Visual Recognition Using Deep Learning Lab1 多工所碩一 313553024 蘇柏叡

1. Introduction

In this lab, the objective is to perform image classification on a dataset consisting of 100 categories, with the constraint of employing only models from the ResNet family as the backbone architecture. After a comparative evaluation of various ResNet variants, we adopt ResNeSt-101e as our baseline model to surpass. The rationale behind this choice lies in the fact that **ResNeSt-101e incorporates the Split-Attention mechanism**, which enables grouped feature processing within each residual block and dynamically reweights the importance of different branches through attention. This architectural enhancement allows the model to **more effectively capture both local details and global semantic information**, thereby contributing to improved overall performance.

To surpass this strong baseline model, we propose incorporating multi-scale feature fusion within the ResNeSt-101e architecture. The motivation stems from the observation that ResNeSt-101e performs classification primarily based on high-level semantic features extracted from its final layer. Although residual connections can preserve certain information from earlier stages, in practice, such reliance may lead to the loss of local details and mid-level representations. Therefore, in this work, we experiment with two fusion strategies that aim to integrate both shallow and deep features. By doing so, we seek to simultaneously retain fine-grained local cues and global semantic context, thereby enhancing the model's ability to recognize multi-scale objects and discriminate subtle visual patterns.

In addition to designing an effective model architecture and fine-tuning hyperparameters, we first considered whether the dataset itself could offer richer information to the model. Accordingly, in this study, we applied a variety of data augmentation techniques to the training set, including random rotations, random cropping, color jittering, and other transformations. Furthermore, we employed CutMix[1], a strategy that enhances data diversity by mixing both images and their corresponding labels, thereby encouraging the model to learn more generalizable representations. We also conducted a comparative analysis between the **conventional Cross Entropy Loss** and the **Label Smoothing Cross Entropy Loss** to investigate their respective impacts on model performance.

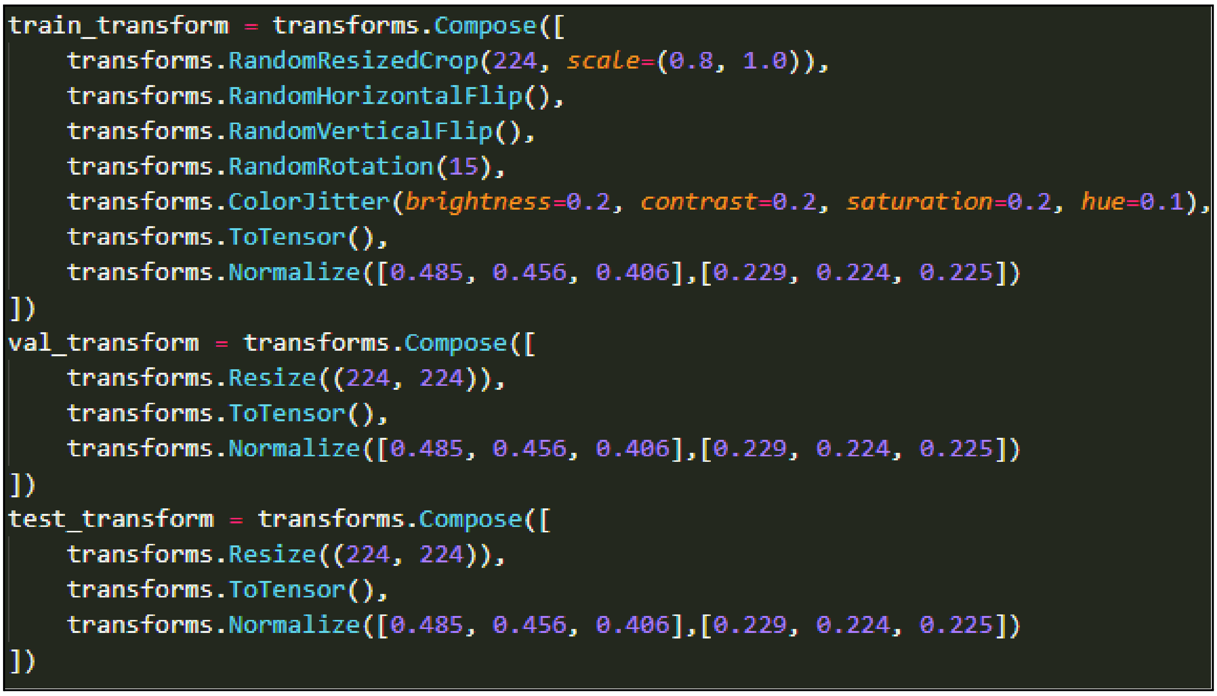
2. Method

2.1 Data Preprocessing

In this lab, all images from the training, validation, and test sets were first resized to a fixed resolution of 224 × 224, followed by normalization using the statistical parameters of the ImageNet dataset. We then applied various data augmentation techniques to the training set to enhance model generalization. Specifically, we employed random horizontal and vertical flips, random resized cropping, random image rotations (±15 degrees), and ColorJitter. In addition, we incorporated the CutMix augmentation strategy[1], which operates by randomly cutting a region from one image and pasting a corresponding region from another image to form a new composite image. The associated labels are also linearly mixed in proportion to the area of the cut region. This technique enables the model to learn from blended samples and improves its robustness. The corresponding implementation is illustrated in Fig. 1.

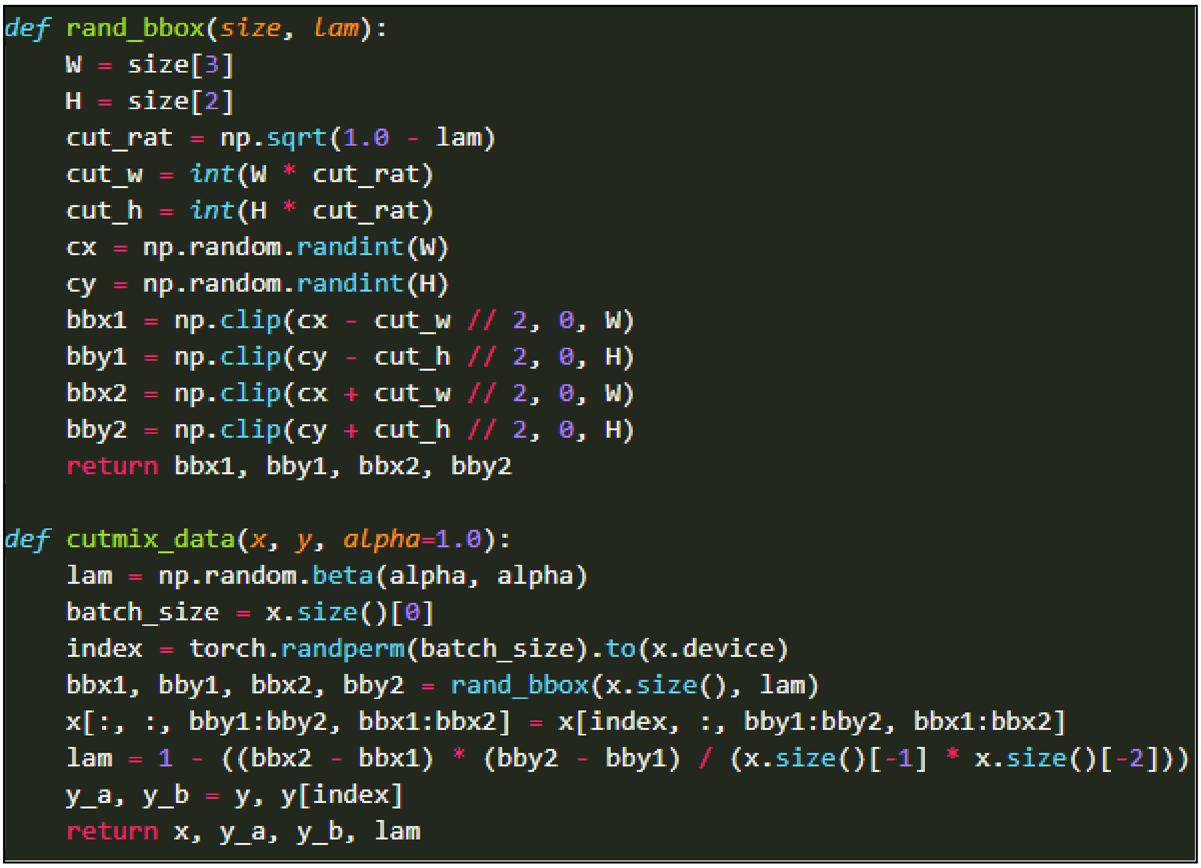
Furthermore, our experiments indicated that **applying the aforementioned data augmentation techniques alone was insufficient to prevent overfitting.** To mitigate this issue, we explored additional augmentation strategies such as Mixup and CutMix. In this lab, we ultimately adopted CutMix to complement the existing augmentation methods. CutMix operates by “cutting” a rectangular region from one image and “pasting” it onto a corresponding region of another image, thereby generating a mixed training sample. Simultaneously, the labels of the two original images are combined in a weighted manner, proportional to the area of the cut region. This allows the model to be exposed to richer and partially occluded visual content during training, enabling it to learn more diverse features and reducing the risk of overfitting, thereby improving generalization.

Regarding the parameter settings, we set CUTMIX_ALPHA to 1.0, indicating that the mixing ratio for the cut region is sampled from a Beta distribution. Meanwhile, CUTMIX_PROB was set to 0.5, meaning that CutMix augmentation is applied to 50% of the mini-batches during training. This probabilistic application further enhances data diversity and helps to regularize the model. The implementation details are shown in Fig. 2.



```python
train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])
])
val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])
])
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])
])
```

Fig 1. The data augmentation of Training, Validation, and Testing datasets



```python
def rand_bbox(size, lam):
    W = size[3]
    H = size[2]
    cut_rat = np.sqrt(1.0 - lam)
    cut_w = int(W * cut_rat)
    cut_h = int(H * cut_rat)
    cx = np.random.randint(W)
    cy = np.random.randint(H)
    bbx1 = np.clip(cx - cut_w // 2, 0, W)
    bby1 = np.clip(cy - cut_h // 2, 0, H)
    bbx2 = np.clip(cx + cut_w // 2, 0, W)
    bby2 = np.clip(cy + cut_h // 2, 0, H)
    return bbx1, bby1, bbx2, bby2

def cutmix_data(x, y, alpha=1.0):
    lam = np.random.beta(alpha, alpha)
    batch_size = x.size()[0]
    index = torch.randperm(batch_size).to(x.device)
    bbx1, bby1, bbx2, bby2 = rand_bbox(x.size(), lam)
    x[:, :, bby1:bby2, bbx1:bbx2] = x[index, :, bby1:bby2, bbx1:bbx2]
    lam = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (x.size()[-1] * x.size()[-2]))
    y_a, y_b = y, y[index]
    return x, y_a, y_b, lam
```

Fig 2. The implementation of CutMix operations

2.2 Model Architecture

2.2.1 The Architecture of ResNeSt-101e

Since ResNeSt is essentially an extension of the ResNet architecture, our selected backbone—ResNeSt-101e—maintains the same depth as the original ResNet-101, with identical stage-wise configurations (refer to Fig. 5 for architectural details). The primary distinction lies in the substitution of the conventional residual blocks with the Split-Attention blocks. Specifically, in each residual block, the input features are divided into k cardinal groups for multi-path feature extraction, and each group is further split into r sub-paths. These sub-paths are then adaptively fused using an attention mechanism, known as Split-Attention, which enables the model to dynamically reweight different feature paths.

