Assignment 3 - PyTorch Training Pipeline

Repede Monica-Gabriela

November 23, 2025

1 Introduction

For this assignment, I implemented a generic training pipeline using PyTorch. This pipeline can be configurable through configuration files. It supports different datasets (MNIST, CIFAR-10, CIFAR-100, OxfordIIITPet), models (resnet18, resnet50, resnest14d, resnest26d, MLP), optimizers (SGD, Adam, AdamW, Muon, SAM), learning rate schedulers (StepLR, ReduceLROnPlateau), batch size scheduler and is integrated with TensorBoard and Weights & Biases (wandb). The pipeline also has an early stopping mechanism.

2 Setup + How to run

This project is implemented in PyTorch, and several additional dependencies are required to reproduce the experiments.

2.1 Installation

PyTorch
Install PyTorch using the official installation command corresponding to your system.

Muon Optimizer
The Muon optimizer must be installed from its GitHub repository:

pip install git+https://github.com/KellerJordan/Muon

W&B and TensorBoard
The framework uses Weights & Biases (wandb) for experiment tracking and visualization. Install it together with TensorBoard:

pip install wandb tensorboard

Log in to your wandb account using: *wandb login* and and provide your API key when prompted.

Timm
To load the models, you need to insall also timm with: *pip install timm*.

Optional: if you run on Kaggle, you need to put *pip install protobuf==4.25.3* in order to make the project runnable.

2.2 Running the Pipeline

All experiments can be executed from the **train.ipynb** notebook. Inside the notebook:

1. **Running a predefined configuration**

   Set: *use_config = True* in the first cell to load the parameters from a YAML configuration file. If needed, update the path for the config.yaml in *config_path*.

2. **Running custom experiments (hyperparameter sweeps)**

   Set: *use_config = False* and manually modify the hyperparameters inside the *sweep_config* dictionary provided in the second cell. This allows running experiments with different architectures, optimizers, learning rates, weight decays and schedulers.

2.3 Execution

After completing the steps above, run the notebook cells in order. The training process will start, and all metrics (loss, accuracy, learning rate, etc.) will be automatically logged to wandb and viewable in the dashboard. After that, you will be able to run the project.

3 Requirements

3.1 Hyperparameter sweep

I used wandb sweeps to explore hyperparameters on CIFAR-100:

- Model architectures: resnet50, resnest26d
- Optimizers: SGD, AdamW, SAM(SGD), Muon
- Learning rate (for SGD and SAM): 0.01, 0.005 (with pretrain), 0.025, 0.1, 0.05 (without pretrain)
- Learning rate (for AdamW and Muon): 2e-4, 3e-4, 5e-4, 1e-4 (for both with and without pretrain)
- Weight decay (for SGD and SAM): 5e-5, 1e-4, 1e-3
- Weight decay (for AdamW and Muon): 0.02, 0.05
- Pretraining: True / False

From these sweeps, I selected 8 configurations that achieved $> 70\%$ test accuracy on CIFAR-100.

ID	Model	Pretrain	Optimizer	lr	wd	Scheduler	T Acc	Time	Img size
golden-sweep-10	resnet50	True	SAM	0.005	0.00005	StepLR	0.7423	57m 13s	32
celestial-sweep-8	resnest26d	True	SAM	0.005	0.0001	StepLR	0.7389	59m 53s	32
valiant-sweep-7	resnest26d	True	SAM	0.005	0.00005	StepLR	0.7385	58m 39s	32
legendary-sweep-1	resnest14d	False	SAM	0.1	0.0005	StepLR	0.7234	27m 46s	32
young-sweep-1	resnest26d	False	SAM	0.025	0.0005	StepLR	0.7253	1h 6m 17s	32
vague-yogurt-4	resnest26d	False	SAM	0.05	0.0005	StepLR	0.7249	1h 27m	32
vocal-sun-1 (3.3)	resnet50	True	Muon	0.0005	0.05	ReduceLROnPlateau	0.8542	50m 50s	224
tough-sweep-2	resnet50	True	Muon	0.0001	0.05	StepLR	0.8496	1h 54m 12s (Kaggle)	224

Table 1: Test accuracy

As augmentations, I used Resize, RandomCrop and RandomHorizontalFlip. For non-pretrain I added RandAugment, RandomErasing and ColorJitter.

