Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 8 Assignment
CS7DS2 Optimisation for Machine Learning

Ujjayant Kadian (22330954)

April 4, 2025

Introduction

The following report details an exploration of optimization strategies implemented in Python, using both a simple global random search approach and a population-based variant to solve different cost minimization problems. Special focus is placed on two key applications: (i) optimizing two functions originally encountered in the week-4 assignment, and (ii) tuning hyperparameters for a convolutional neural network (CNN) trained on the CIFAR-10 dataset. The motivation is to investigate how gradient-based methods compare with purely random or population-oriented search, and how these methods handle smooth and non-smooth cost landscapes.

A baseline CNN model on CIFAR-10 is first introduced, along with two additional cost-function metrics (cross-entropy and AUC) for a more comprehensive assessment. The first algorithm, Global Random Search, randomly samples parameter vectors from bounded intervals and retains the minimum-cost solution encountered. Its performance is then compared against Gradient Descent when optimizing two simple functions with known optima. Afterward, a Population-Based Random Search (PRS) extension refines the sampling process by retaining and perturbing top candidates each generation. Finally, both of these random search approaches are employed to tune the hyperparameters of the CNN, and the outcomes are discussed with respect to search efficiency, final accuracy, and the observed limitations when searching a large hyperparameter space.

CNN on CIFAR-10 with Additional Cost Function Evaluation

The first step in this study involved training a simple CNN on the CIFAR-10 dataset and evaluating performance using test accuracy, cross-entropy loss, and the area under the ROC curve (AUC). The CNN's architecture comprised several convolution layers followed by dropout and a final dense layer for classification. The dataset was loaded and divided into training and testing subsets. A moderate subset of the training set (5,000 images) was used in order to maintain computational feasibility.

Architecture and Training Procedure

- **Convolutional Layers:** Each convolutional layer (with or without stride and kernel changes) was followed by an activation function (ReLU) and occasional pooling steps.
- **Dropout:** A dropout layer mitigated overfitting by randomly disabling certain neurons during training.
- **Output Layer:** A fully connected layer with softmax activation mapped feature vectors to the 10 class probabilities.

Metrics

- **Cross-Entropy Loss:**
  - Measured the divergence between the predicted and true distribution.
  - Computed via `tf.keras.losses.CategoricalCrossentropy()`, yielding an average score for the test set.

- **AUC (Area Under the ROC Curve):**
  - Calculated using `tf.keras.metrics.AUC()`.
  - Reflects how well the model ranked positive examples above negative ones across all possible thresholds.

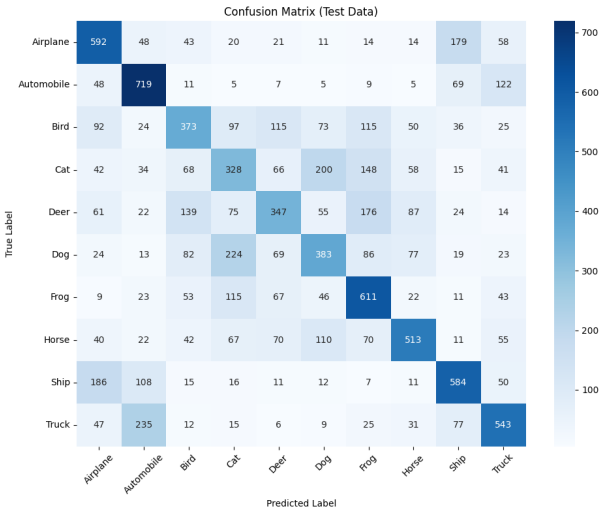


Figure 1: Confusion Matrix for Test Data.

Key Observations

- Test Accuracy hovered around 50%, indicating moderate success, which is reasonable for a relatively small training subset and limited innovation in the model architecture.
- Cross-Entropy Loss (about 1.3943) signaled a moderate level of predictive confidence.
- Test AUC (0.8885) showed that the model ranked correct classes reasonably well, even if class prediction was sometimes incorrect.
- Confusion Matrix analysis revealed high confusion among visually similar classes (Bird, Cat, Dog). Meanwhile, classes such as Automobile and Frog were identified with higher clarity.

(a) (i) Global Random Search Algorithm

The Global Random Search (GRS) algorithm is a straightforward yet effective approach to explore high-dimensional parameter spaces. It is particularly helpful when gradient-based methods are infeasible, or the cost function is discontinuous, noisy, or non-differentiable.

Implementation Strategy

- **Inputs:** The algorithm takes the following inputs:
  - (1) A cost function f
  - (2) The number of parameters n
  - (3) The bounds (l_i, u_i) for each parameter
  - (4) The number of samples N
- **Sampling:** In each iteration (from 1 to N), a parameter vector is generated by uniformly sampling each parameter from its specified bounds $[l_i, u_i]$.
- **Cost Evaluation:** The algorithm evaluates the cost function at the sampled parameter vector. If the new cost is lower than the previously recorded best cost, the best solution is updated accordingly.
- **Tracking:** The algorithm logs each candidate parameter vector, its corresponding cost, and the timestamp of the iteration for subsequent analysis (e.g., generating plots of cost versus iteration and cost versus time).

Advantages:

- Straightforward to implement.
- Can handle complex, non-smooth cost surfaces.

Limitations:

- Potentially requires a large number of function evaluations to locate good solutions.
- No mechanism to exploit promising regions (it is purely random).

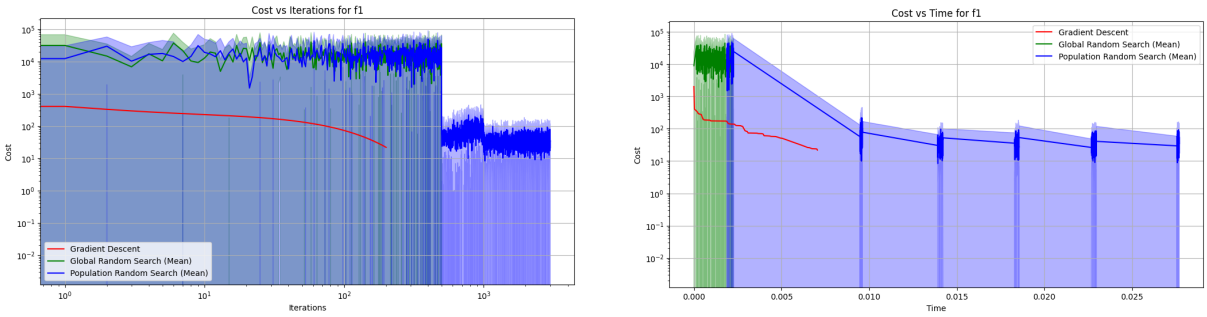
Note for upcoming sections:

The graphs used in sections (a)(ii) and (b)(ii) are shown below for ease of reference. The functions f_1 and f_2 will be defined and analyzed in detail in the following sections.

Cost vs Iterations and Cost vs Time Plot Generation:

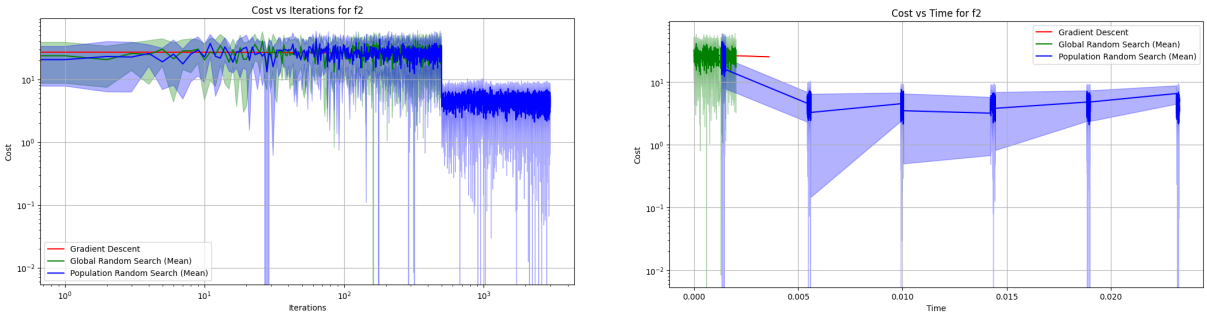
The plots were generated using matplotlib with a custom comparison function that:

- Takes optimization histories from Gradient Descent (GD), Global Random Search (GRS), and Population Random Search (PRS) runs as input
- Computes statistics (mean and standard deviation) across multiple runs for Gradient Descent, Global Random Search, and Population Random Search
- Plots Gradient Descent trajectory (red line) directly and random search results with error bands (lines with shaded error bands, particularly blue/green lines are the average cost of the runs and the shaded area is the standard deviation of the runs)
- Uses logarithmic scaling for both cost and iterations axes (except time)



