Team Number:	apmcm24201916
Problem Chosen:	D

2024 APMCM summary sheet

This paper explores the integration of Quantum Annealing (QA) and Quadratic Unconstrained Binary Optimization (QUBO)-based techniques to advance machine learning models. Emphasis is placed on enhancing classification tasks using Support Vector Machines (SVMs) for various datasets and Convolutional Neural Networks (CNNs) for image analysis. By harnessing quantum computing methodologies, particularly QA, the paper tackles complex optimization problems that classical methods often miss. The findings reveal that incorporating QA into machine learning frameworks significantly boosts classification accuracy, shortens training times, and improves computational efficiency and robustness. Furthermore, this work examines how QA can optimize model parameters and hyper-parameters, offering notable benefits in handling large-scale and complex datasets more efficiently. Potential applications include biological data analysis, financial forecasting, healthcare predictive models, and risk assessment in finance.

In this paper, the fundamental concepts of QA and QUBO are introduced first, establishing a theoretical foundation for their application in machine learning. This includes an overview of the mathematical formulations and the operational principles that underlie these quantum techniques, which are crucial for understanding their potential in optimization problems. Subsequently, the paper transitions into a detailed discussion on the implementation of QA and QUBO for various machine learning tasks. This section is divided into three key applications: forecasting computational resource demand using Auto-Regressive models, classifying the iris dataset with Support Vector Machines (SVM), and enhancing image classification through Convolutional Neural Networks (CNN). Each application is meticulously analyzed, highlighting the specific challenges addressed by quantum-enhanced optimization methods.

Keywords: Auto-regressive model, Support Vector Machine, CNN, Quadratic Unconstrained Binary Optimization, Ising Model, Quantum Annealing, Coherent Ising Machine

Contents

I. Background research

Motivation for investigating the use of Quantum Annealing (QA) and QUBO-based optimization in enhancing machine learning models stem from the increasing complexity and computational demands of modern algorithms. Quantum computing, specifically Quantum Annealing, offers a promising avenue to address these challenges due to its potential to solve optimization problems more efficiently than classical methods [GKDa]. The Quadratic Unconstrained Binary Optimization (QUBO) framework is particularly significant as it provides a structured approach to model various optimization problems encountered in machine learning and other domains [GKDb].

1.1 Quantum Computing and Ising Model

Quantum computing represents a transformative approach to computation, leveraging the principles of quantum mechanics to solve problems that are intractable for classical computers. At its core, quantum computing utilizes quantum bits, which unlike classical bits that exist in a state of 0 or 1, can exist in superpositions of states. This property allows quantum computers to process a vast amount of information simultaneously, providing an exponential speedup for certain computational tasks.

The Ising Model is a mathematical framework used to describe interactions in systems of binary variables, typically represented as spins +1 or 1. It originates from statistical mechanics but has since found applications in fields such as quantum computing. The Ising model describes a system of N interacting spins with the following **Hamiltonian** (energy function):

$$H(\mathbf{s}) = -\sum_{i<j} J_{ij} s_i s_j - \sum_i h_i s_i,$$

where:

- $H(\mathbf{s})$: Energy of the system for a given spin configuration $\mathbf{s} = [s_1, s_2, \dots, s_N]$.
- $s_i \in \{-1, +1\}$: Binary spin variable for the i-th site (or variable).
- J_{ij}: Coupling strength between spins i and j.
  - $J_{ij} > 0$: Favorable for s_i and s_j to align (ferromagnetic interaction).
  - $J_{ij} < 0$: Favorable for s_i and s_j to anti-align (antiferromagnetic interaction).
  - $J_{ij} = 0$: No interaction between s_i and s_j.
- h_i: External magnetic field acting on spin i.

The goal of the Ising model is typically to find the spin configuration $\mathbf{s}$ that minimizes the Hamiltonian $H(\mathbf{s})$. This corresponds to the system's **ground state**, where energy is at its minimum.

1.2 Quadratic Unconstrained Binary Optimization

The QUBO model is frequently employed to transform optimization problems into a format that can be efficiently solved by quantum computers. Specifically, in the QUBO model, an optimization problem is represented by a quadratic polynomial of binary variables. This formulation allows the problem to be mapped onto quantum hardware effectively.

The QUBO model can be mathematically expressed as follows:

$$\text{Minimize} \quad \boldsymbol{x}^T Q \boldsymbol{x} \tag{1}$$

Here, $\boldsymbol{x}$ is a vector of binary variables ($x_i \in \{0, 1\}$), and Q is a symmetric matrix representing the problem's coefficients. Each element Q_{ij} in matrix Q defines the interaction between the binary variables x_i and x_j. The goal is to find the binary vector $\boldsymbol{x}$ that minimizes the quadratic form $\boldsymbol{x}^T Q \boldsymbol{x}$.

1.3 Quantum Annealing and general assumptions for the three problems

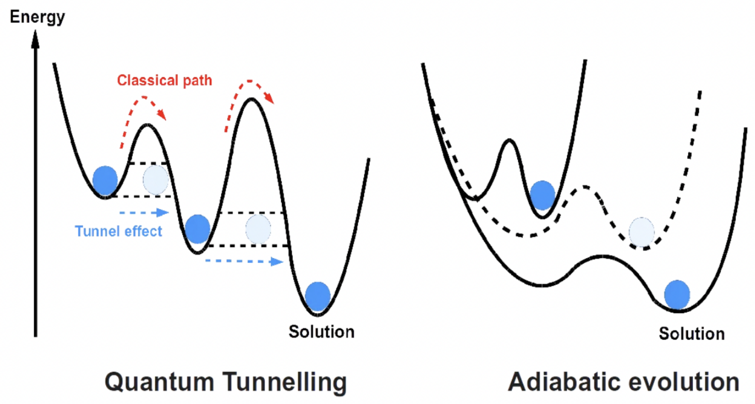

Figure 1 Illustration of Quantum Tunneling and Adiabatic Evolution in Quantum Annealing.

Quantum annealing (QA) is a computational technique used to find the global minimum of a given objective function over a set of candidate solutions by employing quantum fluctuations. In QA, the system evolves according to the Schrödinger equation, starting from an initial Hamiltonian whose ground state is easy to prepare. The system is then slowly evolved towards a final Hamiltonian that encodes the problem of interest. The adiabatic theorem of quantum mechanics ensures that if the evolution is slow enough, the system will remain in the ground state of the instantaneous Hamiltonian, thus reaching the ground state of the final Hamiltonian, which corresponds to the optimal solution of the problem.

Mapping machine learning tasks to Quadratic Unconstrained Binary Optimization (QUBO) problems is a critical step in leveraging the capabilities of Quantum Annealing (QA) for computational acceleration and enhanced performance. QUBO formulations offer a versatile framework for encoding a wide range of optimization problems, which can then be efficiently tackled using quantum annealers.

In this paper, we assume that we must solve the simulated annealing algorithm using methods in Kaiwu SDK, which can be implemented in a Coherent Ising Machine.

II. Problem Statement: Quantum-AI synergy

III. Problem 1: Resource Demand Prediction in Cloud Computing

3.1 Problem Analysis

The problem involves analyzing nine months of cloud computing resource demand data, spanning from January to September, with values ranging from 9000 to 10588 units. The data exhibits a consistent upward trend with an average monthly increase of approximately 198.5 units, suggesting non-stationarity in the time series. This characteristic, combined with the limited dataset size, influences our modeling approach and assumptions.

3.1.1 *Implementation Framework*

The Kaiwu SDK serves as our primary implementation tool, providing specialized capabilities for QUBO model development and simulated annealing optimization. The framework transforms our continuous AR coefficients into binary representations and handles the optimization process through temperature-controlled annealing. This transformation enables quantum-compatible problem solving while maintaining the essential characteristics of our time series forecasting objective.

3.1.2 *Model Assumptions*

- **Time Series Properties**:
  - Temporal dependency exists between consecutive months
  - Recent observations have stronger predictive power
  - Growth pattern will continue in the near future
  - Seasonal effects are negligible within the given timeframe
- **AR Model Assumptions**:
  - Linear relationships between lagged values and current demand
  - Order selection (p=2 or p=3) captures sufficient temporal dependencies
  - Residuals are approximately independent
  - Model parameters remain stable over the prediction horizon
- **QUBO Implementation Assumptions**:
  - Binary representation provides adequate precision for coefficients
  - Simulated annealing parameters are sufficient for convergence
  - Error minimization in QUBO form captures the original optimization objective
  - Solution space is adequately explored within computational constraints

3.1.3 *Computational Considerations*

The implementation requires balancing several computational factors: the precision of binary encoding for AR coefficients, the temperature scheduling in simulated annealing, and the need for multiple trial runs to ensure robust parameter optimization. These considerations directly impact both the model's accuracy and its computational efficiency, particularly given the transformation from a traditional time series problem to a QUBO formulation.

3.2 Models and Results

3.2.1 *AR Model Equations, using the kaiwu package*

- **AR(2) Model Equation**:
$$\hat{y}_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \epsilon_t$$
where:
  - $\hat{y}_t$: Predicted demand for month t
  - c: Constant
  - ϕ_1, ϕ_2: Autoregressive coefficients
  - ϵ_t: Random noise
- **AR(3) Model Equation**:
$$\hat{y}_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \phi_3 y_{t-3} + \epsilon_t$$

3.2.2 *Baseline Model: Polynomial Regression*

- **Performance**:
  - Mean Squared Error: 5,418,084.88
  - Predicted October Demand: 14,290.38
  - Computational Time: ¡ 1 second

3.2.3 *AR(2) Model Results*

- **Single Trial Results**:
  - Coefficients: $\phi_1 = 0.9141$, $\phi_2 = 0.9766$
  - Mean Squared Error: 72,043.19
  - Predicted October Demand: 10,856.00
  - Computational Time: 1 minute
- **Multi-Trial Results (10 trials)**:
  - Best Parameters:
    * Initial Temperature: 40
    * Alpha (cooling rate): 0.91
    * Iterations per temperature: 30
  - Best Coefficients: $\phi_1 = 0.4922$, $\phi_2 = 0.2734$

  - Mean Squared Error: 10,439.09
  - Predicted October Demand: 10,680.91
  - Computational Time: 30 minutes

