Homework 3

Adriana Vlad, MIAO2

The following pipeline includes all the required features specified in the homework. More specifically, it has the following properties:

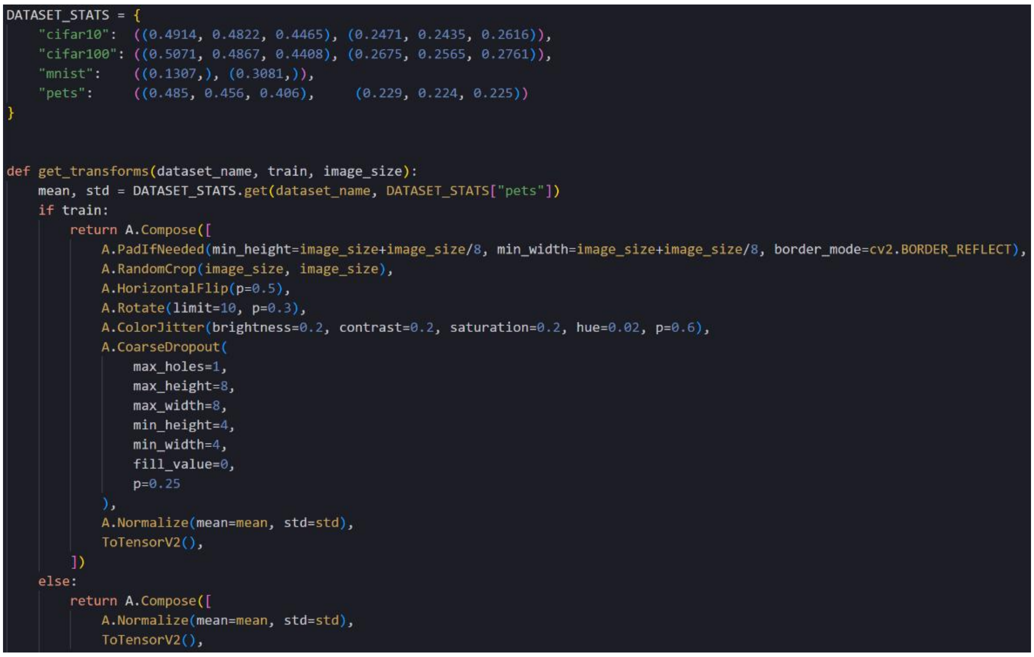
- it is device agnostic: uses gpu (cuda) when available, and cpu otherwise.
- the user is allowed to choose between MNIST, CIFAR-10, CIFAR-100, and OxfordIIITPet datasets, with exact or alternative spellings checked for each.
- the imported datasets inherently support data augmentation. most of the augments are fixed and cannot be changed via in-line parameters, but they do adjust to user selected target image size (images are up/downscaled after augments but before being sent to the main model, should the user choose so). regarding efficiency, the augments are done with 'albumentations', and the data loaders use 'num_workers=2, pin_memory=True'. I also tried adding 'persistent_workers' to the dataloaders with no noticible difference.
- the user can choose between ResNet18, ResNet50, ResNest14d, ResNest26d and a simple MLP, imported with 'timm'. I also gave the user the option to import the models pretrained or not.
- the only tested optimizers were SGD and AdamW, but Adam, Muon and SAM support is also implemented. Modifiable parameters are 'lr', 'weight_decay', and 'momentum' (only for SGD)
- both StepLR and ReduceLROnPlateau are supported, with 'step_size' and 'gamma' / 'patience' and 'factor' as parameters modifiable by the user.
- There exists a batch size scheduler: the user can input starting batch size 'bs'. Every 'bs_growth_interval' epochs batch size increases by 'bs_growth_factor', not exceeding 'max_bs'. All parameters in quotation marks are accepted in-line arguments.
- For metrics reporting i used wandb. There are 2 early stopping mechanisms: first of all, 'ctrl+c' will stop the current run and save results. Secondly, if training accuracy has not improved by at least 0.001 in the last 'early_stop' (user arg, default 5) epochs, results are saved and run is stopped.

