# Efficient Deep Learning with Accelerate: A Quick Overview

### DataLoaderConfiguration

| Argument | Description | Default |
| --- | --- | --- |
| **split\_batches** | Whether to split batches across devices. | False |
| **dispatch\_batches** | Whether to iterate dataloader on main process only. | True for IterableDataset, False otherwise. |
| **even\_batches** | Whether to duplicate samples to ensure equal batch division. | True |
| **use\_seedable\_sampler** | Whether to use a fully seedable random sampler. | False |
| **data\_seed** | Seed to use for the generator when use\_seedable\_sampler is True. | None |
| **non\_blocking** | Whether to use non-blocking host-to-device transfers. | False |
| **use\_stateful\_dataloader** | Whether to use a torchdata.StatefulDataLoader. | False |

Detailed explanations for each argument are provided below:

* **split\_batches** : This argument determines how batches are distributed across devices. If set to True, batches yielded by the dataloaders are split across devices, resulting in the same batch size being used on all processes. However, the batch size must be a multiple of the number of processes being used. If set to False, the actual batch size used is the one specified in the script multiplied by the number of processes.
* **dispatch\_batches** : When set to True, the dataloader is iterated only on the main process, and the resulting batches are split and broadcast to each process. This is the default behavior for DataLoader instances based on IterableDataset. For other types of datasets, the default is False.
* **even\_batches** : If the total batch size across all processes doesn’t divide the dataset evenly, setting this argument to True will duplicate samples at the start of the dataset to ensure that the batch can be divided equally among all workers. This ensures consistent batch sizes across all processes, which can be beneficial for certain training scenarios. The default value is True.
* **use\_seedable\_sampler** : Enabling this option ensures that training results are fully reproducible by employing a fully seedable random sampler (data\_loader.SeedableRandomSampler). This technique ensures that the data is shuffled in the same order across different runs, leading to consistent results. While seed-to-seed results may vary slightly, the overall differences are generally negligible when comparing results across multiple seeds. For optimal reproducibility, it’s recommended to use use\_seedable\_sampler in conjunction with ~utils.set\_seed.
* **data\_seed** : When use\_seedable\_sampler is set to True, this argument specifies the seed to be used for the underlying generator. If set to None, the generator will use the current default seed from PyTorch. This provides control over the random shuffling of the dataset when using a seedable sampler.
* **non\_blocking** : By setting this argument to True, the prepared dataloader will utilize non-blocking host-to-device transfers. This allows for improved overlap between dataloader communication and computation, potentially reducing training time. For this to work effectively, it’s recommended to set pin\_memory to True in the dataloader.
* **use\_stateful\_dataloader** : This argument, if set to True, configures the dataloader to be backed by a torchdata.StatefulDataLoader. This feature requires the installation of torchdata version 0.8.0 or higher. StatefulDataLoader offers advantages in certain scenarios, such as handling stateful datasets or improving performance.

### Accelerator

| Argument | Description | Default |
| --- | --- | --- |
| **device\_placement** | Determines whether the accelerator should automatically place objects on the correct device. | True |
| **mixed\_precision** | Specifies the mixed precision mode to use. | Value of the ACCELERATE\_MIXED\_PRECISION environment variable, or the default in the accelerate config. |
| **gradient\_accumulation\_steps** | Number of steps to accumulate gradients before updating model weights. | 1 |
| **cpu** | Forces the script to run on the CPU, ignoring available GPUs. | False |
| **dataloader\_config** | Configuration for handling dataloaders in distributed settings. See the previous response for more details on this argument. | DataLoaderConfiguration() |
| **deepspeed\_plugin** | Allows customization of DeepSpeed-related arguments. Can be a DeepSpeedPlugin object or a dictionary of DeepSpeedPlugin objects. | DeepSpeedPlugin() if the environment variable ACCELERATE\_USE\_DEEPSPEED is “true”, None otherwise. |
| **fsdp\_plugin** | Allows customization of Fully Sharded Data Parallel (FSDP) arguments. This should be a FullyShardedDataParallelPlugin object. | None |
| **megatron\_lm\_plugin** | Allows customization of Megatron-LM arguments. This should be a MegatronLMPlugin object. | None |
| **rng\_types** | Specifies the random number generators to synchronize at each iteration. | ["generator"] |
| **log\_with** | List of loggers to set up for experiment tracking. | None |
| **project\_dir** | Path to a directory to store logs and checkpoints. | None |
| **project\_config** | Configuration for managing the saving of the training state. | None |
| **gradient\_accumulation\_plugin** | Allows fine-grained control over gradient accumulation. | None |
| **step\_scheduler\_with\_optimizer** | Determines if the learning rate scheduler is updated at the same time as the optimizer. | True |
| **kwargs\_handlers** | Allows passing in keyword arguments specific to the backend. This should be a list of KwargsHandler objects. | None |
| **dynamo\_backend** | Specifies the backend to use for TorchDynamo. | None |
| **dynamo\_plugin** | Allows customization of TorchDynamo related arguments. | TorchDynamoPlugin() |

These arguments offer significant control over the behavior of the accelerator, allowing you to configure it for different hardware, distributed training strategies, mixed-precision training, and other settings. For example:

* **device\_placement:** This argument controls whether the accelerator will place the model, optimizer, and dataloader on the appropriate device. If you have multiple GPUs, this argument will place these objects on the correct GPU.
* **mixed\_precision**: This argument lets you enable mixed precision training. This can speed up training, but may require careful tuning of the learning rate and other hyperparameters.
* **gradient\_accumulation\_steps**: This argument lets you specify how many batches to process before updating the model weights. This can be useful when training on very large batches, or when using a limited amount of GPU memory.

