2025 Visual Recognition Deep Learning – HW 1

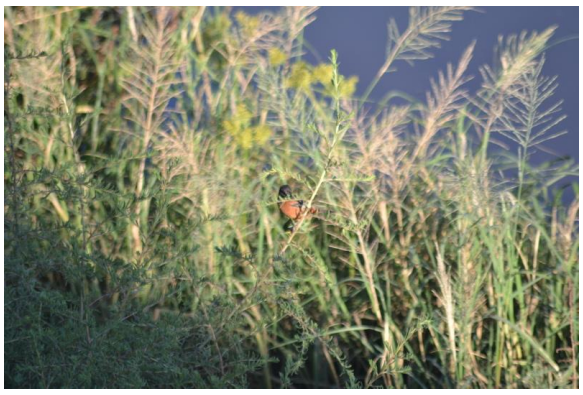
HW1_GithubLink

313553014 廖怡誠

1. Introduction

The task in this homework is image classification. The dataset contains various plants and animals. Some images have complex or unrecognizable backgrounds, as shown in Picture 1, and in some cases, the target object is difficult to locate, which may make the model harder to train and more prone to overfitting. For example, in Picture 1, we can see that the classification target is an orange-and-black bird, but it is partially covered by surrounding grass, making it difficult to recognize. In Picture 2, although there are many leaves and grasses visible, the actual classification target is the pink flower. This image may be intended to show the leaves associated with the flower, but it could confuse the model and prevent it from learning the correct features.

To address this issue, I manually checked all images in the training set and found that, in most cases, the object to be classified is located at the center of the image. Based on this observation, I cropped the borders of the images to reduce the influence of the background. Additionally, to avoid overfitting and improve model performance, I kept the image size as large as possible—up to the limit my GPU could handle—and applied various data augmentation techniques, such as color jitter, random affine transformations, random erasing, and random rotation, to increase the diversity of the training data. Of course, I also experimented with different hyperparameters, optimizers, and learning rate schedulers to further enhance the model's performance. More details will be introduced in the following sections.



Pic 1. Complex background case.

Pic 2. Miss target object case.

2. Method

In this homework, my goal was to improve generalization and reduce overfitting, which I observed during early training experiments where the validation loss was consistently higher than the

training loss. To address this issue, I implemented an extensive set of data augmentation techniques during the preprocessing stage.

For the training data, each image is first resized to 560×560 and then center cropped to 512×512, ensuring uniform input size while maintaining most of the image content. To introduce variability and simulate real-world distortions, I applied several augmentations: random horizontal flipping with a 50% probability, random color jittering to slightly alter brightness, contrast, saturation, and hue, and geometric transformations including random affine translation and random rotation of up to 20 degrees. These spatial and color-based perturbations help the model learn invariant features and reduce reliance on overly specific patterns in the training data. I also applied Random Erasing, which randomly removes rectangular regions from the image with a 50% chance, mimicking occlusions or missing information in the real world and further regularizing the model. After all augmentations, the image is converted to a tensor and normalized using the standard ImageNet mean and standard deviation values to match the distribution of the pretrained backbone. For the validation set, I kept the preprocessing minimal to accurately reflect real-world performance. Each image is resized to 512×512, converted to a tensor, and normalized in the same way as the training data, but without any augmentation.

Pic 3. Results after Data augmentation.

As for the model architecture, I used a **modified version of ResNet-152**, which total parameters are **58,348,708**. During training, I initialized the model with ImageNet pretrained weights to leverage existing visual representations. I then replaced the final fully connected layer with a dropout layer (p=0.5) followed by a linear classification head matching the number of classes in my dataset. The dropout layer is especially important for mitigating overfitting by randomly deactivating neurons during training, thus preventing co-adaptation of features.

```python
class Resnet152(nn.Module):
    def __init__(self, num_classes=100, mode="train"):
        super(Resnet152, self).__init__()

        if mode == "train":
            self.resnet = resnet152(weights=ResNet152_Weights.DEFAULT)
        else:
            self.resnet = resnet152(weights=None)

        self.resnet.fc = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(self.resnet.fc.in_features, num_classes)
        )

    def forward(self, x):
        return self.resnet(x)
```

The training setup uses **stochastic gradient descent (SGD)** with a **learning rate of 0.01**, **momentum of 0.9**, and **weight decay of 5e-4**, all of which are standard but effective choices for large-scale image classification tasks. To adjust the learning rate over time and help the model converge more smoothly, I used a **cosine annealing scheduler** with a maximum number of iterations set to 10. This learning rate schedule gradually reduces the learning rate in a cosine curve, which helps prevent the model from overfitting in later epochs when the gradients become small. Finally, training is conducted using PyTorch's Distributed Data Parallel (DDP) framework with the NCCL backend, allowing efficient scaling across multiple GPUs. I set the **batch size to 16** and used 4 data loading workers per process to balance throughput and memory usage. All these design choices, especially the heavy use of data augmentation, dropout regularization, and learning rate scheduling—were motivated by the need to combat overfitting and improve validation performance.