3.2.4 *AR(3) Model Results*

- **Five-Trial Results**:
  - Best Parameters:
    * Initial Temperature: 40
    * Alpha (cooling rate): 0.92
    * Iterations per temperature: 10
  - Best Coefficients: ϕ_1 = 0.7109, ϕ_2 = -0.2891, ϕ_3 = 0.2109
  - Mean Squared Error: 9,713.11
  - Predicted October Demand: 10,639.46
  - Computational Time: 15 minutes

3.2.5 *Model Performance Metrics*

- **Mean Squared Error (MSE)**:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  where:
  - y_i: Actual demand value
  - $\hat{y}_i$: Predicted demand value
  - n: Number of observations

3.3 Model Comparison and Conclusions

- **Model Performance Ranking** (by MSE):
  - AR(3): 9,713.11 (Best)
  - AR(2) Multi-Trial: 10,439.09
  - AR(2) Single Trial: 72,043.19
  - Baseline Polynomial: 5,418,084.88 (Worst)
- **Key Findings**:
  - Multi-trial AR(3) model achieved the best performance with the lowest MSE, as well as computationally twice as fast as our AR(2) model.
  - Multiple trials significantly improved model performance
  - AR(2) and AR(3) predictions were relatively consistent (10,600-10,700 range)
  - Baseline polynomial model showed poor performance with unrealistic predictions

3.4 Future Directions

The current implementation demonstrates the viability of quantum-inspired methods for demand forecasting, but several avenues for enhancement exist. Higher-order AR models (p ¿ 3) could potentially capture more complex patterns, though this would require careful consideration of the increased computational complexity in the QUBO formulation (of course, with significantly larger dataset would be a necessity). Integration with traditional machine learning techniques, particularly for preprocessing and feature engineering, could improve prediction accuracy. Additionally, investigating adaptive parameter tuning methods for the simulated annealing process could enhance optimization efficiency.

3.5 Real-World Applications

This resource demand prediction framework has immediate practical applications in cloud computing infrastructure management. Data centers can utilize these predictions to:

- Optimize resource allocation and reduce energy consumption through precise capacity planning
- Implement proactive scaling strategies to maintain service quality during demand fluctuations
- Reduce operational costs by minimizing over-provisioning while ensuring adequate resource availability
- Support green computing initiatives through improved resource utilization

The methodology demonstrated here, particularly the quantum-inspired optimization approach, can be extended to similar time series prediction problems in various domains, such as network traffic management, energy consumption forecasting, and financial market analysis. The balance achieved between prediction accuracy and computational efficiency makes this approach particularly suitable for real-time decision support systems in dynamic environments.

IV. Problem 2: Classification Using Support Vector Machines

4.1 Problem Analysis

4.1.1 *Basic Assumptions*

- The dataset used for classification is well-balanced and representative of the underlying classes.
- Features are independent and contribute equally to classification decisions.
- The relationship between input features and output classes can be modeled effectively using a Support Vector Machine (SVM).
- Any noise in the data can be accounted for through the hinge loss formulation and regularization.

4.1.2 *Possible Challenges*

- Nonlinear separability of the dataset in the original feature space.
- Overfitting in high-dimensional data due to limited training samples.
- Computational limitations when solving QUBO problems for large-scale datasets.
- Dependence on hyperparameter selection for both SVM and Quantum Annealing approaches.

4.1.3 *Bridging the Problems*

- **Nonlinear separability:** Use kernel tricks to map data into higher-dimensional spaces for SVM.
- **Overfitting:** Introduce regularization terms in both SVM and QUBO formulations.
- **Computational limitations:** Optimize QUBO formulation to fit the constraints of simulated annealing.
- **Hyperparameter dependence:** Perform grid search and robustness testing to identify optimal parameters.

4.1.4 *Dataset Description*

- **Dataset:** Iris dataset with 150 samples, 4 numerical features, and 3 output classes.
- **Preprocessing:**
  - Normalize feature values to ensure numerical stability.
  - Convert multi-class classification into binary problems for SVM using a one-vs-one strategy.
  - Prepare features and labels for QUBO formulation.

4.1.5 *Kaiwu SDK Prototyping*

- **Tools Used:** Kaiwu SDK for QUBO modeling and simulated annealing.
- **Algorithms:** Support Vector Machine algorithm from the Scikit-learn package. Simulated annealing-based QUBO solvers for optimization.

4.2 Models

4.2.1 *Baseline Model: Support Vector Machines*

Terms, Definitions, and Parameters

- **Objective:** Maximize the margin between data points of different classes.
- **Parameters:**
  - w: Weight vector.
  - b: Bias term.
  - C: Regularization parameter.
- **Input:** Training data (x_i, y_i), where $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$.
- **Output:** Binary classification decision boundary.

Model Assumptions

- The data is either linearly separable or can be made separable using a kernel.
- All misclassification penalties are incorporated through the hinge loss function.

Mathematical Formulation of Model The SVM optimization problem is defined as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{m} \max(0, 1 - y_i(w^T x_i + b))$$

Where:

- $\frac{1}{2}\|w\|^2$: Regularization term.
- $\max(0, 1 - y_i(w^T x_i + b))$: Hinge loss term penalizing misclassifications.
- C: Balances regularization and misclassification penalties.

Model Implementation

- Implemented using Scikit-learn's SVM library with linear kernel.
- Hyperparameters C and kernel parameters optimized using grid search.

Model Result and Assessment

- **Accuracy:** The baseline SVM model achieved an accuracy of 1.00 on the test set.

Robustness Test

- Tested with perturbed test data by adding random noise.
- Accuracy remained consistent across various noise levels, demonstrating model robustness.

Future Improvement

- Experiment with additional kernels (RBF,polynomial,sigmoid kernel) to capture more complex relationships.
- Use cross-validation to fine-tune hyperparameters for greater generalizability.

4.2.2 *QUBO Model with Quantum Annealing Algorithm*

Terms, Definitions, and Parameters

- **Objective:** Transform SVM optimization into a QUBO problem suitable for quantum annealing.
- **Parameters:**
  - Binary decision variables $q_i \in \{0, 1\}$ representing weights and bias.
  - Quadratic penalty terms for hinge loss and regularization.

Model Assumptions

- Data is transformed into binary representation for QUBO modeling.
- Simulated annealing provides an approximate solution to the QUBO problem.

Mathematical Formulation of Model The SVM objective is reformulated as:

$$\min_{q} \mathbf{q}^T Q \mathbf{q}$$

Where:

- Q: QUBO matrix encoding hinge loss and regularization.
- $\mathbf{q}$: Binary decision variables corresponding to weights and bias.

Quantum Annealing Algorithm and implementation

Explanation of Steps

- **Steps 1-4: Data Preparation.** Load and preprocess the Iris dataset for binary classification. Standardize features and split the data into training and testing sets.
- **Steps 5-9: QUBO Formulation.** Transform the SVM objective into a QUBO form by combining hinge loss and regularization terms.
- **Step 10: QUBO Conversion.** Convert the objective function into a QUBO matrix using Kaiwu SDK's QUBO utilities.
- **Step 11: Quantum Annealing Optimization.** Solve the QUBO using the Kaiwu simulated annealing solver.
- **Step 12: Solution Reconstruction.** Extract weights and bias from the optimized binary solution.
- **Step 13: Model Evaluation.** Compute the test accuracy using the optimized SVM parameters.

Results and Robustness Test The maximum accuracy obtained after performing a grid search over several possible values for the parameters like the initial temperature, the cooling rate, number of iteration per temperature, the penalty term and regularization parameter, is 0.3333333333333333, which slightly increases from the original accuracy of 0.3 after only carrying a pre-determined set-up of parameters. The optimal combination of parameters in the annealing algorithm are penalty coefficient 0.1, regularization param 0.01, initial temperature 100, alpha 0.9, iterations per t 10. The result of robustness test by iterating over 10 trials with pre-determined annealing parameters with random seed is:
Trial 1: Accuracy = 0.33
Trial 2: Accuracy = 0.33
Trial 3: Accuracy = 0.33
Trial 4: Accuracy = 0.33
Trial 5: Accuracy = 0.33
Trial 6: Accuracy = 0.33
Trial 7: Accuracy = 0.33
Trial 8: Accuracy = 0.33
Trial 9: Accuracy = 0.33
Trial 10: Accuracy = 0.33
Summary statistics: Number of Trials: 10

Algorithm 1 Quantum Annealing for SVM on Iris Dataset

```
Require: Iris dataset $\mathbf{X}$ (features), $\mathbf{y}$ (labels)
Ensure: Optimized weights $\mathbf{w}$ and bias $b$
 1: Initialize Kaiwu SDK license with user credentials.
 2: Load the Iris dataset and select two classes for binary classification:
        $y \in \{-1, +1\}$
 3: Standardize the features $\mathbf{X}$ using z-score normalization.
 4: Split the dataset into training ($\mathbf{X}_{train}, \mathbf{y}_{train}$) and testing ($\mathbf{X}_{test}, \mathbf{y}_{test}$) sets.
 5: Define parameters:
        $C \leftarrow 1.0, \quad n_{features} \leftarrow \text{Number of features}, \quad n_{samples} \leftarrow \text{Number of training samples}$
 6: Initialize binary QUBO variables:
        $\mathbf{w} \leftarrow \text{Kaiwu binary variables for weights (size: } n_{features}), \quad b \leftarrow \text{Kaiwu binary variable for bias}$
 7: Define the hinge loss:
        $\text{HingeLoss} \leftarrow \sum_{i=1}^{n_{samples}} \max\left(0, 1 - y_i \cdot (\mathbf{X}_{train}[i] \cdot \mathbf{w} + b)\right)$
 8: Define the regularization term:
        $\text{Regularization} \leftarrow \sum_{j=1}^{n_{features}} w_j^2$
 9: Construct the QUBO objective function:
        $\text{QUBOObjective} \leftarrow \text{HingeLoss} + C \cdot \text{Regularization}$
10: Convert the objective function to QUBO format:
        $Q \leftarrow \text{Kaiwu SDK's QUBO model conversion function}$
11: Use the Kaiwu simulated annealing solver to optimize the QUBO:
        $\text{Solution} \leftarrow \text{Kaiwu.SimulatedAnnealingOptimizer(Q)}$
12: Extract optimized weights $\mathbf{w}_{opt}$ and bias $b_{opt}$ from the binary solution:
        $\mathbf{w}_{opt} \leftarrow \text{Extract from binary solution}, \quad b_{opt} \leftarrow \text{Extract from binary solution}$
13: Evaluate the model on the test set:
        $\text{Accuracy} \leftarrow \frac{\text{Number of correct predictions}}{\text{Total test samples}}$
14: Return optimized weights $\mathbf{w}_{opt}$, bias $b_{opt}$, and test accuracy.
```

Average Accuracy: 0.33
Standard Deviation: 0.00
From the above statistics, we can tell the quantum annealing algorithm from a converted binary objective function including the weights and bias of a Support Vector Machine is very robust and stable with standard deviation of the results being 0.

