Práctica 2: Deep Learning para Clasificación

Sistemas Inteligentes para la Gestión en la Empresa

Autor: Pablo Valenzuela Álvarez (pvalenzuela@correo.ugr.es)

```
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import random_split, DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
from collections import Counter
import pandas as pd

# Definimos el dispositivo (GPU o CPU)
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)

random_seed = 12345
torch.manual_seed(random_seed)
```

```
cuda:0
<torch._C.Generator at 0x7b176033ddb0>
```

Carga de datos

```
from google.colab import drive
drive.mount('/content/drive')
PATH = "/content/drive/MyDrive/Colab Notebooks/pr2-starting-package/starting-package"

# PATH = "G:\Mi unidad\Colab Notebooks\pr2-starting-package\starting-package"

DATA_x20 = PATH + "/data x20"
DATA_x200 = PATH + "/data x200"

MODEL_x20 = "model_x20.pth"
MODEL_x200 = "model_x200.pth"
```

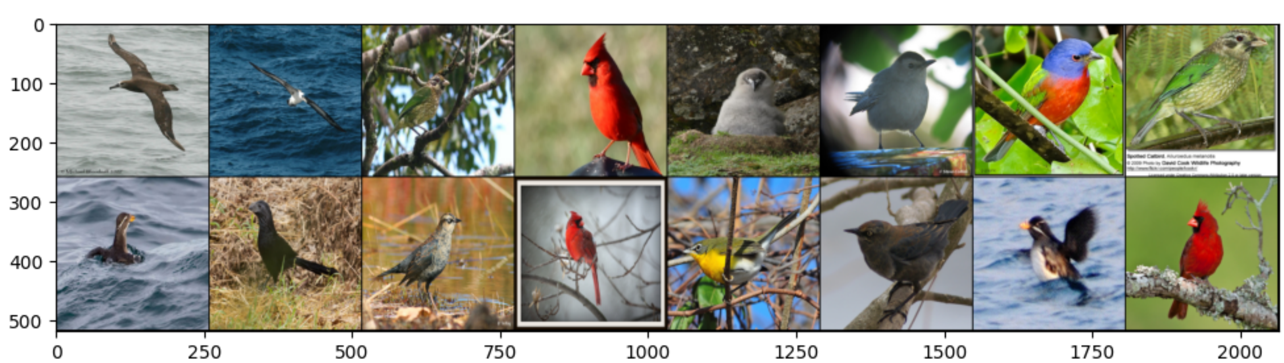
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.ı
```

Exploración de datos

```
# dimensions = ImageFolder(DATA_x20)
# shapes = [(img.height, img.width) for img, _ in dimensions]
# heights, widths = [[h for h,_ in shapes], [w for _,w in shapes]]
# median_height = int(np.median(heights))
# median_width = int(np.median(widths))
median_height = 375 # son los resultados que salen de ejecutar las lineas de arriba
median_width = 500
print(f"Tamaño medio de las imagenes [height: {median_height}, width: {median_width}]")
```