The overall ResNeSt framework begins with several convolutional layers (e.g., stacked 3×3 convolutions or other ResNet-D variants) that perform early-stage feature extraction and downsampling. The network is then partitioned into multiple stages according to its depth (101 layers), where each stage contains multiple stacked Split-Attention blocks. Downsampling is conducted at transition points between stages. Finally, global average pooling is employed to aggregate spatial information, and a fully connected layer serves as the final classification head. This design enables ResNeSt to effectively capture both local details and global semantic context throughout the image classification process.

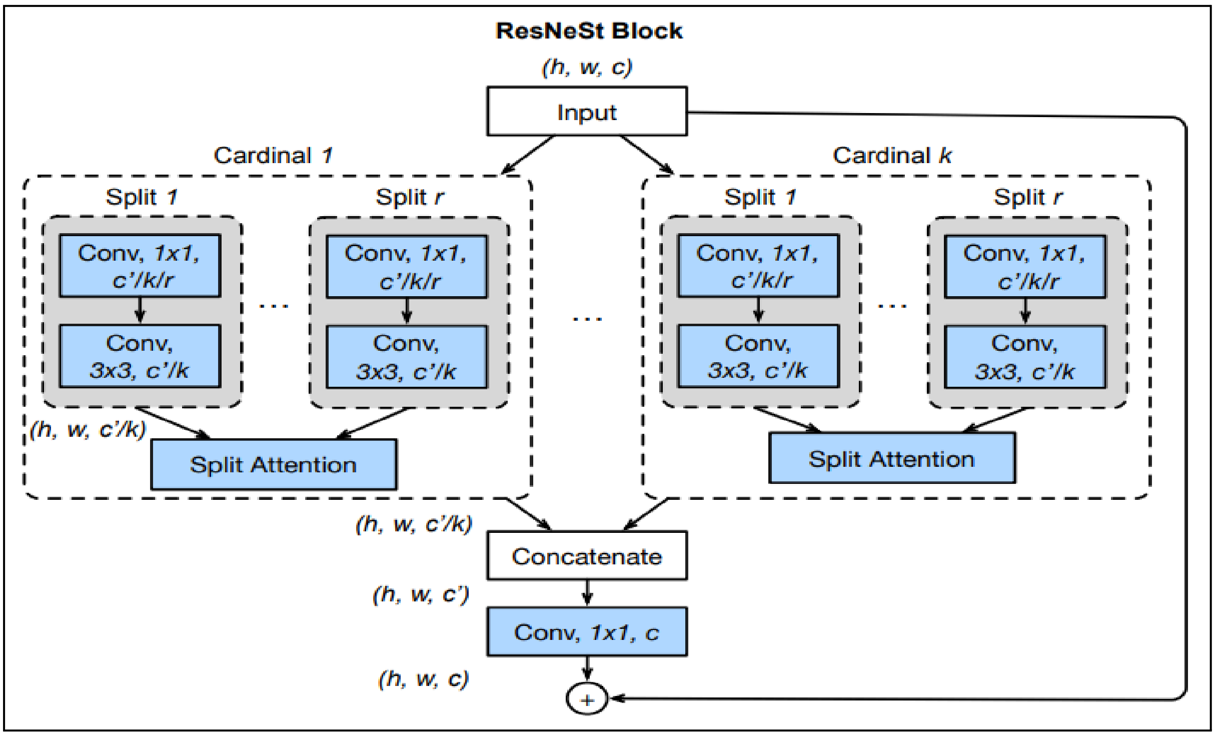
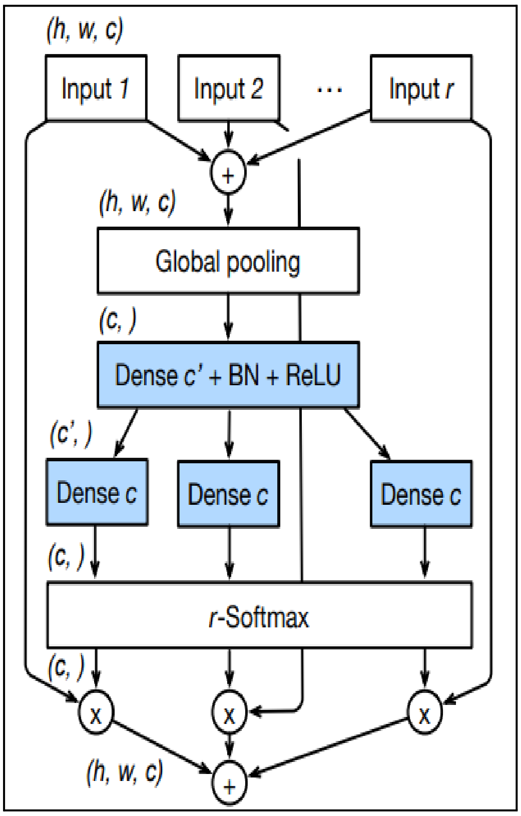


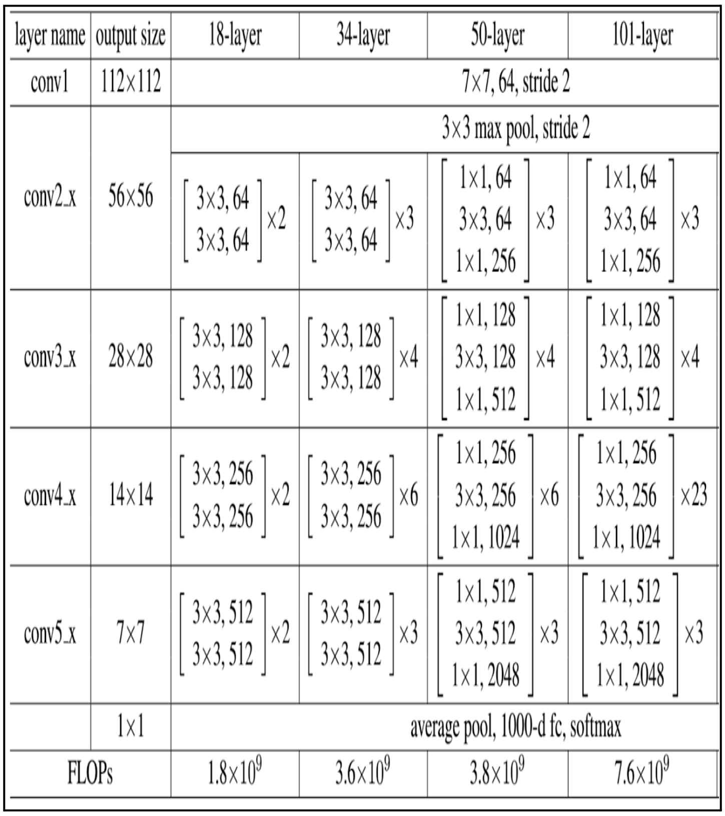
Fig 3. The architecture of ResNeSt

As illustrated in Fig. 4, the mechanism of the Split-Attention block lies in dynamically weighting multiple feature splits within each cardinal group. Given an input feature map divided into rrr splits of shape (h,w,c), the splits are first aggregated via element-wise summation across spatial and channel dimensions to produce a unified representation.

This representation undergoes global average pooling to extract a global context vector of shape (c,), which captures high-level semantic information. The vector is then passed through a lightweight fully connected network—including a reduction layer (Dense c′), batch normalization, and ReLU—and projected into r channel-wise attention vectors (Dense c), corresponding to each split. These vectors are normalized using r-Softmax to ensure their sum across splits equals 1 for each channel.The resulting attention weights modulate the original splits via element-wise multiplication, emphasizing informative paths while suppressing less relevant ones. The weighted splits are finally summed to form the output of the cardinal group, allowing adaptive multi-path fusion and richer feature representation.

As shown in Fig. 5 (right), ResNeSt-101 retains the overall layer configuration of ResNet-101 but replaces traditional residual blocks with Split-Attention blocks, introducing attention-based fusion throughout the network for enhanced semantic representation.





layer name	output size	18-layer	34-layer	50-layer	101-layer
conv1	112×112	7×7, 64, stride 2			
conv2_x	56×56	3×3 max pool, stride 2			
		[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax			
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9

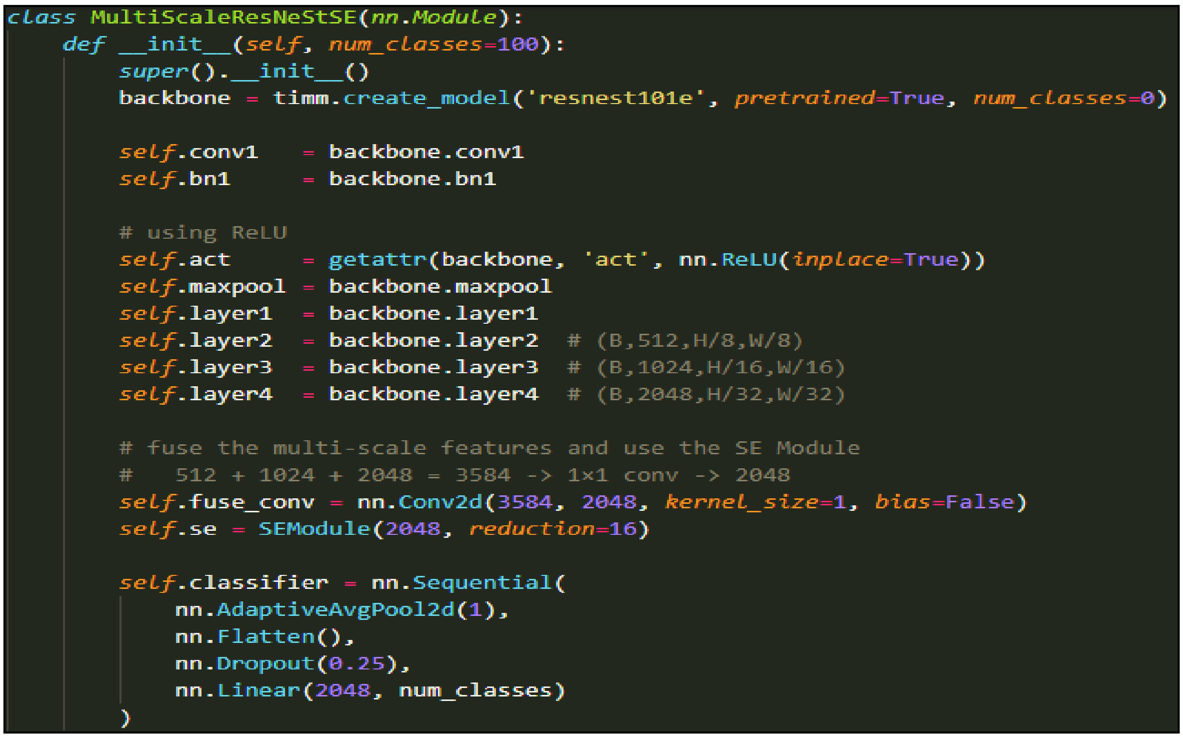
Fig 4、5. The operations of cardinal groups and the per layers structure of resnet-101.