Other args include:

- epochs, meaning maxium number of epochs before run stops
- use_tta: true/false, 3 small augments + original. test prediction considered the average prediction between them.
- dropout: added dropout layer before the output layer of the models. this value represents the percentage affected.
- label_smoothing: 'torch.nn.CrossEntropyLoss(label_smoothing=args.label_smoothing)' was used
- mix_prob: percentage of inputs that will be modificed with an equal random choice between MixUp and CutUp (v2 implementations).
- mix_alpha: parameter for MixUp and CutUp

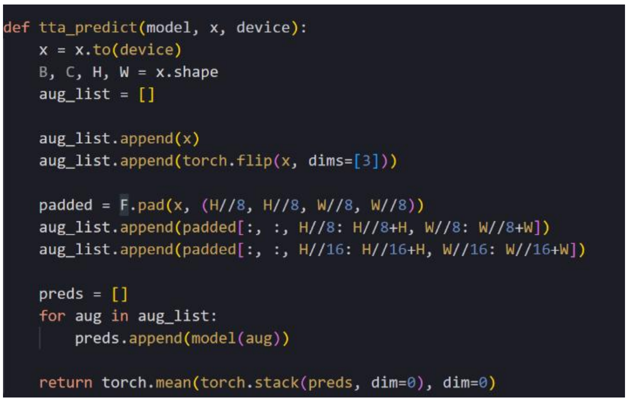
The general pipeline is as follows:

- parsing args
- get datasets, paired with transforms (not yet applied)



```python
DATASET_STATS = {
    "cifar10":  ((0.4914, 0.4822, 0.4465), (0.2471, 0.2435, 0.2616)),
    "cifar100": ((0.5071, 0.4867, 0.4408), (0.2675, 0.2565, 0.2761)),
    "mnist":    ((0.1307,), (0.3081,)),
    "pets":     ((0.485, 0.456, 0.406),     (0.229, 0.224, 0.225))
}


def get_transforms(dataset_name, train, image_size):
    mean, std = DATASET_STATS.get(dataset_name, DATASET_STATS["pets"])
    if train:
        return A.Compose([
            A.PadIfNeeded(min_height=image_size+image_size/8, min_width=image_size+image_size/8, border_mode=cv2.BORDER_REFLECT),
            A.RandomCrop(image_size, image_size),
            A.HorizontalFlip(p=0.5),
            A.Rotate(limit=10, p=0.3),
            A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.02, p=0.6),
            A.CoarseDropout(
                max_holes=1,
                max_height=8,
                max_width=8,
                min_height=4,
                min_width=4,
                fill_value=0,
                p=0.25
            ),
            A.Normalize(mean=mean, std=std),
            ToTensorV2(),
        ])
    else:
        return A.Compose([
            A.Normalize(mean=mean, std=std),
            ToTensorV2(),
```

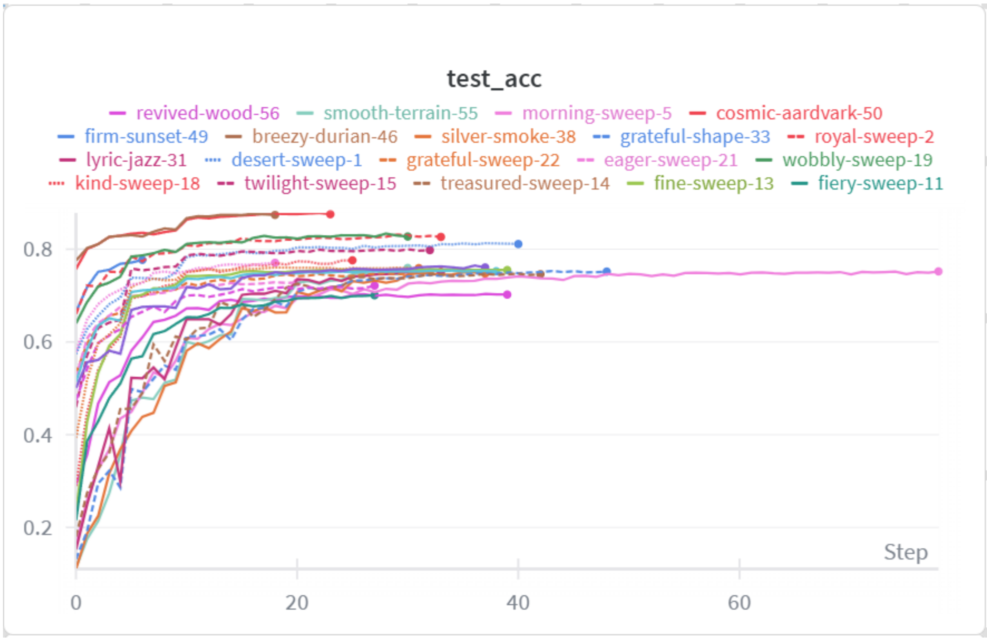
- get model, with upscaler (and grayscale to 3d transform if needed) attached before real model start, and dropout added before last layer
- initialize dataloaders, training functions and wandb
- each epoch,
    - for each batch (with transform applied on get_item): apply mixup/cutmix if needed, send to device, learn with autocast and scaler (amp used, described later), get train acc.
    - for each batch, apply tta if needed and get test acc
    - log results
    - update scheduler and batch_size if needed
    - check early_stop
- log final results

