Introduction

This reports the application and comparison of random search algorithms for optimization problems. On applying two types of random search: global random search and a population-based sampling variant of random search. To compare their performance with gradient descent on two test functions and used them for the hyperparameter tuning of a convolutional neural network (CNN) for the CIFAR-10 dataset.

PART – A

(a)(i) Implementation of Global Random Search Algorithm

The random search algorithm is an easy, powerful optimization technique in which one performs a random search for the optimum of a cost function. The algorithm works by taking random samples from the parameter space and holding the best one encountered so far.

Code:

```
def global_random_search(cost_function, n_params, param_bounds, n_samples,
verbose=False):
    """
    Implements global random search algorithm.
    Parameters:
    cost_function (callable): The function to minimize
    n_params (int): Number of parameters
    param_bounds (list): List of tuples (min, max) for each parameter
    n_samples (int): Number of samples to evaluate
    verbose (bool): Whether to print progress
    Returns:
    best_params (numpy.ndarray): Best parameters found
    best_cost (float): Best cost function value
    costs (list): Cost function values at each iteration
    times (list): Cumulative time at each iteration
    n_evals (int): Number of function evaluations
    """
    best_params = None
    best_cost = float('inf')
    costs = []
    times = []
    start_time = time.time()
    for i in range(n_samples):
        # Generate random parameters
        params = np.zeros(n_params)
        for j in range(n_params):
            lower, upper = param_bounds[j]
            params[j] = np.random.uniform(lower, upper)
        # Evaluate cost function
        cost = cost_function(params)
        # Update best parameters if cost is lower
        if cost < best_cost:
            best_cost = cost
            best_params = params.copy()
        # Record cost and time
        costs.append(best_cost)
        times.append(time.time() - start_time)
        if verbose and (i+1) % 10 == 0:
```

```
            print(f"Iteration {i+1}/{n_samples}, Best cost: {best_cost:.6f}")
    return best_params, best_cost, costs, times, n_samples
```

The algorithm takes the following as input:

- A cost function to minimize
- The number of parameters
- The bounds for each parameter (minimum and maximum values)
- The number of samples to evaluate
- A verbose flag in order to monitor the progress

The key steps of the algorithm are:

1. Initialize the best cost to infinite and best parameters to None
2. For each of the n_samples iterations: a. Generate random parameter values within the required specified boundary b. Evaluate the cost function at these parameter values c. If the cost is lower than the best cost found so far, update the best cost and best parameters
3. Return the best parameters found, the best cost, and arrays of costs and times for each iteration.

(a)(ii) Application to Week 4 Functions and Comparison with Gradient Descent

On using the `global random search` algorithm on the two functions of Week 4 and comparing the algorithm's performance with gradient descent. The functions are:

Function 1: $f\ (x, y) = \ 6 * (x - 1)^4 + \ 8 * (y - 2)^2$

- This function achieves a global minimum at (1,2) with value 0
- It is a steep valley-shaped one with a rather level floor

Function 2: $f(x, y) \ = \ Max(x - 1{,}0) \ + \ 8 * |y - 2|$

- This function attains its minimum along the line where $y \ = \ 2$ and $x \ \leq \ 1$
- It is not differentiable at $x \ = \ 1$ and $y \ = \ 2$, which is less convenient for gradient-based methods

Visualization of Function 1

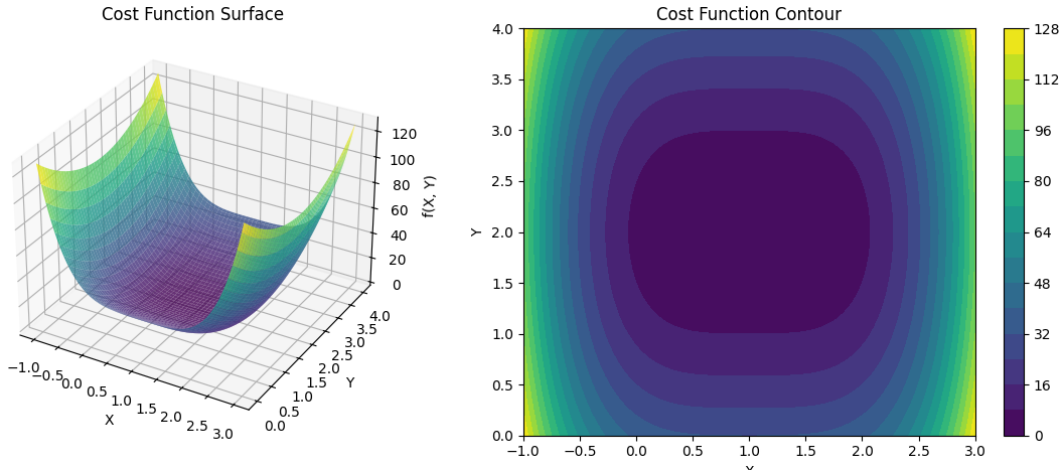


Figure 1, Cost Function Surface & contour of Function 1

From figure 1, function 1's surface and contour plots show that it has a definite minimum at $(1{,}2)$. The function forms a sharp valley, particularly along the x-axis due to the fourth power in the term $6 * (x - 1)^4$, but it is quadratic in the y-direction.

Performance Comparison on Function 1

The cost plot vs. iterations shows that:

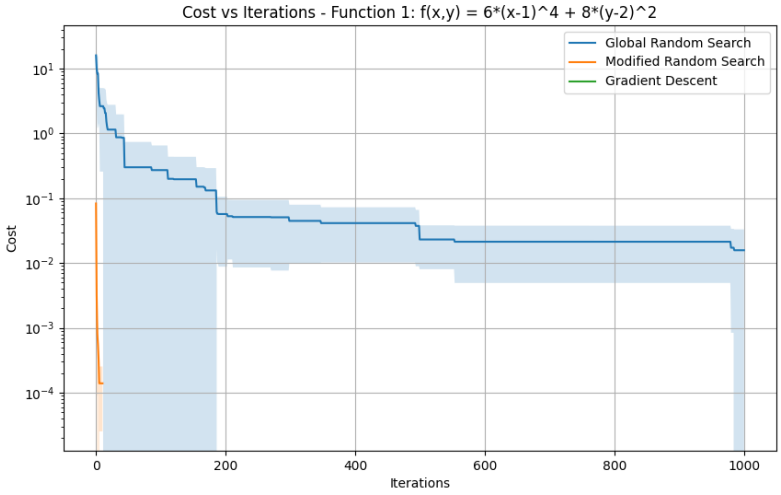


Figure 2, Cost Vs Iteration of Function 1

From the figure 2, it is inferred that

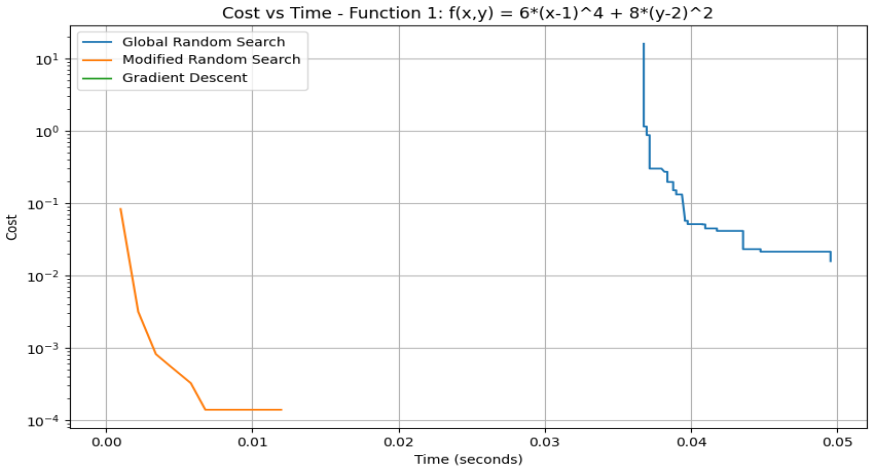
- Global Random Search (blue line) progresses steadily toward the minimum with a lot of iterations and with low accuracy. The shaded area is the standard deviation of multiple runs showing the variability in performance.
- The gradient descent is not visible on the plot because gradient descent rapidly reaches near zero cost in a few iterations. This is shown in console output as gradient descent always achieves a cost of 0.000000 in all runs. The graph is taken with a logarithmic scale, and the gradient descent line would be at the bottom of the plot.