Normalization: for pretrain models, the mean and standard deviations were according to the pre-trained model; for non-pretrained, the mean and standard deviations were manually calculated.

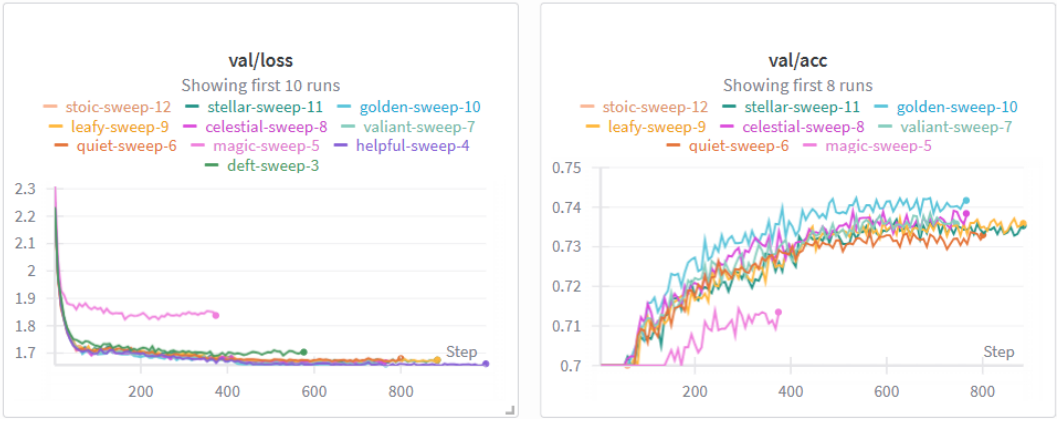

Figure 1: Loss and accuracy for the first 3 models

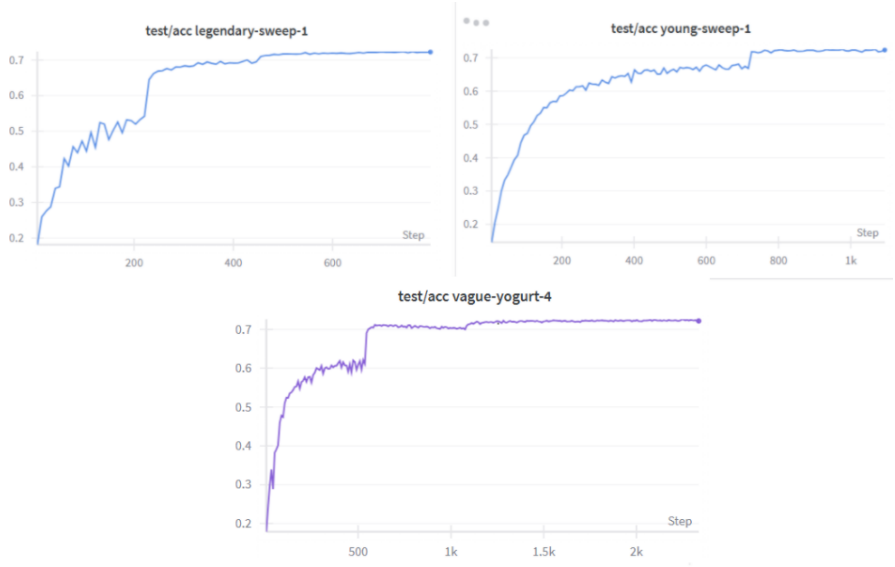

Figure 2: Test accuracy

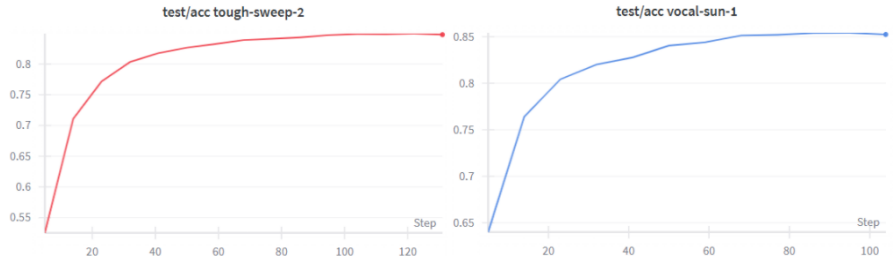

Figure 3: Test accuracy

3.2 Efficient pipeline

To improve the efficiency of the training process, the following optimizations were integrated into the pipeline:

1. The `DataLoader` is configured with `num_workers = 4` to enable parallel data preprocessing, reducing the waiting time between batches. Additionally, `pin_memory = True` accelerates the transfer of data from RAM to GPU.

2. Moving batches to the device with `non_blocking = True` avoids unnecessary memory copies and allows data transfer to overlap with GPU computation.

3. Training with mixed precision (FP16/BF16) reduces memory usage. The use of `GradScaler` ensures numerical stability and prevents gradient underflow.

4. Training is stopped automatically when no significant improvement is observed on the validation set, saving time and computational resources.

5. Using `optimizer.zero_grad(set_to_none = True)` avoids unnecessary gradient accumulation, reduces memory overhead, and improves parameter update efficiency.

6. Applying `torch.jit.script` optimizes the execution graph by removing Python-level overhead in critical sections, leading to faster inference.