```
Tamaño medio de las imagenes [height: 375, width: 500]
```

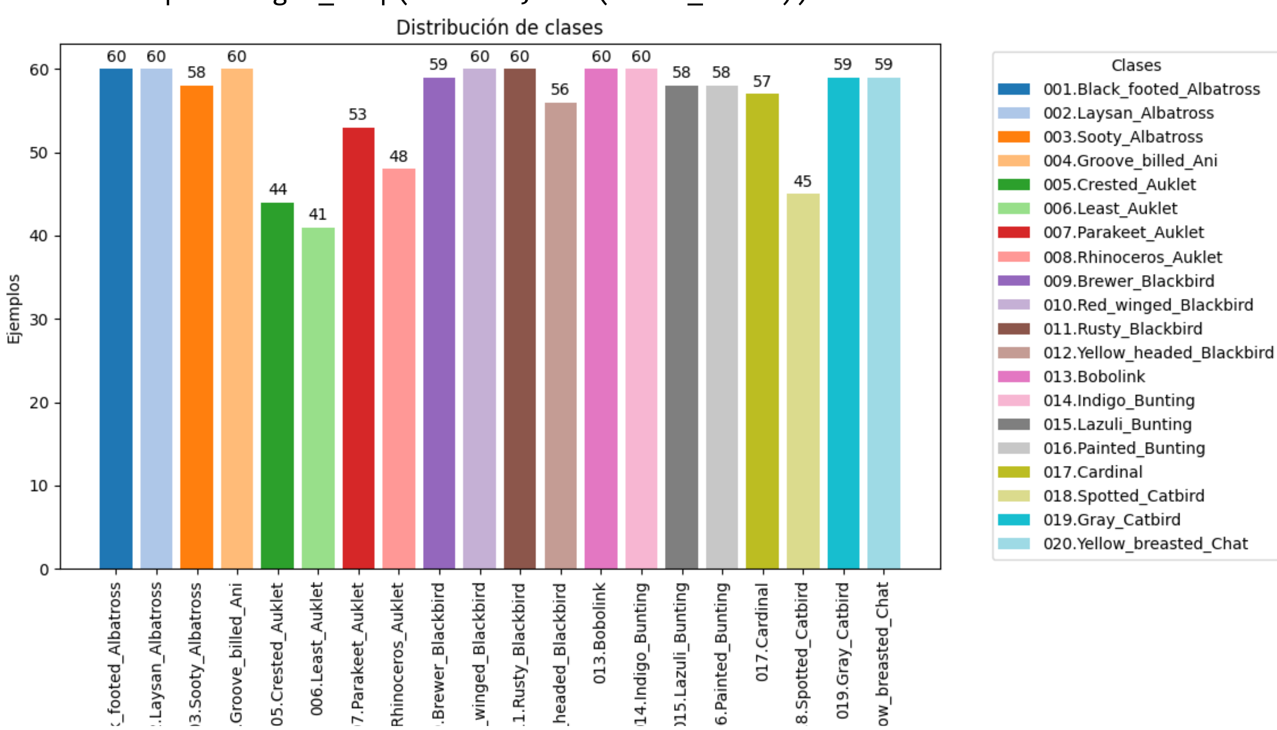
```
exp_transforms = transforms.Compose([
    transforms.Resize((256,256)),
    transforms.ToTensor()
])

dataset = ImageFolder(root=DATA_x20, transform=exp_transforms)
classes = dataset.classes

mini_batch=16
sample = DataLoader(dataset, batch_size=mini_batch, shuffle=True)

def imshow(img):
    npimg = img.numpy()
    plt.figure(figsize=(12, 8))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

dataiter = iter(sample)
images, labels = next(dataiter)
imshow(torchvision.utils.make_grid(images))
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(mini_batch)))
```

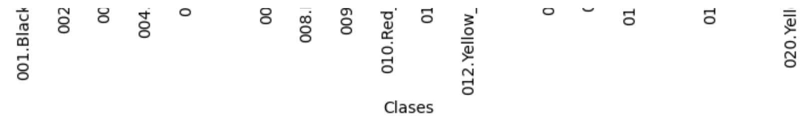



```
001.Black_footed_Albatross 002.Laysan_Albatross 018.Spotted_Catbird 017.Cardinal 003
```

```
class_counts = Counter(dataset.targets)

class_names = [classes[i] for i in class_counts.keys()]
counts = [class_counts[i] for i in class_counts.keys()]

colors = plt.cm.get_cmap('tab20', len(class_names)).colors

plt.figure(figsize=(10, 6))
bars = plt.bar(class_names, counts, color=colors)
plt.xlabel('Clases')
plt.ylabel('Ejemplos')
plt.title('Distribución de clases')
plt.xticks(rotation=90)


for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval + 0.5, int(yval), ha='center', va='bo

plt.legend(bars, class_names, title="Clases", bbox_to_anchor=(1.05, 1), loc='upper left'

plt.show()
```

```
<ipython-input-39-13fb55626ea7>:6: MatplotlibDeprecationWarning: The get_cmap functi
  colors = plt.cm.get_cmap('tab20', len(class_names)).colors
```





Particionamiento de datos

```
# valores de normalizacion
imagenet_stats = ([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

# transformaciones para entrenar
train_transforms = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(*imagenet_stats)
])

# transformaciones para validar
test_transforms = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(*imagenet_stats)
])

# Cargamos el conjunto de datos
full_dataset = ImageFolder(root=DATA_x20)
classes = full_dataset.classes

# Dividimos los conjuntos
split = 0.8  # 80/20
```

```
train_size = int(split * len(full_dataset))
test_size = len(full_dataset) - train_size
trainset, testset = random_split(full_dataset, [train_size, test_size])

print(f"Tamaño del conjunto de entreno: {train_size}")
print(f"Tamaño del conjunto de validación: {test_size}")

# Aplicamos las transformaciones
trainset.dataset.transform = train_transforms
testset.dataset.transform = test_transforms

# Creamos los DataLoaders
trainloader = DataLoader(trainset, batch_size=16, shuffle=True, num_workers=2)
testloader = DataLoader(testset, batch_size=16, shuffle=False, num_workers=2)
```