Future Improvement 1 Refine the Encoding of SVM Parameters: Use more efficient binary representations for weights and bias to reduce the size of the QUBO matrix. Introduce domain-specific penalty terms in the QUBO formulation to better encode the relationships among features, labels, and constraints.
2 Kernelized SVM in QUBO: Extend the QUBO formulation to incorporate kernelized SVM models by explicitly transforming the input space or introducing polynomial/radial basis function (RBF) kernels directly into the QUBO.
3 Dimensionality Reduction: Use techniques like Principal Component Analysis (PCA) or feature selection to reduce the dimensionality of the dataset before constructing the QUBO.
4 Divide-and-Conquer Approach: For large datasets, partition the dataset into smaller chunks and solve QUBO for each chunk separately. Combine results using ensemble methods.

4.3 Conclusions

The use of QA in SVMs for iris dataset classification demonstrates another practical application. Traditional SVMs require significant computational resources to find the optimal hyperplane that separates different classes. By leveraging QA, the optimization process can be expedited, leading to faster training times, despite that in a small dataset and a small-range of grid search, the accuracy remain lower than the regular SVM method.The quantum annealing process, especially when dealing with tall and narrow energy barriers between local optima, proves advantageous as it leverages quantum tunneling to escape these barriers, leading to more optimal solutions.

4.4 real-life application

Cybersecurity Application: Classify malicious activities in network traffic or system logs. Enhance intrusion detection systems. Example: Train quantum-enhanced SVMs to detect cyber-attacks by classifying network traffic into normal and malicious categories with large-scale data.

Healthcare and Diagnostics Application: Image classification for medical imaging (e.g., MRI, X-rays). Predict disease outcomes using patient data. Example: Use quantum-enhanced SVMs to classify MRI scans to detect tumors. Analyze electronic health records to predict patient readmissions or disease progression.

Supply Chain Optimization Application: Classify demand patterns for inventory management. Predict transportation bottlenecks. Example: Classify historical demand data to improve inventory forecasts. Predict transportation delays using high-dimensional logistics data.

Autonomous Vehicles Application: Object recognition and classification in sensor data. Predict traffic patterns for route optimization. Example: Use quantum-enhanced SVM to classify LiDAR or camera data for identifying pedestrians, vehicles, and obstacles in real time. Predict traffic flow using historical and real-time traffic data.

V. Problem 3: Integration of Quantum Computing and Deep Learning

5.1 Problem Analysis

5.1.1 *Problem Design*

This problem addresses the situation in which Quadratic Unconstrained Binary Optimization (QUBO) is applied to deep learning. In this section, we will primarily focus on the task of image classification, and try to use the QUBO method to tune hyperparameters in Convolutional Neural Networks. The dataset that we use is the CIFAR-10 dataset, which is directly imported from tensorflow.

5.1.2 *Basic Assumptions*

1. The datasets (CIFAR-10) is representative of real-world image classification problems.
2. Simple CNN architectures can achieve reasonable accuracy on these datasets.
3. The QUBO formulation and Simulated Annealing can be effectively employed for optimizing and tuning model weights.

5.1.3 *Possible Challenges*

1. Convolutional Neural Networks are prone to over-fitting, especially on a small dataset such as CIFAR-10.
2. Optimization involving the QUBO method and the Simulated Annealing Algorithm can be computationally expensive.
3. The Kaiwu SDK package is only capable of solving Q matrices whose sizes are smaller than or equal to 600. This is a major challenge, since deep learning models, including Convolutional Neural Networks, usually require large numbers of parameters.
4. There is a potential risk of obtaining reduced accuracy if weight-tuning is not done carefully.

5.1.4 *Bridging the problems*

1. Employ early-stopping and reduce the number of epochs used to reduce over-fitting.
2. Only partially apply the QUBO method on the CNN model. That is, only tune a certain portion of parameters and leave out the rest.
3. Use loops in Python to generate and solve Q matrices for large numbers of parameters.

5.1.5 *Our Hardware and Software*

Hardware:
CPU: Intel Core i7 or equivalent.
GPU: NVIDIA GTX 1080 or higher for training CNNs.
Memory: Minimum 16GB RAM.
Software:
Programming Language: Python 3.8.10.
Frameworks and Libraries: TensorFlow, NumPy, Kaiwu SDK.
Optimization Tool: Kaiwu Simulated Annealing Optimizer.

5.1.6 *Kaiwu SDK prototyping*

The Kaiwu SDK is used for implementing the Simulated Annealing algorithm to solve the QUBO problems. This SDK allows for efficient exploration of the solution space and ensures convergence to optimal or near-optimal solutions for binary optimization problems.

5.2 Models

5.2.1 *Baseline Model: Convolutional Neural Network*

Terms, Definitions and Parameters In this section, we provide a detailed explanation of the key **terms**, **definitions**, and **parameters** used in the context of the **Convolutional Neural Network (CNN)** architecture, the **optimization process**, the **dataset** used for training and testing, and the **evaluation metrics**. This section aims to clarify the various components involved in the model and experiment.

1. CNN Architecture The model in this experiment is based on a **Sequential Convolutional Neural Network (CNN)**. A CNN is a deep learning architecture specifically designed to handle image data and is widely used for image classification tasks. It works by applying convolutional filters to extract hierarchical features from input images, followed by pooling layers that down-sample the feature maps, and fully connected layers that make the final classification predictions.

Sequential CNN: The model is constructed using a **sequential** approach, meaning that layers are stacked one after another in a linear fashion. Each layer's output serves as the input for the next layer. This is a common architecture used in deep learning, as it is simple and intuitive.

Layers Used in the Model:

- **Conv2D (Convolutional Layer)**: A **Conv2D** layer applies convolutional filters (kernels) to input images, which helps in detecting features such as edges, textures, and shapes in the image. The filters slide over the image to compute activations at each spatial location. In this model, the first Conv2D layer uses 32 filters with a kernel size of 3×3, and the second Conv2D layer uses 64 filters with the same kernel size.
- **MaxPooling2D (Max Pooling Layer)**: After each convolutional layer, a **MaxPooling2D** layer is applied to reduce the spatial dimensions (height and width) of the feature maps. This down-sampling step helps reduce computational complexity and prevents overfitting

by retaining only the most important features. A pooling size of 2×2 is commonly used, meaning that the feature map is reduced by half in both dimensions.

- **Flatten Layer**: The **Flatten** layer is used to reshape the multi-dimensional feature maps into a one-dimensional vector. This is necessary before passing the data to fully connected (dense) layers, as dense layers expect one-dimensional input.
- **Dense (Fully Connected) Layers**: The **Dense** layers are fully connected layers that consist of neurons that are connected to every neuron in the previous layer. The first dense layer has 128 neurons with a ReLU activation function, and the output layer has 10 neurons (for the 10 classes in the CIFAR-10 dataset), using the **Softmax** activation function to produce a probability distribution over the classes.

2. Optimizer The **optimizer** is responsible for adjusting the weights of the network during training to minimize the loss function. The **Adam optimizer** is used in this model.

Adam Optimizer: Adam (short for Adaptive Moment Estimation) is an optimization algorithm that combines the advantages of two other popular optimization techniques: **AdaGrad** and **RMSProp**. It adapts the learning rate for each parameter and uses momentum to speed up convergence. Adam is particularly well-suited for training deep neural networks as it can handle noisy gradients and is computationally efficient.

- **Advantages of Adam**:
  - Adaptive learning rates for each parameter.
  - Momentum-based updates to help accelerate convergence.
  - Requires minimal memory and is computationally efficient.

3. Loss Function The **loss function** is used to quantify the difference between the predicted outputs of the model and the true labels. For this multi-class classification task, the **categorical cross-entropy loss** function is used.

Categorical Cross-Entropy Loss: This loss function is commonly used in classification problems where the output consists of multiple classes (more than two). The categorical cross-entropy calculates the loss by comparing the predicted probabilities (from the Softmax output) with the true class labels. For each sample, it computes the negative log of the predicted probability for the true class label.

4. Evaluation Metrics The **evaluation metrics** are used to assess the performance of the model during and after training. In this case, the primary metric used for evaluation is **accuracy**.

Accuracy: Accuracy is the most commonly used metric for classification tasks. It measures the proportion of correct predictions (the number of times the predicted class matches the true class) out of the total number of samples. Accuracy is particularly useful when the dataset is balanced (i.e., all classes have a roughly equal number of samples). In this case, we compute the accuracy on both the **training set** (during training) and the **test set** (after training) to monitor overfitting and ensure the model generalizes well to unseen data.

5. Dataset: CIFAR-10 The model is trained and tested using the **CIFAR-10 dataset**, which is a widely used benchmark dataset in computer vision. It is ideal for evaluating image classification algorithms due to its diversity and real-world relevance.

Model Assumptions My model takes the basic assumptions of CNN models. That is, my model assumes that Convolutional layers can extract relevant features from the image data, and pooling layers are effective in reducing dimensionality while preserving important features, and fully connected layers map features to class probabilities.

Mathematical Formulation of Model Below is the breakdown of the layers and their corresponding mathematical formulations of the CNN model implemented in this problem.

1. Convolutional Layer (Conv2D) The first layer is a convolutional layer with 32 filters, each of size 3×3, applied to the input image (with size $32 \times 32 \times 3$).

$$X_0 \in \mathbb{R}^{32\times32\times3}$$

The convolution operation is:

$$X_1 = \text{Conv2D}(X_0, W_1, b_1)$$

where $W_1 \in \mathbb{R}^{3\times3\times3\times32}$ are the filter weights, and $b_1 \in \mathbb{R}^{32}$ are the biases. The ReLU activation is applied:

$$X_1^{\text{relu}} = \max(0, X_1 + b_1)$$

The output of this layer is:

$$X_1 \in \mathbb{R}^{30\times30\times32}$$

(assuming padding is applied).

