Laboratorio 9

andre marroquin 22266 sergio orellana 221122 nelson garcia 22434 joaquin puente 22296

Arquitectura AlexNet para CIFAR-10

Imports y configuración inicial

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

device = get_device()
device
```

```
device(type='cpu')
```

Funciones de entrenamiento y evaluación

```
# calcular accuracy
def accuracy_from_logits(logits: torch.Tensor, targets: torch.Tensor)
    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: t
    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, mi
    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()
```

```
48
49          batch_acc = accuracy_from_logits(logits, y)
50          running_loss += loss.item()
51          running_acc += batch_acc
52          count += 1
53
54          if step % log_every == 0:
55              print(f"step {step:04d} | loss {running_loss / count:.4f}
56
57      dt = time.time() - t0
58      return {
59          "train_loss": running_loss / max(count, 1),
60          "train_acc":  running_acc / max(count, 1),
61          "train_time_s": dt
62      }
63
64  # evaluación completa
65  @torch.no_grad()
66  def evaluate(model: nn.Module,
67               loader: DataLoader,
68               criterion: nn.Module,
69               device: torch.device,
70               num_classes: int) -> Dict[str, float]:
71      model.eval()
72      total_loss = 0.0
73      total_acc = 0.0
74      count = 0
75      cm = torch.zeros((num_classes, num_classes), dtype=torch.long)
76
77      for x, y in loader:
78          x, y = x.to(device), y.to(device)
79          logits = model(x)
80          loss = criterion(logits, y)
81
82          total_loss += loss.item()
83          total_acc += accuracy_from_logits(logits, y)
84          count += 1
85
86          preds = logits.argmax(dim=1)
87          cm += confusion_matrix(num_classes, preds.cpu(), y.cpu())
88
89      macro_f1 = macro_f1_from_confusion(cm)
90      return {
91          "val_loss": total_loss / max(count, 1),
92          "val_acc":  total_acc / max(count, 1),
93          "val_macro_f1": macro_f1
94      }
95
96  # bucle de entrenamiento de varias épocas con mejor modelo por accurac
97  def fit(model: nn.Module,
98          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,
        val_stats = evaluate(model, val_loader, criterion, device, num

        if scheduler is not None:
            scheduler.step()

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

        # 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_d

        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
```

Preparación de datos CIFAR-10

```
# transforms para cifar-10
# media y desviación estándar de cifar-10
cifar10_mean = (0.4914, 0.4822, 0.4465)
cifar10_std = (0.2470, 0.2435, 0.2616)

# data augmentation para entrenamiento
train_tf_cifar10 = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
```

```
    transforms.Normalize(cifar10_mean, cifar10_std),
])

# transforms para validación/test
test_tf_cifar10 = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(cifar10_mean, cifar10_std),
])

# cargar datasets cifar-10
data_root = "./data"

train_cifar10 = datasets.CIFAR10(root=data_root, train=True, download=T
test_cifar10 = datasets.CIFAR10(root=data_root, train=False, download=T

batch_size_cifar10 = 128
num_workers = min(4, os.cpu_count() or 0)

train_loader_cifar10 = DataLoader(train_cifar10, batch_size=batch_size_
val_loader_cifar10 = DataLoader(test_cifar10, batch_size=batch_size_cif

print(f"Train samples: {len(train_cifar10)}, Test samples: {len(test_ci
print(f"Classes: {train_cifar10.classes}")
```

```
Train samples: 50000, Test samples: 10000
Classes: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'h
```

Implementación de AlexNet

```
# implementación de AlexNet adaptada para CIFAR-10 (32x32)
# la arquitectura original fue diseñada para ImageNet (224x224)
# esta versión adapta las dimensiones para imágenes pequeñas

class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 10):
        super().__init__()

        # capa convolucional 1: entrada 3x32x32 -> salida 64x16x16
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_s
        self.relu1 = nn.ReLU(inplace=True)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # capa convolucional 2: entrada 64x16x16 -> salida 192x8x8
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=192, kernel
        self.relu2 = nn.ReLU(inplace=True)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # capa convolucional 3: entrada 192x8x8 -> salida 384x8x8
        self.conv3 = nn.Conv2d(in_channels=192, out_channels=384, kerne
```

```
        self.relu3 = nn.ReLU(inplace=True)

