[Github link] [Experiment Result]

Data Preprocessing

Data Augmentation

Methods	Discription
`RandomResizedCrop (224, scale=(0.6, 1.2), ratio=(0.75, 1.33))`	• Randomly crops a region from the image and resizes it to **224×224 pixels** to fit backbone model. • The **scale** defines the possible size of the crop relative to the original image (between **60% and 120%**). • The **aspect ratio** is randomly adjusted between **0.75 (taller)** and **1.33 (wider)**.
`RandomHorizontalFlip (p=0.5)`	• Flips the image **horizontally** with a probability of **50%**.
`RandomVerticalFlip (p=0.5)`	• Flips the image **vertically** with a probability of **50%**.
`RandomAffine (degrees=20, translate=(0.1, 0.1), shear=5)`	• **Rotates** the image randomly by up to **±20°**. • **Translates** (shifts) the image by **up to 10%** of its width/height. • Applies a **shear transformation** of up to **±5°**.
`ColorJitter (brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)`	• Brightness/Contrast/Saturation: Adjusted within **±20%**. • Hue: Adjusted within **±10%**, to retain correct color feature.

Random Erasing & Normalize

`ToTensor()`	• Converts the image to a **PyTorch tensor** with values normalized to **[0, 1]**.
`Normalize (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])`	• Standardizes pixel values using the mean and standard deviation of **ImageNet**. • Helps stabilize training and improve convergence.
`RandomErasing (p=0.3, scale=(0.02, 0.15), ratio=(0.3, 3.3), value='random')`	• With a **30% chance**, randomly removes a **small rectangular region** of the image. • The erased pixels are replaced with **random values** to prevent overfitting.

Training Time Augmentation

Based on previous experiment results, resnet101 and resnet50 tended to be overfitting to training data. I tried **MixUp** and **CutMix** for stronger augmentation.

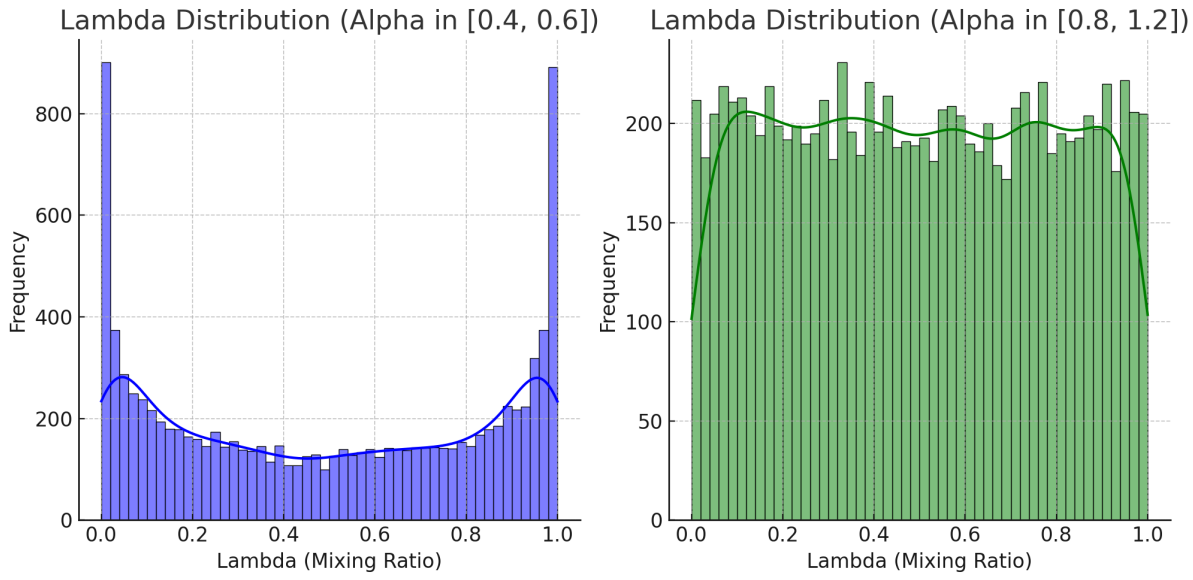
MixUp generates **virtual training samples** by mixing two images and their labels in a weighted manner. How it works:

- **Randomly select another image** from the batch.
- **Blend the two images** using a mix ratio **lambda** (sampled from a **Beta(α, α) distribution**).
- **Blend the labels** using the same mix ratio.

