International Centre for Education and Research (ICER)
VIT-Bangalore

A Comparative Study of Methods for Mitigating Catastrophic Forgetting in Neural Networks

CS7510 – PROJECT 2

REPORT

Submitted by

Aniket Shinde – 24MSP3038

In partial fulfilment for the award of the degree of

POST GRADUATE PROGRAMME

INTERNATIONAL CENTRE FOR HIGHER EDUCATION AND RESEARCH

VIT BANGALORE

March, 2025

International Centre for Education and Research (ICER)
VIT-Bangalore

BONAFIDE CERTIFICATE

Certified that this project report **"A Comparative Study of Methods for Mitigating Catastrophic Forgetting in Neural Networks"** is the bonafide record of work done by **"Aniket Shinde – 24MSP3038"** who carried out the project work under my supervision.

Signature of the Supervisor

Dr. G. Shiyamala Gowri

Associate Professor,

ICER

VIT Bangalore

Signature of Director

Prof. Prema M

Director,

ICER

VIT Bangalore.

Evaluation Date:

International Centre for Education and Research (ICER)
VIT-Bangalore

ACKNOWLEDGEMENT

I express my sincere gratitude to our director of ICER **Prof. Prema M.** for their support and for providing the required facilities for carrying out this study.

I wish to thank my faculty supervisor, **Dr. G. Shiyamala Gowri**, **Associate Professor,** ICER for extending help and encouragement throughout the project. Without her continuous guidance and persistent help, this project would not have been a success for me.

I am grateful to all the members of ICER, my beloved parents, and friends for extending the support, who helped us to overcome obstacles in the study.

TABLE OF CONTENTS

ABSTRACT

Catastrophic forgetting, a phenomenon in which neural networks lose previously acquired knowledge while learning new tasks, poses a significant challenge to the development of adaptable and robust artificial intelligence systems. This study conducts a comparative analysis of three prominent approaches for mitigating catastrophic forgetting in continual learning settings: **Regularization-Based Methods**, **Replay-Based Methods**, and **Architectural Methods**. The research addresses the central question: *How do these methods perform in retaining past knowledge while learning new tasks, and what are the key trade-offs in terms of accuracy retention, forgetting rate, and computational efficiency?*

The experiments are conducted using the **Permutated-MNIST dataset**, a benchmark for continual learning, under two incremental learning scenarios: **Task-Incremental Learning (Task-IL)**, where task identity is provided, and **Class-Incremental Learning (Class-IL)**, where the model must classify all seen classes without task labels. A **Vanilla Stochastic Gradient Descent (SGD)** baseline is established to quantify the extent of catastrophic forgetting. The three mitigation techniques are then evaluated: Regularization-Based Methods (L2 Regularization, Elastic Weight Consolidation, Synaptic Intelligence), Replay-Based Methods (Naïve Rehearsal, Experience Replay), and Architectural Methods (PackNet). Performance is measured using accuracy retention, forgetting rate, and computational efficiency.

The results reveal distinct trade-offs among the methods. In **Task-IL scenarios**, Regularization-Based Methods, particularly **Elastic Weight Consolidation (EWC)**, demonstrate superior accuracy retention by penalizing changes to critical weights for past tasks. In **Class-IL scenarios**, Replay-Based Methods, especially **Experience Replay**, outperform others by maintaining a diverse buffer of past task data, enabling effective knowledge consolidation. Architectural Methods, such as **PackNet**, show promise but face challenges in balancing stability and plasticity.

This study highlights the strengths and limitations of each approach, offering valuable insights for selecting appropriate continual learning strategies based on specific application requirements. The findings contribute to the broader goal of developing AI systems capable of learning sequentially without catastrophic forgetting, paving the way for more robust and adaptable models in real-world applications such as autonomous systems and personalized AI.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1. INTRODUCTION

The ability of artificial intelligence systems to learn continuously from sequential data streams is a cornerstone of their adaptability and effectiveness in real-world applications. However, a significant challenge in this domain is **catastrophic forgetting**, a phenomenon where neural networks lose previously acquired knowledge while learning new tasks. This issue hinders the development of AI systems that can operate in dynamic environments, such as autonomous vehicles, personalized recommendation systems, and robotics, where models must adapt to new information without forgetting past experiences.

The field of **continual learning** seeks to address this challenge by developing methods that enable models to learn tasks sequentially while retaining knowledge from previous tasks. Over the years, researchers have proposed various strategies to mitigate catastrophic forgetting, broadly categorized into **Regularization-Based Methods**, **Replay-Based Methods**, and **Architectural Methods**. Regularization-Based Methods, such as Elastic Weight Consolidation (EWC) and Synaptic Intelligence (SI), aim to protect critical weights for past tasks during new learning. Replay-Based Methods, including Naïve Rehearsal and Experience Replay, retain and reuse past task data to reinforce previous knowledge. Architectural Methods, like PackNet, dynamically allocate network resources to prevent interference between tasks. Despite these advancements, there is a lack of comprehensive comparative studies that evaluate the effectiveness of these approaches across different continual learning scenarios.

This study aims to fill this gap by conducting a systematic comparison of these three mitigation techniques under two incremental learning scenarios: **Task-Incremental Learning (Task-IL)** and **Class-Incremental Learning (Class-IL)**. The research addresses the following key questions:

1. How do Regularization-Based, Replay-Based, and Architectural Methods perform in mitigating catastrophic forgetting in Task-IL and Class-IL settings?

2. What are the trade-offs between these methods in terms of accuracy retention, forgetting rate, and computational efficiency?

3. Which method is most effective for specific scenarios, and what are their limitations?

The study employs the **Permutated-MNIST dataset**, a widely used benchmark for continual learning, to evaluate the performance of each method. A **Vanilla Stochastic Gradient Descent (SGD)** baseline is used to establish the extent of catastrophic forgetting, followed by experiments with the three mitigation techniques. Performance is measured using metrics such as accuracy retention, forgetting rate, and computational efficiency.

While this study provides valuable insights into the effectiveness of different continual learning strategies, it is important to note its scope and limitations. The research focuses on basic continual learning methods and excludes more advanced approaches, such as meta-learning, which are

beyond the scope of this work. Additionally, the study is limited to the Permutated-MNIST dataset, and the findings may not generalize to more complex or diverse datasets. Despite these limitations, this research contributes to the broader understanding of continual learning strategies and provides a foundation for future work in developing robust and adaptable AI systems.

2. LITERATURE REVIEW

Continual learning addresses one of the core challenges in artificial intelligence: enabling models to learn new tasks sequentially without forgetting previously acquired knowledge. This phenomenon, known as "catastrophic forgetting," occurs when neural networks overwrite existing parameters while adapting to new tasks.

Current approaches to continual learning can be organized into three main categories: replay-based methods, regularization-based methods, and architectural methods. Replay-based approaches (such as experience replay, generative replay, and iCaRL) maintain performance on previous tasks by storing or generating samples from past data distributions. Regularization-based methods (like Elastic Weight Consolidation and Learning without Forgetting) add constraints to the loss function to prevent important parameters from changing. Architectural techniques (such as PackNet and HAT) dedicate specific model components to individual tasks.

Evaluation frameworks for continual learning have been standardized through scenarios of increasing difficulty: Task-Incremental Learning (where task identity is provided), Domain-Incremental Learning (where task boundaries are known but identity is not provided), and Class-Incremental Learning (where task identity must be inferred). These frameworks help compare methods systematically and highlight the strengths and limitations of different approaches.

Empirical studies have shown that parameter isolation methods like PackNet perform exceptionally well in task-incremental settings with zero forgetting, but they require knowing task identity during inference. Replay-based methods demonstrate strong performance across all scenarios, particularly in class-incremental learning where regularization-based methods often fail. Model capacity, regularization techniques (especially dropout), and the stability-plasticity trade-off significantly impact continual learning performance.

Recent innovations include Bayesian approaches that combine generative regularization with probabilistic inference. Methods like Bayesian Generative Regularization (BGR) show promise in maintaining performance across tasks without storing past data, addressing both privacy concerns and computational limitations.

Despite these advances, several gaps remain in the literature. Clinical applications of continual learning face regulatory hurdles, as current FDA approval processes are designed for "locked" algorithms rather than continuously evolving systems. Additionally, there is limited research on effectively validating continual learning models after deployment and ensuring they maintain performance across highly diverse tasks.

The field is moving toward developing more generalizable representations through self-supervised learning and pre-training while exploring how continual learning can be integrated with other AI paradigms like foundation models, diffusion models, and embodied AI. Future research directions include addressing unbalanced task distributions, improving resource efficiency, and developing robust evaluation metrics for real-world continual learning applications.

3. OBJECTIVE

The primary objectives of this study are as follows:

1. **Focus of Study**:
    - Compare the effectiveness of **Regularization-Based Methods**, **Replay-Based Methods**, and **Architectural Methods** in mitigating catastrophic forgetting under two incremental learning scenarios:
        - **Task-Incremental Learning (Task-IL)**: Task identity is provided during inference.
        - **Class-Incremental Learning (Class-IL)**: Task identity is unknown, requiring a single classifier for all seen classes.
2. **Variables to Be Measured**:
    - **Accuracy Retention**: The ability of the model to maintain performance on previously learned tasks.
    - **Forgetting Rate**: The rate at which the model loses knowledge of past tasks as new tasks are learned.
    - **Computational Efficiency**: The resource requirements (e.g., memory, training time) of each method.
3. **Steps to Be Involved**:
    - **Data Preparation**: Prepare and preprocess the Permutated-MNIST dataset for continual learning experiments.
    - **Exploratory Data Analysis (EDA)**: Analyze dataset characteristics to inform model design and training.
    - **Forgetting Analysis**: Establish a baseline using Vanilla SGD to quantify catastrophic forgetting.

- **Model Implementation**: Implement and train models using Regularization-Based, Replay-Based, and Architectural Methods.
- **Training Across Learning Scenarios**: Evaluate models under Task-IL and Class-IL settings.
- **Performance Evaluation & Comparison**: Compare methods using accuracy retention, forgetting rate, and computational efficiency.
- **Inference & Conclusion**: Draw insights and conclusions based on experimental results.

4. **Limits of the Study**:
   - This study is a comparative analysis focused on basic continual learning methods and does not introduce new techniques.
   - The scope is limited to Task-IL and Class-IL scenarios, excluding Domain-IL due to its reliance on more advanced approaches like meta-learning.

4. PROPOSED METHODOLOGY

This study employs a systematic and comparative research design to evaluate the effectiveness of three approaches for mitigating catastrophic forgetting in neural networks: **Regularization-Based Methods**, **Replay-Based Methods**, and **Architectural Methods**. The rationale for this methodology lies in the need to understand the trade-offs between these approaches in different continual learning scenarios, specifically **Task-Incremental Learning (Task-IL)** and **Class-Incremental Learning (Class-IL)**. By comparing these methods under controlled conditions, the study aims to provide actionable insights into their strengths and limitations, guiding the development of more robust continual learning systems.

The primary dataset used in this study is the **Permutated-MNIST dataset**, derived from the widely recognized **MNIST dataset** of handwritten digits. The MNIST dataset consists of 60,000 training and 10,000 test images, each representing a 28x28 grayscale image of digits ranging from 0 to 9. To simulate continual learning scenarios, the Permutated-MNIST dataset is created by applying random pixel permutations to the original MNIST images, generating distinct but related tasks. This approach allows for a controlled evaluation of how well each method retains knowledge across sequential tasks.

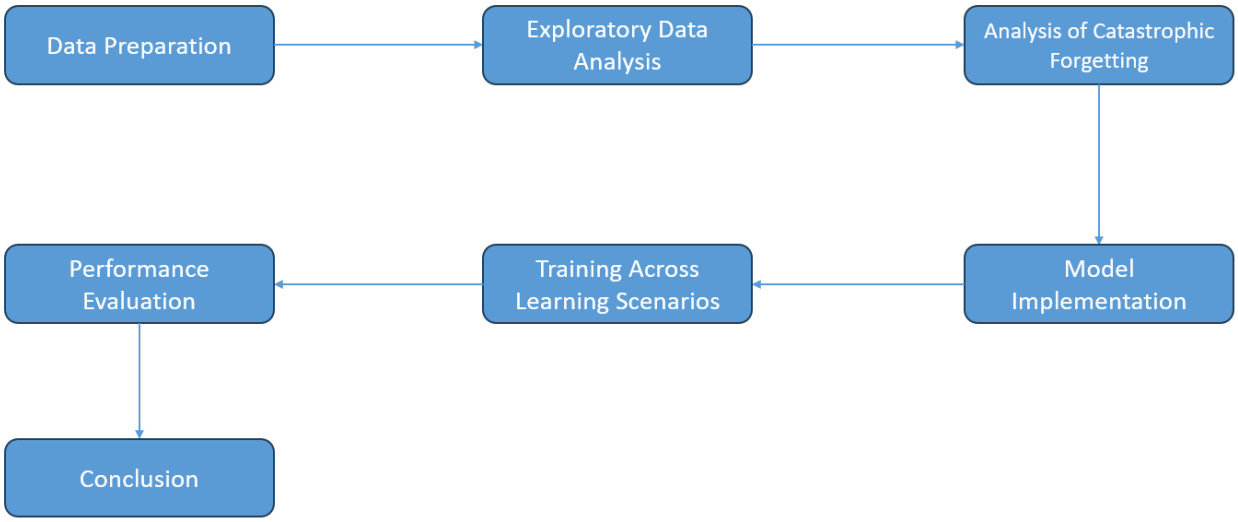


Fig 4.1 Methodology

The methodology is structured into the following key steps:

1. **Data Preparation**: The Permutated-MNIST dataset is preprocessed and divided into sequential tasks, each representing a unique permutation of the original MNIST images. This ensures that the model is exposed to a series of related but distinct classification problems, simulating real-world continual learning scenarios.

2. **Exploratory Data Analysis (EDA)**: A thorough analysis of the dataset is conducted to understand its characteristics. This includes examining the distribution of digit classes, pixel intensity patterns, and the clustering behavior of digit representations using techniques like t-SNE. The EDA reveals that while most digit classes form distinct clusters, some visually similar digits (e.g., 4 and 7) exhibit overlapping regions, highlighting potential challenges for classification.

3. **Analysis of Catastrophic Forgetting**: A **Vanilla Stochastic Gradient Descent (SGD)** baseline is established to quantify the extent of catastrophic forgetting. The baseline model is trained sequentially on the Permutated-MNIST tasks, and its performance on previous tasks is evaluated after learning new ones. This step provides a benchmark for comparing the effectiveness of the mitigation techniques.

4. **Model Implementation**: Three categories of mitigation techniques are implemented and evaluated:

    - **Regularization-Based Methods**: Techniques like L2 Regularization, Elastic Weight Consolidation (EWC), and Synaptic Intelligence (SI) are applied to penalize changes to critical weights for past tasks.

    - **Replay-Based Methods**: Naïve Rehearsal and Experience Replay are used to retain and reuse past task data during training, reinforcing previous knowledge.

    - **Architectural Methods**: PackNet is employed to dynamically allocate and freeze network parameters for different tasks, preventing interference between tasks.

5. **Training Across Learning Scenarios**: Each method is evaluated under both Task-IL and Class-IL settings. In Task-IL, task identity is provided during inference, allowing

separate classifiers for each task. In Class-IL, task identity is unknown, requiring a single classifier for all seen classes.

6. **Performance Evaluation**: The effectiveness of each method is measured using three key metrics:
    - **Accuracy Retention**: The ability of the model to maintain performance on previously learned tasks.
    - **Forgetting Rate**: The rate at which the model loses knowledge of past tasks as new tasks are learned.
    - **Computational Efficiency**: The resource requirements (e.g., memory, training time) of each method.
7. **Conclusion**: The results are analyzed to identify the strengths and limitations of each method, providing insights into their suitability for different continual learning scenarios.

To ensure the validity and reliability of the findings, the study employs rigorous experimental controls, including consistent training epochs, learning rates, and evaluation protocols across all methods. Additionally, the use of a well-established benchmark dataset (Permutated-MNIST) and standardized metrics enhances the generalizability of the results. By following this structured methodology, the study aims to deliver a comprehensive and reliable comparison of continual learning strategies for mitigating catastrophic forgetting.

5. TOOLS AND TECHNIQUES

This project leverages a combination of powerful deep learning frameworks, data manipulation libraries, and optimization techniques to implement and evaluate strategies for mitigating catastrophic forgetting in Class Incremental Learning (Class IL) and Task Incremental Learning (Task IL) scenarios. The following tools and techniques were utilized:

1. **PyTorch**
   PyTorch is an open-source deep learning framework developed by Facebook. It provides dynamic computation graphs, making it easier to modify models and debug during training. It was used to build and train the neural network models, handle datasets, and apply loss functions and optimizers for the continual learning experiments.