        # capa convolucional 4: entrada 384x8x8 -> salida 256x8x8
        self.conv4 = nn.Conv2d(in_channels=384, out_channels=256, kerne
        self.relu4 = nn.ReLU(inplace=True)

        # capa convolucional 5: entrada 256x8x8 -> salida 256x4x4
        self.conv5 = nn.Conv2d(in_channels=256, out_channels=256, kerne
        self.relu5 = nn.ReLU(inplace=True)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2)

        # adaptive pooling para garantizar tamaño fijo
        self.avgpool = nn.AdaptiveAvgPool2d((4, 4))

        # clasificador (fully connected layers)
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(256 * 4 * 4, 2048),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(2048, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # feature extraction
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.relu3(self.conv3(x))
        x = self.relu4(self.conv4(x))
        x = self.pool5(self.relu5(self.conv5(x)))

        x = self.avgpool(x)

        # flatten
        x = x.view(x.size(0), -1)

        # classification
        x = self.classifier(x)
        return x

# instanciar alexnet
alexnet = AlexNet(num_classes=10).to(device)
alexnet
```

```
AlexNet(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu1): ReLU(inplace=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
1))
  (relu2): ReLU(inplace=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (relu3): ReLU(inplace=True)
  (conv4): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (relu4): ReLU(inplace=True)
  (conv5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (relu5): ReLU(inplace=True)
  (pool5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (avgpool): AdaptiveAvgPool2d(output_size=(4, 4))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=4096, out_features=2048, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=2048, out_features=2048, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=2048, out_features=10, bias=True)
  )
)
```

Configuración del optimizador y scheduler

```
# configuración de entrenamiento para alexnet
criterion_cifar10 = nn.CrossEntropyLoss()
optimizer_cifar10 = optim.SGD(alexnet.parameters(), lr=0.01, momentum=0
scheduler_cifar10 = optim.lr_scheduler.StepLR(optimizer_cifar10, step_s

print("Configuración de entrenamiento:")
print(f"Criterio: CrossEntropyLoss")
print(f"Optimizador: SGD (lr=0.01, momentum=0.9, weight_decay=5e-4)")
print(f"Scheduler: StepLR (step_size=10, gamma=0.5)")
```

```
Configuración de entrenamiento:
Criterio: CrossEntropyLoss
Optimizador: SGD (lr=0.01, momentum=0.9, weight_decay=5e-4)
Scheduler: StepLR (step_size=10, gamma=0.5)
```

Entrenamiento de AlexNet en CIFAR-10

```
# entrenar alexnet en cifar-10
epochs_cifar10 = 20

alexnet, history_cifar10 = fit(
```

```
    model=alexnet,
    train_loader=train_loader_cifar10,
    val_loader=val_loader_cifar10,
    criterion=criterion_cifar10,
    optimizer=optimizer_cifar10,
    scheduler=scheduler_cifar10,
    device=device,
    num_classes=10,
    epochs=epochs_cifar10,
    name="alexnet-cifar10"
)
```

```
==> alexnet-cifar10 | epoch 1/20
/home/Japo/Documents/workspaces/uvg/deepLearning/Lab9-DL/.venv/lib/python3
  warnings.warn(warn_msg)
step 0100 | loss 2.3024 | acc 0.1035
step 0200 | loss 2.3013 | acc 0.1082
step 0300 | loss 2.2707 | acc 0.1350
train  | loss 2.2060 acc 0.1606 time 270.7s
valid  | loss 1.8994 acc 0.2955 macro_f1 0.2446

==> alexnet-cifar10 | epoch 2/20
step 0100 | loss 1.8874 | acc 0.2709
step 0200 | loss 1.8454 | acc 0.2919
step 0300 | loss 1.8092 | acc 0.3096
train  | loss 1.7721 acc 0.3248 time 294.3s
valid  | loss 1.5812 acc 0.4044 macro_f1 0.3747

==> alexnet-cifar10 | epoch 3/20
step 0100 | loss 1.5969 | acc 0.3925
step 0200 | loss 1.5710 | acc 0.4049
step 0300 | loss 1.5461 | acc 0.4173
train  | loss 1.5246 acc 0.4266 time 281.8s
valid  | loss 1.3690 acc 0.4812 macro_f1 0.4483

