

# Reproducibility Report on *Defense without Forgetting* (*Continual Adversarial Defense with Anisotropic Isotropic Pseudo Replay*)

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

February 24, 2025

## Abstract

In this project, I fully implemented the concepts outlined in the paper *Defense without Forgetting: Continual Adversarial Defense with Anisotropic Isotropic Pseudo Replay* by Yuhang Zhou and Zhongyun Hua. The primary objective of this paper was to address the challenge of continual adversarial defense in deep neural networks (DNNs). The authors focused on how adversarial defenses often suffer from catastrophic forgetting when they are adapted to new attacks, leading to degraded performance on previously defended tasks.

## 1 Introduction

In recent years, the security of deep learning models has become a critical focus of research, particularly due to their vulnerability to adversarial attacks. These attacks, which involve small perturbations to the input data that lead to incorrect predictions, highlight significant weaknesses in machine learning systems deployed in real-world applications. While adversarial defense methods such as adversarial training (AT) have proven effective against certain types of attacks, they typically operate in a one-shot setting, assuming that once a model is trained to withstand a particular attack, it remains robust indefinitely. However, in practice, new attacks continuously emerge, requiring models to adapt and remain robust to evolving threats.

A major challenge in continual learning, including continual adversarial defense, is catastrophic forgetting—the phenomenon where a model, while adapting to new tasks or attacks, loses its ability to perform well on previously learned tasks. This issue becomes especially pronounced when the model is exposed to a sequence of adversarial attacks, which may cause it to forget the defensive capabilities developed against earlier attacks.

To address this, the paper "Defense without Forgetting: Continual Adversarial Defense with Anisotropic Isotropic Pseudo Replay" introduces an innovative approach to continual adversarial defense, called Anisotropic Isotropic Replay (AIR). AIR combines self-distillation and advanced data augmentation techniques to prevent catastrophic forgetting while allowing the model to adapt to new adversarial threats. This report details the implementation of AIR, its application to various datasets and adversarial attacks, and the results of its performance compared to traditional defense methods. The goal is to explore how continual adversarial defenses can be improved to ensure the robustness of models in the face of emerging threats.

## 2 Methodology

### 2.1 Anisotropic & Isotropic Replay (AIR) Method

The AIR method is designed to prevent catastrophic forgetting while adapting to new adversarial attacks. This technique integrates Anisotropic Replay (AR) and Isotropic Replay (IR) to form a continual adversarial defense system that is capable of learning new attacks without forgetting how to defend against previous ones.

#### Key Components of AIR

##### 1. Isotropic Replay (IR):

- **Goal:** To augment the current adversarial samples with small perturbations and generate new pseudo-samples. These augmented samples help the model focus on robust features that generalize well across different adversarial attacks.
- **Mathematical Formulation:** The noisy inputs are generated by adding small Gaussian noise to the original adversarial examples, which is then augmented with random transformations:

$$X_{\text{IR}} = T(X_t + \lambda \cdot r)$$

where:

- $T$  is a random augmentation operator.
  - $\lambda$  is a hyperparameter controlling the strength of the perturbation.
  - $r$  is the noise sample from a Gaussian distribution.
  - $X_t$  is the adversarial example at time  $t$ .
- **Loss Function:** The loss for isotropic replay is:

$$L_{\text{IR}} = \text{KL}(f_{w_t}(X_{\text{IR}}), f_{w_{t-1}}(X_{\text{IR}}))$$

where  $f_{w_t}(X)$  represents the model output at time  $t$ , and  $f_{w_{t-1}}(X)$  is the output of the model at the previous time step.

##### 2. Anisotropic Replay (AR):

- **Goal:** To prevent forgetting by blending current adversarial examples with samples from earlier adversarial examples. The approach applies a stochastic mixing technique (similar to mixup) to combine data from different sources.
- **Mathematical Formulation:** The augmented data is generated by mixing two adversarial examples and their corresponding labels:

$$X_{\text{AR}} = \alpha \cdot X_t + (1 - \alpha) \cdot X_{\text{shuffle}}$$

where:

- $\alpha$  is a mixing coefficient sampled from a uniform distribution.
  - $X_{\text{shuffle}}$  is a shuffled version of the current batch  $X_t$ .
- **Loss Function:** The loss for anisotropic replay is:

$$L_{\text{AR}} = \text{KL}(f_{w_{t-1}}(X_{\text{AR}}), f_{w_t}(X_{\text{AR}}))$$

### 3. Self-Distillation:

A key technique in the AIR method is self-distillation, which ensures that the model’s output does not deviate drastically when exposed to noisy or augmented data. This is achieved by applying a regularization term that encourages consistency in the model’s predictions across different perturbations of the input.