(a) Cost vs. Iterations for f_1 (b) Cost vs. Time for f_1

Figure 2: Optimization progress for function f_1 showing convergence behavior



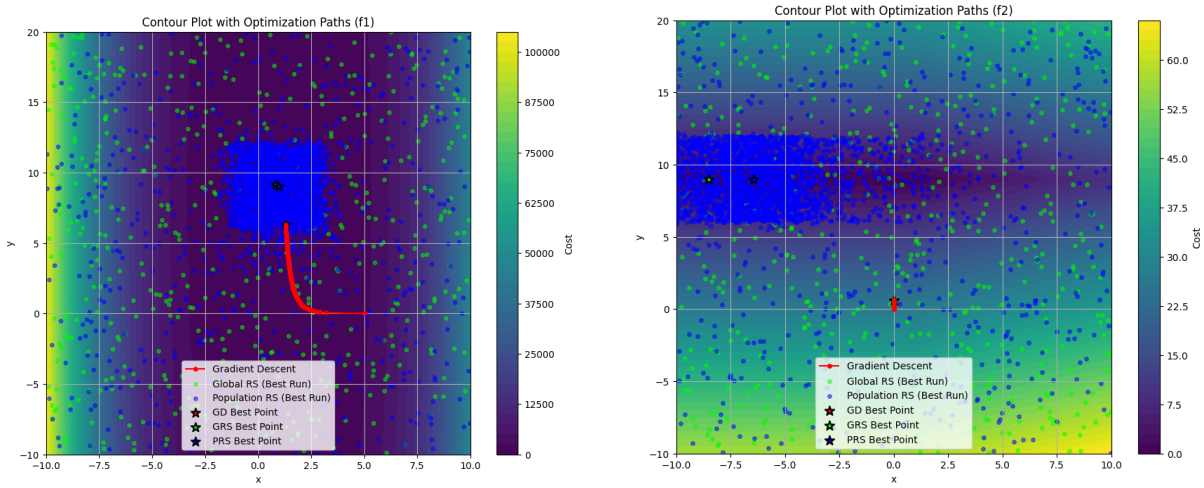
(a) Cost vs. Iterations for f_2 (b) Cost vs. Time for f_2

Figure 3: Optimization progress for function f_2 showing convergence behavior

Contour Plot Generation:

The contour plots were generated using matplotlib with a custom plotting function that:

- Creates a fine mesh grid (400x400 points) over the search space bounds
- Evaluates the cost function at each grid point to create the contour surface
- Plots the optimization paths:
  - Gradient Descent path shown as a red line with markers
  - Best Global Random Search run shown as lime scattered points
  - Best Population Random Search run shown as blue scattered points
- Highlights the best points found by each method using star markers
- Uses a viridis colormap with 50 contour levels to visualize the cost function landscape
- Includes a colorbar showing the cost scale for the random search scatter points



(a) Contour Plot with Optimization Paths for f_1

(b) Contour Plot with Optimization Paths for f_2

Figure 4: Contour plots showing the optimization paths taken by different algorithms

Performance Metrics Summary

Tables 1 and 2 show the comparative performance metrics for each optimization method on functions f_1 and f_2 respectively. These tables will be also referred to in the upcoming sections. The function and gradient evaluation counts were tracked using Python decorators that increment global counters each time the respective functions are called. For gradient descent, both the cost function and its gradient were counted separately, while for random search methods only the cost function evaluations were counted since no gradients were used.

The following values were generated using the histories gathered from multiple runs of each algorithm.

Table 1: Performance Summary for f_1

Method	Best Cost (mean±std)	Function Evals	Gradient Evals	Time (s)
Gradient Descent	21.938390	201	200	0.007
Global Random Search	1.289±1.465	500	N/A	0.002
Population Random Search	0.015±0.013	3000	N/A	0.026

Table 2: Performance Summary for f_2

Method	Best Cost (mean±std)	Function Evals	Gradient Evals	Time (s)
Gradient Descent	25.200000	201	200	0.004
Global Random Search	0.239±0.236	500	N/A	0.002
Population Random Search	0.021±0.028	3000	N/A	0.025

(a)(ii) Random Search on Two Functions

Functions, Theoretical Optima, and Gradient Descent Implementation

- $f_1(x) = 7(x-1)^4 + 3(y-9)^2$ *Optimum:* $(1, 9)$ with a cost of 0.
- $f_2(x) = \max(x-1, 0) + 3\,|y-9|$ *Optimum:* Achieved for $x \leq 1$ and $y = 9$ (cost of 0).

Similar to the week-4 assignment, the functions f_1 and f_2 were implemented as cost functions taking NumPy arrays $x = [x, y]$ as input. For f_1, both the function and its analytical gradient were defined, with the gradient being $[28(x-1)^3, 6(y-9)]$. For f_2, the function and its subgradient were implemented, with special care taken at non-differentiable points. The subgradient for the $\max(x-1, 0)$ term was set to 1 when $x > 1$, 0 when $x < 1$, and 0.5 at $x = 1$. For the $3|y-9|$ term, the subgradient was ± 3 based on whether $y > 9$ or $y < 9$.

Algorithm for Multiple Runs

The Global Random Search (GRS) algorithm is executed multiple times ($num\_runs = 10$) to assess performance variability and gather statistical measures. For each run:

- Function evaluation counters are reset
- GRS samples N random points within specified bounds
- The best solution, cost, evaluation history, and timing data are recorded
- Function evaluation counts are tracked separately for f_1 and f_2

After all runs complete, the collected data includes histories from each run, best costs achieved, total function evaluations performed, and execution times - enabling statistical analysis of the algorithm's performance and resulting in the plots descibed earlier.

Performance Comparison with Gradient Descent (GD)

- GD uses analytical or subgradients to update parameters iteratively. It requires both function and gradient evaluations each iteration, making it efficient on smooth functions such as f_1, but potentially less robust for non-smooth functions like f_2.
- **Function Evaluations:**
  - GD makes one function and one gradient evaluation per iteration.
  - GRS makes one function evaluation per iteration but may require many more iterations (and multiple runs) to reliably find the optimum. (refer to tables 1 and 2)
- **Measurement Issues:** Timing results can vary due to system overhead and Python interpreter variability, meaning that cost versus time plots provide an approximate comparison.

Cost vs. Time and Cost vs. Iterations Plots

- **Cost vs. Iterations:**
  - **Gradient Descent (GD):** The plot (Figure 2a and 3a) shows a steep initial decline in cost as GD rapidly descends along the gradient. However, its progress may plateau if the gradient becomes very small or if it gets trapped in a local minimum.
  - **Global Random Search (GRS):** The points are more scattered, reflecting the random nature of the search. Although the average cost (mean over multiple runs - see the lime lines) is higher, some runs can achieve near-optimal values. The variability (error bands representing standard deviation) is notable.
- **Cost vs. Time:** (see Figure 2b and 3b)
  These plots reveal how quickly each method reduces the cost when considering elapsed time. GD tends to show rapid cost reduction early on, but the time per iteration includes gradient computation overhead.

Observations on Final Parameter Values and Evaluations

- **For f_1:**
  - GD may converge to a solution but with the current number of iterations, it was not be able to find the optimum, it found $[1.29, 6.30]$ with a relatively high cost (21.94) (Table 1).
  - GRS yielded parameters like $[0.82, 9.11]$ with a cost of 0.04 after 10 runs (Table 1).
- **For f_2:**
  - GD may start converging to suboptimal values or take a long time to converge but in the current number of iterations, it was not able to find the optimum, it found $[0.0, 0.6]$ and a high cost (25.20) due to its reliance on gradients in a non-smooth region.
  - GRS typically identify solutions with $x < 1$ and $y \approx 9$, leading to costs near 0.
- **Function/Gradient Evaluations:**
  - Although this random search method required many more function evaluations, it often has no gradient overhead.
  - In contrast, GD incurs both function and gradient evaluations per iteration, which may be efficient for smooth functions but becomes a limitation for non-smooth cases.

Key Insight on Averages vs. Optimality

Even though the average cost over multiple runs for GRS remains higher than the lowest cost obtained by a single GD run, the inherent randomness means that some runs find near-optimal solutions. This variability highlights the exploration capability of random search methods; while many runs yield suboptimal costs, a few fortunate runs achieve values very close to the theoretical optimum.

(b) (i) Population Based Random Search Algorithm

The Population-Based Random Search (PRS) algorithm addresses the pure randomness of GRS. This algorithm maintains a population of candidate solutions and iteratively refines this population by focusing on the top M candidates each generation.

Algorithm Outline

- **Initialization:** Generate N random candidates uniformly within parameter bounds.
- **Selection:** Evaluate cost and select the M best performers (lowest cost).
- **Neighborhood Sampling:** Around each of these M best solutions, create new candidate solutions by adding small random perturbations, keeping solutions within bounds.
- **Repetition:** Over multiple generations, track the best solutions.
- **Output:** The best overall solution from all generations is returned. (useful parameters are returned as done in GRS - for plotting purposes)

Advantages:

- Balances exploration and exploitation by focusing on neighborhoods of top candidates.
- Can converge more rapidly than pure GRS once promising regions are found.

Limitations:

- Increased computational cost per generation (more function evaluations).
- Might over-exploit certain regions if the hyperparameters for neighborhood size are not managed carefully.

Compared to GRS, theoretically, PRS should yield better overall minima, though at a higher computational cost.

(b) (ii) Population-Based Random Search and Contour Plot Analysis

Algorithm for Multiple Runs

To evaluate algorithm performance across multiple runs, we implemented a wrapper function that executes the PRS algorithm 10 times with consistent parameters. Similar to GRS implementation, the wrapper function resets function evaluation counters, tracks the best solution, cost, and optimization history for each run, and records total function evaluations and runtime.

