CS434 - Computer Security

Final Project

Adversarial Attacks for Improving Robustness of Image Classifiers

22125003 - Trần Công Lâm Anh

22125032- Trần Quang Huy

22125090 - Nguyễn Ngọc Duy Tân

22125112 - Dương Thanh Triều

Table of Content:

1. Introduction

Deep learning has become the foundation of modern computer vision, enabling state-of-the-art performance across a wide range of tasks such as image classification, object detection, and semantic segmentation. Despite these successes, neural networks are highly vulnerable to **adversarial examples**—inputs that have been perturbed by small, carefully crafted noise that is almost imperceptible to humans but can drastically change the model's prediction. For example, adding an adversarial perturbation to an image of a *tiger* might cause a classifier to predict *trucks* with high confidence. This vulnerability poses a serious risk to deploying deep learning models in real-world applications, especially in safety-critical domains such as autonomous driving, medical imaging, and security systems.

The existence of adversarial examples has led to a rapidly growing research area on **adversarial robustness**: how to measure it, how to attack models, and how to defend against such attacks. Among many proposed defense strategies, **adversarial training** has emerged as one of the most practical and effective methods. In adversarial training, the model is trained not only on clean images but also on adversarially perturbed ones, thereby learning to classify correctly even in the presence of malicious perturbations. While adversarial training is conceptually simple, it raises important trade-offs: models may become more robust, but at the potential cost of reduced accuracy on clean data, and training can become more computationally expensive.

This project, titled **"Adversarial Training for Improving Robustness of Image Classifiers"**, investigates these issues in the context of the **CIFAR-100 dataset** using a modern lightweight backbone, **MobileNetV3-Small**. The project is motivated by the following research questions:

- *How vulnerable is a cleanly trained MobileNetV3-Small to adversarial perturbations on CIFAR-100?*
- *To what extent does adversarial training improve robustness against attacks such as FGSM?*
- *What is the trade-off between clean accuracy and adversarial robustness in this setting?*

By conducting a controlled comparison between a baseline model (trained only on clean data) and an adversarially trained model (trained with a 50% mix of clean and adversarial examples), this project aims to provide empirical

evidence for the benefits and limitations of adversarial training. The results will highlight whether a lightweight model such as MobileNetV3 can achieve robustness gains without sacrificing efficiency or clean accuracy, and will provide insights that could inform future work on deploying robust deep learning models in resource-constrained environments.

2. Project scope

This project is focused on studying adversarial robustness in deep learning by implementing, training, and evaluating models under both clean and adversarial training regimes. The goal is to determine how adversarial training affects classification performance and robustness against adversarial attacks, specifically on the CIFAR-100 dataset.

2.1 Objectives

- **Baseline Measurement**: Train a standard MobileNetV3-Small classifier on clean CIFAR-100 data and evaluate its performance on clean and adversarial test sets.
- **Defense Evaluation**: Implement adversarial training using the **Fast Gradient Sign Method (FGSM)** to improve model robustness.
- **Comparative Analysis**: Compare the performance of the baseline and adversarially trained models across:
    - Clean accuracy
    - Adversarial accuracy
    - Per-class vulnerability
    - Visual examples of adversarial success/failure cases
- **Visualization**: Provide qualitative insights by visualizing:
    - CIFAR-100 dataset samples
    - Preprocessing and augmentation pipeline
    - Adversarial perturbations and their effect on predictions

2.2 Dataset

- **Dataset CIFAR-100**: The CIFAR-100 is a benchmark dataset for image classification comprising 60,000 32x32 color images spread across 100 classes, with each class containing 500 training and 100 testing images.
- **Data loader**:
    - Data loading and preprocessing are implemented in ***cifar100_data.py*** via the function

get_cifar100_dataloaders. The dataset loading and configuration are defined as:

```python
# cifar100_data.py
@dataclass
class CIFAR100Config:
    data_dir: str = "./data"
    img_size: int = 224             # Always resize to 224 as requested
    batch_size: int = 256
    num_workers: int = 2            # 2 on Colab, 4 for Kaggle P100
    val_split: float = 0.0          # set >0 to split train into train/val
    seed: int = 1337
    # Augmentations
    randaugment: bool = True
    randaugment_N: int = 2
    randaugment_M: int = 9
    random_erasing_p: float = 0.25
    # MixUp/CutMix (requires timm)
    use_mixup: bool = False
    mixup_alpha: float = 0.2
    cutmix_alpha: float = 1.0
    mixup_prob: float = 1.0
    # Norm: use ImageNet stats to match ImageNet-pretrained backbones
    mean: Tuple[float, float, float] = (0.485, 0.456, 0.406)
    std: Tuple[float, float, float] = (0.229, 0.224, 0.225)
    # Dataloader niceties
    pin_memory: bool = True
    persistent_workers: bool = True
    # Evaluation
    drop_last_train: bool = True
```

- Preprocessing pipeline includes **resizing, normalization, data augmentation (RandAugment, horizontal flip, random erasing)**.