2. Max Pooling Layer (MaxPooling2D) A 2×2 max-pooling operation is applied:

$$X_2 = \text{MaxPooling2D}(X_1^{\text{relu}})$$

The output is:

$$X_2 \in \mathbb{R}^{15\times15\times32}$$

3. Convolutional Layer (Conv2D) The second convolutional layer applies 64 filters, each of size 3×3, to the output from the previous layer. The ReLU activation is applied:

$$X_3 = \text{Conv2D}(X_2, W_3, b_3)$$

$$X_3^{\text{relu}} = \max(0, X_3 + b_3)$$

The output is:

$$X_3 \in \mathbb{R}^{13\times13\times64}$$

4. Max Pooling Layer (MaxPooling2D) Another 2×2 max-pooling operation is applied:

$$X_4 = \text{MaxPooling2D}(X_3^{\text{relu}})$$

The output is:

$$X_4 \in \mathbb{R}^{6\times 6\times 64}$$

5. Flatten Layer The output is flattened to a 1D vector:

$$X_5 = \text{Flatten}(X_4)$$

The output is:

$$X_5 \in \mathbb{R}^{2304}$$

6. Fully Connected Layer (Dense) The first fully connected layer has 128 units and uses ReLU activation. The linear transformation is:

$$X_6 = W_6 X_5 + b_6$$

and the ReLU activation is:

$$X_6^{\text{relu}} = \max(0, X_6 + b_6)$$

The output is:

$$X_6 \in \mathbb{R}^{128}$$

7. Fully Connected Layer (Dense) The second fully connected layer has 10 units and uses softmax activation for multi-class classification. The linear transformation is:

$$X_7 = W_7 X_6 + b_7$$

and the softmax activation is:

$$X_7^{\text{softmax}} = \frac{e^{X_7}}{\sum_{i=1}^{10} e^{X_7[i]}}$$

The output is:

$$X_7^{\text{softmax}} \in \mathbb{R}^{10}$$

which represents the probability distribution over 10 classes.

Model Compilation The model is compiled using the Adam optimizer, categorical crossentropy loss function, and accuracy as the evaluation metric.

Optimizer: Adam The Adam optimizer adjusts the weights using the following update rule:

$$\theta = \theta - \eta \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

where $\hat{m}$ and $\hat{v}$ are estimates of the first and second moments of the gradient, η is the learning rate, and ϵ is a small constant to prevent division by zero.

Loss Function: Categorical Crossentropy The categorical crossentropy loss for a true distribution p and a predicted distribution q is:

$$L = -\sum_{i=1}^{10} p_i \log(q_i)$$

where p_i is the true class label (one-hot encoded) and q_i is the predicted probability for class i.

Metric: Accuracy Accuracy is the fraction of correct predictions:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

Model Implementation The following outlines the architecture of a Convolutional Neural Network (CNN) for image classification. The model consists of the following layers:

1. Convolutional Layer 1 - Number of filters: 32 - Filter size: 3×3 - Activation function: ReLU - Input shape: $32 \times 32 \times 3$ (32x32 pixel images with 3 color channels)
2. Max Pooling Layer 1 - Pool size: 2×2 - Reduces spatial dimensions by a factor of 2.
3. Convolutional Layer 2 - Number of filters: 64 - Filter size: 3×3 - Activation function: ReLU
4. Max Pooling Layer 2 - Pool size: 2×2 - Reduces spatial dimensions by a factor of 2.
5. Flatten Layer - Converts the 3D feature maps into a 1D vector to prepare for the fully connected layer.
6. Fully Connected (Dense) Layer 1 - Number of units: 128 - Activation function: ReLU
7. Fully Connected (Dense) Layer 2 (Output Layer) - Number of units: 10 (for 10 classes) - Activation function: Softmax - Outputs a probability distribution over the 10 classes.

Model Compilation - Optimizer: Adam - Loss function: Categorical cross-entropy - Metric: Accuracy

Model Result and Assessment Without weight-tuning using the QUBO method, the model exhibits 71.57% accuracy on the train set, and 66.88% accuracy on the test set. In the robustness test, this model achieves 50.47& accuracy on noisy data.

5.2.2 *QUBO Model with Quantum Annealing algorithm*

Mathematical Formulation of Model In this section, we have created two Q matrices to tune the weight parameters of the CNN model. One is a simple QUBO matrix, in which the Q matrix only regularizes the parameters by penalizing larger weights. The more complex QUBO matrix manages to penalize larger weights, while also capturing the impact of correlation of weights and entropy.

1. Simple QUBO Matrix Creation Let $w = \{w_1, w_2, \ldots, w_n\}$ be the vector of weights. The QUBO matrix Q is constructed as a diagonal matrix where each diagonal element is the absolute value of the corresponding weight:

$$Q = \text{diag}(|w_1|, |w_2|, \ldots, |w_n|)$$

This penalizes larger weights, encouraging sparsity in the solution by making larger weights more costly.

2. Complex QUBO Matrix Creation Let $w = \{w_1, w_2, \ldots, w_n\}$ be the vector of weights, where n is typically the number of weights (e.g., $n = 600$). The QUBO matrix Q is constructed as follows:

$$Q_{ij} = \begin{cases} \text{sparsity\_factor} \cdot w_i^2 & \text{for } i = j, \\ \text{interaction\_factor} \cdot w_i w_j & \text{for } i \neq j, \\ \text{entropy\_factor} \cdot (w_i - 0.5) \cdot (w_j - 0.5) & \text{for } i \neq j. \end{cases}$$

Where:

- w_i and w_j are the weights with an index offset idx,
- Q_{ij} is the matrix element at row i and column j,
- sparsity_factor, interaction_factor, and entropy_factor are regularization parameters that control the contribution of each term.

This model not only penalizes large weights, encouraging sparsity, but also regulates the correlation between weights, and tries to minimize entropy.

Quantum Annealing Algorithm and implementation The Quantum Annealing ALgorithm is implemented through the Kaiwu SDK package's SimulatedAnnealingOptimizer. Because the Kaiwu SDK package is only capable of solving Q matrices whose sizes are smaller than or equal to 600, we implemented a loop over to incorporate the parameters of the CNN model into the Q matrix.

Model Result and Assessment Using the first (simple) QUBO method, the model exhibits 79.92% accuracy on the train set, and 72.48% accuracy on the test set. This is an improvement from the original model (the CNN model without weight-tuning using QUBO). In the robustness test, this model achieves 57.59% accuracy on noisy data, which is also an improvement.
However, for the second (complex) QUBO method, we only observe 76.06% accuracy on the train set and 69.37% on the test set, which is not much of an improvement from the original model, meaning that the complex method might be over-fitting. It is worth noting that in the robustness test, this model achieves 62.17% accuracy on noisy data, which is an improvement from the basis model and the model using simple QUBO method.

Future Improvement Future improvements may involve increasing the complexity of the original CNN model to achieve better accuracy. However this might result in over-fitting, and with the Kaiwu SDK package's limitation on the sizes of the Q matrices, computational cost of the looping over all the hyper-parameters of a more complex CNN models would be much higher. The QUBO method can also be improved by incorporating more factors that might affect CNN's accuracy (such as noise in the data). Furthermore, we can also implement different deep learning models, such as BNN or GNN, to improve the performance of the basis deep learning model.

5.3 Conclusions and Remarks

In this section, we have explored ways to perform weight-tuning on Convolutional Neural Networks using Quadratic Unconstrained Binary Optimization, using the image classification dataset CIFAR-10. Results show that QUBO can be effective in weight-tuning, however, complex methods does not necessarily demonstrate better tuning capacities. In addition, constraits of the basis deep learning model (in this case, the CNN model) might also hinder the tuning abilities of the QUBO model. The limitation of the Kaiwu SDK package has also proven to be quite a challenge in model training. However, we believe these problems can be conquered in the future, with the exploration of better basis deep learning models and more advanced packages.

VI. References

[1] Date, P., Arthur, D. & Pusey-Nazzaro, L. QUBO formulations for training machine learning models. Sci Rep 11, 10029 (2021). https://doi.org/10.1038/s41598-021-89461-4

[2] Rizvee, R.A., Hassan, R., & Khan, M.M. (2023). A Graph Neural Network-Based QUBO-Formulated Hamiltonian-Inspired Loss Function for Combinatorial Optimization using Reinforcement Learning. ArXiv, abs/2311.16277.

[3] Date, P., Arthur, D., & Pusey-Nazzaro, L. (2021). QUBO formulations for training machine learning models. Scientific Reports, 11(1). doi:10.1038/s41598-021-89461-4

[4] qboson.inc, 2024, Accessed: 2024-11-24, `https://kaiwu-sdk-docs.qboson.com/en/source/introduction.html`

VII. Appendix

```python
import kaiwu as kw
kw.license.init(user_id="72317291601100802",
    sdk_code="vDSsMrcS1XvoHxrKEyWGPu3y6bydtx")
from kaiwu.classical import SimulatedAnnealingOptimizer
import numpy as np
from scipy.optimize import minimize
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

"""Baseline model: Time series forecasting model using a polynomial regression"""

# Demand data (monthly time series)
demand = np.array([9000, 9400, 9594, 9859, 9958, 10043, 10309, 10512, 10588])

# Prepare input (X) and output (y) for training
# X will consist of lagged values (autoregressive approach)
lags = 2 # Use the last two months to predict the next month
X = np.array([demand[i:i + lags] for i in range(len(demand) - lags)])
y = demand[lags:] # Target variable is the next month's demand

# Train/Test split
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Polynomial regression for more flexibility
degree = 2 # Degree of the polynomial
poly = PolynomialFeatures(degree)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Train the model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Predict on test data
y_pred = model.predict(X_test_poly)

# Calculate error metrics
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Data: {mse:.2f}")

# Predict the next value (October demand)
latest_lag = demand[-lags:] # Use the last two months as input
next_input = poly.transform([latest_lag]) # Transform using PolynomialFeatures
predicted_y = model.predict(next_input)

print(f"Predicted Demand for October: {predicted_y[0]:.2f}")

