DeiT-S Transfer Learning & Fine-Tuning (PyTorch + `timm`)

1 Setup

```
pip install torch torchvision timm==1.*
```

Folder layout (ImageFolder):

```
data/
  train/
    class_a/ img001.jpg ...
    class_b/ ...
  val/
    class_a/ ...
    class_b/ ...
```

2 Choices & Defaults

- **Backbone**: `deit_small_patch16_224` (or distilled: `deit_small_distilled_patch16_224`).
- **Input**: 224×224, ImageNet normalization.
- **Optim**: AdamW; **Schedule**: warmup + cosine.
- **Augment**: RandAugment/AutoAugment, Mixup, CutMix, label smoothing.
- **LR scaling**: $\texttt{base\_lr} = 5\text{e}{-4} \times \frac{\texttt{batch}}{256}$ (full fine-tune).

3 Minimal Working Script (train + eval)

```
import torch, timm
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from timm.data import create_transform
from timm.loss import LabelSmoothingCrossEntropy,
    SoftTargetCrossEntropy
from timm.optim import create_optimizer_v2
from timm.scheduler import create_scheduler

# ---- Config ----
data_dir    = "data"
model_name  = "deit_small_patch16_224"   # or "
    deit_small_distilled_patch16_224"
num_classes = 10                         # <-- set to your dataset
epochs      = 50
batch_size  = 64
img_size    = 224
```

```python
use_mixup    = True
device       = "cuda" if torch.cuda.is_available() else "cpu"

# ---- Transforms ----
train_tf = create_transform(
    input_size=img_size, is_training=True,
    color_jitter=0.4, auto_augment='rand-m9-mstd0.5-inc1',
    re_prob=0.25, re_mode='pixel', re_count=1, interpolation='bicubic'
)
val_tf = create_transform(input_size=img_size, is_training=False, interpolation='bicubic')

# ---- Datasets / Loaders ----
train_ds = datasets.ImageFolder(f"{data_dir}/train", transform=train_tf)
val_ds   = datasets.ImageFolder(f"{data_dir}/val",   transform=val_tf)
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=8, pin_memory=True)
val_loader   = DataLoader(val_ds,   batch_size=batch_size, shuffle=False, num_workers=8, pin_memory=True)

# ---- Model ----
model = timm.create_model(model_name, pretrained=True, num_classes=num_classes).to(device)

# ---- Mixup/CutMix & Loss ----
mixup_fn = None
if use_mixup:
    from timm.data.mixup import Mixup
    mixup_fn = Mixup(mixup_alpha=0.8, cutmix_alpha=1.0, label_smoothing=0.1, num_classes=num_classes)
    criterion = SoftTargetCrossEntropy()
else:
    criterion = LabelSmoothingCrossEntropy(smoothing=0.1)

# ---- Optim / Scheduler ----
opt = create_optimizer_v2(model, opt='adamw', weight_decay=0.05, lr=5e-4 * (batch_size/256))
sched, _ = create_scheduler({'sched':'cosine','epochs':epochs,'warmup_epochs':5,'cooldown_epochs':0,'min_lr':1e-6}, opt)
scaler = torch.cuda.amp.GradScaler(enabled=(device=="cuda"))

def train_one_epoch():
    model.train()
    total, correct, loss_sum = 0, 0, 0.0
    for images, targets in train_loader:
        images, targets = images.to(device), targets.to(device)
        if mixup_fn: images, targets = mixup_fn(images, targets)
        opt.zero_grad(set_to_none=True)
        with torch.cuda.amp.autocast(enabled=(device=="cuda")):
            outputs = model(images)
            loss = criterion(outputs, targets)
        scaler.scale(loss).backward()
        scaler.step(opt); scaler.update()
        loss_sum += loss.item() * images.size(0)
        if not mixup_fn:
            pred = outputs.argmax(1); correct += (pred == targets).sum().item()
```

```
        total += images.size(0)
    sched.step(None)
    acc = (correct/total*100) if not mixup_fn else float('nan')
    return loss_sum/total, acc

@torch.no_grad()
def evaluate():
    model.eval()
    total, correct, loss_sum = 0, 0, 0.0
    for images, targets in val_loader:
        images, targets = images.to(device), targets.to(device)
        with torch.cuda.amp.autocast(enabled=(device=="cuda")):
            outputs = model(images)
            loss = nn.CrossEntropyLoss()(outputs, targets)
        loss_sum += loss.item() * images.size(0)
        pred = outputs.argmax(1); correct += (pred == targets).sum().item()
        total += images.size(0)
    return loss_sum/total, correct/total*100

best_acc, best_path = 0.0, "best_deit_s.pth"
for epoch in range(1, epochs+1):
    tr_loss, tr_acc = train_one_epoch()
    val_loss, val_acc = evaluate()
    if val_acc > best_acc:
        best_acc = val_acc
        torch.save({'model': model.state_dict()}, best_path)
    print(f"Epoch {epoch:03d} | train {tr_loss:.4f}/{tr_acc:.2f} | val {val_loss:.4f}/{val_acc:.2f}")
```

