# Visual Recognitionusing Deep Learning

## Lab1 report

### 110550130 劉秉驊

### My github

1. https://github.com/Potato-TW/visual_dl/tree/main

### Introduction

In this homework, we use resnext101_32x8d.fb_swsl_ig1b_ft_in1k with pre-trained weights from timm to classify 100 kinds of plants.

We implement several methods to raise accuracy and suppress loss on valid like adjustment of hyperparameters, different transformation of images, and even freezing layers in model.

We hope our fine-tuned model can reach higher performance than official training.

### Method

1. Choosing model
   Our model is resnext101_32x8d.fb_swsl_ig1b_ft_in1k.
   In timm leaderboard of image classification, this model gets 90% in top-5 accuracy with only 88M parameters.
   It's really powerful.

| model | img_size | avg_top1 | avg_top5 | imagenet_top1 | imagenet_top5 | real_top1 | real_top5 | v2_top1 | v2_top5 |
|---|---|---|---|---|---|---|---|---|---|
| resnetrs200.tf_in1k | 256 | 66.02 | 80.95 | 83.89 | 96.76 | 88.29 | 97.94 | 73.39 | 91.28 |
| resnetrs200.tf_in1k | 320 | 65.66 | 80.46 | 84.45 | 97.07 | 88.75 | 98.10 | 74.19 | 91.86 |
| resnext101_32x8d.fb_swsl_ig1b_ft_in1k | 224 | 76.11 | 90.80 | 84.28 | 97.17 | 88.76 | 98.14 | 75.46 | 92.68 |
| resnext101_32x8d.fb_wsl_ig1b_ft_in1k | 224 | 75.09 | 90.09 | 82.72 | 96.62 | 88.17 | 97.86 | 73.71 | 92.18 |
| resnext101_32x8d.tv2_in1k | 224 | 64.62 | 79.36 | 82.83 | 96.23 | 87.32 | 97.56 | 72.24 | 90.16 |
| resnext101_32x8d.fb_ssl_yfcc100m_ft_in1k | 224 | 64.61 | 80.98 | 81.62 | 96.04 | 86.80 | 97.47 | 71.43 | 90.31 |
| resnext101_32x8d.tv2_in1k | 176 | 64.22 | 79.28 | 81.99 | 95.72 | 86.65 | 97.20 | 71.29 | 89.31 |
| resnext101_32x8d.tv_in1k | 224 | 60.87 | 77.03 | 79.31 | 94.52 | 85.22 | 96.44 | 67.76 | 87.35 |
| seresnext101_64x4d.gluon_in1k | 224 | 63.24 | 78.63 | 80.89 | 95.31 | 85.95 | 96.98 | 70.47 | 89.18 |
| resnetrs152.tf_in1k | 256 | 64.90 | 80.04 | 82.89 | 96.29 | 87.69 | 97.52 | 71.84 | 90.20 |
| resnetrs152.tf_in1k | 320 | 64.85 | 79.75 | 83.71 | 96.61 | 88.26 | 97.73 | 72.90 | 91.21 |
| resnext101_64x4d.tv_in1k | 224 | 65.17 | 80.10 | 82.98 | 96.25 | 87.72 | 97.58 | 72.20 | 90.61 |

## Model Details

- **Model Type:** Image classification / feature backbone
- **Model Stats:**
  - Params (M): 88.8
  - GMACs: 16.5
  - Activations (M): 31.2
  - Image size: 224 x 224

2. Image transformation
   Here is model configure. Model takes 224 $*$ 224 images as input.

| | 0 |
|---|---|
| url | https://download.pytorch.org/models/resnext101_32x8d-8ba56ff5.pth |
| hf_hub_id | timm/ |
| custom_load | False |
| input_size | (3, 224, 224) |
| fixed_input_size | False |
| interpolation | bilinear |
| crop_pct | 0.875 |
| crop_mode | center |
| mean | (0.485, 0.456, 0.406) |
| std | (0.229, 0.224, 0.225) |
| num_classes | 1000 |
| pool_size | (7, 7) |
| first_conv | conv1 |
| classifier | fc |
| license | bsd-3-clause |
| origin_url | https://github.com/pytorch/vision |

Then we implement several motheds on training data, but only crop validation data in center to $224 * 224$:

1. Random crop $224 * 224$
2. Horizonal, vertical flip
3. Rotation 30 degree

4. Gaussian blurring

visual_dl [WSL: Ubuntu] - 9.ipynb

```python
import torchvision.transforms.v2 as T

train_tfms = T.Compose([
    T.RandomResizedCrop(224),
    T.RandomHorizontalFlip(p=0.8),
    T.RandomVerticalFlip(p=0.3),

    T.RandomRotation(30),
    T.RandomApply([T.GaussianBlur(3)], p=0.5),
    T.ToImage(),
    T.ToDtype(torch.float32, scale=True),
    T.Normalize(*norm_stats)
])


valid_tfms = T.Compose([
    T.Resize(train_sz, antialias=True),
    T.CenterCrop(int(train_sz * 0.875)),
    T.ToImage(),
    T.ToDtype(torch.float32, scale=True),
    T.Normalize(*norm_stats)
])
```

3. Hyperparameter

    1. batch size: 96
       (training on 24G vram of L4 GPU)
    2. learning rate: 0.003

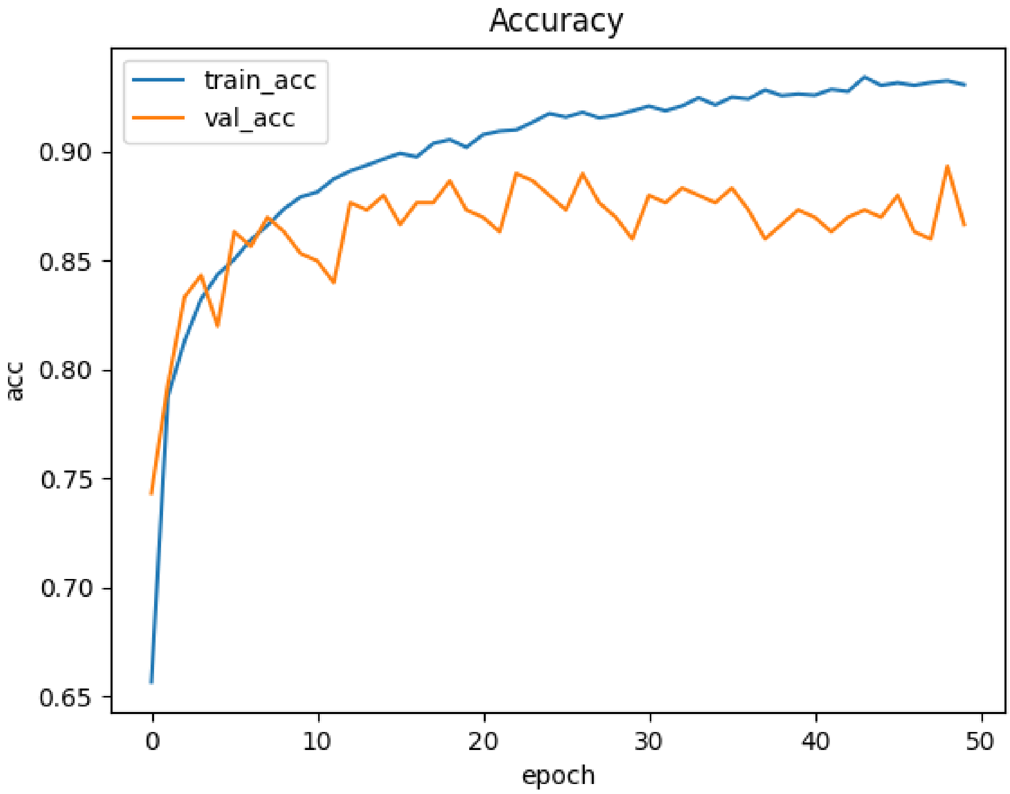
3. epochs: 50

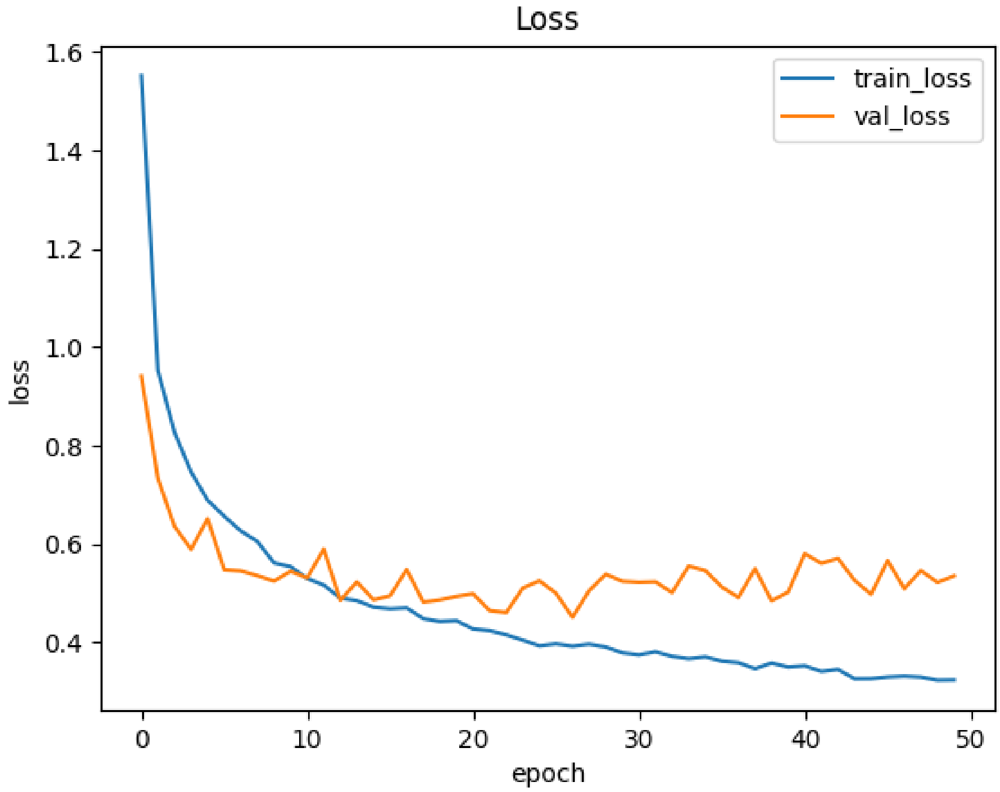
visual_dl [WSL: Ubuntu] - 9.ipynb

```
# Learning rate for the model
lr = 1e-3

# Number of training epochs
epochs = 50

# AdamW optimizer; includes weight decay for regularization
optimizer = torch.optim.AdamW(model.parameters(), lr=lr, eps=1e-5)

# Learning rate scheduler; adjusts the learning rate during training
lr_scheduler = torch.optim.lr_scheduler.OneCycleLR(
                            optimizer,
                            max_lr=lr,
                            total_steps=epochs*len(train_dataloader))
```

## Results

1. After training, we can see the trend of accuracy and loss below.
   When in $48^{th}$ epoch, we get the lowest loss 0.155 on training and 0.543 on validation, and meanwhile we get 0.88 of accuracy on validation.
   Finally, we obtain 0.94 of accuracy on test case.




2. At the same time, we also capture the $49^{th}$ epoch of highest accuracy.
   0.153 on training loss
   0.552 on validation loss