Contour Plot Analysis and Description

- **Contour Plots:** (see Figure 4)
  - The plots visualize the cost function landscape with contours showing cost levels (as explained earlier).
  - PRS trajectories exhibit clustering of points that concentrate near optima over generations, demonstrating the algorithm's refinement capability.
  - For f_1, points progressively cluster around $(1, 9)$, while for f_2, they gather near the line $x \leq 1, y = 9$ (see the star points in the plots).
- **Convergence Behavior:**
  - Initial generations show wide dispersion similar to GRS, exploring the parameter space broadly.
  - Middle generations begin focusing around promising regions identified by top performers.
  - Final generations exhibit tight clustering near optimal points, showing effective exploitation.
- **Performance Characteristics:**
  - PRS achieves more consistent convergence compared to GRS, with final solutions typically closer to theoretical optima.
  - For f_1, best solutions reached costs below 0.01 (e.g., parameters $[0.999, 9.005]$).
  - For f_2, algorithm reliably found solutions with $x < 1$ and $y \approx 9$ (e.g., $[-0.06, 9.00]$), yielding near-zero costs.

Comparative Analysis with GD and GRS

- **Search Strategy:**
  - Unlike GD's gradient-following path or GRS's purely random sampling, PRS shows evolving search patterns.
  - Initial random exploration transitions to focused exploitation around promising regions.
- **Computational Efficiency:**
  - Higher function evaluation count per iteration compared to both GD and GRS, and that is expected as it is dependent on how many generations PRS runs for.
  - However, more reliable convergence to near-optimal solutions justifies the increased computational cost.
- **Solution Quality:**
  - More consistent in finding near-optimal solutions compared to GRS.
  - Particularly effective for f_2 where GD struggles with non-smoothness.
  - Final solutions typically achieve lower costs than both GD and GRS.

In summary, while the average cost across multiple runs of random search methods (GRS and PRS) may be higher than single GD run (as explained earlier), their stochastic nature enables them to occasionally locate solutions very close to the theoretical optima. The detailed contour plots illustrate the distinct paths taken by each algorithm: GD follows a smooth descent, whereas GRS exhibits dispersed sampling and PRS shows progressive concentration near the optimal region. These plots, along with the comparisons of function and gradient evaluations, underscore the trade-offs between efficiency and robustness inherent in these optimization approaches.

(c) Hyperparameter Tuning for the CNN via Random Search

Implementation Strategy

The hyperparameter optimization process utilized the above two stochastic search algorithms to tune key hyperparameters of the CNN. The hyperparameters under consideration were the mini-batch size, Adam optimizer parameters (learning rate (α), β_1, and β_2), and number of epochs.

Global Random Search (GRS)

- **Algorithm:** The GRS implementation follows the same approach as before:
  1. **Input:** A cost function (validation loss), hyperparameters, and their bounds
  2. **Sampling:** Uniform sampling of candidate hyperparameter vectors within bounds
  3. **Evaluation:** Train CNN with each candidate (by building the model corresponding to the candidate hyperparameters), logging validation loss and metrics
  4. **Selection:** Choose candidate with lowest validation loss
- **Logging:** Track hyperparameters, cost, time, and progress for analysis

Population-Based Random Search (PRS)

- **Algorithm:** The PRS implementation maintains its evolutionary approach:
  1. **Initialization:** Generate initial population of hyperparameter candidates
  2. **Evolution:** Sort by validation loss, select top performers, create new candidates via perturbation
  3. **Evaluation:** Train models with new candidates and update population
  4. **Selection:** Choose best candidate from final generation
- **Logging:** Track performance metrics across generations

Search Space Explored

The hyperparameter search space encompassed a broad range of configurations:

- **Batch Size:** 16 to 256
  Range covering very small mini-batches to large batches, affecting training variance and memory requirements.
- **Learning Rate:** 0.0001 to 0.01
  Range from low learning rates to moderately high rates, affecting convergence speed and optimization stability.
- β_1**:** 0.8 to 0.99 and β_2**:** 0.9 to 0.9999
  Bounds exploring different momentum dynamics in the Adam optimizer.
- **Epochs:** 5 to 30
  Range balancing between rapid evaluation and model convergence depth.

Performance Comparison

After sequentially running the GRS and PRS on the CNN model for 20 times each (with $M = 3$ (best candidates) and $pop\_iterations = 3$ (number of generations) for PRS), the extensive search space results in many hyperparameter combinations yielding similar performance levels to the baseline model. While optimal configurations may achieve very low validation loss, the overall average cost across multiple runs remains higher. A narrower search space could potentially yield more significant improvements, with the trade-off of potentially missing unexpected beneficial configurations.
The following convergence plot was generated by using the logged validation loss values for each run:

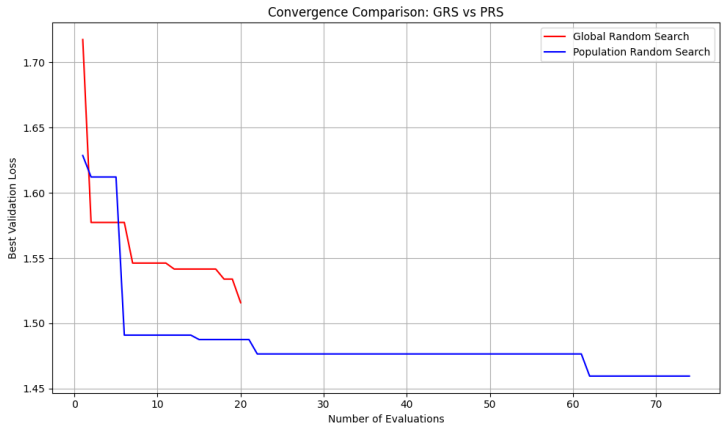


Figure 5: Convergence comparison between Global Random Search (GRS) and Population-based Random Search (PRS) during hyperparameter optimization. The plot shows validation loss over evaluations, with PRS demonstrating gradual improvement across generations compared to GRS's uniform sampling approach.

The logged metrics and the convergence plot reveal:

- **Best Validation Loss:**
  - *Global RS:* 1.515856
  - *Population RS:* 1.459570
  - *Difference (PRS - GRS):* –0.056286

  PRS achieved marginally lower loss through iterative refinement.

- **Total Evaluations and Time:**
  - *Global RS:* 20 evaluations; ∼235.50 seconds total
  - *Population RS:* 74 evaluations; ∼913.53 seconds total
  - *Difference:* PRS required 54 additional evaluations and approximately 678 seconds more.

- **Average Evaluation Time:**
  - Global RS: ∼11.78 seconds/evaluation
  - Population RS: ∼12.35 seconds/evaluation

  Similar evaluation times indicate extra PRS evaluations stem from multiple generations rather than slower individual evaluations.

- **Convergence Behavior:**
  - Similar losses between GRS and PRS at early checkpoints
  - PRS gradually discovers lower-loss configurations compared to GRS
  - Average loss remains higher for stochastic methods despite some near-optimal runs

Final Model Performance Comparison

The best performing model was selected to compare the performance with the original model. This helps to understand the impact of our current hyperparameter tuning process and where it lags behind.
Post-hyperparameter tuning results compared to the original model:

- **Original Model (Baseline):**
  - Test Accuracy: ∼50%
  - Test Loss: 1.3943
  - Test AUC: 0.8885
- **Final Model (Post-Random Search):**

- Test Accuracy: ∼49%
- Test Loss: ∼1.4904
- Test Accuracy (logged): ∼48.91%

The modest performance change can be attributed to the vast search space, where many configurations resulted in similar performance levels.

Conclusion

This report highlighted the implementation and comparison of three core optimization methods: Gradient Descent (GD), Global Random Search (GRS), and Population-Based Random Search (PRS) across both mathematical functions and CNN hyperparameter tuning tasks. Several insights emerged:

Global Random Search

- Effective when gradients are unavailable or unreliable.
- Requires many function evaluations to reliably locate near-optimal solutions.
- Demonstrates strong exploration, though with high variance in final outcomes.

Population-Based Random Search

- Builds upon GRS by maintaining a population of solutions.
- Converges more reliably than GRS once promising regions are discovered.
- Incurred higher computational demands in the experiments.

Gradient Descent

- Performs well on smooth, differentiable functions such as polynomial-like objectives.
- Struggles with non-smooth cost surfaces.
- Can be trapped in local minima if not carefully tuned for step size or momentum.

The experiments on tuning CNN hyperparameters showed that random search approaches can indeed find viable configurations, but marginal gains over a baseline are common when the dataset and architecture provide inherent performance limits (in addition to a large search space). Timing considerations and the potential for large hyperparameter spaces remain practical concerns in real-world scenarios.

Overall, each approach: GD, GRS, and PRS holds unique advantages and drawbacks. GD is fast and effective when the function is smooth and gradients are easy to compute, but it can fail on highly non-smooth terrains. GRS and PRS excel at broad exploration, particularly helpful for complex or unknown landscapes, but at the cost of numerous function evaluations. Population-based methods improve on pure random sampling by focusing on high-potential regions, which often leads to better outcomes if one is prepared to pay the additional computational cost.

These findings underscore the importance of tailoring the optimization strategy to the characteristics of both the cost function (e.g., smooth vs. non-smooth) and the problem domain (e.g., neural network hyperparameters). By combining a careful analysis of cost metrics, timing data, and the shape of the search space, it becomes easier to select or develop an optimization approach that balances efficiency and reliability for a given application.

