TML Assignment #3: Robustness

Github Link :- https://github.com/arsalanmm010/TML25_A3_19

Implementation and Workflow

In this assignment, we aimed to train a robust classifier capable of making correct predictions on both clean and adversarially perturbed data, specifically against attacks generated using Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD). Our overall workflow involved the following key stages:

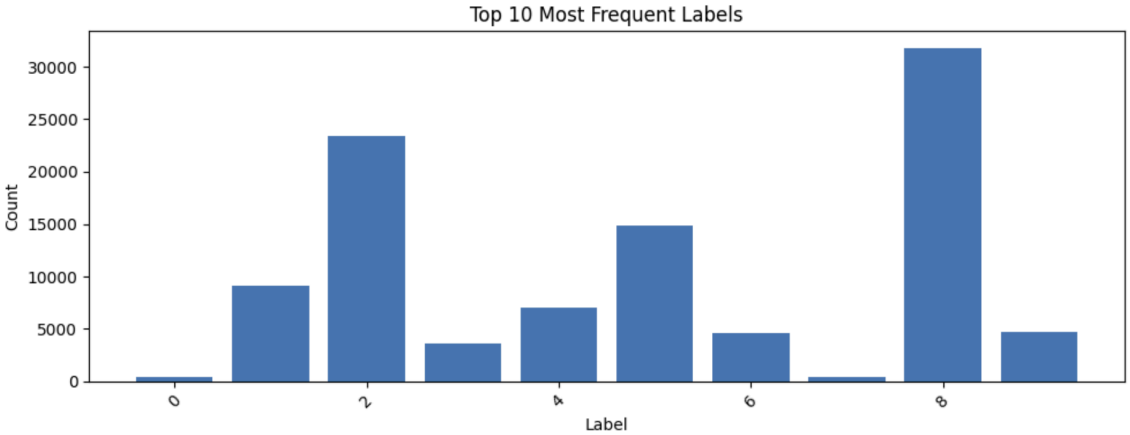
1. **Exploratory Data Analysis and Data Preparation**: We loaded the provided training dataset, performed necessary data analysis, preprocessing, and ensured a representative train-test split.
2. **Model Selection and Initialization**: We strategically chose a robust model architecture i.e., resnet50 (instead of resnet18 & resnet34) and initialized it for our classification task.
3. **Adversarial Training**: We trained our model by incorporating adversarial examples generated dynamically during the training loop, focusing on enhancing its resilience.
4. **Hyperparameter Tuning and Checkpointing**: We systematically tuned hyperparameters (like epoch, batch-size, learning rate and probability for adversarial sample training) and implemented checkpointing to monitor performance and prevent overfitting.
5. **Model Export**: Finally, we saved the trained model's state dictionary for submission and evaluation.

Each step was automated in our code, ensuring reproducibility and facilitating iterative improvements. **Within the train folder, model_training.py is the main code file.**

1. Data Preparation

Our initial step involved thoroughly preparing the image dataset for training.

- **Dataset Loading**: We loaded the training data from the `Train.pt` file. The dataset structure was consistent with previous assignments, with image labels encoded as integers.
- **Exploratory Data Analysis (EDA)**: Through EDA, we identified 10 distinct classes within the dataset. We meticulously checked their distribution to understand any potential imbalances.



- **RGB Conversion**: A crucial finding during EDA was the presence of grayscale images. To ensure consistent input dimensions (3×32×32) for our chosen model, we implemented a `to_rgb` utility function. This function converted all grayscale images to RGB by replicating their single channel across three channels, ensuring uniform input.

```
Image Sizes: Counter({(32, 32): 100000})
Image Modes: Counter({'RGB': 89979, 'L': 10021})
```

- **Normalization**: Images were transformed into PyTorch tensors using `transforms.ToTensor()`. Further normalization, consistent with the expected input for pre-trained models, would be applied implicitly or explicitly.
- **Stratified Split with Rare Tag Handling**: To guarantee that both our training and testing sets accurately reflected the overall distribution of labels in the complete dataset, we performed a stratified shuffle split. This was vital for ensuring that each class was adequately represented in both splits. Furthermore, we incorporated a specific mechanism to handle "rare tags" (classes with fewer than 2 instances), explicitly including them in the training set to prevent their exclusion during the stratification process. The `random_state=42` was consistently used to ensure the reproducibility of our data splits.