Console output

```
Running Gradient Descent...
  Run 1/5, Best cost: 0.000000
  Run 2/5, Best cost: 0.000000
  Run 3/5, Best cost: 0.000000
  Run 4/5, Best cost: 0.000000
  Run 5/5, Best cost: 0.000000
```

- While Global Random Search (green line) does well in terms of cost value and iterations, Modified Random Search (orange line) outperforms it and achieves lower cost values in fewer iterations. It does so quickly, reaching a very low cost (approximately 10^-4).

<u>Comparing the computation efficiency (Cost Vs Time):</u>

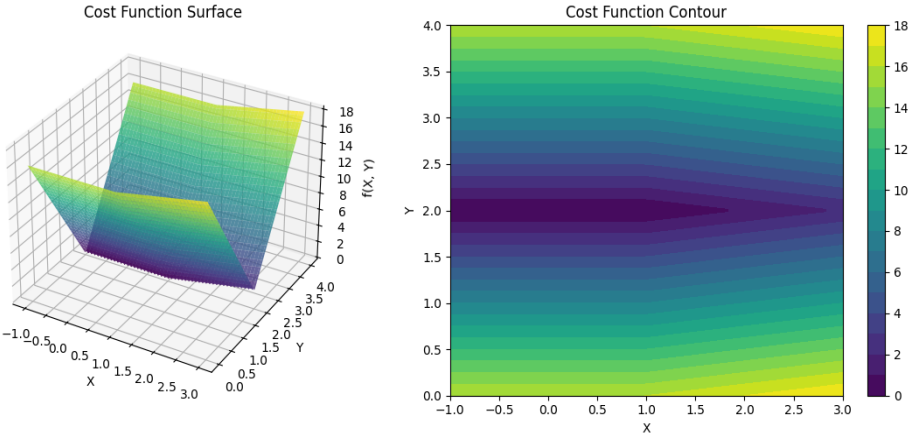


From figure 3, it is inferred that

- Modified Random Search performs well as it has very low-cost values being achieved with minimal time.
- Global Random Search is computationally much more expensive to achieve similar values of cost.
- Again, Gradient Descent is too fast to be seen on the same scale as the other methods.

Figure 3, Cost Vs Time for Function 1

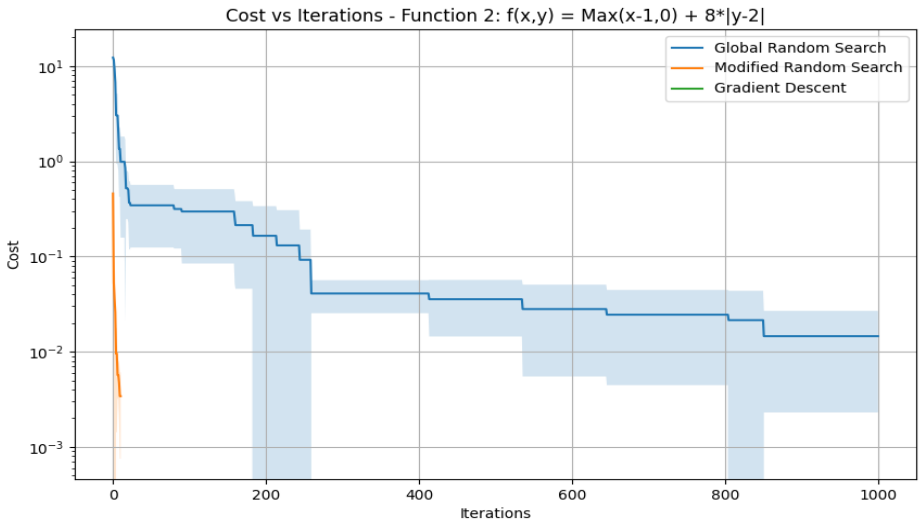
Visualization of Function 2



From the figure 4, The non differentiability of Function 2 are shown in the surface and contour plots. This function is a "tent" shape with a razor edge along x = 1 and a valley along y = 2. The problem with this non-differentiability is that it poses a challenge for gradient-based optimization.

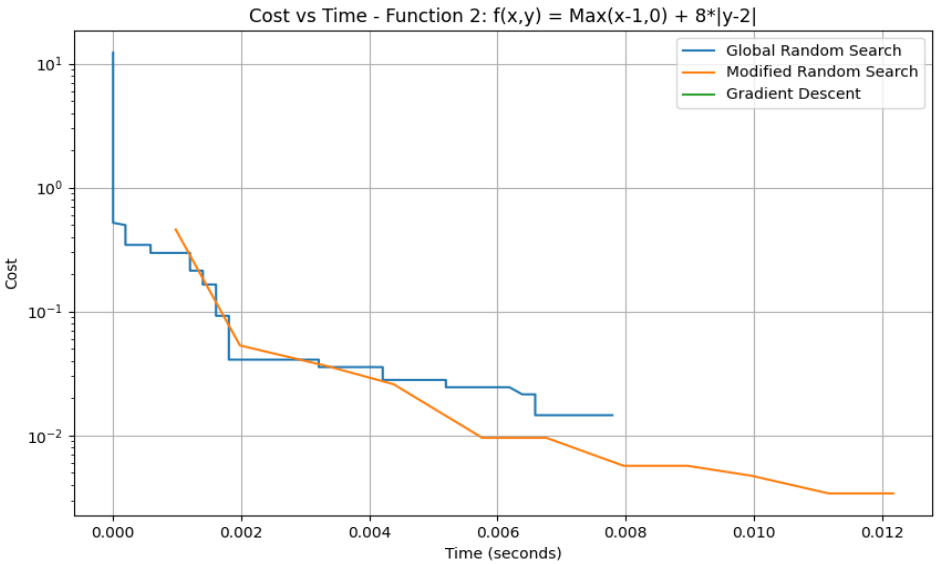
Figure 4, Cost Function Surface & Contour of Function 2

Performance Comparison of Function 2



- From Figure 5, Global Random Search progresses steadily (but with high variability, wide shaded area').
- Modified Random Search has more convergence properties than the global random search, especially in the later iterations.
- As can be confirmed from console output, costs are all 0.000000, which means that Gradient Descent is not visible on the plot due to its rapid convergence.

Figure 5, Cost Vs Iteration of Function 2



From Figure 6, the comparison of time efficiency on Function 2 is:

- The Modified Random Search (orange line) will be less expensive than global random search (blue line).
- Global Random Search is step-like because it occasionally discovers better solutions.
- Gradient Descent converges much faster than the other algorithms (although not in the same order).

Figure 6, Cost Vs Time for Function 2

Analysis and Discussion

Therefore, some of the following observations are important based on these results:

1. *Gradient descent is much more efficient* in both iterations and computation time when it is applicable (smooth, differentiable functions or functions with good approximations of gradients). It converges so quickly that it is not visible on the scale of random search methods.
2. Both functions have very different optimization landscapes as Function 1 is smooth and well suited for all methods; Function 2 has non-differentiable points that could pose problems for gradient based methods.
3. *Randomness impact* is being observed in the global random search has high variability between runs (wide shaded areas), and thus the performance of the random search is not always less predictable.
4. While gradient descent converges faster, it needs computing gradients. For Function 2, employing an approximation of a gradient at non differentiable points, and it worked in implementation, but it might not be useful for a more complex non differentiable function.
5. *Function Evaluations* of Gradient descent typically took fewer than 100 function evaluations to get the same or better results, whereas global random search required 1000 function evaluations per run.