Figure 4: Efficiency metrics

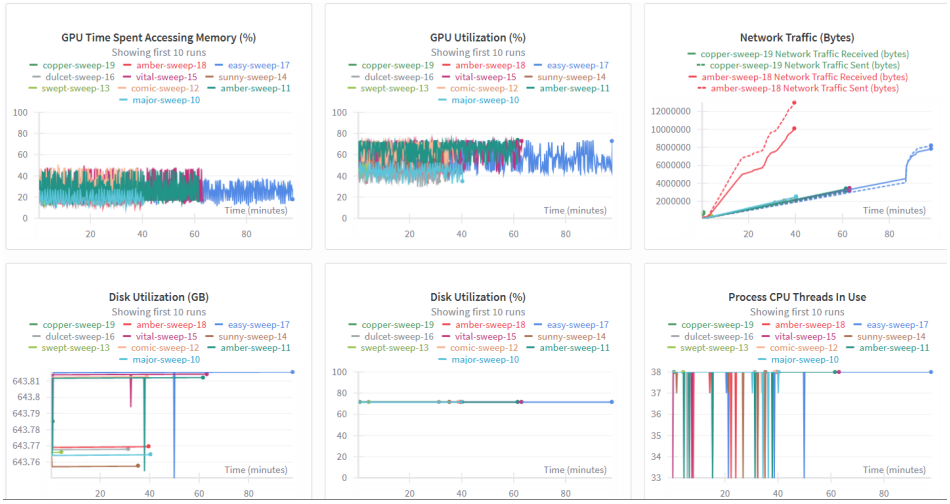

Figure 5: Efficiency metrics

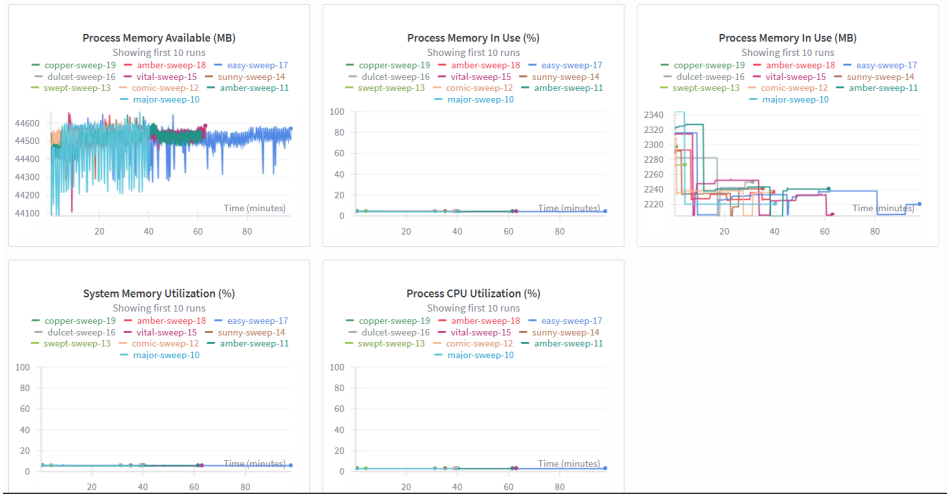

Figure 6: Efficiency metrics

3.3 Comparing results: No Pretraining vs Pretraining

In this section, I evaluate model performance on CIFAR-100 under two conditions:

1. Training from scratch (no pretraining)
2. Using pretrained models

Results: No Pretraining

Even though I didn't achieve the scores mentioned in the requirements, I will present one of my best configuration on the non-pretrained models, with 72.34% accuracy on test data.

- **Model:** ResNest14d (pretrained=False)
- **Batch size:** 128