### 4. Regularization for Stability and Plasticity Trade-off:

To balance the model’s ability to adapt to new tasks (plasticity) and its ability to retain previous knowledge (stability), a regularization term is added. This is typically achieved by minimizing the KL divergence between the outputs of the model with and without dropout.

### 5. Final AIR Loss Function:

The overall loss function that combines all components is:

$$L_{\text{AIR}} = L_{\text{AT}} + \lambda_{\text{SD}}(L_{\text{IR}} + L_{\text{AR}}) + \lambda_{\text{Reg}}L_{\text{Reg}}$$

where:

- $L_{\text{AT}}$  is the adversarial training loss for the current attack.
- $L_{\text{IR}}$  and  $L_{\text{AR}}$  are the isotropic and anisotropic replay losses, respectively.
- $L_{\text{Reg}}$  is the regularization term for stability and plasticity.

## 2.2 Traditional Adversarial Defense Methods

While the AIR method offers a continual adversarial defense approach, it can be complemented by traditional adversarial defense methods. These methods focus on different aspects of adversarial defense, including robustness to new attacks, preservation of knowledge from old attacks, and the maintenance of model accuracy across tasks. The following methods play a crucial role in generating a comprehensive adversarial defense framework:

### 2.2.1 Vanilla Adversarial Training

Vanilla Adversarial Training is one of the simplest and most widely used techniques for adversarial defense. The basic idea is to train the model on both clean and adversarial examples simultaneously, using a weighted combination of both loss terms. The goal is to ensure the model learns to generalize well on adversarial examples while still maintaining accuracy on clean data.

**Loss Function:** The total loss in Vanilla Adversarial Training is the weighted sum of the cross-entropy loss on clean data and the adversarial loss:

$$L_{\text{total}} = \lambda \cdot L_{\text{adv}} + (1 - \lambda) \cdot L_{\text{clean}}$$

Where:

- $L_{\text{adv}}$  is the cross-entropy loss for adversarial examples:

$$L_{\text{adv}} = - \sum_i y_i \log(p_i(x_{\text{adv}}))$$

- $L_{\text{clean}}$  is the cross-entropy loss for clean examples:

$$L_{\text{clean}} = - \sum_i y_i \log(p_i(x_{\text{clean}}))$$

- $\lambda$  is a hyperparameter that controls the balance between clean and adversarial losses. A higher value of  $\lambda$  makes the model focus more on defending against adversarial examples, while a lower value emphasizes clean data.

#### Training Procedure:

1. Adversarial Example Generation: Adversarial examples are generated using a chosen attack method, such as Projected Gradient Descent (PGD) or Fast Gradient Sign Method (FGSM).
2. Loss Calculation: Both the clean and adversarial examples are passed through the model, and the respective losses are computed.
3. Model Update: The total loss is computed, and backpropagation is performed to update the model's parameters using an optimizer (e.g., Adam or SGD).

### 2.2.2 Elastic Weight Consolidation (EWC)

Elastic Weight Consolidation (EWC) is a method designed to mitigate catastrophic forgetting during the training of deep neural networks. In the context of adversarial defense, it ensures that the model retains the knowledge learned from previous adversarial examples while being trained on new ones. EWC achieves this by regularizing the model's weights to prevent significant changes to important parameters, which were crucial for solving earlier tasks.

**Loss Function:** The total loss in EWC is a combination of the task loss (e.g., cross-entropy) and a regularization term that penalizes changes to important weights:

$$L_{\text{total}} = L_{\text{task}} + \lambda_{\text{ewc}} \sum_i F_i \cdot (\theta_i - \theta_i^*)^2$$

Where:

- $L_{\text{task}}$  is the cross-entropy loss for the current task (adversarial examples in this case):

$$L_{\text{task}} = - \sum_i y_i \log(p_i(x))$$

- $F_i$  is the Fisher information for the  $i$ -th parameter, which measures the importance of each parameter to the task.
- $\theta_i^*$  is the optimal value of the parameter  $\theta_i$  from the previous task.
- $\lambda_{\text{ewc}}$  is a hyperparameter that controls the strength of the regularization.