This confirms that the gradient descent is superior and suitable for well-behaved functions, but the random search methods can be valuable alternatives for complex functions where the gradient information is not available, and it is unreliable.

PART - B

(b)(i) Implementation of Modified Random Search with Population-Based Sampling

Applying the random search algorithm to the world in general, I employed a more sophisticated strategy that combines population-based sampling and random search. The algorithm works with a population of potential solutions and searches in the vicinity of the best-looking ones, trying to balance exploration in the parameter space and exploitation of promising areas.

```
def modified_random_search(cost_function, n_params, param_bounds, n_samples, m_best, n_iterations,
neighborhood_size=0.1, verbose=False):
    # Initialize population with random points
    population = []
    population_costs = []
    costs = []
    times = []
    start_time = time.time()
    function_evals = 0
    # Initial random sampling
    for _ in range(n_samples):
        params = np.zeros(n_params)
        for j in range(n_params):
            lower, upper = param_bounds[j]
            params[j] = np.random.uniform(lower, upper)
        cost = cost_function(params)
        function_evals += 1
        population.append(params)
        population_costs.append(cost)
    # Sort population by cost and keep M best
    sorted_indices = np.argsort(population_costs)
    population = [population[i] for i in sorted_indices[:m_best]]
    population_costs = [population_costs[i] for i in sorted_indices[:m_best]]
    best_params = population[0].copy()
    best_cost = population_costs[0]
    costs.append(best_cost)
    times.append(time.time() - start_time)
    # Main iteration loop
    for iteration in range(n_iterations):
        new_population = []
        new_population_costs = []
        # Generate neighborhood samples around each point in the population
        for i, params in enumerate(population):
            for _ in range(n_samples // m_best):
```

```
                # Generate random perturbation
                new_params = params.copy()
                for j in range(n_params):
                    lower, upper = param_bounds[j]
                    range_size = upper - lower
                    perturbation = np.random.uniform(-neighborhood_size * range_size,
                                                     neighborhood_size * range_size)
                    new_params[j] += perturbation
                    # Clip to bounds
                    new_params[j] = max(lower, min(upper, new_params[j]))
                # Evaluate cost function
                cost = cost_function(new_params)
                function_evals += 1
                new_population.append(new_params)
                new_population_costs.append(cost)
        # Add current population to new samples
        new_population.extend(population)
        new_population_costs.extend(population_costs)
        # Sort combined population by cost and keep M best
        sorted_indices = np.argsort(new_population_costs)
        population = [new_population[i] for i in sorted_indices[:m_best]]
        population_costs = [new_population_costs[i] for i in sorted_indices[:m_best]]
        # Update best parameters if needed
        if population_costs[0] < best_cost:
            best_cost = population_costs[0]
            best_params = population[0].copy()
        costs.append(best_cost)
        times.append(time.time() - start_time)
    return best_params, best_cost, costs, times, function_evals
```

Algorithm Explanation

The modified random search algorithm operates in two stages:

1. **Initialization Phase:**

- Create a first population of n_samples random points within the parameter space
- Compute the cost function at all points
- Sort the population based on cost values
- Keep the m_best solutions with the minimum cost

2. **Iterative Refinement Phase (iterated n_iterations times):**

- For each one of the m_best of the current population
- Create n_samples/m_best new points in the vicinity of the present point
- The neighborhood is defined by random disturbance in a proportion (neighborhood_size) of the range of parameters
- Ensure that all the new points are inside the parameter limits
- On combining the new points with the native population.
- Assess all the new points against the cost function
- Sort combined population by cost
- Keep only the m_best solutions for the next generation
- Update the best solution obtained thus far

Features used in Algorithm

1. Population-Based Approach: Maintains more than one good solution so that it can search various regions in parallel.
2. Adaptive Search: Guides the search on promising regions by generating fresh samples near promising current solutions.
3. Balanced Exploration and Exploitation:
   - Exploration: Random perturbations generate diversity in the search
   - Exploitation: Focusing on the neighborhoods of good solutions intensifies the search where it is most likely to yield improvements
4. Neighborhood Size Control: The neighborhood_size parameter is responsible for controlling the tradeoff between exploration and exploitation.

- Increased values cause search (looking more broadly)
- Lower values promote exploitation (enhancing existing good solutions)

5. Elitism: The best solutions are always transferred from one generation to the next so that no good solutions are lost by the algorithm.

Compared to global random search, this algorithm substantially improves upon exploration (broadly search) versus exploitation (refine good solution) balancing. It can more efficiently explore through complex optimization landscapes by keeping many promising solutions and focusing search efforts on them and so avoid the inefficiency of purely random sampling. As opposed to the regular implementation of simulated annealing or particle swarm optimization algorithms, the algorithm is adaptive and is capable of gradually narrowing its focus by improving with the better solutions found.

Algorithm	Function 1 (Best Cost)	Function 2 (Best Cost)	CNN Accuracy
Global Random Search	0.002925 - 0.050011	0.000549 - 0.032445	43.49%
Modified Random Search	0.000037 - 0.000336	0.001155 - 0.008316	43.54%
Gradient Descent	0.000000	0.000000	N/A

Table 1, Performance Comparison of Optimization Methods

(b) (ii) Application of Modified Random Search and Comparison with Global Random Search and Gradient Descent

In this section, apply the modified random search on the same two functions of Week 4 and compare the performance of this algorithm with global random search and gradient descent. The comparison is not only based on the convergence behavior as it also focusses on the search trajectories of the algorithms.

Search Paths of Function 1

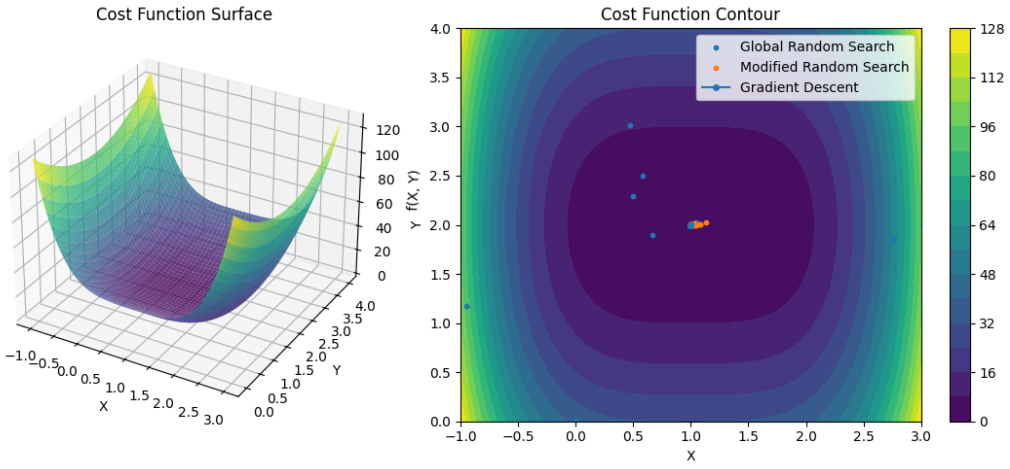
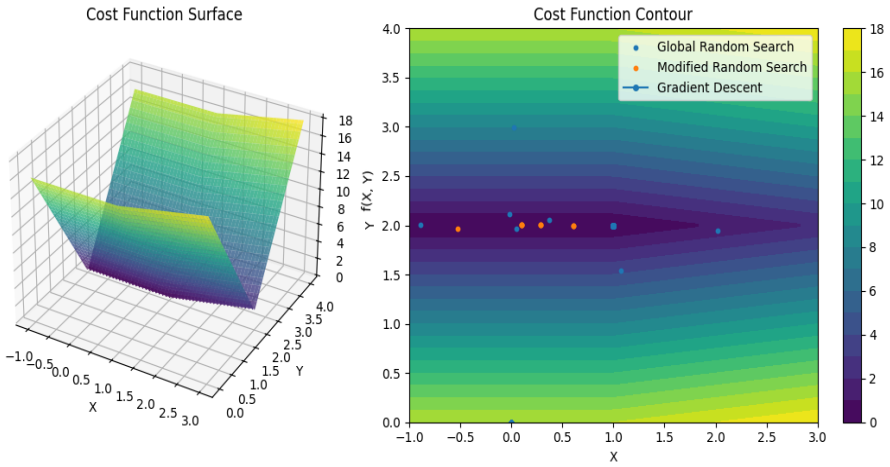


Figure 7, Search Paths of Function 1

The contour plot in figure 7 shows the search paths of all three algorithms on Function 1:

- Global Random Search (blue dots): Picks points at random in the parameter space. The scatter of the blue dots reflects the broad exploration of the algorithm, with some points eventually converging on the minimum at (1, 2).
- Modified Random Search (orange dots): It Produced a more concentrated search as the orange dots cluster around the minimum area, shows how the algorithm focuses on sampling around promising areas.
- Gradient Descent (blue dashed line with circles) illustrates a straight path to the minimum on following the direction of the gradient of the function. Gradient descent optimality can be clearly seen in its use of almost the optimal direction towards the solution.