==> alexnet-cifar10 | epoch 4/20
step 0100 | loss 1.4078 | acc 0.4814
step 0200 | loss 1.3772 | acc 0.4886
step 0300 | loss 1.3583 | acc 0.4976
train  | loss 1.3427 acc 0.5048 time 238.2s
valid  | loss 1.1928 acc 0.5558 macro_f1 0.5450

==> alexnet-cifar10 | epoch 5/20
step 0100 | loss 1.2291 | acc 0.5494
step 0200 | loss 1.2159 | acc 0.5552
step 0300 | loss 1.2069 | acc 0.5597
train  | loss 1.1904 acc 0.5658 time 238.7s
valid  | loss 1.0453 acc 0.6263 macro_f1 0.6171

==> alexnet-cifar10 | epoch 6/20
step 0100 | loss 1.0938 | acc 0.5991
step 0200 | loss 1.0687 | acc 0.6130
step 0300 | loss 1.0545 | acc 0.6203
train  | loss 1.0436 acc 0.6240 time 238.3s
```

```
valid  | loss 1.0076 acc 0.6398 macro_f1 0.6390

==> alexnet-cifar10 | epoch 7/20
step 0100 | loss 0.9856 | acc 0.6435
step 0200 | loss 0.9801 | acc 0.6449
step 0300 | loss 0.9609 | acc 0.6535
train  | loss 0.9484 acc 0.6596 time 240.3s
valid  | loss 0.9087 acc 0.6803 macro_f1 0.6720

==> alexnet-cifar10 | epoch 8/20
step 0100 | loss 0.8760 | acc 0.6833
step 0200 | loss 0.8629 | acc 0.6887
step 0300 | loss 0.8634 | acc 0.6893
train  | loss 0.8610 acc 0.6919 time 239.9s
```

Evaluación final del modelo

```
# evaluación final en test set
final_stats_cifar10 = evaluate(alexnet, val_loader_cifar10, criterion_c

print("\n== Resultados finales AlexNet en CIFAR-10 ==")
print(f"Test Loss: {final_stats_cifar10['val_loss']:.4f}")
print(f"Test Accuracy: {final_stats_cifar10['val_acc']:.4f}")
print(f"Test Macro-F1: {final_stats_cifar10['val_macro_f1']:.4f}")

final_stats_cifar10
```

```
== Resultados finales AlexNet en CIFAR-10 ==
Test Loss: 0.4930
Test Accuracy: 0.8287
Test Macro-F1: 0.8306
{'val_loss': 0.49304004202160656,
 'val_acc': 0.8287183544303798,
 'val_macro_f1': 0.8305532336235046}
```

Métrica de desempeño:

Las métricas utilizadas para evaluar el rendimiento del modelo son la precisión (accuracy), la matriz de confusión y el Macro-F1 Score. La precisión mide la proporción de predicciones correctas sobre el total de ejemplos, siendo una métrica básica pero importante para evaluar el rendimiento global del modelo. La matriz de confusión permite analizar el desempeño del modelo en términos de verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos, proporcionando una visión detallada de cómo el modelo clasifica cada clase. Finalmente, el Macro-F1 Score es crucial en este contexto porque toma en cuenta el balance entre precisión y recall para cada clase y luego calcula un promedio,

lo que lo hace especialmente útil cuando se desea una evaluación equilibrada de todas las clases, incluso en conjuntos de datos con un número equilibrado de clases, como es el caso de CIFAR-10. Estas métricas, en conjunto, permiten una evaluación más completa del modelo, ayudando a identificar tanto la exactitud general como el rendimiento por clase.