2.2.2 The modification of the ResNeSt-101e

(1) Feature Pyramid Fusion

To extract richer contextual information from different stages of the ResNeSt-101e backbone, we draw inspiration from the concept of Feature Pyramid Networks (FPN) [3]. After the initial feature extraction, we selectively retrieve outputs from layer2 (H/8, 512 channels), layer3 (H/16, 1024 channels), and layer4 (H/32, 2048 channels). While layer2 retains more spatial detail but encodes relatively shallow semantics, layer4 captures high-level semantic features at a lower resolution. By fusing representations from these three stages, the network can simultaneously preserve fine-grained local details and capture global semantic context.

To integrate features from different depths of the ResNeSt-101e backbone, we first upsample the outputs of layer3 and layer4 via bilinear interpolation to match the spatial resolution of layer2 (H/8, W/8). These three feature maps are then concatenated along the channel dimension, resulting in a composite tensor of 3584 channels (512 + 1024 + 2048). This fusion allows the model to simultaneously leverage low-level spatial details and high-level semantic cues from multiple stages. To streamline computation, a 1×1 convolution is applied to reduce the number of channels to 2048. The compressed feature is then refined by a Squeeze-and-Excitation (SE) module [4], which applies channel-wise attention to enhance informative responses and suppress redundancy. The detailed implementation is shown in Fig. 6 and Fig. 7.



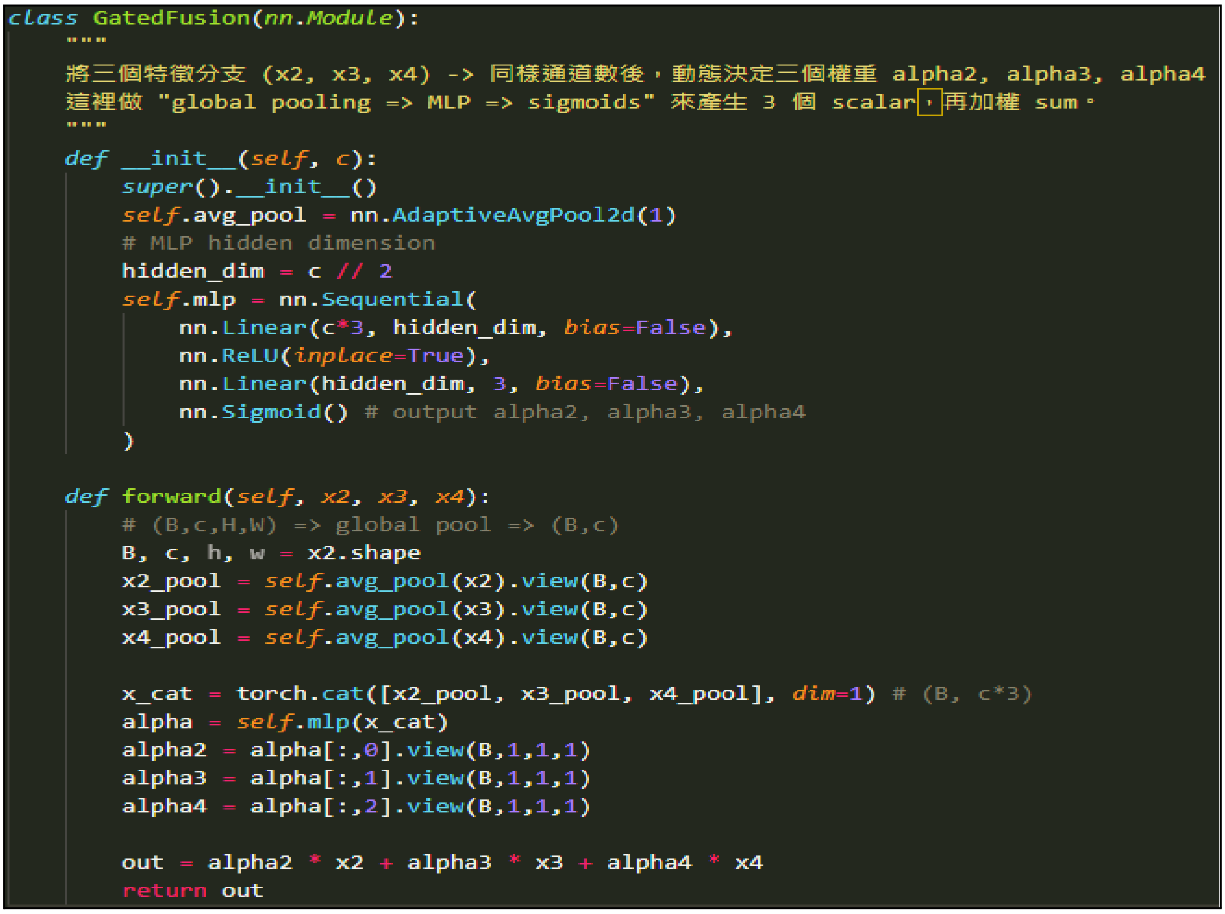
```
class MultiScaleResNeStSE(nn.Module):
    def __init__(self, num_classes=100):
        super().__init__()
        backbone = timm.create_model('resnest101e', pretrained=True, num_classes=0)

        self.conv1   = backbone.conv1
        self.bn1     = backbone.bn1

        # using ReLU
        self.act     = getattr(backbone, 'act', nn.ReLU(inplace=True))
        self.maxpool = backbone.maxpool
        self.layer1  = backbone.layer1
        self.layer2  = backbone.layer2  # (B,512,H/8,W/8)
        self.layer3  = backbone.layer3  # (B,1024,H/16,W/16)
        self.layer4  = backbone.layer4  # (B,2048,H/32,W/32)

        # fuse the multi-scale features and use the SE Module
        #   512 + 1024 + 2048 = 3584 -> 1x1 conv -> 2048
        self.fuse_conv = nn.Conv2d(3584, 2048, kernel_size=1, bias=False)
        self.se = SEModule(2048, reduction=16)

        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Dropout(0.25),
            nn.Linear(2048, num_classes)
        )
```

Fig 6. The code of Feature Pyramid Fusion which modifying the original ResNeSt

(2) Gate Fusion

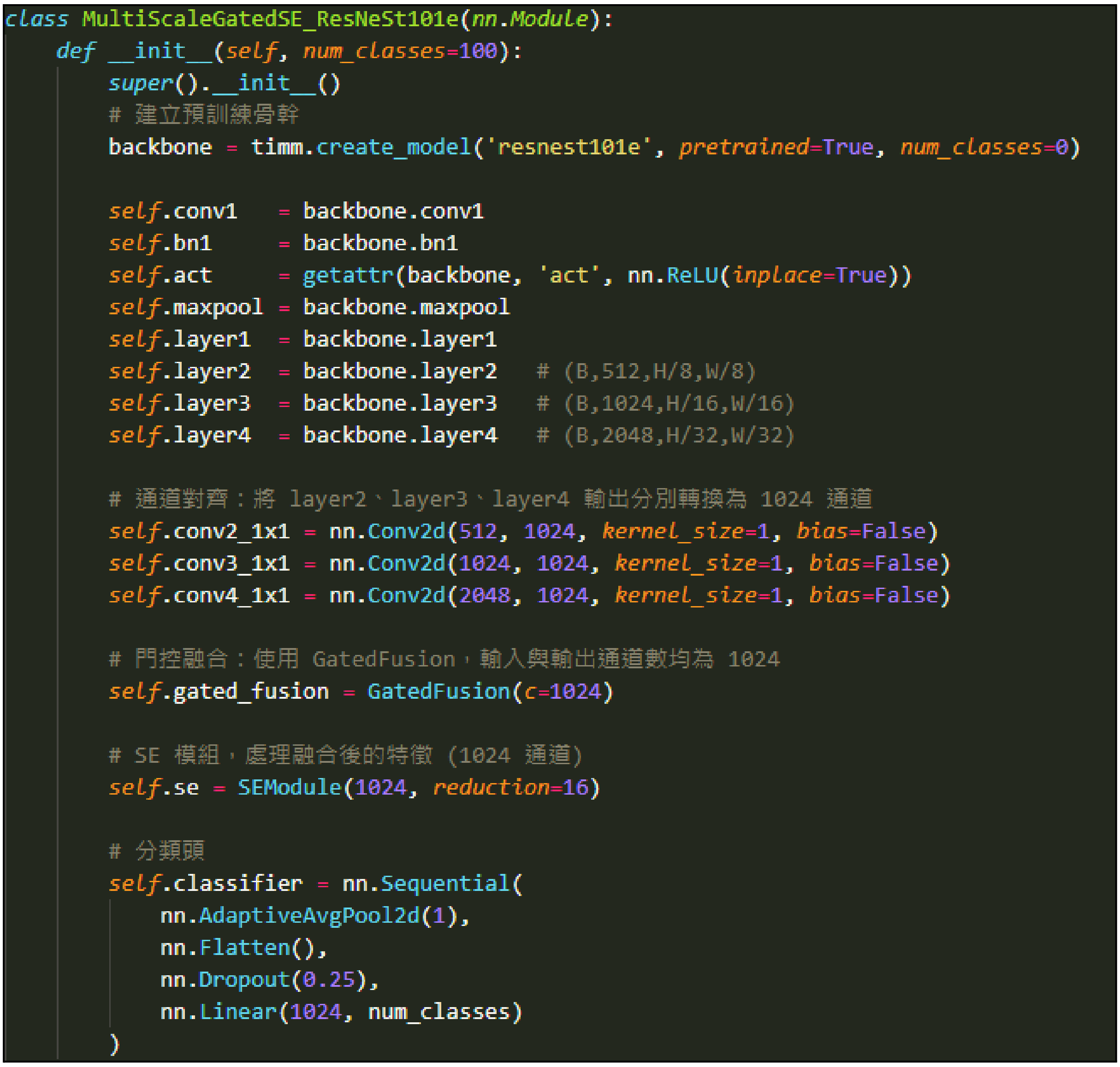
The core idea of Gate Fusion [5] is to dynamically control the contribution of multiple feature branches by learning distinct weights for each, enabling adaptive fusion across scales or semantic levels. As shown in Fig. 7 and Fig. 8, global average pooling is first applied to the outputs from different layers, reducing each to a (B, c) vector. These vectors are concatenated and passed through a lightweight MLP followed by a Sigmoid activation to generate scalar gating weights. Each branch is then modulated by its corresponding weight, and the weighted features are summed to produce the fused representation.