   0.887 on validation accuracy

   As the result, on test case, we surprisingly get as the same score as we choose lowest loss.

We think the performance of lowest loss and highest accuracy is the same, which even they have different accuracy on validation data.

## Additional experiments

1. Freeze sensitive layers
   First we need to define sensitive layers.
   We use validation data to compute the loss of each layer in the original model.

```
[('layer4.1.conv1.weight', 22.528321981430054),
 ('layer4.2.conv3.weight', 13.603535413742065),
 ('layer4.0.downsample.0.weight', 9.315202116966248),
 ('layer4.2.conv1.weight', 7.713109374046326),
 ('fc.weight', 7.454432964324951),
 ('layer4.0.conv3.weight', 6.771357178688049),
 ('layer4.0.conv1.weight', 6.156463623046875),
 ('layer4.1.conv2.weight', 5.513914704322815),
 ('layer4.0.conv2.weight', 5.070958375930786),
 ('layer3.8.conv1.weight', 5.070551514625549),
 ('layer3.0.conv1.weight', 5.056488037109375),
 ('layer2.0.conv1.weight', 4.999824523925781),
 ('layer4.0.downsample.1.bias', 4.8650208711624146),
 ('layer2.1.conv2.weight', 4.862960934638977),
 ('layer4.2.bn1.bias', 4.816706299781799),
 ('layer3.10.conv2.weight', 4.806696653366089),
 ('layer3.19.conv1.weight', 4.786763548851013),
 ('layer4.0.bn1.weight', 4.784423828125),
 ('layer4.0.bn3.bias', 4.7754669189453125),
 ('layer2.0.conv3.weight', 4.714258670806885),
 ('layer3.5.conv1.weight', 4.7090911865234375),
 ('layer3.1.conv3.weight', 4.708287477493286),
 ('layer3.12.conv1.weight', 4.692952394485474),
 ('layer3.0.bn1.bias', 4.682795286178589),
 ('layer3.0.bn1.weight', 4.67974853515625),
 ('layer3.4.conv1.weight', 4.677983641624451),
 ('layer3.20.conv1.weight', 4.677922487258911),
 ('layer3.16.bn3.bias', 4.677424073219299),
 ('layer3.1.bn3.bias', 4.677154541015625),
 ('layer3.15.conv1.weight', 4.675959348678589),
```

