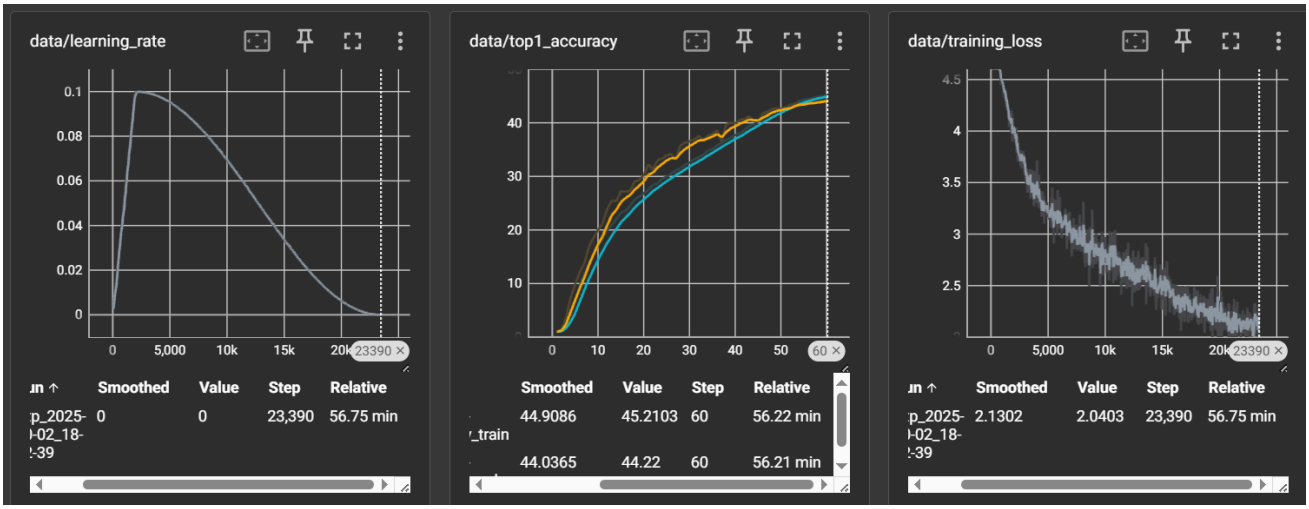
comp771 hw2

3.1.2 Convolution

Original CNN

Training curves:

- Learning Rate:
  The learning rate follows a warm-up + cosine decay schedule. It first **increases gradually** from 0 to around 0.1 during the early iterations (warm-up phase), then **decreases smoothly** toward zero following a cosine decay pattern as training progresses (up to step ≈ 23k).
- Top-1 Accuracy:
  The top-1 accuracy consistently improves across epochs, reaching about **45%** by epoch 60, indicating steady model convergence.
- Training Loss:
  The training loss drops from roughly **4.5 to 2.1**, showing smooth and stable optimization without divergence.



Evaluation result:
At epoch 60, the model achieves:

- Top-1 accuracy (Acc@1): 44.22%

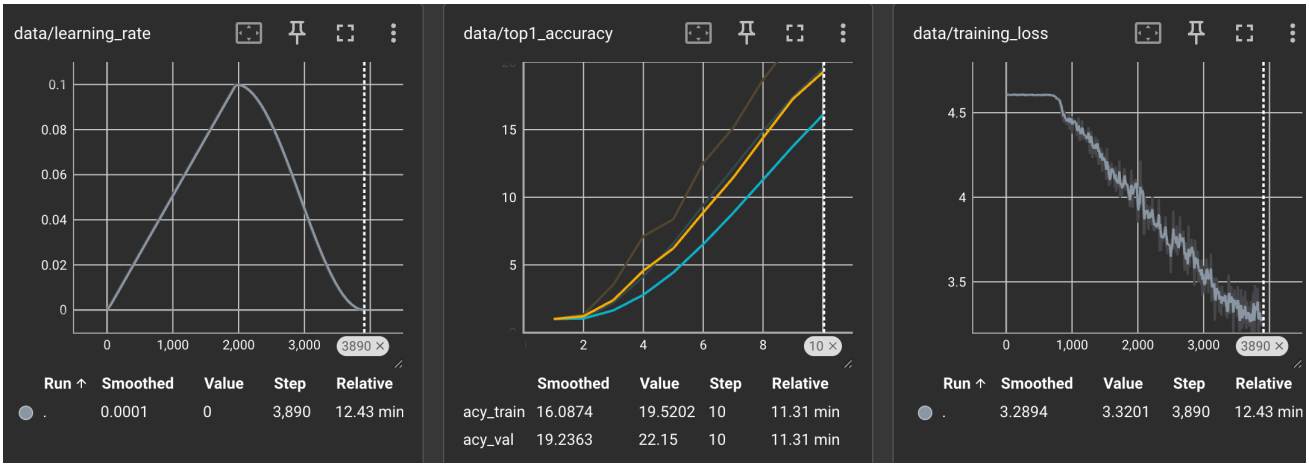
- Top-5 accuracy (Acc@5): 73.39%

```
5.23)    Acc@5  75.78  (74.32)
[Training]: Epoch 60 finished with lr=0.000000
Test: [0/100]    Time 9.328 (9.328)      Acc@1 46.00 (46.00)      Acc@5 81.00 (81.00)
Test: [10/100]   Time 0.011 (0.868)      Acc@1 41.00 (47.91)      Acc@5 75.00 (77.00)
Test: [20/100]   Time 0.074 (0.469)      Acc@1 37.00 (45.05)      Acc@5 75.00 (74.90)
Test: [30/100]   Time 0.011 (0.325)      Acc@1 41.00 (44.48)      Acc@5 72.00 (74.65)
Test: [40/100]   Time 0.074 (0.253)      Acc@1 43.00 (44.49)      Acc@5 62.00 (74.29)
Test: [50/100]   Time 0.011 (0.208)      Acc@1 42.00 (44.63)      Acc@5 69.00 (73.84)
Test: [60/100]   Time 0.071 (0.179)      Acc@1 38.00 (44.33)      Acc@5 74.00 (73.90)
Test: [70/100]   Time 0.015 (0.157)      Acc@1 41.00 (44.32)      Acc@5 70.00 (73.56)
Test: [80/100]   Time 0.081 (0.141)      Acc@1 47.00 (44.48)      Acc@5 82.00 (73.64)
Test: [90/100]   Time 0.011 (0.128)      Acc@1 38.00 (44.33)      Acc@5 72.00 (73.58)
******Acc@1 44.220 Acc@5 73.390
```

Custom CNN

Training curves:

- Learning Rate:
  The learning rate follows a warm-up + cosine decay schedule. It first **increases gradually** from 0 to around 0.1 during the early iterations (warm-up phase), then **decreases smoothly** toward zero following a cosine decay pattern as training progresses (up to step ≈ 4k).
- Top-1 Accuracy:
  The top-1 accuracy consistently improves across epochs, reaching about **20%** by epoch 60, indicating steady model convergence.
- Training Loss:
  The training loss drops from roughly **4.5 to 3.3**, showing smooth and stable optimization without divergence.

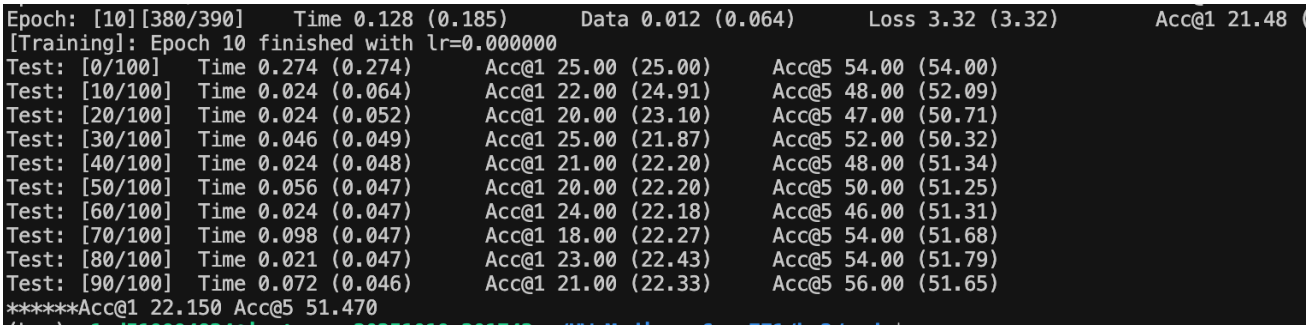


Evaluation result:

At epoch 10, the model achieves:

- Top-1 accuracy (Acc@1): 22.15%

- Top-5 accuracy (Acc@5): 51.47%



Comparison with torch version

Memory Consumption:

The custom convolution implementation consumes significantly more GPU memory than PyTorch's native version. This is because the custom implementation uses unfold to expand the input feature map and performs matrix multiplication via bmm
In contrast, PyTorch's nn.Conv2d leverages low-level C++/CUDA fused kernels (cuDNN kernels), which avoid such intermediate expansions and thus achieve higher memory efficiency.

Training Speed:

The custom implementation trains noticeably slower. The PyTorch version completes **six times more steps in only about 4.5× the time**, thanks to its highly optimized cuDNN convolution kernels that execute fused operations efficiently on GPU Tensor Cores, resulting in significantly higher throughput and computational efficiency.

Improvements of Custom CNN

Model Design

- Stem: A 7×7 convolution layer with stride=2, followed by Batch Normalization (BN) and ReLU activation, and then a 3×3 max pooling layer with stride=2. This stage rapidly reduces the spatial resolution while expanding the receptive field.
- Bottleneck-1 (64→256): A sequence of 1×1 reduction → 3×3 spatial convolution → 1×1 expansion, where each convolution is followed by BN and ReLU. A 3×3 max pooling layer is added at the end to further downsample the feature map.
- Bottleneck-2 (256→512): Follows the same 1×1 → 3×3 → 1×1 bottleneck design, with BN and ReLU after each convolution.
- Head: A global adaptive average pooling (GAP) layer followed by a fully connected layer (512→num_classes).

Training Scheme

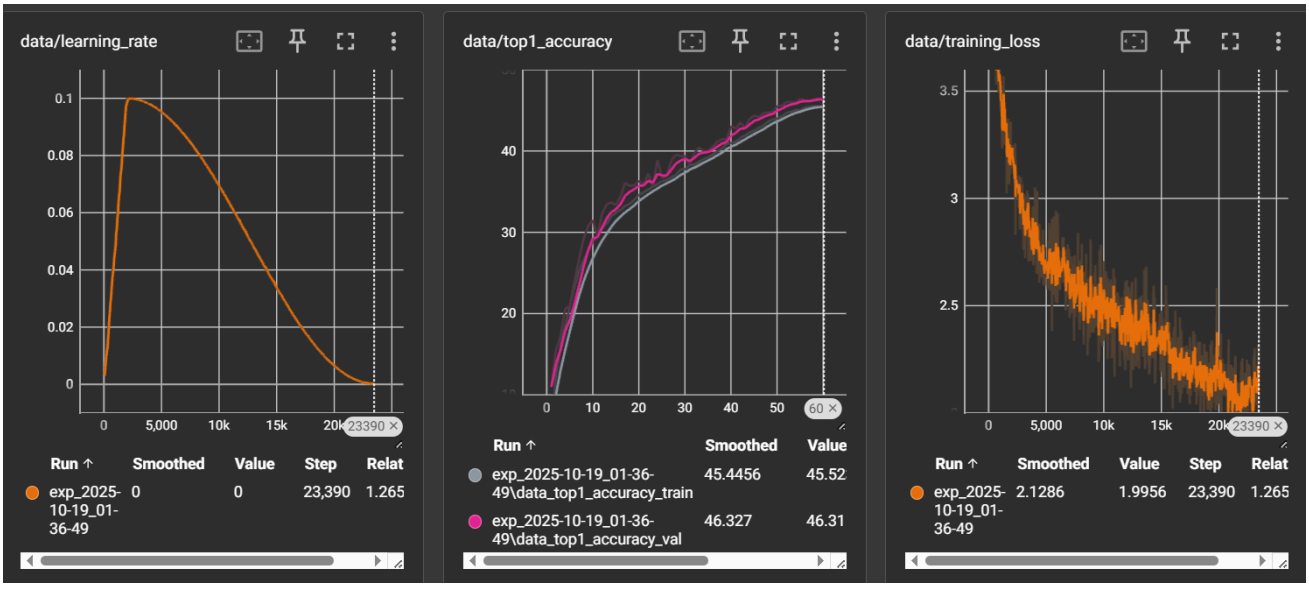
The model was trained with the following hyper-parameter configuration:

- Optimizer: SGD(momentum=0.9)
- Learning Rate Scheduler: cosine decay
- Initial Learning Rate: 0.01
- Weight Decay: 0.05
- Training Epochs: 60
- Batch Size: default

Result

Training curves:

- Learning Rate:
  The learning rate follows a warm-up + cosine decay schedule. It first **increases gradually** from 0 to around 0.1 during the early iterations (warm-up phase), then **decreases smoothly** toward zero following a cosine decay pattern as training progresses (up to step ≈ 23k).
- Top-1 Accuracy:
  The top-1 accuracy consistently improves across epochs, reaching about **45%** by epoch 60, indicating steady model convergence.
- Training Loss:
  The training loss drops from roughly **3.5 to 2.1**, showing smooth and stable optimization without divergence.



Evaluation result:

At epoch 60, the model achieves:

- Top-1 accuracy (Acc@1): 46.31%

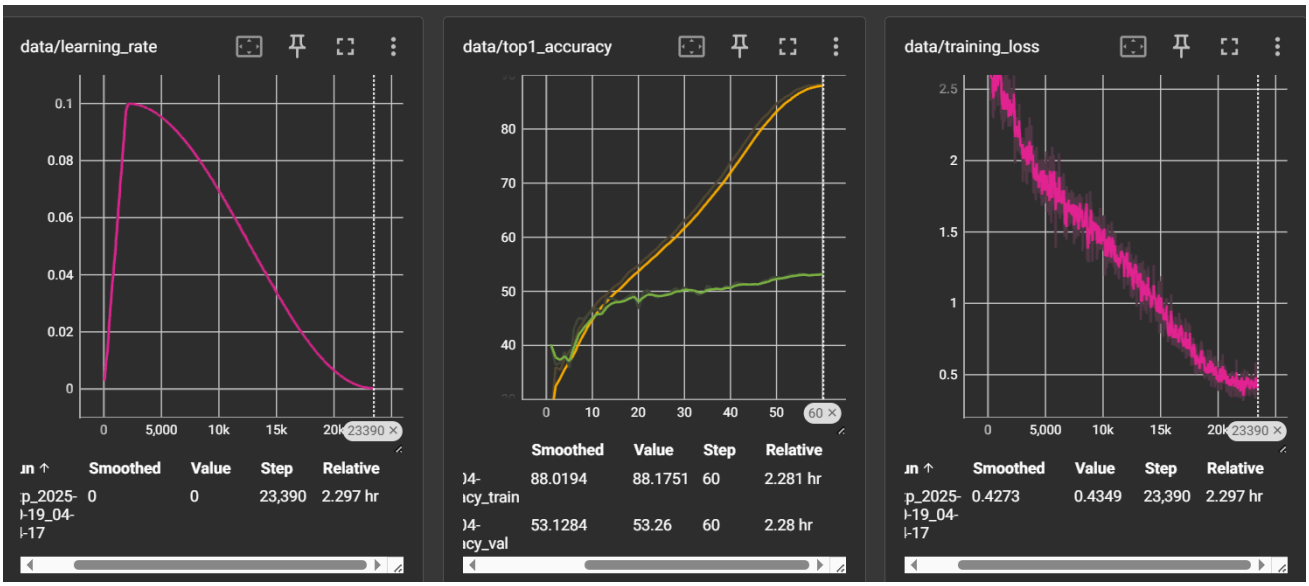
- Top-5 accuracy (Acc@5): 76.08%

```
[Training]: Epoch 60 finished with lr=0.000000
Test: [0/100]   Time 12.562 (12.562)    Acc@1 45.00 (45.00)     Acc@5 80.00 (80.00)
Test: [10/100]  Time 0.016 (1.169)      Acc@1 51.00 (47.64)     Acc@5 77.00 (77.73)
Test: [20/100]  Time 0.062 (0.630)      Acc@1 52.00 (46.71)     Acc@5 79.00 (77.00)
Test: [30/100]  Time 0.019 (0.437)      Acc@1 48.00 (46.19)     Acc@5 77.00 (76.90)
Test: [40/100]  Time 0.043 (0.339)      Acc@1 45.00 (46.41)     Acc@5 72.00 (76.93)
Test: [50/100]  Time 0.016 (0.279)      Acc@1 45.00 (46.45)     Acc@5 70.00 (76.27)
Test: [60/100]  Time 0.062 (0.239)      Acc@1 46.00 (46.41)     Acc@5 71.00 (76.21)
Test: [70/100]  Time 0.020 (0.210)      Acc@1 46.00 (46.61)     Acc@5 70.00 (76.20)
Test: [80/100]  Time 0.069 (0.188)      Acc@1 48.00 (46.64)     Acc@5 83.00 (76.23)
Test: [90/100]  Time 0.020 (0.171)      Acc@1 42.00 (46.49)     Acc@5 82.00 (76.23)
******Acc@1 46.310 Acc@5 76.080
```