To integrate this mechanism with layer2, layer3, and layer4 of ResNeSt-101e—which differ in both resolution and channel size—we first upsample layer3 and layer4 to match the spatial resolution of layer2. All feature maps are then projected to 1024 channels using 1×1 convolutions (taking layer3 as the reference). After fusion, a Squeeze-and-Excitation (SE) module is applied to enhance important channels and suppress irrelevant ones. The final output retains a dimensionality of 1024 channels, which differs from the original 2048 channels of ResNeSt-101e, offering a trade-off between representational power and efficiency.



```
class GatedFusion(nn.Module):
    """
    將三個特徵分支 (x2, x3, x4) -> 同樣通道數後，動態決定三個權重 alpha2, alpha3, alpha4
    這裡做 "global pooling => MLP => sigmoids" 來產生 3 個 scalar，再加權 sum。
    """
    def __init__(self, c):
        super().__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        # MLP hidden dimension
        hidden_dim = c // 2
        self.mlp = nn.Sequential(
            nn.Linear(c*3, hidden_dim, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(hidden_dim, 3, bias=False),
            nn.Sigmoid() # output alpha2, alpha3, alpha4
        )

    def forward(self, x2, x3, x4):
        # (B,c,H,W) => global pool => (B,c)
        B, c, h, w = x2.shape
        x2_pool = self.avg_pool(x2).view(B,c)
        x3_pool = self.avg_pool(x3).view(B,c)
        x4_pool = self.avg_pool(x4).view(B,c)

        x_cat = torch.cat([x2_pool, x3_pool, x4_pool], dim=1) # (B, c*3)
        alpha = self.mlp(x_cat)
        alpha2 = alpha[:,0].view(B,1,1,1)
        alpha3 = alpha[:,1].view(B,1,1,1)
        alpha4 = alpha[:,2].view(B,1,1,1)

        out = alpha2 * x2 + alpha3 * x3 + alpha4 * x4
        return out
```

Fig 7. The code of Gate Fusion Function



```python
class MultiScaleGatedSE_ResNeSt101e(nn.Module):
    def __init__(self, num_classes=100):
        super().__init__()
        # 建立預訓練骨幹
        backbone = timm.create_model('resnest101e', pretrained=True, num_classes=0)

        self.conv1   = backbone.conv1
        self.bn1     = backbone.bn1
        self.act     = getattr(backbone, 'act', nn.ReLU(inplace=True))
        self.maxpool = backbone.maxpool
        self.layer1  = backbone.layer1
        self.layer2  = backbone.layer2   # (B,512,H/8,W/8)
        self.layer3  = backbone.layer3   # (B,1024,H/16,W/16)
        self.layer4  = backbone.layer4   # (B,2048,H/32,W/32)

        # 通道對齊：將 layer2、layer3、layer4 輸出分別轉換為 1024 通道
        self.conv2_1x1 = nn.Conv2d(512, 1024, kernel_size=1, bias=False)
        self.conv3_1x1 = nn.Conv2d(1024, 1024, kernel_size=1, bias=False)
        self.conv4_1x1 = nn.Conv2d(2048, 1024, kernel_size=1, bias=False)

        # 門控融合：使用 GatedFusion，輸入與輸出通道數均為 1024
        self.gated_fusion = GatedFusion(c=1024)

        # SE 模組，處理融合後的特徵 (1024 通道)
        self.se = SEModule(1024, reduction=16)

        # 分類頭
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Dropout(0.25),
            nn.Linear(1024, num_classes)
        )
```

Fig 8. The code of Multi-Scale Feature Fusion which using Gate Fusion

2.3 Hyperparameters Settings and Training Configurations

2.3.1 Hyperparameters Settings

The hyperparameters adopted in this study are summarized as follows:

- BATCH_SIZE: 32
- LR: 1e-4
- EPOCHS: 100
- WEIGHT_DECAY: 1e-5
- Optimizer: Adam

2.3.2 Training Configurations

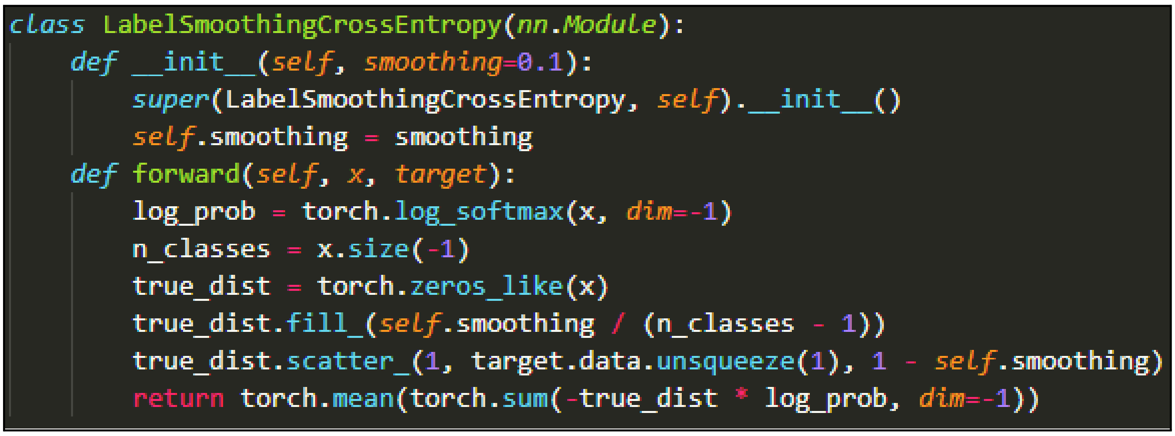
(1) LR scheduler and Early Stopping mechanisms:

To monitor model performance, the **validation accuracy** is adopted as the evaluation metric. If the validation accuracy does not improve for **five consecutive epochs**, the learning rate is reduced by a factor of **0.25**. Furthermore, an **early stopping** mechanism is employed: if no improvement is observed in the validation accuracy for **15 consecutive epochs**, the training process is terminated. The corresponding implementation is shown below:

ReduceLROnPlateau(optimizer, mode="max", factor=0.25, patience=5, verbose=True)

(2) Criterion:

In this work, we employ two types of loss functions as the training criterion: **Cross Entropy Loss** and **Label Smoothing Cross Entropy**. The concept of label smoothing is to modify the target distribution by replacing the hard one-hot labels with a softened version, where the ground-truth class is assigned a probability of **$1-\epsilon$ and the remaining ϵ is uniformly distributed among the other classes.** This technique discourages the model from becoming overly confident in its predictions, thereby enhancing training stability and generalization, and also helps mitigate overfitting. The corresponding implementation is presented below:



```python
class LabelSmoothingCrossEntropy(nn.Module):
    def __init__(self, smoothing=0.1):
        super(LabelSmoothingCrossEntropy, self).__init__()
        self.smoothing = smoothing
    def forward(self, x, target):
        log_prob = torch.log_softmax(x, dim=-1)
        n_classes = x.size(-1)
        true_dist = torch.zeros_like(x)
        true_dist.fill_(self.smoothing / (n_classes - 1))
        true_dist.scatter_(1, target.data.unsqueeze(1), 1 - self.smoothing)
        return torch.mean(torch.sum(-true_dist * log_prob, dim=-1))
```

Fig 9. The code of Label Smoothing Cross Entropy Loss

3. Results & Additional experiments

3.1 The ablation study of the augmentation and criterion of ResNeSt

(1) ResNeSt (with CutMix and Label Smoothing Cross Entropy):

In this experiment, model convergence was observed around epochs 35 to 40, at which point the early stopping mechanism was triggered. As illustrated in Fig. 10 and Fig. 11, both the training loss and accuracy curves indicate a stable learning process without signs of overfitting. Due to the incorporation of **CutMix** and **Label Smoothing Cross Entropy**, the final loss values consistently converged to approximately **1.15**. The model achieved a peak validation accuracy of **0.8967**, while the corresponding performance on the **CodaBench** testing platform reached **0.92**, demonstrating the model's strong generalization capability.

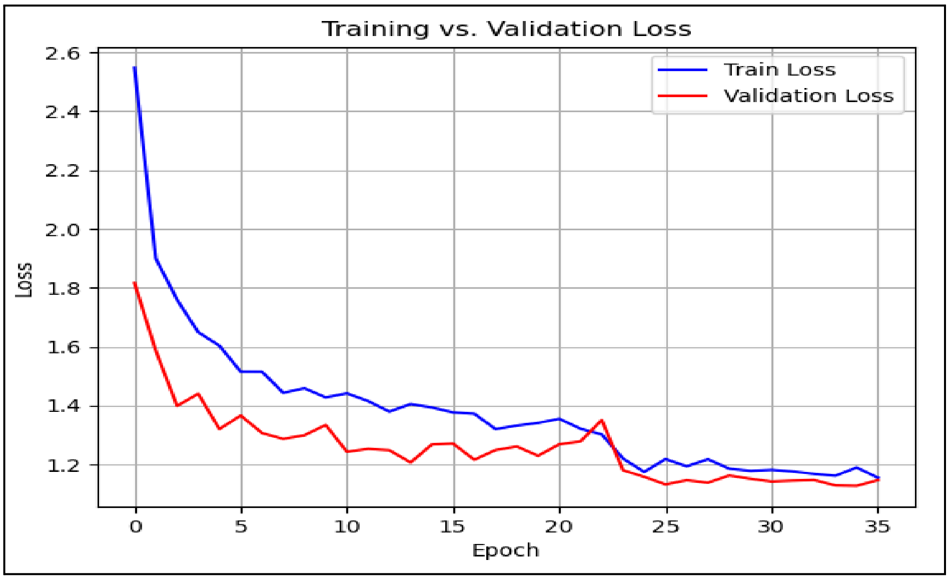


Fig 10. The plot of Training and Validation Loss value which using ResNeSt with CutMix and using Label Smoothing Cross Entropy as Loss Function

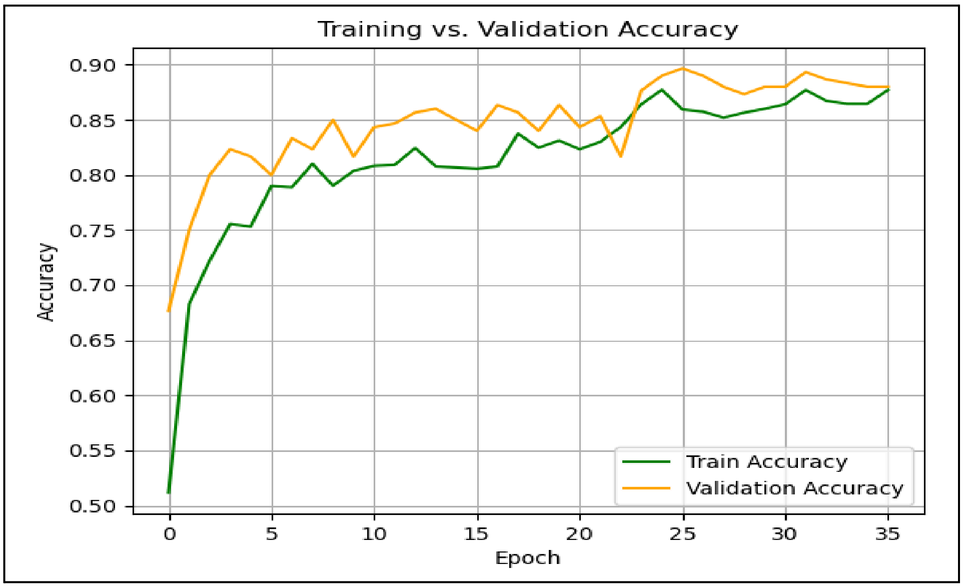


Fig 11. The plot of Training and Validation Accuracy value which using ResNeSt with CutMix and using Label Smoothing Cross Entropy as Loss Function

(2) ResNeSt w/ CutMix and Cross Entropy Loss:

In this experiment, using CutMix combined with Cross Entropy Loss, the model converged around the 50th epoch and triggered the early stopping mechanism. As shown in Fig. 12 and Fig. 13, both the loss and accuracy curves indicate a smooth and stable training process, with no signs of overfitting. The final training loss converged to approximately 0.4, and the highest validation accuracy reached 0.9033. Additionally, the model achieved a test accuracy of 0.93 on the CodaBench evaluation platform, further validating the generalization performance under this training configuration.