#### Training Procedure:

1. Model Training: The model is trained on adversarial examples, and the Fisher information is computed for each parameter during training.
2. Loss Calculation: The loss consists of both the adversarial loss and the EWC regularization term, which prevents large changes in important weights.
3. Parameter Update: The model is updated by minimizing the total loss, which includes the EWC regularization to preserve important parameters.

### 2.2.3 Feature Extraction

Feature extraction methods focus on ensuring that the model learns robust features that are invariant to small perturbations in the input data, which is crucial for defending against adversarial attacks. In this method, the model is trained to produce consistent feature representations for both clean and noisy inputs. This prevents the model from overfitting to specific input values and encourages it to focus on more robust, generalizable features.

**Loss Function:** The total loss in feature extraction is a combination of the cross-entropy loss for clean data and the feature consistency loss, which is the MSE between the features extracted from clean and noisy inputs:

$$L_{\text{total}} = L_{\text{CE}} + \lambda_{\text{feat}} \cdot L_{\text{feat}}$$

Where:

- $L_{\text{CE}}$  is the cross-entropy loss for clean data:

$$L_{\text{CE}} = - \sum_i y_i \log(p_i(x))$$

- $L_{\text{feat}}$  is the feature consistency loss, calculated as the MSE between the clean and noisy feature representations:

$$L_{\text{feat}} = \|f_{\text{student}}(x) - f_{\text{student}}(x_{\text{noisy}})\|_2^2$$

- $\lambda_{\text{feat}}$  is a hyperparameter controlling the importance of the feature consistency loss.

#### Training Procedure:

1. Noise Addition: Noise is added to the clean input data to generate noisy inputs. The noise can be Gaussian or from any other distribution.
2. Feature Extraction: Features are extracted from both clean and noisy inputs using the model.
3. Loss Calculation: The model is trained to minimize the cross-entropy loss for clean data and the feature consistency loss for the clean and noisy features.
4. Model Update: The model is updated by backpropagating the total loss, which ensures robust feature extraction.

### 2.2.4 Joint Training

Joint Training combines adversarial training with the learning of new tasks. This method ensures that the model learns to defend against new adversarial attacks without forgetting how to defend against previous ones. The total loss in Joint Training is a weighted sum of the clean data loss and adversarial data loss.

**Loss Function:** The total loss in Joint Training is:

$$L_{\text{total}} = \lambda \cdot L_{\text{clean}} + (1 - \lambda) \cdot L_{\text{adv}}$$

Where:

- $L_{\text{clean}}$  is the cross-entropy loss for clean data:

$$L_{\text{clean}} = - \sum_i y_i \log(p_i(x_{\text{clean}}))$$

- $L_{\text{adv}}$  is the cross-entropy loss for adversarial data:

$$L_{\text{adv}} = - \sum_i y_i \log(p_i(x_{\text{adv}}))$$

- $\lambda$  is a hyperparameter controlling the trade-off between clean and adversarial losses.

#### Training Procedure:

1. Adversarial Example Generation: Adversarial examples are generated for each clean input using a chosen attack method.
2. Loss Calculation: The total loss is computed by combining the clean and adversarial losses, with the appropriate weight given to each.
3. Model Update: The model is updated by backpropagating the total loss and using an optimizer.

### 2.2.5 Less-Forgetting Learning (LFL)

Less-Forgetting Learning (LFL) is a method used to ensure that a model retains knowledge from previous tasks while learning new ones, thereby mitigating catastrophic forgetting. The total loss in LFL is a combination of the cross-entropy loss and the feature consistency loss.

**Loss Function:** The total loss in LFL is:

$$L_{\text{total}} = L_{\text{CE}} + \lambda_e \cdot L_{\text{feature\_dist}}$$

Where:

- $L_{\text{CE}}$  is the cross-entropy loss:

$$L_{\text{CE}} = - \sum_i y_i \log(p_i(x))$$

Here,  $p_i(x)$  is the predicted probability for class  $i$ , and  $y_i$  is the ground truth label for class  $i$ .

- $L_{\text{feature\_dist}}$  is the feature consistency loss, which measures the similarity between the feature representations of the student model and the teacher model:

$$L_{\text{feature\_dist}} = \|f_{\text{student}}(x) - f_{\text{teacher}}(x)\|_2^2$$

Here,  $f_{\text{student}}(x)$  and  $f_{\text{teacher}}(x)$  are the feature representations of the student and teacher models, respectively, for input  $x$ .