ResNet18

Training curves:

- Learning Rate:
  The learning rate follows a warm-up + cosine decay schedule. It first **increases gradually** from 0 to around 0.1 during the early iterations (warm-up phase), then **decreases smoothly** toward zero following a cosine decay pattern as training progresses (up to step ≈ 23k).
- Top-1 Accuracy:
  The top-1 accuracy consistently improves across epochs, reaching about **88%** by epoch 60, indicating steady model convergence.
- Training Loss:
  The training loss drops from roughly **2.5 to 0.4**, showing smooth and stable optimization without divergence.



Evaluation result:

At epoch 60, the model achieves:

- Top-1 accuracy (Acc@1): 53.26%
- Top-5 accuracy (Acc@5): 79.29%

```
[Training]: Epoch 60 finished with lr=0.000000
Test: [0/100]   Time 10.824 (10.824)    Acc@1 46.00 (46.00)     Acc@5 84.00 (84.00)
Test: [10/100]  Time 0.029 (1.014)      Acc@1 52.00 (55.36)     Acc@5 80.00 (82.18)
Test: [20/100]  Time 0.070 (0.547)      Acc@1 55.00 (53.43)     Acc@5 78.00 (80.71)
Test: [30/100]  Time 0.025 (0.380)      Acc@1 56.00 (53.19)     Acc@5 80.00 (80.06)
Test: [40/100]  Time 0.049 (0.296)      Acc@1 50.00 (53.80)     Acc@5 76.00 (80.15)
Test: [50/100]  Time 0.025 (0.244)      Acc@1 52.00 (53.90)     Acc@5 80.00 (79.88)
Test: [60/100]  Time 0.052 (0.209)      Acc@1 46.00 (53.30)     Acc@5 79.00 (79.39)
Test: [70/100]  Time 0.028 (0.184)      Acc@1 50.00 (53.41)     Acc@5 84.00 (79.51)
Test: [80/100]  Time 0.049 (0.166)      Acc@1 57.00 (53.62)     Acc@5 84.00 (79.46)
Test: [90/100]  Time 0.026 (0.151)      Acc@1 47.00 (53.35)     Acc@5 80.00 (79.40)
******Acc@1 53.260 Acc@5 79.290
```

Comparison

The learning rate adopts a warmup strategy at the beginning of training, gradually increasing to its peak, and then follows a cosine decay schedule that smoothly decreases it toward zero, ensuring overall training stability.

In terms of training loss, MySimpleNet shows a slow decrease from approximately **3.6 to 2.2**, with the training accuracy plateauing around **45.45%**, indicating limited representational capacity and clear signs of underfitting.