2.3 Model

- **Backbone**: MobileNetV3-Small
  - Implemented in ***models/mobilenetv3.py***

```python
# models/mobilenetv3.py
def build_mobilenetv3_small(num_classes=100, pretrained=True):
    try:
        from torchvision.models import MobileNet_V3_Small_Weights
        weights = MobileNet_V3_Small_Weights.IMAGENET1K_V1 if pretrained else None
        model = mobilenet_v3_small(weights=weights)
    except Exception:
        model = mobilenet_v3_small(pretrained=pretrained)

    in_feats = model.classifier[-1].in_features
    model.classifier[-1] = nn.Linear(in_feats, num_classes)
    return model
```

    - Pretrained on ImageNet for improved initialization
    - Lightweight and computationally efficient architecture with depthwise separable convolutions and squeeze-and-excitation modules
- **Output**: Modified *classifier head* to produce logits for 100 CIFAR-100 classes

2.4 Adversarial Attack

- **Method**: Fast Gradient Sign Method (FGSM)
    - Implemented in ***`attacks/fgsm.py`***

```python
# attacks/fgsm.py
@torch.enable_grad()
def fgsm_attack(model, images, targets, eps=2/255.0):
    """
    Perform FGSM in *normalized* space.
    Inputs are already normalized by dataset transforms.
    We clamp per-channel to the valid normalized range of [0,1] after Normalize.
    """
    images = images.detach().clone().requires_grad_(True)
    logits = model(images)
    if _is_soft(targets):
        # soft CE
        logp = torch.log_softmax(logits, dim=-1)
        loss = -(targets * logp).sum(dim=-1).mean()
    else:
        loss = F.cross_entropy(logits, targets)

    model.zero_grad(set_to_none=True)
    loss.backward()
    adv = images + eps * images.grad.sign()
    # No absolute clamp to [0,1] because we are in normalized space.
    # Just prevent runaway: clamp to images' min/max range (safe + simple).
    lo = images.amin(dim=(2,3), keepdim=True)
    hi = images.amax(dim=(2,3), keepdim=True)
    adv = torch.max(torch.min(adv, hi), lo)
    return adv.detach()
```

    - Perturbs input image x by: $x_{adv} = x + \epsilon \cdot sign(\nabla x L(f(x), y))$
    - Default perturbation strength: ε = 2/255 (imperceptible to humans, but highly effective against clean models)
- **Purpose**:
    - Used both for evaluating robustness of the baseline model and generating adversarial samples during adversarial training.

2.5 Training Pipeline

- **Baseline Training** (***`scripts/train_baseline.py`***)

- Model trained only on clean data
- Optimizer: AdamW, learning rate = 1e-3, weight decay = 0.02
- Scheduler: CosineAnnealingLR
- Mixed Precision (AMP) used for faster training

```python
# scripts/train_baseline.py
optimizer = AdamW(model.parameters(), lr=args.lr, weight_decay=args.weight_decay)
scheduler = CosineAnnealingLR(optimizer, T_max=args.epochs)

scaler = None if args.no_amp or device == "cpu" else torch.amp.GradScaler("cuda")
use_mixup = (cfg.use_mixup and _HAS_TIMM)
loss_hard, loss_soft = make_losses(use_mixup, has_timm_soft_ce=_HAS_TIMM)

for e in range(args.epochs):
    t0 = time.time()
    train_loss = train_epoch(model, train_loader, optimizer, device, scaler, loss_hard, loss_soft)
    scheduler.step()
    dt = time.time() - t0
    clean = evaluate_clean(model, test_loader, device)
    adv   = evaluate_fgsm(model, test_loader, device, eps=args.eps)
    print(f"[Epoch {e+1}/{args.epochs}] loss={train_loss:.4f}  "
        f"clean@1={clean['top1']:.2f} clean@5={clean['top5']:.2f}  "
        f"fgsm@1={adv['top1']:.2f} fgsm@5={adv['top5']:.2f}")
```