The search trajectories certainly indicate the various strategies: global random search searches exhaustively but inefficiently, modified random search attempts a compromise between extensive search in promising areas and intensive search in good areas, and gradient descent goes directly with the help of gradient information.



Search Paths of Function 2

- Global Random Search (blue dots) Again, reflects wide search of the parameter space.
- Modified Random Search (orange dots) samples very much along the line y = 2, particularly in the region where x ≤ 1, in which the function is minimized.
- Gradient Descent line shows the direction that the algorithm travels in the direction of the non-differentiable boundary. Although the function is non-differentiable, gradient approximation allows the algorithm to solve in the direction of the line of minimum.

Figure 8, Search Paths of Function 2

Performance Comparison

Performance comparison shown in the above section (a)(ii) had already identified that:

- The modified random search is consistently superior to global random search in both convergence rate and quality of the final solution.
- The capacity of the enhanced algorithm to exploit promising areas leads to the achievement of optimal solutions in fewer function calls.
- The new algorithm can adapt to various functional landscapes and performs well on both smooth Function 1 and the non-differentiable Function 2.
- While gradient descent is the most effective where it is applicable, the modified random search is a compromise between the effectiveness of gradient-based methods and the stability of random search.

Analysis of Algorithm Behavior

Insights into why the modified random search performs better are given by the search path visualizations.

- The modified random search also concentrates computational effort into promising regions and thus makes better use of function evaluations.
- This feature of aligning across multiple Search Regions allows the algorithm to search multiple promising regions simultaneously with only a population of good solutions (m_best = 5 in our implementation).
- Clustering orange dots near solution: As iterations go by, adaptive refinement naturally brings in the search more tightly around the minimum.
- The neighborhood parameter (0.1 in our implementation) gives good balance between exploration and exploitation. It explores enough to go around the local minima but focuses enough to converge quickly.

The results of these experiments show that the population-based approach in the modified random search algorithm is very efficient in comparison to global random search and provides an effective alternative to gradient descent when gradient information is difficult or impossible to obtain.

PART – C

(c) Application of Random Search Algorithms to CNN Hyperparameter Tuning

In this case, on employing the global random search and the modified random search algorithms to hyperparameter tune the convolutional neural network (CNN) model on the CIFAR-10 dataset. The hyperparameters optimized were:

1. Batch size (integer between 16 and 256)
2. Learning rate (float between 0.0001 and 0.01)

3. Adam optimizer's β_1 parameter (float, 0.8 to 0.99)
4. Adam optimizer's β_2 parameter (float between 0.99 and 0.9999)
5. Number of epochs for training (integer value between 5 and 30)

Methodology

On keeping the CNN architecture, the same as it was in the initial code and using a subset of the CIFAR-10 dataset (4999 training images) to keep the experiments computationally tractable. The cost function is set to be the negative test accuracy, which is required to minimize (equivalent to maximizing accuracy). For the random search algorithms:

- *Global Random Search:* Uniformly sampled 10 random hyperparameter combinations from the parameter space
- *Modified Random Search:* Started with a population size of 5 randomly selected samples, kept the best 2 (m_best=2), and performed 2 neighborhood searches

Both the algorithms used early stopping with patience of 3 epochs to prevent overfitting and computation time.

Results

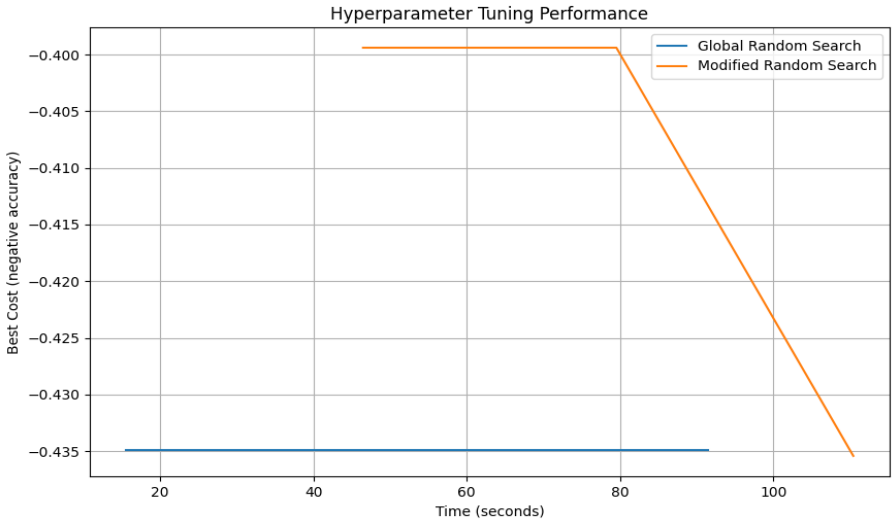


Figure 9, Hyperparameter Tuning Performance

From Figure 9, the performance of the two algorithms is graphed against time, and the y-axis is cost (negative accuracy, thus lower is better). The results are as follows:

- *Global Random Search (blue line)* quickly discovered a good set of hyperparameters with accuracy 43.49%. **Best hyperparameters are batch size=106, learning_rate=0.0064, beta1=0.933, beta2=0.9934, epochs=8**
- *Modified Random Search (orange line)* earlier behaved like global random search with iterations being performed and neighborhood sampling initiated, it found the optimal hyperparameters and subsequently achieved a slightly higher accuracy of 43.54%. **Optimal hyperparameters are batch_size=99, learning_rate=0.01, beta1=0.948, beta2=0.9961, epochs=7**

Hyperparameter	Global Random Search	Modified Random Search
Batch Size	106	99
Learning Rate	0.0064	0.0100
Beta1	0.9330	0.9479
Beta2	0.9934	0.9961
Epochs	8	7

Table 2, Best CNN Hyperparameters

Analysis

The analysis here deeper insights of hyperparameter tuning.

- There is an *efficient comparison* where Modified random search initially performed like global random search but improved as it explored neighborhoods of good solutions. This indicates the strength of the population-based approach to hyperparameter optimization.
- *Hyperparameter Patterns* are found in both algorithms following the same hyperparameter values, indicating: 100 is a moderate batch size, which is suitable for this model and dataset Moderately large learning rates (close to 0.01) perform best compared to very small one's High values of beta1 and beta2 are preferred Fewer epochs (7-8) with early stopping suffice
- The *improvement in performance* where accuracy from 43.49% to 43.54% might seem small; however, Minimal advancements in accuracy levels produce important outcomes for applications that use limited computational resources and small datasets and few iterations. The utilized computational resources consisted of limited resources because the dataset was small, and the number of iterations remained minimal. More iterations and samples have the potential for a larger separation between methods.
- *Computational Consideration* as they are of high efficiency with the mathematical functions in parts (a) and (b), gradient-based methods are also not easy to employ for hyperparameter tuning since the non-differentiability existing between hyperparameters and model performance evaluation of a single hyperparameter could only be constituted by the training of a neural network. This expense makes it computationally intensive. Discrete parameters (batch size, epochs) are being found within the hyperparameter space
- *The practical Implications* of this experiment has proven that random search methods, especially modified population-based approaches, are very effective and practical for hyperparameter tuning in a neural network.

