TP CNN & Transfer Learning (PyTorch, Jupyter/Colab)

Kevin Helvig
DTIS, ONERA, France

Aurélien Plyer
DTIS, ONERA, France

Objectifs. Construire un ConvNet simple sur MNIST, mesurer l'impact des augmentations, puis faire du *transfer learning* avec `timm` (ResNet-18). Chaque partie alterne **rappels de cours concis** et **cellules à compléter**.

Ressources utiles :

- `torch.nn.Conv2d` (doc PyTorch)
- `torchvision.transforms` (doc)
- `timm` (overview) & `pip timm` (install)

> **Rappel express CNN.** Une `Conv2d` applique C_{out} filtres $k \times k$ sur une entrée de C_{in} canaux. Sortie (H_{out}, W_{out}) : $H_{out} = \lfloor \frac{H+2P-k}{s} \rfloor + 1$ (*idem* pour W), où P est le padding et s le stride. Les opérations usuelles : ReLU, MaxPool (réduction spatiale), puis une tête MLP.

0) Préparation (environnement & utilitaires)

GPU recommandé. Si `timm` n'est pas installé, décommentez la ligne.

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

Cellule utilitaires (**fourni**) : métriques.

> **À faire.** Complétez les quelques trous manquants dans la boucle d'entraînement. Décrivez en commentaire à quoi correspondent ces pièces manquantes.

```
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

# metrique : accuracy
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 = ...
    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.(?) # TODO
            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

Données. MNIST : 60k train / 10k test, images 28×28 en niveaux de gris. Normalisez avec les stats canoniques ($\mu = 0{,}1307$, $\sigma = 0{,}3081$). Séparez un *validation set*.

> **À faire.** Complétez les quelques trous.

```
BATCH = 128 # TODO: ajuster si besoin / pas assez de RAM
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
# lignes de chargement du dataset
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 # TODO: 0.1 par dfaut
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))
```

```
# astuce windows : supprimer les num_workers en cas de plantage noyau
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)

len(train_set), len(val_set), len(test_set)
```

Modèle. Deux blocs `Conv2d+ReLU+MaxPool`. Calculez la taille de la carte finale pour définir le `Linear`. (Conv2d doc).

> **À faire.** Complétez le nombre de canaux, le `kernel_size`, et la dimension d'entrée du `Linear`. Expliquez en commentaire comment vous avez obtenu la dimensionalité souhaitée.

```
class SmallCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=..., out_channels=32, kernel_size=..., padding=1), # TODO
                : completer
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2), # 28->14
            nn.Conv2d(32, .., kernel_size=..., padding=...),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2), # trouver la dimensionalit ici :)
        )
        self.head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*..*.., 128), # TODO: calcul intermdiaire si vous changez le rseau
            nn.ReLU(inplace=True),
            nn.Linear(128, num_classes)
        )
    def forward(self, x):
        # TODO : complter forward pass
        ...
        ...
        return ...

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}%")
```

> **Question.** Que se passe-t-il si vous remplacez `MaxPool2d(2)` par un `stride=2` dans la convolution précédente ?

> **Question.** Tracez l'évolution des loss en train et en validation, par curiosité.

> **Question.** Ajoutez un bloc Convolution-RelU de votre choix dans self.features. Il va falloir réajuster des choses.

Bonus. Regarder le dataset CIFAR-10, ce qu'il faut adapter pour que l'architecture fonctionne. (Doc `CIFAR-10`)

2) Effet des augmentations (torchvision)

Idée. Les augmentations simulent des variations (rotations, translations, bruit, *erasing*) pour améliorer la robustesse. (Doc `transforms`).

À faire. Choisissez **3** transformations pertinentes pour MNIST (`RandomRotation`, `RandomAffine`, `ColorJitter` n'a pas d'effet en N&B, etc.). Justifiez vos choix en une ou deux phrases.

```
aug_transform = transforms.Compose([
    # TODO: choisir et paramtrer 2-3 transforms pertinentes (ordre inclus)
    ...
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])

train_full_aug = datasets.MNIST(root="./data", train=True, download=False, transform=
    aug_transform)
n_val = int(len(train_full_aug) * VAL_RATIO)
n_train = len(train_full_aug) - n_val
train_set_aug, val_set_aug = random_split(train_full_aug, [n_train, n_val], generator=
    torch.Generator().manual_seed(SEED))

train_loader_aug = DataLoader(train_set_aug, batch_size=BATCH, shuffle=True, num_workers
    =2, pin_memory=True)
val_loader_aug = DataLoader(val_set_aug, batch_size=BATCH, shuffle=False, num_workers=2,
    pin_memory=True)