For more details on any specific argument, please refer to the documentation for the [Accelerator](https://huggingface.co/docs/accelerate/en/package_reference/accelerator) class in the Hugging Face Accelerate library.

import torch  
from torch.utils.data import DataLoader  
from torchvision.datasets import MNIST  
from torchvision.transforms import ToTensor  
from accelerate import Accelerator, DataLoaderConfiguration  
  
# Define the model  
class SimpleModel(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.linear = torch.nn.Linear(28\*28, 10)  
 def forward(self, x):  
 return self.linear(x.view(x.size(0), -1))  
model = SimpleModel()  
  
# Define the optimizer  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  
  
# Define the learning rate scheduler  
scheduler = torch.optim.lr\_scheduler.LambdaLR(optimizer, lr\_lambda=lambda epoch: 0.95 \*\* epoch)  
  
# Create the dataloaders  
train\_dataset = MNIST(root='./data', train=True, download=True, transform=ToTensor())  
test\_dataset = MNIST(root='./data', train=False, download=True, transform=ToTensor())  
train\_dataloader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)  
test\_dataloader = DataLoader(test\_dataset, batch\_size=64, shuffle=False)  
  
  
# Initialize the Accelerator   
accelerator = Accelerator(  
 gradient\_accumulation\_steps=2, # accumulate gradients for 2 steps  
 mixed\_precision="fp16", # Enable fp16 mixed precision   
 dataloader\_config=DataLoaderConfiguration(split\_batches=True), # Split batches across devices  
   
)  
  
# Prepare the model, optimizer, and dataloaders  
model, optimizer, train\_dataloader, test\_dataloader, scheduler = accelerator.prepare(  
 model, optimizer, train\_dataloader, test\_dataloader, scheduler  
)  
  
# Training loop  
for epoch in range(5):  
 for batch\_idx, (inputs, targets) in enumerate(train\_dataloader):  
 # Forward pass  
 outputs = model(inputs)  
 loss = torch.nn.functional.cross\_entropy(outputs, targets)  
   
 # Backward pass  
 accelerator.backward(loss)  
  
 # Update weights   
 if accelerator.sync\_gradients:  
 optimizer.step()  
 scheduler.step()  
 optimizer.zero\_grad()  
  
# Evaluate the model  
with torch.no\_grad():  
 for batch\_idx, (inputs, targets) in enumerate(test\_dataloader):  
 outputs = model(inputs)  
 # Gather results across processes  
 all\_outputs = accelerator.gather(outputs)  
 all\_targets = accelerator.gather(targets)

**Explanation**

* **Import Necessary Libraries**: The code starts by importing essential modules from torch, torchvision, and accelerate.
* **Model, Optimizer, Scheduler, Dataloaders**: It defines a basic neural network model (SimpleModel), an optimizer (torch.optim.Adam), a learning rate scheduler (torch.optim.lr\_scheduler.LambdaLR), and creates dataloaders for training and testing using the MNIST dataset.
* **Accelerator Initialization**: An Accelerator instance is created, enabling features like gradient accumulation (gradient\_accumulation\_steps), mixed precision training (mixed\_precision), and batch splitting (dataloader\_config).
* **Preparation with accelerator.prepare()**: The prepare method readies the model, optimizer, dataloaders, and scheduler for distributed training and mixed precision.
* **Training Loop**: A standard training loop iterates through epochs and batches, performing the forward pass, calculating the loss, and using accelerator.backward() for backpropagation. The optimizer steps and scheduler adjustments are done when gradients are synchronized (accelerator.sync\_gradients).
* **Evaluation**: The model is evaluated on the test set, and accelerator.gather() is used to collect predictions and targets from different processes.

**Key Features of Accelerator used:**

* **prepare()**: This function is central to the Accelerator functionality, as it prepares the model, optimizer, dataloaders, and learning rate scheduler for distributed training and mixed precision. It handles device placement, wraps objects for distributed operations, and enables the specified mixed precision mode.
* **backward()**: This method replaces the standard loss.backward() call and ensures proper gradient scaling in distributed and mixed precision settings. It’s crucial for correct backpropagation, especially when using techniques like gradient accumulation.
* **sync\_gradients**: This attribute indicates whether gradients are currently synchronized across processes. It’s used to control when the optimizer updates the model weights, ensuring proper gradient accumulation and synchronization.
* **gather()**: This function collects tensors from all processes and concatenates them along the first dimension, enabling aggregation of results, particularly in evaluation or when gathering all predictions or labels.

The accelerator.prepare() method sets up the model and other objects for distributed training, and the other features handle tasks like gradient calculation, synchronization, and gathering of distributed data.