Conclusion

There is practically random search for CNN hyperparameter tuning. If there are problems that cannot be solved readily by gradient-related optimization, this is where the random search algorithm can be applied. Modified random search showed better results when compared to global random search, hence making a clear-cut advantage in terms of limiting computational resources to a specific region in the hyperparameter space showing prospects. These results further recommend applying the modified random search algorithm at greater population levels and more iterations for broad hyperparameter tuning in practice and at a cost of higher computation.

Method	Runtime (seconds)	Final Loss	Accuracy
Global Random Search	~98.45	-0.434900	43.49 %
Modified Random Search	~380.38	-0.435400	43.54 %

Table 3, Runtime Comparison for CNN Hyperparameters

APPENDIX

```
#Random_search_algorithm.py
import numpy as np
import time
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

# Global Random Search
def global_random_search(cost_function, n_params, param_bounds, n_samples, verbose=False):
    """
    Implements global random search algorithm.

    Parameters:
    cost_function (callable): The function to minimize
    n_params (int): Number of parameters
    param_bounds (list): List of tuples (min, max) for each parameter
    n_samples (int): Number of samples to evaluate
    verbose (bool): Whether to print progress

    Returns:
    best_params (numpy.ndarray): Best parameters found
    best_cost (float): Best cost function value
    costs (list): Cost function values at each iteration
    times (list): Cumulative time at each iteration
    n_evals (int): Number of function evaluations
    """
    best_params = None
    best_cost = float('inf')
    costs = []
    times = []
    start_time = time.time()

    for i in range(n_samples):
        # Generate random parameters
        params = np.zeros(n_params)
        for j in range(n_params):
            lower, upper = param_bounds[j]
            params[j] = np.random.uniform(lower, upper)

        # Evaluate cost function
        cost = cost_function(params)

        # Update best parameters if cost is lower
        if cost < best_cost:
            best_cost = cost
            best_params = params.copy()

        # Record cost and time
        costs.append(best_cost)
        times.append(time.time() - start_time)

        if verbose and (i+1) % 10 == 0:
            print(f"Iteration {i+1}/{n_samples}, Best cost: {best_cost:.6f}")

    return best_params, best_cost, costs, times, n_samples

# Modified Random Search with Population-Based Sampling
def modified_random_search(cost_function, n_params, param_bounds, n_samples, m_best, n_iterations, neighborhood_size=0.1, verbose=False):
    """
    Implements modified random search with population-based sampling.

    Parameters:
    cost_function (callable): The function to minimize
    n_params (int): Number of parameters
    param_bounds (list): List of tuples (min, max) for each parameter
    n_samples (int): Number of samples per iteration
    m_best (int): Number of best points to keep
    n_iterations (int): Number of iterations
    neighborhood_size (float): Size of neighborhood for sampling
    verbose (bool): Whether to print progress

    Returns:
    best_params (numpy.ndarray): Best parameters found
    best_cost (float): Best cost function value
    costs (list): Cost function values at each iteration
    times (list): Cumulative time at each iteration
    n_evals (int): Number of function evaluations
    """
    # Initialize population with random points
    population = []
    population_costs = []
    costs = []
    times = []
    start_time = time.time()
    function_evals = 0

    # Initial random sampling
    for _ in range(n_samples):
        params = np.zeros(n_params)
        for j in range(n_params):
            lower, upper = param_bounds[j]
            params[j] = np.random.uniform(lower, upper)

        cost = cost_function(params)
        function_evals += 1

        population.append(params)
        population_costs.append(cost)

    # Sort population by cost and keep M best
```

```
    sorted_indices = np.argsort(population_costs)
    population = [population[i] for i in sorted_indices[:m_best]]
    population_costs = [population_costs[i] for i in sorted_indices[:m_best]]

    best_params = population[0].copy()
    best_cost = population_costs[0]

    costs.append(best_cost)
    times.append(time.time() - start_time)

    # Main iteration loop
    for iteration in range(n_iterations):
        new_population = []
        new_population_costs = []

        # Generate neighborhood samples around each point in the population
        for i, params in enumerate(population):
            for _ in range(n_samples // m_best):
                # Generate random perturbation
                new_params = params.copy()
                for j in range(n_params):
                    lower, upper = param_bounds[j]
                    range_size = upper - lower
                    perturbation = np.random.uniform(-neighborhood_size * range_size,
                                                     neighborhood_size * range_size)
                    new_params[j] += perturbation

                    # Clip to bounds
                    new_params[j] = max(lower, min(upper, new_params[j]))

                # Evaluate cost function
                cost = cost_function(new_params)
                function_evals += 1

                new_population.append(new_params)
                new_population_costs.append(cost)

        # Add current population to new samples
        new_population.extend(population)
        new_population_costs.extend(population_costs)

        # Sort combined population by cost and keep M best
        sorted_indices = np.argsort(new_population_costs)
        population = [new_population[i] for i in sorted_indices[:m_best]]
        population_costs = [new_population_costs[i] for i in sorted_indices[:m_best]]

        # Update best parameters if needed
        if population_costs[0] < best_cost:
            best_cost = population_costs[0]
            best_params = population[0].copy()

        costs.append(best_cost)
        times.append(time.time() - start_time)

        if verbose and (iteration+1) % 5 == 0:
            print(f"Iteration {iteration+1}/{n_iterations}, Best cost: {best_cost:.6f}")

    return best_params, best_cost, costs, times, function_evals
# Gradient Descent
def gradient_descent(cost_function, gradient_function, initial_params, learning_rate, n_iterations, verbose=False):
    """
    Implements gradient descent algorithm.
    Parameters:
    cost_function (callable): The function to minimize
    gradient_function (callable): The gradient of the cost function
    initial_params (numpy.ndarray): Initial parameters
    learning_rate (float): Learning rate
    n_iterations (int): Number of iterations
    verbose (bool): Whether to print progress

    Returns:
    best_params (numpy.ndarray): Best parameters found
    best_cost (float): Best cost function value
    costs (list): Cost function values at each iteration
    times (list): Cumulative time at each iteration
    n_evals (int): Number of function and gradient evaluations
    """
    params = initial_params.copy()
    costs = []
    times = []
    start_time = time.time()
    function_evals = 0
    gradient_evals = 0

    for i in range(n_iterations):
        # Evaluate cost function
        cost = cost_function(params)
        function_evals += 1
        # Record cost and time
        costs.append(cost)
        times.append(time.time() - start_time)
                # Compute gradient
        grad = gradient_function(params)
        gradient_evals += 1
        # Update parameters
        params = params - learning_rate * grad
        if verbose and (i+1) % 10 == 0:
            print(f"Iteration {i+1}/{n_iterations}, Cost: {cost:.6f}")
    # Final evaluation
    final_cost = cost_function(params)
    function_evals += 1
    return params, final_cost, costs, times, (function_evals, gradient_evals)
```

```
# Define the two functions from Week 4
def function1(x):
    """Function 1: f(x,y) = 6*(x-1)^4 + 8*(y-2)^2"""
    return 6 * (x[0]-1)**4 + 8 * (x[1]-2)**2
def function1_gradient(x):
    """Gradient of function1"""
    dx = 24 * (x[0]-1)**3
    dy = 16 * (x[1]-2)
    return np.array([dx, dy])
def function2(x):
    """Function 2: f(x,y) = Max(x-1,0) + 8*|y-2|"""
    return max(x[0]-1, 0) + 8 * abs(x[1]-2)

def function2_gradient(x):
    """Gradient of function2 (approximation for non-differentiable points)"""
    dx = 1 if x[0] > 1 else 0
    dy = 8 if x[1] > 2 else -8 if x[1] < 2 else 0
    return np.array([dx, dy])
# Helper function to visualize the cost landscape
def plot_cost_landscape(cost_function, bounds, resolution=100):
    """
    Plots the cost landscape of a 2D function.
    Parameters:
    cost_function (callable): The function to plot