In contrast, ResNet-18 demonstrates a rapid and consistent drop in training loss to about **0.43**, achieving a training accuracy of **88%**, which reflects stronger feature extraction capability and higher model capacity.

From the validation performance perspective, ResNet-18 exhibits little or slow improvement in validation accuracy during the first few epochs. This is typically observed when pretrained weights are used—initially, the linear classification head adapts to the new dataset while the backbone features remain mostly unchanged. As the learning rate gradually decays, the backbone begins fine-tuning, leading to a steady rise in validation accuracy that eventually saturates around **53%**.

Meanwhile, SimpleNet shows nearly identical training and validation accuracies (**45.5% vs. 46.3%**), with no clear generalization gap, further confirming that the model suffers from underfitting due to its limited capacity.

3.2 ViT

Model Design

The model first uses a **Patch Embedding** layer (implemented via convolution) to divide the input image into non-overlapping patches and map them into embedding vectors. Each patch has an embedding dimension of 192, with a patch size of 16×16. The model then stacks **four**

Transformer blocks in the embedding space, each consisting of a **LayerNorm**, **Multi-Head Self-Attention (MSA)**, and **MLP** submodule.

In the attention mechanism, the configuration window_block_indexes=(0, 2) specifies that the 1st and 3rd blocks use **local window attention** (window_size=4), while the others employ global attention, balancing local dependency modeling and global context understanding.

Local attention is implemented through the window_partition and window_unpartition functions, which split the feature map into smaller windows and compute self-attention within each subregion. This reduces the complexity from $O(T^2D)$ to $O(W^2TD)$, significantly lowering memory consumption.

Additionally, the model incorporates:

- Pre-LayerNorm for improved training stability;
- Stochastic Depth (drop_path) for layer-wise depth regularization;
- Absolute Positional Embedding to enhance spatial awareness;
- GELU activation and Truncated Normal initialization for stable convergence.

Training Scheme

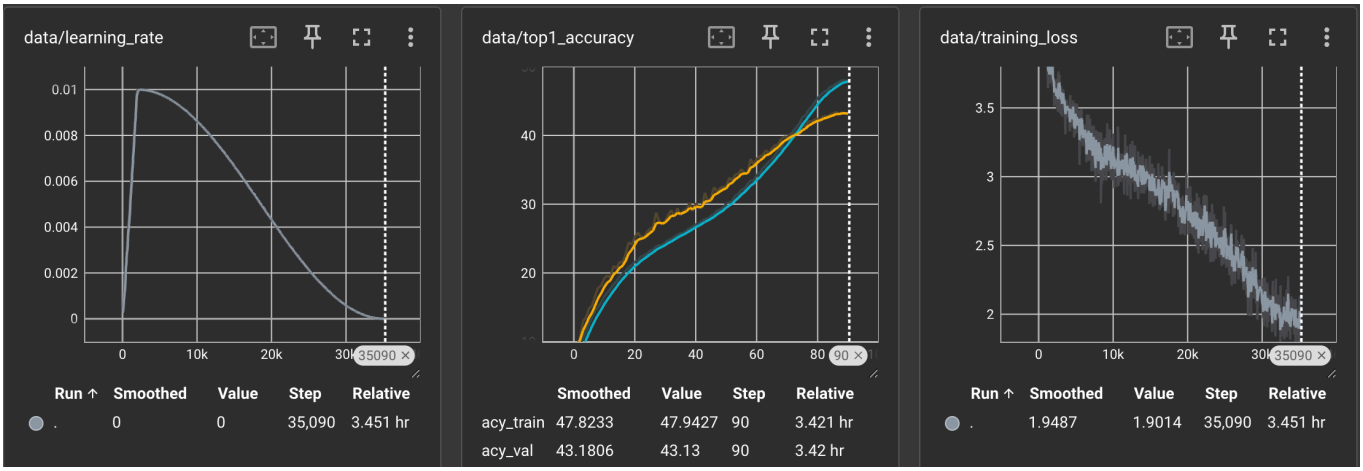
The model was trained with the following hyperparameter configuration:

- **Optimizer:** AdamW
- **Learning Rate Scheduler:** cosine decay
- **Initial Learning Rate:** 0.01
- **Weight Decay:** 0.05
- **Training Epochs:** 90
- **Drop Path Rate:** 0.1
- **Batch Size:** default

A warmup + cosine decay strategy was applied — the learning rate gradually increases during early training for stability, then smoothly decays toward zero to avoid late-stage oscillation.