Tag	Train	Test	Total	Train %	Test %
0	338	86	424	79.72%	20.28%
1	6650	1682	8332	79.81%	20.19%
2	17005	4214	21219	80.14%	19.86%
3	2613	648	3261	80.13%	19.87%
4	5105	1310	6415	79.58%	20.42%
5	10829	2697	13526	80.06%	19.94%
6	3380	861	4241	79.70%	20.30%
7	329	80	409	80.44%	19.56%
8	22994	5795	28789	79.87%	20.13%
9	3414	854	4268	79.99%	20.01%

2. Model Selection and Initialization

For this challenging task, we strategically selected the ResNet-50 model from `torchvision.models` as our base classifier.

- **Rationale for ResNet-50**: We chose ResNet-50 due to its well-documented complexity and proven efficacy in various image classification benchmarks. Our initial tests and understanding of adversarial robustness suggested that more complex models like ResNet-50, with their deeper architectures and richer feature extraction capabilities, tend to exhibit greater inherent robustness against adversarial perturbations compared to simpler models. This complexity provides a larger capacity to learn intricate decision

boundaries that are less susceptible to subtle adversarial manipulations. We specifically tested this hypothesis during our experimentation phase, observing its superior performance in maintaining accuracy under adversarial conditions.

- **Model Architecture and Adaptation**: We initialized the model using `models.resnet50(pretrained=True)`, leveraging ImageNet pre-trained weights. This allowed us to benefit from features learned on a large dataset, accelerating convergence and improving generalization.
- **Output Layer Modification**: To tailor the model for our specific 10-class classification problem, we replaced the original fully connected layer (`model.fc`) of ResNet-50 with a new linear layer mapping to `len(set(dataset.labels))` (i.e., 10) output features.
- **Loss Function and Optimizer**:
    - **Loss**: We employed `nn.CrossEntropyLoss()`, a standard and effective loss function for multi-class classification tasks.
    - **Optimizer**: We utilized `optim.Adam` with an initial learning rate of `1e-4`. Adam is an adaptive learning rate optimizer known for its robustness and efficient convergence properties across a wide range of deep learning problems.

3. Adversarial Training

The core of our approach involved adversarially training the selected ResNet-50 model against FGSM and PGD attacks. This strategy was crucial for building a classifier that performs robustly on both clean and perturbed data.

- **FGSM Attack Implementation (`fgsm_sample_training`)**:
    - This function was implemented to generate adversarial examples using the Fast Gradient Sign Method (FGSM).
    - It computes the gradient of the loss with respect to the input images.
    - The perturbed image is then constructed by adding `epsilon` (a small constant, default 0.01) multiplied by the sign of the gradient to the original image.
    - All pixel values of the perturbed images were clamped to the valid range of [0, 1].
- **PGD Attack Implementation (`pgd_sample_training`)**:
    - We implemented the Projected Gradient Descent (PGD) attack, an iterative and more potent variant of FGSM.
    - The process begins with an initial perturbation (optionally random) and iteratively applies the sign of the gradient, projecting the perturbations back into an `epsilon`-ball (default 0.03) around the original image after each step.
    - `alpha` (default 0.007) served as the step size for each iteration, and `iters` (default 10) defined the number of iterations.
    - The `random_start` parameter was crucial for adding randomness to the initial perturbation, which helped in escaping local optima and improving the attack's effectiveness during adversarial training.
    - Similar to FGSM, all pixel values were clamped to the valid range.