4 Strategy A: Feature Extraction (freeze backbone)

Use when data are scarce / risk of overfitting is high.

```
# Freeze everything except classifier head(s)
for n, p in model.named_parameters():
    if not (n.startswith('head') or n.startswith('fc') or 'distill' in n):
        p.requires_grad = False

# Train small head with a slightly higher LR and no weight decay
opt = create_optimizer_v2(
    filter(lambda p: p.requires_grad, model.parameters()),
    opt='adamw', weight_decay=0.0, lr=1e-3
)
```

Progressive unfreezing: after 5–10 epochs, unfreeze the last transformer block; then more blocks if validation improves.

5 Strategy B: Full Fine-Tuning (+ Layer-Wise LR Decay)

```
def param_groups_llrd(model, base_lr=5e-4, weight_decay=0.05, decay_rate=0.75):
    layers = []
    layers.append([*model.patch_embed.parameters(), model.pos_embed]) # shallowest
```

```
    for blk in model.blocks: layers.append(list(blk.parameters()))
    layers.append(list(model.head.parameters()))  # head
    n = len(layers); groups = []
    for i, params in enumerate(layers):
        lr = base_lr * (decay_rate ** (n - i - 1))
        groups.append({'params': params, 'lr': lr, 'weight_decay': weight_decay})
    return groups

opt = torch.optim.AdamW(param_groups_llrd(model), betas=(0.9, 0.999))
```

6 Distilled Variant (Optional)

If you choose `deit_small_distilled_patch16_224`, a distillation token/head is present. With pretrained distilled weights you can fine-tune *without* a teacher; train as usual (timm handles the two heads).

7 Hyperparameters That Usually Work

- Epochs: 50–100 (30–50 for smaller datasets).
- Batch size: as large as fits (32–256). Scale LR accordingly.
- LR: 5e−4 full fine-tune; 1e−3 head-only; 5 warmup epochs; cosine decay.
- Weight decay: 0.05 (head-only: 0.0–0.02).
- Augment: RandAugment; Mixup=0.8; CutMix=1.0; label smoothing=0.1.
- Regularization: keep DeiT-S default `drop_path` (∼0.1); use AMP.
- Early stopping: monitor val loss/acc (patience ≈10).

8 Evaluation Tips

Track top-1/top-5, confusion matrix, macro-F1 for imbalance. Consider EMA of weights (`ModelEmaV2` in `timm`) for stability.

9 Export / Deployment

```
model.eval()
example = torch.randn(1,3,224,224).to(device)
traced = torch.jit.trace(model, example)
traced.save("deit_s_traced.pt")
```

10 Sanity Checklist

☐ Correct `num_classes`.

☐ Proper `data/` layout and splits.

☐ 224 input, ImageNet normalization.

☐ LR scaled to batch; warmup on; cosine decay.

☐ AMP enabled; best checkpoint saved.

☐ Final test on a held-out set.