3. Final & Additional Experimental Result

In addition to data augmentation, the main hyperparameters I adjusted were batch size, image size, and the learning rate scheduler. Increasing the input image size generally led to clearer and more detailed images, which improved the model's performance. However, larger images also significantly increased GPU memory consumption. To compensate for this, I reduced the batch size accordingly. I experimented with several combinations, including (Batch Size, Image Size): (64, 224), (32, 256), (16, 376), and (16, 512).

As for learning rate scheduling, I tested different strategies to help the model escape local minimum during training. Specifically, I experimented with StepLR, CosineAnnealingLR, and ReduceLROnPlateau. Each scheduler controls how the learning rate changes over time, and I observed that a well-chosen schedule could significantly improve convergence and final accuracy. In the following section, I will present and analyze a few selected configurations to illustrate their impact on model performance.

My Final Results

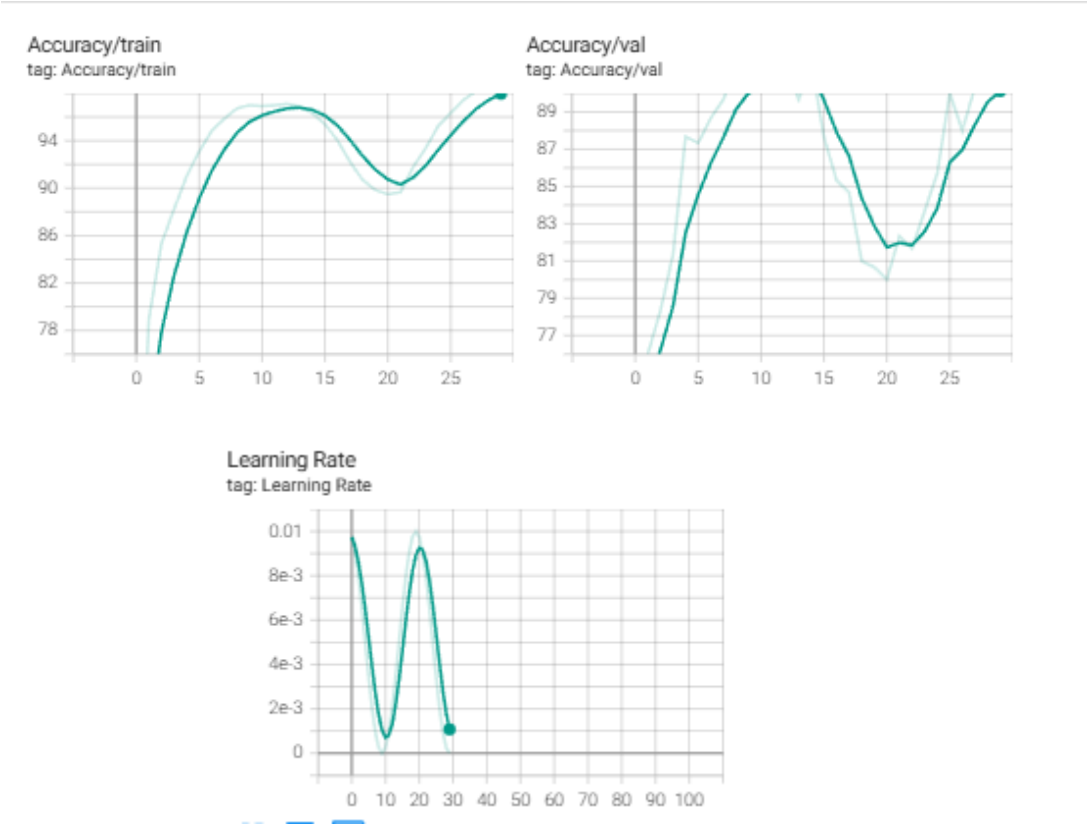
In my final results, I observed that the training accuracy steadily improved over time, reaching a peak around epoch 10. Although there was a slight dip afterward, it picked up again toward the