2. **Torchvision**
   Torchvision, an extension library for PyTorch, was used to import and preprocess the MNIST dataset. It offers built-in datasets, transformations, and utilities that simplify loading and preparing data for training and evaluation.

3. **skopt (Scikit-Optimize)**
   To fine-tune the hyperparameter λ (lambda) — which controls the regularization strength in methods like Elastic Weight Consolidation (EWC) and Synaptic Intelligence (SI) — Bayesian Optimization was implemented using skopt. Specifically, gp_minimize from skopt efficiently searched the lambda space to find the value that maximized performance, particularly task retention for early tasks.

4. **NumPy**
   NumPy is a foundational library for numerical computing in Python. It was extensively used for data manipulation, performance tracking, and computing performance metrics such as forgetting rates and task accuracies.

5. **Matplotlib & Seaborn**
   Matplotlib, paired with Seaborn, was used for visualizing training results. This included generating detailed learning curves, task accuracy plots, and forgetting rate graphs to analyze model performance across multiple tasks. Seaborn's enhanced visualization capabilities provided clearer, more informative plots.

6. **Time Module**
   The built-in Python time library was used to track the duration of model training for each task. This allowed performance analysis not only in terms of accuracy but also in training efficiency.

7. **Defaultdict (Collections)**
   To manage and store metrics like loss, accuracy, and forgetting rates seamlessly during each training phase, Python's defaultdict was used. This streamlined the logging of per-task performance metrics without needing to predefine array sizes.

6 IMPLEMENTATION

6.1 Task Incremental Learning (Task-IL) Scenario

6.1.1. ABOUT THE DATASET:

The dataset used in this project is the **MNIST Database** of handwritten digits, a benchmark dataset widely used in the machine learning community. It contains grayscale images of handwritten digits ranging from **0 to 9**.

- **Source of the Dataset:**
  Yann LeCun, Corinna Cortes, Christopher J.C. Burges. *"The MNIST Database of handwritten digits."* Link to dataset
- **Number of Observations:**
  - **Training Samples:** 60,000 images
  - **Test Samples:** 10,000 images
- **Column/Feature Details:**
  - **Input Shape:** (1, 28, 28) — 1 channel (grayscale), 28 pixels by 28 pixels
  - **Total Number of Classes:** 10 (Digits **0 to 9**)

The dataset's compact size and balanced class distribution make it ideal for studying catastrophic forgetting in incremental learning setups. Each image represents a single digit, offering a simple yet effective environment to evaluate performance across multiple learning scenarios.

6.1.2. EXPLORATORY DATA ANALYSIS (EDA)

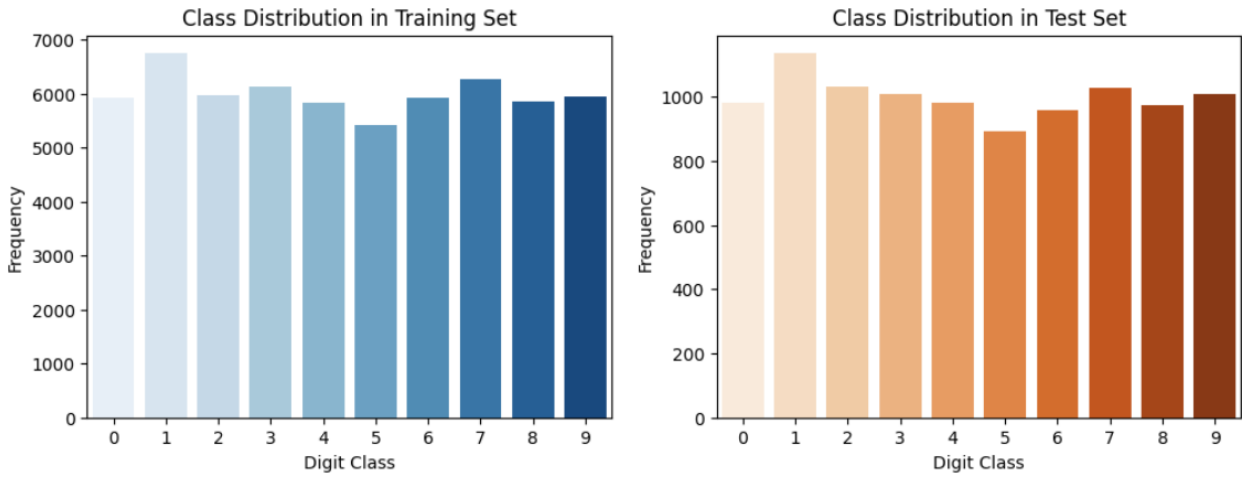


Fig 6.1 Class Distribution in Dataset

The plot shows the distribution of digit classes (0-9) in a training dataset, with each class having a relatively similar frequency, indicating a fairly balanced dataset.

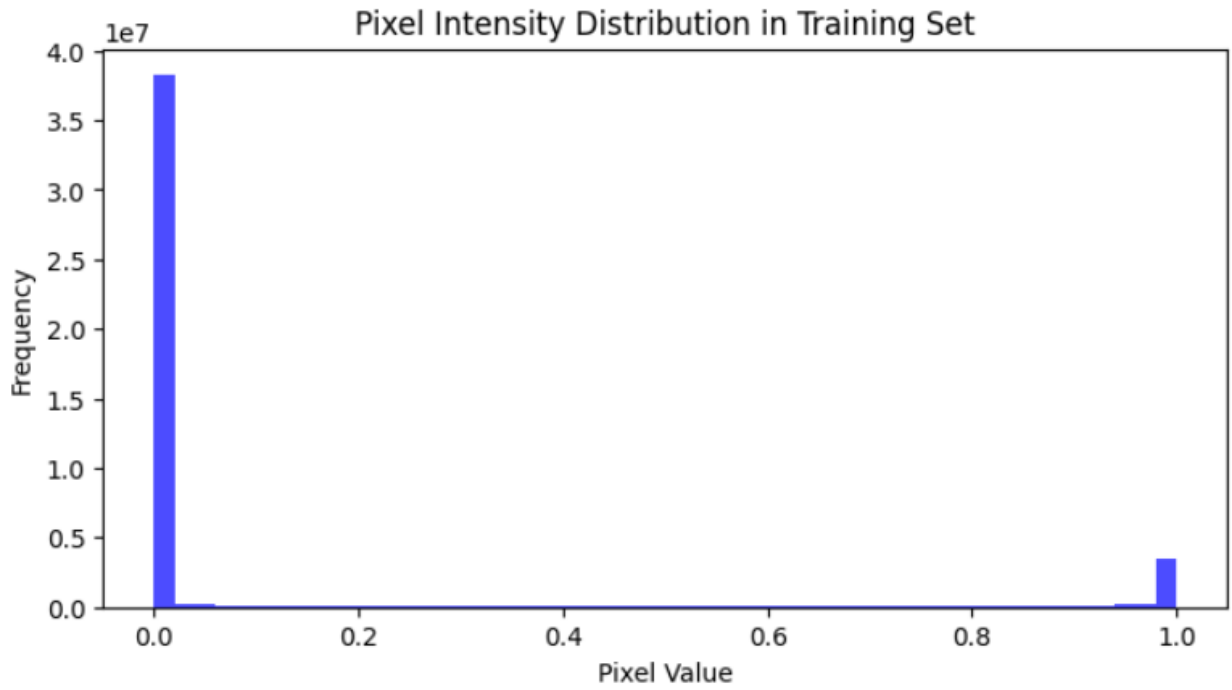


Fig 6.2 Pixel Intensity Distribution in Training Set

The pixel intensity distribution shows that MNIST consists primarily of **black background (0)** and **white handwritten strokes (1)**, with minimal mid-range values, ensuring the dataset does not need to be corrected for noise.

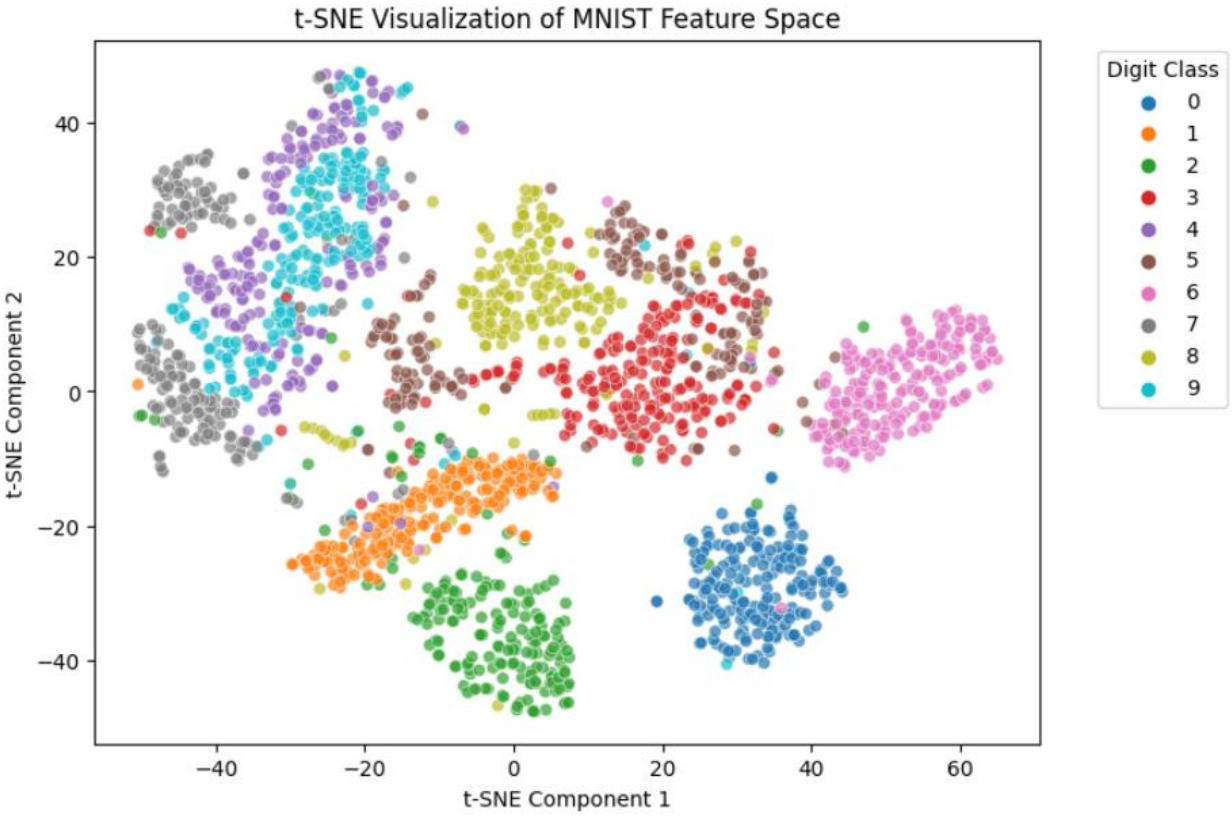


Fig 6.3 t-SNE Visualization of MNIST Feature Space

The t-SNE plot reveals that while most digit classes form distinct clusters, some visually similar digits exhibit overlapping regions, suggesting that a simple classifier may not achieve high classification accuracy.

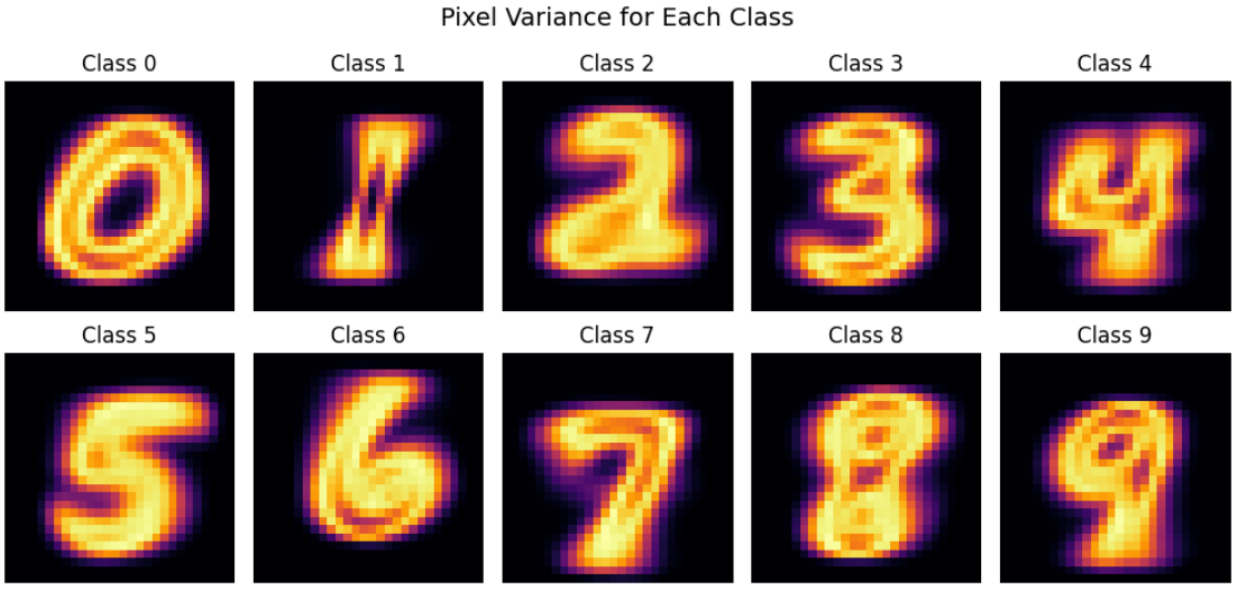


Fig 6.4 Pixel Variance for Each Class

This plot visualizes the variation in pixel intensity within each digit class. Digits like 4 and 7 show high variance due to differing handwriting styles, which could make them more challenging for models to classify accurately.

6.1.3. PREPROCESSING

To simulate a more challenging continual learning environment, the standard MNIST dataset was transformed into a **Permuted MNIST** variant. This setup introduces domain shifts across multiple tasks, forcing the model to adapt without forgetting prior knowledge.

- **Dataset Modification:**
  Each task applies a unique, random pixel permutation to the original MNIST images. This ensures that the visual structure of the digits is scrambled differently for every task, while the labels remain unchanged.
- **Number of Tasks:** 5 tasks were created, each representing a new permutation of the dataset.
- **Permutation Details:**
  - The images are flattened into a **784-dimensional vector** (28 × 28).
  - A random permutation is generated for each task, rearranging the pixel order.
  - The permuted vector is reshaped back to **(1, 28, 28)** for compatibility with the model.

- **Purpose of Permutation:**
  The permutation ensures that the tasks are distinct yet retain the same underlying data distribution (digit labels remain the same). This setup is crucial for evaluating **task incremental learning** strategies, where the model must continually adapt to new tasks without revisiting old ones.

By implementing this approach, the dataset evolves into **5 unique tasks**, each demanding the model to learn from a visually altered version of the digits — promoting robustness and exposing catastrophic forgetting when no mitigation strategies are applied.

6.1.4. METHODS

Vanilla Stochastic Gradient Descent (SGD)

The **Vanilla Stochastic Gradient Descent (SGD)** approach serves as the **baseline model** in this experiment to demonstrate the effects of catastrophic forgetting. It follows a straightforward training procedure where the model learns each task sequentially without any mechanisms to retain knowledge from prior tasks. This makes it particularly vulnerable to forgetting, as weights are updated purely based on the current task's data.

Key components of the model setup:

- **Architecture:** A simple feedforward neural network with **2 fully connected layers** and a **ReLU activation** function between them.
- **Training:** Each task is trained independently for **5 epochs** using **CrossEntropyLoss** as the criterion and **SGD** as the optimizer with a **learning rate of 0.01**.
- **Sequential Evaluation:** After learning each task, the model is evaluated on all previously learned tasks to observe performance degradation — highlighting the extent of catastrophic forgetting.
- **Metric Tracking:** The training tracks **loss**, **accuracy**, and **forgetting rate** — measuring how much the model forgets earlier tasks as it progresses.

This baseline setup offers a raw look at how the network performs when trained sequentially without any mitigation strategies, establishing a benchmark to compare the performance of more advanced methods.