```
Tamaño del conjunto de entreno: 892
Tamaño del conjunto de validación: 223
```

Clasificación

```
class BirdCNN(nn.Module):
    def __init__(self):
        super(BirdCNN, self).__init__()

        # Capas convolucionales
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)

        # Capa de pooling
        self.pool = nn.MaxPool2d(2, 2)

        # Capas totalmente conectadas
        self.fc1 = nn.Linear(256 * 16 * 16, 512)
        self.fc3 = nn.Linear(512, len(classes))

        # Capa dropout
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc3(x)
```

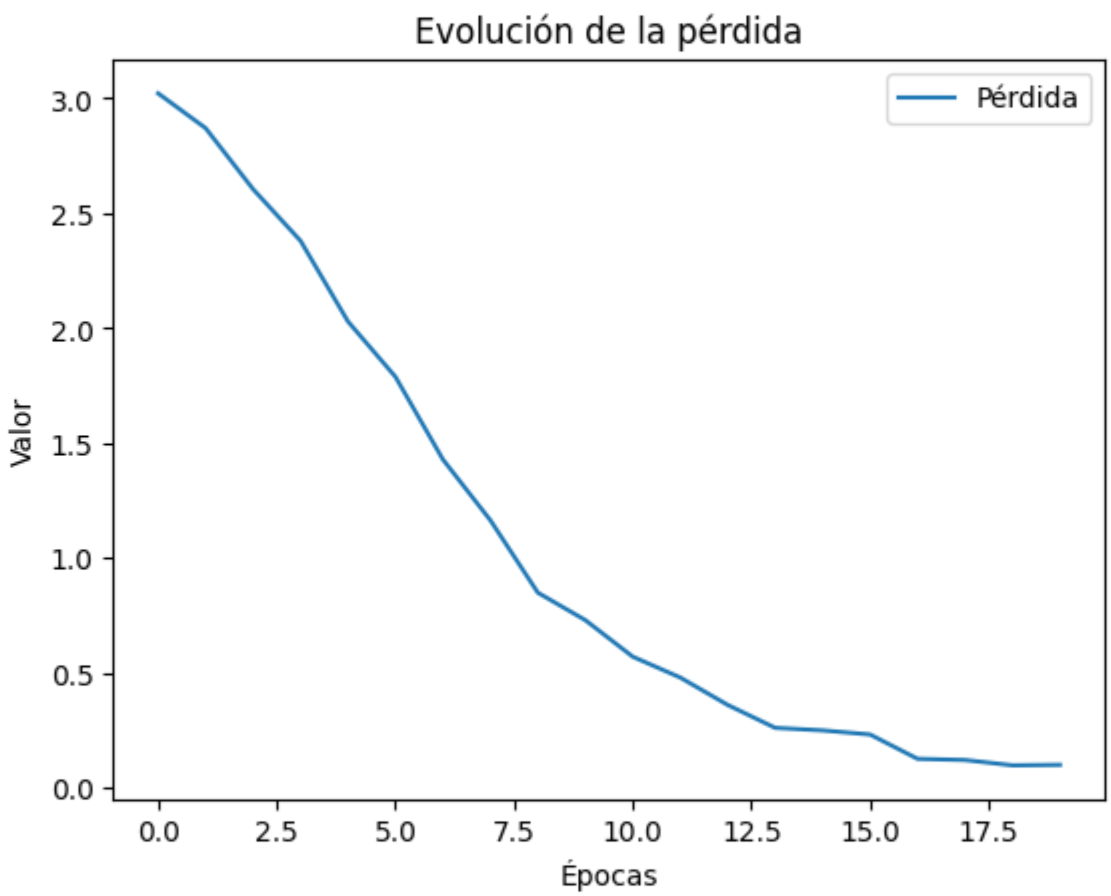
```
        return x

model = BirdCNN()

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

def train_model(model, train_loader, test_loader, criterion, optimizer, num_epochs=10):

    metrics = {"loss": [], "accuracy": []}

    for epoch in range(num_epochs):

        ### Entrenamiento ###

        running_loss = 0.0
        model.train()

        tqdm_train = tqdm(enumerate(train_loader), total=len(train_loader))

        for i, data in tqdm_train:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device) # movemos a la GPU, si
            optimizer.zero_grad()               # resetea los gradientes
            outputs = model(inputs)             # calcula las salidas para las entradas (
            loss = criterion(outputs, labels)   # calcula las perdidas
            loss.backward()                     # calcula los gradientes con las perdidas
            optimizer.step()                    # actualiza los parametros del modelo
            running_loss += loss.item()
            tqdm_train.set_description(f'Epoch {epoch + 1}/{num_epochs}, Loss: {running_]