model_aug = SmallCNN().to(device)
train(model_aug, train_loader_aug, val_loader_aug, epochs=5, lr=1e-3, wd=1e-4)

test_loss_aug, test_acc_aug = evaluate(model_aug, test_loader)
print(f"[Aug] Test | loss: {test_loss_aug:.4f} | acc: {test_acc_aug*100:.2f}%")
```

Analyse. Comparez `acc` sans/avec augmentations. Que se passe-t-il si vous augmentez l'amplitude des rotations ? Quelle est l'augmentation la plus nuisible ? la plus utile ?

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

Concept. On réutilise un backbone pré-entraîné ImageNet (3 canaux, 224×224), on adapte la tête à 10 classes. Deux phases fréquentes : *freeze* du backbone, puis *fine-tuning* complet léger. (Doc `Timm`)

À faire. Préparez le *pipeline* d'entrée :

- redimensionner en 224×224,
- dupliquer le canal ($1 \to 3$),
- normaliser avec les stats ImageNet ($\mu = (0.485, 0.456, 0.406)$, $\sigma = (0.229, 0.224, 0.225)$).

Expliquez en une phrase pourquoi on duplique le canal.

```
IMG_SIZE = 224

tf_train_tl = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    # TODO : completer
    transforms.Normalize(mean=(0.485,0.456,0.406), std=(0.229,0.224,0.225)),
])

tf_eval_tl = tf_train_tl # TODO: vous pouvez diffrencier train/val

train_full_tl = datasets.MNIST(root="./data", train=True, download=False, transform=
    tf_train_tl)
test_set_tl = datasets.MNIST(root="./data", train=False, download=False, transform=
    tf_eval_tl)
n_val = int(len(train_full_tl) * VAL_RATIO)
n_train = len(train_full_tl) - n_val
train_set_tl, val_set_tl = random_split(train_full_tl, [n_train, n_val], generator=torch.
    Generator().manual_seed(SEED))

#TODO: remplir les trous # astuce windows : enlever num_workers
train_loader_tl = DataLoader(..., batch_size=..., shuffle=..., num_workers=2, pin_memory=
    True)
val_loader_tl = DataLoader(val_set_tl, batch_size=BATCH, shuffle=..., num_workers=2,
    pin_memory=True)
test_loader_tl = DataLoader(..., batch_size=BATCH, shuffle=..., num_workers=2, pin_memory
    =True)
```

Modèle & stratégie. Créez un `resnet18` pré-entraîné et remplacez sa tête par 10 classes. Phase 1 : *freeze* tout sauf la tête ; Phase 2 : *fine-tuning* global léger.

> **À faire.** Complétez les parties `TODO` : gel des poids du backbone, puis dégel complet. Testez 3+2 époques (indicatif).

```
model_tl = timm.create_model('resnet18', pretrained=..., num_classes=10) #TODO : choisir
    le modle pr-entran, c'est plus mieux ;-)

# (1) Geler tout sauf la tte
for name, p in model_tl.named_parameters():
    if not name.startswith('fc.'): # TODO: vrifier le nom exact de la tte selon timm
        p.requires_grad = False

model_tl = model_tl.to(device)
train(model_tl, train_loader_tl, val_loader_tl, epochs=..., lr=..., wd=...) #TODO:
    completer

# (2) Dgeler pour affiner finement le modle
for p in model_tl.parameters():
    p.requires_grad = ... # TODO: dgeler les couches

train(model_tl, train_loader_tl, val_loader_tl, epochs=..., lr=5e-4, wd=1e-4) #Expliquer
    pourquoi on prends un learning rate si faible

test_loss_tl, test_acc_tl = evaluate(model_tl, test_loader_tl)
print(f"[ResNet18 TL] Test | loss: {test_loss_tl:.4f} | acc: {test_acc_tl*100:.2f}%")
```

Discussion. Le *transfer learning* apporte vitesse de convergence et régularisation implicite. Quels risques si l'écart de domaine est trop fort ? Que changeriez-vous pour des images bien plus variées ou éloignées, comme CIFAR-10 ?

Extensions (optionnel, guidées)

— Ajouter un `Dropout` après la couche cachée de la tête et mesurer l'effet.
— Implémenter un scheduler (p.ex. `CosineAnnealingLR`) et une label smoothing (`CrossEntropyLoss(label_s` discuter de l'effet.
— Tester d'autres architecture `timm` : `efficientnet_b0`, `convnext_tiny` et comparer le nombre de paramètres.

Conseils. Sur Colab, activez le GPU (Exécution → Modifier le type d'exécution → Accélérateur matériel = GPU). Gardez une trace de vos expériences (hyperparamètres, accuracy val/test, observations sur les augmentations).