Integrate Weights & Biases with PyTorch

Archive note: all links and images have been disabled as their destinations and/or source locations no longer exist. The main navigation is also missing.

Rewritten and edited by Brian Immel, last modified on Dec 28, 2025

- Before you begin
- Prerequisites
- Step 1: Install and Authenticate
- Step 2: Define Model and Data
- Step 3: Initialize the Run and Hyperparameters
- Step 4: Track Metrics and Gradients
- Step 5: Version and Save the Model
- Best Practices for Reproducibility
- Advanced Configuration
    - Hyperparameter Sweeps
    - Environment Settings
- Next steps

This guide demonstrates how to integrate Weights & Biases (W&B) into your PyTorch pipeline. W&B helps you track machine learning experiments, visualize model performance, and ensure reproducibility across teams.

Before you begin

This guide is intended for machine learning engineers and data scientists who are familiar with PyTorch and Python. Using W&B allows you to move beyond manual logging in spreadsheets to an automated, central dashboard for all your model metadata.

By the end of this guide, you will know how to:

- Initialize a W&B run and configure hyperparameters.
- Define and "watch" a PyTorch model for gradient tracking.
- Track real-time metrics and save model artifacts for reproducibility.

Prerequisites

- A W&B account.
- A Python environment with `torch` and `torchvision` installed.

Step 1: Install and Authenticate

Install the `wandb` and `onnx` libraries and log in to your account.

```
# Install dependencies
pip install wandb onnx -Uq

import wandb
wandb.login()
```

Step 2: Define Model and Data

The following boilerplate defines a standard Convolutional Neural Network (CNN) and data loaders for the MNIST dataset.

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

# Select device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class ConvNet(nn.Module):
    def __init__(self, kernels, classes):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, kernels[0], kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(kernels[0], kernels[1], kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(7 * 7 * kernels[1], classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        return out

def make_loader(config, train=True):
    full_dataset = torchvision.datasets.MNIST(root=".", train=train, transform=transform
s.ToTensor(), download=True)
    return torch.utils.data.DataLoader(dataset=full_dataset, batch_size=config.batch_siz
e, shuffle=True, pin_memory=True, num_workers=2)
```

Step 3: Initialize the Run and Hyperparameters

Initialize a W&B run to track your experiment. Use the `config` dictionary to capture hyperparameters, which allows you to filter and query runs in the W&B dashboard.

```
# Define experiment metadata and hyperparameters
config = {
    "epochs": 5,
    "classes": 10,
    "kernels": [16, 32],
    "batch_size": 128,
    "learning_rate": 0.005,
    "dataset": "MNIST",
    "architecture": "CNN"
}

def model_pipeline(hyperparameters):
    # Initialize a new W&B run.
    with wandb.init(project="<PROJECT_NAME>", config=hyperparameters) as run:
        config = run.config

        # Build model, data, and optimizer
        train_loader = make_loader(config, train=True)
        test_loader = make_loader(config, train=False)
        model = ConvNet(config.kernels, config.classes).to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=config.learning_rate)

        # Train and track performance
        train(model, train_loader, criterion, optimizer, config)

        # Evaluate final performance
        test(model, test_loader)

    return model
```

Step 4: Track Metrics and Gradients

Use `wandb.watch` to log model gradients and `wandb.log` to capture training metrics such as loss and accuracy.

```python
def train(model, loader, criterion, optimizer, config):
    # Log gradients and topology
    wandb.watch(model, criterion, log="all", log_freq=10)

    example_ct = 0
    for epoch in range(config.epochs):
        for images, labels in loader:
            loss = train_batch(images, labels, model, optimizer, criterion)
            example_ct += len(images)
            # Log metrics to the dashboard every 25 batches
            wandb.log({"epoch": epoch, "loss": loss}, step=example_ct)

def train_batch(images, labels, model, optimizer, criterion):
    images, labels = images.to(device), labels.to(device)
    outputs = model(images)
    loss = criterion(outputs, labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss
```

Step 5: Version and Save the Model

Export your model to the ONNX format and use `wandb.save` to upload the file. This associates the model artifact directly with the training run.

```python
def test(model, test_loader):
    model.eval()
    with torch.no_grad():
        correct, total = 0, 0
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = correct / total
        wandb.log({"test_accuracy": accuracy})

    # Save the model
    torch.onnx.export(model, images, "model.onnx")
    wandb.save("model.onnx")
```

Best Practices for Reproducibility

To ensure deterministic behavior across your experiments, it is essential to set random seeds for all libraries involved in the computation.

> **Note:** While setting seeds improves reproducibility, some GPU operations remain non-deterministic. Results may still vary slightly across different hardware configurations. See the PyTorch randomness guide (https://pytorch.org/docs/stable/notes/randomness.html) for details.

```
import random
import numpy as np
import torch

# Set random seeds
random.seed(hash("setting random seeds") % 2**32 - 1)
np.random.seed(hash("improves reproducibility") % 2**32 - 1)
torch.manual_seed(hash("by removing variation") % 2**32 - 1)
torch.cuda.manual_seed_all(hash("across runs") % 2**32 - 1)
```

Advanced Configuration

Hyperparameter Sweeps

Automate model tuning by defining a search strategy and running an agent.

```
# Initialize and run a sweep
sweep_id = wandb.sweep(sweep_config)
wandb.agent(sweep_id, function=train)
```

Environment Settings

- **Offline Mode**: Set `WANDB_MODE=dryrun` to train without an internet connection. Use `wandb sync` to upload your data later.
- **Headless Authentication**: Use the `WANDB_API_KEY` environment variable to authenticate on automated managed clusters.

Next steps

- Learn how to use W&B Artifacts for dataset versioning.
- Explore more W&B PyTorch Examples on GitHub.