- **Adversarial Training** (***`scripts/train_adv.py`***)
  - Model trained on a mix of clean and adversarial examples (controlled by **`--adv_ratio`**, default = 0.5)
  - Adversarial samples crafted with FGSM at **ε = 2/255**
  - Resumes from baseline checkpoint if available, otherwise starts from ImageNet-pretrained weights (In this project, we init init the adversarial training with the **ImageNet-pretrained** weights)
  - Hyperparameters same as baseline for fair comparison

```python
# scripts/train_adv.py
optimizer = AdamW(model.parameters(), lr=args.lr, weight_decay=args.weight_decay)
scheduler = CosineAnnealingLR(optimizer, T_max=args.epochs_adv)

scaler = None if args.no_amp or device == "cpu" else torch.amp.GradScaler("cuda")
use_mixup = (cfg.use_mixup and _HAS_TIMM)
loss_hard, loss_soft = make_losses(use_mixup, has_timm_soft_ce=_HAS_TIMM)

# --- Training loop
for e in range(args.epochs_adv):
    t0 = time.time()
    train_loss = train_epoch_adv(
        model, train_loader, optimizer, device, scaler,
        loss_hard=loss_hard, loss_soft=loss_soft,
        eps=args.eps, adv_ratio=args.adv_ratio
    )
    scheduler.step()
    dt = time.time() - t0
    clean = evaluate_clean(model, test_loader, device)
    adv   = evaluate_fgsm(model, test_loader, device, eps=args.eps)
    print(f"[Adv Epoch {e+1}/{args.epochs_adv}] loss={train_loss:.4f}  "
          f"clean@1={clean['top1']:.2f} clean@5={clean['top5']:.2f}  "
          f"fgsm@1={adv['top1']:.2f} fgsm@5={adv['top5']:.2f}")
```

2.6 Evaluation

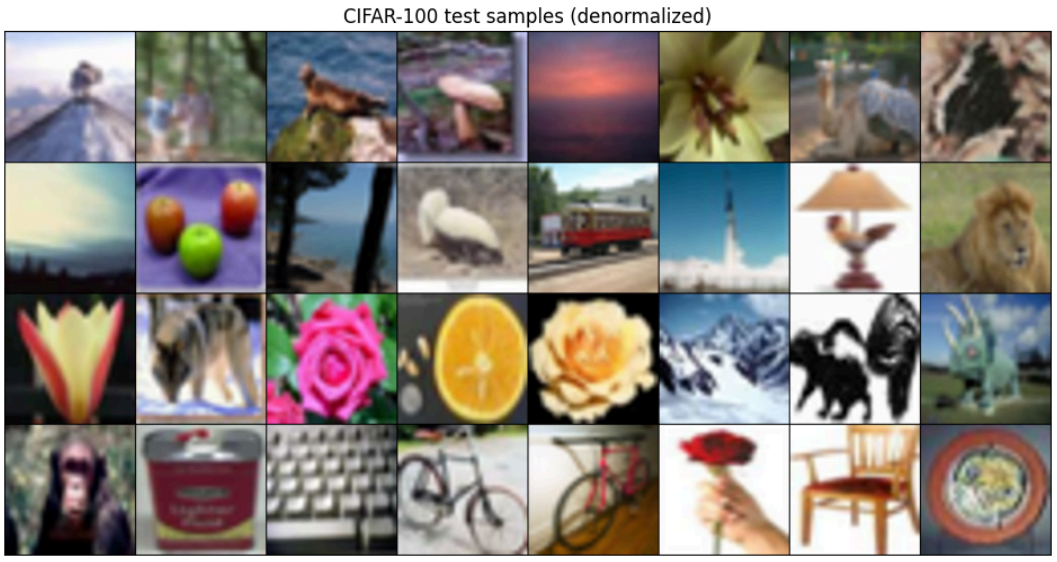
- Implemented in ***`engine/evaluate.py`***
- Metrics:
    - **Top-1 Accuracy**: percentage of correct top predictions
    - **Top-5 Accuracy**: percentage of samples where true label is in top-5 predictions
- Models are tested under two conditions:
    - **Clean evaluation** (unaltered test set)
    - **Adversarial evaluation** (FGSM perturbed test set)
- Additional analysis in ***`report_visual.ipynb`***:
    - Visualization of adversarial perturbations and their effect
    - ε-sweep plots (accuracy degradation with increasing ε)
    - Per-class accuracy curves comparing baseline vs adversarial training

3. Preprocessing and Training Methodology

3.1 Dataset Statistics

We use the **CIFAR-100 dataset**, which consists of:

- 50,000 training images, 10,000 test images
- 100 fine-grained classes grouped into 20 superclasses
    - The detailed 100 classes are: apple, aquarium_fish, baby, bear, beaver, bed, bee, beetle, bicycle, bottle, bowl, boy, bridge, bus, butterfly, camel, can, castle, caterpillar, cattle, chair, chimpanzee, clock, cloud, cockroach, couch, crab, crocodile, cup, dinosaur, dolphin, elephant, flatfish, forest, fox, girl, hamster, house, kangaroo, keyboard, lamp, lawn_mower, leopard, lion, lizard, lobster, man, maple_tree, motorcycle, mountain, mouse, mushroom, oak_tree, orange, orchid, otter, palm_tree, pear, pickup_truck, pine_tree, plain, plate, poppy, porcupine, possum, rabbit, raccoon, ray, road, rocket, rose, sea, seal, shark, shrew, skunk, skyscraper, snail, snake, spider, squirrel, streetcar, sunflower, sweet_pepper, table, tank, telephone, television, tiger, tractor, train, trout, tulip, turtle, wardrobe, whale, willow_tree, wolf, woman, worm
- Each class contains 500 training and 100 test samples
- Image size: 32×32 RGB (upsampled to 224×224 for compatibility with pretrained ImageNet models)



In this project, images are resized to 224×224 to match the input size of **ImageNet-pretrained MobileNetV3-Small**.