CutMix creates augmented samples by **cutting a region from one image** and **replacing it with a patch from another image**. How it works:

- **Randomly select another image** from the batch.
- **Randomly select a rectangular region** within the first image.
- **Replace that region** with a patch from the second image.
- **Recalculate lambda** (sampled from a **Beta(α, α) distribution**) based on the area of the replaced region.

I use Mix_prob to control the probability of both methods, and only one of them would be used in a batch. I set MixUp_alpha = random(0.4, 0.6) and CutMix_alpha = random(0.8, 1.2), and their beta (lambda) distribution is shown as follow:



Model Architecture & Training Strategy

I've tried different pre-trained backbones, but they shared the same classification head:

```
nn.Linear(in_features, 1024),
nn.Dropout(0.3),
nn.ReLU(),
nn.Linear(1024, num_classes)
```

Training Setup

Optimizer	`optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0001, weight_decay=1e-2)`
Scheduler	`optim.lr_scheduler.CosineAnnealingWarmRestarts( optimizer, T_0=schedulerT_0, T_mult=schedulerT_mult )`
Criterion	`nn.CrossEntropyLoss()`
LR	Tried on 1e-3 and 1e-4, **1e-4 was more suitable.**
Batch size	Tried on 128, 64, 32. **Choose 32 as default**, but Val Acc tended to converge faster when batch size was big.

I stored the model parameters with the highest Val Acc. For the best model set up, I reached leaderboard Acc 0.93 and Val Acc 0.9067 with backbone: `timm.seresnextaa101d_32x8d.sw_in12k_ft_in1k_288` at 50th epoch.

Experiment

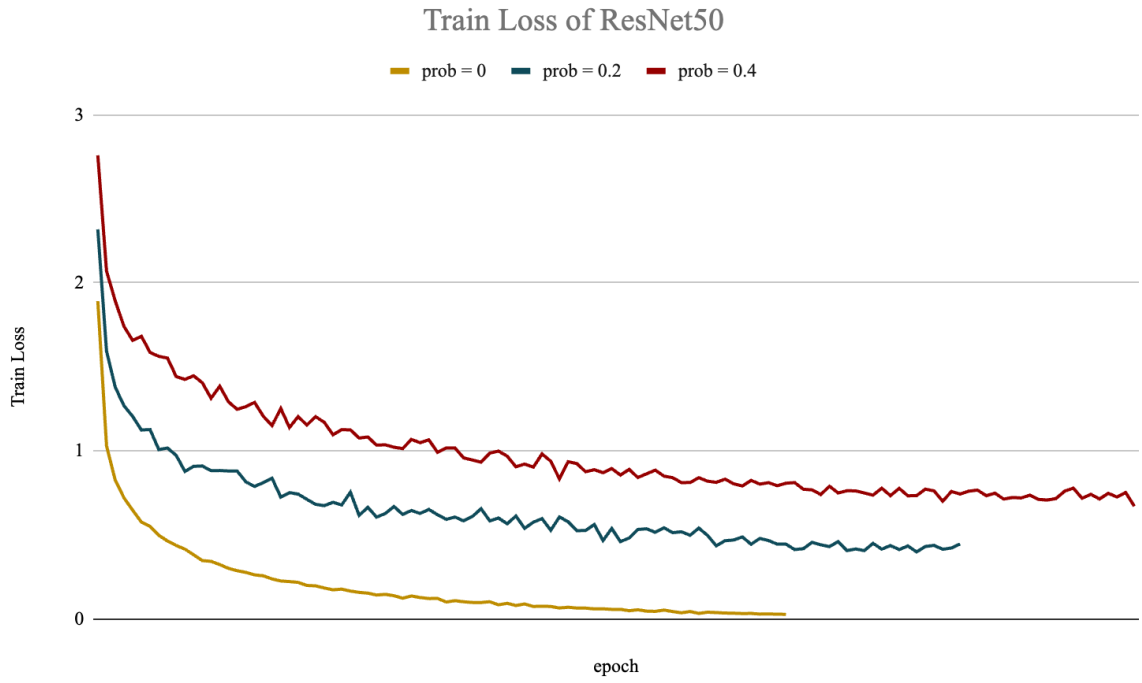
This experiment evaluates the effect of different mix augmentation probabilities and pretrained backbones on model performance. The probability (`prob`) represents the likelihood of applying MixUp and CutMix, meaning the probability of using the original data is `1 - 2 * prob`. The model architecture remains unchanged across all experiments.