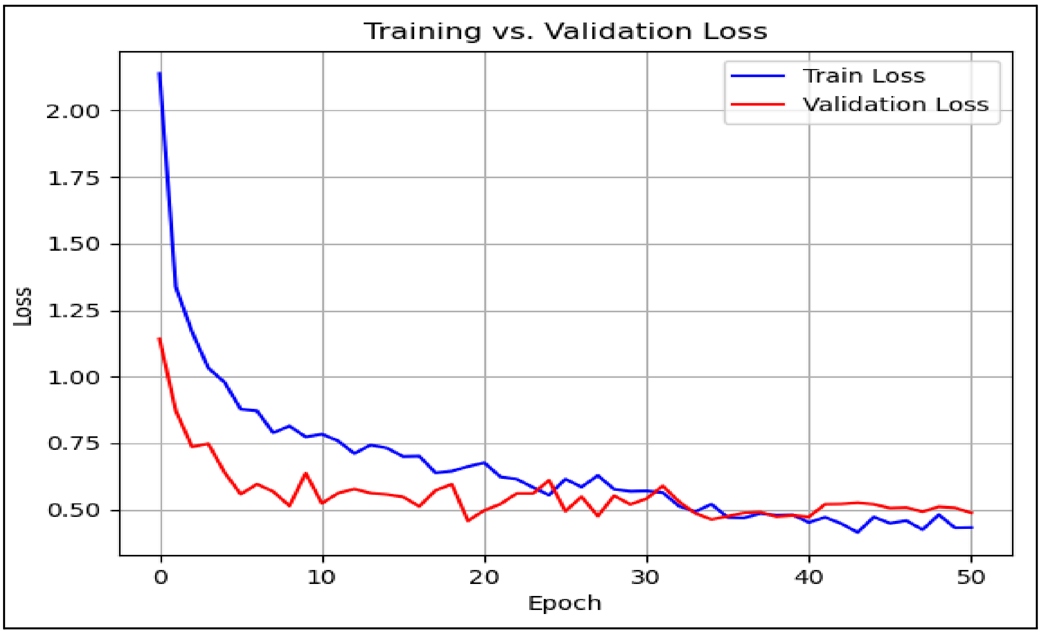


Fig 12. The plot of Training and Validation Loss value which using ResNeSt with CutMix and using Cross Entropy as Loss Function

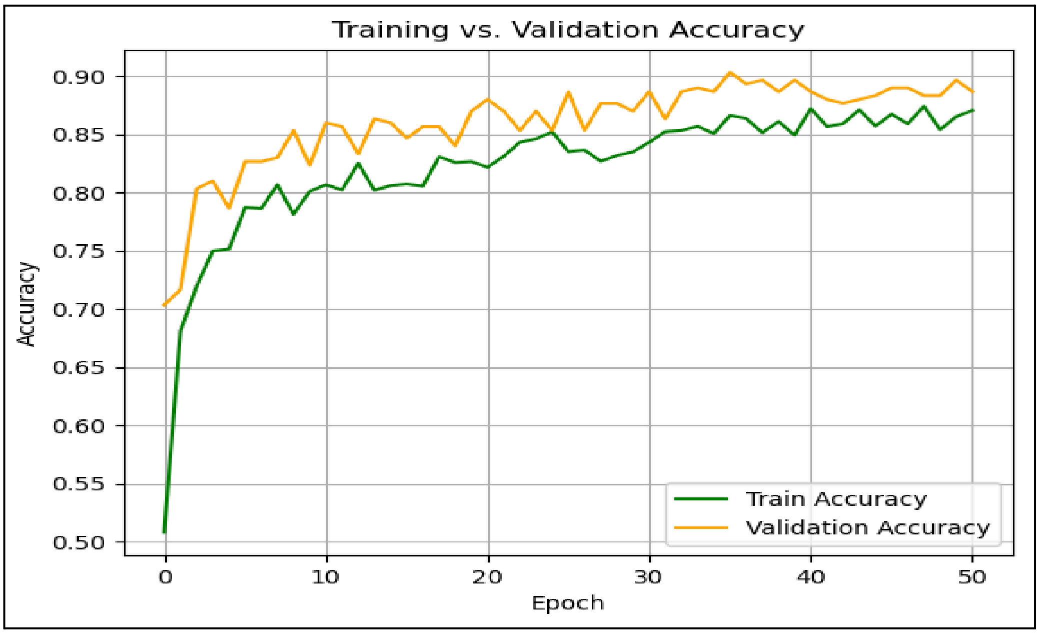


Fig 13. The plot of Training and Validation Accuracy value which using ResNeSt with CutMix and using Cross Entropy as Loss Function

(3) ResNeSt o/ CutMix but still using Label Smoothing Cross Entropy:

In this experiment, the training process converged between the 35th and 40th epoch, at which point the early stopping mechanism was triggered. As illustrated in Fig. 14 and Fig. 15, the training loss curve exhibited a significantly faster convergence rate than the validation curve. Notably, while the training loss converged to approximately **0.8**, the validation loss remained around **1.2**, indicating a clear **overfitting** issue. In terms of accuracy, the training set achieved a near-perfect accuracy close to **1.0**, whereas the best **validation accuracy** was only **0.8733**. These findings suggest that, despite employing the same loss function, the absence of **CutMix** in the data augmentation pipeline can lead to substantial overfitting and hinder generalization performance.

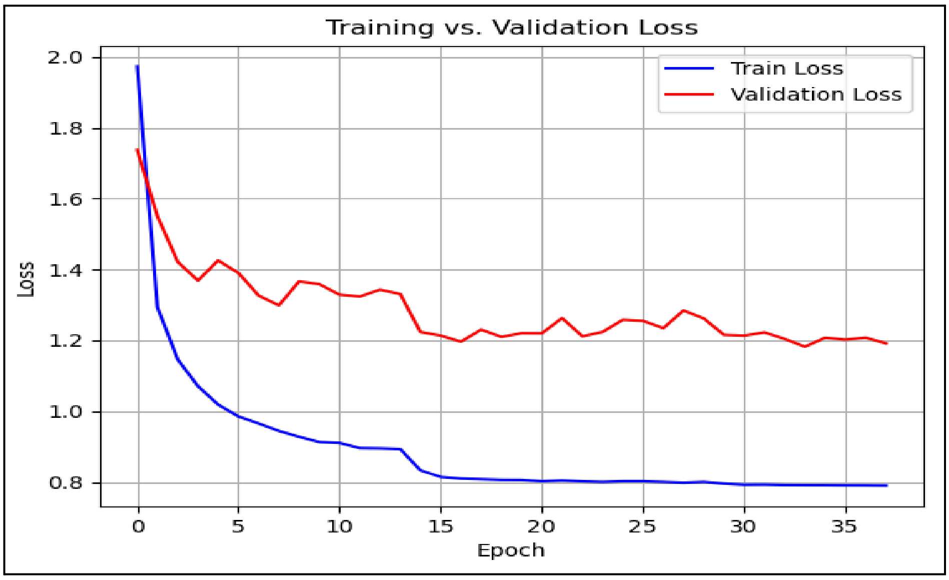


Fig 14. The plot of Training and Validation Loss value which using ResNeSt without CutMix but using Label Smoothing Cross Entropy as Loss Function

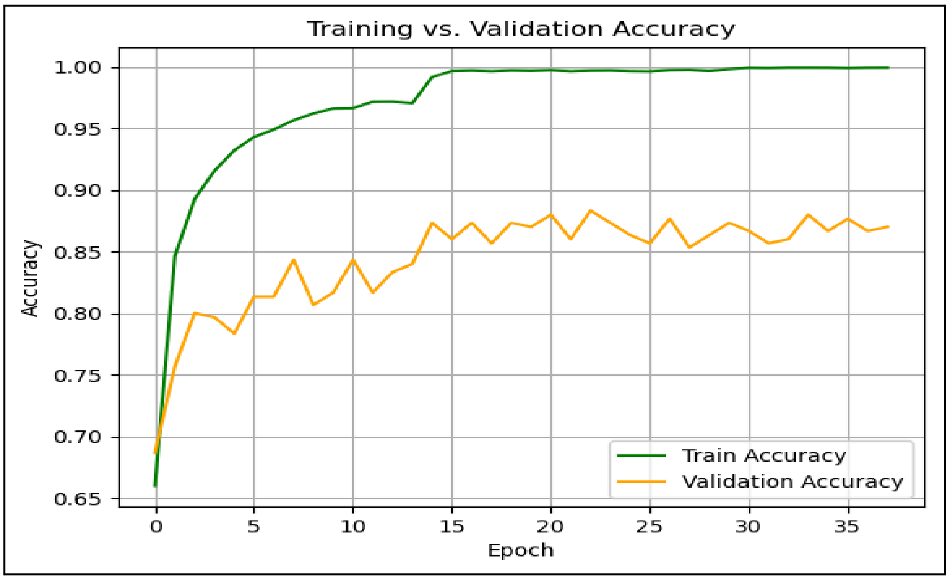


Fig 15. The plot of Training and Validation Accuracy value using ResNeSt without CutMix but using Label Smoothing Cross Entropy as Loss Function

(4) ResNeSt o/ CutMix but still using Cross Entropy as criterion:

In this experiment, training converged around epoch 55–60 and triggered the early stopping mechanism, as illustrated in the loss and accuracy curves shown in Fig. 16 and Fig. 17. Throughout the training process, the training loss decreased significantly faster than the validation loss, indicating a noticeable overfitting issue—particularly near convergence, where the validation loss plateaued at approximately 0.6, while the training loss approached near-zero. Similarly, in terms of accuracy, the validation accuracy peaked at 0.8900, whereas the training accuracy approached 1.0. These observations suggest that although using Cross Entropy Loss without CutMix resulted in slightly better validation accuracy compared to using Label Smoothing Cross Entropy Loss, the model exhibited a more severe overfitting tendency. In contrast, the use of Label Smoothing appears to mitigate overfitting more effectively under similar conditions.