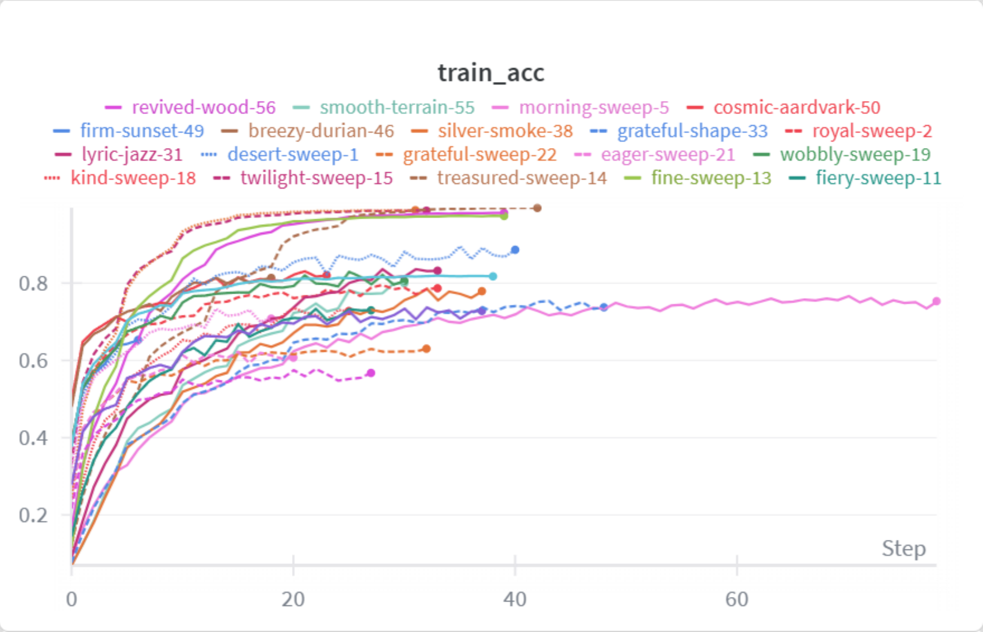


```python
def tta_predict(model, x, device):
    x = x.to(device)
    B, C, H, W = x.shape
    aug_list = []

    aug_list.append(x)
    aug_list.append(torch.flip(x, dims=[3]))

    padded = F.pad(x, (H//8, H//8, W//8, W//8))
    aug_list.append(padded[:, :, H//8: H//8+H, W//8: W//8+W])
    aug_list.append(padded[:, :, H//16: H//16+H, W//16: W//16+W])

    preds = []
    for aug in aug_list:
        preds.append(model(aug))

    return torch.mean(torch.stack(preds, dim=0), dim=0)
```

After finalizing the general pipeline, i created a series of sweeps, for broader to narrower searches, using a sweep.yaml and wandb.