- $\lambda_e$  is a hyperparameter that controls the trade-off between the cross-entropy loss and the feature consistency loss. A higher value of  $\lambda_e$  places more emphasis on retaining previous knowledge.

#### Training Procedure:

1. Student and Teacher Models: The teacher model is typically a pre-trained model that has learned previous tasks. The student model is trained on the current task and aims to learn both the classification task and preserve knowledge from the teacher.
2. Loss Calculation: The total loss is calculated by combining the cross-entropy loss (to ensure correct classification) and the feature consistency loss (to retain previous knowledge).
3. Model Update: The student model is updated using the total loss, and an optimizer is used to backpropagate the gradients and adjust the model parameters.

## 2.3 Experimental Setup

### 2.3.1 Hardware and Software Environment

The experiments were conducted using the resources available on Google Colab, equipped with an NVIDIA A100 GPU. This powerful GPU allowed for efficient model training and adversarial attack testing, ensuring that the computations could be handled efficiently. The software environment was configured with the following dependencies:

- `numpy` version 1.21.0 or higher for numerical operations.
- `torch` version 2.0.0 or higher for model building and training.
- `torchvision` version 0.15.0 or higher for image transformations and dataset handling.
- `matplotlib` version 3.4 for plotting and visualizing results.
- `scikit-learn` version 1.0 for auxiliary machine learning functions.

These libraries ensured smooth model building, training, and evaluation.

### 2.3.2 Dataset Preparation and Preprocessing

The experiments utilized the following datasets: MNIST, CIFAR-10, and CIFAR-100. These datasets are standard in adversarial robustness research and provide varied challenges for model evaluation.

- **MNIST:** Handwritten digit dataset with grayscale images (28x28 pixels).
- **CIFAR-10 and CIFAR-100:** Image datasets with 32x32 RGB images belonging to 10 and 100 classes, respectively.

For all datasets, the images were normalized using their respective mean and standard deviation values. Data augmentation techniques such as random rotations, flips, and crops were applied to increase the robustness of the models.

### 2.3.3 Hyperparameters and Configurations

The following hyperparameters and configurations were defined in the experimental setup:

#### General Parameters

- **seed:** A fixed seed used for random number generation to ensure reproducibility of results.
- **device:** Specifies the hardware device for computation, either `cpu` or `cuda` for GPU acceleration.

#### Training Parameters

- **epochs:** The number of full passes over the training dataset.
- **batch\_size:** The number of samples used in one iteration of training.
- **learning\_rate:** The step size for updating model weights during training.
- **momentum:** The momentum factor used in the optimization algorithm to accelerate convergence and reduce oscillations.
- **weight\_decay:** A regularization parameter used to penalize large weights to prevent overfitting.

## Adversarial Attack Parameters

- **epsilon**: The magnitude of perturbation added to the input images in adversarial attacks.
- **alpha**: The step size in the iterative adversarial attack algorithm.
- **num\_steps**: The number of iterations in the adversarial attack process.
- **random\_init**: A flag indicating whether the adversarial attack should start with a random initialization.

## Defense Method

- **defense\_method**: Specifies the defense method used during training. In this case, it uses AIR (Anisotropic & Isotropic Replay).

## AIR Parameters

- **lambda\_SD**: A scaling factor for the self-distillation loss, which aligns the outputs of the current and previous models.
- **lambda\_IR**: A scaling factor for the isotropic replay loss, which ensures consistency between the current model and previous adversarial examples.
- **lambda\_AR**: A scaling factor for the anisotropic replay loss, which mixes data from previous and current attacks to retain prior knowledge.
- **lambda\_Reg**: A scaling factor for the regularization loss, used to control the trade-off between stability and plasticity in the model.
- **alpha\_range**: The range of mixing factors used in anisotropic replay augmentation.
- **use\_rdrop**: A flag to determine whether regularization with random dropout (R-Drop) should be applied.