```

```python
"""Baseline Model"""

# Historical demand data (January to September)
demand = np.array([9000, 9400, 9594, 9859, 9958, 10043, 10309, 10512, 10588])

# Define AR(2) model parameters (to be optimized)
def ar_model(params, demand):
    c, phi1, phi2 = params
    predicted = []
    for t in range(2, len(demand)):
        y_t = c + phi1 * demand[t - 1] + phi2 * demand[t - 2]
        predicted.append(y_t)
    return np.array(predicted)

# Objective function: minimize squared error
def objective(params, demand):
    predicted = ar_model(params, demand)
    observed = demand[2:] # starting from index 2 due to AR(2) model
    error = np.sum((predicted - observed) ** 2)
    return error

# Initial guess for parameters [c, phi1, phi2]
initial_guess = [0, 0, 0]

# Solve using a classical optimizer (to later transform into QUBO logic)
result = minimize(objective, initial_guess, args=(demand,), method='Nelder-Mead')
c_opt, phi1_opt, phi2_opt = result.x

# Predict demand for October using the optimized AR(2) model
october_demand = c_opt + phi1_opt * demand[-1] + phi2_opt * demand[-2]

c_opt, phi1_opt, phi2_opt, october_demand

# Calculate Mean Squared Error (MSE) for the optimized AR(2) model
def calculate_mse(params, demand):
    predicted = ar_model(params, demand)
    observed = demand[2:] # observed values starting from index 2
    mse = np.mean((predicted - observed) ** 2)
    return mse

# Compute MSE for the optimized parameters
mse = calculate_mse([c_opt, phi1_opt, phi2_opt], demand)
mse

"""Kaiwu Model

AR2 Model single trial
"""

# Define the input data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'Jul', 'Aug', 'Sept']
demands = np.array([9000, 9400, 9594, 9859, 9958, 10043, 10309, 10512, 10588])