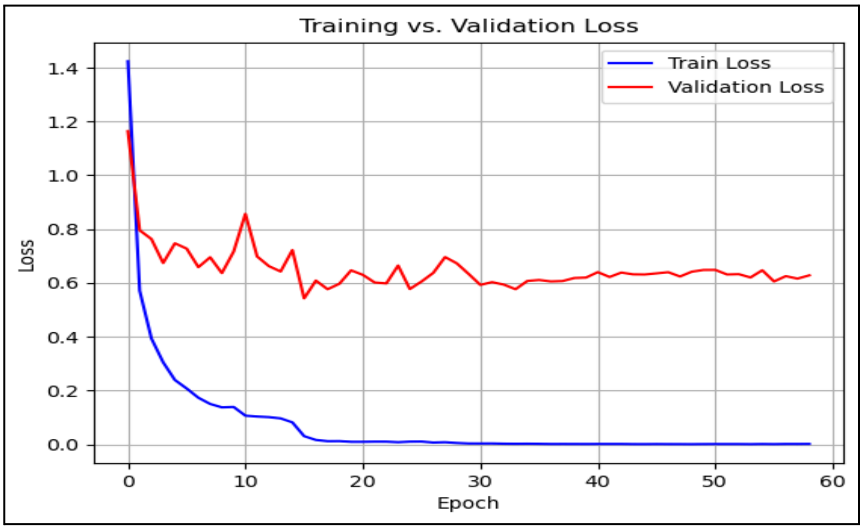


Fig 16. The plot of Training and Validation Loss value which using ResNeSt without CutMix but using Cross Entropy as Loss Function

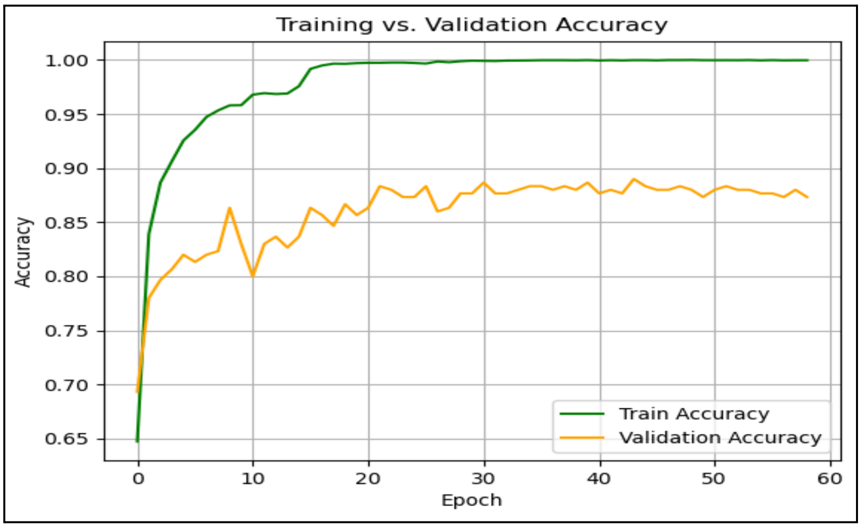


Fig 17. The plot of Training and Validation Accuracy value using ResNeSt without CutMix but using Cross Entropy as Loss Function

(5) Baseline Experiment Summary and Insights

Based on the results from four experimental groups using ResNeSt as the baseline model, we observe that incorporating **CutMix** consistently mitigates overfitting throughout training. In contrast, models trained solely with conventional augmentation methods such as rotation, cropping, and brightness adjustment tend to exhibit overfitting regardless of whether **Label Smoothing Cross Entropy Loss** or standard **Cross Entropy Loss** is employed. While the use of Label Smoothing introduces noticeable differences in loss values, its impact on classification accuracy in the baseline setting remains marginal. Consequently, in subsequent experiments involving enhanced architectures, we adopt **CutMix** as part of the data augmentation strategy and systematically evaluate the effect of both loss functions across different model variants.

3.2 The experiment results of proposed methods

（1） ResNeSt + Gate Fusion w/ CutMix and Label Smoothing Cross Entropy Loss

In this experiment, we designated the 1024-channel output from the third layer as the reference for aligning the channel dimensions of the second and fourth layers prior to feature fusion. According to the training dynamics, the model converged around the 40th epoch, at which point the early stopping mechanism was triggered. As shown in Fig. 18 and Fig. 19, both the training loss and accuracy curves exhibited stable convergence, with no signs of overfitting throughout the training process. Owing to the integration of **CutMix** and **Label Smoothing Cross Entropy Loss**, the final loss value converged to approximately **1.1**. The best validation accuracy reached **0.9200**, while the model achieved a test accuracy of **0.95** on the CodaBench platform.

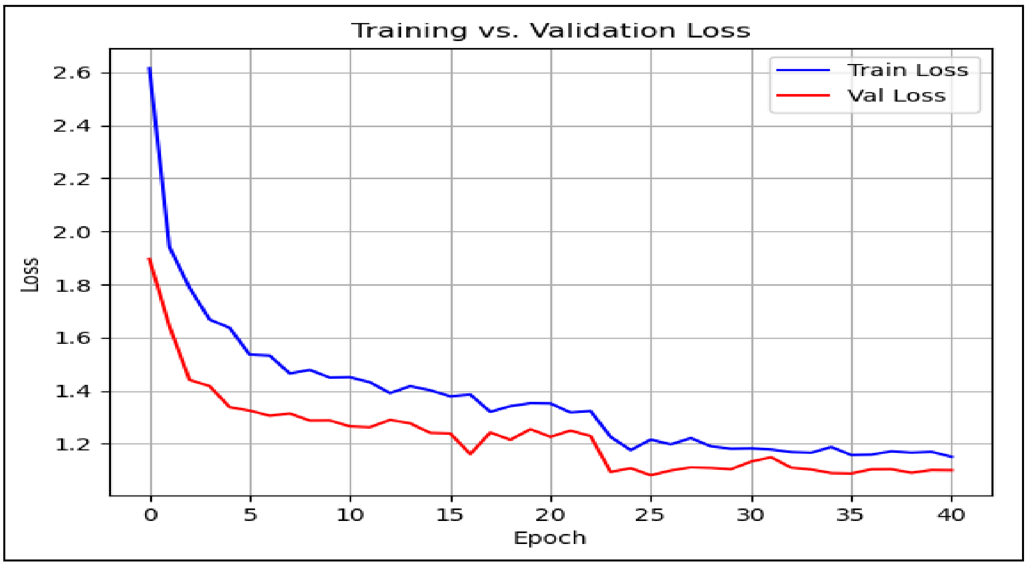


Fig 18. The plot of Training and Validation Loss value which using ResNeSt + Gate Fusion with CutMix and using Label Smoothing Cross Entropy as Loss Function

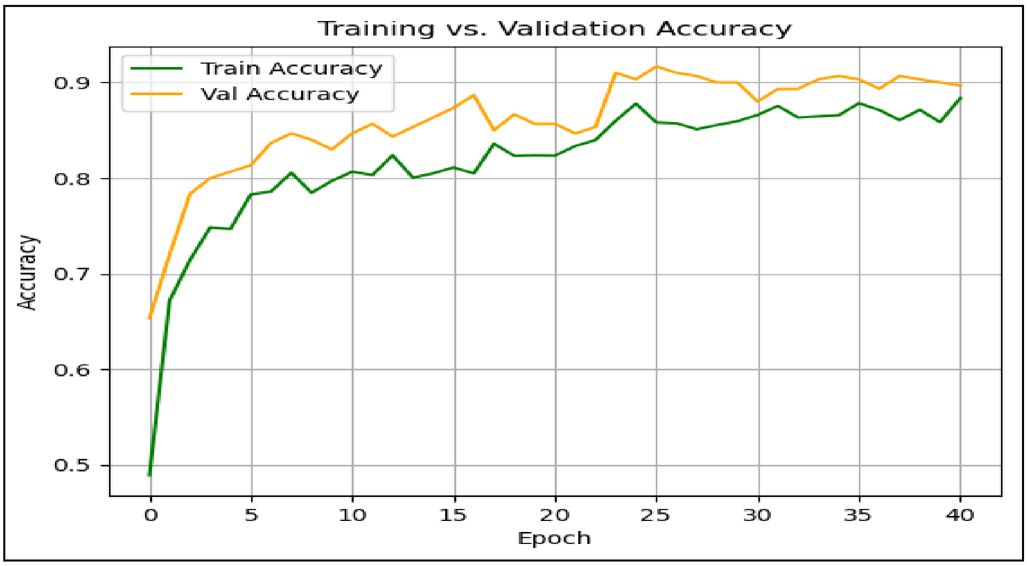


Fig 19. The plot of Training and Validation Accuracy value which using ResNeSt + Gate Fusion with CutMix and using Label Smoothing Cross Entropy as Loss Function

（2） ResNeSt + Gate Fusion w/ CutMix and Cross Entropy Loss

In this experiment, we selected the 1024-channel output from the third layer as the reference for aligning the channel dimensions of the second and fourth layers during feature fusion. As illustrated in Fig. 20 and Fig. 21, the training process converged around the 40th epoch, at which point the early stopping mechanism was triggered. Throughout the training phase, both the loss and accuracy curves exhibited stable trends without any indication of overfitting. This experiment employed CutMix as the data augmentation strategy and used the Cross Entropy Loss as the objective function, with the final training loss converging to approximately 0.4. The best validation accuracy reached 0.9033, while the model achieved a 0.93 accuracy on the CodaBench test set.

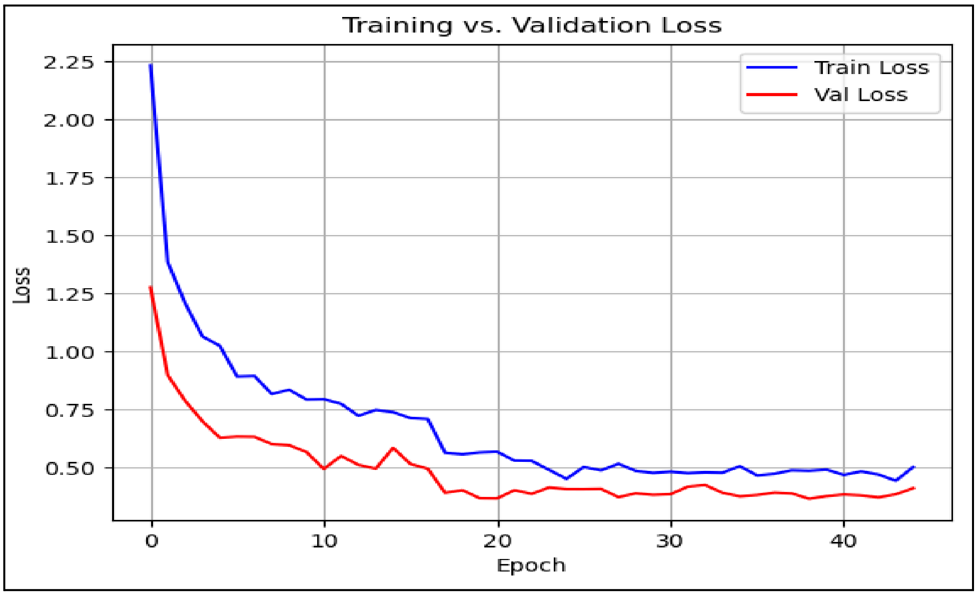


Fig 20. The plot of Training and Validation Loss value which using ResNeSt + Gate Fusion with CutMix and using Cross Entropy as Loss Function