Compared with convolutional networks, the Transformer requires much longer training time (approximately several times that of SimpleNet) due to its larger number of parameters and reliance on global feature interactions, though its convergence is generally more stable.

Results



1. **Validation accuracy increases steadily:** The model progressively improves its generalization performance during later epochs, with no clear signs of overfitting.
2. **Slower convergence speed:** Compared with SimpleNet, the ViT converges more slowly, which is consistent with the typical behavior of Transformers trained on smaller datasets, where longer optimization is required.
3. **Local attention improves efficiency:** Under the same memory budget, using local window attention (window_size=4) effectively reduces computational cost while maintaining accuracy close to that of global attention.

Overall, compared with the conventional CNN (SimpleNet), the Vision Transformer demonstrates **more stable convergence** and **stronger generalization capability**, but at the expense of **higher computational cost** and **longer training time**.

3.3 Adversarial Samples

In each step, PGDAttack::perturb enables gradient computation on the current adversarial tensor, runs a forward pass to obtain logits, and on the first iteration determines and fixes the least-likely class as the target. It then forms a targeted loss toward that fixed target, back-propagates to obtain the gradient with respect to the adversarial input, and updates the adversarial example by subtracting the step size multiplied by the sign of the gradient (which moves the prediction toward the chosen target). After the update, it projects the perturbation back into the allowed range $[-\epsilon, +\epsilon]$ around the original input using element-wise clamping to enforce the constraint. It then replaces the adversarial tensor with the projected result (detached from the computation graph) to avoid accumulating gradients, and disables gradients until the next iteration. The loop repeats these steps for the specified number of PGD iterations while keeping all operations in the model's (normalized) input space and maintaining the perturbation constraint.

Type	Acc@1	Acc@5
without attack	44.220	73.390
with attack	0.230	4.770

Can you see the difference between the original images and the adversarial samples?

At first glance the adversarial samples look essentially the same as the originals — you won't notice large semantic changes such as objects moving or major color shifts.
If you look closely or magnify the difference, you will see faint high-frequency perturbations: tiny speckled noise, slight local contrast/color shifts, or edge shimmering. Those perturbations are often sub-pixel and spatially distributed, so they're easy to miss in a collage but become visible if you subtract the images and scale the residual, or if you zoom in very close.

What if you increase the number of iterations and reduce the error bound (ε)?

Increasing the number of PGD iterations while reducing the ℓ_∞ bound (ϵ) is a standard way to make attacks stronger yet less visually obvious. Many small, carefully directed steps allow the attack to find a more effective adversarial direction inside a tighter ball, so we often obtain higher attack success while keeping per-pixel changes tiny.

Adversarial Training

We compared the performance of the baseline **SimpleNet** and our adversarially trained model **AdvSimpleNet** under PGD attacks.
After adversarial perturbations were applied, SimpleNet's accuracy dropped drastically to **Top-1: 0.23% / Top-5: 4.77%**, indicating that the model is highly vulnerable to adversarial noise.
In contrast, AdvSimpleNet maintained much higher robustness, achieving **Top-1: 35.53% / Top-5: 63.75%** under the same attack setting.
This demonstrates that adversarial training significantly improves the model's ability to defend against gradient-based perturbations, greatly enhancing robustness at the cost of increased training time.