    bounds (list): List of tuples (min, max) for each parameter
    resolution (int): Number of points along each dimension
    Returns:
    fig (matplotlib.figure.Figure): The figure object
    X, Y (numpy.ndarray): Coordinate meshgrid
    Z (numpy.ndarray): Function values
    """
    x_min, x_max = bounds[0]
    y_min, y_max = bounds[1]

    X, Y = np.meshgrid(np.linspace(x_min, x_max, resolution),
                       np.linspace(y_min, y_max, resolution))
    Z = np.zeros_like(X)

    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = cost_function(np.array([X[i, j], Y[i, j]]))

    fig = plt.figure(figsize=(12, 5))

    # 3D surface plot
    ax1 = fig.add_subplot(121, projection='3d')
    surf = ax1.plot_surface(X, Y, Z, cmap=cm.viridis, linewidth=0, antialiased=True, alpha=0.8)
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')
    ax1.set_zlabel('f(X, Y)')
    ax1.set_title('Cost Function Surface')

    # 2D contour plot
    ax2 = fig.add_subplot(122)
    contour = ax2.contourf(X, Y, Z, 20, cmap=cm.viridis)
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_title('Cost Function Contour')
    fig.colorbar(contour, ax=ax2)

    plt.tight_layout()

    return fig, X, Y, Z

# Function to visualize the search paths
def plot_search_paths(cost_function, bounds, methods_results, resolution=100):
    """
    Plots the search paths of different optimization methods.

    Parameters:
    cost_function (callable): The function being optimized
    bounds (list): List of tuples (min, max) for each parameter
    methods_results (dict): Dictionary of results for each method
    resolution (int): Number of points along each dimension
    Returns:
    fig (matplotlib.figure.Figure): The figure object
    """
    fig, X, Y, Z = plot_cost_landscape(cost_function, bounds, resolution)
    ax = fig.axes[1]  # Get the contour plot axis
    # Plot the search paths
    for method_name, results in methods_results.items():
        if method_name == 'Gradient Descent':
            # For gradient descent, we have the full path
            path = results['path']
            ax.plot(path[:, 0], path[:, 1], 'o-', label=method_name, markersize=4)
        else:
            # For random search methods, we plot the best points found
            path = results['path']
            ax.scatter(path[:, 0], path[:, 1], label=method_name, s=10)
    ax.legend()
    plt.tight_layout()
    return fig
# Function to compare optimization methods
def compare_methods(cost_function, gradient_function, bounds, n_params, n_runs=10):
    """
    Compares different optimization methods on a given cost function.
    Parameters:
    cost_function (callable): The function to minimize
    gradient_function (callable): The gradient of the cost function
    bounds (list): List of tuples (min, max) for each parameter
    n_params (int): Number of parameters
    n_runs (int): Number of runs to average over
    Returns:
```

```
    results (dict): Dictionary of results for each method
    """
    methods = {
        'Global Random Search': {
            'params': {
                'n_samples': 1000,
                'verbose': False
            },
            'costs': [],
            'times': [],
            'evals': []
        },
        'Modified Random Search': {
            'params': {
                'n_samples': 100,
                'm_best': 5,
                'n_iterations': 10,
                'neighborhood_size': 0.1,
                'verbose': False
            },
            'costs': [],
            'times': [],
            'evals': []
        },
        'Gradient Descent': {
            'params': {
                'learning_rate': 0.001,
                'n_iterations': 1000,
                'verbose': False
            },
            'costs': [],
            'times': [],
            'evals': []
        }
    }

    for method_name, method_info in methods.items():
        print(f"Running {method_name}...")
        for run in range(n_runs):
            if method_name == 'Global Random Search':
                best_params, best_cost, costs, times, n_evals = global_random_search(
                    cost_function,
                    n_params,
                    bounds,
                    **method_info['params']
                )
            elif method_name == 'Modified Random Search':
                best_params, best_cost, costs, times, n_evals = modified_random_search(
                    cost_function,
                    n_params,
                    bounds,
                    **method_info['params']
                )
            elif method_name == 'Gradient Descent':
                # For gradient descent, we use an initial point in the middle of the bounds
                initial_params = np.zeros(n_params)
                for j in range(n_params):
                    lower, upper = bounds[j]
                    initial_params[j] = (lower + upper) / 2
                best_params, best_cost, costs, times, n_evals = gradient_descent(
                    cost_function,
                    gradient_function,
                    initial_params,
                    **method_info['params']
                )
            # Store results
            method_info['costs'].append(costs)
            method_info['times'].append(times)
            method_info['evals'].append(n_evals)

            print(f"  Run {run+1}/{n_runs}, Best cost: {best_cost:.6f}")
    # Process results
    results = {}
    for method_name, method_info in methods.items():
        costs_array = np.array(method_info['costs'])
        times_array = np.array(method_info['times'])
        # Compute average and standard deviation
        avg_costs = np.mean(costs_array, axis=0)
        std_costs = np.std(costs_array, axis=0)
        avg_times = np.mean(times_array, axis=0)
        # For gradient descent, we need to compute the parameter path
        if method_name == 'Gradient Descent':
            # Use the last run for visualization
            path = np.zeros((method_info['params']['n_iterations'] + 1, n_params))
            initial_params = np.zeros(n_params)
            for j in range(n_params):
                lower, upper = bounds[j]
                initial_params[j] = (lower + upper) / 2
            path[0] = initial_params
            params = initial_params.copy()
            for i in range(method_info['params']['n_iterations']):
                grad = gradient_function(params)
                params = params - method_info['params']['learning_rate'] * grad
                path[i+1] = params
        else:
            # For random search methods, we use the best points found
            if method_name == 'Global Random Search':
                path = np.zeros((method_info['params']['n_samples'], n_params))
            else:
                path = np.zeros((method_info['params']['n_iterations'] + 1, n_params))
            # Generate random path for visualization
            best_cost = float('inf')
            if method_name == 'Global Random Search':
```

```
                for i in range(method_info['params']['n_samples']):
                    params = np.zeros(n_params)
                    for j in range(n_params):
                        lower, upper = bounds[j]
                        params[j] = np.random.uniform(lower, upper)
                    cost = cost_function(params)
                    if cost < best_cost:
                        best_cost = cost
                        path[i] = params
            else:
                # Initialize population with random points
                population = []
                population_costs = []
                for _ in range(method_info['params']['n_samples']):
                    params = np.zeros(n_params)
                    for j in range(n_params):
                        lower, upper = bounds[j]
                        params[j] = np.random.uniform(lower, upper)
                    cost = cost_function(params)
                    population.append(params)
                    population_costs.append(cost)
                # Sort population by cost and keep M best
                sorted_indices = np.argsort(population_costs)
                population = [population[i] for i in sorted_indices[:method_info['params']['m_best']]]
                population_costs = [population_costs[i] for i in sorted_indices[:method_info['params']['m_best']]]

                path[0] = population[0]
                for iteration in range(method_info['params']['n_iterations']):
                    new_population = []
                    new_population_costs = []

                    for params in population:
                        for _ in range(method_info['params']['n_samples'] // method_info['params']['m_best']):
                            new_params = params.copy()
                            for j in range(n_params):
                                lower, upper = bounds[j]
                                range_size = upper - lower