A Code

```
import tensorflow as tf
gpus = tf.config.list_physical_devices('GPU')
if gpus:
  tf.config.set_visible_devices(gpus[0], 'GPU')
  tf.config.experimental.set_memory_growth(gpus[0], True)
print("GPU is available: ", tf.config.list_physical_devices('GPU'))

import numpy as np
```

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import time
from functools import wraps
import statistics

def plot_confusion_matrix(y_true, y_pred, title):
    # Create confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    # Create figure and axes
    plt.figure(figsize=(10, 8))

    # Create heatmap
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
                             'Dog', 'Frog', 'Horse', 'Ship', 'Truck'],
                yticklabels=['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
                             'Dog', 'Frog', 'Horse', 'Ship', 'Truck'])

    # Add labels and title
    plt.title(title)
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')

    # Rotate x-axis labels for better readability
    plt.xticks(rotation=45)

    plt.tight_layout()
    plt.show()

# For reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# Model / data parameters
num_classes = 10
input_shape = (32, 32, 3)

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
n = 5000  # use a subset for faster training
x_train = x_train[:n]
y_train = y_train[:n]

# Scale images to [0, 1]
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

print("Training data shape:", x_train.shape)

# Convert class vectors to one-hot encoded labels
y_train_cat = keras.utils.to_categorical(y_train, num_classes)
y_test_cat = keras.utils.to_categorical(y_test, num_classes)

# Build the CNN model
use_saved_model = False
if use_saved_model:
    model = keras.models.load_model("cifar.model")
else:
    model = keras.Sequential([
        layers.Conv2D(16, (3,3), padding='same', activation='relu',
            input_shape=input_shape),
        layers.Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'),
        layers.Conv2D(32, (3,3), padding='same', activation='relu'),
        layers.Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'),
        layers.Dropout(0.5),
        layers.Flatten(),
        layers.Dense(num_classes, activation='softmax',
            kernel_regularizer=regularizers.l1(0.0001))
    ])
```

```
        model.compile(loss="categorical_crossentropy", optimizer='adam',
            metrics=["accuracy"])
        model.summary()

        batch_size = 128
        epochs = 20
        history = model.fit(x_train, y_train_cat, batch_size=batch_size, epochs=epochs,
            validation_split=0.1)
        model.save("cifar.keras")

        # Plot training history
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        plt.plot(history.history['accuracy'], label='Train')
        plt.plot(history.history['val_accuracy'], label='Val')
        plt.title('Model Accuracy')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.legend()

        plt.subplot(1, 2, 2)
        plt.plot(history.history['loss'], label='Train')
        plt.plot(history.history['val_loss'], label='Val')
        plt.title('Model Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()

        plt.show()

    # Evaluate on training data
    train_preds = model.predict(x_train)
    y_train_pred = np.argmax(train_preds, axis=1)
    print("Classification Report (Train):")
    print(classification_report(np.argmax(y_train_cat, axis=1), y_train_pred))
    print("Confusion Matrix (Train):")
    # print(confusion_matrix(np.argmax(y_train_cat, axis=1), y_train_pred))
    plot_confusion_matrix(
        np.argmax(y_train_cat, axis=1),
        y_train_pred,
        'Confusion Matrix (Training Data)'
    )
    # Evaluate on test data
    test_preds = model.predict(x_test)
    y_test_pred = np.argmax(test_preds, axis=1)
    print("Classification Report (Test):")
    print(classification_report(np.argmax(y_test_cat, axis=1), y_test_pred))
    print("Confusion Matrix (Test):")
    # print(confusion_matrix(np.argmax(y_test_cat, axis=1), y_test_pred))
    plot_confusion_matrix(
        np.argmax(y_test_cat, axis=1),
        y_test_pred,
        'Confusion Matrix (Test Data)'
    )

    # Here we calculate the cross-entropy loss and AUC for the test set.
    loss_fn = tf.keras.losses.CategoricalCrossentropy()
    test_loss = loss_fn(y_test_cat, test_preds).numpy()

    auc_metric = tf.keras.metrics.AUC()
    auc_metric.update_state(y_test_cat, test_preds)
    test_auc = auc_metric.result().numpy()

    print(f"\nTest Cross-Entropy Loss: {test_loss:.4f}")
    print(f"Test AUC: {test_auc:.4f}")

    def global_random_search(cost_func, n, bounds, N):
        """
        Perform global random search.

        Returns:
          best_x: best parameter vector found
          best_cost: its cost
          history: list of (candidate, cost) for each sample
        """
        start_time = time.time()
```

```
    best_cost = np.inf
    best_x = None
    history = []

    for i in range(N):
        current_time = time.time() - start_time
        candidate = np.array([np.random.uniform(low, high) for (low, high) in
            bounds])
        cost = cost_func(candidate)
        history.append((i, cost, candidate.copy(), current_time))

        if cost < best_cost:
            best_cost = cost
            best_x = candidate

    return best_x, best_cost, history

def population_random_search(cost_func, n, bounds, N, M, iterations):
    """
    Perform a population-based random search.

    Parameters:
      cost_func: function mapping a parameter vector (np.array) to a cost (float)
      n: number of parameters
      bounds: list of (lower, upper) bounds for each parameter, length n
      N: number of samples per generation
      M: number of best candidates to keep
      iterations: number of generations

    Returns:
      best_x: best parameter vector found
      best_cost: its cost
      history: list of tuples (generation, best_M_points, best_M_costs)
    """
    start_time = time.time()
    # Initial random population
    population = [np.array([np.random.uniform(low, high) for (low, high) in bounds])
        for _ in range(N)]
    population_costs = [cost_func(x) for x in population]
    history = []
    iteration_count = 0

    # Record initial population
    for i, (point, cost) in enumerate(zip(population, population_costs)):
        current_time = time.time() - start_time
        history.append((iteration_count, cost, point.copy(), current_time))
        iteration_count += 1

    for it in range(iterations):
        # Sort population based on cost (ascending order)
        sorted_idx = np.argsort(population_costs)
        population = [population[i] for i in sorted_idx]
        population_costs = [population_costs[i] for i in sorted_idx]

        # Keep best M candidates
        best_M = population[:M]

        # Generate new candidates in the neighbourhood of each best candidate
        new_population = []
        for point in best_M:
            for _ in range(N // M):
                candidate = np.array([
                    np.clip(point[i] +
                        np.random.uniform(-0.1*(bounds[i][1]-bounds[i][0]),
                                                0.1*(bounds[i][1]-bounds[i][0])),
                            bounds[i][0], bounds[i][1])
                    for i in range(n)
                ])
                new_population.append(candidate)

        population = new_population
        population_costs = [cost_func(x) for x in population]

        # Record new population
        for point, cost in zip(population, population_costs):
            current_time = time.time() - start_time
```

```
            history.append((iteration_count, cost, point.copy(), current_time))
            iteration_count += 1

    # Return the best candidate from the final population
    best_idx = np.argmin(population_costs)
    best_x = population[best_idx]
    best_cost = population_costs[best_idx]

    return best_x, best_cost, history

# Define the two functions as cost functions that take a NumPy array x = [x, y]
def f1_func(x):
    return 7*(x[0]-1)**4 + 3*(x[1]-9)**2

def f2_func(x):
    return max(x[0]-1, 0) + 3*abs(x[1]-9)

def grad_f1(x):
    # Analytical gradient for f1: [28*(x-1)^3, 6*(y-9)],
    # could have used sympy to get this, but just for simplicity just borrowed from
        week 4 assignment
    return np.array([28*(x[0]-1)**3, 6*(x[1]-9)], dtype=float)

def grad_f2(x):
    # Subgradient for f2
    # For x: derivative of max(x-1, 0)
    if x[0] > 1:
        grad_x = 1.0
    elif x[0] < 1:
        grad_x = 0.0
    else:
        grad_x = 0.5  # subgradient at the non-differentiable point
    # For y: derivative of 3*|y-9|
    if x[1] > 9:
        grad_y = 3.0
    elif x[1] < 9:
        grad_y = -3.0
    else:
        grad_y = 0.0
    return np.array([grad_x, grad_y], dtype=float)

# Add counters for function and gradient evaluations
f1_eval_count = 0
f2_eval_count = 0
grad_f1_eval_count = 0
grad_f2_eval_count = 0

def count_calls(counter_name):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            globals()[counter_name] += 1
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Redefine the functions with call counting
@count_calls('f1_eval_count')
def f1_func_counted(x):
    return 7*(x[0]-1)**4 + 3*(x[1]-9)**2

@count_calls('f2_eval_count')
def f2_func_counted(x):
    return max(x[0]-1, 0) + 3*abs(x[1]-9)

@count_calls('grad_f1_eval_count')
def grad_f1_counted(x):
    # Analytical gradient for f1: [28*(x-1)^3, 6*(y-9)]
    return np.array([28*(x[0]-1)**3, 6*(x[1]-9)], dtype=float)

@count_calls('grad_f2_eval_count')
def grad_f2_counted(x):
    # Subgradient for f2
    if x[0] > 1:
        grad_x = 1.0
    elif x[0] < 1:
```