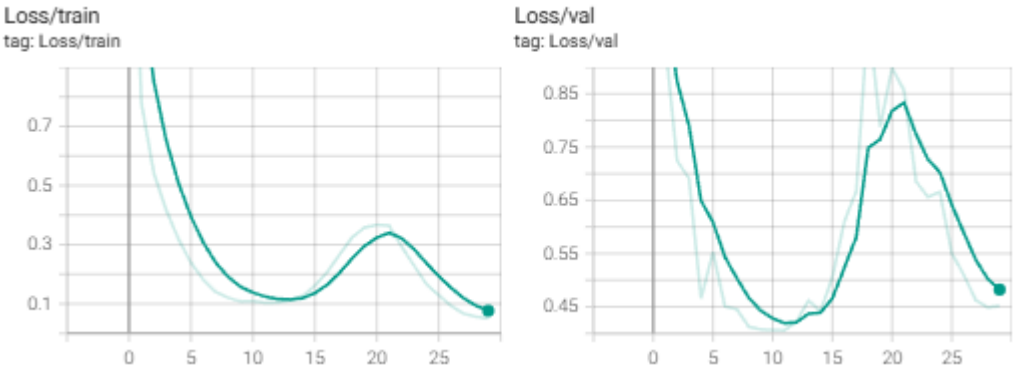
end, suggesting that the model continued to learn effectively. The validation accuracy followed a similar trend but was more fluctuating, which is expected due to the nature of unseen data. Still, the upward trend near the end indicates that the model was able to generalize reasonably well and didn't overfit too severely.

The learning rate plot shows a cyclical pattern, which suggests I used a learning rate scheduler—likely something like the One Cycle Policy or cosine annealing. This strategy helped the model explore a wider solution space in the earlier stages and then refine its learning as the rate decreased. Interestingly, the points where the learning rate dipped seem to align with improvements in both validation accuracy and loss, reinforcing the effectiveness of the scheduling.

When looking at the loss curves, both the training and validation losses dropped sharply in the early epochs. There was a noticeable increase midway, probably corresponding to the phase when the learning rate peaked, but then both losses decreased again toward the end. The final loss values were low and the gap between training and validation loss remained narrow, which indicates that the model not only fit the training data well but also maintained strong generalization.

Overall, I believe this training run was successful. The learning rate schedule played a significant role in navigating the loss landscape effectively, and the model converged with good performance on both the training and validation sets.