The effect of forgetting

Evaluation Point	Task 1	Task 2	Task 3	Task 4	Task 5
After Task 1	99.9%	Not trained yet	Not trained yet	Not trained yet	Not trained yet
After Task 2	77.3%	97.4%	Not trained yet	Not trained yet	Not trained yet
After Task 3	48.1%	75.5%	98.6%	Not trained yet	Not trained yet
After Task 4	65.7%	82.7%	80.2%	99.2%	Not trained yet
After Task 5	41.3%	59.1%	18.2%	95.6%	96.7%

Table 6.1 Baseline Accuracies during the Training Process (Task-IL)

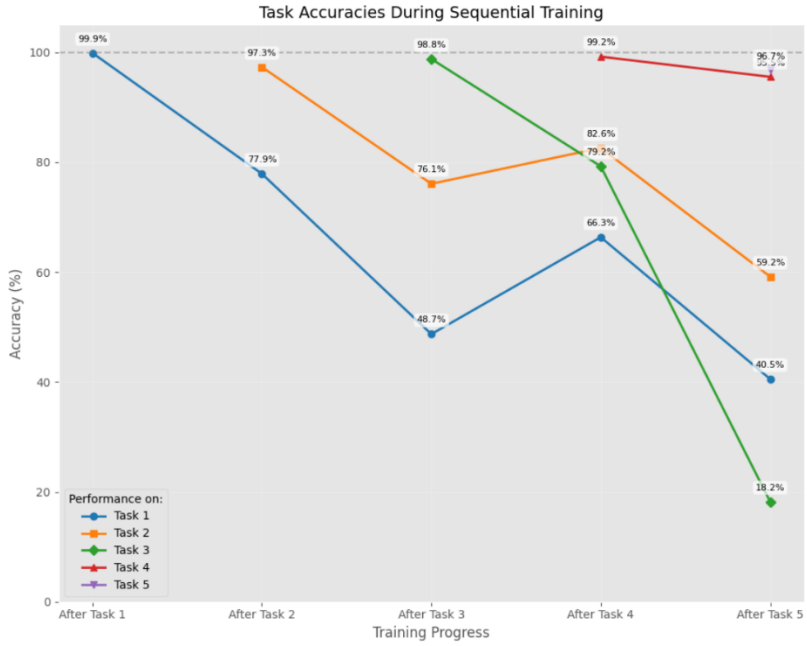


Fig 6.5 Task Accuracies During Sequential Training

In this plot, we can observe the effects of forgetting as new tasks are trained. A general downward trend in accuracy is observed after subsequent tasks are trained with a few exceptions of increased accuracy which can be attributed to factors such as positive transfer or randomness.

L2 Regularization [Regularization- based]

L2 regularization, also known as **weight decay**, is a penalty method that discourages large weight updates, helping to retain knowledge from earlier tasks. In this implementation, the model penalizes significant deviations from the previously learned weights, reducing the risk of catastrophic forgetting.

Key components of the method:

- **Regularization Term:** During training on new tasks, an additional **L2 penalty** is added to the loss function. This penalty measures the squared difference between current model parameters and saved initial parameters from prior tasks.
- **Controlled Updates:** The penalty strength is controlled by **λ (lambda)**, which determines how much weight changes are restricted. A higher lambda enforces stronger regularization.
- **Sequential Learning:** The model still learns tasks sequentially, but by keeping parameters close to those learned earlier, it maintains performance on old tasks better than vanilla SGD.

This approach enhances the model's stability, making it a simple yet effective baseline for regularization-based lifelong learning methods.

Elastic Weight Consolidation (EWC) [Regularization- based]

Elastic Weight Consolidation (EWC) is a regularization-based technique that combats catastrophic forgetting by selectively restricting weight updates for parameters crucial to previously learned tasks. It introduces a **penalty** based on the **Fisher Information Matrix**, which identifies important parameters that must remain close to their optimal values from prior tasks.

Key components of the method:

- **Fisher Information Matrix:** EWC computes this matrix during training to estimate how important each parameter is to the current task.
- **Selective Penalty:** When learning a new task, the model adds a penalty term to the loss function. This penalty is proportional to both the Fisher matrix and the squared difference between the current and previously learned parameters.
- **Lambda Control:** The strength of the penalty is controlled by **λ (lambda)** — a hyperparameter that determines how rigidly the model retains past knowledge.

By penalizing changes to important weights, EWC helps preserve old knowledge while still allowing flexibility to learn new tasks, striking a balance between stability and plasticity.

Synaptic Intelligence (SI) [Regularization- based]

Synaptic Intelligence (SI) is a regularization-based approach that helps mitigate catastrophic forgetting by preserving essential parameters for previous tasks. It tracks how much each weight contributes to the loss reduction during training, accumulating this "importance" information over time. This importance is then used to constrain changes to critical weights when learning new tasks.

Key components of the method:

- **Path Integral Tracking:** SI maintains a running total of parameter changes, weighted by their gradient contributions (path integral), reflecting how crucial each weight was to past tasks.
- **Importance (Omega):** After each task, SI computes an importance score (omega) for each parameter, ensuring weights essential to old tasks don't drift too far during new learning.
- **SI Penalty:** A penalty is added to the loss function based on the squared distance between the current and original weights, scaled by their omega values — preventing the model from overwriting important knowledge.

SI offers a dynamic, data-efficient way to preserve old task knowledge by anchoring significant parameters while allowing less important ones to adapt, making it a versatile tool for lifelong learning scenarios.

PackNet [Architectural]

PackNet is an architectural-based method designed to combat catastrophic forgetting by progressively "packing" new tasks into a single model. It achieves this by iteratively pruning and freezing parts of the network after each task, creating room for new tasks while preserving performance on previously learned ones.

Key components of the method:

- **Weight Pruning:** After training each task, a portion of the weights with the smallest magnitudes are pruned — freeing up capacity for the next task while retaining the most important connections.
- **Task-Specific Masks:** For each task, a binary mask is created to track which weights were retained. These masks are later applied to restore task-specific configurations during inference.
- **Weight Freezing:** Once a task is learned, the important weights are frozen to prevent them from being modified during subsequent training. This ensures that knowledge from previous tasks remains intact.

PackNet stands out by reusing a single network without expanding its size, making it efficient in terms of memory and computation while preserving task-specific knowledge. This makes it an effective approach for scenarios requiring sequential learning without catastrophic forgetting.

6.1.5. EVALUATION

Evaluation is essential for understanding how well the model performs across multiple tasks and how effectively the mitigation methods prevent catastrophic forgetting. In this project, several key metrics were tracked throughout training and after each task to assess model performance:

1. **Accuracy:** This measures the percentage of correct predictions among all predictions. For each task, accuracy is evaluated on both the current task and all previously learned tasks to observe how well the model retains knowledge.
2. **Forgetting Rate:** This measures how much performance degrades on earlier tasks after learning new ones. It is calculated as the difference between the model's accuracy on a task immediately after training it and the accuracy on that same task after learning subsequent tasks. A lower forgetting rate indicates better knowledge retention.
3. **Training Time:** The time taken to train each task is tracked to ensure the methods remain practical and efficient.
4. **Learning Curves:** Loss and accuracy are recorded during each epoch of training to monitor how quickly and effectively the model converges on each task. This helps visualize how different methods impact the stability and speed of learning.

Together, these metrics provide a comprehensive view of how well each approach prevents forgetting and maintains performance across sequential tasks.

6.2. Class Incremental Learning (Class-IL) Scenario

6.2.1. ABOUT THE DATASET

Same dataset as that for Task Incremental Learning

6.2.2. PREPROCESSING

For the Class Incremental Learning (CIL) setup, the MNIST dataset was reorganized into sequential tasks, each introducing new, non-overlapping classes. This setup simulates a scenario where the model encounters entirely new categories over time, without revisiting old ones.

Dataset Modification:
The dataset is divided into **5 tasks**, each containing **2 unique classes** (e.g., Task 1 has digits 0 and 1, Task 2 has 2 and 3, and so on).

Class Splitting Details:

- Each task filters the dataset to retain only images belonging to the task's assigned classes.
- The original 28 × 28 structure and labels are preserved.

Purpose of Class Splitting:
The separation ensures the model learns each task independently, forcing it to differentiate between new classes without retaining access to past data — a hallmark challenge in class incremental learning.

6.2.5. METHODS

Vanilla Stochastic Gradient Descent (SGD) [Baseline]

Vanilla Stochastic Gradient Descent (SGD) serves as the baseline in the **Class Incremental Learning (Class-IL)** scenario, demonstrating the extent of **catastrophic forgetting** when no mitigation strategies are applied.

The model sequentially learns **5 tasks**, each introducing **2 new classes**. Unlike Task Incremental Learning, where the task ID is known, the model here must classify across all previously seen classes.

After each task, the model is evaluated on the **cumulative test set** — containing data from all encountered classes so far — making it progressively harder to maintain performance on earlier classes. This highlights vanilla SGD's vulnerability to forgetting in class incremental setups, where old knowledge is overwritten by new class representations.

Naïve Rehearsal [Replay-based]

Naive Rehearsal is a replay-based method that mitigates **catastrophic forgetting** by storing a subset of data from previous tasks and **reintroducing it during training on new tasks**. This keeps the model anchored to older knowledge while learning new information, promoting better retention of earlier tasks.

Key components of the method:

- **Replay Buffer:** A memory buffer stores a small number of samples from each task after training. These samples serve as a lightweight proxy for revisiting older data without needing the full dataset.

- **Data Mixing:** When training on a new task, the model combines current task data with replayed data, ensuring the learning process considers both old and new tasks.
- **Balanced Replay:** The replay buffer is updated with a fixed number of examples per task. This prevents older tasks from being completely overshadowed by the newer ones, though the balance may degrade as more tasks are added.

Naive Rehearsal is a **straightforward, memory-efficient approach** that helps maintain performance on earlier tasks. However, **it's susceptible to data imbalance** — older tasks contribute less over time due to limited buffer size. Despite this, it remains a foundational technique for **class incremental learning** and sets the stage for more advanced replay methods.

Experience Replay [Replay-based]

Experience Replay is a replay-based approach that **maintains a buffer of past experiences** and **replays a mix of old and new data** during training. This continuous exposure helps the model retain knowledge from earlier tasks while integrating new information.

Key components of the method:

- **Replay Buffer:** A memory buffer stores samples from previous tasks. Unlike Naive Rehearsal, it **dynamically updates** — discarding older data when the buffer reaches capacity, ensuring memory efficiency.
- **Replay Sampling:** Each batch during training includes a combination of current task data and replayed data. This **reinforces past knowledge** alongside learning new information, reducing the chance of older tasks being overwritten.
- **Balanced Replay Ratio:** The training batch draws a portion of data from the buffer (e.g., 50% new, 50% replayed). This **maintains task balance**, ensuring older knowledge remains represented.

Experience Replay strikes a balance between **efficiency and performance** — it avoids the storage burden of keeping full datasets while still offering strong resistance to catastrophic forgetting. This makes it a powerful, scalable choice for **class incremental learning** scenarios.

6.2.6. EVALUATION

Evaluation in class incremental learning (CIL) requires a different approach from task incremental learning, as the model progressively encounters new classes without knowing which task a sample belongs to during testing. The evaluation focuses on **cumulative performance** across all seen classes, reflecting how well the model integrates new knowledge without forgetting prior classes.

Key metrics tracked include:

- **Cumulative Accuracy:** After each task, the model is evaluated on a combined test set of all classes seen so far. This measures the overall performance on previously learned classes alongside the newly introduced ones — providing a realistic measure of how well the model generalizes across all encountered classes.

- **Forgetting Rate:** Similar to task IL, forgetting rate is tracked to measure how much accuracy degrades on earlier classes after learning new ones. A lower forgetting rate indicates better knowledge retention.
- **Replay Efficiency (for replay-based methods):** For methods like Naive Rehearsal and Experience Replay, the size of the stored memory buffer is considered alongside performance to assess how efficiently the model balances memory constraints and accuracy.
- **Training Time:** The time taken for each task remains an important factor to ensure methods remain practical, especially as the number of classes increases.

This evaluation strategy reflects the true challenge of class incremental learning — where the model must continuously expand its knowledge base while preserving performance on previously learned classes, without task labels for guidance.

7. RESULTS AND DISCUSSIONS

7.1 Task Incremental Learning (Task-IL) Scenario

7.1.0 Baseline (Vanilla SGD)

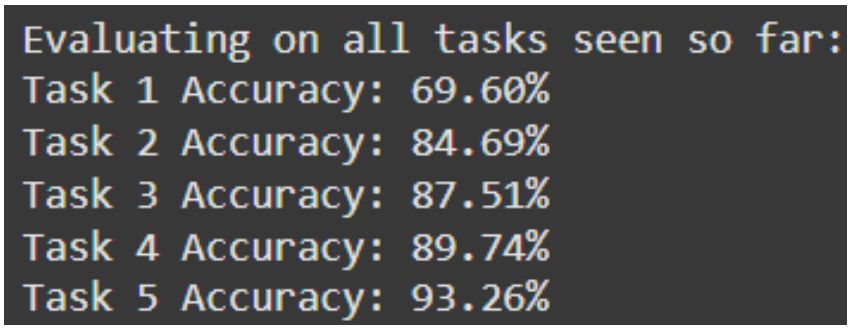


Table 7.1 Baseline Task Accuracies after all tasks are learned (Task-IL)

7.1.1 Regularization-based methods

L2 Regularization

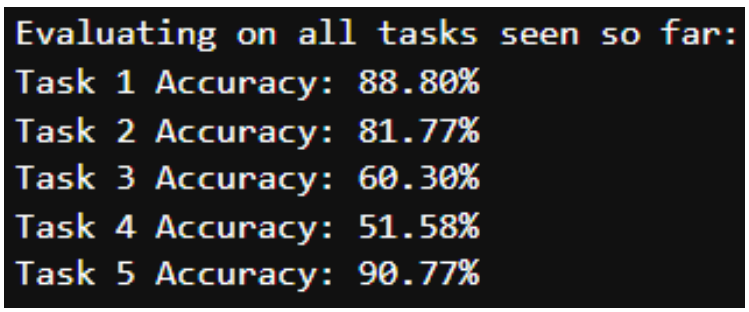


Table 7.2 L2 Regularization Task Accuracies after all tasks are learned (Task-IL)

Elastic Weight Consolidation (EWC)

```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 80.54%
Task 2 Accuracy: 85.56%
Task 3 Accuracy: 87.65%
Task 4 Accuracy: 90.53%
Task 5 Accuracy: 93.12%
```

Table 7.3 EWC Task Accuracies after all tasks are learned (Task-IL)

Synaptic Intelligence (SI)

```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 70.87%
Task 2 Accuracy: 83.39%
Task 3 Accuracy: 85.21%
Task 4 Accuracy: 90.29%
Task 5 Accuracy: 93.15%
```

Table 7.4 SI Task Accuracies after all tasks are learned (Task-IL)

7.1.2 Architectural method

PackNet

```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 89.71%
Task 2 Accuracy: 21.75%
Task 3 Accuracy: 12.42%
Task 4 Accuracy: 24.61%
Task 5 Accuracy: 24.20%
```

Table 7.5 PackNet Task Accuracies after all tasks are learned (Task-IL)

7.2 Class Incremental Learning (Class-IL) Scenario

7.1.0 Baseline (Vanilla SGD)

Task	Accuracy (%)
After Training Task 1	99.91
After Training Task 2	47.75
After Training Task 3	30.66
After Training Task 4	24.62
After Training Task 5	19.20

Table 7.6 Baseline Model accuracy as tasks are trained sequentially (Class-IL)

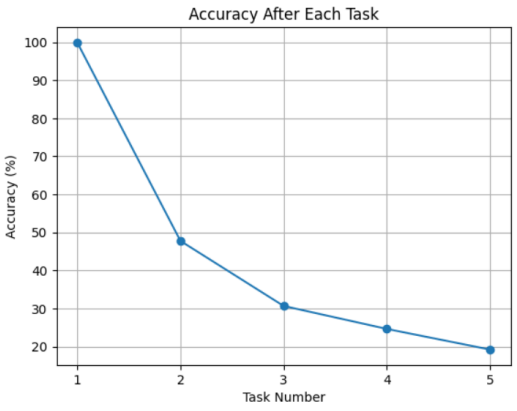


Fig 7.1 Baseline Accuracy After Each Task (Class-IL)

7.1.1 Replay-based methods

Naïve Rehearsal

Task	Accuracy (%)
After Training Task 1	99.91
After Training Task 2	54.94
After Training Task 3	44.12
After Training Task 4	44.17
After Training Task 5	34.93

Table 7.7 Naïve Rehearsal Model accuracy as tasks are trained sequentially (Class-IL)

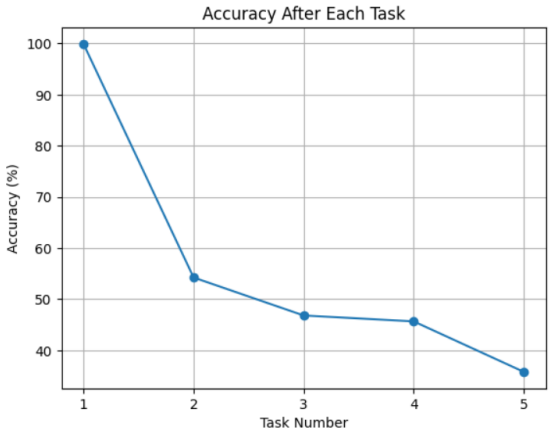


Fig 7.2 Naïve Rehearsal Accuracy After Each Task

Experience Replay

Task	Accuracy (%)
After Training Task 1	99.91
After Training Task 2	96.49
After Training Task 3	68.03
After Training Task 4	47.64
After Training Task 5	36.64

Table 7.8 Experience Replay accuracy as tasks are trained sequentially (Class-IL)

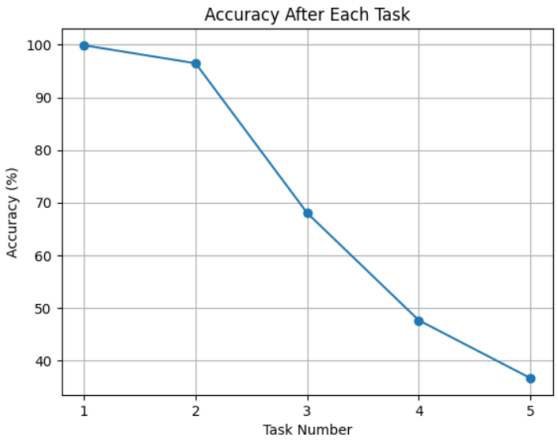


Fig 7.3 Experience Replay Accuracy After Each Task

8.CONCLUSION

In this study, I explored various mitigation strategies to combat catastrophic forgetting in both task incremental learning (Task-IL) and class incremental learning (Class-IL) scenarios. The experiments highlighted the strengths and limitations of regularization-based, architectural, and replay-based approaches.