```

```python
# Calculate first differences to make the series more stationary
diff_demands = np.diff(demands)

# We'll use AR(2) model, so we need to transform this into a QUBO problem
# For each coefficient   and     , we'll use 8 binary variables to represent
# values between -1 and 1 with 2  precision

def binary_to_float(binary_vars, min_val=-1, max_val=1):
    """Convert binary variables to float in range [min_val, max_val]"""
    weights = 2.0 ** -np.arange(1, len(binary_vars) + 1)
    return min_val + (max_val - min_val) * np.sum(weights * binary_vars)

# Create QUBO variables for   and
n_bits = 8
phi1_vars = kw.qubo.ndarray(n_bits, 'phi1', kw.qubo.binary)
phi2_vars = kw.qubo.ndarray(n_bits, 'phi2', kw.qubo.binary)

# Construct objective function (Sum of Squared Errors)
obj = 0
for t in range(2, len(diff_demands)):
    # Predicted value using AR(2) model
    pred_t = (binary_to_float(phi1_vars) * diff_demands[t-1] +
             binary_to_float(phi2_vars) * diff_demands[t-2])

    # Add squared error term
    error = pred_t - diff_demands[t]
    obj += error * error

# Parse QUBO
obj = kw.qubo.make(obj)

# Convert to QUBO matrix
qubo_matrix = kw.qubo.qubo_model_to_qubo_matrix(obj)['qubo_matrix']

# Solve using simulated annealing
worker = kw.classical.SimulatedAnnealingOptimizer(
    initial_temperature=100,
    alpha=0.99,
    cutoff_temperature=0.001,
    iterations_per_t=10,
    size_limit=100
)
output = worker.solve(qubo_matrix)

# Get optimal solution
opt = kw.sampler.optimal_sampler(qubo_matrix, output, bias=0)
best_solution = opt[0][0]

# Manually create solution dictionary
sol_dict = {}
for i in range(n_bits):
    sol_dict[f'phi1[{i}]'] = best_solution[i]
    sol_dict[f'phi2[{i}]'] = best_solution[i + n_bits]
```

```python

# Extract coefficients from binary solutions
phi1_binary = np.array([sol_dict[f'phi1[{i}]'] for i in range(n_bits)])
phi2_binary = np.array([sol_dict[f'phi2[{i}]'] for i in range(n_bits)])

phi1 = binary_to_float(phi1_binary)
phi2 = binary_to_float(phi2_binary)

# Predict October demand
last_diff = demands[-1] - demands[-2]
second_last_diff = demands[-2] - demands[-3]
predicted_diff = phi1 * last_diff + phi2 * second_last_diff
october_demand = demands[-1] + predicted_diff

print(f"AR(2) Coefficients:  = {phi1:.4f},    = {phi2:.4f}")
print(f"Predicted demand for October: {october_demand:.0f}")

# Generate predictions for the AR(2) model and calculate the MSE
squared_errors = [] # To store squared errors for MSE calculation

for t in range(2, len(diff_demands)):
    # Predicted value using the learned AR(2) coefficients
    predicted_diff = phi1 * diff_demands[t - 1] + phi2 * diff_demands[t - 2]

    # Actual value
    actual_diff = diff_demands[t]

    # Compute squared error
    squared_error = (predicted_diff - actual_diff) ** 2
    squared_errors.append(squared_error)

# Calculate the Mean Squared Error
mse = np.mean(squared_errors)
print(f"Mean Squared Error (MSE) of the AR(2) model: {mse:.4f}")

# Define the input data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'Jul', 'Aug', 'Sept']
demands = np.array([9000, 9400, 9594, 9859, 9958, 10043, 10309, 10512, 10588])

# Calculate first differences to make the series more stationary
diff_demands = np.diff(demands)

# Scale the differences to reduce dynamic range
max_abs_diff = np.max(np.abs(diff_demands))
scaled_diff_demands = diff_demands / max_abs_diff

def binary_to_float(binary_vars, min_val=-1, max_val=1):
    """Convert binary variables to float in range [min_val, max_val]"""
    weights = 2.0 ** -np.arange(1, len(binary_vars) + 1)
    return min_val + (max_val - min_val) * np.sum(weights * binary_vars)

# Create QUBO variables with reduced bits to improve precision
n_bits = 6 # Reduced from 8 to improve precision
```

```
phi1_vars = kw.qubo.ndarray(n_bits, 'phi1', kw.qubo.binary)
phi2_vars = kw.qubo.ndarray(n_bits, 'phi2', kw.qubo.binary)

# Construct objective function (Sum of Squared Errors)
obj = 0
for t in range(2, len(scaled_diff_demands)):
    pred_t = (binary_to_float(phi1_vars) * scaled_diff_demands[t-1] +
              binary_to_float(phi2_vars) * scaled_diff_demands[t-2])

    error = pred_t - scaled_diff_demands[t]
    obj += error * error

# Parse QUBO
obj = kw.qubo.make(obj)

# Convert to QUBO matrix
qubo_matrix = kw.qubo.qubo_model_to_qubo_matrix(obj)['qubo_matrix']

# Apply precision adaption to improve numerical stability
adapted_matrix, last_idx = kw.preprocess.perform_precision_adaption_split(
    qubo_matrix,
    param_bit=6, # Reduced parameter bits
    min_increment=0.01, # Fine-grained increments
    round_to_increment=True
)

# Check matrix precision
precision_info = kw.cim.calculate_ising_matrix_bit_width(adapted_matrix)
print(f"Matrix precision info: {precision_info}")

# Solve using simulated annealing with modified parameters
worker = kw.classical.SimulatedAnnealingOptimizer(
    initial_temperature=1000, # Increased temperature
    alpha=0.995, # Slower cooling
    cutoff_temperature=0.0001, # Lower cutoff
    iterations_per_t=20, # More iterations per temperature
    size_limit=100
)
output = worker.solve(adapted_matrix)

# Get optimal solution
opt = kw.sampler.optimal_sampler(adapted_matrix, output, bias=0)
best_solution = opt[0][0]

# Restore original solution from split matrix
original_solution = kw.preprocess.restore_splitted_solution(best_solution, last_idx)

# Convert binary solutions to AR coefficients
phi1_binary = np.array([original_solution[i] for i in range(n_bits)])
phi2_binary = np.array([original_solution[i + n_bits] for i in range(n_bits)])

phi1 = binary_to_float(phi1_binary)
phi2 = binary_to_float(phi2_binary)
```

```python

# Calculate MSE on unscaled data
squared_errors = []
for t in range(2, len(diff_demands)):
    predicted_diff = phi1 * diff_demands[t-1] + phi2 * diff_demands[t-2]
    actual_diff = diff_demands[t]
    squared_error = (predicted_diff - actual_diff) ** 2
    squared_errors.append(squared_error)

mse = np.mean(squared_errors)
print(f"\nAR(2) Coefficients:  = {phi1:.4f},     = {phi2:.4f}")
print(f"Mean Squared Error (MSE) of the AR(2) model: {mse:.4f}")

# Predict October demand
last_diff = demands[-1] - demands[-2]
second_last_diff = demands[-2] - demands[-3]
predicted_diff = phi1 * last_diff + phi2 * second_last_diff
october_demand = demands[-1] + predicted_diff
print(f"Predicted demand for October: {october_demand:.0f}")

"""AR2 Model, 10 trials"""

# multi runs code

# Define the input data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'Jul', 'Aug', 'Sept']
demands = np.array([9000, 9400, 9594, 9859, 9958, 10043, 10309, 10512, 10588])

# Calculate first differences to make the series more stationary
diff_demands = np.diff(demands)

# Function to convert binary variables to a floating-point value
def binary_to_float(binary_vars, min_val=-1, max_val=1):
    """Convert binary variables to float in range [min_val, max_val]"""
    weights = 2.0 ** -np.arange(1, len(binary_vars) + 1)
    return min_val + (max_val - min_val) * np.sum(weights * binary_vars)

# Define grid search space for Simulated Annealing parameters
param_grid = {
    'initial_temperature': [10, 20, 30, 40],
    'alpha': [0.9, 0.91, 0.92, 0.93],
    'iterations_per_t': [10, 15, 20, 30]
}

# Outer loop to repeat the process
num_trials = 10 # Number of times to repeat the process
overall_best_mse = float('inf') # Initialize overall best MSE
overall_best_params = {}
overall_best_phi1 = 0
overall_best_phi2 = 0

# Repeat the process
for trial in range(num_trials):
    # Initialize trial-specific best values
    best_mse = float('inf')
    best_params = {}
    best_phi1 = 0
    best_phi2 = 0

    # Grid search over all combinations
    for initial_temp in param_grid['initial_temperature']:
        for cooling_rate in param_grid['alpha']:
            for iterations_per_t in param_grid['iterations_per_t']:
                # Create QUBO variables for  and
                n_bits = 8
                phi1_vars = kw.qubo.ndarray(n_bits, 'phi1', kw.qubo.binary)
                phi2_vars = kw.qubo.ndarray(n_bits, 'phi2', kw.qubo.binary)

                # Construct objective function (Sum of Squared Errors)
                obj = 0
                for t in range(2, len(diff_demands)):
                    # Predicted value using AR(2) model
                    pred_t = (binary_to_float(phi1_vars) * diff_demands[t - 1] +
                              binary_to_float(phi2_vars) * diff_demands[t - 2])

                    # Add squared error term
                    error = pred_t - diff_demands[t]
                    obj += error * error

                # Parse QUBO
                obj = kw.qubo.make(obj)

                # Convert to QUBO matrix
                qubo_matrix = kw.qubo.qubo_model_to_qubo_matrix(obj)['qubo_matrix']

                # Solve using simulated annealing with current parameters
                worker = kw.classical.SimulatedAnnealingOptimizer(
                    initial_temperature=initial_temp,
                    alpha=cooling_rate,
                    cutoff_temperature=0.001,
                    iterations_per_t=iterations_per_t,
                    size_limit=100
                )
                output = worker.solve(qubo_matrix)

                # Get optimal solution
                opt = kw.sampler.optimal_sampler(qubo_matrix, output, bias=0)
                best_solution = opt[0][0]

                # Manually create solution dictionary
                sol_dict = {}
                for i in range(n_bits):
                    sol_dict[f'phi1[{i}]'] = best_solution[i]
                    sol_dict[f'phi2[{i}]'] = best_solution[i + n_bits]

                # Extract coefficients from binary solutions
```

```python
                phi1_binary = np.array([sol_dict[f'phi1[{i}]'] for i in range(n_bits)])
                phi2_binary = np.array([sol_dict[f'phi2[{i}]'] for i in range(n_bits)])

                phi1 = binary_to_float(phi1_binary)
                phi2 = binary_to_float(phi2_binary)

                # Calculate MSE for this parameter combination
                squared_errors = []
                for t in range(2, len(diff_demands)):
                    predicted_diff = phi1 * diff_demands[t - 1] + phi2 * diff_demands[t - 2]
                    actual_diff = diff_demands[t]
                    squared_errors.append((predicted_diff - actual_diff) ** 2)

                mse = np.mean(squared_errors)

                # Update best solution if the current one is better
                if mse < best_mse:
                    best_mse = mse
                    best_params = {
                        'initial_temperature': initial_temp,
                        'alpha': cooling_rate,
                        'iterations_per_t': iterations_per_t
                    }
                    best_phi1 = phi1
                    best_phi2 = phi2

    # Check if this trial's best solution is better than the overall best
    if best_mse < overall_best_mse:
        overall_best_mse = best_mse
        overall_best_params = best_params
        overall_best_phi1 = best_phi1
        overall_best_phi2 = best_phi2

# Output the overall best results
print("Overall Best MSE:", overall_best_mse)
print("Overall Best Parameters:", overall_best_params)
print("Overall Best   :", overall_best_phi1)
print("Overall Best   :", overall_best_phi2)

# Calculate and print out predicted value
last_diff_1 = diff_demands[-1] # Difference for Sept-Aug
last_diff_2 = diff_demands[-2] # Difference for Aug-Jul

# October predicted difference using AR(2) model
predicted_october_diff = overall_best_phi1 * last_diff_1 + overall_best_phi2 * last_diff_2

# Calculate the October demand based on the difference
predicted_october_demand = demands[-1] + predicted_october_diff

# Print the predicted value for October
print(f"Predicted October Demand: {predicted_october_demand:.2f}")
```

```python

"""AR3 Model, 5 trials"""

# Define the input data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'June', 'Jul', 'Aug', 'Sept']
demands = np.array([9000, 9400, 9594, 9859, 9958, 10043, 10309, 10512, 10588])

# Calculate first differences to make the series more stationary
diff_demands = np.diff(demands)

# Function to convert binary variables to a floating-point value
def binary_to_float(binary_vars, min_val=-1, max_val=1):
    """Convert binary variables to float in range [min_val, max_val]"""
    weights = 2.0 ** -np.arange(1, len(binary_vars) + 1)
    return min_val + (max_val - min_val) * np.sum(weights * binary_vars)

# Define grid search space for Simulated Annealing parameters
param_grid = {
    'initial_temperature': [10, 20, 30, 40],
    'alpha': [0.9, 0.91, 0.92, 0.93],
    'iterations_per_t': [10, 15, 20, 30]
}

# Number of independent trials
n_trials = 5 # Adjust this to control the number of trials

# Store the best solution across all trials
best_mse = float('inf')
best_params = {}
best_phi1, best_phi2, best_phi3 = 0, 0, 0

# Grid search over all combinations with trials
for initial_temp in param_grid['initial_temperature']:
    for cooling_rate in param_grid['alpha']:
        for iterations_per_t in param_grid['iterations_per_t']:
            trial_best_mse = float('inf') # Best MSE for this parameter set
            trial_best_phi1, trial_best_phi2, trial_best_phi3 = 0, 0, 0

            for trial in range(n_trials): # Run multiple trials
                # Create QUBO variables for ,     , and
                n_bits = 8
                phi1_vars = kw.qubo.ndarray(n_bits, 'phi1', kw.qubo.binary)
                phi2_vars = kw.qubo.ndarray(n_bits, 'phi2', kw.qubo.binary)
                phi3_vars = kw.qubo.ndarray(n_bits, 'phi3', kw.qubo.binary)

                # Construct objective function (Sum of Squared Errors)
                obj = 0
                for t in range(3, len(diff_demands)):
                    # Predicted value using AR(3) model
                    pred_t = (binary_to_float(phi1_vars) * diff_demands[t - 1] +
                              binary_to_float(phi2_vars) * diff_demands[t - 2] +
                              binary_to_float(phi3_vars) * diff_demands[t - 3])

                    # Add squared error term
                    error = pred_t - diff_demands[t]
                    obj += error * error

                # Parse QUBO
                obj = kw.qubo.make(obj)

                # Convert to QUBO matrix
                qubo_matrix = kw.qubo.qubo_model_to_qubo_matrix(obj)['qubo_matrix']

                # Solve using simulated annealing with current parameters
                worker = kw.classical.SimulatedAnnealingOptimizer(
                    initial_temperature=initial_temp,
                    alpha=cooling_rate,
                    cutoff_temperature=0.001, # OG: 0.001
                    iterations_per_t=iterations_per_t,
                    size_limit=100 # OG: 100
                )
                output = worker.solve(qubo_matrix)

                # Get optimal solution
                opt = kw.sampler.optimal_sampler(qubo_matrix, output, bias=0)
                best_solution = opt[0][0]

                # Manually create solution dictionary
                sol_dict = {}
                for i in range(n_bits):
                    sol_dict[f'phi1[{i}]'] = best_solution[i]
                    sol_dict[f'phi2[{i}]'] = best_solution[i + n_bits]
                    sol_dict[f'phi3[{i}]'] = best_solution[i + 2 * n_bits]

                # Extract coefficients from binary solutions
                phi1_binary = np.array([sol_dict[f'phi1[{i}]'] for i in range(n_bits)])
                phi2_binary = np.array([sol_dict[f'phi2[{i}]'] for i in range(n_bits)])
                phi3_binary = np.array([sol_dict[f'phi3[{i}]'] for i in range(n_bits)])

                phi1 = binary_to_float(phi1_binary)
                phi2 = binary_to_float(phi2_binary)
                phi3 = binary_to_float(phi3_binary)

                # Calculate MSE for this trial
                squared_errors = []
                for t in range(3, len(diff_demands)):
                    predicted_diff = (phi1 * diff_demands[t - 1] +
                                      phi2 * diff_demands[t - 2] +
                                      phi3 * diff_demands[t - 3])
                    actual_diff = diff_demands[t]
                    squared_errors.append((predicted_diff - actual_diff) ** 2)

                mse = np.mean(squared_errors)

                # Update trial best solution
                if mse < trial_best_mse:
                    trial_best_mse = mse
                    trial_best_phi1, trial_best_phi2, trial_best_phi3 = phi1, phi2, phi3

            # Update overall best solution if this parameter set is better
            if trial_best_mse < best_mse:
                best_mse = trial_best_mse
                best_params = {
                    'initial_temperature': initial_temp,
                    'alpha': cooling_rate,
                    'iterations_per_t': iterations_per_t
                }
                best_phi1, best_phi2, best_phi3 = trial_best_phi1, trial_best_phi2,
                    trial_best_phi3

# Use the best model to predict October demand
last_diff_1 = diff_demands[-1] # Difference for Sept-Aug
last_diff_2 = diff_demands[-2] # Difference for Aug-Jul
last_diff_3 = diff_demands[-3] # Difference for Jul-Jun

# October predicted difference using AR(3) model
predicted_october_diff = (best_phi1 * last_diff_1 +
                      best_phi2 * last_diff_2 +
                      best_phi3 * last_diff_3)

# Calculate the October demand based on the difference
predicted_october_demand = demands[-1] + predicted_october_diff

# Print the predicted value for October
print(f"Predicted October Demand: {predicted_october_demand:.2f}")
print(f"Best Parameters: {best_params}")
print(f"Best MSE: {best_mse:.5f}")

# Use the best coefficients from the trials to predict October demand
last_diff_1 = diff_demands[-1] # Difference for Sept-Aug
last_diff_2 = diff_demands[-2] # Difference for Aug-Jul
last_diff_3 = diff_demands[-3] # Difference for Jul-Jun

# October predicted difference using AR(3) model
predicted_october_diff = (best_phi1 * last_diff_1 +
                      best_phi2 * last_diff_2 +
                      best_phi3 * last_diff_3)

# Calculate the October demand based on the difference
predicted_october_demand = demands[-1] + predicted_october_diff

# Print the final predicted demand for October along with key results
print("=== Final Prediction Results ===")
print(f"Predicted October Demand: {predicted_october_demand:.2f}")
print(f"Best Parameters: {best_params}")
print(f"Best Coefficients:  ={best_phi1:.5f},   ={best_phi2:.5f},   ={best_phi3:.5f}")
print(f"Best MSE across trials: {best_mse:.5f}")

"""RobustnessTest"""
```

```

# Best model parameters for robustness test
robustness_params = {
    'initial_temperature': 40,
    'alpha': 0.91,
    'iterations_per_t': 30
}

# Number of runs for robustness testing
num_runs = 10
mse_list = []
phi1_list = []
phi2_list = []

for _ in range(num_runs):
    # Create QUBO variables for  and
    n_bits = 8
    phi1_vars = kw.qubo.ndarray(n_bits, 'phi1', kw.qubo.binary)
    phi2_vars = kw.qubo.ndarray(n_bits, 'phi2', kw.qubo.binary)

    # Construct objective function (Sum of Squared Errors)
    obj = 0
    for t in range(2, len(diff_demands)):
        # Predicted value using AR(2) model
        pred_t = (binary_to_float(phi1_vars) * diff_demands[t - 1] +
                  binary_to_float(phi2_vars) * diff_demands[t - 2])

        # Add squared error term
        error = pred_t - diff_demands[t]
        obj += error * error

    # Parse QUBO
    obj = kw.qubo.make(obj)

    # Convert to QUBO matrix
    qubo_matrix = kw.qubo.qubo_model_to_qubo_matrix(obj)['qubo_matrix']

    # Solve using simulated annealing with the best parameters
    worker = kw.classical.SimulatedAnnealingOptimizer(
        initial_temperature=robustness_params['initial_temperature'],
        alpha=robustness_params['alpha'],
        cutoff_temperature=0.001, # Default value
        iterations_per_t=robustness_params['iterations_per_t'],
        size_limit=100
    )
    output = worker.solve(qubo_matrix)

    # Get optimal solution
    opt = kw.sampler.optimal_sampler(qubo_matrix, output, bias=0)
    best_solution = opt[0][0]

    # Manually create solution dictionary
    sol_dict = {}
    for i in range(n_bits):
        sol_dict[f'phi1[{i}]'] = best_solution[i]
        sol_dict[f'phi2[{i}]'] = best_solution[i + n_bits]

    # Extract coefficients from binary solutions
    phi1_binary = np.array([sol_dict[f'phi1[{i}]'] for i in range(n_bits)])
    phi2_binary = np.array([sol_dict[f'phi2[{i}]'] for i in range(n_bits)])

    phi1 = binary_to_float(phi1_binary)
    phi2 = binary_to_float(phi2_binary)

    # Calculate MSE for this parameter combination
    squared_errors = []
    for t in range(2, len(diff_demands)):
        predicted_diff = phi1 * diff_demands[t - 1] + phi2 * diff_demands[t - 2]
        actual_diff = diff_demands[t]
        squared_errors.append((predicted_diff - actual_diff) ** 2)

    mse = np.mean(squared_errors)

    # Store results for analysis
    mse_list.append(mse)
    phi1_list.append(phi1)
    phi2_list.append(phi2)

# Robustness analysis results
mean_mse = np.mean(mse_list)
std_mse = np.std(mse_list)

mean_phi1 = np.mean(phi1_list)
std_phi1 = np.std(phi1_list)

mean_phi2 = np.mean(phi2_list)
std_phi2 = np.std(phi2_list)

# Print robustness results
print(f"Robustness Test Results ({num_runs} runs):")
print(f"Mean MSE: {mean_mse:.4f}, Standard Deviation of MSE: {std_mse:.4f}")
print(f"Mean    : {mean_phi1:.4f}, Standard Deviation of : {std_phi1:.4f}")
print(f"Mean    : {mean_phi2:.4f}, Standard Deviation of : {std_phi2:.4f}")
```