3.2 Data Preprocessing and Augmentation

Data preprocessing is defined in ***cifar100_data.py***, in the function *build_transforms*.
Below is the exact transform pipeline:

```python
# cifar100_data.py
def build_transforms(cfg: CIFAR100Config):
    # Always upsample to 224; use bicubic for quality
    train_tfms = [
        transforms.Resize(cfg.img_size, interpolation=InterpolationMode.BICUBIC),
        transforms.RandomHorizontalFlip(p=0.5),
    ]
    if cfg.randaugment:
        # torchvision RandAugment works well; N=2, M=9 default is stable
        train_tfms.append(transforms.RandAugment(num_ops=cfg.randaugment_N, magnitude=cfg.randaugment_M))

    train_tfms.extend([
        transforms.ToTensor(),
        transforms.Normalize(mean=cfg.mean, std=cfg.std),
    ])

    # RandomErasing happens after normalization
    if cfg.random_erasing_p > 0:
        train_tfms.append(transforms.RandomErasing(p=cfg.random_erasing_p, value='random'))

    train_tfms = transforms.Compose(train_tfms)

    # Eval: strict resize + center crop (optional); CIFAR is small, so just Resize is fine.
    eval_tfms = transforms.Compose([
        transforms.Resize(cfg.img_size, interpolation=InterpolationMode.BICUBIC),
        transforms.ToTensor(),
        transforms.Normalize(mean=cfg.mean, std=cfg.std),
    ])

    return train_tfms, eval_tfms
```

Transformation used:

- **Resize (224×224)**: Upscales CIFAR-100 to match pretrained backbones.
- **Random Horizontal Flip**: Improves invariance to orientation.
- **RandAugment (N=2, M=9)**: Applies random color/geometry augmentations.
- **Random Erasing (p=0.25)**: Simulates occlusion robustness.
- **Normalization**: Uses ImageNet mean/std:
    - mean = (0.485, 0.456, 0.406)
    - std = (0.229, 0.224, 0.225)

Illustration:

- **Train sample**:

- [train 1] original (32×32)
- [train 2] resize 224
- [train 3] randaugment N=2 M=9
- [train 4] toTensor + normalize
- [train 5] random erasing (forced for demo)

- **Evaluation sample:**

- [eval 1] toTensor + resize + normalize

3.3 Loss Functions

- Losses are defined in ***losses/losses.py*** two cases are considered:

```python
# losses/losses.py
def is_soft(t: torch.Tensor) -> bool:
    return t.dtype.is_floating_point

class SoftTargetCrossEntropy(nn.Module):
    def forward(self, logits, targets):
        # targets: probabilities
        logp = F.log_softmax(logits, dim=-1)
        return -(targets * logp).sum(dim=-1).mean()

def make_losses(mixup_enabled: bool, has_timm_soft_ce: bool):
    """
    Returns (loss_for_hard, loss_for_soft)
    """
    hard = nn.CrossEntropyLoss()
    if mixup_enabled and has_timm_soft_ce:
        try:
            from timm.loss import SoftTargetCrossEntropy as TimmSoftCE
            soft = TimmSoftCE()
        except Exception:
            soft = SoftTargetCrossEntropy()
    else:
        soft = SoftTargetCrossEntropy()
    return hard, soft
```

- **Hard Loss (CrossEntropyLoss):** Used when labels are integers (standard supervised learning).
- **Soft Loss (SoftTargetCrossEntropy):** Used when labels are probability distributions, such as after **MixUp** or **CutMix**. The implementation computes cross-entropy between the predicted logits and soft targets.
- **MixUp Support:** If `timm` is available and MixUp is enabled, the project uses *`timm.loss.SoftTargetCrossEntropy`*, otherwise falls back to the local implementation.

Notice: This modular design makes the training code automatically pick the correct loss depending on whether MixUp or CutMix is active.