For **Task-IL**, regularization methods proved effective in preserving performance on previously learned tasks. Among them, **Elastic Weight Consolidation (EWC)** emerged as the most successful, leveraging the Fisher Information Matrix to selectively protect parameters crucial to earlier tasks. **Synaptic Intelligence (SI)** followed closely, showing notable improvements by tracking the importance of weights through a path integral approach. **L2 regularization**, though

simpler, still outperformed the baseline **Vanilla Stochastic Gradient Descent (SGD)** by encouraging weight stability. **PackNet**, representing an architectural approach, demonstrated potential by progressively pruning and freezing weights. However, our implementation revealed limitations — the network favored the initial task excessively, restricting the model's adaptability to later tasks. This resulted in an unstable stability-plasticity balance, making PackNet's overall effectiveness inconclusive.

For **Class-IL**, where the model encounters new classes sequentially without task labels, replay-based strategies were evaluated. **Experience Replay** demonstrated significant superiority over **Naïve Rehearsal**. By maintaining a dynamic, diverse buffer of past samples, Experience Replay promoted better knowledge retention and improved cumulative accuracy across all learned classes. Naïve Rehearsal, despite its simplicity, still provided a measurable improvement over the baseline, reinforcing the importance of revisiting prior data in Class-IL settings.

Overall, the findings underscore the trade-offs between stability and plasticity inherent to continual learning methods. Regularization techniques effectively slow forgetting in Task-IL, while replay strategies — particularly Experience Replay — excel in Class-IL by reinforcing older knowledge. Architectural methods like PackNet show promise but require further refinement to achieve balanced performance across tasks. This work emphasizes the importance of aligning the mitigation strategy with the specific continual learning scenario for optimal performance.

9. FUTURE ENHANCEMENT

Several promising directions could further improve the effectiveness and efficiency of the implemented continual learning methods. **Hyperparameter optimization** — particularly fine-tuning lambda values, learning rates, and buffer sizes — could lead to better accuracy retention across both Task-IL and Class-IL setups. Exploring **hybrid approaches**, combining regularization methods with replay strategies, may strike a better balance between stability and plasticity, capitalizing on the strengths of each technique. Additionally, **streamlining the project structure** to eliminate redundant computations would improve overall time efficiency, enabling faster experimentation and analysis. Future work could also incorporate **advanced methods** — such as dynamic architectures or meta-learning strategies — to address limitations observed in approaches like PackNet. Lastly, **enhanced visualizations** could offer clearer insights into model performance, making it easier to interpret learning curves, forgetting rates, and task-specific accuracy trends. Together, these enhancements would create a more robust and adaptable framework for tackling catastrophic forgetting in continual learning scenarios.

10.APPENDICIES

10.1 FULL CODE

Exploratory Data Analysis (EDA)

```
import torch
```

```
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms

# Load MNIST Dataset
transform = transforms.Compose([transforms.ToTensor()])
train_data = datasets.MNIST(root="./data", train=True, download=True,
transform=transform)
test_data = datasets.MNIST(root="./data", train=False, download=True,
transform=transform)

# Split-MNIST Setup: Dividing into 5 tasks of binary classification
split_mnist_tasks = [(0,1), (2,3), (4,5), (6,7), (8,9)]

# Number of observations
num_train = len(train_data)
num_test = len(test_data)

# Feature details
sample_image, sample_label = train_data[0]
input_shape = sample_image.shape  # Shape of an image
num_classes = len(set(train_data.targets.numpy()))  # Unique class labels

# Print results
print(f"Number of Training Samples: {num_train}")
print(f"Number of Test Samples: {num_test}")
print(f"Input Feature Shape: {input_shape}")  # (1,28,28) -> 1 channel,
28x28 pixels
print(f"Number of Classes: {num_classes}")  # Should be 10 (0-9)

# Print task-wise split details
for i, (c1, c2) in enumerate(split_mnist_tasks):
    print(f"Task {i+1}: Class {c1} vs Class {c2}")

# Function to get interleaved samples for each class
def get_balanced_interleaved_samples(dataset, classes, num_samples=6):
    """Ensures an equal and alternating number of samples from both
classes."""
    class_samples = {c: [] for c in classes}

    # Collect samples for each class
    for img, label in dataset:
        if label in classes and len(class_samples[label]) < num_samples:
            class_samples[label].append(img)

    # Interleave samples (e.g., [0,1,0,1,0,1] instead of [0,0,0,1,1,1])
```

```
    interleaved_samples = [img for pair in zip(class_samples[classes[0]],
class_samples[classes[1]]) for img in pair]
    interleaved_labels = [classes[i % 2] for i in
range(len(interleaved_samples))]

    return interleaved_samples, interleaved_labels

# Plot sample images
fig, axes = plt.subplots(nrows=5, ncols=6, figsize=(8, 8))  # 5 tasks, 6
images per task

# Set background color of the figure
fig.patch.set_facecolor('black')

for task_idx, (c1, c2) in enumerate(split_mnist_tasks):
    samples, labels = get_balanced_interleaved_samples(train_data, [c1,
c2], num_samples=3)  # 3 per class

    for i in range(6):  # 3 from class 1, 3 from class 2
        ax = axes[task_idx, i]
        ax.imshow(samples[i].squeeze(), cmap="gray")

        # Set title with white text and black background
        ax.set_title(f"Label: {labels[i]}", color='white',
backgroundcolor='black')

        # Set subplot background color to black
        ax.set_facecolor('black')

        ax.axis("off")

plt.tight_layout()
plt.show()

import seaborn as sns
import numpy as np

# Get class labels from dataset
train_labels = np.array(train_data.targets)
test_labels = np.array(test_data.targets)

# Count occurrences of each digit
train_counts = np.bincount(train_labels, minlength=10)
test_counts = np.bincount(test_labels, minlength=10)

# Plot class distribution
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
sns.barplot(x=np.arange(10), y=train_counts, ax=axes[0], palette="Blues")
sns.barplot(x=np.arange(10), y=test_counts, ax=axes[1], palette="Oranges")
```

```
# Set titles and labels
axes[0].set_title("Class Distribution in Training Set")
axes[1].set_title("Class Distribution in Test Set")
for ax in axes:
    ax.set_xlabel("Digit Class")
    ax.set_ylabel("Frequency")

plt.show()

# Flatten images into 1D arrays
all_pixels = torch.cat([img.view(-1) for img, _ in train_data])

# Plot histogram of pixel intensities
plt.figure(figsize=(8, 4))
plt.hist(all_pixels.numpy(), bins=50, color='blue', alpha=0.7)
plt.title("Pixel Intensity Distribution in Training Set")
plt.xlabel("Pixel Value")
plt.ylabel("Frequency")
plt.show()

from sklearn.manifold import TSNE

# Sample a subset of data for speed (taking 2000 random samples)
subset_idx = torch.randperm(len(train_data))[:2000]
subset_images = torch.stack([train_data[i][0].flatten() for i in
subset_idx])
subset_labels = [train_data[i][1] for i in subset_idx]

# Apply t-SNE
tsne = TSNE(n_components=2, random_state=42)
embedded_features = tsne.fit_transform(subset_images.numpy())

# Plot
plt.figure(figsize=(8, 6))
sns.scatterplot(x=embedded_features[:, 0], y=embedded_features[:, 1],
hue=subset_labels, palette="tab10", alpha=0.7)
plt.title("t-SNE Visualization of MNIST Feature Space")
plt.xlabel("t-SNE Component 1")
plt.ylabel("t-SNE Component 2")
plt.legend(title="Digit Class", bbox_to_anchor=(1.05, 1), loc="upper
left")
plt.show()

# Compute variance image per class
var_images = {c: torch.zeros((28, 28)) for c in range(10)}

for img, label in train_data:
    var_images[label] += (img.squeeze() - mean_images[label]) ** 2
```

```
# Normalize by class count
for c in var_images:
    var_images[c] /= class_counts[c]

# Plot the variance images
fig, axes = plt.subplots(2, 5, figsize=(10, 5))
for i, ax in enumerate(axes.flat):
    ax.imshow(var_images[i].numpy(), cmap="inferno")
    ax.set_title(f"Class {i}")
    ax.axis("off")

plt.suptitle("Pixel Variance for Each Class", fontsize=14)
plt.tight_layout()
plt.show()
```

Baseline (Task-IL)

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
import pandas as pd
from collections import defaultdict

# Define Permuted MNIST Dataset
class PermutedMNIST(Dataset):
    def __init__(self, root, train=True, transform=None,
permutations=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.permutations = permutations
        self.train = train

    def __len__(self):
        return len(self.mnist_dataset)

    def __getitem__(self, idx):
        image, label = self.mnist_dataset[idx]
        if self.permutations is not None:
            image = image.view(-1)[self.permutations].view(image.shape)
        if self.transform:
```

```
            image = self.transform(image)
        return image, label

# Setup Permuted MNIST Tasks
num_tasks = 5
input_size = 28 * 28  # Flattened MNIST image
permutations = [torch.randperm(input_size) for _ in range(num_tasks)]

# Load Permuted MNIST Datasets for each task
train_tasks = [PermutedMNIST(root="./data", train=True,
permutations=permutations[i]) for i in range(num_tasks)]
test_tasks = [PermutedMNIST(root="./data", train=False,
permutations=permutations[i]) for i in range(num_tasks)]

# Function to create DataLoaders for each task
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Function to train the model on a specific task
def train_task(model, task_idx, criterion, optimizer, epochs=5):
    train_loader, _ = get_task_data(task_idx)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
```

```
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total

        task_train_loss.append(epoch_loss)
        task_train_acc.append(epoch_acc)

        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

    return task_train_loss, task_train_acc

# Function to evaluate the model on all seen tasks
def evaluate_all_tasks(model, num_tasks):
    accuracies = []

    for i in range(num_tasks):
        _, test_loader = get_task_data(i)

        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
```

```
        accuracy = 100 * correct / total
        accuracies.append(accuracy)
        print(f'Task {i+1} Accuracy: {accuracy:.2f}%')

    return accuracies

# Main function to demonstrate catastrophic forgetting
def demonstrate_catastrophic_forgetting():
    # Hyperparameters
    input_size = 28 * 28  # Flattened MNIST image
    hidden_size = 256
    output_size = 10  # 10 classes for Permuted MNIST
    learning_rate = 0.01
    epochs_per_task = 5

    # Initialize model
    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # To store metrics
    training_history = {
        "task_accuracies": [],  # Performance on each task after
sequential training
        "training_time": [],    # Time taken to train each task
        "learning_curves": {    # Loss and accuracy during training
            "loss": [],
            "accuracy": []
        }
    }

    # To compute forgetting metrics
    initial_accuracies = []  # Accuracy on each task right after learning
it

    # Train on each task sequentially
    for task_idx in range(len(train_tasks)):
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}")
        print(f"{'='*50}")

        # Measure training time
        start_time = time.time()

        # Train on current task
        task_loss, task_acc = train_task(model, task_idx, criterion,
optimizer, epochs=epochs_per_task)

        # Record training time
```

```
        end_time = time.time()
        training_time = end_time - start_time
        training_history["training_time"].append(training_time)

        # Save learning curves
        training_history["learning_curves"]["loss"].extend(task_loss)
        training_history["learning_curves"]["accuracy"].extend(task_acc)

        # Evaluate on all tasks seen so far
        print("\nEvaluating on all tasks seen so far:")
        task_accuracies = evaluate_all_tasks(model, task_idx + 1)

        # Store the accuracy on the current task after learning it
        if task_idx == 0:
            initial_accuracies.append(task_accuracies[0])
        else:

training_history["task_accuracies"].append(task_accuracies.copy())
            initial_accuracies.append(task_accuracies[task_idx])

    # Calculate forgetting metrics
    forgetting_rate = calculate_forgetting_metrics(training_history,
initial_accuracies)

    plot_results(training_history, forgetting_rate, initial_accuracies)

    return training_history, forgetting_rate, initial_accuracies

# Function to calculate forgetting metrics
def calculate_forgetting_metrics(training_history, initial_accuracies):
    forgetting_rate = {}

    # For each task (except the last one since we don't have measurements
after it)
    for task_idx in range(len(initial_accuracies) - 1):
        forgetting = []

        # Calculate forgetting for the task at each subsequent evaluation
point
        for eval_idx, accuracies in
enumerate(training_history["task_accuracies"]):
            if task_idx <= eval_idx:  # We only have measurements for
tasks we've seen
                forgetting.append(initial_accuracies[task_idx] -
accuracies[task_idx])

        forgetting_rate[f"Task {task_idx+1}"] = forgetting

    return forgetting_rate
```

```
def plot_results(training_history, forgetting_rate, initial_accuracies):
    # Set plotting style
    plt.style.use('ggplot')

    epochs_per_task = 5
    total_tasks = len(initial_accuracies)

    # 1. Loss learning curve
    plt.figure(figsize=(12, 5))
    plt.plot(training_history['learning_curves']['loss'], label='Training
Loss', color='blue', marker='o')
    plt.title('Loss Learning Curve')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.show()

    # 2. Accuracy by task plot
    plt.figure(figsize=(12, 5))
    for task_idx in range(total_tasks):
        start_idx = task_idx * epochs_per_task
        end_idx = start_idx + epochs_per_task
        task_acc =
training_history['learning_curves']['accuracy'][start_idx:end_idx]
        task_epochs = range(start_idx + 1, end_idx + 1)
        plt.plot(task_epochs, task_acc, marker='o', label=f'Task
{task_idx+1}')

    plt.title('Accuracy by Task (Epochs)')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.show()

    # 3. Task accuracies during sequential training
    plt.figure(figsize=(12, 5))
    for task_idx in range(total_tasks):
        task_accs = [acc[task_idx] if task_idx < len(acc) else None for
acc in training_history['task_accuracies']]
        plt.plot(range(1, len(task_accs) + 1), task_accs, marker='o',
label=f'Task {task_idx+1}')

    plt.title('Task Accuracies During Sequential Training')
    plt.xlabel('Training Progress (Tasks)')
    plt.ylabel('Accuracy (%)')
    plt.grid(True, alpha=0.3)
```

```
    plt.legend()
    plt.show()

    # 4. Table of accuracies
    accuracy_table = pd.DataFrame(training_history['task_accuracies'],
columns=[f'Task {i+1}' for i in range(total_tasks)])
    accuracy_table.index = [f'After Task {i+1}' for i in range(total_tasks
-1 if len(training_history['task_accuracies']) >0 else 0)]
    print('\nTask Accuracies Table:\n')
    print(accuracy_table.to_string())