Listing 1: Problem 1 Source Code

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import kaiwu as kw
import numpy as np
kw.license.init(user_id="72317291601100802",
    sdk_code="vDSsMrcS1XvoHxrKEyWGPu3y6bydtx")

```

```python
# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

# Train SVM classifier
svm = SVC(kernel='linear', random_state=42)
svm.fit(X_train, y_train)

# Make predictions
y_pred = svm.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of SVM model: {accuracy:.2f}")

# QUBO conversion and solved using quantum annealing and get the solution matrix,
# which is the corresponding weights and bias for the SVM.

# Load the Iris dataset
iris = load_iris()
X = iris.data # Feature data
y = iris.target # Labels

# Convert to binary classification
y_binary = np.where(y == 0, 1, -1) # Convert class 0 to 1 and others to -1

# Define dimensions
n_features = X.shape[1] # Number of features in the dataset
n_samples = X.shape[0] # Number of samples

# Define grid search ranges
penalty_coefficients = [0.1, 0.5, 1.0]
regularization_params = [0.01, 0.1, 1.0]
initial_temperatures = [100, 200, 300]
alphas = [0.90, 0.95, 0.99]
iterations_per_temps = [10, 50, 100]

# Initialize variables to store the best results
best_accuracy = 0
best_params = {}

# Perform grid search
for penalty_coefficient in penalty_coefficients:
    for C in regularization_params:
        for initial_temperature in initial_temperatures:
            for alpha in alphas:
                for iterations_per_t in iterations_per_temps:
                    # Create QUBO variables for weights and bias
                    weights = kw.qubo.ndarray(n_features, 'w', kw.qubo.Binary)
                    bias = kw.qubo.Binary('b')

                    # Formulate hinge loss and regularization explicitly
                    loss_terms = []
                    for i in range(n_samples):
                        margin = 1 - y_binary[i] * (np.dot(X[i], weights) + bias)
                        penalty = kw.qubo.Binary(f'margin_violation_{i}')
                        loss_terms.append(penalty_coefficient * penalty * margin)

                    hinge_loss = kw.qubo.quicksum(loss_terms)
                    regularization = kw.qubo.quicksum([w**2 for w in weights])
                    objective = hinge_loss + C * regularization

                    # Parse the QUBO model
                    qubo_model = kw.qubo.make(objective)
                    qubo_matrix_data = kw.qubo.qubo_model_to_qubo_matrix(qubo_model)
                    qubo_matrix = qubo_matrix_data['qubo_matrix']
                    variables_dict = qubo_matrix_data['variables']

                    # Solve using Simulated Annealing
                    solver = kw.classical.SimulatedAnnealingOptimizer(
                        initial_temperature=initial_temperature,
                        alpha=alpha,
                        cutoff_temperature=0.0001,
                        iterations_per_t=iterations_per_t
                    )
                    solution = solver.solve(qubo_matrix)
                    solution_dict = kw.qubo.get_sol_dict(solution[0], variables_dict)

                    # Extract weights and bias
                    weights_values = np.array([solution_dict.get(f'bw[{i}]', 0) for i in
                        range(n_features)])
                    bias_value = solution_dict.get('bb', 0)

                    # Predict and calculate accuracy
                    y_pred = np.sign(X @ weights_values + bias_value)
                    accuracy = accuracy_score(y_binary, y_pred)

                    # Update best parameters if current accuracy is better
                    if accuracy > best_accuracy:
                        best_accuracy = accuracy
                        best_params = {
                            'penalty_coefficient': penalty_coefficient,
                            'regularization_param': C,
                            'initial_temperature': initial_temperature,
                            'alpha': alpha,
                            'iterations_per_t': iterations_per_t,
                        }

print(best_accuracy, best_params)

# The robustness test and visualization:
```

```python

import kaiwu as kw
import numpy as np
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
# Load the Iris dataset
iris = load_iris()
X = iris.data # Feature data
y = iris.target # Labels

# Convert to binary classification (e.g., classify class 0 vs. rest)
y_binary = np.where(y == 0, 1, -1) # Convert class 0 to 1 and others to -1

# Define dimensions
n_features = X.shape[1] # Number of features in the dataset
n_samples = X.shape[0] # Number of samples

# Define hyperparameters for testing
penalty_coefficient = 0.5
regularization_param = 0.1
initial_temperature = 200
alpha = 0.95
iterations_per_t = 50
num_trials = 10 # Number of trials for robustness testing

# Function to train and test the QUBO-based model
def run_quantum_annealing_trial():
    # Create QUBO variables for weights and bias
    weights = kw.qubo.ndarray(n_features, 'w', kw.qubo.Binary)
    bias = kw.qubo.Binary('b')

    # Formulate hinge loss and regularization explicitly
    loss_terms = []
    for i in range(n_samples):
        margin = 1 - y_binary[i] * (np.dot(X[i], weights) + bias)
        penalty = kw.qubo.Binary(f'margin_violation_{i}')
        loss_terms.append(penalty_coefficient * penalty * margin)

    hinge_loss = kw.qubo.quicksum(loss_terms)
    regularization = kw.qubo.quicksum([w**2 for w in weights])
    objective = hinge_loss + regularization_param * regularization

    # Parse the QUBO model
    qubo_model = kw.qubo.make(objective)
    qubo_matrix_data = kw.qubo.qubo_model_to_qubo_matrix(qubo_model)
    qubo_matrix = qubo_matrix_data['qubo_matrix']
    variables_dict = qubo_matrix_data['variables']

    # Solve using Simulated Annealing
    solver = kw.classical.SimulatedAnnealingOptimizer(
        initial_temperature=initial_temperature,
        alpha=alpha,
        cutoff_temperature=0.0001,
        iterations_per_t=iterations_per_t
    )
    solution = solver.solve(qubo_matrix)
    solution_dict = kw.qubo.get_sol_dict(solution[0], variables_dict)

    # Extract weights and bias
    weights_values = np.array([solution_dict.get(f'bw[{i}]', 0) for i in
        range(n_features)])
    bias_value = solution_dict.get('bb', 0)

    # Predict and calculate accuracy
    y_pred = np.sign(X @ weights_values + bias_value)
    accuracy = accuracy_score(y_binary, y_pred)
    return accuracy

# Perform multiple trials
accuracies = []
for trial in range(num_trials):
    accuracy = run_quantum_annealing_trial()
    accuracies.append(accuracy)
    print(f"Trial {trial + 1}: Accuracy = {accuracy:.2f}")

# Analyze results
average_accuracy = np.mean(accuracies)
std_deviation = np.std(accuracies)

print(f"\nRobustness Test Results:")
print(f"Number of Trials: {num_trials}")
print(f"Average Accuracy: {average_accuracy:.2f}")
print(f"Standard Deviation: {std_deviation:.2f}")

def visualize_robustness(accuracies):
    # Plot histogram of accuracies
    plt.figure(figsize=(10, 5))
    plt.hist(accuracies, bins=10, color='blue', alpha=0.7, edgecolor='black')
    plt.title("Histogram of Accuracies from Robustness Test")
    plt.xlabel("Accuracy")
    plt.ylabel("Frequency")
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()

    # Plot boxplot of accuracies
    plt.figure(figsize=(6, 5))
    plt.boxplot(accuracies, vert=False, patch_artist=True,
                boxprops=dict(facecolor='blue', color='black', alpha=0.7),
                medianprops=dict(color='red', linewidth=2))
    plt.title("Boxplot of Accuracies from Robustness Test")
    plt.xlabel("Accuracy")
    plt.grid(axis='x', linestyle='--', alpha=0.7)
    plt.show()