3.4 Training Methodology

Training logic is implemented in ***engine/trainers.py*** and reused by both ***train_baseline.py*** and ***train_adv.py***

Baseline training:

```python
# engine/trainers.py
def train_epoch(model, loader, optimizer, device, scaler, loss_hard, loss_soft):
    """Standard epoch with tqdm. Shows running loss."""
    model.train()
    total_loss, total = 0.0, 0
    pbar = tqdm(loader, desc="train", leave=False)
    for x, y in pbar:
        x, y = x.to(device, non_blocking=True), y.to(device, non_blocking=True)

        optimizer.zero_grad(set_to_none=True)
        with torch.amp.autocast(device_type="cuda", enabled=(scaler is not None)):
            logits = model(x)
            loss = (loss_soft if is_soft_labels(y) else loss_hard)(logits, y)

        if scaler is not None:
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
        else:
            loss.backward()
            optimizer.step()

        bs = x.size(0)
        total_loss += loss.item() * bs
        total += bs
        pbar.set_postfix(loss=f"{total_loss/max(1,total):.4f}")
    return total_loss / max(1, total)
```

- Uses **mixed precision (AMP)** for efficiency.
- Chooses **hard vs. soft loss** automatically based on target type.

Adversarial training:

```python
# engine/trainers.py
def train_epoch_adv(model, loader, optimizer, device, scaler,
                    loss_hard, loss_soft, eps=2/255.0, adv_ratio=0.5):
    """
    Mix clean + FGSM per batch (shown with tqdm).
    """
    from attacks.fgsm import fgsm_attack

    model.train()
    total_loss, total = 0.0, 0
    pbar = tqdm(loader, desc="adv-train", leave=False)

    for x, y in pbar:
        x, y = x.to(device, non_blocking=True), y.to(device, non_blocking=True)

        # Build adversarial copy using hard labels
        y_hard = to_hard(y)
        x_adv = fgsm_attack(model, x, y_hard, eps=eps)

        if adv_ratio <= 0.0:
            mix_x, mix_y = x, y
        elif adv_ratio >= 1.0:
            mix_x, mix_y = x_adv, y
        else:
            n = x.size(0)
            n_adv = int(round(n * adv_ratio))
            mix_x = torch.cat([x_adv[:n_adv], x[n_adv:]], dim=0)
            mix_y = torch.cat([y[:n_adv], y[n_adv:]], dim=0)

        optimizer.zero_grad(set_to_none=True)
        with torch.amp.autocast(device_type="cuda", enabled=(scaler is not None)):
            logits = model(mix_x)
            loss = (loss_soft if is_soft_labels(mix_y) else loss_hard)(logits, mix_y)

        if scaler is not None:
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
        else:
            loss.backward()
            optimizer.step()

        bs = mix_x.size(0)
        total_loss += loss.item() * bs
        total += bs
        pbar.set_postfix(loss=f"{total_loss/max(1,total):.4f}")

    return total_loss / max(1, total)
```

- Splits each batch into clean and adversarial subsets.
- Adversarial examples generated via FGSM at ε = 2/255.
- Controlled by `adv_ratio` (default = 0.5).

3.5 Training Setup

Scripts orchestrating training:

- **Baseline** → ***`scripts/train_baseline.py`***
- **Adversarial** → ***`scripts/train_adv.py`***

Both rely on:

- **AdamW optimizer**
- **Cosine Annealing LR scheduler**
- **AMP mixed precision**
- **Batch size** = 256
- **Weight decay** = 0.05
- **Learning rate** = 5e-4

Hardware used:

- **Google Colab** (NVIDIA T4 GPU)

Training result:

- **Baseline training:**

```
# Baseline training
[Epoch 1/10] loss=1.6386  clean@1=68.63 clean@5=92.71  fgsm@1=12.76 fgsm@5=52.97
[Epoch 2/10] loss=0.9494  clean@1=67.10 clean@5=91.65  fgsm@1=10.52 fgsm@5=45.65
[Epoch 3/10] loss=0.7733  clean@1=74.73 clean@5=95.07  fgsm@1=14.13 fgsm@5=56.51
[Epoch 4/10] loss=0.6552  clean@1=77.43 clean@5=95.62  fgsm@1=17.39 fgsm@5=63.31
[Epoch 5/10] loss=0.5472  clean@1=77.27 clean@5=95.31  fgsm@1=16.86 fgsm@5=63.23
[Epoch 6/10] loss=0.4548  clean@1=78.92 clean@5=96.08  fgsm@1=19.38 fgsm@5=66.05
[Epoch 7/10] loss=0.3859  clean@1=80.43 clean@5=96.58  fgsm@1=19.20 fgsm@5=66.32
[Epoch 8/10] loss=0.3284  clean@1=82.10 clean@5=97.18  fgsm@1=21.54 fgsm@5=70.65
[Epoch 9/10] loss=0.2898  clean@1=82.98 clean@5=97.16  fgsm@1=20.56 fgsm@5=69.98
[Epoch 10/10] loss=0.2706  clean@1=82.97 clean@5=97.25  fgsm@1=20.93 fgsm@5=70.62
```