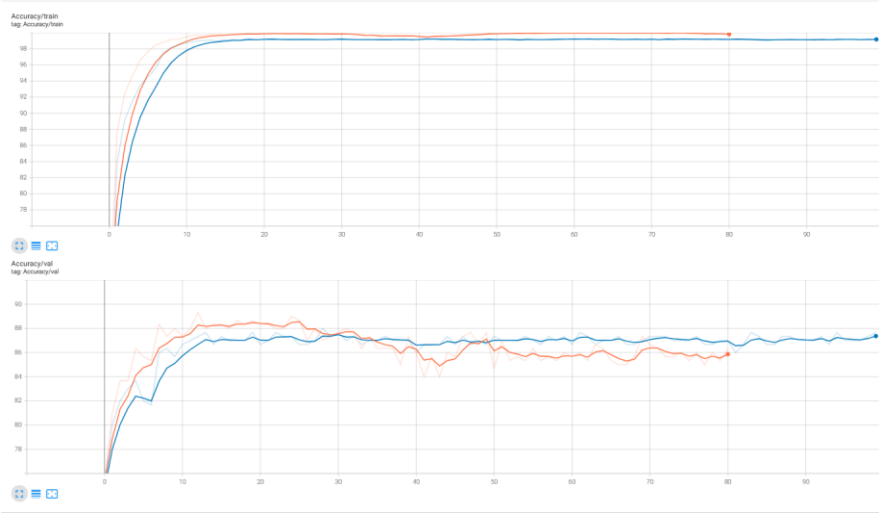




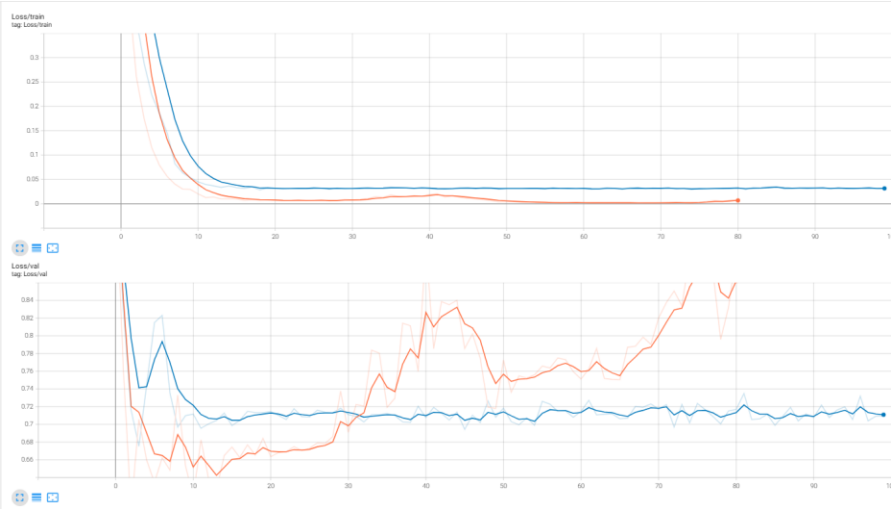
Comparison of different schedulers

When comparing the performance of the CosineAnnealingLR (orange) and ReduceLROnPlateau (blue) schedulers, CosineAnnealingLR achieves stronger results in the early training stages. Specifically, its validation accuracy quickly rises to around 88-89% by epoch 25, outperforming ReduceLROnPlateau, which reaches approximately 86% around the same point. This early advantage is due to the scheduler's cosine-shaped learning rate decay, which encourages rapid convergence during the first phase of training. However, after epoch 40, the validation accuracy for CosineAnnealingLR begins to decline slightly and fluctuates between 84-87%, while ReduceLROnPlateau maintains a more stable trend between 85-86% through the remaining epochs. This behavior is expected from CosineAnnealingLR, as it increases the learning rate again toward the end of training, following the cosine schedule. This change can be beneficial for escaping shallow local minima but may also introduce fluctuations in validation performance if not properly managed.

Looking at the loss curves (Picture 5.), both schedulers reduce the training loss to below 0.02 by the end of training. However, the validation loss under CosineAnnealingLR, after initially dropping below 0.4, starts to climb and reaches around 0.55-0.65, indicating potential instability during the latter phase. In contrast, ReduceLROnPlateau maintains validation loss more consistently around 0.35-0.40 after epoch 20. Overall, CosineAnnealingLR offers better early performance and reaches higher peak validation accuracy, making it ideal when paired with early stopping or checkpointing strategies. ReduceLROnPlateau, on the other hand, provides steadier validation metrics throughout training, making it more reliable in longer training sessions without dynamic intervention.



Pic 4. Training accuracy curve between CosineAnnealingLR (Orange) and ReduceLROnPlateau (Blue).



Pic 5. Training loss curve between CosineAnnealingLR (Orange) and ReduceLROnPlateau (Blue).

Comparison of different image size (resolution)

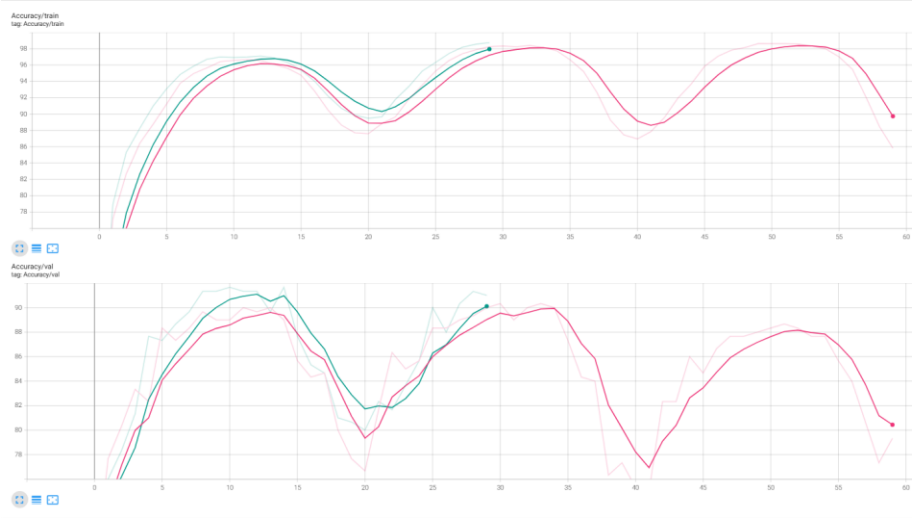
In this experiment, I used CosineAnnealingLR as the learning rate scheduler and fixed the batch size at 16 to investigate the effect of different input image sizes—specifically 376×376 versus 512×512—on model performance.