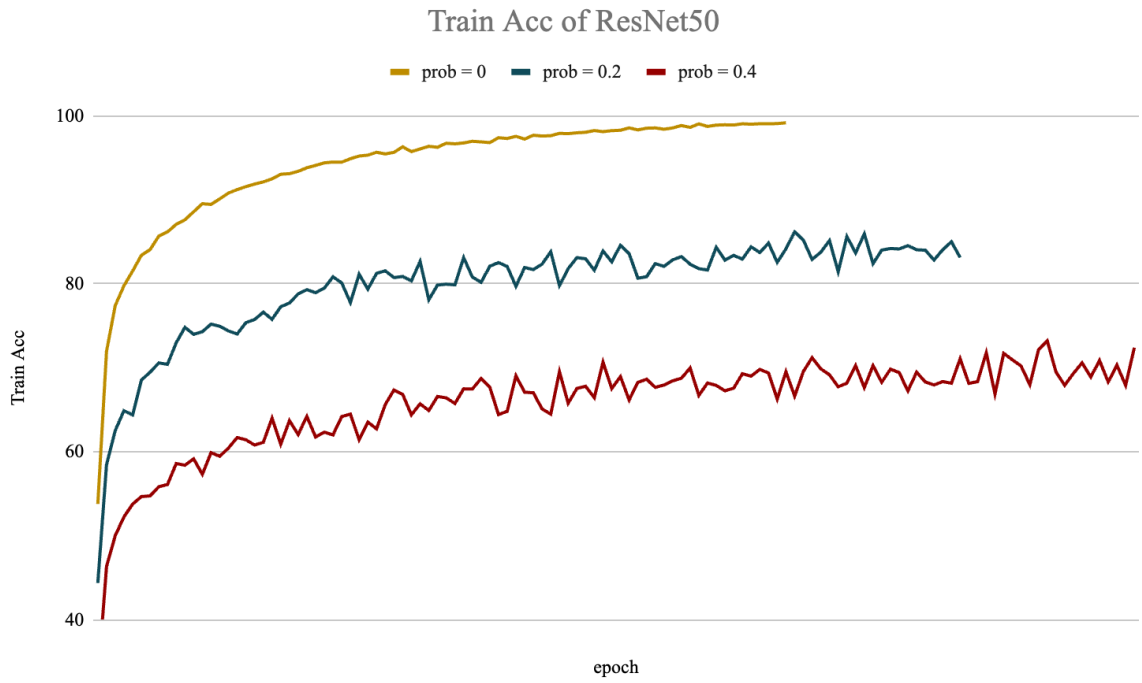
In conclusion, a probability of 0.4 achieved the best performance on the leaderboard data, while probabilities above 0.3 consistently reached the highest validation accuracy. SE blocks can accelerate convergence but wouldn't guarantee performance enhancement. As augmentation increased, training loss increased and training accuracy decreased. An interesting observation is that validation accuracy converged early in training, with higher `prob` values leading to faster convergence. ResNet50 and ResNeXt50 (trained for 200 epochs) exhibited nearly identical performance, as did ResNet101 and ResNeXt101 (trained for over 300 epochs). However, the 101-series models did not provide noticeable improvements over the 50-series while significantly increasing training time and reducing the