```
1 Start coding or generate with AI.
```

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.

Integrantes

- Andre Marroquin
- Joaquin Puente
- Sergio Orellana
- Nelson Garcia

1. **¿En qué casos son útiles estas arquitecturas?**

a) **GoogleNet (Inception)**

Cuándo la usaría:

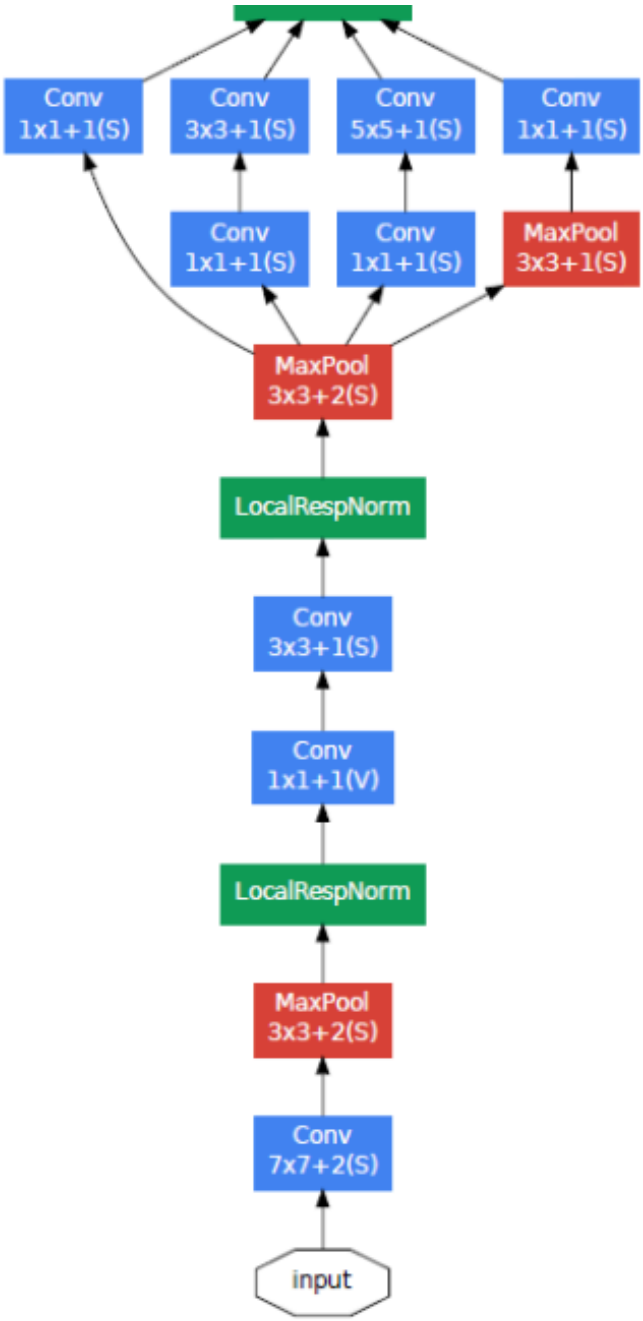
- Cuando necesito capturar características a múltiples escalas (texturas finas y patrones grandes) en la misma capa.

- Si busco buena precisión con presupuesto de cómputo moderado, por ejemplo en clasificación en la nube con recursos limitados.
- Para reducir parámetros frente a CNNs clásicas profundas (gracias a 1×1 conv y “global average pooling”).

Por qué: combina en paralelo conv 1×1, 3×3, 5×5 y pooling dentro de un módulo Inception, y concatena sus salidas; además usa clasificadores auxiliares durante el entrenamiento para estabilizar gradientes.


softmax2
SoftmaxActivation
FC
AveragePool 7x7+1(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv
Conv
Conv
Conv
softmax1
SoftmaxActivation
FC
FC
Conv 1x1+1(S)
AveragePool 5x5+3(V)


1x1+1(S)
3x3+1(S)
5x5+1(S)
1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
softmax0
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
SoftmaxActivation
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
FC
DepthConcat
FC
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat


Conv
1x1+1(S)
Conv
3x3+1(S)
Conv
5x5+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
Conv
1x1+1(S)
MaxPool
3x3+1(S)
MaxPool
3x3+2(S)
LocalRespNorm
Conv
3x3+1(S)
Conv
1x1+1(V)
LocalRespNorm
MaxPool
3x3+2(S)
Conv
7x7+2(S)
input

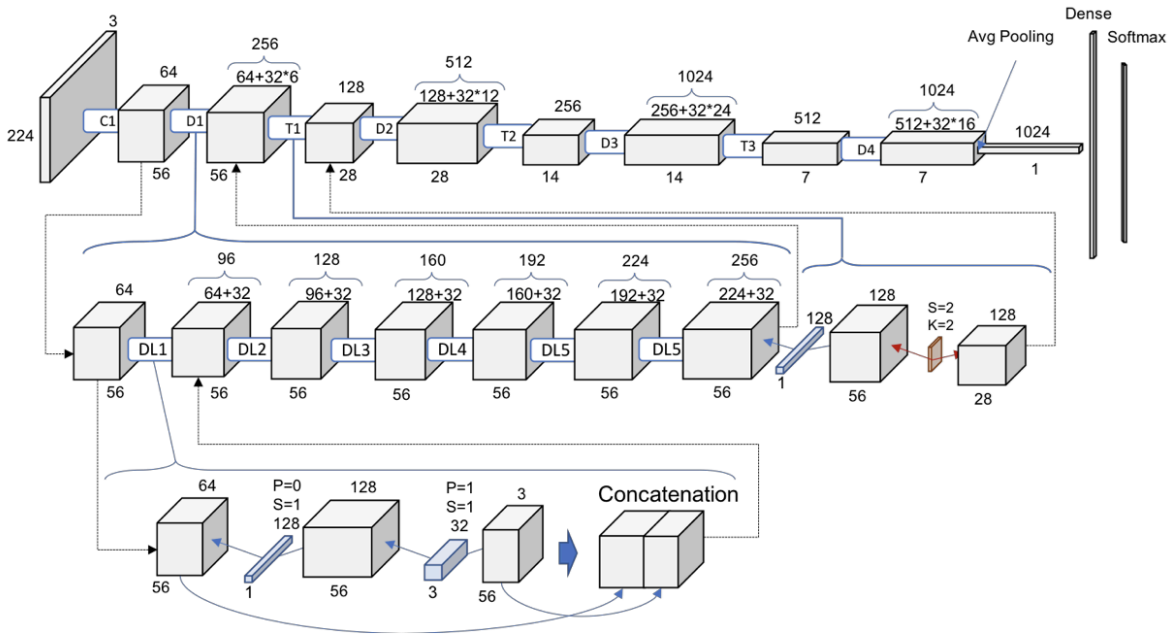