    # 5. Plotting forgetting rates
    plt.figure(figsize=(12, 6))
    for task_idx, forgetting in forgetting_rate.items():
        plt.plot(range(1, len(forgetting) + 1), forgetting, marker='o',
label=f'{task_idx} Forgetting')
    plt.title('Forgetting Rates')
    plt.xlabel('Subsequent Tasks')
    plt.ylabel('Forgetting (Accuracy Drop)')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Example usage
if __name__ == "__main__":
    training_history, forgetting_rate, initial_accuracies =
demonstrate_catastrophic_forgetting()
```

L2 Regularization

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
from collections import defaultdict

# Define Permuted MNIST Dataset
class PermutedMNIST(Dataset):
    def __init__(self, root, train=True, transform=None,
permutations=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.permutations = permutations
        self.train = train
```

```
    def __len__(self):
        return len(self.mnist_dataset)

    def __getitem__(self, idx):
        image, label = self.mnist_dataset[idx]
        if self.permutations is not None:
            image = image.view(-1)[self.permutations].view(image.shape)
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Permuted MNIST Tasks
num_tasks = 5
input_size = 28 * 28  # Flattened MNIST image
permutations = [torch.randperm(input_size) for _ in range(num_tasks)]

# Load Permuted MNIST Datasets for each task
train_tasks = [PermutedMNIST(root="./data", train=True,
permutations=permutations[i]) for i in range(num_tasks)]
test_tasks = [PermutedMNIST(root="./data", train=False,
permutations=permutations[i]) for i in range(num_tasks)]

# Function to create DataLoaders for each task
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

from skopt import gp_minimize
from skopt.space import Real
import numpy as np

def find_best_l2_lambda_task1_focus(
    model_class, input_size, hidden_size, output_size,
    num_tasks=5, epochs_per_task=1, n_calls=15, initial_trials=[0.001,
0.01, 0.1, 1.0]
):
    """
    Optimized L2 lambda tuner focused solely on Task 1 final performance
after Task 5.
    Includes exploration safeguards and defaults to a reliable manual
value if needed.
    """
    import torch.nn as nn
    import torch.optim as optim
```

```
    # Define the search space (log scale - note different range from EWC)
    search_space = [Real(0.0001, 10.0, "log-uniform", name="lambda")]

    # Track best lambda and performance so far
    best_lambda = None
    best_task1_performance = 0.0

    # Objective function: Tracks only Task 1 performance at the end
    def objective_function(params):
        nonlocal best_lambda, best_task1_performance
        current_lambda = params[0]
        print(f"\nTrying L2 lambda = {current_lambda:.6f}")

        # Initialize a new model
        model = model_class(input_size, hidden_size, output_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.01)

        # Storage for initial parameters
        initial_params_list = []

        # Train on each task sequentially
        for task_idx in range(num_tasks):
            # Train on current task using L2 regularization
            task_loss, task_acc = train_task(
                model,
                task_idx,
                criterion,
                optimizer,
                initial_params_list=initial_params_list if task_idx > 0
else None,
                l2_lambda=current_lambda,
                epochs=epochs_per_task
            )

            # Store initial parameters after training
            initial_params = store_initial_params(model)
            initial_params_list.append(initial_params)

        # Evaluate on all tasks
        print("\nEvaluating on all tasks:")
        task_accuracies = evaluate_all_tasks(model, num_tasks)

        # Extract only Task 1 performance at the end
        task1_performance = task_accuracies[0]
        print(f"Lambda {current_lambda:.6f} | Task 1 Final Accuracy:
{task1_performance:.2f}%")

        # Track the best lambda value specifically for Task 1
```

```
        if task1_performance > best_task1_performance:
            best_task1_performance = task1_performance
            best_lambda = current_lambda

        # Return **negative Task 1 performance** for minimization
        return -task1_performance

    # Pre-run manual "anchor" lambdas to avoid small-value traps
    for initial_lambda in initial_trials:
        print(f"\n🔍 Testing anchor lambda = {initial_lambda}")
        objective_function([initial_lambda])

    # Run Bayesian optimization (now with good anchors)
    result = gp_minimize(
        objective_function,
        search_space,
        n_calls=n_calls,
        random_state=42,
        verbose=True
    )

    # If Bayesian result fails to outperform manual values, default to the
manual best
    if -result.fun < best_task1_performance:
        print("\n⚠️ Bayesian optimization underperformed manual anchors.
Defaulting to best manual result.")
    else:
        best_lambda = result.x[0]
        best_task1_performance = -result.fun

    print(f"\n☑️ Best L2 lambda found: {best_lambda:.6f} with Task 1
final accuracy: {best_task1_performance:.2f}%")
    return best_lambda


# Call the function to find the optimal L2 lambda
input_size = 28 * 28
hidden_size = 256
output_size = 10

optimal_l2_lambda = find_best_l2_lambda_task1_focus(
    model_class=SimpleNN,
    input_size=input_size,
    hidden_size=hidden_size,
    output_size=output_size,
    num_tasks=5,
    epochs_per_task=1,
    n_calls=10,
    initial_trials=[0.001, 0.01, 0.05, 0.1, 1.0]
```

```
)

print(f"Optimal L2 lambda value for Task 1 performance:
{optimal_l2_lambda}")

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Function to store initial parameters for L2 regularization
def store_initial_params(model):
    initial_params = {}
    for name, param in model.named_parameters():
        initial_params[name] = param.data.clone()
    return initial_params

# Function to train the model on a specific task with L2 regularization
def train_task(model, task_idx, criterion, optimizer,
initial_params_list=None, l2_lambda=0.01, epochs=5):
    train_loader, _ = get_task_data(task_idx)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Add L2 penalty if not the first task
```

```
            if initial_params_list and task_idx > 0:
                l2_loss = 0
                for name, param in model.named_parameters():
                    # Apply L2 regularization to keep parameters close to
their initial values
                    for init_params in initial_params_list:
                        l2_loss += ((param -
init_params[name]).pow(2)).sum()

                # Scale the L2 loss by lambda and add to the task loss
                loss += (l2_lambda / 2) * l2_loss

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total

        task_train_loss.append(epoch_loss)
        task_train_acc.append(epoch_acc)

        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

    return task_train_loss, task_train_acc

# Function to evaluate the model on all seen tasks
def evaluate_all_tasks(model, num_tasks):
    accuracies = []

    for i in range(num_tasks):
        _, test_loader = get_task_data(i)

        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
```

```
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        accuracies.append(accuracy)
        print(f'Task {i+1} Accuracy: {accuracy:.2f}%')

    return accuracies

# Function to calculate forgetting metrics
def calculate_forgetting_metrics(training_history, initial_accuracies):
    forgetting_rate = {}

    # For each task (except the last one since we don't have measurements
after it)
    for task_idx in range(len(initial_accuracies) - 1):
        forgetting = []

        # Calculate forgetting for the task at each subsequent evaluation
point
        for eval_idx, accuracies in
enumerate(training_history["task_accuracies"]):
            if task_idx <= eval_idx:  # We only have measurements for
tasks we've seen
                forgetting.append(initial_accuracies[task_idx] -
accuracies[task_idx])

        forgetting_rate[f"Task {task_idx+1}"] = forgetting

    return forgetting_rate

# Main function to demonstrate L2 regularization for mitigating
catastrophic forgetting
def demonstrate_l2_regularization():
    # Hyperparameters
    input_size = 28 * 28  # Flattened MNIST image
    hidden_size = 256
    output_size = 10  # 10 classes for Permuted MNIST
    learning_rate = 0.01
    epochs_per_task = 5
    l2_lambda = 0.01  # L2 regularization strength

    # Initialize model
    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()

    # Use SGD without weight decay (we'll implement L2 regularization
manually)
```

```
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # Print model configuration
    print(f"Model Configuration:")
    print(f"- SGD with L2 Regularization (lambda={l2_lambda})")
    print(f"- Learning Rate: {learning_rate}")
    print(f"- Hidden Size: {hidden_size}")
    print(f"- Epochs per Task: {epochs_per_task}")

    # To store metrics
    training_history = {
        "task_accuracies": [],  # Performance on each task after
sequential training
        "training_time": [],    # Time taken to train each task
        "learning_curves": {    # Loss and accuracy during training
            "loss": [],
            "accuracy": []
        }
    }

    # To compute forgetting metrics
    initial_accuracies = []  # Accuracy on each task right after learning
it

    # Store initial parameters for each task
    initial_params_list = []

    # Train on each task sequentially
    for task_idx in range(len(train_tasks)):
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}")
        print(f"{'='*50}")

        # Measure training time
        start_time = time.time()

        # Train on current task using L2 regularization if not the first
task
        task_loss, task_acc = train_task(
            model,
            task_idx,
            criterion,
            optimizer,
            initial_params_list=initial_params_list if task_idx > 0 else
None,
            l2_lambda=l2_lambda,
            epochs=epochs_per_task
        )
```

```
        # Record training time
        end_time = time.time()
        training_time = end_time - start_time
        training_history["training_time"].append(training_time)

        # Save learning curves
        training_history["learning_curves"]["loss"].extend(task_loss)
        training_history["learning_curves"]["accuracy"].extend(task_acc)

        # After training on this task, store the current parameters
        initial_params = store_initial_params(model)
        initial_params_list.append(initial_params)

        # Evaluate on all tasks seen so far
        print("\nEvaluating on all tasks seen so far:")
        task_accuracies = evaluate_all_tasks(model, task_idx + 1)

        # Store the accuracy on the current task after learning it
        if task_idx == 0:
            initial_accuracies.append(task_accuracies[0])
        else:

training_history["task_accuracies"].append(task_accuracies.copy())
            initial_accuracies.append(task_accuracies[task_idx])

    # Calculate forgetting metrics
    forgetting_rate = calculate_forgetting_metrics(training_history,
initial_accuracies)

    return training_history, forgetting_rate, initial_accuracies

# Example usage
if __name__ == "__main__":
    training_history, forgetting_rate, initial_accuracies =
demonstrate_l2_regularization()
```

EWC

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
from collections import defaultdict
```

```
# Define Permuted MNIST Dataset
class PermutedMNIST(Dataset):
    def __init__(self, root, train=True, transform=None,
permutations=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.permutations = permutations
        self.train = train

    def __len__(self):
        return len(self.mnist_dataset)

    def __getitem__(self, idx):
        image, label = self.mnist_dataset[idx]
        if self.permutations is not None:
            image = image.view(-1)[self.permutations].view(image.shape)
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Permuted MNIST Tasks
num_tasks = 5
input_size = 28 * 28  # Flattened MNIST image
permutations = [torch.randperm(input_size) for _ in range(num_tasks)]

# Load Permuted MNIST Datasets for each task
train_tasks = [PermutedMNIST(root="./data", train=True,
permutations=permutations[i]) for i in range(num_tasks)]
test_tasks = [PermutedMNIST(root="./data", train=False,
permutations=permutations[i]) for i in range(num_tasks)]

# Function to create DataLoaders for each task
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

from skopt import gp_minimize
from skopt.space import Real
import numpy as np

def find_best_ewc_lambda_task1_focus(
    model_class, input_size, hidden_size, output_size,
    num_tasks=5, epochs_per_task=1, n_calls=15, initial_trials=[10, 40,
100, 500]
):
```

```
    """
    Optimized EWC lambda tuner focused solely on Task 1 final performance
after Task 5.
    Includes exploration safeguards and defaults to a reliable manual
value if needed.
    """
    import torch.nn as nn
    import torch.optim as optim

    # Define the search space (log scale)
    search_space = [Real(1.0, 1000.0, "log-uniform", name="lambda")]

    # Track best lambda and performance so far
    best_lambda = None
    best_task1_performance = 0.0

    # Objective function: Tracks only Task 1 performance at the end
    def objective_function(params):
        nonlocal best_lambda, best_task1_performance
        current_lambda = params[0]
        print(f"\nTrying EWC lambda = {current_lambda:.2f}")

        # Initialize a new model
        model = model_class(input_size, hidden_size, output_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.01)

        # Storage for Fisher matrices and optimal parameters
        fisher_matrices = []
        optimal_params = []

        # Train on each task sequentially
        for task_idx in range(num_tasks):
            # Train on current task using EWC
            task_loss, task_acc = train_task(
                model,
                task_idx,
                criterion,
                optimizer,
                fisher_matrices=fisher_matrices if task_idx > 0 else None,
                optimal_params=optimal_params if task_idx > 0 else None,
                ewc_lambda=current_lambda,
                epochs=epochs_per_task
            )

            # Compute Fisher Information Matrix
            train_loader, _ = get_task_data(task_idx)
            fisher = compute_fisher_matrix(model, train_loader, criterion)
            fisher_matrices.append(fisher)
```

```
            # Save optimal parameters
            optimal_param_dict = {name: param.data.clone() for name, param
in model.named_parameters()}
            optimal_params.append(optimal_param_dict)

        # Evaluate on all tasks
        print("\nEvaluating on all tasks:")
        task_accuracies = evaluate_all_tasks(model, num_tasks)

        # Extract only Task 1 performance at the end
        task1_performance = task_accuracies[0]
        print(f"Lambda {current_lambda:.2f} | Task 1 Final Accuracy:
{task1_performance:.2f}%")

        # Track the best lambda value specifically for Task 1
        if task1_performance > best_task1_performance:
            best_task1_performance = task1_performance
            best_lambda = current_lambda

        # Return **negative Task 1 performance** for minimization
        return -task1_performance

    # Pre-run manual "anchor" lambdas to avoid small-value traps
    for initial_lambda in initial_trials:
        print(f"\n🔍 Testing anchor lambda = {initial_lambda}")
        objective_function([initial_lambda])

    # Run Bayesian optimization (now with good anchors)
    result = gp_minimize(
        objective_function,
        search_space,
        n_calls=n_calls,
        random_state=42,
        verbose=True
    )

    # If Bayesian result fails to outperform manual values, default to the
manual best
    if -result.fun < best_task1_performance:
        print("\n⚠ Bayesian optimization underperformed manual anchors.
Defaulting to best manual result.")
    else:
        best_lambda = result.x[0]
        best_task1_performance = -result.fun

    print(f"\n☑ Best EWC lambda found: {best_lambda:.2f} with Task 1
final accuracy: {best_task1_performance:.2f}%")
    return best_lambda
```

```
# Call the function to find the optimal EWC lambda
input_size = 28 * 28
hidden_size = 256
output_size = 10

optimal_ewc_lambda = find_best_ewc_lambda_task1_focus(
    model_class=SimpleNN,
    input_size=input_size,
    hidden_size=hidden_size,
    output_size=output_size,
    num_tasks=5,
    epochs_per_task=1,
    n_calls=10,
    initial_trials=[10, 40, 62, 100, 500]
)

print(f"Optimal EWC lambda value for Task 1 performance:
{optimal_ewc_lambda}")

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Function to compute the Fisher Information Matrix for EWC
def compute_fisher_matrix(model, task_loader, criterion):
    model.eval()
    fisher = {}

    # Initialize Fisher matrix (one for each parameter)
    for name, param in model.named_parameters():
        fisher[name] = torch.zeros_like(param)

    # Compute Fisher Information Matrix
    for inputs, labels in task_loader:
        model.zero_grad()
```

```
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()

        # Accumulate squared gradients
        for name, param in model.named_parameters():
            if param.grad is not None:
                fisher[name] += param.grad.pow(2) /
len(task_loader.dataset)

    return fisher

# Function to train the model on a specific task with EWC
def train_task(model, task_idx, criterion, optimizer,
fisher_matrices=None, optimal_params=None, ewc_lambda=40, epochs=5):
    train_loader, _ = get_task_data(task_idx)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Add EWC penalty if not the first task
            if fisher_matrices and optimal_params:
                ewc_loss = 0
                for task_id, (fisher, optim_params) in
enumerate(zip(fisher_matrices, optimal_params)):
                    for name, param in model.named_parameters():
                        # Compute the EWC penalty: importance * squared
distance
                        ewc_loss += (fisher[name] * (param -
optim_params[name]).pow(2)).sum()

                # Scale the EWC loss by lambda and add to the task loss
                loss += (ewc_lambda / 2) * ewc_loss

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
```

```
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total

        task_train_loss.append(epoch_loss)
        task_train_acc.append(epoch_acc)

        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

    return task_train_loss, task_train_acc

# Function to evaluate the model on all seen tasks
def evaluate_all_tasks(model, num_tasks):
    accuracies = []

    for i in range(num_tasks):
        _, test_loader = get_task_data(i)

        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        accuracies.append(accuracy)
        print(f'Task {i+1} Accuracy: {accuracy:.2f}%')

    return accuracies

# Function to calculate forgetting metrics
def calculate_forgetting_metrics(training_history, initial_accuracies):
    forgetting_rate = {}
```

```
    # For each task (except the last one since we don't have measurements
after it)
    for task_idx in range(len(initial_accuracies) - 1):
        forgetting = []