```
            grad_x = 0.0
        else:
            grad_x = 0.5  # subgradient at the non-differentiable point

        if x[1] > 9:
            grad_y = 3.0
        elif x[1] < 9:
            grad_y = -3.0
        else:
            grad_y = 0.0
        return np.array([grad_x, grad_y], dtype=float)

    def gradient_descent(cost_func, grad_func, init, alpha=0.001, max_iter=200,
        tol=1e-6):
        start_time = time.time()
        x = np.array(init, dtype=float)
        history = [(0, cost_func(x), x.copy(), 0.0)]  # include time

        for it in range(1, max_iter+1):
            current_time = time.time() - start_time
            grad = grad_func(x)
            if np.linalg.norm(grad) < tol:
                break
            x = x - alpha * grad
            history.append((it, cost_func(x), x.copy(), current_time))

        return x, history

    # Function to run multiple instances of global random search and track statistics
    def run_multiple_global_random_search(cost_func, n, bounds, N, num_runs=10):
        all_runs_history = []
        best_costs = []
        total_func_evals = []
        total_times = []

        for run in range(num_runs):
            # Reset counters for each run
            if cost_func == f1_func_counted:
                globals()['f1_eval_count'] = 0
                func_counter = 'f1_eval_count'
            else:
                globals()['f2_eval_count'] = 0
                func_counter = 'f2_eval_count'

            best_x, best_cost, history = global_random_search(cost_func, n, bounds, N)

            all_runs_history.append(history)
            best_costs.append(best_cost)
            total_func_evals.append(globals()[func_counter])
            total_times.append(history[-1][3])  # Last recorded time

        return all_runs_history, best_costs, total_func_evals, total_times

    # Function to run multiple instances of population-based random search
    def run_multiple_population_random_search(cost_func, n, bounds, N, M, iterations,
        num_runs=10):
        all_runs_history = []
        best_costs = []
        total_func_evals = []
        total_times = []

        for run in range(num_runs):
            # Reset counters for each run
            if cost_func == f1_func_counted:
                globals()['f1_eval_count'] = 0
                func_counter = 'f1_eval_count'
            else:
                globals()['f2_eval_count'] = 0
                func_counter = 'f2_eval_count'

            best_x, best_cost, history = population_random_search(cost_func, n, bounds,
                N, M, iterations)

            all_runs_history.append(history)
            best_costs.append(best_cost)
            total_func_evals.append(globals()[func_counter])
```

```
            total_times.append(history[-1][3])  # Last recorded time

        return all_runs_history, best_costs, total_func_evals, total_times

    # Function for plotting comparison results
    def plot_optimization_comparison(gd_history, grs_histories, prs_histories,
        func_name, metric='iterations'):
        plt.figure(figsize=(12, 6))

        # Extract data based on the metric (iterations, func_evals, or time)
        if metric == 'iterations':
            x_gd = [it for it, _, _, _ in gd_history]
            y_gd = [cost for _, cost, _, _ in gd_history]

            # For GRS and PRS, compute statistics across runs
            x_grs_all = []
            y_grs_all = []
            for history in grs_histories:
                x_grs = [it for it, _, _, _ in history]
                y_grs = [cost for _, cost, _, _ in history]
                if len(x_grs_all) < len(x_grs):
                    x_grs_all = x_grs
                y_grs_all.append(y_grs)

            x_prs_all = []
            y_prs_all = []
            for history in prs_histories:
                x_prs = [it for it, _, _, _ in history]
                y_prs = [cost for _, cost, _, _ in history]
                if len(x_prs_all) < len(x_prs):
                    x_prs_all = x_prs
                y_prs_all.append(y_prs)

        elif metric == 'time':
            x_gd = [t for _, _, _, t in gd_history]
            y_gd = [cost for _, cost, _, _ in gd_history]

            x_grs_all = []
            y_grs_all = []
            for history in grs_histories:
                x_grs = [t for _, _, _, t in history]
                y_grs = [cost for _, cost, _, _ in history]
                if len(x_grs_all) < len(x_grs):
                    x_grs_all = x_grs
                y_grs_all.append(y_grs)

            x_prs_all = []
            y_prs_all = []
            for history in prs_histories:
                x_prs = [t for _, _, _, t in history]
                y_prs = [cost for _, cost, _, _ in history]
                if len(x_prs_all) < len(x_prs):
                    x_prs_all = x_prs
                y_prs_all.append(y_prs)

        # Ensure all runs have the same length for averaging
        min_length_grs = min(len(y) for y in y_grs_all)
        min_length_prs = min(len(y) for y in y_prs_all)

        y_grs_trimmed = [y[:min_length_grs] for y in y_grs_all]
        y_prs_trimmed = [y[:min_length_prs] for y in y_prs_all]

        # Calculate mean and std dev
        y_grs_mean = [statistics.mean(costs) for costs in zip(*y_grs_trimmed)]
        y_grs_std = [statistics.stdev(costs) if len(costs) > 1 else 0 for costs in
            zip(*y_grs_trimmed)]

        y_prs_mean = [statistics.mean(costs) for costs in zip(*y_prs_trimmed)]
        y_prs_std = [statistics.stdev(costs) if len(costs) > 1 else 0 for costs in
            zip(*y_prs_trimmed)]

        # Plot the data
        plt.plot(x_gd, y_gd, 'r-', label=f'Gradient␣Descent')

        plt.plot(x_grs_all[:min_length_grs], y_grs_mean, 'g-', label=f'Global␣Random␣
            Search␣(Mean)')
```

```
        plt.fill_between(x_grs_all[:min_length_grs],
                         [max(y_grs_mean[i] - y_grs_std[i], 0) for i in
                             range(min_length_grs)],
                         [y_grs_mean[i] + y_grs_std[i] for i in range(min_length_grs)],
                         color='g', alpha=0.3)

        plt.plot(x_prs_all[:min_length_prs], y_prs_mean, 'b-', label=f'Population Random
            Search (Mean)')
        plt.fill_between(x_prs_all[:min_length_prs],
                         [max(y_prs_mean[i] - y_prs_std[i], 0) for i in
                             range(min_length_prs)],
                         [y_prs_mean[i] + y_prs_std[i] for i in range(min_length_prs)],
                         color='b', alpha=0.3)

        plt.xlabel(f'{metric.capitalize()}')
        plt.ylabel('Cost')
        plt.title(f'Cost vs {metric.capitalize()} for {func_name}')
        plt.legend()
        plt.grid(True)
        if metric != 'time':
            plt.xscale('log')
        plt.yscale('log')
        plt.show()

    # Function to run the comparison
    def run_comparison():
        # Set the search bounds and parameters
        bounds = [(-10, 10), (-10, 20)]  # Adjusted bounds for illustration
        n_params = 2
        N_samples = 500  # Number of random samples for global random search
        M_best = 10      # Number of best candidates to keep in population-based search
        pop_iterations = 5

        # Run experiments for f1
        print("\nRunning experiments for f1...")

        # Run gradient descent for f1
        globals()['f1_eval_count'] = 0
        globals()['grad_f1_eval_count'] = 0
        gd_x_f1_tracked, gd_history_f1_tracked = gradient_descent(f1_func_counted,
            grad_f1_counted, [5.0, 0.0], alpha=0.001)
        gd_f1_func_evals = f1_eval_count
        gd_f1_grad_evals = grad_f1_eval_count

        # Run multiple global random searches for f1
        grs_histories_f1, grs_best_costs_f1, grs_func_evals_f1, grs_times_f1 =
            run_multiple_global_random_search(
            f1_func_counted, n_params, bounds, N_samples, num_runs=10)

        # Run multiple population random searches for f1
        prs_histories_f1, prs_best_costs_f1, prs_func_evals_f1, prs_times_f1 =
            run_multiple_population_random_search(
            f1_func_counted, n_params, bounds, N_samples, M_best, pop_iterations,
                num_runs=10)

        # Run experiments for f2
        print("\nRunning experiments for f2...")

        # Run gradient descent for f2
        globals()['f2_eval_count'] = 0
        globals()['grad_f2_eval_count'] = 0
        gd_x_f2_tracked, gd_history_f2_tracked = gradient_descent(f2_func_counted,
            grad_f2_counted, [0.0, 0.0], alpha=0.001)
        gd_f2_func_evals = f2_eval_count
        gd_f2_grad_evals = grad_f2_eval_count

        # Run multiple global random searches for f2
        grs_histories_f2, grs_best_costs_f2, grs_func_evals_f2, grs_times_f2 =
            run_multiple_global_random_search(
            f2_func_counted, n_params, bounds, N_samples, num_runs=10)

        # Run multiple population random searches for f2
        prs_histories_f2, prs_best_costs_f2, prs_func_evals_f2, prs_times_f2 =
            run_multiple_population_random_search(
            f2_func_counted, n_params, bounds, N_samples, M_best, pop_iterations,
                num_runs=10)
```

```
# Print summary statistics
print("\nComparison␣Summary␣for␣f1:")
print(f"{'Method':<25}␣{'Best␣Cost␣(mean+-std)':<30}␣{'Function␣Evals':>15}␣
    {'Gradient␣Evals':>15}␣{'Time␣(s)':>10}")
print("-" * 100)
print(f"{'Gradient␣Descent':<25}␣{f1_func(gd_x_f1_tracked):<30.6f}␣
    {gd_f1_func_evals:>15d}␣{gd_f1_grad_evals:>15d}␣
    {gd_history_f1_tracked[-1][3]:>10.6f}")
print(f"{'Global␣Random␣Search':<25}␣
    {statistics.mean(grs_best_costs_f1):.6f}+-{statistics.stdev(grs_best_costs_f1):.6f}␣
    {statistics.mean(grs_func_evals_f1):>15.1f}␣{'N/A':>15}␣
    {statistics.mean(grs_times_f1):>10.6f}")
print(f"{'Population␣Random␣Search':<25}␣
    {statistics.mean(prs_best_costs_f1):.6f}+-{statistics.stdev(prs_best_costs_f1):.6f}␣
    {statistics.mean(prs_func_evals_f1):>15.1f}␣{'N/A':>15}␣
    {statistics.mean(prs_times_f1):>10.6f}")

print("\nComparison␣Summary␣for␣f2:")
print(f"{'Method':<25}␣{'Best␣Cost␣(mean+-std)':<30}␣{'Function␣Evals':>15}␣
    {'Gradient␣Evals':>15}␣{'Time␣(s)':>10}")
print("-" * 100)
print(f"{'Gradient␣Descent':<25}␣{f2_func(gd_x_f2_tracked):<30.6f}␣
    {gd_f2_func_evals:>15d}␣{gd_f2_grad_evals:>15d}␣
    {gd_history_f2_tracked[-1][3]:>10.6f}")
print(f"{'Global␣Random␣Search':<25}␣
    {statistics.mean(grs_best_costs_f2):.6f}+-{statistics.stdev(grs_best_costs_f2):.6f}␣
    {statistics.mean(grs_func_evals_f2):>15.1f}␣{'N/A':>15}␣
    {statistics.mean(grs_times_f2):>10.6f}")
print(f"{'Population␣Random␣Search':<25}␣
    {statistics.mean(prs_best_costs_f2):.6f}+-{statistics.stdev(prs_best_costs_f2):.6f}␣
    {statistics.mean(prs_func_evals_f2):>15.1f}␣{'N/A':>15}␣
    {statistics.mean(prs_times_f2):>10.6f}")

