TP CNN & Transfer Learning (Correction en vert)

Kevin Helvig Aurélien Plyer
DTIS, ONERA, France

Note. Les ajouts et réponses du corrigé sont en vert.

0) Préparation (environnement & utilitaires)

```
# pip install --quiet timm==1.0.9
```

```
import torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms
import timm
import math
from tqdm.auto import tqdm

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Device:", device)

SEED = 42
torch.manual_seed(SEED)
torch.backends.cudnn.benchmark = True

def accuracy(logits, y):
    return (logits.argmax(1) == y).float().mean().item()

@torch.no_grad()
def evaluate(model, loader, loss_fn=nn.CrossEntropyLoss()):
    model.eval()
    tot_loss, tot_acc, n = 0.0, 0.0, 0
    for x, y in loader:
        x, y = x.to(device), y.to(device)
        logits = model(x)
        loss = loss_fn(logits, y)
        b = y.size(0)
        tot_loss += loss.item() * b
        tot_acc += (logits.argmax(1) == y).float().sum().item()
        n += b
    return tot_loss / n, tot_acc / n

def train(model, train_loader, val_loader, epochs=5, lr=1e-3, wd=0.0):
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=wd)
    loss_fn = nn.CrossEntropyLoss()
    best, best_state = math.inf, None
    for ep in range(1, epochs+1):
        model.train()
        pbar = tqdm(train_loader, desc=f"Epoch {ep}/{epochs}")
        for x, y in pbar:
            x, y = x.to(device), y.to(device)
            logits = model(x)
```

```
            loss = loss_fn(logits, y)
            opt.zero_grad()
            loss.backward() # @@propagation du gradient@@
            opt.step()
            pbar.set_postfix(loss=f"{loss.item():.3f}")
        val_loss, val_acc = evaluate(model, val_loader, loss_fn)
        print(f"Val | loss: {val_loss:.4f} | acc: {val_acc*100:.2f}%")
        if val_loss < best:
            best, best_state = val_loss, {k: v.cpu() for k,v in model.state_dict().items()}
    if best_state is not None:
        model.load_state_dict({k: v.to(device) for k,v in best_state.items()})
```

1) ConvNet basique sur MNIST

```
BATCH = 128
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
train_full = datasets.MNIST(root="./data", train=True, download=True, transform=transform
    )
test_set = datasets.MNIST(root="./data", train=False, download=True, transform=transform)

VAL_RATIO = 0.10
n_val = int(len(train_full) * VAL_RATIO)
n_train = len(train_full) - n_val
train_set, val_set = random_split(train_full, [n_train, n_val], generator=torch.Generator
    ().manual_seed(SEED))

train_loader = DataLoader(train_set, batch_size=BATCH, shuffle=True, num_workers=2,
    pin_memory=True)
val_loader = DataLoader(val_set, batch_size=BATCH, shuffle=False, num_workers=2,
    pin_memory=True)
test_loader = DataLoader(test_set, batch_size=BATCH, shuffle=False, num_workers=2,
    pin_memory=True)
```

```
class SmallCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),
        )
        self.head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*7*7, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, num_classes)
        )
    def forward(self, x):
        x = self.features(x)
        x = self.head(x)
        return x
```

```
model = SmallCNN().to(device)
train(model, train_loader, val_loader, epochs=5, lr=1e-3, wd=1e-4)
test_loss, test_acc = evaluate(model, test_loader)
print(f"Test | loss: {test_loss:.4f} | acc: {test_acc*100:.2f}%")
```

Réponse : Le réseau atteint typiquement entre 98 et 99% d'accuracy sur MNIST après 5 époques.

Question. Que se passe-t-il si vous remplacez `MaxPool2d(2)` par un `stride=2` ?

Réponse : La taille spatiale est aussi réduite, mais la convolution avec stride=2 apprend des filtres supplémentaires — plus flexible mais plus coûteux.

Question. Tracez l'évolution des loss.

Réponse : Les courbes montrent une convergence rapide, puis stabilisation. Si la loss validation augmente alors que train baisse, il y a sur-apprentissage.

Question. Ajoutez un bloc Convolution-ReLU supplémentaire.

Réponse : Cela augmente la profondeur et la capacité du réseau, mais il faut ajuster la taille du Linear. Généralement, légère amélioration du score.

Bonus. Adapter à CIFAR-10.

Réponse : CIFAR-10 est RGB (3 canaux) et 32×32, donc remplacer in_channels=3 et ajuster le Linear.