        # Calculate forgetting for the task at each subsequent evaluation
point
        for eval_idx, accuracies in
enumerate(training_history["task_accuracies"]):
            if task_idx <= eval_idx:  # We only have measurements for
tasks we've seen
                forgetting.append(initial_accuracies[task_idx] -
accuracies[task_idx])

        forgetting_rate[f"Task {task_idx+1}"] = forgetting

    return forgetting_rate

# Main function to demonstrate EWC for mitigating catastrophic forgetting
def demonstrate_ewc():
    # Hyperparameters
    input_size = 28 * 28  # Flattened MNIST image
    hidden_size = 256
    output_size = 10  # 10 classes for Permuted MNIST
    learning_rate = 0.01
    epochs_per_task = 5
    ewc_lambda = 22  # EWC importance weighting for previous tasks

    # Initialize model
    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()

    # Use SGD without weight decay (EWC replaces L2 regularization)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # Print model configuration
    print(f"Model Configuration:")
    print(f"- SGD with EWC Regularization (lambda={ewc_lambda})")
    print(f"- Learning Rate: {learning_rate}")
    print(f"- Hidden Size: {hidden_size}")
    print(f"- Epochs per Task: {epochs_per_task}")

    # To store metrics
    training_history = {
        "task_accuracies": [],  # Performance on each task after
sequential training
        "training_time": [],    # Time taken to train each task
        "learning_curves": {    # Loss and accuracy during training
            "loss": [],
```

```
            "accuracy": []
        }
    }

    # To compute forgetting metrics
    initial_accuracies = []  # Accuracy on each task right after learning
it

    # Store Fisher matrices and optimal parameters for each task
    fisher_matrices = []
    optimal_params = []

    # Train on each task sequentially
    for task_idx in range(len(train_tasks)): #Changed split_mnist_tasks to
train_tasks
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}") #Remove task details from
print
        print(f"{'='*50}")

        # Measure training time
        start_time = time.time()

        # Train on current task using EWC if not the first task
        task_loss, task_acc = train_task(
            model,
            task_idx,
            criterion,
            optimizer,
            fisher_matrices=fisher_matrices if task_idx > 0 else None,
            optimal_params=optimal_params if task_idx > 0 else None,
            ewc_lambda=ewc_lambda,
            epochs=epochs_per_task
        )

        # Record training time
        end_time = time.time()
        training_time = end_time - start_time
        training_history["training_time"].append(training_time)

        # Save learning curves
        training_history["learning_curves"]["loss"].extend(task_loss)
        training_history["learning_curves"]["accuracy"].extend(task_acc)

        # After training on this task, compute Fisher Information Matrix
        train_loader, _ = get_task_data(task_idx)
        fisher = compute_fisher_matrix(model, train_loader, criterion)
        fisher_matrices.append(fisher)
```

```
        # Save optimal parameters for this task
        optimal_param_dict = {}
        for name, param in model.named_parameters():
            optimal_param_dict[name] = param.data.clone()
        optimal_params.append(optimal_param_dict)

        # Evaluate on all tasks seen so far
        print("\nEvaluating on all tasks seen so far:")
        task_accuracies = evaluate_all_tasks(model, task_idx + 1)

        # Store the accuracy on the current task after learning it
        if task_idx == 0:
            initial_accuracies.append(task_accuracies[0])
        else:

training_history["task_accuracies"].append(task_accuracies.copy())
            initial_accuracies.append(task_accuracies[task_idx])

    # Calculate forgetting metrics
    forgetting_rate = calculate_forgetting_metrics(training_history,
initial_accuracies)

    return training_history, forgetting_rate, initial_accuracies

# Example usage
if __name__ == "__main__":
    training_history, forgetting_rate, initial_accuracies =
demonstrate_ewc()
```

SI

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import Dataset, DataLoader, Subset
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
from collections import defaultdict

# Define Permuted MNIST Dataset
class PermutedMNIST(Dataset):
    def __init__(self, root, train=True, transform=None,
permutations=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
```

```
        self.permutations = permutations
        self.train = train

    def __len__(self):
        return len(self.mnist_dataset)

    def __getitem__(self, idx):
        image, label = self.mnist_dataset[idx]
        if self.permutations is not None:
            image = image.view(-1)[self.permutations].view(image.shape)
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Permuted MNIST Tasks
num_tasks = 5
input_size = 28 * 28  # Flattened MNIST image
permutations = [torch.randperm(input_size) for _ in range(num_tasks)]

# Load Permuted MNIST Datasets for each task
train_tasks = [PermutedMNIST(root="./data", train=True,
permutations=permutations[i]) for i in range(num_tasks)]
test_tasks = [PermutedMNIST(root="./data", train=False,
permutations=permutations[i]) for i in range(num_tasks)]

# Function to create DataLoaders for each task
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

from skopt import gp_minimize
from skopt.space import Real
import numpy as np

def find_optimal_si_lambda_task1_focus(
    model_class, input_size, hidden_size, output_size,
    num_tasks=5, epochs_per_task=1, n_calls=10
):
    """
    Find an optimal lambda value for Synaptic Intelligence using Bayesian
optimization,
    focused exclusively on maximizing Task 1 performance after Task 5.
    """
    import torch.nn as nn
    import torch.optim as optim
```

```
    # Define the search space (log scale)
    search_space = [Real(0.1, 100.0, "log-uniform", name="lambda")]

    # Objective function: Only cares about Task 1 performance at the very
end
    def objective_function(params):
        current_lambda = params[0]
        print(f"\nTrying SI lambda = {current_lambda:.2f}")

        # Initialize a new model
        model = model_class(input_size, hidden_size, output_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=0.01)

        # Storage for SI parameters
        omega_sum = None
        initial_params = {}

        # Save initial model parameters
        for name, param in model.named_parameters():
            initial_params[name] = param.data.clone()

        # Train on each task sequentially
        for task_idx in range(num_tasks):
            # Train on current task using SI
            task_loss, task_acc, omega_curr = train_task(
                model,
                task_idx,
                criterion,
                optimizer,
                omega_sum=omega_sum,
                initial_params=initial_params if task_idx > 0 else None,
                si_lambda=current_lambda,
                epochs=epochs_per_task
            )

            # Update accumulated importance weights (omega_sum)
            if omega_sum is None:
                omega_sum = omega_curr
            else:
                for name in omega_sum:
                    omega_sum[name] += omega_curr[name]

            # Save new initial parameters for the next task
            for name, param in model.named_parameters():
                initial_params[name] = param.data.clone()

        # Evaluate on all tasks
        print("\nEvaluating on all tasks:")
```

```
        task_accuracies = evaluate_all_tasks(model, num_tasks)

        # Extract **only Task 1 accuracy** at the end
        task1_performance = task_accuracies[0]
        print(f"Lambda {current_lambda:.2f} | Task 1 Final Accuracy:
{task1_performance:.2f}%")

        # Return **negative Task 1 performance** for minimization
        return -task1_performance

    # Run Bayesian optimization
    result = gp_minimize(
        objective_function,
        search_space,
        n_calls=n_calls,
        random_state=42,
        verbose=True
    )

    # Get the best lambda specifically for Task 1 performance
    best_lambda = result.x[0]
    best_performance = -result.fun

    print(f"\nBest SI lambda found: {best_lambda:.2f} with Task 1 final
accuracy: {best_performance:.2f}%")
    return best_lambda


# Call the function to find the optimal SI lambda
input_size = 28 * 28
hidden_size = 256
output_size = 10

optimal_si_lambda = find_optimal_si_lambda_task1_focus(
    model_class=SimpleNN,
    input_size=input_size,
    hidden_size=hidden_size,
    output_size=output_size,
    num_tasks=5,
    epochs_per_task=1,  # Keep it efficient
    n_calls=10  # Minimum search effort
)

print(f"Optimal SI lambda value for Task 1 performance:
{optimal_si_lambda}")

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
```

```
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Function to initialize synaptic intelligence trackers
def initialize_si_trackers(model):
    # Initialize parameter importance (omega)
    omega = {}
    # Initialize parameter change accumulator (path integral)
    path_integral = {}
    # Initialize old parameter values
    old_params = {}

    for name, param in model.named_parameters():
        omega[name] = torch.zeros_like(param.data)
        path_integral[name] = torch.zeros_like(param.data)
        old_params[name] = param.data.clone()

    return omega, path_integral, old_params

# Function to update synaptic intelligence trackers
def update_si_trackers(model, path_integral, old_params):
    for name, param in model.named_parameters():
        # Compute parameter change
        delta = param.data - old_params[name]
        # Update path integral with parameter change * gradient
        if param.grad is not None:
            path_integral[name] += -param.grad * delta
        # Update old parameter values
        old_params[name] = param.data.clone()

    return path_integral, old_params

# Function to compute the synaptic intelligence omega (importance)
def compute_omega(model, path_integral, old_params, epsilon=0.1):
    omega_new = {}

    for name, param in model.named_parameters():
        # Compute parameter change
        delta = param.data - old_params[name]
```

```
        # Compute importance (omega) based on path integral and parameter
change
        # Add epsilon to avoid division by zero
        delta_norm = torch.norm(delta)
        if delta_norm > 0:
            omega_new[name] = path_integral[name] / (delta.pow(2) +
epsilon)
        else:
            omega_new[name] = torch.zeros_like(param.data)

    return omega_new

# Function to compute the SI penalty
def si_penalty(model, omega_sum, initial_params):
    penalty = 0
    for name, param in model.named_parameters():
        # Compute squared parameter change from old value
        delta = (param.data - initial_params[name]).pow(2)
        # Compute the penalty based on importance and parameter change
        penalty += (omega_sum[name] * delta).sum()

    return penalty

# Function to train the model on a specific task with Synaptic
Intelligence
def train_task(model, task_idx, criterion, optimizer, omega_sum=None,
initial_params=None, si_lambda=1.0, epochs=5):
    train_loader, _ = get_task_data(task_idx)

    # Initialize SI trackers for current task
    omega_curr, path_integral, old_params = initialize_si_trackers(model)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Add SI penalty if not the first task
            if omega_sum is not None and initial_params is not None:
```

```
                si_loss = si_penalty(model, omega_sum, initial_params)
                loss += si_lambda * si_loss

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Update SI trackers
            path_integral, old_params = update_si_trackers(model,
path_integral, old_params)

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total

        task_train_loss.append(epoch_loss)
        task_train_acc.append(epoch_acc)

        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

    # Compute importance (omega) after training on this task
    omega_curr = compute_omega(model, path_integral, old_params)

    return task_train_loss, task_train_acc, omega_curr

# Function to evaluate the model on all seen tasks
def evaluate_all_tasks(model, num_tasks):
    accuracies = []

    for i in range(num_tasks):
        _, test_loader = get_task_data(i)

        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
```

```
                correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        accuracies.append(accuracy)
        print(f'Task {i+1} Accuracy: {accuracy:.2f}%')

    return accuracies

# Function to calculate forgetting metrics
def calculate_forgetting_metrics(training_history, initial_accuracies):
    forgetting_rate = {}

    # For each task (except the last one since we don't have measurements
after it)
    for task_idx in range(len(initial_accuracies) - 1):
        forgetting = []

        # Calculate forgetting for the task at each subsequent evaluation
point
        for eval_idx, accuracies in
enumerate(training_history["task_accuracies"]):
            if task_idx <= eval_idx:  # We only have measurements for
tasks we've seen
                forgetting.append(initial_accuracies[task_idx] -
accuracies[task_idx])

        forgetting_rate[f"Task {task_idx+1}"] = forgetting

    return forgetting_rate

# Main function to demonstrate Synaptic Intelligence for mitigating
catastrophic forgetting
def demonstrate_synaptic_intelligence():
    # Hyperparameters
    input_size = 28 * 28  # Flattened MNIST image
    hidden_size = 256
    output_size = 10  # 10 classes for Permuted MNIST
    learning_rate = 0.01
    epochs_per_task = 5
    si_lambda = 0.2  # Synaptic Intelligence regularization strength

    # Initialize model
    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()

    # Use SGD without weight decay (SI replaces L2 regularization)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # Print model configuration
```

```
    print(f"Model Configuration:")
    print(f"- SGD with Synaptic Intelligence (lambda={si_lambda})")
    print(f"- Learning Rate: {learning_rate}")
    print(f"- Hidden Size: {hidden_size}")
    print(f"- Epochs per Task: {epochs_per_task}")

    # To store metrics
    training_history = {
        "task_accuracies": [],  # Performance on each task after
sequential training
        "training_time": [],    # Time taken to train each task
        "learning_curves": {    # Loss and accuracy during training
            "loss": [],
            "accuracy": []
        }
    }

    # To compute forgetting metrics
    initial_accuracies = []  # Accuracy on each task right after learning
it

    # Store accumulated importance weights (omega) and initial parameters
    omega_sum = None
    initial_params = {}

    # Save initial model parameters
    for name, param in model.named_parameters():
        initial_params[name] = param.data.clone()

    # Train on each task sequentially
    for task_idx in range(len(train_tasks)): # Change split_mnist_tasks to
train_tasks
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}") #Remove task details from
print
        print(f"{'='*50}")

        # Measure training time
        start_time = time.time()

        # Train on current task using SI if not the first task
        task_loss, task_acc, omega_curr = train_task(
            model,
            task_idx,
            criterion,
            optimizer,
            omega_sum=omega_sum,
            initial_params=initial_params if task_idx > 0 else None,
            si_lambda=si_lambda,
```

```
            epochs=epochs_per_task
        )

        # Record training time
        end_time = time.time()
        training_time = end_time - start_time
        training_history["training_time"].append(training_time)

        # Save learning curves
        training_history["learning_curves"]["loss"].extend(task_loss)
        training_history["learning_curves"]["accuracy"].extend(task_acc)

        # Update accumulated importance weights (omega_sum)
        if omega_sum is None:
            omega_sum = omega_curr
        else:
            for name in omega_sum:
                omega_sum[name] += omega_curr[name]

        # Save new initial parameters for the next task
        for name, param in model.named_parameters():
            initial_params[name] = param.data.clone()

        # Evaluate on all tasks seen so far
        print("\nEvaluating on all tasks seen so far:")
        task_accuracies = evaluate_all_tasks(model, task_idx + 1)

        # Store the accuracy on the current task after learning it
        if task_idx == 0:
            initial_accuracies.append(task_accuracies[0])
        else:

training_history["task_accuracies"].append(task_accuracies.copy())
            initial_accuracies.append(task_accuracies[task_idx])

    # Calculate forgetting metrics
    forgetting_rate = calculate_forgetting_metrics(training_history,
initial_accuracies)

    return training_history, forgetting_rate, initial_accuracies

# Example usage (you can call this function in your main script)
if __name__ == "__main__":
    training_history, forgetting_rate, initial_accuracies =
demonstrate_synaptic_intelligence()
```