- **Adversarial training:**

```
# Adversarial training
[Adv Epoch 1/10] loss=2.2521  clean@1=66.64 clean@5=92.05  fgsm@1=52.34 fgsm@5=85.16
[Adv Epoch 2/10] loss=1.3590  clean@1=71.13 clean@5=93.61  fgsm@1=58.32 fgsm@5=88.41
[Adv Epoch 3/10] loss=1.1295  clean@1=71.74 clean@5=93.92  fgsm@1=58.43 fgsm@5=88.87
[Adv Epoch 4/10] loss=0.9864  clean@1=71.19 clean@5=92.86  fgsm@1=57.66 fgsm@5=87.89
[Adv Epoch 5/10] loss=0.8603  clean@1=66.35 clean@5=90.44  fgsm@1=54.41 fgsm@5=84.45
[Adv Epoch 6/10] loss=0.7527  clean@1=45.58 clean@5=75.13  fgsm@1=35.13 fgsm@5=66.82
[Adv Epoch 7/10] loss=0.6645  clean@1=77.86 clean@5=95.46  fgsm@1=64.68 fgsm@5=91.55
[Adv Epoch 8/10] loss=0.5935  clean@1=77.35 clean@5=95.36  fgsm@1=65.09 fgsm@5=91.55
[Adv Epoch 9/10] loss=0.5376  clean@1=80.28 clean@5=96.56  fgsm@1=68.66 fgsm@5=93.66
[Adv Epoch 10/10] loss=0.5134  clean@1=80.32 clean@5=96.65  fgsm@1=68.94 fgsm@5=93.60
```

4. Evaluation and Results

The evaluation focuses on comparing the **baseline model** (trained on clean CIFAR-100 data) and the **adversarially trained model** (trained with 50% FGSM samples, ε=2/255). We measure performance on both **clean test images** and **adversarially perturbed test images**.

4.1 Metrics

Evaluation is based on **Top-1** and **Top-5 Accuracy**:

- **Top-1 Accuracy**: proportion of test samples where the model's most confident prediction matches the ground truth.
- **Top-5 Accuracy**: proportion of test samples where the ground truth label appears in the top-5 predictions.

The computation is implemented in ***`engine/trainers.py`***:

```python
# engine/trainers.py
def topk_accuracy(logits, targets, topk=(1,)):
    """Return list of top-k accuracies in % for the given k values."""
    maxk = max(topk)
    _, pred = logits.topk(maxk, dim=1, largest=True, sorted=True)
    pred = pred.t()
    correct = pred.eq(targets.view(1, -1).expand_as(pred))
    res = []
    for k in topk:
        correct_k = correct[:k].reshape(-1).float().sum(0).item()
        res.append(100.0 * correct_k / targets.size(0))
```

4.2 Quantitative Results:

```
=== Results (Top-1/Top-5 %) ===
Model                       Clean@1 Clean@5 FGSM@1 FGSM@5
MobileNetV3 Baseline         82.97   97.25   20.93   70.62
MobileNetV3 Adv.Trained      80.32   96.65   68.94   93.60
```

Observations:

- The baseline model performs reasonably well on clean data but collapses under FGSM attack.

- Adversarial training reduces **clean accuracy slightly** but provides a **massive gain in adversarial robustness**.

4.3 Visualization of Predictions:

4.3.2 ε-sweep montage

We visualize the clean sample under multiple perturbation strengths ε ∈ {0, 1/255, 2/255, 4/255}.

4.3.2 Adversarial Samples

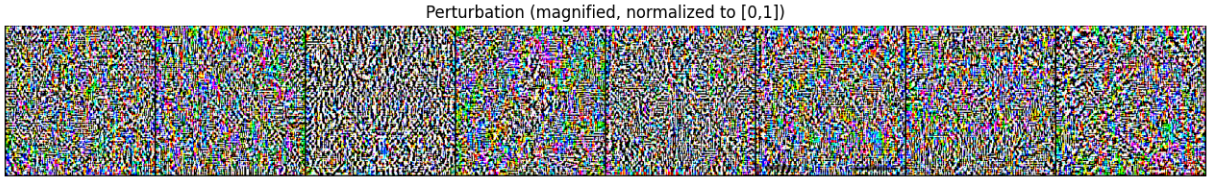
We visualize the effect of adversarial perturbations. Using FGSM (ε=2/255), we compare **clean input**, **adversarial input**, and the **perturbation (magnified for visibility)**.

