/q-2022-11-17-861>.
- [3] Jin, Allison, and Xiao-Yang Liu. 2023. “A Fast Machine Learning Algorithm for the MaxCut Problem.” 2023 IEEE151 MIT Undergraduate Research Technology Conference (URTC), October, 1–5. <https://doi.org/10.1109/urtc60662.2023.10534996>.
- [4] Liang, Zhiding, Gang Liu, Zheyuan Liu, Jinglei Cheng, Tianyi Hao, Kecheng Liu, Hang Ren, et al. n.d. “Graph Learning for Parameter Prediction of Quantum Approximate Optimization Algorithm.” Accessed August 18, 2024. <https://arxiv.org/pdf/2403.03310>.
- [5] Lu, Yicheng, and Xiao- Yang Liu. 2023. “Reinforcement Learning for Ising Model.” NeurIPS .
- [6] Lucas, Andrew. 2014. “Ising Formulations of Many NP Problems.” Frontiers in Physics 2. <https://doi.org/10.3389/fphy.2014.00005>.
- [7] Google Quantum AI, "Quantum Approximate Optimization Algorithm for the Ising Model," 2024. Available: https://quantumai.google/cirq/experiments/qaoa/qaoa_ising.
- [8] Uzair Aslam Bhatti, Hao Tang, Guilu Wu, Shah Marjan, and Aamir Hussain. 2023. “Deep Learning with Graph Convolutional Networks: An Overview and Latest Applications in Computational Intelligence” 2023 (February): 1–28. <https://doi.org/10.1155/2023/8342104>.
- [9] Zhang, Si, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. “Graph Convolutional Networks: A Comprehensive Review.” Computational Social Networks 6 (1). <https://doi.org/10.1186/s40649-019-0069-y>.