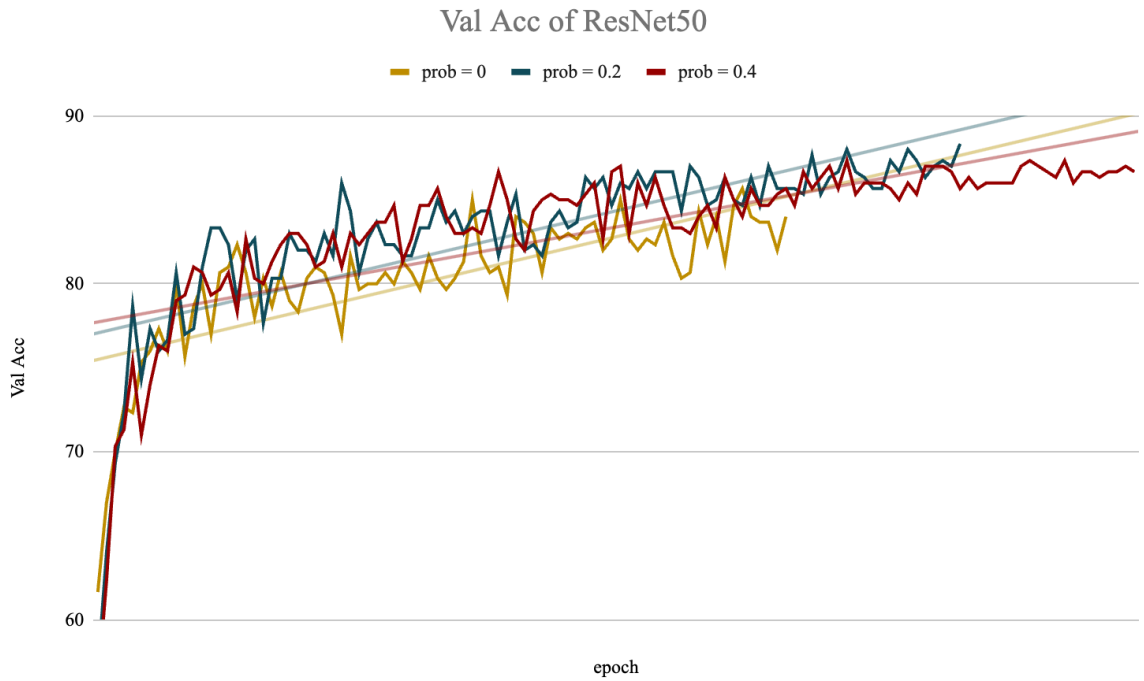
convergence rate. Thus, using deeper architectures may not be beneficial given the trade-off in efficiency.

