```python
import sys
import os
from pathlib import Path
import torch
import torchvision
from torch import nn
from torchvision import transforms, datasets
from torch.utils.data import DataLoader, Dataset
from torchinfo import summary
from timeit import default_timer as timer
from torch.utils.tensorboard import SummaryWriter

# Set device
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

# Define Synthetic Dataset class
class SyntheticDataset(Dataset):
    """A synthetic dataset for quick testing."""
    def __init__(
        self,
        num_samples=1000,
        img_size=224,
        num_classes=1000
    ):
        self.num_samples = num_samples
        self.img_size = img_size
        self.num_classes = num_classes
        self.data = torch.randn(
            num_samples,
            3,
            img_size,
            img_size
        )
        self.labels = torch.randint(
            0,
            num_classes,
            (num_samples,)
        )

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]

# Define Vision Transformer components
class PatchEmbedding(nn.Module):
    def __init__(
        self,
        in_channels: int = 3,
        patch_size: int = 16,
        embedding_dim: int = 768
    ):
        super().__init__()
        self.patcher = nn.Conv2d(
            in_channels=in_channels,
            out_channels=embedding_dim,
            kernel_size=patch_size,
            stride=patch_size
        )
        self.flatten = nn.Flatten(start_dim=2)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.patcher(x)
        x = self.flatten(x)
        return x.permute(0, 2, 1)  # [batch_size, num_patches, embedding_dim]

class MultiheadSelfAttention(nn.Module):
    def __init__(
        self,
        embedding_dim: int = 768,
        num_heads: int = 12,
        attn_dropout: float = 0.0
    ):
```

```
        super().__init__()
        self.layer_norm = nn.LayerNorm(
            normalized_shape=embedding_dim
        )
        self.multihead_attn = nn.MultiheadAttention(
            embed_dim=embedding_dim,
            num_heads=num_heads,
            dropout=attn_dropout,
            batch_first=True
        )

    def forward(self, x):
        x = self.layer_norm(x)
        attn_output, _ = self.multihead_attn(
            query=x,
            key=x,
            value=x,
            need_weights=False
        )
        return attn_output

class MLPBlock(nn.Module):
    def __init__(
        self,
        embedding_dim: int = 768,
        mlp_size: int = 3072,
        dropout: float = 0.1
    ):
        super().__init__()
        self.layer_norm = nn.LayerNorm(
            normalized_shape=embedding_dim
        )
        self.mlp = nn.Sequential(
            nn.Linear(
                in_features=embedding_dim,
                out_features=mlp_size
            ),
            nn.GELU(),
            nn.Dropout(p=dropout),
            nn.Linear(
                in_features=mlp_size,
                out_features=embedding_dim
            ),
            nn.Dropout(p=dropout)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.layer_norm(x)
        return self.mlp(x)

class TransformerEncoderBlock(nn.Module):
    def __init__(
        self,
        embedding_dim: int = 768,
        num_heads: int = 12,
        mlp_size: int = 3072,
        attn_dropout: float = 0.0,
        mlp_dropout: float = 0.1
    ):
        super().__init__()
        self.msa_block = MultiheadSelfAttention(
            embedding_dim=embedding_dim,
            num_heads=num_heads,
            attn_dropout=attn_dropout
        )
        self.mlp_block = MLPBlock(
            embedding_dim=embedding_dim,
            mlp_size=mlp_size,
            dropout=mlp_dropout
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.msa_block(x) + x
        x = self.mlp_block(x) + x
        return x
```

```
class VisionTransformer(nn.Module):
    def __init__(
        self,
        image_size: int = 224,
        patch_size: int = 16,
        num_transformer_layers: int = 12,
        embedding_dim: int = 768,
        num_heads: int = 12,
        mlp_size: int = 3072,
        num_classes: int = 1000
    ):
        super().__init__()
        assert image_size % patch_size == 0, "Image size must be divisible by patch size."
        self.num_patches = (image_size * image_size) // (patch_size * patch_size)
        self.class_embedding = nn.Parameter(
            torch.randn(1, 1, embedding_dim)
        )
        self.position_embeddings = nn.Parameter(
            torch.randn(1, self.num_patches + 1, embedding_dim)
        )
        self.embedding_dropout = nn.Dropout(0.1)
        self.patch_embedding = PatchEmbedding(
            3,
            patch_size,
            embedding_dim
        )
        self.transformer_encoder = nn.Sequential(
            *[
                TransformerEncoderBlock(
                    embedding_dim,
                    num_heads,
                    mlp_size
                ) for _ in range(num_transformer_layers)
            ]
        )
        self.classifier = nn.Sequential(
            nn.LayerNorm(
                normalized_shape=embedding_dim
            ),
            nn.Linear(
                in_features=embedding_dim,
                out_features=num_classes
            )
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        batch_size = x.size(0)
        class_token = self.class_embedding.expand(
            batch_size,
            -1,
            -1
        )
        x = self.patch_embedding(x)
        x = torch.cat((class_token, x), dim=1)
        x = x + self.position_embeddings
        x = self.embedding_dropout(x)
        x = self.transformer_encoder(x)
        return self.classifier(x[:, 0])

# Define training function with TensorBoard logging
def train_model(
    model,
    dataloader,
    criterion,
    optimizer,
    epochs=5,
    log_dir="logs/train"
):
    writer = SummaryWriter(log_dir=log_dir)
    model.train()
    for epoch in range(epochs):