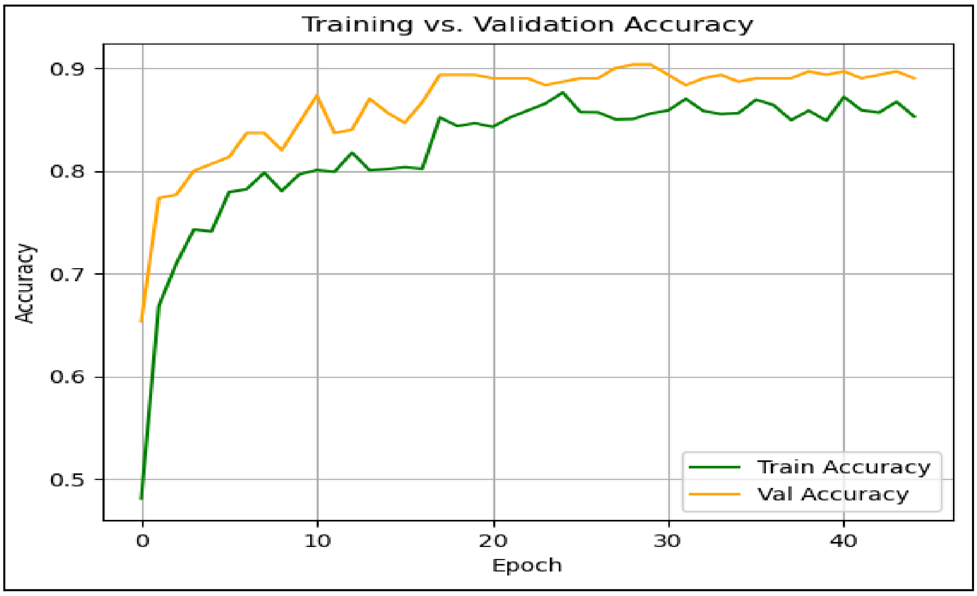


Fig 21. The plot of Training and Validation Accuracy value which using ResNeSt + Gate Fusion with CutMix and using Cross Entropy as Loss Function

（3）**ResNeSt + Pyramid Fusion w/ CutMix and Label Smoothing Cross Entropy Loss**

In this experiment, the feature maps from the second to the fourth layers were first upsampled to match the spatial resolution of the second layer. These features, now aligned in resolution but differing in channel dimensions, were subsequently fused to capture multi-level semantic and spatial information. As illustrated in Fig. 22 and Fig. 23, the model converged around the 45th to 50th epoch, at which point the early stopping mechanism was triggered. Throughout training, both the loss and accuracy curves exhibited stable trends with no signs of overfitting. This setting incorporated CutMix as the data augmentation strategy and employed Label Smoothing Cross Entropy as the loss function, with the training loss converging to approximately 1.1. The best validation accuracy reached 0.9067, and the model achieved 0.94 on the CodaBench test set.

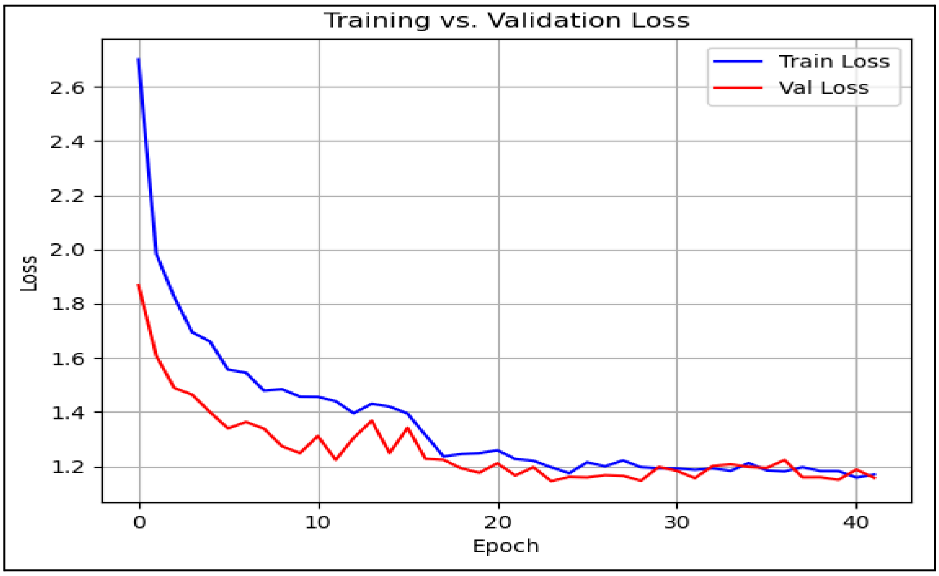


Fig 22. The plot of Training and Validation Loss value using ResNeSt + Pyramid Fusion with CutMix and using Label Smoothing Cross Entropy as Loss Function

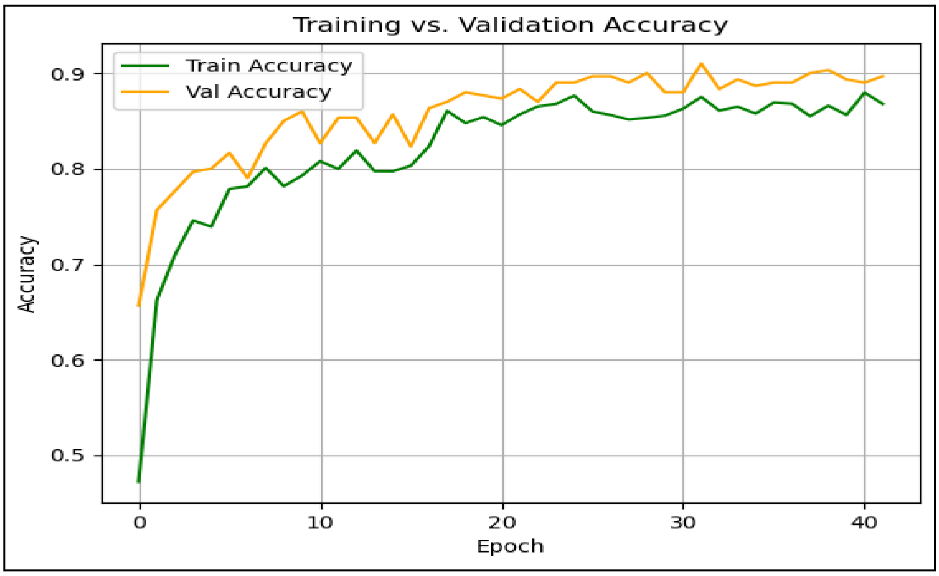


Fig 23. The plot of Training and Validation Accuracy value using ResNeSt + Pyramid Fusion with CutMix and using Label Smoothing Cross Entropy as Loss Function

（4）ResNeSt + Pyramid Fusion w/ CutMix and Cross Entropy Loss

In this experiment, we upsampled the outputs from layers 3 and 4 to match the spatial resolution of layer 2, followed by feature fusion across these three layers with aligned resolutions but different channel dimensions. According to the training results, the model converged around epoch 80, triggering the early stopping mechanism. As illustrated in Fig. 24 and Fig. 25, both the loss and accuracy curves show stable convergence without signs of overfitting. This configuration employed Cross Entropy Loss, with the final training loss converging to approximately 0.4. The model achieved a validation accuracy of 0.9033 and scored 0.93 on the CodaBench test set.

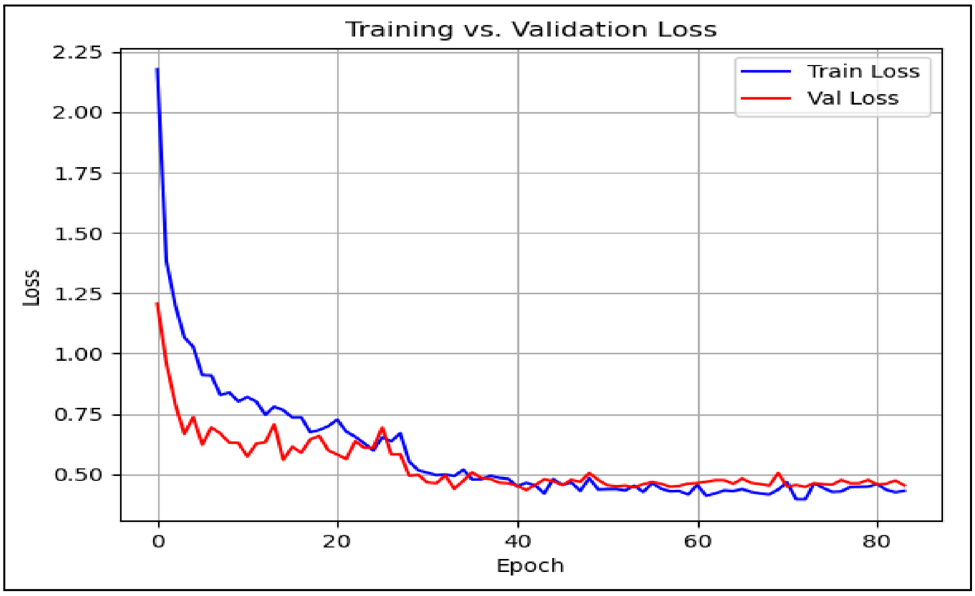


Fig 24. The plot of Training and Validation Loss value using ResNeSt + Pyramid Fusion with CutMix and using Cross Entropy as Loss Function

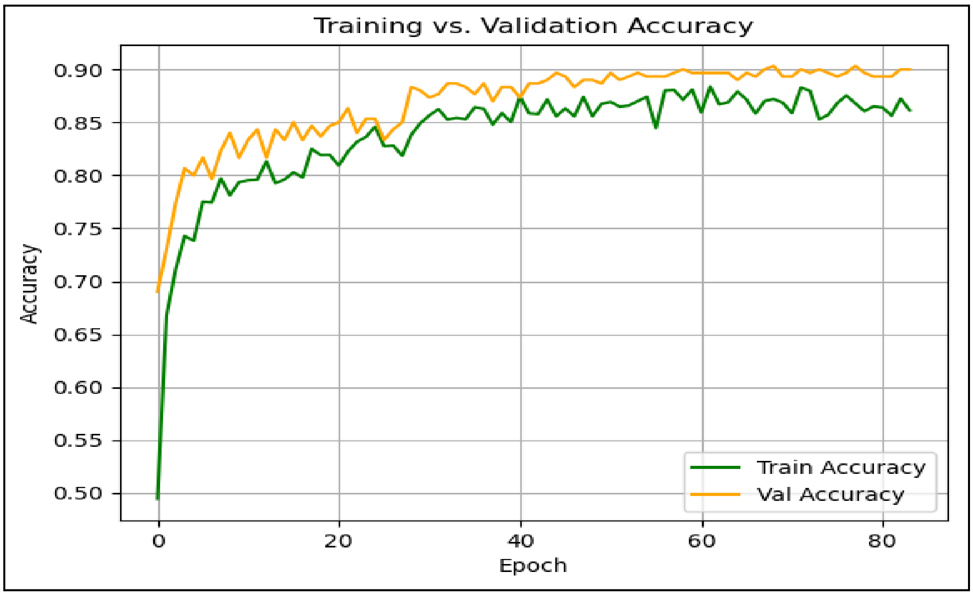


Fig 25. The plot of Training and Validation Accuracy value using ResNeSt + Multi-Scaled Fusion with CutMix and using Cross Entropy as Loss Function

4. Discussion and Conclusion

4.1 Experiment Results

In this study, we primarily adopted and compared two regularization techniques—**CutMix[1]** and **Label Smoothing Cross Entropy Loss[6]**—to mitigate overfitting on the training set. Experimental results indicate that **CutMix** significantly enhances the diversity of training data and effectively reduces overfitting by encouraging the model to learn more generalizable features**[1]**. Under the CutMix setting, we further investigated the impact of using **Label Smoothing Cross Entropy Loss** versus standard **Cross Entropy Loss**. While the baseline model trained with Cross Entropy showed slightly higher accuracy, our proposed feature fusion models built upon ResNeSt-101e generally performed better when trained with Label Smoothing, suggesting that it offers additional regularization benefits in deeper or more complex architectures.

