andre marroquin 22266

sergio orellana 221122

nelson garcia

joaquin puente

Task 1

Arquitectura LeNet-5

imports, semillas y utilidades comunes

In [ ]:
```python
# imports principales
import os
import math
import time
from typing import Dict, Tuple

import torch
from torch import nn, optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, models

# asegurar reproducibilidad básica
def seed_everything(seed: int = 42) -> None:
    import random
    import numpy as np
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = False
    torch.backends.cudnn.benchmark = True

seed_everything(42)

# elegir dispositivo
def get_device() -> torch.device:
    if torch.cuda.is_available():
        return torch.device("cuda")
    if getattr(torch.backends, "mps", None) and torch.backends.mps.is_available():
```

```
        return torch.device("mps")
    return torch.device("cpu")

device = get_device()
device
```

Out[ ]:
```
device(type='cpu')
```

métricas accuracy y macro-f1 y bucles de entrenamiento/evaluación

In [ ]:
```
# calcular accuracy
def accuracy_from_logits(logits: torch.Tensor, targets: torch.Tensor) -> float:
    preds = logits.argmax(dim=1)
    correct = (preds == targets).sum().item()
    total = targets.numel()
    return correct / total

# construir matriz de confusión
def confusion_matrix(num_classes: int, preds: torch.Tensor, targets: torch.Tensor)
    cm = torch.zeros((num_classes, num_classes), dtype=torch.long)
    for t, p in zip(targets.view(-1), preds.view(-1)):
        cm[t.long(), p.long()] += 1
    return cm

# calcular macro-f1 desde matriz de confusión
def macro_f1_from_confusion(cm: torch.Tensor) -> float:
    cm = cm.to(torch.float32)
    tp = torch.diag(cm)
    fp = cm.sum(dim=0) - tp
    fn = cm.sum(dim=1) - tp

    precision = tp / torch.clamp(tp + fp, min=1.0)
    recall = tp / torch.clamp(tp + fn, min=1.0)
    f1 = 2.0 * precision * recall / torch.clamp(precision + recall, min=1e-12)
    return f1.mean().item()

# ciclo de entrenamiento por época
def train_one_epoch(model: nn.Module,
                    loader: DataLoader,
                    criterion: nn.Module,
                    optimizer: optim.Optimizer,
                    device: torch.device,
                    log_every: int = 100) -> Dict[str, float]:
    model.train()
    running_loss = 0.0
    running_acc = 0.0
    count = 0

    t0 = time.time()
    for step, (x, y) in enumerate(loader, 1):
        x, y = x.to(device), y.to(device)

        optimizer.zero_grad(set_to_none=True)
        logits = model(x)
```

```
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

        batch_acc = accuracy_from_logits(logits, y)
        running_loss += loss.item()
        running_acc += batch_acc
        count += 1

        if step % log_every == 0:
            print(f"step {step:04d} | loss {running_loss / count:.4f} | acc {runnin

    dt = time.time() - t0
    return {
        "train_loss": running_loss / max(count, 1),
        "train_acc":  running_acc / max(count, 1),
        "train_time_s": dt
    }

# evaluación completa
@torch.no_grad()
def evaluate(model: nn.Module,
             loader: DataLoader,
             criterion: nn.Module,
             device: torch.device,
             num_classes: int) -> Dict[str, float]:
    model.eval()
    total_loss = 0.0
    total_acc = 0.0
    count = 0
    cm = torch.zeros((num_classes, num_classes), dtype=torch.long)

    for x, y in loader:
        x, y = x.to(device), y.to(device)
        logits = model(x)
        loss = criterion(logits, y)

        total_loss += loss.item()
        total_acc += accuracy_from_logits(logits, y)
        count += 1

        preds = logits.argmax(dim=1)
        cm += confusion_matrix(num_classes, preds.cpu(), y.cpu())

    macro_f1 = macro_f1_from_confusion(cm)
    return {
        "val_loss": total_loss / max(count, 1),
        "val_acc":  total_acc / max(count, 1),
        "val_macro_f1": macro_f1
    }

# bucle de entrenamiento de varias épocas con mejor modelo por accuracy
def fit(model: nn.Module,
        train_loader: DataLoader,
        val_loader: DataLoader,
        criterion: nn.Module,
```

```
        optimizer: optim.Optimizer,
        scheduler,
        device: torch.device,
        num_classes: int,
        epochs: int = 5,
        name: str = "model") -> Tuple[nn.Module, Dict[str, float]]:
    best_state = None
    best_acc = -1.0
    history = {}

    for epoch in range(1, epochs + 1):
        print(f"\n==> {name} | epoch {epoch}/{epochs}")
        train_stats = train_one_epoch(model, train_loader, criterion, optimizer, de
        val_stats = evaluate(model, val_loader, criterion, device, num_classes)

        if scheduler is not None:
            scheduler.step()

        print(f"train  | loss {train_stats['train_loss']:.4f} acc {train_stats['tra
        print(f"valid  | loss {val_stats['val_loss']:.4f} acc {val_stats['val_acc']

        # actualizar mejor
        if val_stats["val_acc"] > best_acc:
            best_acc = val_stats["val_acc"]
            best_state = {k: v.cpu().clone() for k, v in model.state_dict().items()

        history[epoch] = {**train_stats, **val_stats}

    # cargar mejor estado antes de devolver
    if best_state is not None:
        model.load_state_dict(best_state)

    print(f"\n>> best val acc: {best_acc:.4f}")
    return model, history
```