        epoch_loss = 0
        correct = 0
        total = 0
```

```
        for batch_idx, (images, labels) in enumerate(dataloader):
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

            # Log batch loss to TensorBoard
            writer.add_scalar("Batch Loss", loss.item(), epoch * len(dataloader) + batch_idx)

        epoch_loss /= len(dataloader)
        accuracy = correct / total * 100
        print(
            f"Epoch {epoch + 1}/{epochs}, Loss: {epoch_loss:.4f}, Accuracy: {accuracy:.2f}%"
        )

        # Log epoch metrics to TensorBoard
        writer.add_scalar("Epoch Loss", epoch_loss, epoch)
        writer.add_scalar("Epoch Accuracy", accuracy, epoch)

    writer.close()

# Define evaluation function with TensorBoard logging
def evaluate_model(model, dataloader, criterion, log_dir="logs/eval"):
    writer = SummaryWriter(log_dir=log_dir)
    model.eval()
    total_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (images, labels) in enumerate(dataloader):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

            # Log batch loss to TensorBoard
            writer.add_scalar("Batch Loss", loss.item(), batch_idx)

    avg_loss = total_loss / len(dataloader)
    accuracy = correct / total * 100
    print(f"Evaluation - Loss: {avg_loss:.4f}, Accuracy: {accuracy:.2f}%")

    # Log evaluation metrics to TensorBoard
    writer.add_scalar("Average Loss", avg_loss)
    writer.add_scalar("Accuracy", accuracy)

    writer.close()
    return avg_loss, accuracy

# Retain pizza dataset functionality for later use
def load_pizza_data(custom_transform=None):
    data_path = Path("vit/data/pizza_steak_sushi/")
    train_dir = data_path / "train"
    test_dir = data_path / "test"

    train_data = datasets.ImageFolder(
        train_dir,
        transform=custom_transform
    )
    test_data = datasets.ImageFolder(
        test_dir,
        transform=custom_transform
    )

    train_dataloader = DataLoader(
```

```
        train_data,
        batch_size=32,
        shuffle=True
    )
    test_dataloader = DataLoader(
        test_data,
        batch_size=32,
        shuffle=False
    )

    return train_dataloader, test_dataloader

def pretrained_vit_model():
    pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT
    pretrained_vit_transforms = pretrained_vit_weights.transforms()
    pretrained_vit_model = torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)
    for param in pretrained_vit_model.parameters():
        param.requires_grad = False
    torch.manual_seed(42)
    torch.cuda.manual_seed(42)
    pretrained_vit_model.heads = nn.Linear(in_features=768, out_features=3).to(device)
    summary(
        model=pretrained_vit_model,
        input_size=(32, 3, 224, 224),
        col_names=["input_size", "output_size", "num_params", "trainable"],
        col_width=20,
        row_settings=["var_names"]
    )
    return pretrained_vit_model, pretrained_vit_transforms

# Test with pizza dataset
if __name__ == "__main__":
    print("Testing with pizza dataset...")

    # Prompt user to choose between manual training and pretrained model fine-tuning
    choice = input("Choose training mode: 'manual' for manual training or 'pretrained' for fine-tuning pretrained model: ").strip().lower()

    if choice == 'manual':
        # Manual training with VisionTransformer
        print("Using manual training with VisionTransformer...")
        summary(
            model=VisionTransformer(),
            input_size=(32, 3, 224, 224),
            col_names=["input_size", "output_size", "num_params", "trainable"],
            col_width=20,
            row_settings=["var_names"]
        )
        train_dataloader, test_dataloader = load_pizza_data()

        vit_model = VisionTransformer(
            image_size=224,
            patch_size=16,
            num_transformer_layers=12,
            embedding_dim=768,
            num_heads=12,
            mlp_size=3072,
            num_classes=3
        ).to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(
            vit_model.parameters(),
            lr=3e-4
        )

        train_model(
            vit_model,
            train_dataloader,
            criterion,
            optimizer,
            epochs=5,
            log_dir="logs/manual_train"
        )

        print("Evaluating the manually trained model...")
```

```
        evaluate_model(vit_model, test_dataloader, criterion, log_dir="logs/manual_eval")

        # Save the manually trained model
        MODEL_PATH = Path("torch_models") / "manual_vit_model.pth"
        MODEL_PATH.parent.mkdir(parents=True, exist_ok=True)
        torch.save(vit_model.state_dict(), MODEL_PATH)
        print(f"Manually trained model saved to: {MODEL_PATH}")

    elif choice == 'pretrained':
        # Fine-tuning pretrained VisionTransformer
        print("Using pretrained VisionTransformer for fine-tuning...")
        vit_model, vit_transforms = pretrained_vit_model()
        train_dataloader, test_dataloader = load_pizza_data(vit_transforms)

        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(
            vit_model.heads.parameters(),  # Only train the classification head
            lr=3e-4
        )

        print("Training the pretrained model...")
        train_model(
            vit_model,
            train_dataloader,
            criterion,
            optimizer,
            epochs=20,
            log_dir="logs/pretrained_train"
        )

        print("Evaluating the fine-tuned pretrained model...")
        evaluate_model(vit_model, test_dataloader, criterion, log_dir="logs/pretrained_eval")

        # Save the fine-tuned model
        MODEL_PATH = Path("torch_models") / "fine_tuned_vit_modelv1.pth"
        MODEL_PATH.parent.mkdir(parents=True, exist_ok=True)
        torch.save(vit_model.state_dict(), MODEL_PATH)
        print(f"Fine-tuned model saved to: {MODEL_PATH}")

    else:
        print("Invalid choice. Please choose 'manual' or 'pretrained'.")
```