During adversarial training, we used the following *configuration*:

- Loss function: CrossEntropyLoss
- Number of PGD steps: 4
- Step size: 2.0 / 255.0
- Epsilon (perturbation bound): 8.0 / 255.0

- Training epochs: 70

model design:

1. Backbone:
   The feature extractor shares the same convolutional architecture as **SimpleNet**
   (7×7 / stride=2 convolution → MaxPool → two bottleneck blocks → Global Average Pooling → Fully Connected layer).
2. Mean/Std Buffers:
   The model uses register_buffer to store the **ImageNet mean/std**
   (mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]),
   enabling conversion between **pixel space** and **normalized space** during training.
3. Forward Logic (Training Mode):
   - **De-normalization:** Convert normalized input back to pixel space:
     x_pixel = clamp(x_norm * std + mean, 0, 1)
   - **Model Adapter:** Re-normalize the pixel tensor and feed it into _forward_impl,
     so PGD operates in pixel space while the model performs inference in normalized space.
   - **BN-safe Crafting:** Temporarily switch to eval() when calling attack.perturb()
     to avoid batch normalization statistic corruption, then restore train() mode.
   - **Re-normalization:** Re-normalize the adversarial samples and pass them through the backbone to obtain logits.
4. Inference Mode:
   When in eval(), the attack step is skipped and _forward_impl is used directly,
   ensuring that the inference cost and behavior remain identical to **SimpleNet**.

SimpleNet:

```
Test: [0/100]   Time 1.778 (1.778)      Acc@1 0.00 (0.00)       Acc@5 4.00 (4.00)
Test: [10/100]  Time 0.383 (0.712)      Acc@1 0.00 (0.27)       Acc@5 6.00 (5.45)
Test: [20/100]  Time 0.384 (0.649)      Acc@1 2.00 (0.29)       Acc@5 6.00 (5.10)
Test: [30/100]  Time 0.384 (0.625)      Acc@1 2.00 (0.26)       Acc@5 6.00 (5.10)
Test: [40/100]  Time 0.385 (0.639)      Acc@1 0.00 (0.29)       Acc@5 2.00 (4.80)
Test: [50/100]  Time 0.386 (0.626)      Acc@1 0.00 (0.24)       Acc@5 5.00 (4.71)
Test: [60/100]  Time 0.388 (0.620)      Acc@1 1.00 (0.21)       Acc@5 11.00 (4.72)
Test: [70/100]  Time 0.388 (0.616)      Acc@1 0.00 (0.20)       Acc@5 3.00 (4.73)
Test: [80/100]  Time 0.389 (0.611)      Acc@1 0.00 (0.20)       Acc@5 7.00 (4.77)
Test: [90/100]  Time 0.396 (0.610)      Acc@1 1.00 (0.23)       Acc@5 3.00 (4.76)
******Acc@1 0.230 Acc@5 4.770
```

AdvSimpleNet:

```
Test: [0/100]   Time 1.802 (1.802)      Acc@1 40.00 (40.00)     Acc@5 69.00 (69.00)
Test: [10/100]  Time 0.388 (0.686)      Acc@1 36.00 (38.55)     Acc@5 53.00 (63.82)
Test: [20/100]  Time 0.394 (0.654)      Acc@1 31.00 (36.43)     Acc@5 63.00 (63.43)
Test: [30/100]  Time 0.403 (0.687)      Acc@1 39.00 (36.10)     Acc@5 62.00 (63.58)
Test: [40/100]  Time 0.397 (0.693)      Acc@1 36.00 (36.88)     Acc@5 63.00 (64.05)
Test: [50/100]  Time 0.393 (0.688)      Acc@1 30.00 (36.73)     Acc@5 58.00 (63.73)
Test: [60/100]  Time 0.395 (0.671)      Acc@1 34.00 (36.11)     Acc@5 64.00 (63.85)
Test: [70/100]  Time 0.398 (0.661)      Acc@1 35.00 (36.03)     Acc@5 65.00 (63.93)
Test: [80/100]  Time 0.399 (0.655)      Acc@1 41.00 (36.04)     Acc@5 67.00 (64.01)
Test: [90/100]  Time 0.403 (0.649)      Acc@1 37.00 (35.79)     Acc@5 67.00 (63.87)
******Acc@1 35.530 Acc@5 63.750
```

Contribution

- Bohan Wen: Convolutional Neural Networks
- Handan Hu: Vision Transformers
- Chongwei Liu: Adversarial Samples
- Qinxinghao Chen: Adversarial Training