## Isotropic Replay Augmentations

- **iso\_noise\_std**: The standard deviation of isotropic noise applied to adversarial examples.
- **iso\_clamp\_min, iso\_clamp\_max**: The minimum and maximum values for isotropic augmentation, controlling the range of pixel values in the images.
- **iso\_p\_flip**: The probability of applying a random flip to the images during isotropic augmentation.
- **iso\_flip\_dim**: The dimension along which the flip occurs (e.g., 3D RGB channels).
- **iso\_p\_rotation**: The probability of applying random rotation to the images.
- **iso\_max\_rotation**: The maximum degree of rotation applied to the images.
- **iso\_p\_crop, iso\_p\_erase**: The probabilities of applying random cropping and erasing operations to the images.

### Dataset Parameters

- **dataset**: The dataset used for training and evaluation. The choices include MNIST, CIFAR-10, and CIFAR-100.
- **data\_root**: The root directory where the dataset is stored or will be downloaded.
- **num\_workers**: The number of worker threads used for loading the dataset.

### LFL (Less Forgetting Learning) Parameters

- **lambda\_lfl**: A scaling factor for the less-forgetting learning (LFL) loss.
- **feature\_lambda**: A regularization parameter used for feature extraction.
- **freeze\_classifier**: A flag to freeze the classifier during training to prevent its weights from being updated.

### Joint Training Parameters

- **joint\_lambda**: A scaling factor used for joint training in multi-task settings.

### Vanilla Adversarial Training (VanillaAT) Parameters

- **adv\_lambda**: A scaling factor for the adversarial loss used in vanilla adversarial training.

### Feature Extraction Parameters

- **feat\_lambda**: A scaling factor for the feature extraction loss.

### EWC (Elastic Weight Consolidation) Parameters

- **lambda\_ewc**: A regularization factor used in the EWC loss to preserve knowledge from previous tasks.

### Multi-Task or Multi-Attack Scenario

- **attack\_sequence**: A tuple defining the sequence of attacks used for training. Possible sequences include combinations of `None`, `FGSM`, and `PGD`.

## 3 Results

Regarding the results, I have tested all the models that I was supposed to obtain and evaluate, but due to time constraints in the project, these tests were initially run with epochs set to 1. The logs from these tests have been uploaded to the repository. Currently, I'm running the tests with epochs set to 50, and these results should be available in the coming days. I'll present the complete results during the final presentation.

That being said, I'm confident that since the code runs without issues on the methods and datasets with just 1 epoch, there shouldn't be any major problems generating the final outputs. However, there was one completed test where the Small CNN model didn't perform well under a scenario involving two tasks. The first task was an FGSM attack, and the second one was a PGD attack. For the first task (FGSM), which is a simpler attack, the model performed well, achieving a resistance percentage close to what the paper reports. However, for the second task (PGD attack), we didn't reach the same accuracy as the paper's result. I believe this is due to

hyperparameters, which the paper briefly mentioned but didn't clearly define. Still, since the project has a modular setup, these hyperparameters can be adjusted, and we can try obtaining better results.

One other point I should mention is the learning rate. If it's set too high, the loss becomes NaN, meaning the model diverges. By reducing this value, I was able to fix this issue, and the model converged properly.

Now that I've analyzed the full results from training with 50 epochs, I've noticed that for some attacks (such as None to FGSM and PGD to FGSM), our results match the paper's findings. However, for other attack scenarios, there are significant differences.

For the None to PGD case, our model initially performed well in the clean (no attack) setting, showing good accuracy and robustness. But once it started learning the PGD attack, the accuracy dropped from 67% to 10% at epoch 47. The logs show that from epochs 47 to 50, the model couldn't recover, meaning it overfitted instead of generalizing. Because of this, not only did PGD robustness fail, but even clean accuracy suffered.

In the PGD to None scenario, the model reached 78% accuracy at epoch 4, but after that, it completely diverged and never converged back. This seems to be caused by model collapse, which led to the None scenario performing extremely poorly, much worse than expected.

In the FGSM to PGD case, the model initially performed well on FGSM, but once it switched to PGD, the accuracy significantly dropped. By analyzing the training logs, I noticed that the model achieved a good accuracy at epoch 45, but after that, it started diverging and never improved, leading to poor final results.

For the FGSM to None case, the model performed well in the first attack (FGSM), but during the second phase (None), it overfitted. As a result, the model forgot the first task, and only the second task gave acceptable results, which is not ideal.