- **Optimizer:** SAM (weight_decay = 0.0005, lr = 0.1)
- **Scheduler:** StepLR (step_size=25, gamma=0.1)
- **Epochs:** 90
- **Augmentations:** Resize((32, 32)), RandomCrop(32, padding=4),RandomHorizontalFlip(p=0.5), Normalize(mean, std) (with mean and std manually calculated)

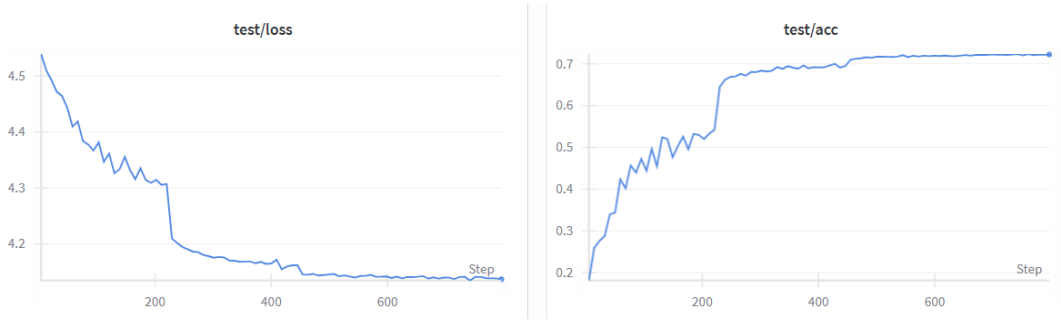

Figure 7: Test accuracy and loss

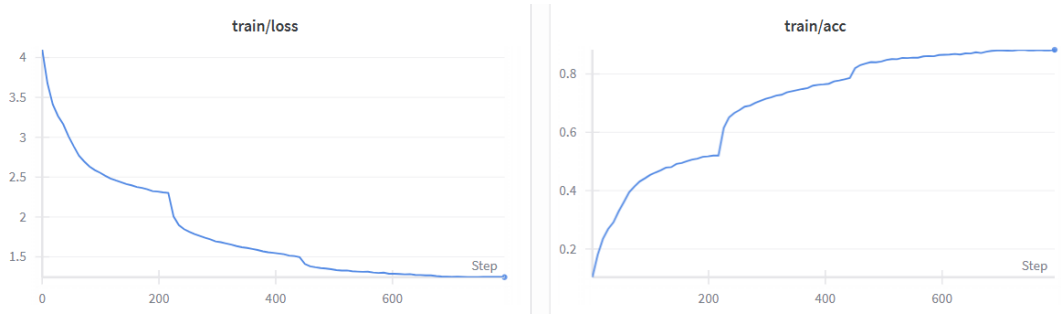

Figure 8: Train accuracy and loss

Results: Using Pretraining

Using pretrained ImageNet weights significantly improves convergence and final accuracy. My best pretrained configuration achieved: 85.42% on test set. This exceeds all required thresholds (82%, 85%).

