```
import torch
import torch . nn as nn
import torch . nn . functional as F
import torchvision
import torchvision . transforms as T
from torch . utils . data import DataLoader , Dataset
```

```
class SimCLRDataset(Dataset):
    def __init__(self, dataset, transform):
        self.dataset = dataset
        self.transform = transform

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

    def __getitem__(self, idx):
        x, _ = self.dataset[idx]        # Ignore labels!
        x1 = self.transform(x)          # First augmentation
        x2 = self.transform(x)          # Second augmentation
        return x1, x2

# Strong augmentations for SimCLR (CIFAR-10)
simclr_transform = T.Compose([
    T.RandomResizedCrop(32, scale=(0.2, 1.0)),
    T.RandomHorizontalFlip(),
    T.RandomApply([T.ColorJitter(0.4,0.4,0.4,0.1)], p=0.8),
    T.RandomGrayscale(p=0.2),
    T.GaussianBlur(3, sigma=(0.1, 2.0)),
    T.ToTensor(),
])
```

```
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1))
        )
        self.fc = nn.Linear(256, 128)   # representation size

    def forward(self, x):
        h = self.conv(x)
        h = h.view(h.size(0), -1)
        h = self.fc(h)
        return F.normalize(h, dim=1)
```

```
class ProjectionHead(nn.Module):
    def __init__(self, in_dim=128, out_dim=128):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(in_dim, in_dim),
            nn.ReLU(),
            nn.Linear(in_dim, out_dim)
        )

    def forward(self, x):
        return F.normalize(self.mlp(x), dim=1)
```

```
def nt_xent_loss(z1, z2, temperature=0.5):
    N = z1.size(0)
    z = torch.cat([z1, z2], dim=0)   # Concatenate: 2N x d

    # Compute similarity matrix
    sim = torch.mm(z, z.t()) / temperature

    # Mask out self-similarities (diagonal)
    mask = torch.eye(2*N, dtype=torch.bool, device=z.device)
    sim = sim.masked_fill(mask, -9e15)
```

```
    # Create positive pair labels
    positives = torch.cat([torch.arange(N, 2*N), torch.arange(0, N)]).to(z.device)
    labels = positives

    # Cross-entropy loss
    loss = F.cross_entropy(sim, labels)
    return loss
```

```
# Load CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True)

# Create SimCLR dataset
simclr_train_dataset = SimCLRDataset(train_dataset, simclr_transform)

# Create DataLoader
train_loader = DataLoader(simclr_train_dataset, batch_size=256, shuffle=True, num_workers=2)
```

```
100%|██████████| 170M/170M [00:02<00:00, 58.3MB/s]
```

```
device = "cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu"
encoder = Encoder().to(device)
projector = ProjectionHead().to(device)
optimizer = torch.optim.Adam(list(encoder.parameters()) + list(projector.parameters()), lr=1e-3)

def train_simclr():
    encoder.train()
    projector.train()
    for epoch in range(10): # Quick demo
        total_loss = 0
        for batch_idx, (x1, x2) in enumerate(train_loader):
            x1, x2 = x1.to(device), x2.to(device)
            # Forward pass
            h1, h2 = encoder(x1), encoder(x2)
            z1, z2 = projector(h1), projector(h2)
            # Compute loss
            loss = nt_xent_loss(z1, z2)
            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch + 1}: Loss = {total_loss / len(train_loader):.4f}")

    # Save the encoder (discard projector)
    torch.save(encoder.state_dict(), "simclr_encoder.pth")

train_simclr()
```

```
Epoch 1: Loss = 5.5854
Epoch 2: Loss = 5.3048
Epoch 3: Loss = 5.2154
Epoch 4: Loss = 5.1548
Epoch 5: Loss = 5.1369
Epoch 6: Loss = 5.1093
Epoch 7: Loss = 5.0900
Epoch 8: Loss = 5.0754
Epoch 9: Loss = 5.0657
Epoch 10: Loss = 5.0472
```

```
# Load pre - trained encoder
encoder = Encoder ()
encoder . load_state_dict ( torch . load ( "simclr_encoder.pth" ) )
class FCNClassifier ( nn . Module ) :
    def __init__ ( self , encoder ) :
        super () . __init__ ()
        self . encoder = encoder
        # Freeze encoder parameters
        for param in self . encoder . parameters () :
            param . requires_grad = False
        self . fc = nn . Linear (128 , 10) # CIFAR -10 has 10 classes