Adversarial samples

Clean (denorm)

Adversarial FGSM ε=0.00784313725490196 (denorm)

Perturbation (magnified, normalized to [0,1])



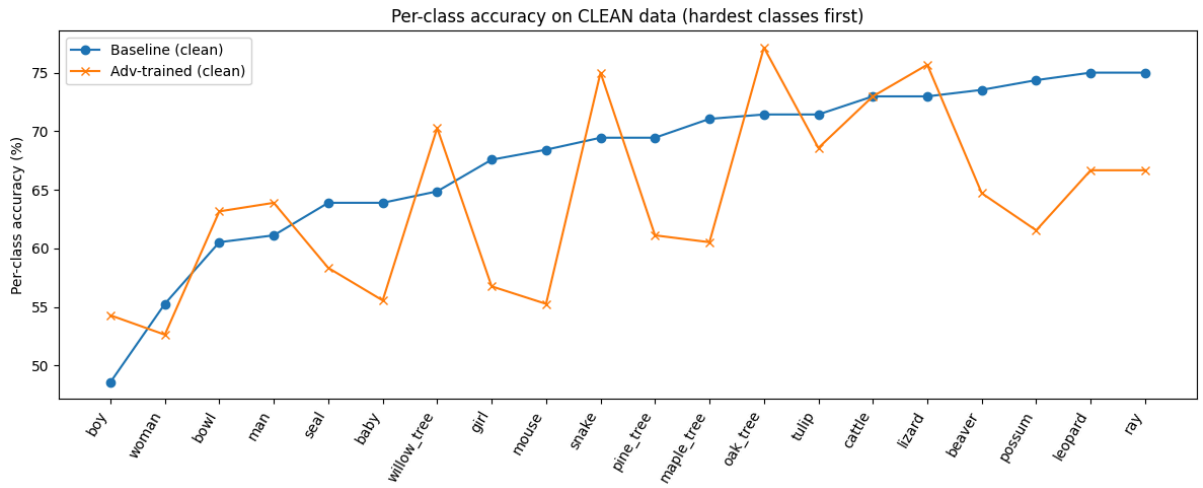
Classified result of the two models on these samples:

```
[0] GT: forest
    Baseline on CLEAN:  pred=forest              conf=0.999
    Baseline on ADV:    pred=willow_tree         conf=0.565
    Adv-trained on ADV: pred=forest              conf=0.938
----------------------------------------------------------------------
[1] GT: mushroom
    Baseline on CLEAN:  pred=mushroom            conf=1.000
    Baseline on ADV:    pred=trout               conf=0.717
    Adv-trained on ADV: pred=mushroom            conf=1.000
----------------------------------------------------------------------
[2] GT: sea
    Baseline on CLEAN:  pred=sea                 conf=0.930
    Baseline on ADV:    pred=cloud               conf=0.575
    Adv-trained on ADV: pred=sea                 conf=0.828
----------------------------------------------------------------------
[3] GT: butterfly
    Baseline on CLEAN:  pred=butterfly           conf=0.969
    Baseline on ADV:    pred=skunk               conf=0.999
    Adv-trained on ADV: pred=butterfly           conf=0.679
----------------------------------------------------------------------
```

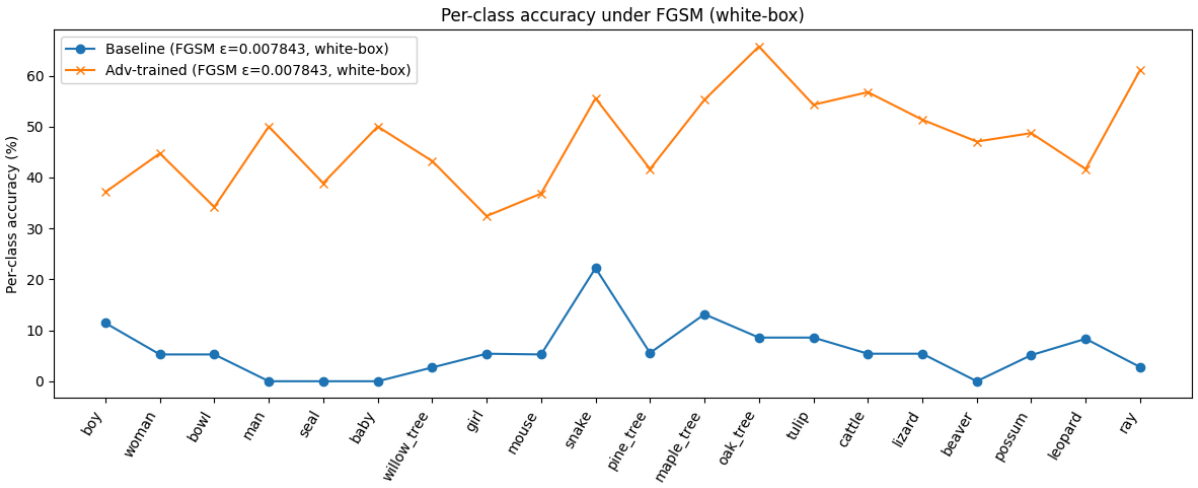
```
[4] GT: apple
    Baseline on CLEAN:  pred=apple              conf=0.637
    Baseline on ADV:    pred=sweet_pepper       conf=0.997
    Adv-trained on ADV: pred=apple              conf=0.594
----------------------------------------------------------------------
[5] GT: skunk
    Baseline on CLEAN:  pred=skunk              conf=0.819
    Baseline on ADV:    pred=mushroom           conf=0.811
    Adv-trained on ADV: pred=skunk              conf=0.967
----------------------------------------------------------------------
[6] GT: streetcar
    Baseline on CLEAN:  pred=train              conf=0.798
    Baseline on ADV:    pred=train              conf=0.997
    Adv-trained on ADV: pred=streetcar          conf=0.509
----------------------------------------------------------------------
[7] GT: lion
    Baseline on CLEAN:  pred=lion               conf=1.000
    Baseline on ADV:    pred=cattle             conf=0.767
    Adv-trained on ADV: pred=lion               conf=1.000
----------------------------------------------------------------------
```