- At first, i decided to try:
    - batch size: 64, 96, 128
    - max batch size: 96, 128, 192
    - dropout: 0, 0.15, 0.25
    - mix_prob: 0, 0.66, 1
    - all models
    - momentum: 0.8, 0.9
    - optimizers: sgd, adamw
    - pretrained: 0, 1 (false, true)
    - scheduler: step, plateau
    - scheduler_patience: 3, 5
    - use_tta: 0, 1
    - weight_decay: 0.0001, 0.0005
- After about 30 runs I saw that some combinations / values would always lead to bad results, and also thought there were other parameters, such as scheduler_step_size, that i should test. I use a few sweeps that i only ran for 5-10 runs to determine what was still worth testing. Considering that pretrained models perform well regardless and I already had a few options, and that i had already found a few options that use sgd (without pretraining) that did well, the last sweep i did was only on adamw and only on no pretraining, with these variations:
    - batch size: 64, 96, 128
    - max baatch size: 64, 96, 128
    - dropout: 0, 0.15, 0.25
    - learning rate: 0.0005, 0.001, 0.0001
    - mix_prob: 0.5, 0.66, 1
    - models: resnest26d, resnest14d, resnet50
    - scheduler: step, plateau
    - scheduler_patience: 3, 5
    - scheduler_step_size: 5, 10
    - use_tta: 0, 1
    - weight_decay: 0.0001, 0.0005

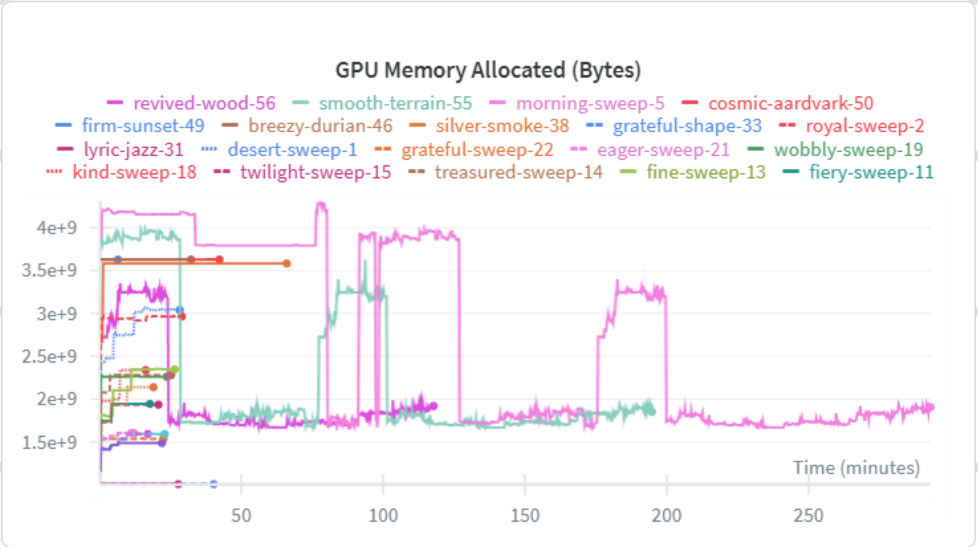




Above are all the tests that achieved >= 70% test accuracy. 24 runs (with different configurations) from a total of about 200 attempts(considering all sweeps, manual tests, and early fails)

Regarding efficiency

- amp is used: autocast during training and GradScaler for updates (loss, optimizer)
- dataloaders use pin_memory and multiple workers.
- augmentations with albumentations, faster than most alternatives
- works with small upscales, meaning even 64x64 images, and can train in less than 30 minutes on batch_size=96.



Each run uses at most 4 GB of VRAM, and even that is rare. The average is less than 3 GB

Without pretraining, the biggest accuracy achieved was 75.95% in 28 minutes, using:

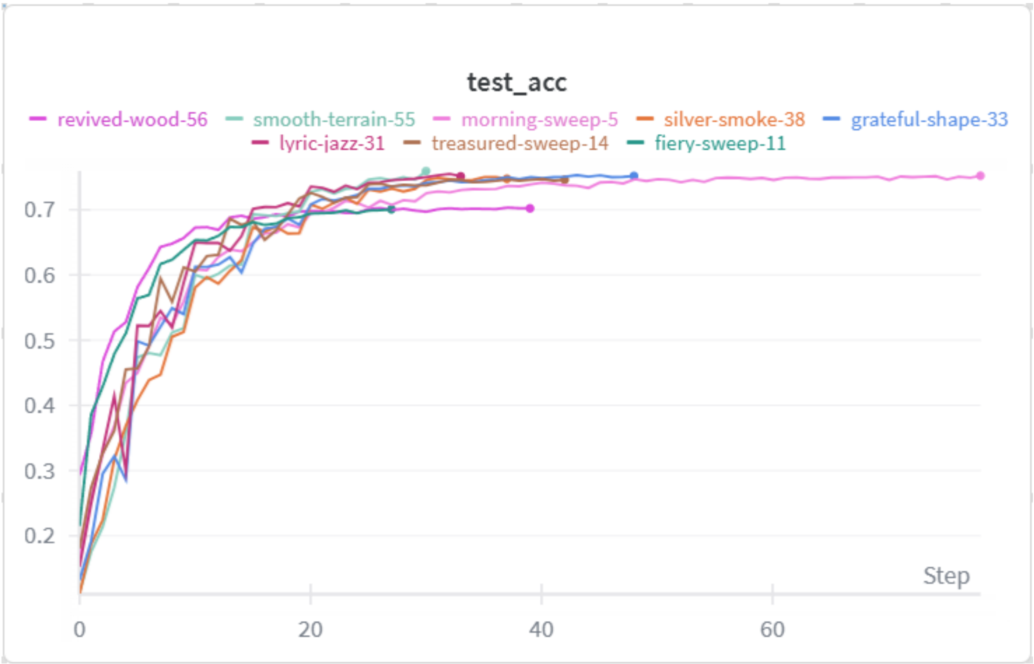
--dataset cifar100 --model resnest26d --epochs 50 --bs 96 --lr 0.05 --opt sgd --scheduler step --scheduler_step_size 5 --scheduler_gamma 0.5 --momentum 0.9 --weight_decay 0.0005 --max_bs 96 --use_tta 0 --dropout 0.25 --label_smoothing 0.1 --mix_prob 0.66 --mix_alpha 0.5 --target_size 64

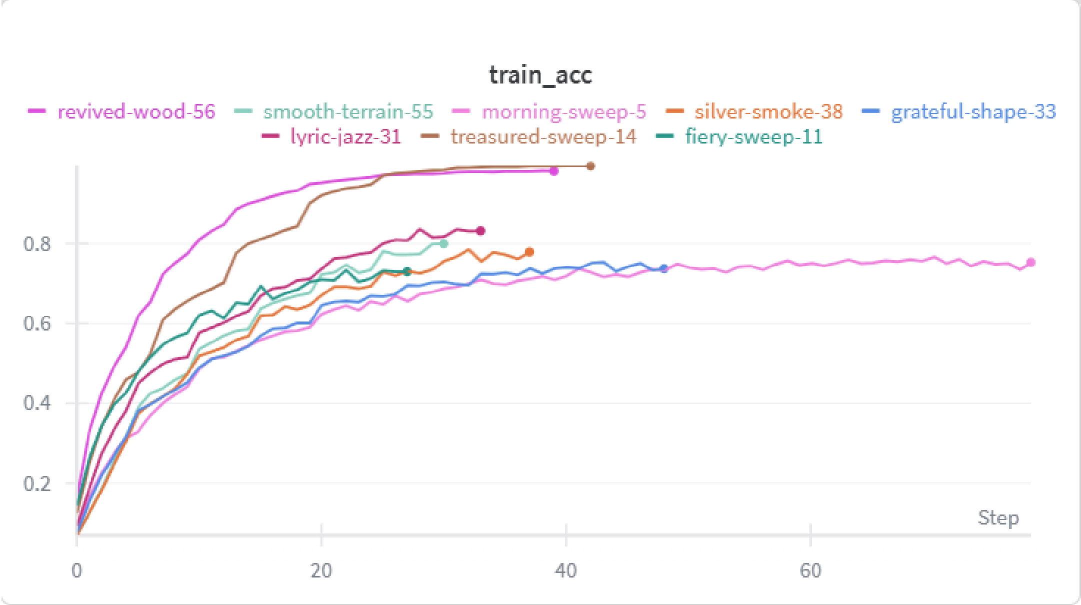
Some variations of this configuration also achieve >= 70%:

- mix_prob 0.66-1.0
- label_smoothing 0.1-0.2
- <= 0.25 dropout
- with/without tta
- 96-128 batch size
- step size 10, gamma 0.1-0.2

Other observations:

- without mixup/cutmix, the models tend to overfit regardless of other parameters. but even then, a few (2 out of the 8 shown here) manage to get good results, though the difference between training and test acc is much bigger than in most configurations.
- with appropriate lr changes, adamw and sgd perform about the same. sgd seems more versatile (works in more configurations) when there is no pretraining, and adamw got better results more frequently with pretraining.
- generally, the models performed best on 64x64 with no pretraining, and slightly worse on bigger upscales. These attempts struggled to achieve over 70% accuracy, though a few of them did.
- the ResNest26d model was almost always better than the others, if all other parameters stayed the same.





run_id	name	test_acc	train_acc	runtime_sec
1qly4w3y	fiery-sweep-11	0.7007	0.72964	1072
1laxl8xz	treasured-sweep-14	0.746	0.99468	1522
mwez3rli	dazzling-glitter-27	0.7463	0.83992	1914
xkkh4vqw	lyric-jazz-31	0.7511	0.83218	1679
9lkmksaa	grateful-shape-33	0.7517	0.73738	2429
wb1e36r1	silver-smoke-38	0.7474	0.7792	3964
qkfdi1ti	morning-sweep-5	0.7521	0.7534	4776
endi7dpp	smooth-terrain-55	0.7595	0.79978	1681
gshlmuxx	revived-wood-56	0.7021	0.98174	1420

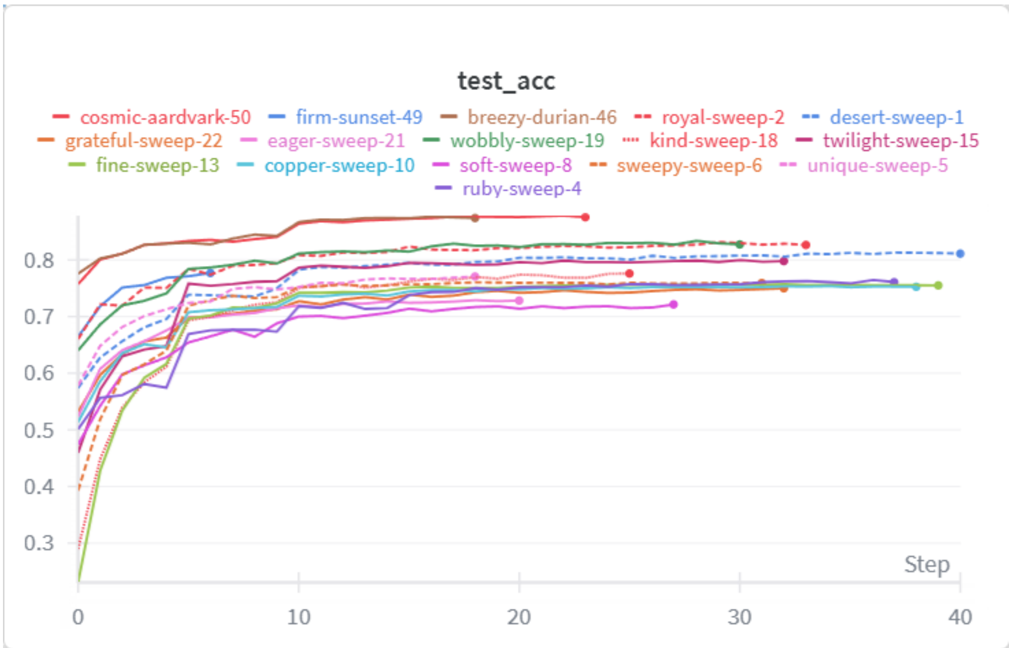
With pretraining, the biggest accuracy achieved was 87.54% in 42 minutes, using:

```
--dataset cifar100 --model resnet50 --epochs 100 --bs 96 --lr 0.001 --opt adamw --scheduler step --scheduler_step_size 10 --scheduler_gamma 0.1 --momentum 0.9 --weight_decay 0.0001
```