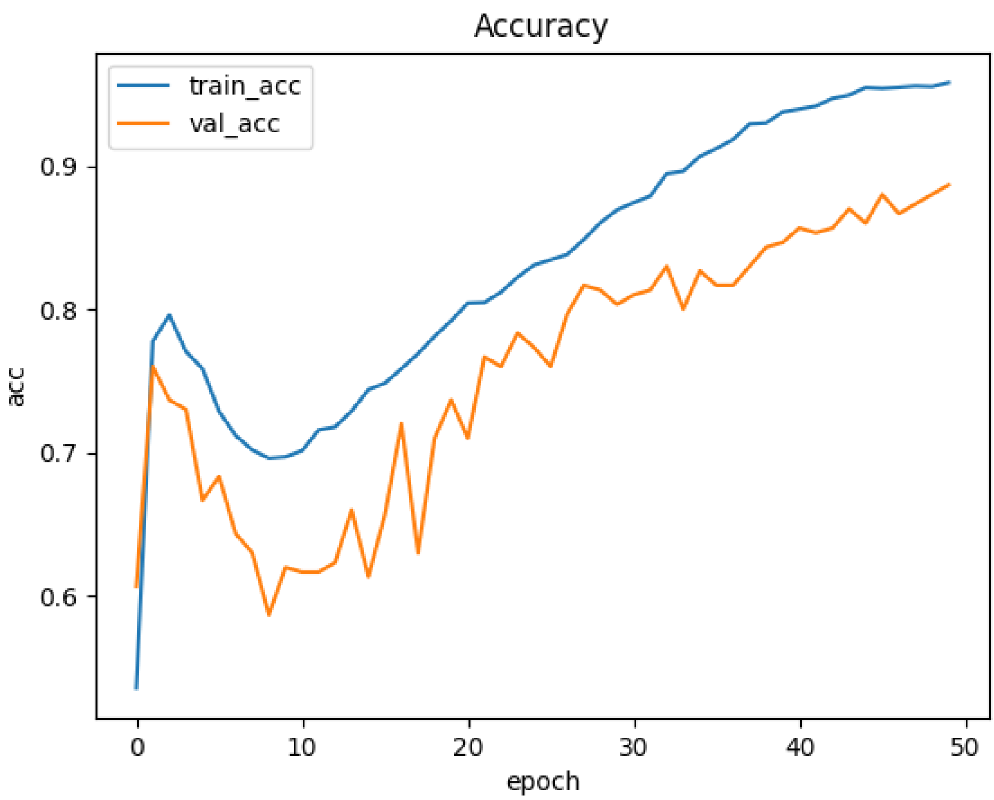
   We freeze the top 2 sensitive layers, expecting this model can keep the capibility of learning images.
   Don't be easily affected by training data.