- **Model:** ResNet50 (pretrained=True)
- **Batch size:** 32

- **Optimizer:** Muon (weight_decay = 0.05, lr = 0.0005)
- **Scheduler:** ReduceLROnPlateau (factor=0.1, mode=min, patience=5)
- **Epochs:** 10
- **Augmentations:** Resize((224, 224)), RandomCrop(224, padding=4),RandomHorizontalFlip(p=0.5), Normalize(mean, std) (with mean and std according to the pretrained model)

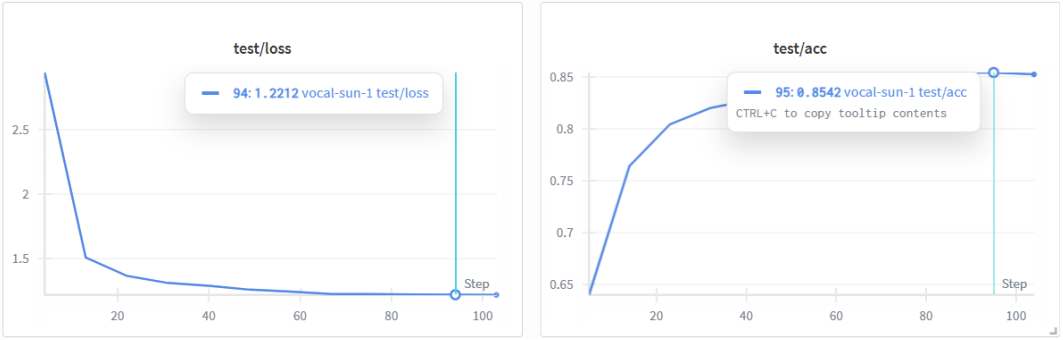

Figure 9: Loss and accuracy for test data

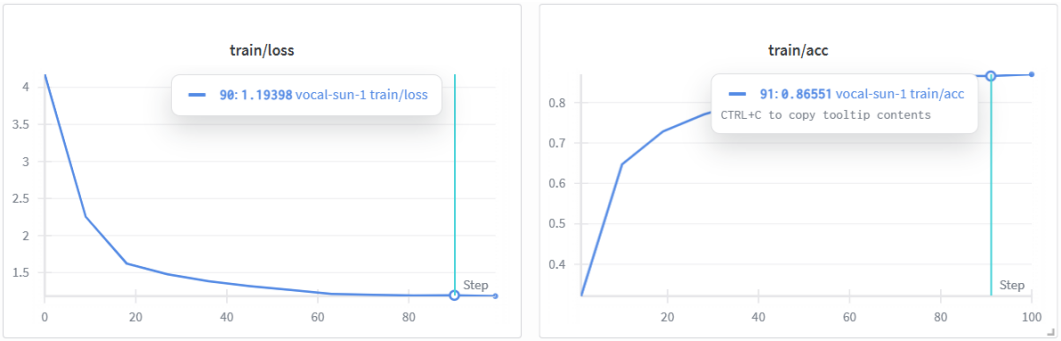

Figure 10: Loss and accuracy for train data

4 Estimated score

- For the first criteria (functionality and requirements for the pipeline), considered that all pipeline requirements are satisfied, I estimate full points.
- Hyperparameter sweep and experimental results: I conducted a sweep with eight configurations that achieve over 70% accuracy on CIFAR-100, and I reported the configurations that contribute to the best results. I also included the metrics from Weights & Biases. Therefore, I estimate full points for this criterion.
- Pipeline efficiency: The details related to this aspect are presented in Section 3.2. Based on these results, I estimate a score of 2.5 points.
- No pretraining: The best accuracy obtained without pretraining was 74% on CIFAR-100, corresponding to 0/3 points.
- With pretraining: The best accuracy obtained with pretraining was 85%, corresponding to 3/3 points.