        ### Validación ###

        model.eval()
        val_loss = 0.0
        correct = 0
        total = 0

        tqdm_val = tqdm(enumerate(test_loader), total=len(test_loader))

        with torch.no_grad():           # funcionamiento similar al entreno
            for j, data in tqdm_val:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)           # obtiene las predic(
                total += labels.size(0)
```

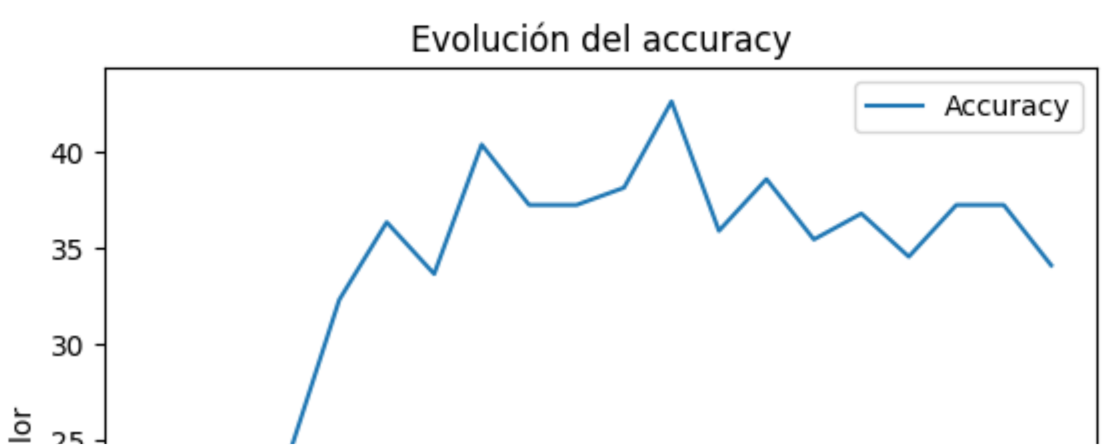
```
                correct += (predicted == labels).sum().item()       # predicciones corre

                tqdm_val.set_description(f'  --> Epoch {epoch + 1} accuracy: {(100 * corr

        metrics["loss"].append(np.round(running_loss/len(train_loader),4))
        metrics["accuracy"].append(np.round((100 * correct / total),4))