Our `train` function iterated through the data loader, dynamically incorporating both standard and adversarial training. For approximately half of the batches (`torch.rand(1).item() < 0.5`), we randomly decided to generate and use adversarial examples. When adversarial examples were employed, we further randomized the choice between generating FGSM or PGD perturbed images. This diversification of attack types during training was a key strategy to enhance the model's overall robustness against a broader range of adversarial threats.

The standard training steps—zeroing gradients, performing a forward pass, calculating the loss, executing a backward pass, and updating optimizer weights—were consistently applied to both clean and adversarial inputs.Training loss and accuracy were meticulously calculated and returned for monitoring.

4. Hyperparameter Tuning and Checkpointing

To optimize model performance and prevent overfitting, we systematically tuned hyperparameters and implemented a robust checkpointing strategy.

- **Hyperparameter Tuning**: We conducted experiments by varying key hyperparameters, including the number of `epochs`, `batch_size`, and `learning_rate`. This iterative process allowed us to identify configurations that yielded the best balance between clean accuracy and adversarial robustness.
- **Checkpointing and Overfitting Prevention**: Our training loop included a crucial checkpointing mechanism. From epoch 10 onwards, we saved the model's state dictionary after each epoch. This allowed us to capture the model's performance at various stages of training. By continuously printing and monitoring both training accuracy and test accuracy, we were able to identify the "elbow curve" – the point where training accuracy continued to increase but test accuracy began to decrease. This critical observation helped us prevent overfitting, as we could then select the model checkpoint from an earlier epoch that demonstrated optimal generalization performance on unseen data, effectively balancing learning with avoiding memorization of the training set.

5. Results Model Export

Used the epoch_17 results as it had the highest accuracy. The final deliverable for this assignment was a PyTorch `*.pt` file containing the model's state dictionary and its class name.

- **Saving Model State**: We used `torch.save(model.state_dict(), f"resnet50_epoch{epoch+1}.pt")` to save only the learned parameters of the model. This strictly adhered to the submission requirements, which explicitly requested the model's state dictionary rather than the entire model instance.

Challenges and Considerations :-

In developing our model, we had to constantly manage the inherent trade-off between *robustness*—the model's ability to resist and perform well on adversarial examples—and *accuracy* on clean, unperturbed data. Typically, enhancing a model's robustness through techniques like adversarial training can lead to a slight decline in its performance on standard inputs, and vice versa. This is a well-known phenomenon in machine learning, where improving resistance to adversarial attacks often compromises the model's ability to generalize to clean data. To address this, we adopted a carefully designed adversarial training strategy that incorporates both clean and adversarial samples during training. This hybrid approach allowed the model to learn to perform well across both domains. Our goal was to achieve an optimal balance—maximizing robustness without sacrificing too much accuracy on clean data, and ensuring that the model remains reliable and effective in real-world applications where both clean and adversarial inputs may be encountered.

References & Implementations :-

1. A Survey of Robust Adversarial Training in Pattern Recognition: Fundamental, Theory, and Methodologies (Qian et al., 2022)
2. A structured, thorough overview of adversarial training—defining fundamentals, unified theories, taxonomy of methods, and open challenges.
3. Adversarial Training: A Survey (Zhao et al., Oct 2024) A recent, broad review addressing implementations via data augmentation, network design, training strategies, and future directions.
4. Recent Advances in Adversarial Training for Adversarial Robustness (Bai et al., IJCAI 2021). Categorizes progress in adversarial training and generalization issues while mapping out future research avenues.
5. Bag of Tricks for Adversarial Training (Pang et al., 2020) Reveals how minor hyperparameter choices, like weight decay and training schedules, can have large effects—changing robust accuracy by over 7%
6. Revisiting Adversarial Training at Scale (Bai et al., 2024) Applies adversarial training to modern architectures (e.g. vision transformers) and highlights scalable, efficient strategies