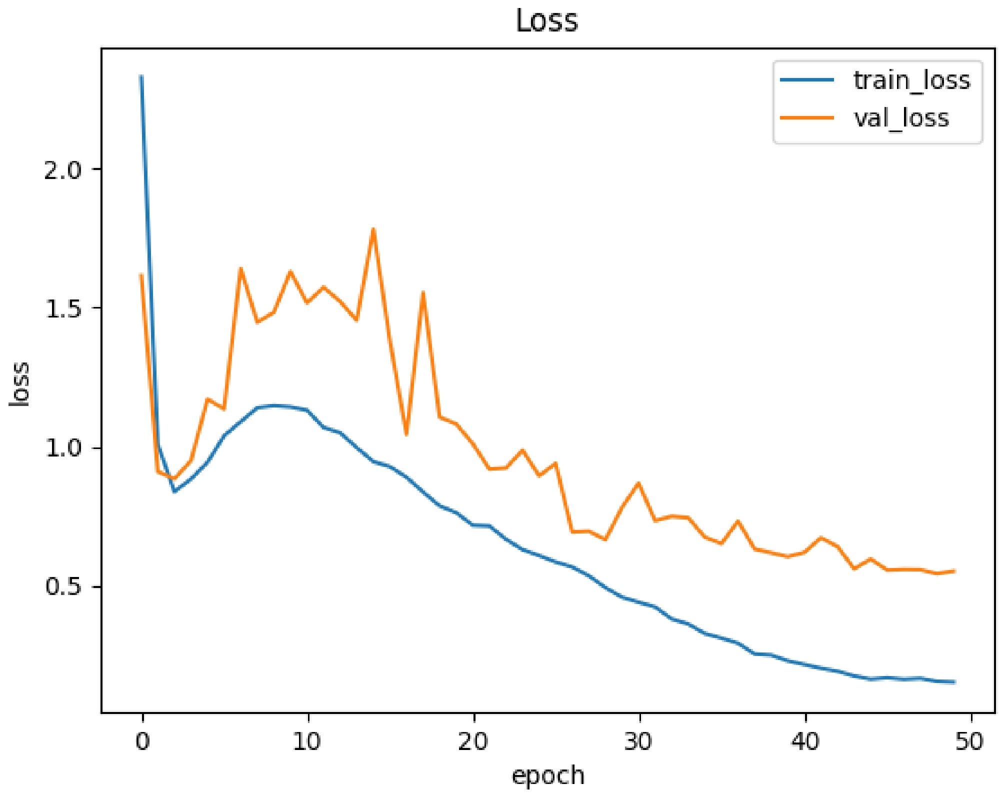
visual_dl [WSL: Ubuntu] - 8.ipynb

```python
protected_layers = [model.layer4[1].conv1, model.layer4[2].conv3]

# 凍結指定層的參數更新
for layer in protected_layers:
    for param in layer.parameters():
        param.requires_grad_(False)  # 關閉梯度計算
        param.detach_()  # 斷開計算圖(可選強化操作)

# 驗證凍結狀態
frozen_params = [
    (name, param.requires_grad)
    for name, param in model.named_parameters()
    if any(layer in name for layer in ['layer4.1.conv1', 'layer4.2.conv3'])
]

print(pd.DataFrame(frozen_params, columns=["Parameter", "Trainable"]))
```

We can se the result below.

Accuracy and loss move up then down sharply. But this model can still reach nice accuracy compared with non freezed model, and it even tries to reach lower loss.






We can say this freeze method gets better results than we think.

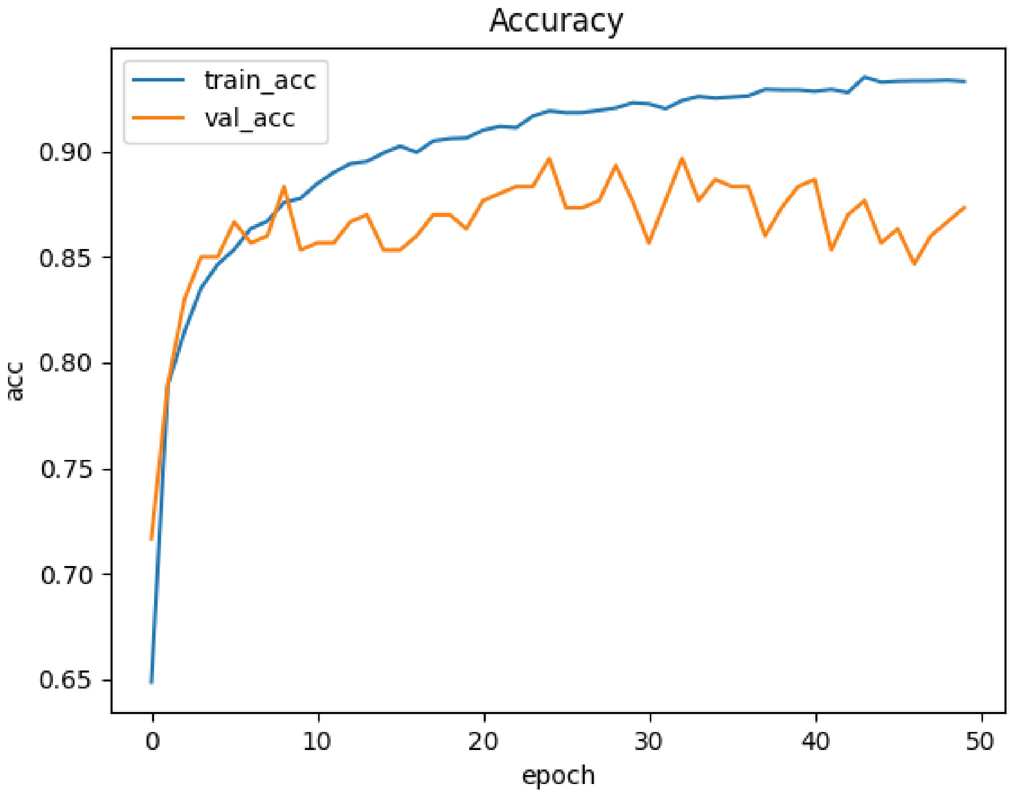
2. Set large learning rate on sensitive layers
   Opposite to freezing method, instead of just freezing layers, we set higher learning rate on those layers, hoping they can learn faster than insensitive layers.