b) **DenseNet (Densely Connected Convolutional Networks)**

Cuándo la usaría:

- Cuando quiero máximo reuso de características y mejor flujo de gradiente (evitar desvanecimiento), útil en datasets medianos/pequeños.
- Si necesito modelos relativamente compactos (sorprendentemente eficientes en parámetros para su profundidad).
- En tareas donde ayuda combinar rasgos de bajo y alto nivel (ej. clasificación, segmentación).

Por qué: cada capa recibe como entrada el concat de todas las salidas

previas en el bloque denso; las transition layers controlan el crecimiento con 1×1 conv y pooling.



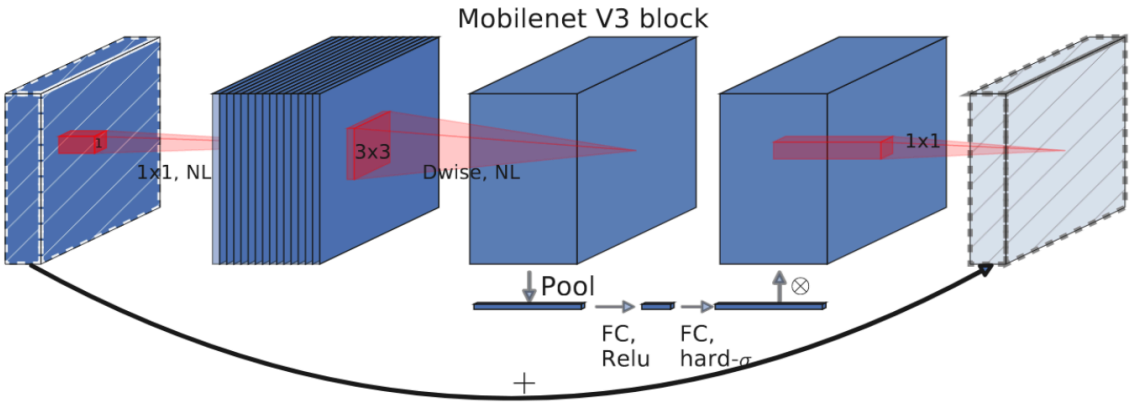
c) **MobileNet**

Cuándo la usaría:

- Para inferencia en dispositivos móviles/embebidos (apps on-device, IoT, robótica con tiempo real).

- Cuando el requisito clave es baja latencia y bajo consumo con una caída mínima de precisión.
- Para despliegues a gran escala donde el costo por consulta importa.

Por qué: usa convoluciones separables en profundidad (depthwise + pointwise 1×1) y, en V2/V3, bottlenecks invertidos y ReLU6, logrando grandes ahorros en cómputo y parámetros.

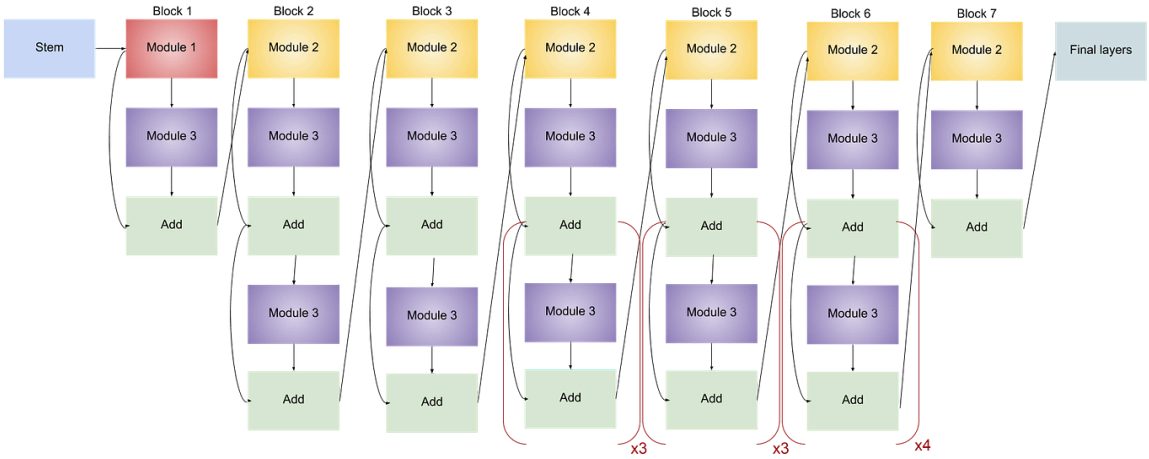


d) **EfficientNet**

Cuándo la usaría:

- Cuando quiero mejor precisión-eficiencia y además escalar el modelo (pequeño → grande) de forma sistemática.
- En sistemas donde puedo elegir entre variantes B0–B7 según mi presupuesto de FLOPs/memoria.
- Para competir con SOTA en clasificación manteniendo un buen costo.

Por qué: introduce el compound scaling (escala coordinadamente profundidad, ancho y resolución) y usa bloques MBConv con Squeeze-and-Excitation (inspirados en MobileNetV2).