# Plot comparison results for f1
print("\nPlotting␣comparison␣for␣f1...")
plot_optimization_comparison(gd_history_f1_tracked, grs_histories_f1,
    prs_histories_f1, "f1", metric='iterations')
plot_optimization_comparison(gd_history_f1_tracked, grs_histories_f1,
    prs_histories_f1, "f1", metric='time')

# Plot comparison results for f2
print("\nPlotting␣comparison␣for␣f2...")
plot_optimization_comparison(gd_history_f2_tracked, grs_histories_f2,
    prs_histories_f2, "f2", metric='iterations')
plot_optimization_comparison(gd_history_f2_tracked, grs_histories_f2,
    prs_histories_f2, "f2", metric='time')

# Extract best parameters and costs for each method
best_params = {
    'f1': {
        'gd': {
            'params': gd_x_f1_tracked,
            'cost': f1_func(gd_x_f1_tracked)
        },
        'grs': {
            'params': min(grs_histories_f1[np.argmin(grs_best_costs_f1)],
                key=lambda x: x[1])[2],
            'cost': min(grs_best_costs_f1)
        },
        'prs': {
            'params': min(prs_histories_f1[np.argmin(prs_best_costs_f1)],
                key=lambda x: x[1])[2],
            'cost': min(prs_best_costs_f1)
        }
    },
    'f2': {
        'gd': {
            'params': gd_x_f2_tracked,
            'cost': f2_func(gd_x_f2_tracked)
        },
        'grs': {
            'params': min(grs_histories_f2[np.argmin(grs_best_costs_f2)],
                key=lambda x: x[1])[2],
            'cost': min(grs_best_costs_f2)
        },
```

```
            'prs': {
                'params': min(prs_histories_f2[np.argmin(prs_best_costs_f2)],
                    key=lambda x: x[1])[2],
                'cost': min(prs_best_costs_f2)
            }
        }
    }

    # Print best parameters and costs
    print("\nBest Parameters and Costs:")
    print("\nFor f1 function:")
    print(f"{'Method':<25} {'Best Parameters':<40} {'Best Cost':<15}")
    print("-" * 80)
    print(f"{'Gradient Descent':<25} {str(best_params['f1']['gd']['params']):<40}
        {best_params['f1']['gd']['cost']:<15.6f}")
    print(f"{'Global Random Search':<25}
        {str(best_params['f1']['grs']['params']):<40}
        {best_params['f1']['grs']['cost']:<15.6f}")
    print(f"{'Population Random Search':<25}
        {str(best_params['f1']['prs']['params']):<40}
        {best_params['f1']['prs']['cost']:<15.6f}")

    print("\nFor f2 function:")
    print(f"{'Method':<25} {'Best Parameters':<40} {'Best Cost':<15}")
    print("-" * 80)
    print(f"{'Gradient Descent':<25} {str(best_params['f2']['gd']['params']):<40}
        {best_params['f2']['gd']['cost']:<15.6f}")
    print(f"{'Global Random Search':<25}
        {str(best_params['f2']['grs']['params']):<40}
        {best_params['f2']['grs']['cost']:<15.6f}")
    print(f"{'Population Random Search':<25}
        {str(best_params['f2']['prs']['params']):<40}
        {best_params['f2']['prs']['cost']:<15.6f}")

    # Return the best parameters along with the histories
    return bounds, gd_history_f1_tracked, grs_histories_f1, prs_histories_f1,
        gd_history_f2_tracked, grs_histories_f2, prs_histories_f2


bounds, gd_history_f1_tracked, grs_histories_f1, prs_histories_f1,
    gd_history_f2_tracked, grs_histories_f2, prs_histories_f2 = run_comparison()


def plot_contour_with_best_trajectories(cost_func, gd_history, grs_histories,
    prs_histories, func_name, bounds):
    x_min, x_max = bounds[0]
    y_min, y_max = bounds[1]

    # Create mesh grid
    x_vals = np.linspace(x_min, x_max, 400)
    y_vals = np.linspace(y_min, y_max, 400)
    X, Y = np.meshgrid(x_vals, y_vals)

    # Evaluate cost function on the grid
    Z = np.array([[cost_func([x, y]) for x in x_vals] for y in y_vals])

    # Plot contours
    plt.figure(figsize=(10, 8))
    contour = plt.contourf(X, Y, Z, levels=50, cmap='viridis')
    plt.colorbar(label='Cost')

    # Plot GD path
    gd_points = np.array([x for _, _, x, _ in gd_history])
    plt.plot(gd_points[:, 0], gd_points[:, 1], 'r-', marker='o', markersize=4,
        label='Gradient Descent')

    # Find best GRS run (run with lowest final cost)
    grs_best_costs = [min(history, key=lambda x: x[1])[1] for history in
        grs_histories]
    best_grs_idx = np.argmin(grs_best_costs)
    best_grs_history = grs_histories[best_grs_idx]

    # Plot best GRS run
    grs_points = np.array([x for _, _, x, _ in best_grs_history])
    plt.scatter(grs_points[:, 0], grs_points[:, 1], c='lime', s=15, alpha=0.5,
        label='Global RS (Best Run)')
```

```
    # Find best PRS run (run with lowest final cost)
    prs_best_costs = [min(history, key=lambda x: x[1])[1] for history in
        prs_histories]
    best_prs_idx = np.argmin(prs_best_costs)
    best_prs_history = prs_histories[best_prs_idx]

    # Plot best PRS run
    prs_points = np.array([x for _, _, x, _ in best_prs_history])
    plt.scatter(prs_points[:, 0], prs_points[:, 1], c='blue', s=15, alpha=0.5,
        label='Population␣RS␣(Best␣Run)')

    # Highlight the best points found by each method
    best_gd_point = min(gd_history, key=lambda x: x[1])[2]
    best_grs_point = min(best_grs_history, key=lambda x: x[1])[2]
    best_prs_point = min(best_prs_history, key=lambda x: x[1])[2]

    plt.scatter(best_gd_point[0], best_gd_point[1], c='red', s=100, marker='*',
                edgecolor='black', linewidth=1.5, label='GD␣Best␣Point')
    plt.scatter(best_grs_point[0], best_grs_point[1], c='lime', s=100, marker='*',
                edgecolor='black', linewidth=1.5, label='GRS␣Best␣Point')
    plt.scatter(best_prs_point[0], best_prs_point[1], c='blue', s=100, marker='*',
                edgecolor='black', linewidth=1.5, label='PRS␣Best␣Point')

    plt.title(f'Contour␣Plot␣with␣Optimization␣Paths␣({func_name})')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)
    plt.show()


plot_contour_with_best_trajectories(f1_func, gd_history_f1_tracked,
    grs_histories_f1, prs_histories_f1, 'f1', bounds = [(-10, 10), (-10, 20)])
plot_contour_with_best_trajectories(f2_func, gd_history_f2_tracked,
    grs_histories_f2, prs_histories_f2, 'f2', bounds = [(-10, 10), (-10, 20)])

import os
import logging
import datetime

# Create logging directory if it doesn't exist
log_dir = "hyperparam_logs"
os.makedirs(log_dir, exist_ok=True)

# Set up loggers for each search method - one file per method
grs_logger = logging.getLogger('grs_logger')
grs_logger.setLevel(logging.INFO)
grs_handler = logging.FileHandler(f'{log_dir}/global_random_search.log')
grs_formatter = logging.Formatter('%(asctime)s␣-␣[Run:%(run_id)s]␣-␣%(message)s')
grs_handler.setFormatter(grs_formatter)
grs_logger.addHandler(grs_handler)

prs_logger = logging.getLogger('prs_logger')
prs_logger.setLevel(logging.INFO)
prs_handler = logging.FileHandler(f'{log_dir}/population_random_search.log')
prs_formatter = logging.Formatter('%(asctime)s␣-␣[Run:%(run_id)s]␣-␣%(message)s')
prs_handler.setFormatter(prs_formatter)
prs_logger.addHandler(prs_handler)

# Generate a unique run ID for this execution
run_id = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

# Create an adapter that adds the run_id to all log records
class RunAdapter(logging.LoggerAdapter):
    def process(self, msg, kwargs):
        if 'extra' not in kwargs:
            kwargs['extra'] = {}
        kwargs['extra']['run_id'] = self.extra['run_id']
        return msg, kwargs