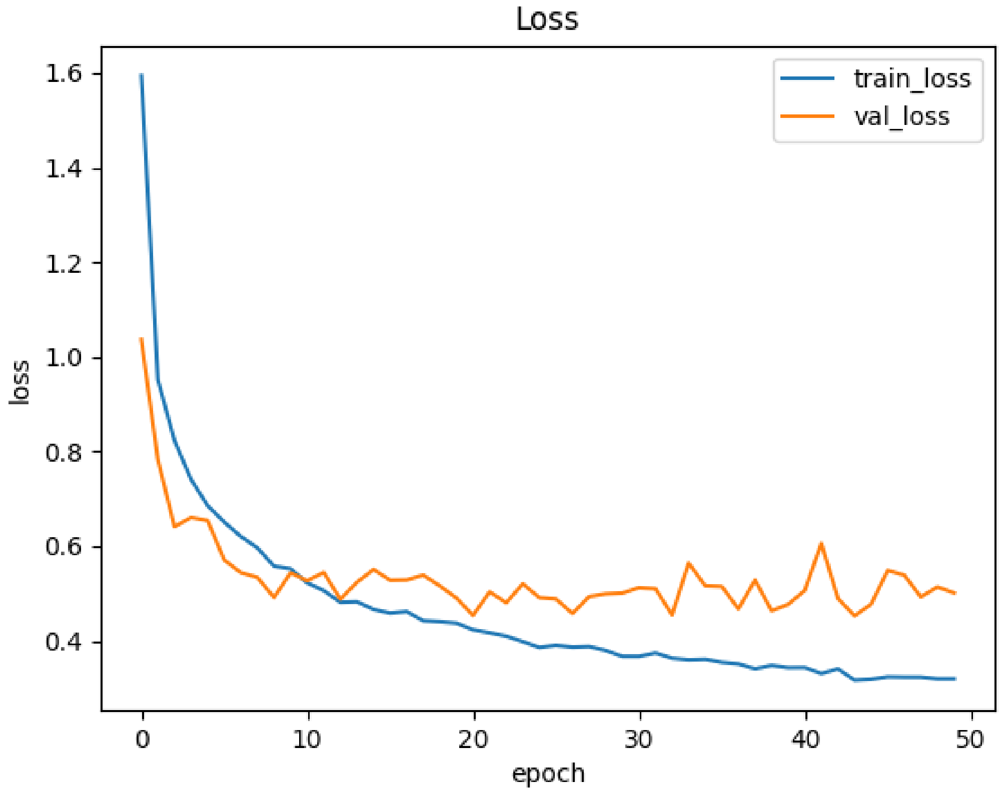
| **sensitive layer** | **Top 1** | **Top 2** |
|---|---|---|
| learning rate | 0.003 | 0.0008 |
| weight decay | 0.005 | 0.007 |

visual_dl [WSL: Ubuntu] - 7.py

```python
# 重構優化器配置
optimizer = torch.optim.AdamW([
    {'params': model.layer4[1].conv1.parameters(),
     'lr': 1e-3, 'weight_decay': 0.005},  # 高敏感層
    {'params': model.layer4[2].conv3.parameters(),
     'lr': 8e-4, 'weight_decay': 0.007}   # 次敏感層
], eps=1e-5)

# 添加梯度裁剪
torch.nn.utils.clip_grad_norm_(
    model.layer4[1].conv1.parameters(),
    max_norm=2.0
)
```

As we can see the result, There is no sharp up and down compared with freeze layers.
However, it loses its potential to get higher accuracy, which both oscillate around that seems to overfit.




## References


1. Code template
2. Billion-scale semi-supervised learning for image classification
3. Aggregated Residual Transformations for Deep Neural Networks

4. Deep Residual Learning for Image Recognition