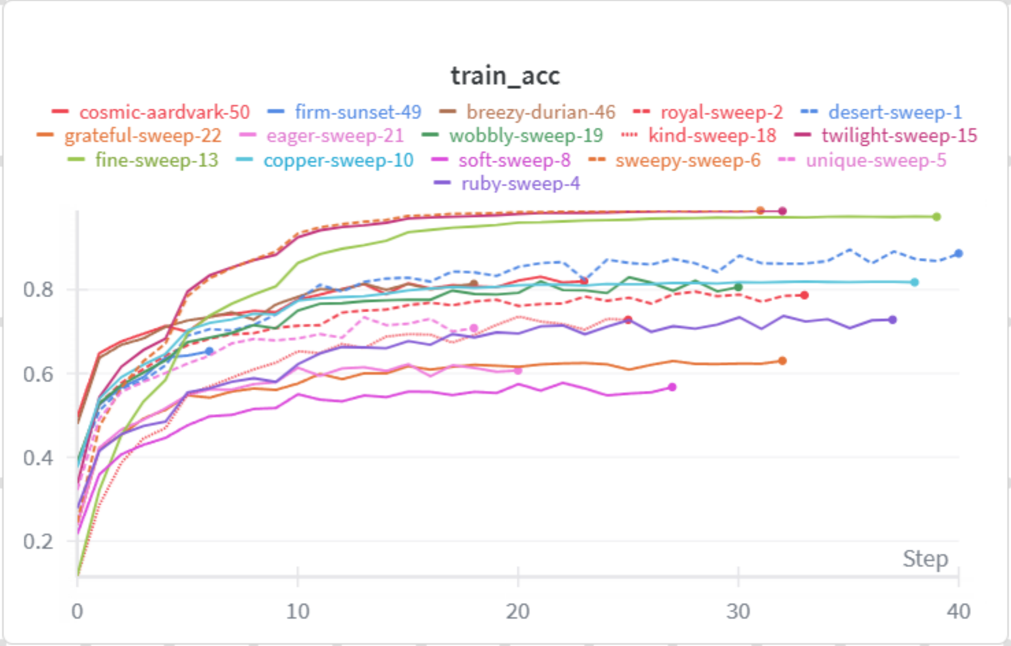
--max_bs 96 --use_tta 1 --dropout 0.25 --label_smoothing 0.1 --mix_prob 1.0 --mix_alpha 0.5 --target_size 128

Other observations:

- here the vast majority of successful runs used 0.25 dropout
- bigger batch sizes (192, instead of 96-128) still achieved good results, unlike before
- about 1/4 of the runs use no mixup/cutmix, same as before
- plateau only used in 2 runs, indicating it wasn't preferred.
- most runs used step 5 with gamma 0.5, but the best result was achieved with step 10 gamma 0.1
- generally, the models performed best on 128x128 with pretraining, and slightly worse on smaller upscales. Almost all attempts with pretraining achieved over 70% accuracy.
- The ResNet50 model seemed the best for these attempts



run_id	name	test_acc	train_acc	runtime_sec
hc9wt7ag	ruby-sweep-4	0.7606	0.72818	1334
qi29epix	unique-sweep-5	0.7707	0.70814	688
yvoawv28	sweepy-sweep-6	0.7592	0.98894	1149
61i2ugub	soft-sweep-8	0.7214	0.56752	1023
ie4695il	copper-sweep-10	0.7525	0.81752	1390
baf4xw4u	fine-sweep-13	0.755	0.9738	1601
nrvhbcet	twilight-sweep-15	0.7979	0.98764	1249
x3cjxw04	kind-sweep-18	0.776	0.72788	986
8fax7nip	wobbly-sweep-19	0.8273	0.8061	1427
x0lrqdl9	eager-sweep-21	0.7282	0.6072	747
1auc1dm1	grateful-sweep-22	0.7499	0.63032	1353
y802o5ck	desert-sweep-1	0.8111	0.88634	1707
hgauqrzv	royal-sweep-2	0.8264	0.78674	1754
5lmkb6eh	breezy-durian-46	0.874	0.8135	1949
olpx9kmm	firm-sunset-49	0.7769	0.653	391
cm41336t	cosmic-aardvark-50	0.8754	0.82	2541



Estimated score:

1. 8 / 8
2. 8 / 8
3. 1 / 3
4. 3 / 6

Setup and How to run:

- locally, i put all 4 files in the same folder. to run, enter in terminal at that path: python train_v4.py --dataset cifar100 --model resnest26d etc. this however assumes all dependencies are downloaded.
- on google colab, run like here, and set as defaults desired configuration.
- you may be asked to log into wandb. if unable or unwilling, remove logging with --log 0 or by changing the code and removing wandb. local epoch progress will still be displayed.