4.3.3 Per-Class Accuracy

We computed per-class accuracy for both models on **clean** and **adversarial** data (**ε=2/255**). The 20 hardest classes for the baseline are plotted below:



Overall mean (clean): baseline=84.21% | adv-trained=80.11%



Overall mean (FGSM white-box ε=0.007843): baseline=20.70% | adv-trained=68.63%

Findings:

Clean Data Performance

- On clean CIFAR-100 test images, the **baseline model** achieves a higher *overall mean Top-1 accuracy* (84.21%) compared to the **adversarially trained model** (80.11%).
- For most of the **hardest classes** (e.g., *boy, woman, bowl, baby, willow_tree*), the baseline is slightly more consistent, while the adversarial model shows larger variance — occasionally outperforming strongly (e.g., *snake, oak_tree*) but underperforming on others (*seal, baby, beaver*).
- This reflects the typical trade-off: adversarial training improves robustness but can reduce performance on clean data for some fine-grained categories.

Adversarial Data (FGSM, white-box ε=2/255)

- Under attack, the **baseline collapses**, dropping to a mean Top-1 accuracy of only **20.70%**, with many classes (e.g., *seal, baby, beaver*) near **0% accuracy**.
- The **adversarially trained model** remains substantially more robust, with a mean accuracy of **68.63%**. Across all the hardest classes, it consistently outperforms the baseline, often by margins of **+30–60 percentage points** (e.g., *snake, oak_tree, tulip*).
- This demonstrates that adversarial training not only improves average robustness but also prevents catastrophic failures in the most vulnerable classes.

Key Contrast

- **Baseline**: Stronger clean accuracy overall, but extremely brittle — even imperceptible perturbations cause near-total failure.
- **Adversarially trained**: Slightly lower clean accuracy, but **much more reliable under attack**, especially in vulnerable fine-grained categories.

5. Conclusion

This project explored the use of adversarial training to improve the robustness of image classifiers, focusing on **MobileNetV3-Small** trained on the **CIFAR-100** dataset. Through a controlled comparison between a baseline model (trained only on clean data) and an adversarially trained model (trained with a 50% mix of clean and FGSM-perturbed examples at ε=2/255), several important insights were obtained.

First, the results confirmed the vulnerability of cleanly trained models: although the baseline achieved strong performance on unperturbed test images (Top-1 ≈ 84.21%), its performance collapsed under adversarial attack, dropping to just 20.70% in white-box FGSM evaluation. Second, adversarial training significantly improved robustness. While it slightly reduced clean accuracy (Top-1 ≈ 80.11%), it increased adversarial accuracy more than threefold, reaching 68.63% under FGSM attack. This trade-off demonstrates the effectiveness of adversarial training as a defense: a small reduction in clean accuracy yields a large gain in robustness.

Visual analyses reinforced these findings. Perturbations crafted with FGSM were nearly invisible yet caused the baseline to misclassify confidently, while the adversarially trained model frequently maintained the correct prediction. Per-class analysis showed that adversarial training not only improved average robustness but also prevented catastrophic failures in the most vulnerable categories, stabilizing performance across fine-grained classes.

In conclusion, adversarial training is a practical and effective defense strategy for improving the robustness of lightweight classifiers such as **MobileNetV3-Small**. While it introduces a modest reduction in clean accuracy, the robustness gain against adversarial perturbations makes it a valuable technique for applications where reliability is critical. Future work could extend this study by evaluating against stronger multi-step attacks (e.g., PGD, AutoAttack), experimenting with alternative defense methods (e.g., TRADES, adversarial weight perturbation), and benchmarking across different model

architectures to further explore the trade-off between clean accuracy, robustness, and efficiency.

6. Appendix:

The repo for the project available at repo, included:

- All the source code for constructing the experiment and training.
- The training script for both model
- The model weight of the Baseline and Adversarial training model.
- The requirements.txt for the environment set up.