                                perturbation = np.random.uniform(-method_info['params']['neighborhood_size'] * range_size,
                                                                 method_info['params']['neighborhood_size'] * range_size)
                                new_params[j] += perturbation
                                new_params[j] = max(lower, min(upper, new_params[j]))

                            cost = cost_function(new_params)
                            new_population.append(new_params)
                            new_population_costs.append(cost)
                    new_population.extend(population)
                    new_population_costs.extend(population_costs)
                    sorted_indices = np.argsort(new_population_costs)
                    population = [new_population[i] for i in sorted_indices[:method_info['params']['m_best']]]
                    population_costs = [new_population_costs[i] for i in sorted_indices[:method_info['params']['m_best']]]

                    path[iteration+1] = population[0]
        results[method_name] = {
            'avg_costs': avg_costs,
            'std_costs': std_costs,
            'avg_times': avg_times,
            'path': path
        }
    return results
```

```
#main_experiment.py
import numpy as np
import matplotlib.pyplot as plt
import time
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
# Import functions from random_search_algorithms.py
from random_search_algorithms import (
    global_random_search, modified_random_search, gradient_descent,
    function1, function1_gradient, function2, function2_gradient,
    plot_cost_landscape, plot_search_paths, compare_methods
)
def run_test_functions_experiments():
    """
    Run experiments on test functions and compare optimization methods.
    """
    print("Running experiments on test functions...")
    # Define parameter bounds for test functions
    # Function 1: f(x,y) = 6*(x-1)^4 + 8*(y-2)^2 has minimum at (1,2)
    function1_bounds = [(-1, 3), (0, 4)]
    # Function 2: f(x,y) = Max(x-1,0) + 8*|y-2| has minimum at (x≤1, y=2)
    function2_bounds = [(-1, 3), (0, 4)
    # Plot cost landscapes
    print("Plotting cost landscapes...")
    fig_func1, X_func1, Y_func1, Z_func1 = plot_cost_landscape(function1, function1_bounds)
    plt.savefig('function1_landscape.png')
    plt.close(fig_func1)
    fig_func2, X_func2, Y_func2, Z_func2 = plot_cost_landscape(function2, function2_bounds)
    plt.savefig('function2_landscape.png')
    plt.close(fig_func2)
    # Compare methods on Function 1
    print("\nComparing methods on Function 1: f(x,y) = 6*(x-1)^4 + 8*(y-2)^2")
    function1_results = compare_methods(function1, function1_gradient, function1_bounds, 2, n_runs=5)
    # Plot cost vs iterations
    plt.figure(figsize=(10, 6))
    for method_name, result in function1_results.items():
        plt.plot(range(len(result['avg_costs'])), result['avg_costs'], label=method_name)
        plt.fill_between(
            range(len(result['avg_costs'])),
            result['avg_costs'] - result['std_costs'],
            result['avg_costs'] + result['std_costs'],
```

```
            alpha=0.2
        )
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.title('Cost vs Iterations - Function 1: f(x,y) = 6*(x-1)^4 + 8*(y-2)^2')
    plt.legend()
    plt.grid(True)
    plt.yscale('log')
    plt.savefig('function1_cost_vs_iterations.png')
    plt.close()
    # Plot cost vs time
    plt.figure(figsize=(10, 6))
    for method_name, result in function1_results.items():
        plt.plot(result['avg_times'], result['avg_costs'], label=method_name)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Cost')
    plt.title('Cost vs Time - Function 1: f(x,y) = 6*(x-1)^4 + 8*(y-2)^2')
    plt.legend()
    plt.grid(True)
    plt.yscale('log')
    plt.savefig('function1_cost_vs_time.png')
    plt.close()
    # Plot search paths
    fig_func1_paths = plot_search_paths(function1, function1_bounds, function1_results)
    plt.savefig('function1_search_paths.png')
    plt.close(fig_func1_paths)
    # Compare methods on Function 2
    print("\nComparing methods on Function 2: f(x,y) = Max(x-1,0) + 8*|y-2|")
    function2_results = compare_methods(function2, function2_gradient, function2_bounds, 2, n_runs=5)
    # Plot cost vs iterations
    plt.figure(figsize=(10, 6))
    for method_name, result in function2_results.items():
        plt.plot(range(len(result['avg_costs'])), result['avg_costs'], label=method_name)
        plt.fill_between(
            range(len(result['avg_costs'])),
            result['avg_costs'] - result['std_costs'],
            result['avg_costs'] + result['std_costs'],
            alpha=0.2
        )
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.title('Cost vs Iterations - Function 2: f(x,y) = Max(x-1,0) + 8*|y-2|')
    plt.legend()
    plt.grid(True)
    plt.yscale('log')
    plt.savefig('function2_cost_vs_iterations.png')
    plt.close()
    # Plot cost vs time
    plt.figure(figsize=(10, 6))
    for method_name, result in function2_results.items():
        plt.plot(result['avg_times'], result['avg_costs'], label=method_name)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Cost')
    plt.title('Cost vs Time - Function 2: f(x,y) = Max(x-1,0) + 8*|y-2|')
    plt.legend()
    plt.grid(True)
    plt.yscale('log')
    plt.savefig('function2_cost_vs_time.png')
    plt.close()
    # Plot search paths
    fig_func2_paths = plot_search_paths(function2, function2_bounds, function2_results)
    plt.savefig('function2_search_paths.png')
    plt.close(fig_func2_paths)
    return function1_results, function2_results
if __name__ == "__main__":
    run_test_functions_experiments()
```

```
#cnn_hyperparameter_tuning.py
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
import time
import sys
# Load the original CIFAR10 code as a base
def load_and_preprocess_cifar10(n_train=5000):
    # Model / data parameters
    num_classes = 10
    input_shape = (32, 32, 3)
    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
    x_train = x_train[1:n_train]; y_train=y_train[1:n_train]
    # Scale images to the [0, 1] range
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    print("Original x_train shape:", x_train.shape)
    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)
    return x_train, y_train, x_test, y_test, num_classes, input_shape
def build_cifar10_model(input_shape, num_classes):
    # Using the same model architecture as in the downloaded code
```

```
    model = keras.Sequential()
    model.add(Conv2D(16, (3,3), padding='same', input_shape=input_shape, activation='relu'))
    model.add(Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
    model.add(Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(num_classes, activation='softmax',kernel_regularizer=regularizers.l1(0.0001)))
    return model
def train_and_evaluate_model(hyperparams, x_train, y_train, x_test, y_test, num_classes, input_shape, verbose=0):
    """
    Train and evaluate the CNN with the given hyperparameters.
    Parameters:
    hyperparams (list): [batch_size, learning_rate, beta1, beta2, epochs]
    x_train, y_train: Training data
    x_test, y_test: Test data
    num_classes: Number of output classes
    input_shape: Input shape of the model
    verbose: Verbosity level (0: silent, 1: progress bar, 2: one line per epoch)
    Returns:
    float: The cost (negative validation accuracy)
    """
    # Extract hyperparameters
    batch_size = int(hyperparams[0])
    learning_rate = hyperparams[1]
    beta1 = hyperparams[2]
    beta2 = hyperparams[3]
    epochs = int(hyperparams[4])
    # Set random seed for reproducibility
    tf.random.set_seed(42)
    np.random.seed(42)
    # Build model
    model = build_cifar10_model(input_shape, num_classes)
    # Compile model with Adam optimizer
    optimizer = keras.optimizers.Adam(learning_rate=learning_rate, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
    # Train model with early stopping
    early_stopping = keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=3,
        restore_best_weights=True
    )
    # Split training data into training and validation sets
    validation_split = 0.1
    val_samples = int(len(x_train) * validation_split)
    x_val = x_train[-val_samples:]
    y_val = y_train[-val_samples:]
    x_train_subset = x_train[:-val_samples]
    y_train_subset = y_train[:-val_samples]
    # Train model
    history = model.fit(
        x_train_subset,
        y_train_subset,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(x_val, y_val),
        verbose=verbose,
        callbacks=[early_stopping]
    )
    # Calculate validation accuracy and use negative accuracy as cost function (for minimization)
    val_loss, val_acc = model.evaluate(x_val, y_val, verbose=0)
    # Calculate cross-entropy loss on test data
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
    # Use negative test accuracy as the cost function (we want to maximize accuracy)