2) Effet des augmentations

```
aug_transform = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.RandomAffine(degrees=0, translate=(0.1,0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
    transforms.RandomErasing(p=0.2),
])
```

Réponse : Ces augmentations améliorent la robustesse du modèle aux variations de position et de forme.

Analyse. Comparez accuracy sans/avec augmentations.

Réponse : L'accuracy peut légèrement baisser sur le test propre mais se maintient mieux sur des entrées bruitées. Trop de rotation nuit à la stabilité.

3) Transfer learning : ResNet-18 (timm) → MNIST

```
tf_train_tl = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Lambda(lambda t: t.expand(3, *t.shape[1:])),
    transforms.Normalize(mean=(0.485,0.456,0.406), std=(0.229,0.224,0.225)),
])
```

Réponse : Le canal est dupliqué car le backbone ResNet attend une image RGB.

```
model_tl = timm.create_model('resnet18', pretrained=True, num_classes=10)
# Freeze
for name, p in model_tl.named_parameters():
    if not name.startswith('fc.'): p.requires_grad = False
train(model_tl, train_loader_tl, val_loader_tl, epochs=3, lr=1e-3)
# Fine-tune complet
for p in model_tl.parameters(): p.requires_grad = True
train(model_tl, train_loader_tl, val_loader_tl, epochs=2, lr=5e-4)
```

Réponse : Le LR faible évite de détruire les représentations pré-entraînées. Après fine-tuning, on atteint souvent 99%+.

Discussion. Si l'écart de domaine est fort ?

Réponse : Le transfert peut échouer. Solutions : backbone pré-entraîné sur un domaine plus proche, auto-supervision, ou pré-entraînement sur les données cibles.

Résumé des points clés du TP

1. Réseaux de convolution (CNN)

- La couche `Conv2d` apprend à extraire des motifs locaux (filtres) grâce à des noyaux glissants sur l'image. Ses paramètres clés sont : `in_channels`, `out_channels`, `kernel_size`, `stride` (pas du filtre) et `padding` (ajout de bordure).
- Les couches `ReLU` ajoutent de la non-linéarité, et `MaxPool2d` réduit la dimension spatiale.
- La hiérarchie de filtres (bords → formes → objets) permet l'invariance aux translations.
- Une tête `Linear` (ou MLP) transforme les cartes de caractéristiques en classes.
- Pour MNIST, deux blocs convolutionnels suffisent pour dépasser 98 % d'accuracy.

2. Importance des augmentations

- Les augmentations simulant des déformations réalistes (rotation, translation, effacement) améliorent la robustesse du modèle.
- Une amplitude excessive peut parfois dégrader les performances (ex. rotation $> 30°$).
- Elles jouent le rôle d'une *régularisation implicite* en enrichissant la distribution d'entraînement. De plus elles forcent le modèle à apprendre des invariance (associer la même classe à une donnée altérée).
- Sur MNIST : amélioration de la généralisation, baisse très limitée sur validation propre.

3. Transfer Learning (ResNet-18, lib. `timm`)

- On réutilise un modèle pré-entraîné (ici ResNet-18 sur ImageNet) pour tirer parti de ses représentations génériques. Hypothèse de filtres génériques.
- Le pipeline doit respecter le format d'entrée du modèle : 224×224, 3 canaux, normalisation ImageNet : permis grace au pipeline *Transforms* de Pytorch.
- Phase 1 : **Gel/Freeze du backbone** — seules les couches finales sont apprises (celles qui séparent la distribution en classe par exemple). Phase 2 : **Fine-tuning complet** avec un faible learning rate (empiriquement divisé par 10).
- Le *transfer learning* accélère la convergence et agit comme régularisation.
- Si le domaine cible diffère trop, on pourrait envisager d'utiliser un pré-entraînement auto-supervisé ou spécifique au domaine.

4. Bilan expérimental attendu

- CNN simple (MNIST) : $\approx$ 98.5–99 % d'accuracy.
- CNN + augmentations : plus robuste, meilleure stabilité sur données bruitées.
- Transfer learning ResNet-18 : $\geq$ 99 % d'accuracy avec convergence plus rapide.
- Impact du learning rate faible : préserve les poids pré-entraînés (fine-tuning stable).

5. À retenir

- Les CNN apprennent des hiérarchies de motifs "filtres" à partir des données brutes type images.
- Les augmentations jouent le rôle d'un “bruit utile”, rendant le modèle robuste aux variations du motif.
- Le transfert de modèles pré-entraînés est aujourd'hui la norme, permettant un apprentissage correct même sur de petits jeux de données.
- Toujours contrôler la compatibilité des tailles, canaux et normalisations entre datasets.