# Create adapted loggers with run_id
grs_adapted_logger = RunAdapter(grs_logger, {'run_id': run_id})
prs_adapted_logger = RunAdapter(prs_logger, {'run_id': run_id})

# Wrap original global_random_search function to add logging
```

```
    def global_random_search_with_logging(cost_func, n, bounds, N):
        """Global random search with logging"""
        grs_adapted_logger.info(f"Starting Global Random Search with N={N} samples")
        grs_adapted_logger.info(f"Parameter bounds: {bounds}")

        # Create a wrapper function that logs each evaluation
        def logged_cost_func(params):
            return cost_func(params, grs_adapted_logger)

        start_time = time.time()
        best_cost = np.inf
        best_x = None
        history = []

        for i in range(N):
            current_time = time.time() - start_time
            candidate = np.array([np.random.uniform(low, high) for (low, high) in
                bounds])
            grs_adapted_logger.info(f"Sample {i+1}/{N}: Testing candidate {candidate}")
            cost = logged_cost_func(candidate)
            history.append((i, cost, candidate.copy(), current_time))

            grs_adapted_logger.info(f"Sample {i+1}/{N}: Cost = {cost:.6f}, Time =
                {current_time:.2f}s")

            if cost < best_cost:
                best_cost = cost
                best_x = candidate
                grs_adapted_logger.info(f"Sample {i+1}/{N}: New best cost =
                    {best_cost:.6f}")

        total_time = time.time() - start_time
        grs_adapted_logger.info(f"Global Random Search completed in {total_time:.2f}s")
        grs_adapted_logger.info(f"Best parameters: {best_x}")
        grs_adapted_logger.info(f"Best cost: {best_cost:.6f}")

        return best_x, best_cost, history

    # Wrap original population_random_search function to add logging
    def population_random_search_with_logging(cost_func, n, bounds, N, M, iterations):
        """Population-based random search with logging"""
        prs_adapted_logger.info(f"Starting Population Random Search with N={N}, M={M},
            iterations={iterations}")
        prs_adapted_logger.info(f"Parameter bounds: {bounds}")

        # Create a wrapper function that logs each evaluation
        def logged_cost_func(params):
            return cost_func(params, prs_adapted_logger)

        start_time = time.time()
        # Initial random population
        prs_adapted_logger.info("Generating initial random population")
        population = [np.array([np.random.uniform(low, high) for (low, high) in bounds])
            for _ in range(N)]
        population_costs = [logged_cost_func(x) for x in population]
        history = []
        iteration_count = 0

        # Record initial population
        for i, (point, cost) in enumerate(zip(population, population_costs)):
            current_time = time.time() - start_time
            history.append((iteration_count, cost, point.copy(), current_time))
            iteration_count += 1

        for it in range(iterations):
            prs_adapted_logger.info(f"Generation {it+1}/{iterations}: Sorting
                population")
            # Sort population based on cost (ascending order)
            sorted_idx = np.argsort(population_costs)
            population = [population[i] for i in sorted_idx]
            population_costs = [population_costs[i] for i in sorted_idx]

            # Keep best M candidates
            best_M = population[:M]
            prs_adapted_logger.info(f"Generation {it+1}/{iterations}: Best cost =
                {population_costs[0]:.6f}")
```

```
        # Generate new candidates in the neighbourhood of each best candidate
        prs_adapted_logger.info(f"Generation␣{it+1}/{iterations}:␣Generating␣new␣
            candidates")
        new_population = []
        for point_idx, point in enumerate(best_M):
            for j in range(N // M):
                candidate = np.array([
                    np.clip(point[i] +
                        np.random.uniform(-0.1*(bounds[i][1]-bounds[i][0]),
                                                    0.1*(bounds[i][1]-bounds[i][0])),
                            bounds[i][0], bounds[i][1])
                    for i in range(n)
                ])
                new_population.append(candidate)

        population = new_population
        prs_adapted_logger.info(f"Generation␣{it+1}/{iterations}:␣Evaluating␣new␣
            population")
        population_costs = [logged_cost_func(x) for x in population]

        # Record new population
        for point, cost in zip(population, population_costs):
            current_time = time.time() - start_time
            history.append((iteration_count, cost, point.copy(), current_time))
            iteration_count += 1

    # Return the best candidate from the final population
    best_idx = np.argmin(population_costs)
    best_x = population[best_idx]
    best_cost = population_costs[best_idx]

    total_time = time.time() - start_time
    prs_adapted_logger.info(f"Population␣Random␣Search␣completed␣in␣
        {total_time:.2f}s")
    prs_adapted_logger.info(f"Best␣parameters:␣{best_x}")
    prs_adapted_logger.info(f"Best␣cost:␣{best_cost:.6f}")

    return best_x, best_cost, history

def evaluate_model_hyperparams(hyperparams, logger=None):
    """Cost function for hyperparameter optimization with logging"""
    batch_size = int(hyperparams[0])
    learning_rate = hyperparams[1]
    beta_1 = hyperparams[2]
    beta_2 = hyperparams[3]
    epochs = int(hyperparams[4])

    if logger:
        logger.info(f"Evaluating␣hyperparams:␣batch_size={batch_size},␣
            lr={learning_rate:.6f},␣"
                    f"beta1={beta_1:.6f},␣beta2={beta_2:.6f},␣epochs={epochs}")

    # Build model with same architecture
    model = keras.Sequential([
        layers.Conv2D(16, (3,3), padding='same', activation='relu',
            input_shape=input_shape),
        layers.Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'),
        layers.Conv2D(32, (3,3), padding='same', activation='relu'),
        layers.Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'),
        layers.Dropout(0.5),
        layers.Flatten(),
        layers.Dense(num_classes, activation='softmax',
            kernel_regularizer=regularizers.l1(0.0001))
    ])

    # Compile with custom optimizer parameters
    optimizer = keras.optimizers.Adam(
        learning_rate=learning_rate,
        beta_1=beta_1,
        beta_2=beta_2
    )

    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
        metrics=["accuracy"])
```

```
        # Train model with early stopping to prevent overfitting
        early_stopping = keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=3,
            restore_best_weights=True
        )

        # Custom callback for logging
        class LoggingCallback(keras.callbacks.Callback):
            def on_epoch_end(self, epoch, logs=None):
                if logger and epoch % 5 == 0:  # Log every 5 epochs to avoid excessive
                    logs
                    logger.info(f"  Epoch {epoch+1}/{epochs}: loss={logs['loss']:.4f}, "
                                f"val_loss={logs['val_loss']:.4f}, "
                                f"accuracy={logs['accuracy']:.4f}, "
                                f"val_accuracy={logs['val_accuracy']:.4f}")

        # Train with validation split
        history = model.fit(
            x_train, y_train_cat,
            batch_size=batch_size,
            epochs=epochs,
            validation_split=0.2,
            callbacks=[early_stopping, LoggingCallback()] if logger else
                [early_stopping],
            verbose=0  # Silent training
        )

        # Get validation loss as our cost to minimize
        val_loss = min(history.history['val_loss'])
        val_acc = max(history.history['val_accuracy'])

        if logger:
            logger.info(f"  Result: val_loss={val_loss:.4f}, val_accuracy={val_acc:.4f},
                "
                        f"stopped at epoch {len(history.history['loss'])}/{epochs}")

        # Clear session to release memory
        tf.keras.backend.clear_session()

        return val_loss

    # Define parameter bounds
    n_params = 5
    bounds = [
        (16, 256),        # batch_size (will be converted to int)
        (0.0001, 0.01),   # learning_rate
        (0.8, 0.99),      # beta_1
        (0.9, 0.9999),    # beta_2
        (5, 30)           # epochs (will be converted to int)
    ]

    # Run global random search
    N_samples = 20  # Adjust based on your computational resources
    print("Running Global Random Search for hyperparameter optimization...")
    best_hyperparams_grs, best_loss_grs, history_grs = global_random_search_with_logging(
        evaluate_model_hyperparams, n_params, bounds, N_samples
    )

    # Run population-based random search
    M_best = 3  # Keep best 3 candidates
    pop_iterations = 3  # Run for 3 generations
    print("Running Population-based Random Search for hyperparameter optimization...")
    best_hyperparams_prs, best_loss_prs, history_prs =
        population_random_search_with_logging(
        evaluate_model_hyperparams, n_params, bounds, N_samples, M_best, pop_iterations
    )

    # Print results and also log them
    result_log = f"""
    Hyperparameter Optimization Results:

    Global Random Search:
    Best Loss: {best_loss_grs:.4f}
    Best Hyperparameters:
      Batch Size: {int(best_hyperparams_grs[0])}
```

```
  Learning Rate: {best_hyperparams_grs[1]:.6f}
  Beta 1: {best_hyperparams_grs[2]:.6f}
  Beta 2: {best_hyperparams_grs[3]:.6f}
  Epochs: {int(best_hyperparams_grs[4])}

Population Random Search:
Best Loss: {best_loss_prs:.4f}
Best Hyperparameters:
  Batch Size: {int(best_hyperparams_prs[0])}
  Learning Rate: {best_hyperparams_prs[1]:.6f}
  Beta 1: {best_hyperparams_prs[2]:.6f}
  Beta 2: {best_hyperparams_prs[3]:.6f}
  Epochs: {int(best_hyperparams_prs[4])}
"""

print(result_log)
grs_adapted_logger.info(result_log)
prs_adapted_logger.info(result_log)