    cost = -test_acc
    return cost
# Apply global random search to CNN hyperparameter tuning
def cnn_global_random_search(x_train, y_train, x_test, y_test, num_classes, input_shape, param_bounds, n_samples, verbose=False):
    """
    Apply global random search for CNN hyperparameter tuning.
    Parameters:
    x_train, y_train: Training data
    x_test, y_test: Test data
    num_classes: Number of output classes
    input_shape: Input shape
    param_bounds (list): List of tuples (min, max) for each hyperparameter
    n_samples (int): Number of samples to evaluate
    verbose (bool): Whether to print progress
    Returns:
    best_params (numpy.ndarray): Best hyperparameters found
    best_cost (float): Best cost value (negative test accuracy)
    costs (list): Cost values at each iteration
    times (list): Cumulative time at each iteration
    """
    n_params = len(param_bounds)
    best_params = None
    best_cost = float('inf')
    costs = []
    times = []
```

```
    start_time = time.time()
    # Define wrapper function for cost evaluation
    def evaluate_params(params):
        return train_and_evaluate_model(params, x_train, y_train, x_test, y_test, num_classes, input_shape, verbose=0)
    for i in range(n_samples):
        # Generate random parameters
        params = np.zeros(n_params)
        for j in range(n_params):
            lower, upper = param_bounds[j]
            params[j] = np.random.uniform(lower, upper)
        # Evaluate cost function
        cost = evaluate_params(params)
        # Update best parameters if cost is lower
        if cost < best_cost:
            best_cost = cost
            best_params = params.copy()
        # Record cost and time
        costs.append(best_cost)
        times.append(time.time() - start_time)
        if verbose:
            print(f"Iteration {i+1}/{n_samples}, Params: {params}, Cost: {cost:.6f}, Best Cost: {best_cost:.6f}")
    return best_params, best_cost, costs, times
def cnn_modified_random_search(x_train, y_train, x_test, y_test, num_classes, input_shape, param_bounds, n_samples, m_best, n_iterations,
neighborhood_size=0.1, verbose=False):
    """
    Apply modified random search for CNN hyperparameter tuning.
    Parameters:
    x_train, y_train: Training data
    x_test, y_test: Test data
    num_classes: Number of output classes
    input_shape: Input shape
    param_bounds (list): List of tuples (min, max) for each hyperparameter
    n_samples (int): Number of samples per iteration
    m_best (int): Number of best points to keep
    n_iterations (int): Number of iterations
    neighborhood_size (float): Size of neighborhood for sampling
    verbose (bool): Whether to print progress
    Returns:
    best_params (numpy.ndarray): Best hyperparameters found
    best_cost (float): Best cost value (negative test accuracy)
    costs (list): Cost values at each iteration
    times (list): Cumulative time at each iteration
    """
    n_params = len(param_bounds)
    population = []
    population_costs = []
    costs = []
    times = []
    start_time = time.time()
    # Define wrapper function for cost evaluation
    def evaluate_params(params):
        return train_and_evaluate_model(params, x_train, y_train, x_test, y_test, num_classes, input_shape, verbose=0)
    # Initial random sampling
    print("Generating initial population...")
    for i in range(n_samples):
        params = np.zeros(n_params)
        for j in range(n_params):
            lower, upper = param_bounds[j]
            params[j] = np.random.uniform(lower, upper)
        cost = evaluate_params(params)
        population.append(params)
        population_costs.append(cost)
        if verbose:
            print(f"  Sample {i+1}/{n_samples}, Params: {params}, Cost: {cost:.6f}")
    # Sort population by cost and keep M best
    sorted_indices = np.argsort(population_costs)
    population = [population[i] for i in sorted_indices[:m_best]]
    population_costs = [population_costs[i] for i in sorted_indices[:m_best]]
    best_params = population[0].copy()
    best_cost = population_costs[0]
    costs.append(best_cost)
    times.append(time.time() - start_time)
    # Main iteration loop
    for iteration in range(n_iterations):
        print(f"Iteration {iteration+1}/{n_iterations}")
        new_population = []
        new_population_costs = []
        # Generate neighborhood samples around each point in the population
        for i, params in enumerate(population):
            for j in range(n_samples // m_best):
                # Generate random perturbation
                new_params = params.copy()
                for k in range(n_params):
                    lower, upper = param_bounds[k]
                    range_size = upper - lower
                    perturbation = np.random.uniform(-neighborhood_size * range_size,
                                                     neighborhood_size * range_size)
                    new_params[k] += perturbation
                    # Clip to bounds
```

```
                    new_params[k] = max(lower, min(upper, new_params[k]))
                # Evaluate cost function
                cost = evaluate_params(new_params)
                new_population.append(new_params)
                new_population_costs.append(cost)
                if verbose:
                    print(f"  Sample {i*n_samples//m_best+j+1}/{n_samples}, Params: {new_params}, Cost: {cost:.6f}")
        # Add current population to new samples
        new_population.extend(population)
        new_population_costs.extend(population_costs)
        # Sort combined population by cost and keep M best
        sorted_indices = np.argsort(new_population_costs)
        population = [new_population[i] for i in sorted_indices[:m_best]]
        population_costs = [new_population_costs[i] for i in sorted_indices[:m_best]]
        # Update best parameters if needed
        if population_costs[0] < best_cost:
            best_cost = population_costs[0]
            best_params = population[0].copy()
                costs.append(best_cost)
        times.append(time.time() - start_time)
                print(f"  Best cost so far: {best_cost:.6f}, Best params: {best_params}")
    return best_params, best_cost, costs, times
# Main function to run the hyperparameter tuning
def main():
    # Load and preprocess CIFAR10 data
    print("Loading and preprocessing CIFAR10 data...")
    x_train, y_train, x_test, y_test, num_classes, input_shape = load_and_preprocess_cifar10()
    # Define hyperparameter bounds
    # [batch_size, learning_rate, beta1, beta2, epochs]
    param_bounds = [
        (16, 256),       # batch_size
        (0.0001, 0.01),  # learning_rate
        (0.8, 0.99),     # beta1
        (0.99, 0.9999),  # beta2
        (5, 30)          # epochs
    ]
    # Apply global random search
    print("\nApplying Global Random Search...")
    grs_best_params, grs_best_cost, grs_costs, grs_times = cnn_global_random_search(
        x_train, y_train, x_test, y_test, num_classes, input_shape,
        param_bounds,
        n_samples=10,  # Reduced for demonstration
        verbose=True
    )
    print("\nGlobal Random Search Results:")
    print(f"Best Parameters: {grs_best_params}")
    print(f"Best Cost (negative accuracy): {grs_best_cost:.6f}")
    print(f"Best Accuracy: {-grs_best_cost:.6f}")
    # Apply modified random search
    print("\nApplying Modified Random Search...")
    mrs_best_params, mrs_best_cost, mrs_costs, mrs_times = cnn_modified_random_search(
        x_train, y_train, x_test, y_test, num_classes, input_shape,
        param_bounds,
        n_samples=5,     # Reduced for demonstration
        m_best=2,
        n_iterations=2,  # Reduced for demonstration
        neighborhood_size=0.1,
        verbose=True
    )
    print("\nModified Random Search Results:")
    print(f"Best Parameters: {mrs_best_params}")
    print(f"Best Cost (negative accuracy): {mrs_best_cost:.6f}")
    print(f"Best Accuracy: {-mrs_best_cost:.6f}")
    # Convert hyperparameters to more readable format
    def format_hyperparams(params):
        return {
            'batch_size': int(params[0]),
            'learning_rate': params[1],
            'beta1': params[2],
            'beta2': params[3],
            'epochs': int(params[4])
        }
    print("\nGlobal Random Search Best Hyperparameters:")
    print(format_hyperparams(grs_best_params))
    print("\nModified Random Search Best Hyperparameters:")
    print(format_hyperparams(mrs_best_params))
    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(grs_times, grs_costs, label='Global Random Search')
    plt.plot(mrs_times, mrs_costs, label='Modified Random Search')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Best Cost (negative accuracy)')
    plt.title('Hyperparameter Tuning Performance')
    plt.legend()
    plt.grid(True)
    plt.savefig('hyperparameter_tuning_results.png')
    plt.show()
if __name__ == "__main__":
    main()
```