From the accuracy curves (Picture 6.), we can observe that both configurations reach high training accuracy early in training, but the model trained with 512×512 images achieve better validation accuracy overall. At around epoch 15, the 512×512 setting reaches a peak validation accuracy of

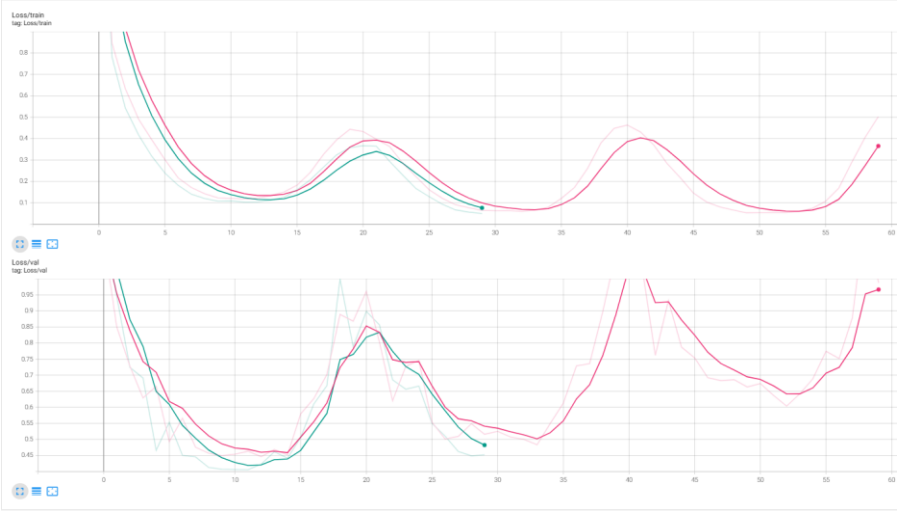
approximately 89%, while the 376×376 setting peaks closer to 86%. Additionally, the model trained on 512×512 images show more stable accuracy during the second half of training, maintaining values above 86%, whereas the 376×376 setting shows a gradual decline, dropping below 84% after epoch 40.

The loss plots (Picture 7.) further reinforce this observation. The validation loss for the 512×512 configuration consistently stays below 0.5 after the first 10 epochs, and dips to as low as 0.35, whereas the 376×376 setting fluctuates more and rises above 0.55 toward the end of training. Training loss is low for both, but the higher image resolution allows the model to learn finer features, resulting in a lower validation loss and improved generalization.

In summary, increasing the input image size from 376×376 to 512×512 led to a noticeable improvement of about 2-3% in validation accuracy and more stable performance across epochs. This confirms that higher-resolution inputs can enhance classification performance, especially when using smaller batch sizes and a scheduler like CosineAnnealingLR that encourages effective early learning.



Pic 6. Training accuracy curve between size-512 (Blue) and size-376 (Pink).