PackNet

```
import torch
```

```
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import torchvision
from torchvision import transforms
import time
import numpy as np

# Define Permuted MNIST Dataset
class PermutedMNIST(Dataset):
    def __init__(self, root, train=True, transform=None,
permutations=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.permutations = permutations
        self.train = train

    def __len__(self):
        return len(self.mnist_dataset)

    def __getitem__(self, idx):
        image, label = self.mnist_dataset[idx]
        if self.permutations is not None:
            image = image.view(-1)[self.permutations].view(image.shape)
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Permuted MNIST Tasks
num_tasks = 5
input_size = 28 * 28  # Flattened MNIST image
permutations = [torch.randperm(input_size) for _ in range(num_tasks)]

# Load Permuted MNIST Datasets for each task
train_tasks = [PermutedMNIST(root="./data", train=True,
permutations=permutations[i]) for i in range(num_tasks)]
test_tasks = [PermutedMNIST(root="./data", train=False,
permutations=permutations[i]) for i in range(num_tasks)]

# Function to create DataLoaders for each task
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

# Define a simple neural network
```

```
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

class PackNet:
    def __init__(self, model, prune_percent=0.1):
        self.model = model
        self.prune_percent = prune_percent
        self.masks = {}  # To store task-specific masks
        self.task_masks = {}  # To store masks for each task
        self.original_weights = {}  # To store original weights for
resetting

    def prune_weights(self, task_idx):
        """Prune a percentage of weights to free up capacity for the next
task."""
        for name, param in self.model.named_parameters():
            if 'weight' in name:  # Only prune weights, not biases
                # Store original weights for resetting later
                self.original_weights[name] = param.data.clone()

                # Prune based on weight magnitude
                weights = param.data.cpu().numpy()
                flat_weights = np.abs(weights.flatten())
                threshold = np.percentile(flat_weights, self.prune_percent
* 100)
                mask = (np.abs(weights) > threshold).astype(float)
                self.masks[name] = torch.tensor(mask,
dtype=torch.float32).to(param.device)
                self.task_masks[f"task_{task_idx}_{name}"] =
self.masks[name].clone()

    def apply_masks(self, task_idx):
        """Apply masks to freeze pruned weights for the current task."""
        for name, param in self.model.named_parameters():
            if name in self.masks:
                param.data *= self.masks[name]
```

```
    def freeze_weights(self):
        """Freeze the remaining weights for the current task."""
        for name, param in self.model.named_parameters():
            if name in self.masks:
                param.requires_grad = False

    def load_task_masks(self, task_idx):
        """Load task-specific masks for inference."""
        for name, param in self.model.named_parameters():
            if f"task_{task_idx}_{name}" in self.task_masks:
                # Reset weights to original state before applying the mask
                param.data = self.original_weights[name].clone()
                # Apply task-specific mask
                param.data *= self.task_masks[f"task_{task_idx}_{name}"]

# Function to train the model on a specific task with PackNet
def train_task_packnet(model, packnet, task_idx, criterion, optimizer,
epochs=5):
    train_loader, _ = get_task_data(task_idx)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total
```

```
            task_train_loss.append(epoch_loss)
            task_train_acc.append(epoch_acc)

            print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

        # Prune and freeze weights after training the task
        packnet.prune_weights(task_idx)
        packnet.apply_masks(task_idx)
        packnet.freeze_weights()

        return task_train_loss, task_train_acc

# Function to evaluate the model on all seen tasks with PackNet
def evaluate_all_tasks_packnet(model, packnet, num_tasks):
    accuracies = []

    for i in range(num_tasks):
        _, test_loader = get_task_data(i)

        # Load task-specific masks for inference
        packnet.load_task_masks(i)

        model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        accuracies.append(accuracy)
        print(f'Task {i+1} Accuracy: {accuracy:.2f}%')

    return accuracies

# Main function to demonstrate PackNet
def demonstrate_packnet():
    # Hyperparameters
    input_size = 28 * 28  # Flattened MNIST image
    hidden_size = 256
    output_size = 10  # 10 classes for Permuted MNIST
    learning_rate = 0.01
    epochs_per_task = 5
```

```
    prune_percent = 0.05  # Reduced pruning percentage

    # Initialize model and PackNet
    model = SimpleNN(input_size, hidden_size, output_size)
    packnet = PackNet(model, prune_percent=prune_percent)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # To store metrics
    training_history = {
        "task_accuracies": [],  # Performance on each task after
sequential training
        "training_time": [],    # Time taken to train each task
        "learning_curves": {    # Loss and accuracy during training
            "loss": [],
            "accuracy": []
        }
    }

    # To compute forgetting metrics
    initial_accuracies = []  # Accuracy on each task right after learning
it

    # Train on each task sequentially
    for task_idx in range(len(train_tasks)):
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}")
        print(f"{'='*50}")

        # Measure training time
        start_time = time.time()

        # Train on current task with PackNet
        task_loss, task_acc = train_task_packnet(model, packnet, task_idx,
criterion, optimizer, epochs=epochs_per_task)

        # Record training time
        end_time = time.time()
        training_time = end_time - start_time
        training_history["training_time"].append(training_time)

        # Save learning curves
        training_history["learning_curves"]["loss"].extend(task_loss)
        training_history["learning_curves"]["accuracy"].extend(task_acc)

        # Evaluate on all tasks seen so far
        print("\nEvaluating on all tasks seen so far:")
        task_accuracies = evaluate_all_tasks_packnet(model, packnet,
task_idx + 1)
```

```
        # Store the accuracy on the current task after learning it
        if task_idx == 0:
            initial_accuracies.append(task_accuracies[0])
        else:

training_history["task_accuracies"].append(task_accuracies.copy())
            initial_accuracies.append(task_accuracies[task_idx])

    # Calculate forgetting metrics
    forgetting_rate = calculate_forgetting_metrics(training_history,
initial_accuracies)

    return training_history, forgetting_rate, initial_accuracies

# Function to calculate forgetting metrics
def calculate_forgetting_metrics(training_history, initial_accuracies):
    forgetting_rate = {}

    # For each task (except the last one since we don't have measurements
after it)
    for task_idx in range(len(initial_accuracies) - 1):
        forgetting = []

        # Calculate forgetting for the task at each subsequent evaluation
point
        for eval_idx, accuracies in
enumerate(training_history["task_accuracies"]):
            if task_idx <= eval_idx:  # We only have measurements for
tasks we've seen
                forgetting.append(initial_accuracies[task_idx] -
accuracies[task_idx])

        forgetting_rate[f"Task {task_idx+1}"] = forgetting

    return forgetting_rate

# Run the PackNet demonstration
if __name__ == "__main__":
    training_history, forgetting_rate, initial_accuracies =
demonstrate_packnet()

Baseline (Class-IL)
import torch
import torch.nn as nn
import torch.optim as optim
import time
import random
import numpy as np
```

```
from torch.utils.data import Dataset, DataLoader, ConcatDataset
import torchvision
from torchvision import transforms

# Define Class Incremental MNIST Dataset
class ClassIncrementalMNIST(Dataset):
    def __init__(self, root, train=True, transform=None, classes=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.classes = classes
        self.train = train
        # Filter data to include only the specified classes
        self.data = []
        self.targets = []
        for image, label in self.mnist_dataset:
            if label in self.classes:
                self.data.append(image)
                self.targets.append(label)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        image, label = self.data[idx], self.targets[idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Class Incremental MNIST Tasks
num_tasks = 5
classes_per_task = 2

# Divide classes into tasks
class_splits = [list(range(i * classes_per_task, (i + 1) *
classes_per_task)) for i in range(num_tasks)]

# Load datasets for each task
train_tasks = [ClassIncrementalMNIST(root="./data", train=True,
classes=class_splits[i]) for i in range(num_tasks)]
test_tasks = [ClassIncrementalMNIST(root="./data", train=False,
classes=class_splits[i]) for i in range(num_tasks)]

# Function to create DataLoaders
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
```

```
    return train_loader, test_loader

# Define SimpleNN model
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Train function with improved monitoring
def train_task(model, task_idx, criterion, optimizer, epochs=5):
    train_loader, _ = get_task_data(task_idx)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total

        task_train_loss.append(epoch_loss)
        task_train_acc.append(epoch_acc)

        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

    return task_train_loss, task_train_acc

# Evaluate function for cumulative classes
def evaluate_cumulative_classes(model, num_tasks):
    combined_test_set = ConcatDataset(test_tasks[:num_tasks])
    test_loader = DataLoader(combined_test_set, batch_size=64,
shuffle=False)

    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy

# Baseline continual learning demonstration
def demonstrate_baseline():
    input_size = 28 * 28
    hidden_size = 256
    learning_rate = 0.01
    epochs_per_task = 5

    # Start with output size for first task
    output_size = 10  # All MNIST digits (0-9)

    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    accuracies = []

    for task_idx in range(len(class_splits)):
        print(f"\n{'='*50}")
```

```
        print(f"Training on Task {task_idx+1}: Classes
{class_splits[task_idx]}")
        print(f"{'='*50}")

        train_loss, train_acc = train_task(model, task_idx, criterion,
optimizer, epochs_per_task)

        accuracy = evaluate_cumulative_classes(model, task_idx + 1)
        accuracies.append(accuracy)

        print(f"\nModel Accuracy after Task {task_idx + 1}:
{accuracy:.2f}%")

    return accuracies

# Example usage
if __name__ == "__main__":
    accuracies = demonstrate_baseline()
```

Naïve Rehearsal

```
import torch
import torch.nn as nn
import torch.optim as optim
import time
import random
import numpy as np
from torch.utils.data import Dataset, DataLoader, ConcatDataset
import torchvision
from torchvision import transforms

# Define Class Incremental MNIST Dataset
class ClassIncrementalMNIST(Dataset):
    def __init__(self, root, train=True, transform=None, classes=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.classes = classes
        self.train = train
        # Filter data to include only the specified classes
        self.data = []
        self.targets = []
        for image, label in self.mnist_dataset:
            if label in self.classes:
                self.data.append(image)
                self.targets.append(label)

    def __len__(self):
        return len(self.data)
```

```
    def __getitem__(self, idx):
        image, label = self.data[idx], self.targets[idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Class Incremental MNIST Tasks
num_tasks = 5
classes_per_task = 2

# Divide classes into tasks
class_splits = [list(range(i * classes_per_task, (i + 1) *
classes_per_task)) for i in range(num_tasks)]

# Load datasets for each task
train_tasks = [ClassIncrementalMNIST(root="./data", train=True,
classes=class_splits[i]) for i in range(num_tasks)]
test_tasks = [ClassIncrementalMNIST(root="./data", train=False,
classes=class_splits[i]) for i in range(num_tasks)]

# Function to create DataLoaders
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

# Define SimpleNN model
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Train function with naive rehearsal
def train_task_with_replay(model, task_idx, criterion, optimizer,
replay_data, replay_labels, epochs=5):
    train_loader, _ = get_task_data(task_idx)
```

```
    # Combine current task data with replay data
    combined_data = torch.utils.data.TensorDataset(
        torch.cat([torch.stack([x for x, _ in train_loader.dataset]),
replay_data]),
        torch.cat([torch.tensor([y for _, y in train_loader.dataset],
dtype=torch.long), replay_labels])
    )
    combined_loader = DataLoader(combined_data, batch_size=64,
shuffle=True)

    # For collecting metrics
    task_train_loss = []
    task_train_acc = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in combined_loader:
            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(combined_loader)
        epoch_acc = 100 * correct / total

        task_train_loss.append(epoch_loss)
        task_train_acc.append(epoch_acc)

        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

    return task_train_loss, task_train_acc

# Evaluate function for cumulative classes
```

```
def evaluate_cumulative_classes(model, num_tasks):
    combined_test_set = ConcatDataset(test_tasks[:num_tasks])
    test_loader = DataLoader(combined_test_set, batch_size=64,
shuffle=False)

    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy

# Naive Rehearsal continual learning demonstration
def demonstrate_naive_rehearsal():
    input_size = 28 * 28
    hidden_size = 256
    learning_rate = 0.01
    epochs_per_task = 5

    # Start with output size for first task
    output_size = 10  # All MNIST digits (0-9)

    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    accuracies = []
    replay_data = torch.tensor([])
    replay_labels = torch.tensor([], dtype=torch.long)  # Ensure
replay_labels is of type torch.long

    for task_idx in range(len(class_splits)):
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}: Classes
{class_splits[task_idx]}")
        print(f"{'='*50}")

        # Train on current task with replay
        train_loss, train_acc = train_task_with_replay(model, task_idx,
criterion, optimizer, replay_data, replay_labels, epochs_per_task)

        # Evaluate on cumulative classes
```

```
        accuracy = evaluate_cumulative_classes(model, task_idx + 1)
        accuracies.append(accuracy)

        print(f"\nModel Accuracy after Task {task_idx + 1}:
{accuracy:.2f}%")

        # Store a subset of current task data for replay
        current_task_data = torch.stack([x for x, _ in
train_tasks[task_idx]])
        current_task_labels = torch.tensor([y for _, y in
train_tasks[task_idx]], dtype=torch.long)

        # Randomly select a subset of data for replay
        replay_size = 100  # Number of samples to store for replay
        indices = torch.randperm(len(current_task_data))[:replay_size]
        replay_data = torch.cat([replay_data, current_task_data[indices]])
        replay_labels = torch.cat([replay_labels,
current_task_labels[indices]])

    return accuracies

# Run the demonstration
if __name__ == "__main__":
    naive_rehearsal_accuracies = demonstrate_naive_rehearsal()

Experience Replay
import torch
import torch.nn as nn
import torch.optim as optim
import time
import random
import numpy as np
from torch.utils.data import Dataset, DataLoader, ConcatDataset
import torchvision
from torchvision import transforms

# Define Class Incremental MNIST Dataset
class ClassIncrementalMNIST(Dataset):
    def __init__(self, root, train=True, transform=None, classes=None):
        self.mnist_dataset = torchvision.datasets.MNIST(root=root,
train=train, transform=transforms.ToTensor(), download=True)
        self.transform = transform
        self.classes = classes
        self.train = train
        # Filter data to include only the specified classes
        self.data = []
        self.targets = []
        for image, label in self.mnist_dataset:
            if label in self.classes:
```

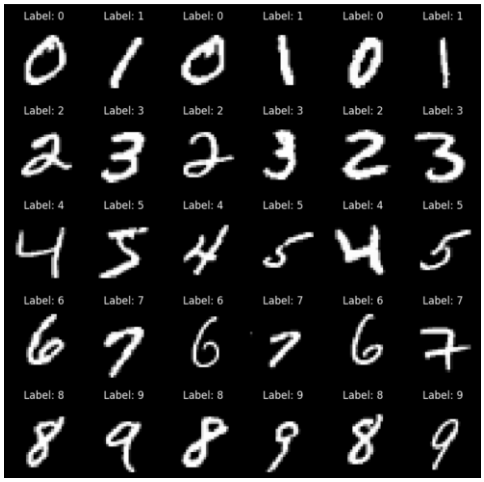
```
                self.data.append(image)
                self.targets.append(label)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        image, label = self.data[idx], self.targets[idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# Setup Class Incremental MNIST Tasks
num_tasks = 5
classes_per_task = 2

# Divide classes into tasks
class_splits = [list(range(i * classes_per_task, (i + 1) *
classes_per_task)) for i in range(num_tasks)]

# Load datasets for each task
train_tasks = [ClassIncrementalMNIST(root="./data", train=True,
classes=class_splits[i]) for i in range(num_tasks)]
test_tasks = [ClassIncrementalMNIST(root="./data", train=False,
classes=class_splits[i]) for i in range(num_tasks)]

# Function to create DataLoaders
def get_task_data(task_idx, batch_size=64):
    train_loader = DataLoader(train_tasks[task_idx],
batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_tasks[task_idx], batch_size=batch_size,
shuffle=False)
    return train_loader, test_loader

# Define Experience Replay Buffer
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.data = []
        self.labels = []

    def add(self, images, labels):
        self.data.extend(images)
        self.labels.extend(labels)

        # Keep buffer size within capacity
        if len(self.data) > self.capacity:
            self.data = self.data[-self.capacity:]
            self.labels = self.labels[-self.capacity:]
```

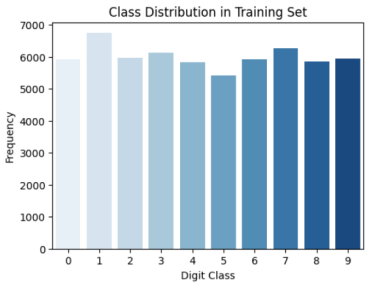
```
    def sample(self, batch_size):
        indices = random.sample(range(len(self.data)), min(batch_size,
len(self.data)))
        return torch.stack([self.data[i] for i in indices]),
torch.tensor([self.labels[i] for i in indices], dtype=torch.long)

# Define SimpleNN model
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Train function with experience replay
def train_with_experience_replay(model, task_idx, criterion, optimizer,
buffer, epochs=5, batch_size=64):
    train_loader, _ = get_task_data(task_idx)

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct, total = 0, 0

        for inputs, labels in train_loader:
            # Sample from the buffer
            if len(buffer.data) > 0:
                replay_inputs, replay_labels = buffer.sample(batch_size //
2)
                inputs = torch.cat((inputs, replay_inputs))
                labels = torch.cat((labels, replay_labels))

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

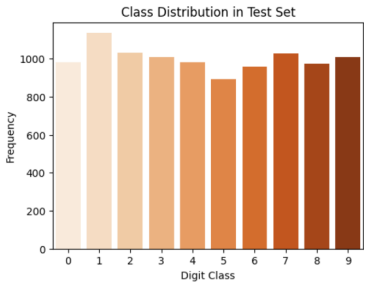
```
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total
        print(f'Task {task_idx+1}, Epoch {epoch+1}/{epochs}, Loss:
{epoch_loss:.4f}, Accuracy: {epoch_acc:.2f}%')

# Evaluate function for cumulative classes
def evaluate_cumulative_classes(model, num_tasks):
    combined_test_set = ConcatDataset(test_tasks[:num_tasks])
    test_loader = DataLoader(combined_test_set, batch_size=64,
shuffle=False)

    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy

# Full continual learning setup with experience replay
def demonstrate_experience_replay():
    input_size = 28 * 28
    hidden_size = 256
    learning_rate = 0.01
    epochs_per_task = 5
    replay_capacity = 500  # Buffer capacity