When analyzing the three-task scenarios, two main issues appeared. In both cases, the model did well in task 1, but when it reached task 2 (FGSM phase), it forgot the previous task and couldn't generalize. Right now, I don't have a clear explanation for why this happened. However, in the third task (None scenario), we achieved good results, but due to overfitting, the model struggled with earlier tasks. Meanwhile, in the third task (PGD scenario), since the model was training from a bad state, it completely collapsed and never converged.

Another key point is that the training process was extremely time-consuming and required significant hardware resources. Running the experiments for 50 epochs took a long time, and future experiments will likely require even more computational power.

In my opinion, hyperparameter tuning, especially  $L_{\text{reg}}$  (regularization term), could play an important role in improving results. However, the paper did not discuss this aspect. Additionally, the way the paper selects the best model during training is unclear, making it difficult to compare results directly. Their description of the training process is very brief. I also think machine learning techniques for preventing overfitting could be helpful in stabilizing the model's performance.

| Tasks                        | None to FGSM |       | FGSM to None |       | None to PGD |       | PGD to None |       | FGSM to PGD |       | PGD to FGSM |       |
|------------------------------|--------------|-------|--------------|-------|-------------|-------|-------------|-------|-------------|-------|-------------|-------|
|                              | Task1        | Task2 | Task1        | Task2 | Task1       | Task2 | Task1       | Task2 | Task1       | Task2 | Task1       | Task3 |
| Clean Accuracy after Task 1  | 99.22        |       | 99.15        |       | 99.22       |       | 11.35       |       | 98.88       |       | 87.68       |       |
| Clean Accuracy after Task 2  | 99.22        |       | 99.18        |       | 10.48       |       | 11.35       |       | 10.32       |       | 99.31       |       |
| Robust Accuracy after Task 1 | 99.22        | 9.39  | 98.20        | 99.15 | 99.22       | 8.60  | 11.35       | 11.35 | 97.38       | 97.07 | 86.73       | 87.39 |
| Robust Accuracy after Task 2 | 99.22        | 98.03 | 9.82         | 99.18 | 10.48       | 10.10 | 11.35       | 11.35 | 10.32       | 10.32 | 98.10       | 98.29 |

Figure 1: Adaptation between two attacks with AIR method for MNIST dataset

| Tasks                        | None to FGSM |       | FGSM to None |       | None to PGD |       | PGD to None |       | FGSM to PGD |       | PGD to FGSM |       |
|------------------------------|--------------|-------|--------------|-------|-------------|-------|-------------|-------|-------------|-------|-------------|-------|
|                              | Task1        | Task2 | Task1        | Task2 | Task1       | Task2 | Task1       | Task2 | Task1       | Task2 | Task1       | Task3 |
| Robust Accuracy after Task 2 | 99.37        | 98.84 | 98.18        | 98.84 | 98.89       | 94.26 | 95.93       | 99.06 | 97.45       | 95.67 | 96.25       | 97.93 |

Figure 2: Adaptation between two attacks with AIR method for MNIST dataset(Paper)

| Tasks                        | None to FGSM to PGD |       |       | PGD to FGSM to None |       |       |
|------------------------------|---------------------|-------|-------|---------------------|-------|-------|
|                              | Task1               | Task2 | Task3 | Task1               | Task2 | Task3 |
| Clean Accuracy after Task 1  | 99.22               |       |       | 48.34               |       |       |
| Clean Accuracy after Task 2  | 99.11               |       |       | 9.58                |       |       |
| Clean Accuracy after Task 3  | 10.28               |       |       | 99.15               |       |       |
| Robust Accuracy after Task 1 | 99.22               | 9.39  | 8.60  | 47.05               | 47.19 | 48.34 |
| Robust Accuracy after Task 2 | 99.11               | 98.14 | 97.88 | 9.58                | 9.58  | 9.58  |
| Robust Accuracy after Task 3 | 10.28               | 10.28 | 10.28 | 9.58                | 9.58  | 99.15 |

Figure 3: Results among None, FGSM, and PGD with AIR method for MNIST dataset

## 4 Challenges and Discussion

The most significant challenge I faced while implementing this paper was my lack of prior experience in AI model security. This project itself motivated me to enter this field, and initially, many of the concepts in the paper were unclear to me. Terms like *Isotropic Pseudo Replay*, *Anisotropic Pseudo Replay*, and *Self-distillation Pseudo Replay*, which are central to the paper, were difficult to grasp. To address this, I began by studying these core concepts in detail. Without understanding these ideas, implementation would have been impossible. Once I had a solid grasp of these concepts, I proceeded with the implementation phase.