Among the methods evaluated, all proposed variants outperformed the baseline model, with the sole exception of the **ResNeSt + Pyramid Fusion w/ CutMix and Cross Entropy Loss configuration**, which achieved comparable performance to the baseline. Notably, the ResNeSt + Gate Fusion w/ CutMix and Label Smoothing Cross Entropy Loss combination achieved the best performance, **surpassing the baseline by approximately 2% on the validation set, and achieving a test accuracy of around 0.95**. These results empirically support our initial hypothesis: while the Split-Attention module in ResNeSt preserves earlier layer information to a certain extent, introducing explicit feature fusion mechanisms across layers enables the model to retain richer and more diverse representations. This in turn leads to enhanced generalization and overall performance across both validation and test sets.

The following table summarizes the results presented in Chapter 3. The *Accuracy* column reports the performance obtained on the CodaBench testing platform. *CE* refers to **Cross Entropy**, and *LSCE* refers to **Label Smoothing Cross Entropy**.

Model	Accuracy	Training Configuration
ResNeSt-101e	0.93	**w/ CutMix + CE**
ResNeSt-101e + Pyramid Fusion	0.94	**w/ CutMix + LSCE**
ResNeSt-101e + Gate Fusion	0.95	**w/ CutMix + LSCE**

Table 1. Model Performance Comparison

4.2 Future Works

In this study, we proposed two feature fusion strategies—**Gate Fusion** and a **Pyramid Fusion approach inspired by Feature Pyramid Networks (FPN) [3]**—both of which yielded moderate improvements over the original ResNeSt-101e architecture. These results suggest that **multi-level feature fusion can be beneficial for enhancing the representational capacity of ResNet-based models**. With more flexible hardware support in the future, more advanced fusion mechanisms could be implemented to further boost overall model performance.

Additionally, our experiments revealed that while **CutMix** is effective in reducing overfitting by increasing data diversity, it may also **lower training accuracy**, likely due to the increased complexity of mixed samples. This trade-off suggests that although CutMix enhances generalization, it may **impede the model's ability to fully learn from the training set**. Future work may involve more fine-grained calibration of augmentation parameters or exploration of **alternative augmentation strategies**, aiming to improve learning on the training set while maintaining strong performance on validation and test data without overfitting.

4.3 Model Parameters Comparison

We computed the parameter counts for the Baseline Model and the two proposed methods. The results indicate that the Pyramid Fusion and Gate Fusion architectures introduce approximately 6 million and 3.5 million additional parameters, respectively, compared to the original Baseline Model.

Model	Parameters
ResNeSt-101e	48,275,016
ResNeSt-101e + Pyramid Fusion	54,295,236
ResNeSt-101e + Gate Fusion	51,704,004

Table 2. Model Parameters Comparison

Reference

[1] Yun, S., Han, D., Oh, S. J., Chun, S., Choe, J., & Yoo, Y. (2019). Cutmix: Regularization strategy to train strong classifiers with localizable features. In Proceedings of the IEEE/CVF international conference on computer vision (pp. 6023-6032).

[2] Zhang, H., Wu, C., Zhang, Z., Zhu, Y., Lin, H., Zhang, Z., ... & Smola, A. (2022). Resnest: Split-attention networks. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 2736-2746).

[3] Lin, T. Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature pyramid networks for object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2117-2125).

[4] Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 7132-7141).

[5] Arevalo, J., Solorio, T., Montes-y-Gómez, M., & González, F. A. (2017). Gated multimodal units for information fusion. arXiv preprint arXiv:1702.01992.

[6] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). *Rethinking the Inception Architecture for Computer Vision*. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 2818–2826). https://doi.org/10.1109/CVPR.2016.308

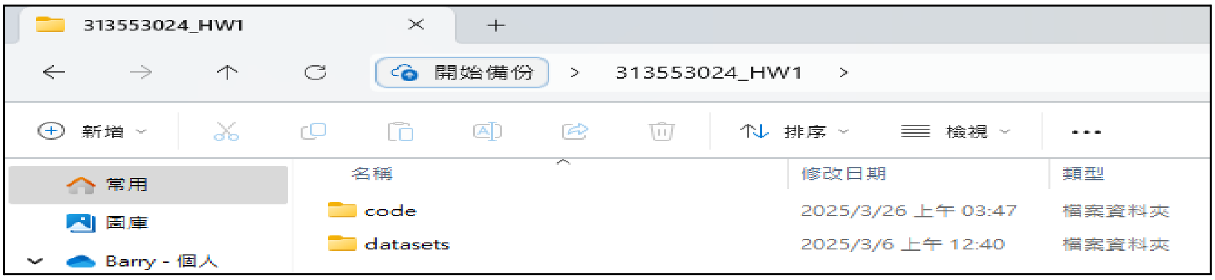
[7] https://blog.csdn.net/qq_36560894/article/details/118424356

[8] https://huggingface.co/docs/timm/main/en/models/resnest

Appendix

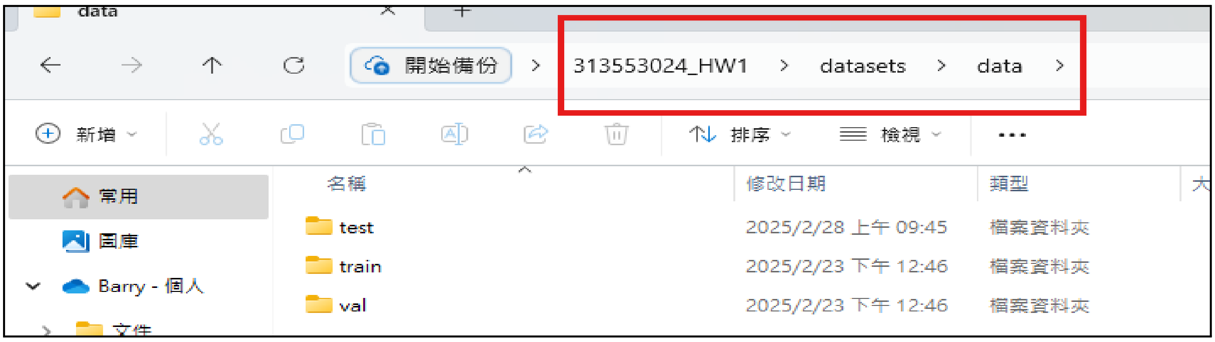
(1) File Structures:

We first created two folders(which called code and datasets) in the 313553024_HW1 folder, then we will download the data.zip—which includes the train, val, and test directories—extract it, and then place it under the "datasets" folder. Next, we create some .py files in the code folder.



名稱	修改日期	類型
code	2025/3/26 上午 03:47	檔案資料夾
datasets	2025/3/6 上午 12:40	檔案資料夾

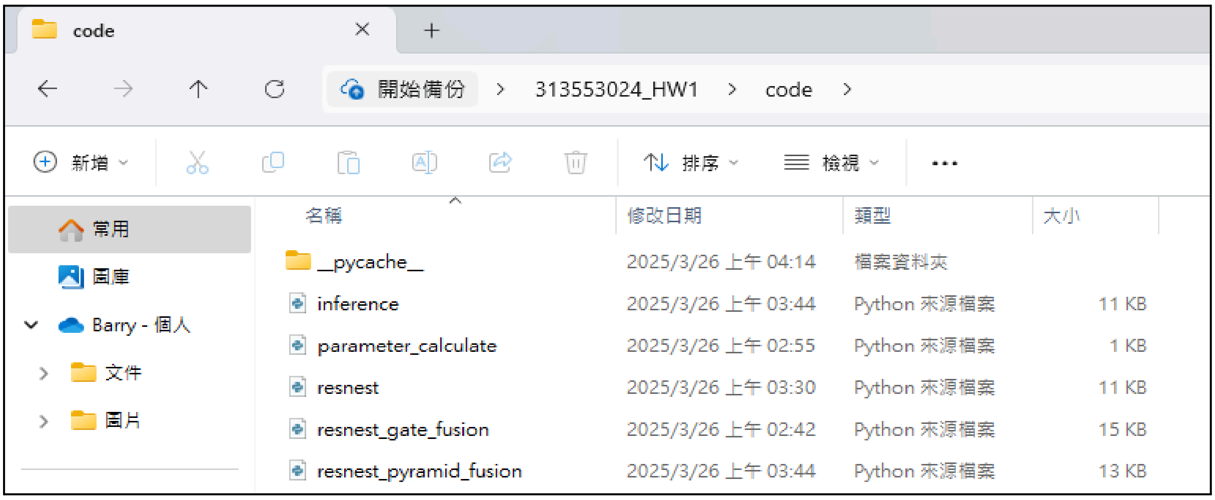
Fig 26. code folder is for our .py files and the datasets folder is for our datasets



名稱	修改日期	類型
test	2025/2/28 上午 09:45	檔案資料夾
train	2025/2/23 下午 12:46	檔案資料夾
val	2025/2/23 下午 12:46	檔案資料夾

Fig 27. The file structures of our datasets

In this repository (Fig 28.), we have implemented five Python files. Three of these, prefixed with "resnest," define different model architectures based on the ResNeSt backbone. Additionally, "parameter_calculate.py" is used to compute the total number of model parameters, and "inference.py" is provided for performing model inference.



名稱	修改日期	類型	大小
__pycache__	2025/3/26 上午 04:14	檔案資料夾	
inference	2025/3/26 上午 03:44	Python 來源檔案	11 KB
parameter_calculate	2025/3/26 上午 02:55	Python 來源檔案	1 KB
resnest	2025/3/26 上午 03:30	Python 來源檔案	11 KB
resnest_gate_fusion	2025/3/26 上午 02:42	Python 來源檔案	15 KB
resnest_pyramid_fusion	2025/3/26 上午 03:44	Python 來源檔案	13 KB

Fig 28. Our python codes in the code folder

(2) Command Line in Inference phase:

--model: Three model options are available: ResNeSt, Pyramid, and Gate.

--weights: Specify the file path of the trained model weights.

Below are example command lines for the three models. Please ensure that you replace the placeholder for the trained model weights with the file you have generated and named.

ResNeSt:

```
inference.py --model resnest --weights resnest_best.pt
```

Pyramid:

```
inference.py --model pyramid --weights pyramid_best.pt
```

Gate:

```
inference.py --model gate --weights gate_best.pt
```

(3) Github Link:

https://github.com/rmd926/NYCU_CV2025_Spring/tree/main/Lab1