    # Start with output size for all 10 MNIST digits
    output_size = 10

    model = SimpleNN(input_size, hidden_size, output_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    buffer = ReplayBuffer(replay_capacity)
    accuracies = []
```

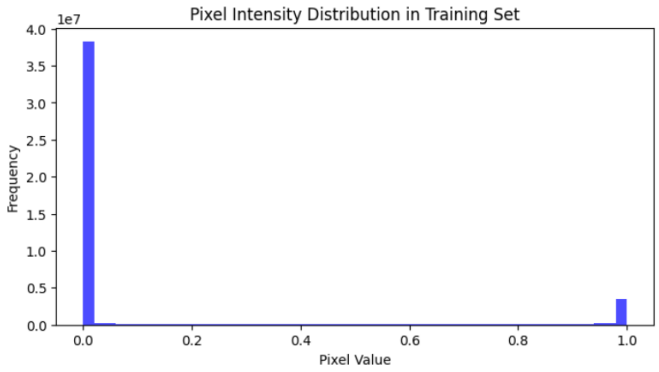
```
    for task_idx in range(len(class_splits)):
        print(f"\n{'='*50}")
        print(f"Training on Task {task_idx+1}: Classes
{class_splits[task_idx]}")
        print(f"{'='*50}")

        # Train with experience replay
        train_with_experience_replay(model, task_idx, criterion,
optimizer, buffer, epochs_per_task)

        # Evaluate on cumulative tasks
        accuracy = evaluate_cumulative_classes(model, task_idx + 1)
        accuracies.append(accuracy)
        print(f"\nModel Accuracy after Task {task_idx + 1}:
{accuracy:.2f}%")

        # Add current task data to buffer
        current_task_data = torch.stack([x for x, _ in
train_tasks[task_idx]])
        current_task_labels = torch.tensor([y for _, y in
train_tasks[task_idx]], dtype=torch.long)
        buffer.add(current_task_data, current_task_labels)

    return accuracies

# Run the demonstration
if __name__ == "__main__":
    experience_replay_accuracies = demonstrate_experience_replay()
```

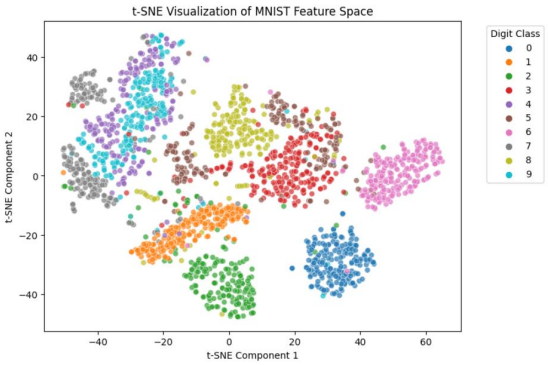
SCREENSHOTS:

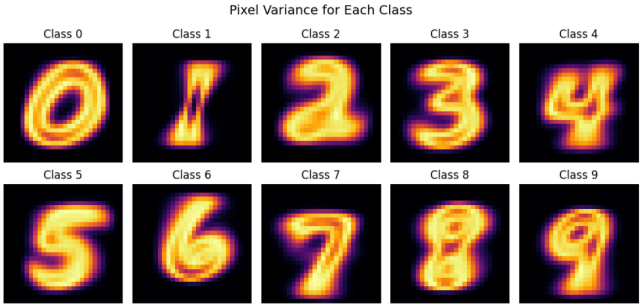








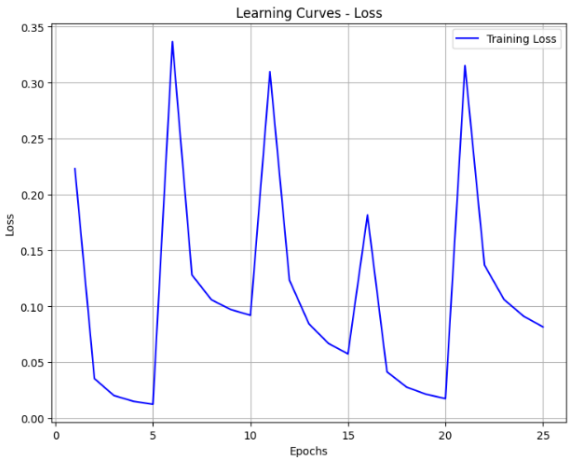


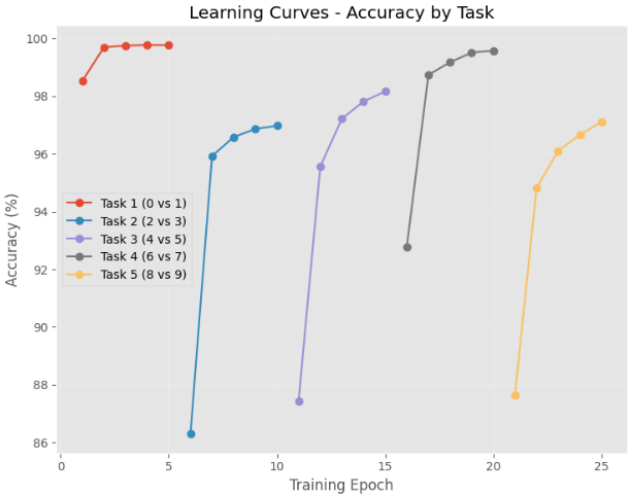


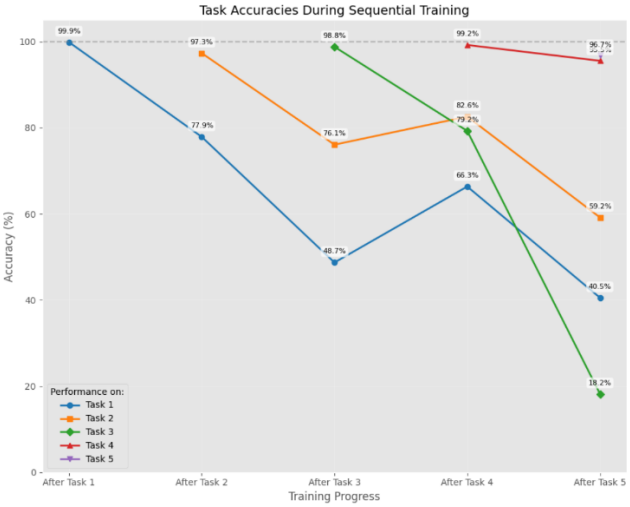

```
Number of Training Samples: 60000
Number of Test Samples: 10000
Input Feature Shape: torch.Size([1, 28, 28])
Number of Classes: 10
Task 1: Class 0 vs Class 1
Task 2: Class 2 vs Class 3
Task 3: Class 4 vs Class 5
Task 4: Class 6 vs Class 7
Task 5: Class 8 vs Class 9
```

```
==================================================
Training on Task 1: 0 vs 1
==================================================
Task 1, Epoch 1/5, Loss: 0.2312, Accuracy: 98.09%
Task 1, Epoch 2/5, Loss: 0.0365, Accuracy: 99.68%
Task 1, Epoch 3/5, Loss: 0.0209, Accuracy: 99.72%
Task 1, Epoch 4/5, Loss: 0.0156, Accuracy: 99.75%
Task 1, Epoch 5/5, Loss: 0.0128, Accuracy: 99.75%

Evaluating on all tasks seen so far:
Task 1 Accuracy: 99.91%

==================================================
Training on Task 2: 2 vs 3
==================================================
Task 2, Epoch 1/5, Loss: 0.3406, Accuracy: 85.99%
Task 2, Epoch 2/5, Loss: 0.1303, Accuracy: 95.85%
Task 2, Epoch 3/5, Loss: 0.1076, Accuracy: 96.56%
Task 2, Epoch 4/5, Loss: 0.0983, Accuracy: 96.72%
Task 2, Epoch 5/5, Loss: 0.0928, Accuracy: 96.84%

Evaluating on all tasks seen so far:
Task 1 Accuracy: 77.26%
Task 2 Accuracy: 97.40%

==================================================
Training on Task 3: 4 vs 5
==================================================
Task 3, Epoch 1/5, Loss: 0.3055, Accuracy: 87.37%
Task 3, Epoch 2/5, Loss: 0.1241, Accuracy: 95.45%
Task 3, Epoch 3/5, Loss: 0.0853, Accuracy: 97.17%
Task 3, Epoch 4/5, Loss: 0.0680, Accuracy: 97.82%
Task 3, Epoch 5/5, Loss: 0.0583, Accuracy: 98.23%

Evaluating on all tasks seen so far:
Task 1 Accuracy: 48.13%
Task 2 Accuracy: 75.51%
Task 3 Accuracy: 98.61%
```

```
==================================================
Training on Task 4: 6 vs 7
==================================================
Task 4, Epoch 1/5, Loss: 0.1853, Accuracy: 93.18%
Task 4, Epoch 2/5, Loss: 0.0414, Accuracy: 98.68%
Task 4, Epoch 3/5, Loss: 0.0272, Accuracy: 99.19%
Task 4, Epoch 4/5, Loss: 0.0209, Accuracy: 99.45%
Task 4, Epoch 5/5, Loss: 0.0173, Accuracy: 99.57%

Evaluating on all tasks seen so far:
Task 1 Accuracy: 65.67%
Task 2 Accuracy: 82.66%
Task 3 Accuracy: 80.20%
Task 4 Accuracy: 99.19%

==================================================
Training on Task 5: 8 vs 9
==================================================
Task 5, Epoch 1/5, Loss: 0.3229, Accuracy: 87.22%
Task 5, Epoch 2/5, Loss: 0.1389, Accuracy: 94.86%
Task 5, Epoch 3/5, Loss: 0.1089, Accuracy: 96.03%
Task 5, Epoch 4/5, Loss: 0.0928, Accuracy: 96.61%
Task 5, Epoch 5/5, Loss: 0.0829, Accuracy: 97.02%

Evaluating on all tasks seen so far:
Task 1 Accuracy: 41.28%
Task 2 Accuracy: 59.06%
Task 3 Accuracy: 18.20%
Task 4 Accuracy: 95.62%
Task 5 Accuracy: 96.67%
```








```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 69.60%
Task 2 Accuracy: 84.69%
Task 3 Accuracy: 87.51%
Task 4 Accuracy: 89.74%
Task 5 Accuracy: 93.26%
```

```
⚠️ Bayesian optimization underperformed manual anchors. Defaulting to best manual result.

✅ Best L2 lambda found: 0.050000 with Task 1 final accuracy: 86.08%
Optimal L2 lambda value for Task 1 performance: 0.05
```

```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 88.80%
Task 2 Accuracy: 81.77%
Task 3 Accuracy: 60.30%
Task 4 Accuracy: 51.58%
Task 5 Accuracy: 90.77%
```

```
✅ Best EWC lambda found: 21.75 with Task 1 final accuracy: 77.13%
Optimal EWC lambda value for Task 1 performance: 21.75195311877766
```

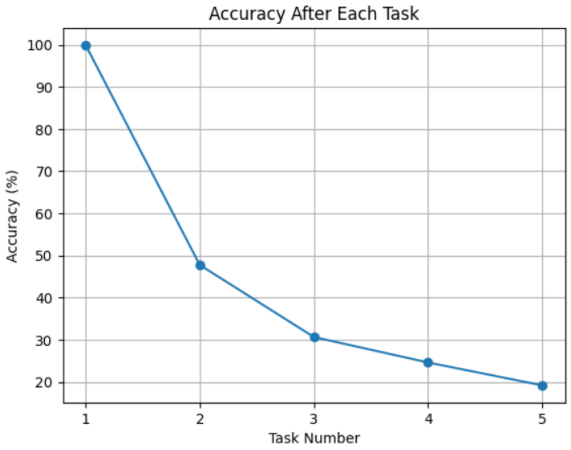
```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 80.54%
Task 2 Accuracy: 85.56%
Task 3 Accuracy: 87.65%
Task 4 Accuracy: 90.53%
Task 5 Accuracy: 93.12%
```

```
Best SI lambda found: 0.20 with Task 1 final accuracy: 78.57%
Optimal SI lambda value for Task 1 performance: 0.19949166150633935
```

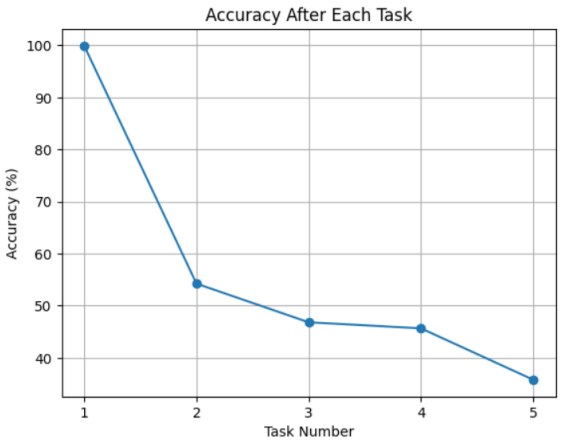
```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 70.87%
Task 2 Accuracy: 83.39%
Task 3 Accuracy: 85.21%
Task 4 Accuracy: 90.29%
Task 5 Accuracy: 93.15%
```

```
Evaluating on all tasks seen so far:
Task 1 Accuracy: 89.71%
Task 2 Accuracy: 21.75%
Task 3 Accuracy: 12.42%
Task 4 Accuracy: 24.61%
Task 5 Accuracy: 24.20%
```

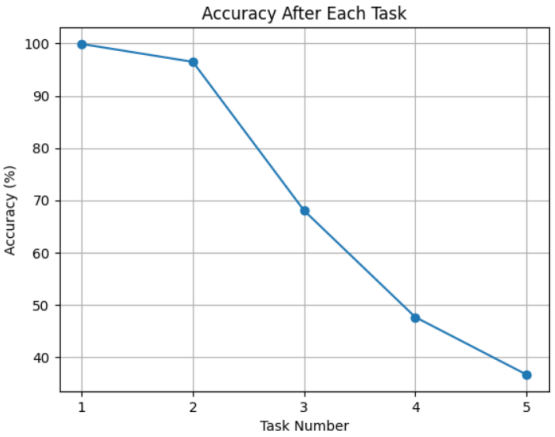
Task	Accuracy (%)
After Training Task 1	99.91
After Training Task 2	47.75
After Training Task 3	30.66
After Training Task 4	24.62
After Training Task 5	19.20



Task	Accuracy (%)
After Training Task 1	99.91
After Training Task 2	54.94
After Training Task 3	44.12
After Training Task 4	44.17
After Training Task 5	34.93



Task	Accuracy (%)
After Training Task 1	99.91
After Training Task 2	96.49
After Training Task 3	68.03
After Training Task 4	47.64
After Training Task 5	36.64



10.2 PLAGIARISM REPORT

11. REFERENCES

[1] Van de Ven, G. M., & Tolias, A. S. (2019). *Three scenarios for continual learning.* arXiv preprint arXiv:1904.07734.

[2] Wang, L., Zhang, X., Su, H., & Zhu, J. (2024). *A comprehensive survey of continual learning: Theory, method and application. IEEE Transactions on Pattern Analysis and Machine Intelligence,* 46(8), 5362-5383.

[3] De Lange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., & Tuytelaars, T. (2022). *A continual learning survey: Defying forgetting in classification tasks. IEEE Transactions on Pattern Analysis and Machine Intelligence,* 44(7), 3366-3385.

[4] Lee, C. S., & Lee, A. Y. (2020). *Clinical applications of continual learning machine learning. The Lancet Digital Health,* 2(6), e279-e281.

[5] Chen, P. H., Wei, W., Hsieh, C.-j., & Dai, B. (2021). *Overcoming catastrophic forgetting by generative regularization.* arXiv preprint arXiv:1912.01238.

12.WORKLOG

Day	Date	Task Done
Day 1	19/02/2025	Project domain discussion: Continual Learning Title: A Comparative Study of Methods for Mitigating Catastrophic Forgetting in Neural Networks
Day 2	20/02/2025	Abstract, Problem Statement, Objectives, Dataset, Methodology, Related Literature
Day 3	21/02/2025	Objective, Architecture, References, EDA
Day 4	23/02/2025	Zeroth Review
Day 5	24/02/2025	Preprocessing, Setting up Baseline Scenarios
Day 6	22/02/2025	Implementing Baseline Scenarios
Day 7	25/02/2025	Implementing Baseline Scenarios
Day 8	26/02/2025	Demonstrating Forgetting in Neural Networks using Baseline
Day 9	28/02/2025	First Review Presentation
Day 10	03/03/2025	Selecting methods to mitigate forgetting

Day 11	04/03/2025	Implementing L2 Regularization and EWC
Day 12	05/03/2025	Implementing SI and PackNet
Day 13	06/03/2025	Tuning hyperparameters
Day 14	07/03/2025	Evaluation and comparison of results
Day 15	10/03/2025	Second Review Presentation
Day 16	11/03/2025	Setting up Class Incremental Learning Scenario
Day 17	12/03/2025	Implementing Class-IL Baseline
Day 18	13/03/2025	Implementing Naïve Rehearsal and Experience Replay
Day 19	17/03/2025	Evaluation and Comparison of Results
Day 20	18/03/2025	Final Review Presentation