2. **¿Cómo puedo usar Transformers para image recognition?**

Así lo haría con un Vision Transformer (ViT):

1. Particionar la imagen en parches (p.ej., 16×16), aplanarlos y proyectarlos linealmente para obtener tokens; añado embeddings posicionales.
2. Paso la secuencia por un encoder Transformer (múltiples capas de

auto-atención multi-cabeza + MLP + normalizaciones).

3. Prependo un token [CLS] (o equivalente) y su representación final alimenta una capa densa de clasificación.
4. Usar pre-entrenamiento grande y fine-tuning en mi dataset; o variantes híbridas que mezclan CNNs y Transformers cuando quiero inductive bias local con contexto global.

Cuándo lo usaría:

- Cuando necesito contexto global explícito y flexibilidad para múltiples tareas (clasificación, detección, segmentación) sin depender de convoluciones.

- Si dispongo de muchos datos (o técnicas de data-efficient training) y busco escalabilidad del modelo.

Referencias

- GeeksforGeeks. (2025a, June 30). Depth wise Separable Convolutional Neural Networks. GeeksforGeeks. https://www.geeksforgeeks.org/machine-learning/depth-wise-separable-convolutional-neural-networks/
- GeeksforGeeks. (2025b, July 15). Understanding GoogLeNet Model CNN Architecture. GeeksforGeeks. https://www.geeksforgeeks.org/ma

chine-learning/understanding-googlenet-model-cnn-architecture/

- GeeksforGeeks. (2025c, July 23). DenseNet explained. GeeksforGeeks. https://www.geeksforgeeks.org/computer-vision/densenet-explained/
- GeeksforGeeks. (2025d, July 23). EfficientNet Architecture. GeeksforGeeks. https://www.geeksforgeeks.org/computer-vision/efficientnet-architecture/
- GeeksforGeeks. (2025e, July 23).