datos para lenet-5 (mnist)

In [ ]:

```
# transforms para mnist
mnist_mean, mnist_std = (0.1307,), (0.3081,)

train_tf_mnist = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize(mnist_mean, mnist_std),
])

test_tf_mnist = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize(mnist_mean, mnist_std),
])

# cargar datasets y dataloaders
data_root = "./data"
```

```
train_mnist = datasets.MNIST(root=data_root, train=True, download=True, transform=t
test_mnist  = datasets.MNIST(root=data_root, train=False, download=True, transform=

batch_size_mnist = 128
num_workers = min(4, os.cpu_count() or 0)

train_loader_mnist = DataLoader(train_mnist, batch_size=batch_size_mnist, shuffle=T
val_loader_mnist   = DataLoader(test_mnist,  batch_size=batch_size_mnist, shuffle=F

len(train_mnist), len(test_mnist)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 14.9MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 436kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 3.75MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 4.80MB/s]
```

Out[ ]: (60000, 10000)

modelo lenet-5 (implementación clásica)

In [ ]:
```
# implementación clásica de LeNet-5
class LeNet5(nn.Module):
    def __init__(self, num_classes: int = 10):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, strid
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, str
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, num_classes)
        self.act = nn.Tanh()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.act(self.conv1(x))
        x = self.pool1(x)
        x = self.act(self.conv2(x))
        x = self.pool2(x)
        x = self.act(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = self.act(self.fc1(x))
        x = self.fc2(x)
        return x

# instanciar lenet-5
lenet5 = LeNet5(num_classes=10).to(device)

# definir criterio y optimizador
criterion_mnist = nn.CrossEntropyLoss()
optimizer_mnist = optim.SGD(lenet5.parameters(), lr=0.01, momentum=0.9, weight_deca
scheduler_mnist = optim.lr_scheduler.StepLR(optimizer_mnist, step_size=5, gamma=0.5

lenet5
```

Out[ ]:
```
LeNet5(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool1): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (conv3): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=120, out_features=84, bias=True)
  (fc2): Linear(in_features=84, out_features=10, bias=True)
  (act): Tanh()
)
```

entrenamiento y evaluación de lenet-5 (mnist)