ResNet50 Result

- Backbone: timm.resnet50.fb_swsl_ig1b_ft_in1k

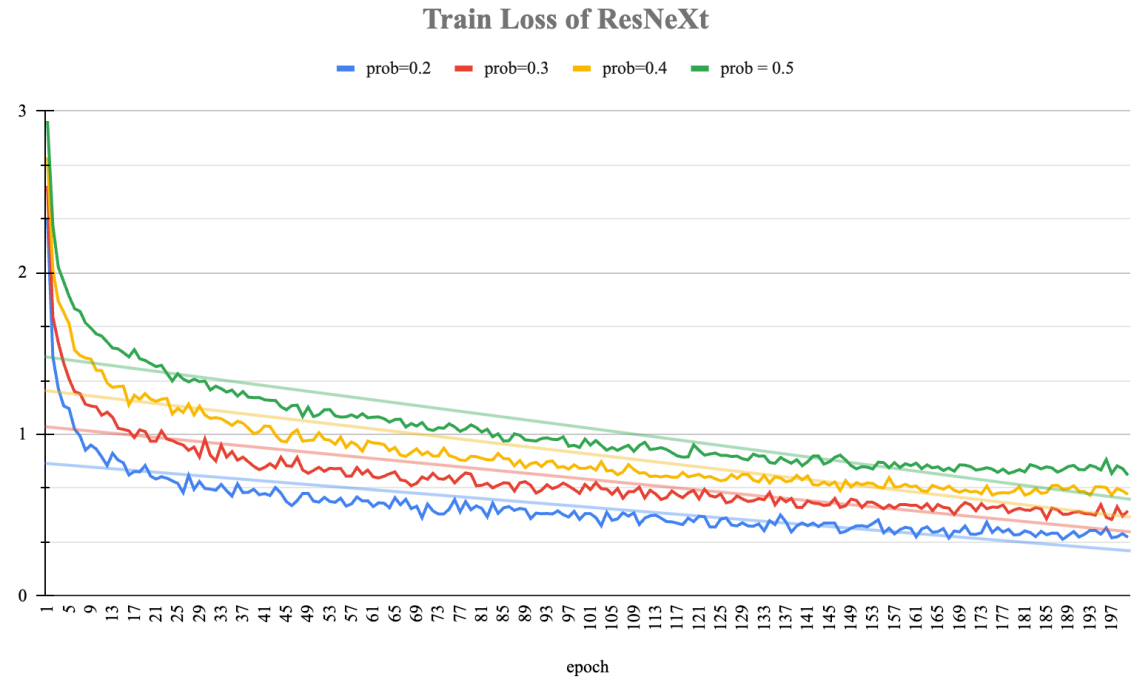


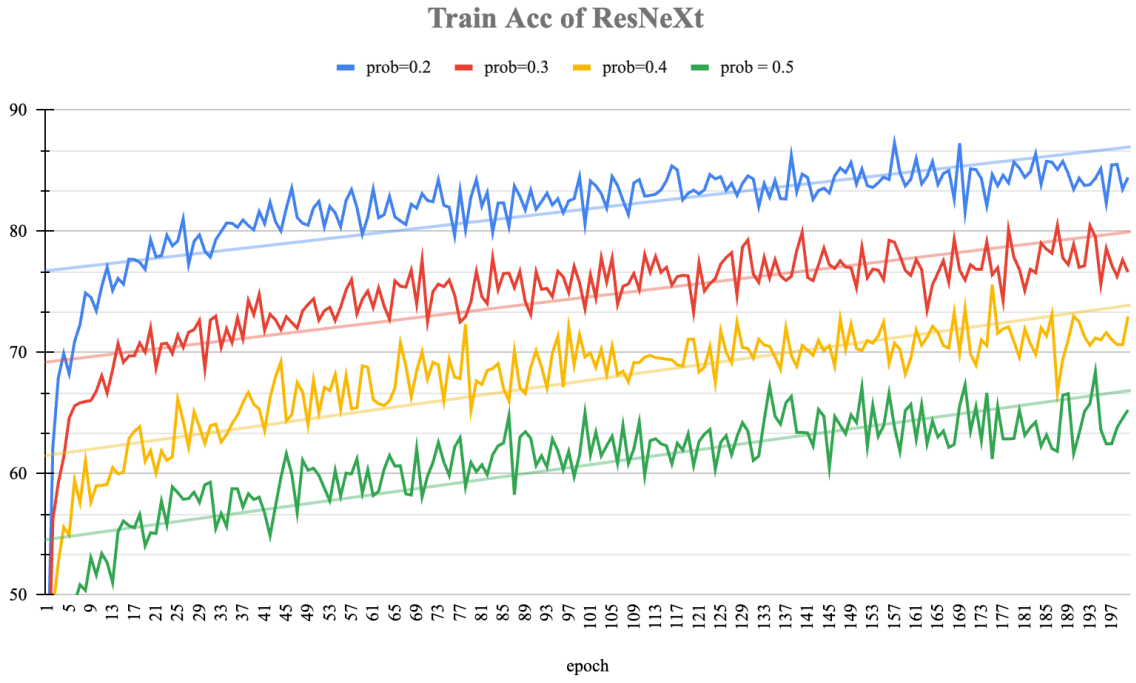


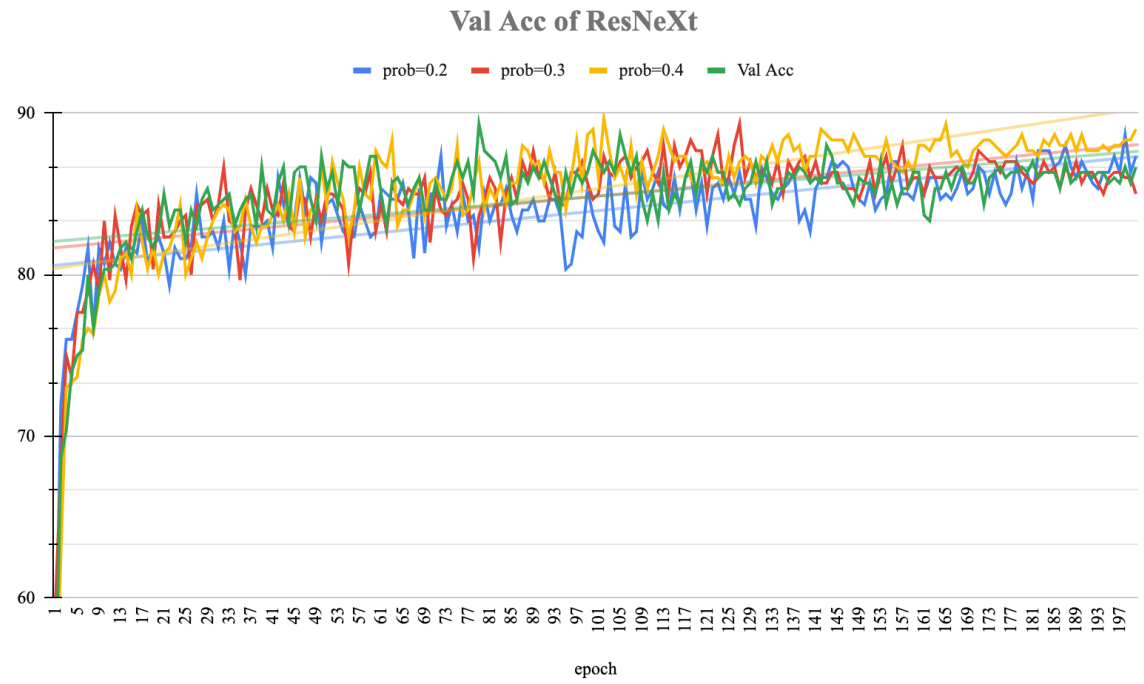

Val Acc of ResNet50
prob = 0
prob = 0.2
prob = 0.4
90
80
70
60
Val Acc
epoch

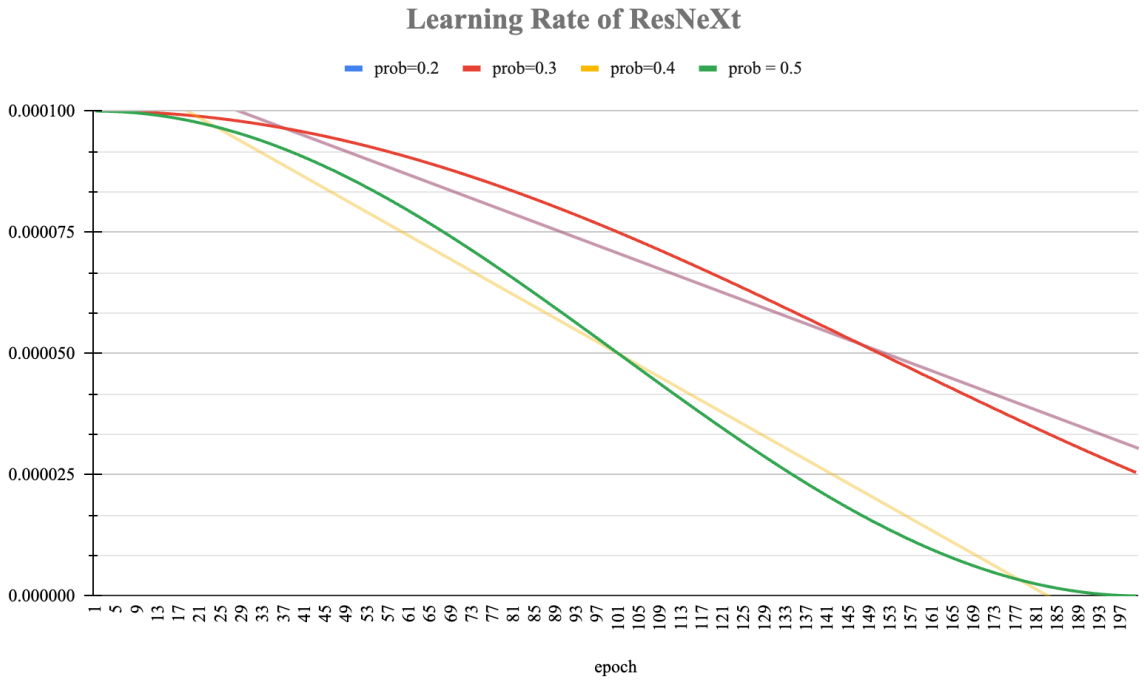
ResNext50 Result

- Backbone: timm.resnext50_32x4d.fb_swsl_ig1b_ft_in1k






Val Acc of ResNeXt
prob=0.2
prob=0.3
prob=0.4
Val Acc
90
80
70
60
epoch


Learning Rate of ResNeXt
prob=0.2
prob=0.3
prob=0.4
prob = 0.5
0.000100
0.000075
0.000050
0.000025
0.000000
epoch

Integrated Comparison

When comparing **ResNeXt50**, **ResNeXt101**, and **SEResNeXt101** with a mix probability of 0.4, we observe that backbones incorporating **SE blocks** tend to converge faster in **validation accuracy** but slower in **training loss** and **training accuracy**. This pattern is consistent across different model sizes, as **ResNeXt101** exhibits the same relationship when compared to **ResNeXt50**.