# Visualize results
```

```
visualize_robustness(accuracies)
```

Listing 2: Problem 2 Source Code

```

import numpy as np
import kaiwu as kw
kw.license.init(user_id="72317291601100802",
    sdk_code="vDSsMrcS1XvoHxrKEyWGPu3y6bydtx")
from kaiwu.classical import SimulatedAnnealingOptimizer

import ssl
ssl._create_default_https_context = ssl._create_unverified_context

"""# First Model"""

import numpy as np
import random
import math
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load CIFAR-10 Dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define a simple CNN model for CIFAR-10
def build_cnn_model():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(layers.Flatten())
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(10, activation='softmax')) # Output layer for 10 classes
    return model

# Compile the model
model = build_cnn_model()
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

```

```python
# Train the model (for initial weights)
model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test, y_test))

# Extract the trained weights (kernel and bias) from the first Conv2D layer
conv2d_layer = model.layers[0] # Assuming the first Conv2D layer
kernel_weights, bias_weights = conv2d_layer.get_weights() # Extract kernel and bias

# binarized_kernel_weights = np.sign(kernel_weights) # Binarize the kernel weights to
    +1 or -1
# bias_weights will be left unchanged in this case
binarized_kernel_weights = kernel_weights

predictions = model.predict(x_test) # x_test is the CIFAR-10 test data
predicted_labels = np.argmax(predictions, axis=1) # Get the index with the highest
    probability for each sample
true_labels = np.argmax(y_test, axis=1)
accuracy = np.mean(predicted_labels == true_labels) # Compare predicted vs. true
    labels

# Print the accuracy
print(f"Test Set Accuracy: {accuracy * 100:.2f}%")

def test_robustness_cnn(model, x_test, y_test, noise_factor=0.1):
    """
    Tests the robustness of the CNN by adding random noise to the test data.
    """
    # Add random noise to the test data
    x_test_noisy = x_test + noise_factor * np.random.normal(0, 1, x_test.shape)
    x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0) # Ensure values are within [0, 1]

    # Evaluate the model on noisy test data
    test_loss_noisy, test_acc_noisy = model.evaluate(x_test_noisy, y_test, verbose=0)
    print(f"Robustness Test (CNN): Accuracy on noisy data = {test_acc_noisy *
        100:.2f}%")

    return test_acc_noisy

# Example usage:
cnn_accuracy_noisy = test_robustness_cnn(model, x_test, y_test)

def generate_qubo_matrix(weights,idx):
    # num_weights = len(weights)
    num_weights = 600
    Q = np.zeros((num_weights, num_weights))

    # Penalize large weights and encourage sparsity
    for i in range(num_weights):
        Q[i, i] = abs(weights[i+idx]) # Larger weights are penalized more

    return Q


initial_weights = []
```

```python
initial_biases = []
for layer in model.layers:
    if isinstance(layer, layers.Conv2D) or isinstance(layer, layers.Dense):
        kernel, bias = layer.get_weights()
        initial_weights.append(kernel.flatten()) # Flatten kernels (weights)
        initial_biases.append(bias.flatten()) # Flatten biases

# Concatenate the initial weights and biases
initial_weights = np.concatenate(initial_weights)
initial_biases = np.concatenate(initial_biases)

binary_weights_all = np.zeros(1)
for i in range(int(initial_weights.shape[0] / 600)):
    idx = i*600
    Q = Q = generate_qubo_matrix(initial_weights,idx)
    solver = SimulatedAnnealingOptimizer()

    # Solve the QUBO problem
    solution = solver.solve(Q)
    best_solution = solution[0]

    optimized_x = best_solution
    binary_weights = np.zeros(600)
    binary_weights[optimized_x == 1] = initial_weights[i*600:i*600+600][optimized_x
        == 1]
    binary_weights_all = np.concatenate((binary_weights_all, binary_weights))

binary_weights_all_new = binary_weights_all[1:]
binary_weights_all_new = np.concatenate((binary_weights_all_new, np.array([0] *
    (len(initial_weights) - len(binary_weights_all_new)))))

# Reconstruct weights and biases for each layer
layer_idx_w = 0
layer_idx_b = 0
updated_weights = []
updated_biases = []
for layer in model.layers:
    if isinstance(layer, layers.Conv2D) or isinstance(layer, layers.Dense):
        num_params = layer.get_weights()[0].size # Kernel size
        num_biases = layer.get_weights()[1].size # Bias size

        # Get the kernel weights for the layer
        layer_kernel = binary_weights_all_new[layer_idx_w:layer_idx_w +
            num_params].reshape(layer.get_weights()[0].shape)
        updated_weights.append(layer_kernel)

        # Get the biases for the layer
        # layer_bias = binary_weights_all_new[layer_idx + num_params:layer_idx +
            num_params + num_biases].reshape(layer.get_weights()[1].shape)
        layer_bias = initial_biases[layer_idx_b:layer_idx_b +
            num_biases].reshape(layer.get_weights()[1].shape)
        updated_biases.append(layer_bias)

        # Move to the next set of weights
        # layer_idx += num_params + num_biases
        layer_idx_w += num_params
        layer_idx_b += num_biases

# Now set the weights and biases to the model
 updated_weights = [w for w in updated_weights]
 updated_biases = [b for b in updated_biases]

 # Set the new weights and biases for the model
 model.set_weights([w for pair in zip(updated_weights, updated_biases) for w in pair])

 # Verify the model summary after setting the weights
 model.summary()

 model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test, y_test))

 # Evaluate the model on the test set
 test_loss, test_acc = model.evaluate(x_test, y_test)
 print(f"Test accuracy: {test_acc * 100:.2f}%")

 cnn_accuracy_noisy = test_robustness_cnn(model, x_test, y_test)

 """# Second Model"""

 import numpy as np
 import tensorflow as tf
 from tensorflow.keras import layers, models

 # Load CIFAR-10 dataset
 (x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

 # Preprocess the data
 x_train = x_train.astype('float32') / 255.0
 x_test = x_test.astype('float32') / 255.0
 y_train = tf.keras.utils.to_categorical(y_train, 10)
 y_test = tf.keras.utils.to_categorical(y_test, 10)

 # Define the deep learning model (a simple CNN)
 def create_model():
     model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(10, activation='softmax')
     ])
     model.compile(optimizer='adam', loss='categorical_crossentropy',
        metrics=['accuracy'])
     return model

```

```python
# Define a function to create a more complex QUBO matrix


def create_complex_qubo(weights, idx,sparsity_factor=0.1, interaction_factor=0.01,
     entropy_factor=0.1):
    # n_weights = len(weights)
    n_weights = 600
    Q = np.zeros((n_weights, n_weights))

    # Add sparsity regularization (penalize large weights, encourage sparsity)
    for i in range(n_weights):
        Q[i, i] += sparsity_factor * weights[i+idx] ** 2 # Penalize non-zero weights

    # Add interaction regularization (encourage some weights to be correlated or
        anticorrelated)
    for i in range(n_weights):
        for j in range(i + 1, n_weights):
            Q[i, j] += interaction_factor * weights[i+idx] * weights[j+idx] # Penalize
                uncorrelated weights

    # Add entropy minimization (encourage confident predictions)
    for i in range(n_weights):
        for j in range(i + 1, n_weights):
            Q[i, j] += entropy_factor * (weights[i+idx] - 0.5) * (weights[j+idx] -
                0.5) # Enforce weights to be either 0 or 1

    return Q

# Initialize the model and get the weights
model = create_model()
initial_weights = []
initial_biases = []
for layer in model.layers:
    if isinstance(layer, layers.Conv2D) or isinstance(layer, layers.Dense):
        kernel, bias = layer.get_weights()
        initial_weights.append(kernel.flatten()) # Flatten kernels (weights)
        initial_biases.append(bias.flatten()) # Flatten biases

# Concatenate the initial weights and biases
initial_weights = np.concatenate(initial_weights)
initial_biases = np.concatenate(initial_biases)


# Create a more complex QUBO matrix
binary_weights_all = np.zeros(1)
for i in range(int(initial_weights.shape[0] / 600)):
    idx = i*600
    Q = create_complex_qubo(initial_weights,idx)
    solver = SimulatedAnnealingOptimizer()

    # Solve the QUBO problem
    solution = solver.solve(Q)
    best_solution = solution[0]

    optimized_x = best_solution
    binary_weights = np.zeros(600)
    binary_weights[optimized_x == 1] = initial_weights[i*600:i*600+600][optimized_x
        == 1]
    binary_weights_all = np.concatenate((binary_weights_all, binary_weights))

binary_weights_all_new = binary_weights_all[1:]
binary_weights_all_new = np.concatenate((binary_weights_all_new, np.array([0] *
    (len(initial_weights) - len(binary_weights_all_new)))))

# Reconstruct weights and biases for each layer
layer_idx_w = 0
layer_idx_b = 0
updated_weights = []
updated_biases = []
for layer in model.layers:
    if isinstance(layer, layers.Conv2D) or isinstance(layer, layers.Dense):
        num_params = layer.get_weights()[0].size # Kernel size
        num_biases = layer.get_weights()[1].size # Bias size

        # Get the kernel weights for the layer
        layer_kernel = binary_weights_all_new[layer_idx_w:layer_idx_w +
            num_params].reshape(layer.get_weights()[0].shape)
        updated_weights.append(layer_kernel)

        # Get the biases for the layer
        # layer_bias = binary_weights_all_new[layer_idx + num_params:layer_idx +
            num_params + num_biases].reshape(layer.get_weights()[1].shape)
        layer_bias = initial_biases[layer_idx_b:layer_idx_b +
            num_biases].reshape(layer.get_weights()[1].shape)
        updated_biases.append(layer_bias)

        # Move to the next set of weights
        # layer_idx += num_params + num_biases
        layer_idx_w += num_params
        layer_idx_b += num_biases

# Now set the weights and biases to the model
updated_weights = [w for w in updated_weights]
updated_biases = [b for b in updated_biases]

# Set the new weights and biases for the model
model.set_weights([w for pair in zip(updated_weights, updated_biases) for w in pair])

# Verify the model summary after setting the weights
model.summary()

model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test,
    y_test))

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
```

```
print(f"Test accuracy: {test_acc * 100:.2f}%")

cnn_accuracy_noisy = test_robustness_cnn(model, x_test, y_test)
```

Listing 3: Problem 3 Source Code