    def forward ( self , x ) :
        h = self . encoder ( x ) # Frozen features
        return self . fc ( h ) # Only train classifier
# Create limited labeled dataset (10% per class )
def create_limited_dataset(dataset, samples_per_class=500):
```

```
    limited_indices = []
    targets = dataset.targets if hasattr(dataset, 'targets') else dataset._targets # Handle different dataset types

    # Get indices for each class
    class_indices = {}
    for i, target in enumerate(targets):
        if target not in class_indices:
            class_indices[target] = []
        class_indices[target].append(i)

    # Select a limited number of samples per class
    for class_idx, indices in class_indices.items():
        num_samples = min(samples_per_class, len(indices))
        limited_indices.extend(torch.randperm(len(indices))[:num_samples].tolist())

    # Create a subset of the original dataset
    limited_dataset = torch.utils.data.Subset(dataset, limited_indices)
    return limited_dataset

# Assuming 'train_dataset' is already defined and loaded
# For example, if using CIFAR10:
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True)

limited_dataset = create_limited_dataset(train_dataset, samples_per_class=500)
classifier = FCNClassifier ( encoder ) . to ( device )
```

```
# Create DataLoader for the limited dataset
# Define transformation for the classifier
classifier_transform = T.Compose([
    T.ToTensor(),
    T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)), # CIFAR-10 normalization
])

# Apply transformation to the limited dataset
class ClassifierDataset(Dataset):
    def __init__(self, dataset, transform=None):
        self.dataset = dataset
        self.transform = transform

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

    def __getitem__(self, idx):
        x, y = self.dataset[idx]
        if self.transform:
            x = self.transform(x)
        return x, y

limited_classifier_dataset = ClassifierDataset(limited_dataset, transform=classifier_transform)
limited_train_loader = DataLoader(limited_classifier_dataset, batch_size=64, shuffle=True, num_workers=0)

# Define loss function and optimizer for the classifier
criterion = nn.CrossEntropyLoss()
optimizer_classifier = torch.optim.Adam(classifier.fc.parameters(), lr=1e-3)

def train_classifier():
    classifier.train()
    for epoch in range(10): # Train for a few epochs
        total_loss = 0
        for batch_idx, (inputs, targets) in enumerate(limited_train_loader):
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass
            outputs = classifier(inputs)
            loss = criterion(outputs, targets)

            # Backward pass
            optimizer_classifier.zero_grad()
            loss.backward()
            optimizer_classifier.step()
            total_loss += loss.item()

        print(f"Classifier Epoch {epoch + 1}: Loss = {total_loss / len(limited_train_loader):.4f}")

train_classifier()
```

```
Classifier Epoch 1: Loss = 1.8435
Classifier Epoch 2: Loss = 1.8317
Classifier Epoch 3: Loss = 1.8211
Classifier Epoch 4: Loss = 1.8081
Classifier Epoch 5: Loss = 1.7950
Classifier Epoch 6: Loss = 1.7879
Classifier Epoch 7: Loss = 1.7808
Classifier Epoch 8: Loss = 1.7750
Classifier Epoch 9: Loss = 1.7703
Classifier Epoch 10: Loss = 1.7678
```

```
# Load CIFAR-10 test dataset
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True)

# Apply the same transformation as the classifier training data
test_classifier_dataset = ClassifierDataset(test_dataset, transform=classifier_transform)

# Create DataLoader for the test set
test_loader = DataLoader(test_classifier_dataset, batch_size=64, shuffle=False, num_workers=0)

def evaluate_classifier():
    classifier.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = classifier(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy of the classifier on the 10000 test images: {accuracy:.2f}%")

evaluate_classifier()
```

```
Test Accuracy of the classifier on the 10000 test images: 34.62%
```