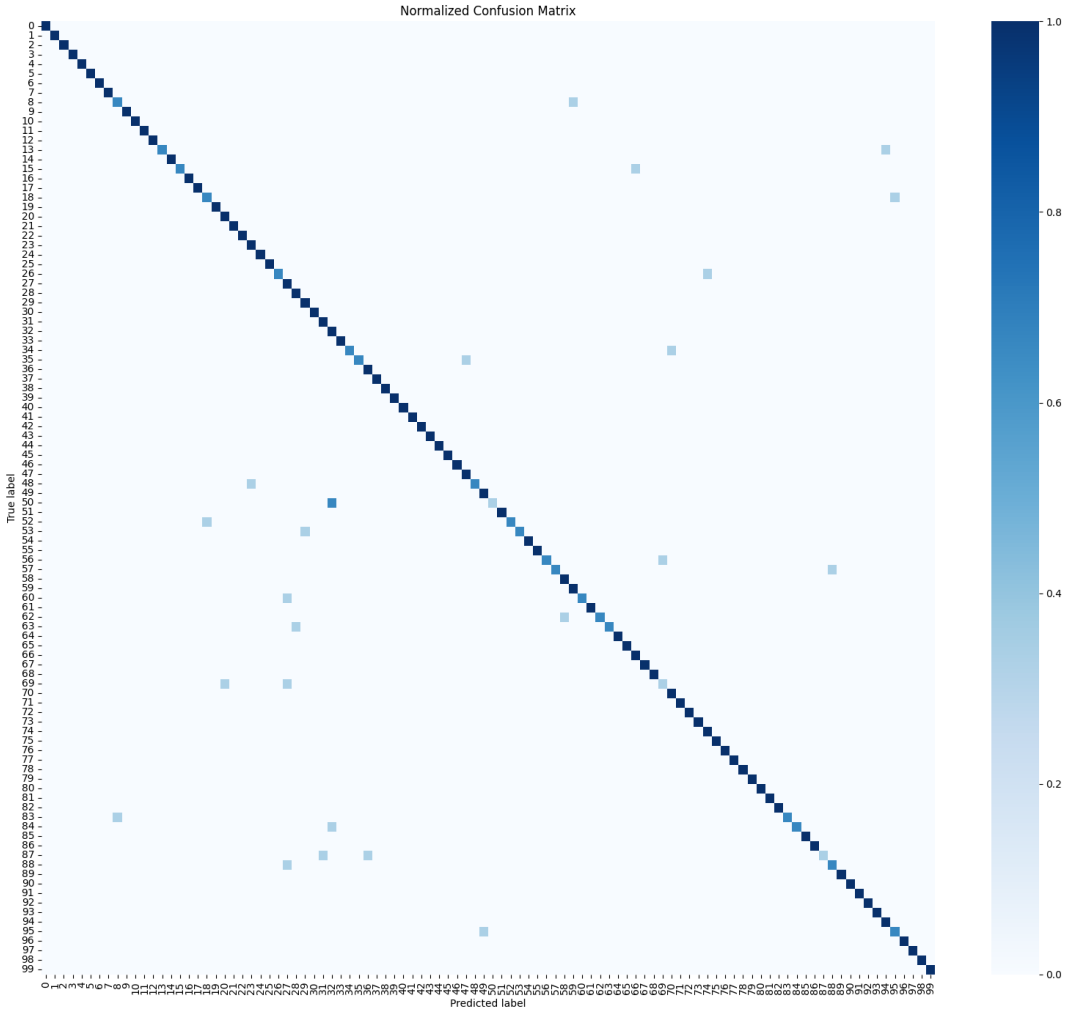


Pic 7. Training loss curve between size-512 (Blue) and size-376 (Pink).

The confusion matrix provides a clear view of the model's classification performance across all 100 categories. Overall, the matrix reveals a strong diagonal trend, indicating that most predictions match the ground truth labels. I use the validation dataset to create the confusion matrix. (Picture 8.) This is consistent with the reported accuracy of 91.67% and macro F1-score of 0.9140, suggesting that the model performs well across the majority of classes.

However, a closer inspection of the confusion matrix highlights several off-diagonal errors, particularly in classes such as 50, 69, and 87. (Picture 9, 10, 11) These are the classes with the lowest per-class F1-scores, and their confusion appears to stem from significant intra-class visual diversity. For example, Class 50 contains images of pinecones and full pine trees from various perspectives, which may have made it difficult for the model to form a consistent representation. Similarly, Class 69 and Class 87 include different plant structures such as berries, vines, flowers, and leaves, often under varying lighting and background conditions. These intra-class variations likely contributed to the observed misclassifications.

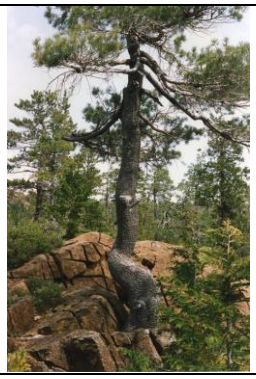
The confusion matrix thus not only confirms the model’s overall strength but also highlights specific areas where performance could be improved. These errors point to challenges in visual consistency and feature learning in certain natural classes, and they suggest potential directions for enhancing the model, such as including more diverse examples during training or leveraging fine-grained visual attention mechanisms.



Pic 8. Confusion matrix with validation dataset.

Accuracy	F1 Score	Recall	Precision
0.9167	0.9140	0.9167	0.9367





Pic 9. Images of Class 50



Pic 10. Images of Class 69



Pic 11. Images of Class 87

4. References

- PyTorch documentation
- ChatGPT