A challenge during the implementation process was designing a modular structure for the code that would allow for easy debugging, optimization, and future extensions. I organized the code into folders related to specific tasks: one for attacks, one for datasets, one for methods, one for models, and another for replay-related code used in the AIR method. Additionally, I created a folder for utility code, including logging, plotting, and model evaluation. The code was split into two main parts: one for training and another for the main script, which orchestrated everything.

Another difficulty I encountered was the lack of clear hyperparameter settings in the paper. The paper referenced other studies for model details and hyperparameters but did not provide exact values. This forced me to read four additional papers to determine the necessary hyperparameters and model architectures, such as *Small\_cnn*, *WRN-34-10*, and *WRN-34-20*. After analyzing these sources, I documented the hyperparameters and summarized my findings in a mind map.

In order to make the code more manageable, I created a config file that centralized all the parameters. This allowed any part of the code requiring hyperparameters to access them from one location.

The paper compared its results with five other methods, which required me to read through these papers to understand the approaches used. Integrating these methods into a unified structure was a challenging task, but I successfully completed it.

One issue that caused significant trouble during training was an error I encountered when generating attack images. The error, `RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn`, occurred because I had enabled gradients for the input. After some research, I realized that adding `torch.set_grad_enabled(True)` in the attack gen-

| Tasks                        | None to FGSM to PGD |       |       | PGD to FGSM to None |       |       |
|------------------------------|---------------------|-------|-------|---------------------|-------|-------|
|                              | Task1               | Task2 | Task3 | Task1               | Task2 | Task3 |
| Robust Accuracy after Task 3 | 99.39               | 97.21 | 94.54 | 91.55               | 97.34 | 99.33 |

Figure 4: Results among None, FGSM, and PGD with AIR method for MNIST dataset(Paper)

eration methods for FGSM and PGD would resolve the issue.

The paper suggested training for 50 epochs, which I did, but this led to long training times and high hardware costs.

Another challenge came with the Isotropic Pseudo Replay section, where the paper did not specify hyperparameters or explain the augmentation methods clearly. I had to explore PyTorch methods and determine an appropriate approach that aligned with the paper’s intentions.

Fortunately, the datasets used in the paper were publicly available, which made working with them straightforward.

An additional problem I encountered was that under a heavier attack, such as PGD, the model could not improve and faced significant difficulties. I believe it would be better to use pretrained models or pretrain the model before training on the attacks. However, the article did not provide enough clarity on this issue, and I could not fully understand how to approach it.

## 5 Conclusion

In this project, I successfully implemented the methods described in *Defense without Forgetting: Continual Adversarial Defense with Anisotropic & Isotropic Pseudo Replay* (AIR). The implementation of self-distillation and data augmentation techniques allowed the model to adapt to new adversarial attacks without forgetting how to defend against previous ones. The results showed good performance, particularly under simpler attacks like FGSM, but some discrepancies arose with more complex attacks like PGD, likely due to hyperparameter tuning challenges.

Although the experiments were limited to one epoch due to time constraints, the project established a modular code structure and identified key areas for improvement, such as fine-tuning hyperparameters and exploring different model architectures.

For future work, potential improvements could include:

- Finding optimal hyperparameters for better performance.
- Adding new attacks to evaluate the model’s robustness.
- Implementing and testing other CNN models.
- Expanding this approach to language models and assessing its effectiveness.
- Comparing AIR with other adversarial defense methods to further validate its advantages.

These directions will help refine the method and broaden its applicability to various tasks and domains.

## References

- Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric Xing, Laurent El Ghaoui, and Michael Jordan. Theoretically principled trade-off between robustness and accuracy
- Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attack.

- Zhizhong Li and Derek Hoiem. Learning without forgetting.
- James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska Barwinska, et al. Overcoming catastrophic forgetting in neural networks.
- Heechul Jung, Jeongwoo Ju, Minju Jung, and Junmo Kim. Less-forgetting learning in deep neural networks.
- Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks.
- <https://medium.com/sciforce/adversarial-attacks-explained-and-how-to-defend-ml-models-against-them-d76f7d013b18>
- <https://engineering.purdue.edu/ChanGroup/ECE595/files/chapter3.pdf>
- <https://viso.ai/deep-learning/adversarial-machine-learning/>