# Compare metrics between methods
print("\nComparison between Global Random Search and Population Random Search:")
print(f"{'Metric':<30} {'Global RS':<15} {'Population RS':<15} {'Difference (PRS - GRS)':<20}")
print("-" * 80)

# Compare final validation loss
print(f"{'Best validation loss':<30} {best_loss_grs:<15.6f} {best_loss_prs:<15.6f} {best_loss_prs - best_loss_grs:<20.6f}")

# Compare number of function evaluations
grs_evals = len(history_grs)
prs_evals = len(history_prs)
print(f"{'Total evaluations':<30} {grs_evals:<15d} {prs_evals:<15d} {prs_evals - grs_evals:<20d}")

# Compare time efficiency
grs_time = history_grs[-1][3]  # Last time entry
prs_time = history_prs[-1][3]  # Last time entry
print(f"{'Total search time (s)':<30} {grs_time:<15.2f} {prs_time:<15.2f} {prs_time - grs_time:<20.2f}")

# Compare average evaluation time
grs_avg_time = grs_time / grs_evals
prs_avg_time = prs_time / prs_evals
print(f"{'Avg evaluation time (s)':<30} {grs_avg_time:<15.2f} {prs_avg_time:<15.2f} {prs_avg_time - grs_avg_time:<20.2f}")

# Calculate and compare convergence rate
# (how quickly each method finds a good solution)
def get_best_loss_at_evals(history, eval_points):
    """Get best loss found after n evaluations"""
    results = []
    for n in eval_points:
        if n > len(history):
            results.append(np.nan)
        else:
            best_at_n = min([h[1] for h in history[:n]])
            results.append(best_at_n)
    return results

# Check best loss at different evaluation points
eval_checkpoints = [5, 10, 20, 50, 100, min(len(history_grs), len(history_prs))]
eval_checkpoints = [n for n in eval_checkpoints if n <= min(len(history_grs), len(history_prs))]

grs_convergence = get_best_loss_at_evals(history_grs, eval_checkpoints)
prs_convergence = get_best_loss_at_evals(history_prs, eval_checkpoints)

print("\nConvergence comparison:")
print(f"{'Evaluations':<15} {'GRS best loss':<15} {'PRS best loss':<15} {'Difference':<15}")
print("-" * 60)
for i, n in enumerate(eval_checkpoints):
    print(f"{n:<15d} {grs_convergence[i]:<15.6f} {prs_convergence[i]:<15.6f} {prs_convergence[i] - grs_convergence[i]:<15.6f}")
```

```
# Visualization of convergence
plt.figure(figsize=(10, 6))
grs_min_losses = np.minimum.accumulate([h[1] for h in history_grs])
prs_min_losses = np.minimum.accumulate([h[1] for h in history_prs])

plt.plot(range(1, len(grs_min_losses)+1), grs_min_losses, 'r-', label='Global Random
    Search')
plt.plot(range(1, len(prs_min_losses)+1), prs_min_losses, 'b-', label='Population
    Random Search')
plt.xlabel('Number of Evaluations')
plt.ylabel('Best Validation Loss')
plt.title('Convergence Comparison: GRS vs PRS')
plt.legend()
plt.grid(True)
plt.tight_layout()

convergence_plot_path = f"{log_dir}/convergence_comparison_{run_id}.png"
plt.savefig(convergence_plot_path)
plt.show()

# Log the comparison
with open(f"{log_dir}/method_comparison_{run_id}.txt", "w") as f:
    f.write("Comparison between Global Random Search and Population Random
        Search:\n\n")
    f.write(f"{'Metric':<30} {'Global RS':<15} {'Population RS':<15} {'Difference
        (PRS-GRS)':<20}\n")
    f.write("-" * 80 + "\n")
    f.write(f"{'Best validation loss':<30} {best_loss_grs:<15.6f}
        {best_loss_prs:<15.6f} {best_loss_prs-best_loss_grs:<20.6f}\n")
    f.write(f"{'Total evaluations':<30} {grs_evals:<15d} {prs_evals:<15d}
        {prs_evals-grs_evals:<20d}\n")
    f.write(f"{'Total search time (s)':<30} {grs_time:<15.2f} {prs_time:<15.2f}
        {prs_time-grs_time:<20.2f}\n")
    f.write(f"{'Avg evaluation time (s)':<30} {grs_avg_time:<15.2f}
        {prs_avg_time:<15.2f} {prs_avg_time-grs_avg_time:<20.2f}\n\n")

    f.write("Convergence comparison:\n")
    f.write(f"{'Evaluations':<15} {'GRS best loss':<15} {'PRS best loss':<15}
        {'Difference':<15}\n")
    f.write("-" * 60 + "\n")
    for i, n in enumerate(eval_checkpoints):
        f.write(f"{n:<15d} {grs_convergence[i]:<15.6f} {prs_convergence[i]:<15.6f}
            {prs_convergence[i]-grs_convergence[i]:<15.6f}\n")


# Train final model with best hyperparameters
best_hyperparams = best_hyperparams_prs if best_loss_prs < best_loss_grs else
    best_hyperparams_grs
best_batch_size = int(best_hyperparams[0])
best_learning_rate = best_hyperparams[1]
best_beta_1 = best_hyperparams[2]
best_beta_2 = best_hyperparams[3]
best_epochs = int(best_hyperparams[4])

final_logger = logging.getLogger('final_model')
final_logger.setLevel(logging.INFO)
final_handler = logging.FileHandler(f'{log_dir}/final_models.log')
final_formatter = logging.Formatter('%(asctime)s - [Run:%(run_id)s] - %(message)s')
final_handler.setFormatter(final_formatter)
final_logger.addHandler(final_handler)
final_adapted_logger = RunAdapter(final_logger, {'run_id': run_id})

final_adapted_logger.info("Training final model with best hyperparameters:")
final_adapted_logger.info(f"Batch Size: {best_batch_size}")
final_adapted_logger.info(f"Learning Rate: {best_learning_rate:.6f}")
final_adapted_logger.info(f"Beta 1: {best_beta_1:.6f}")
final_adapted_logger.info(f"Beta 2: {best_beta_2:.6f}")
final_adapted_logger.info(f"Epochs: {best_epochs}")

print("\nTraining final model with best hyperparameters...")
final_model = keras.Sequential([
    layers.Conv2D(16, (3,3), padding='same', activation='relu',
        input_shape=input_shape),
    layers.Conv2D(16, (3,3), strides=(2,2), padding='same', activation='relu'),
    layers.Conv2D(32, (3,3), padding='same', activation='relu'),
```

```
    layers.Conv2D(32, (3,3), strides=(2,2), padding='same', activation='relu'),
    layers.Dropout(0.5),
    layers.Flatten(),
    layers.Dense(num_classes, activation='softmax',
        kernel_regularizer=regularizers.l1(0.0001))
])

final_optimizer = keras.optimizers.Adam(
    learning_rate=best_learning_rate,
    beta_1=best_beta_1,
    beta_2=best_beta_2
)

final_model.compile(loss="categorical_crossentropy", optimizer=final_optimizer,
    metrics=["accuracy"])

# Custom callback for final model training logging
class FinalModelLoggingCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        final_adapted_logger.info(f"Epoch {epoch+1}/{best_epochs}: "
            loss={logs['loss']:.4f}, "
                        f"val_loss={logs['val_loss']:.4f}, "
                        f"accuracy={logs['accuracy']:.4f}, "
                        f"val_accuracy={logs['val_accuracy']:.4f}")

# Train final model
history = final_model.fit(
    x_train, y_train_cat,
    batch_size=best_batch_size,
    epochs=best_epochs,
    validation_split=0.1,
    callbacks=[FinalModelLoggingCallback()],
    verbose=1  # Show progress bars for final training
)

# Save the final model
model_path = f"{log_dir}/best_model_{run_id}.keras"
final_model.save(model_path)
final_adapted_logger.info(f"Model saved to {model_path}")

# Evaluate final model
test_preds = final_model.predict(x_test)
y_test_pred = np.argmax(test_preds, axis=1)
test_report = classification_report(np.argmax(y_test_cat, axis=1), y_test_pred)
print("Classification Report (Test):")
print(test_report)
final_adapted_logger.info("Classification Report (Test):\n" + test_report)

# Calculate and log final metrics
test_loss, test_acc = final_model.evaluate(x_test, y_test_cat, verbose=0)
final_adapted_logger.info(f"Final test loss: {test_loss:.4f}")
final_adapted_logger.info(f"Final test accuracy: {test_acc:.4f}")

# Plot and save the training history
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Val')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Val')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
history_plot_path = f"{log_dir}/training_history_{run_id}.png"
plt.savefig(history_plot_path)
final_adapted_logger.info(f"Training history plot saved to {history_plot_path}")
plt.show()
```

```
    # Plot and save confusion matrix
    cm = confusion_matrix(np.argmax(y_test_cat, axis=1), y_test_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
                             'Dog', 'Frog', 'Horse', 'Ship', 'Truck'],
                yticklabels=['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
                             'Dog', 'Frog', 'Horse', 'Ship', 'Truck'])
    plt.title('Confusion Matrix (Test Data)')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.xticks(rotation=45)
    plt.tight_layout()

    cm_plot_path = f"{log_dir}/confusion_matrix_{run_id}.png"
    plt.savefig(cm_plot_path)
    final_adapted_logger.info(f"Confusion matrix plot saved to {cm_plot_path}")
    plt.show()
```