In [ ]:
```python
# entrenar lenet-5
epochs_mnist = 5

lenet5, history_mnist = fit(
    model=lenet5,
    train_loader=train_loader_mnist,
    val_loader=val_loader_mnist,
    criterion=criterion_mnist,
    optimizer=optimizer_mnist,
    scheduler=scheduler_mnist,
    device=device,
    num_classes=10,
    epochs=epochs_mnist,
    name="lenet5-mnist"
)

# evaluación final en test
final_stats_mnist = evaluate(lenet5, val_loader_mnist, criterion_mnist, device, num
final_stats_mnist
```

```
==> lenet5-mnist | epoch 1/5
C:\Users\andre\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfr
a8p0\LocalCache\local-packages\Python312\site-packages\torch\utils\data\dataloader.p
y:665: UserWarning: 'pin_memory' argument is set as true but no accelerator is foun
d, then device pinned memory won't be used.
  warnings.warn(warn_msg)
```

```
step 0100 | loss 1.5063 | acc 0.6021
step 0200 | loss 0.9771 | acc 0.7418
step 0300 | loss 0.7628 | acc 0.7953
step 0400 | loss 0.6385 | acc 0.8272
train  | loss 0.5815 acc 0.8415 time 29.4s
valid  | loss 0.2084 acc 0.9386 macro_f1 0.9380

==> lenet5-mnist | epoch 2/5
step 0100 | loss 0.1998 | acc 0.9414
step 0200 | loss 0.1911 | acc 0.9446
step 0300 | loss 0.1803 | acc 0.9481
step 0400 | loss 0.1701 | acc 0.9507
train  | loss 0.1616 acc 0.9530 time 28.2s
valid  | loss 0.1109 acc 0.9687 macro_f1 0.9680

==> lenet5-mnist | epoch 3/5
step 0100 | loss 0.1056 | acc 0.9698
step 0200 | loss 0.1060 | acc 0.9699
step 0300 | loss 0.1020 | acc 0.9708
step 0400 | loss 0.0994 | acc 0.9715
train  | loss 0.0970 acc 0.9721 time 22.0s
valid  | loss 0.0712 acc 0.9775 macro_f1 0.9771

==> lenet5-mnist | epoch 4/5
step 0100 | loss 0.0750 | acc 0.9788
step 0200 | loss 0.0732 | acc 0.9791
step 0300 | loss 0.0733 | acc 0.9790
step 0400 | loss 0.0731 | acc 0.9791
train  | loss 0.0720 acc 0.9792 time 22.6s
valid  | loss 0.0618 acc 0.9806 macro_f1 0.9802

==> lenet5-mnist | epoch 5/5
step 0100 | loss 0.0593 | acc 0.9830
step 0200 | loss 0.0600 | acc 0.9825
step 0300 | loss 0.0600 | acc 0.9825
step 0400 | loss 0.0592 | acc 0.9827
train  | loss 0.0595 acc 0.9827 time 30.4s
valid  | loss 0.0493 acc 0.9858 macro_f1 0.9856

>> best val acc: 0.9858
```

Out[ ]:
```
{'val_loss': 0.049348667984580784,
 'val_acc': 0.9857594936708861,
 'val_macro_f1': 0.9855717420578003}
```

impresión de resultados

In [7]:
```
print("== resultados finales ==")
print(f"lenet-5 (mnist)  -> acc: {final_stats_mnist['val_acc']:.4f} | macro-f1: {fi
```

```
== resultados finales ==
lenet-5 (mnist)  -> acc: 0.9858 | macro-f1: 0.9856
```

Métrica de desempeño (definición y justificación)

Métrica elegida para LeNet-5 con el dataset MNIST: accuracy como métrica principal.
La razón es que MNIST tiene diez clases balanceadas y el objetivo es identificar correctamente el dígito.
La accuracy muestra de forma directa qué proporción de predicciones fue correcta y es muy fácil de interpretar.
También se usa la métrica macro F1 como apoyo, ya que muestra el equilibrio entre clases y ayuda a tener una visión más completa, aunque en este caso el dataset está bien balanceado.

Respuestas teóricas

a. Diferencia principal entre ambas arquitecturas
LeNet-5 es una red pequeña y de las primeras en su tipo. Tiene pocas capas, usa funciones de activación suaves y fue pensada para imágenes en blanco y negro de baja resolución, como las de 32 por 32 píxeles.
AlexNet, en cambio, es más grande y profunda. Utiliza activaciones más rápidas, técnicas de regularización como dropout y trabaja con imágenes a color de alta resolución.
En resumen, AlexNet tiene mucha más capacidad, maneja imágenes más grandes y usa técnicas más modernas para mejorar el rendimiento.

b. ¿Podría usarse LeNet-5 para el problema de AlexNet? ¿Y viceversa?
LeNet-5 podría usarse en problemas más complejos si se adapta, por ejemplo, agregando más canales de color o ampliando la entrada, pero su capacidad no sería suficiente para tareas tan grandes y diversas como las que resuelve AlexNet.
AlexNet, por otro lado, sí podría entrenarse con MNIST, pero sería demasiado grande para un problema tan simple. En ese caso podría sobreajustar o desperdiciar recursos sin aportar beneficios reales frente a modelos más ligeros como LeNet-5.

c. Aspectos más interesantes de cada arquitectura
En LeNet-5 destaca su simplicidad y eficiencia. A pesar de tener pocas capas, logra un gran desempeño en el reconocimiento de dígitos escritos a mano y es muy útil para aprender los fundamentos de las redes convolucionales.
En AlexNet, lo más interesante es cómo combina varias ideas que marcaron un cambio importante en la visión por computadora: el uso de activaciones rápidas, regularización y una estructura más profunda, lo que permitió alcanzar resultados sobresalientes en tareas a gran escala.