1. Custom Training Loop

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset

# A simple custom dataset
class TensorDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

# Training loop
def train(model, dataset, epochs=10, lr=1e-3, batch_size=32):
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        total_loss = 0.0
        num_batches = 0

        model.train()  # set to training mode
        for batch_X, batch_y in dataloader:
            # Forward pass
            predictions = model(batch_X)
            loss = loss_fn(predictions, batch_y)

            # Backward pass
            optimizer.zero_grad()  # clear old gradients
            loss.backward()        # compute new gradients
            optimizer.step()       # update weights

            total_loss += loss.item()
            num_batches += 1

        avg_loss = total_loss / num_batches
        print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")

class SimpleNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()  # MUST call this
        self.layer1 = nn.Linear(input_dim, hidden_dim)
        self.activation = nn.ReLU()
        self.layer2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.layer1(x)
        x = self.activation(x)
        x = self.layer2(x)
        return x

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

model = SimpleNet(784, 256, 10).to(device)

# Data must be on the same device as the model
for batch_X, batch_y in dataloader:
    batch_X = batch_X.to(device)
    batch_y = batch_y.to(device)
    output = model(batch_X)
```

2. Saving and Loading (Checkpointing)

```
# Save
def save_checkpoint(model, optimizer, epoch, loss, path):
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss,
    }, path)

# Load
def load_checkpoint(path, model, optimizer=None):
    checkpoint = torch.load(path)
    model.load_state_dict(checkpoint['model_state_dict'])
    if optimizer:
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    return checkpoint['epoch'], checkpoint['loss']
```

3. PyTorch DistributedDataParallel (DDP)

```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data.distributed import DistributedSampler

def setup(rank, world_size):
    """Initialize the distributed environment."""
    dist.init_process_group(
        backend="nccl",      # use nccl for GPU, gloo for CPU
        rank=rank,            # this process's ID (0, 1, 2, ...)
        world_size=world_size # total number of processes
    )

def cleanup():
    dist.destroy_process_group()

def train_distributed(rank, world_size, dataset):
    setup(rank, world_size)

    # Each GPU gets a different subset of data
    sampler = DistributedSampler(dataset, num_replicas=world_size, rank=rank)
    dataloader = DataLoader(dataset, batch_size=32, sampler=sampler)

    # Wrap model in DDP
    model = SimpleNet(784, 256, 10).to(rank)
    model = DDP(model, device_ids=[rank])

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    loss_fn = nn.CrossEntropyLoss()

    for epoch in range(10):
        sampler.set_epoch(epoch)  # IMPORTANT: shuffles differently each epoch

        for batch_X, batch_y in dataloader:
            batch_X = batch_X.to(rank)
            batch_y = batch_y.to(rank)

            output = model(batch_X)
            loss = loss_fn(output, batch_y)

            optimizer.zero_grad()
            loss.backward()   # DDP handles gradient sync automatically
            optimizer.step()

    cleanup()
```

4. Training Engine

```
import torch
import torch.nn as nn
import os
import time

class TrainingEngine:
    def __init__(
        self,
        model,
        optimizer,
        loss_fn,
        device="cpu",
        checkpoint_dir="./checkpoints",
        checkpoint_every=5,
        max_grad_norm=1.0,
        patience=10,
    ):
        self.model = model.to(device)
        self.optimizer = optimizer
        self.loss_fn = loss_fn
        self.device = device
        self.checkpoint_dir = checkpoint_dir
        self.checkpoint_every = checkpoint_every
        self.max_grad_norm = max_grad_norm
        self.patience = patience

        self.current_epoch = 0
        self.best_val_loss = float('inf')
        self.epochs_without_improvement = 0
        self.history = []

        os.makedirs(checkpoint_dir, exist_ok=True)

    def _train_one_epoch(self, dataloader):
        self.model.train()
        total_loss = 0.0
        num_batches = 0

        for X, y in dataloader:
            X, y = X.to(self.device), y.to(self.device)

            preds = self.model(X)
            loss = self.loss_fn(preds, y)

            self.optimizer.zero_grad()
            loss.backward()

            # Gradient clipping - prevents exploding gradients
            if self.max_grad_norm:
                nn.utils.clip_grad_norm_(
                    self.model.parameters(), self.max_grad_norm
                )

            self.optimizer.step()

            total_loss += loss.item()
            num_batches += 1

        return total_loss / num_batches

    def _validate(self, dataloader):
        self.model.eval()
        total_loss = 0.0
        num_batches = 0
        with torch.no_grad():  # no gradients needed for validation
```

```
        for X, y in dataloader:
            X, y = X.to(self.device), y.to(self.device)
            preds = self.model(X)
            loss = self.loss_fn(preds, y)
            total_loss += loss.item()
            num_batches += 1

    return total_loss / num_batches

def save_checkpoint(self):
    path = os.path.join(
        self.checkpoint_dir, f"checkpoint_epoch_{self.current_epoch}.pt"
    )
    torch.save({
        'epoch': self.current_epoch,
        'model_state_dict': self.model.state_dict(),
        'optimizer_state_dict': self.optimizer.state_dict(),
        'best_val_loss': self.best_val_loss,
        'history': self.history,
    }, path)
    return path

def load_checkpoint(self, path):
    checkpoint = torch.load(path, map_location=self.device)
    self.model.load_state_dict(checkpoint['model_state_dict'])
    self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    self.current_epoch = checkpoint['epoch']
    self.best_val_loss = checkpoint['best_val_loss']
    self.history = checkpoint['history']

def train(self, train_loader, val_loader, epochs):
    start_epoch = self.current_epoch

    for epoch in range(start_epoch, start_epoch + epochs):
        self.current_epoch = epoch + 1
        epoch_start = time.time()

        train_loss = self._train_one_epoch(train_loader)
        val_loss = self._validate(val_loader)

        elapsed = time.time() - epoch_start

        record = {
            'epoch': self.current_epoch,
            'train_loss': train_loss,
            'val_loss': val_loss,
            'time': elapsed,
        }
        self.history.append(record)

        # Check for improvement
        if val_loss < self.best_val_loss:
            self.best_val_loss = val_loss
            self.epochs_without_improvement = 0
            self.save_checkpoint()  # always save best
        else:
            self.epochs_without_improvement += 1

        # Periodic checkpoint
        if self.current_epoch % self.checkpoint_every == 0:
            self.save_checkpoint()

        # Early stopping
        if self.epochs_without_improvement >= self.patience:
            print(f"Early stopping at epoch {self.current_epoch}")
            break

    return self.history
```

Usage:

```
model = SimpleNet(784, 256, 10)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

engine = TrainingEngine(
    model=model,
    optimizer=optimizer,
    loss_fn=loss_fn,
    device="cuda",
    checkpoint_every=5,
    patience=10,
    max_grad_norm=1.0,
)

# Train
history = engine.train(train_loader, val_loader, epochs=100)

# Resume later
engine.load_checkpoint("checkpoints/checkpoint_epoch_50.pt")
engine.train(train_loader, val_loader, epochs=50)  # continues from 50
```