# Configurable Deep Learning Pipeline for CIFAR Datasets with Pytorch

## Training pipeline is here. Code overview:

The pipeline supports **device agnosticism** by automatically selecting between GPU and CPU, depending on availability, with the get_default_device() function. All data and models are moved to the selected device.

The pipeline is designed to support CIFAR10, CIFAR100, and MNIST. The **dataset configuration** is handled through command-line arguments, allowing easy switching between datasets without altering the core code. Additionally, there's flexibility to add custom datasets using ImageFolder (for new datasets that follow the folder structure of labeled images). The function get_dataset() manages this process, taking as parameters the dataset name, path, split (train, val, or test), and augmentation type. Each dataset has unique preprocessing needs, and the code adapts to those using targeted transformation pipelines. For instance:

- **CIFAR10 and CIFAR100**: These datasets use similar transformation pipelines with normalization based on their respective mean and standard deviation values. Augmentation options such as random cropping, horizontal flipping, and color jittering are applied conditionally based on the chosen augmentation scheme (explained below).
- **MNIST**: Being a grayscale dataset, MNIST transformations are much simpler. Augmentation options are also more limited here, with rotation, affine transformation, and blurring applied in specific cases.

Caching is implemented using the CachedDataset class, which stores dataset elements as a list when caching is enabled. During each call to __getitem__, it retrieves data directly from this cached list if caching is enabled. If caching is disabled, it loads data from the disk on-demand. To ensure compatibility with random augmentations, the caching process stores the original data and applies augmentations dynamically at runtime

DataLoaders for both training and validation are configurable in terms of batch size, pinning memory, and shuffling options. These are managed through command-line arguments and configurations.

For CIFAR datasets, it supports resnet18 (from HuggingFace) and PreActResNet18. For MNIST, it supports the classic MLP and LeNet models. This flexibility is achieved through the get_model() function, which initializes the model based on the dataset and specified architecture, and is controlled through command-line parameters.

The code allows users to select from optimizers: SGD, SGD with momentum, SGD with Nesterov, SGD with weight decay, Adam, AdamW, and RMSprop. They are initialized in get_optimizer() function, which takes the optimizer type, learning rate, and additional options from the command-line.

Two learning rate schedulers are implemented: StepLR and ReduceLROnPlateau. The get_scheduler() function configures these based on the provided command-line argument, enabling automated learning rate adjustments during training for better convergence. The StepLR scheduler reduces the learning rate at fixed intervals, while ReduceLROnPlateau reacts dynamically to improvements in validation loss.

Early stopping is implemented to halt training if validation accuracy plateaus, avoiding overfitting and unnecessary computation. This feature is configurable with patience and delta parameters, allowing fine control over training duration based on validation performance.

The pipeline supports cutmix, mixup, advanced, and default augmentations and are applied based on the provided arguments. The augmentations are handled within the get_dataset() function and during training in the train() function. Cutmix and mixup are directly applied during the forward pass, while advanced augmentations include techniques like color jitter, Gaussian blur, and random erasing.

Metrics and results are logged using both Weights & Biases - the code logs metrics such as training and validation accuracy, loss, and learning rate.

The parameters that I varied for configurations on CIFAR100:

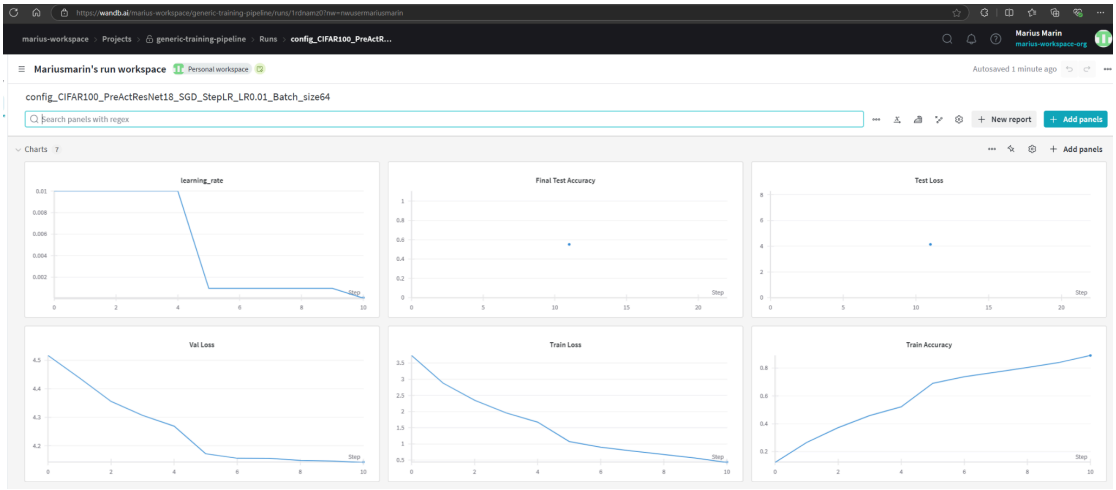
```
sweep_config = {
    "method": "random",
    "metric": {"name": "Val Accuracy", "goal": "maximize"},
    "parameters": {
        "dataset": {"value": "CIFAR100"},
        "model": {"values": ["resnet18", "PreActResNet18"]},
        "augmentations": {"values": ["cutmix", "mixup"]},
        "optimizer": {"values": ["SGD", "AdamW"]},
        "lr": {"values": [0.001, 0.01]},
        "weight_decay": {"values": [0.0001, 0.0005]},
        "batch_size": {"values": [32, 64]},
        "scheduler": {"values": ["StepLR", "ReduceLROnPlateau", "None"]},
        "epochs": {"value": 20},
        "data_path": {"value": "./data"}
    }
}
```

Best results:

| Run ID | Augmentation | Model | Optimizer | Scheduler | Test Accuracy | Test Loss | Batch Size | Learning Rate | Weight Decay |
|---|---|---|---|---|---|---|---|---|---|
| sweet-sweep-19 | mixup | resnet18 | SGD | ReduceLROnPlateau | 0.7116 | 4.1264 | 64 | 0.001 | 0.0005 |
| rare-sweep-2 | mixup | resnet18 | SGD | StepLR | 0.7078 | 4.1312 | 64 | 0.001 | 0.0001 |
| confused-sweep-17 | mixup | resnet18 | SGD | StepLR | 0.7031 | 4.1021 | 64 | 0.001 | 0.0001 |
| playful-sweep-10 | mixup | resnet18 | SGD | ReduceLROnPlateau | 0.6919 | 4.1398 | 64 | 0.001 | 0.0005 |
| celestial-sweep-6 | mixup | resnet18 | SGD | None | 0.6916 | 4.1352 | 64 | 0.001 | 0.0005 |
| stellar-sweep-5 | cutmix | resnet18 | AdamW | StepLR | 0.6217 | 4.2009 | 64 | 0.001 | 0.0005 |
| lilac-sweep-14 | mixup | PreActResNet18 | AdamW | StepLR | 0.5929 | 4.1936 | 64 | 0.01 | 0.0001 |
| peachy-sweep-9 | cutmix | resnet18 | AdamW | StepLR | 0.5927 | 4.2145 | 64 | 0.001 | 0.0005 |
| effortless-sweep-21 | cutmix | PreActResNet18 | AdamW | ReduceLROnPlateau | 0.5653 | 4.225 | 32 | 0.001 | 0.0001 |
| fresh-sweep-4 | mixup | resnet18 | AdamW | ReduceLROnPlateau | 0.5452 | 4.239 | 32 | 0.001 | 0.0001 |

Wandb metrics: generic-training-pipeline Workspace – Weights & Biases


config_CIFAR100_PreActResNet18_SGD_StepLR_LR0.01_Batch_size64 | generic-training-pipeline – Weights & Biases




Efficiency considerations:

Implemented gradient accumulation, which effectively simulates a larger batch size by accumulating gradients over several smaller batches before updating model weights. This allows the model to process smaller batches sequentially and combine their gradients, effectively performing as though we had a larger batch size. The accumulation_steps parameter allows control over how many batches are combined in this way.

Used torch.amp.GradScaler to scale gradients and enable mixed-precision training. This allows compatible GPUs to process certain computations in float16 precision, reducing memory load and speeding up training by leveraging more efficient computations. Mixed precision is applied selectively during the forward pass and gradient calculations, minimizing numerical precision loss while maximizing efficiency.

By setting pin_memory=True in our DataLoader, I enabled faster data transfer from host (CPU) to device (GPU) memory. When data loading is pinned, the system can access it more quickly, reducing latency during batch transfers. This option, combined with num_workers, which allows

for parallel data loading, maximizes the throughput of data into the model, keeping the GPU fed with data efficiently.
The early stopping mechanism and configurable learning rate schedulers also help for efficiency.

I expect around 17 points.