    return metrics

model.to(device)
metrics = train_model(model, trainloader, testloader, criterion, optimizer, num_epochs=20)
```

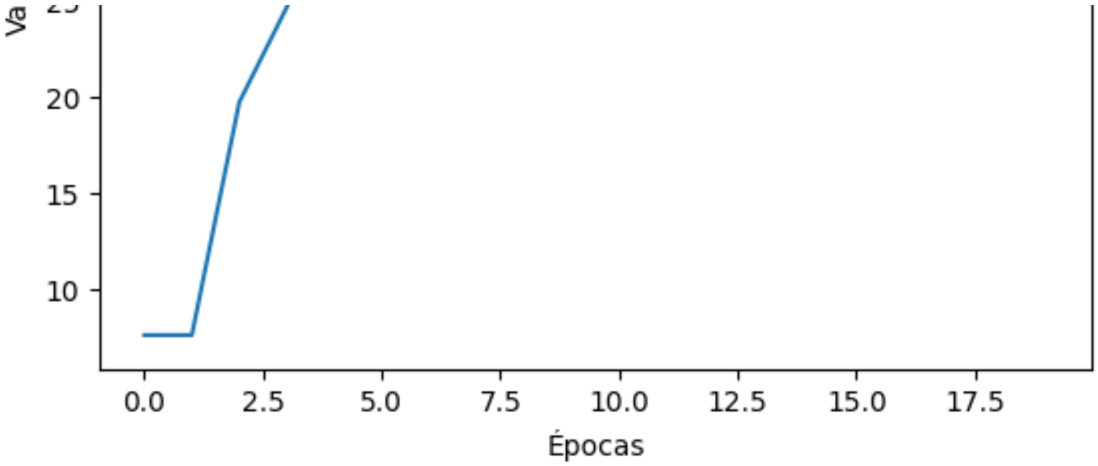
```
Epoch 1/20, Loss: 3.022: 100%|██████████| 56/56 [00:07<00:00,  7.03it/s]
  --> Epoch 1 accuracy: 7.62%: 100%|██████████| 14/14 [00:02<00:00,  6.30it/s]
Epoch 2/20, Loss: 2.871: 100%|██████████| 56/56 [00:09<00:00,  5.99it/s]
  --> Epoch 2 accuracy: 7.62%: 100%|██████████| 14/14 [00:01<00:00,  7.34it/s]
Epoch 3/20, Loss: 2.605: 100%|██████████| 56/56 [00:09<00:00,  5.83it/s]
  --> Epoch 3 accuracy: 19.73%: 100%|██████████| 14/14 [00:02<00:00,  5.06it/s]
Epoch 4/20, Loss: 2.380: 100%|██████████| 56/56 [00:08<00:00,  6.72it/s]
  --> Epoch 4 accuracy: 24.66%: 100%|██████████| 14/14 [00:03<00:00,  4.54it/s]
Epoch 5/20, Loss: 2.029: 100%|██████████| 56/56 [00:07<00:00,  7.42it/s]
  --> Epoch 5 accuracy: 32.29%: 100%|██████████| 14/14 [00:01<00:00,  7.46it/s]
Epoch 6/20, Loss: 1.788: 100%|██████████| 56/56 [00:09<00:00,  5.87it/s]
  --> Epoch 6 accuracy: 36.32%: 100%|██████████| 14/14 [00:01<00:00,  7.41it/s]
Epoch 7/20, Loss: 1.429: 100%|██████████| 56/56 [00:09<00:00,  5.65it/s]
  --> Epoch 7 accuracy: 33.63%: 100%|██████████| 14/14 [00:02<00:00,  5.71it/s]
Epoch 8/20, Loss: 1.164: 100%|██████████| 56/56 [00:08<00:00,  6.94it/s]
  --> Epoch 8 accuracy: 40.36%: 100%|██████████| 14/14 [00:03<00:00,  4.45it/s]
Epoch 9/20, Loss: 0.849: 100%|██████████| 56/56 [00:08<00:00,  6.88it/s]
  --> Epoch 9 accuracy: 37.22%: 100%|██████████| 14/14 [00:01<00:00,  7.31it/s]
Epoch 10/20, Loss: 0.729: 100%|██████████| 56/56 [00:09<00:00,  5.79it/s]
  --> Epoch 10 accuracy: 37.22%: 100%|██████████| 14/14 [00:03<00:00,  4.65it/s]
Epoch 11/20, Loss: 0.570: 100%|██████████| 56/56 [00:10<00:00,  5.14it/s]
  --> Epoch 11 accuracy: 38.12%: 100%|██████████| 14/14 [00:01<00:00,  7.34it/s]
Epoch 12/20, Loss: 0.479: 100%|██████████| 56/56 [00:08<00:00,  7.00it/s]
  --> Epoch 12 accuracy: 42.60%: 100%|██████████| 14/14 [00:03<00:00,  4.41it/s]
Epoch 13/20, Loss: 0.360: 100%|██████████| 56/56 [00:07<00:00,  7.55it/s]
  --> Epoch 13 accuracy: 35.87%: 100%|██████████| 14/14 [00:01<00:00,  7.87it/s]
Epoch 14/20, Loss: 0.261: 100%|██████████| 56/56 [00:09<00:00,  5.88it/s]
  --> Epoch 14 accuracy: 38.57%: 100%|██████████| 14/14 [00:01<00:00,  8.15it/s]
Epoch 15/20, Loss: 0.249: 100%|██████████| 56/56 [00:08<00:00,  6.44it/s]
  --> Epoch 15 accuracy: 35.43%: 100%|██████████| 14/14 [00:02<00:00,  5.83it/s]
Epoch 16/20, Loss: 0.231: 100%|██████████| 56/56 [00:07<00:00,  7.62it/s]
  --> Epoch 16 accuracy: 36.77%: 100%|██████████| 14/14 [00:01<00:00,  7.20it/s]
Epoch 17/20, Loss: 0.126: 100%|██████████| 56/56 [00:09<00:00,  6.02it/s]
  --> Epoch 17 accuracy: 34.53%: 100%|██████████| 14/14 [00:01<00:00,  7.83it/s]
Epoch 18/20, Loss: 0.120: 100%|██████████| 56/56 [00:09<00:00,  5.93it/s]
  --> Epoch 18 accuracy: 37.22%: 100%|██████████| 14/14 [00:01<00:00,  7.69it/s]
Epoch 19/20, Loss: 0.097: 100%|██████████| 56/56 [00:07<00:00,  7.56it/s]
  --> Epoch 19 accuracy: 37.22%: 100%|██████████| 14/14 [00:02<00:00,  5.16it/s]
Epoch 20/20, Loss: 0.099: 100%|██████████| 56/56 [00:08<00:00,  6.62it/s]
  --> Epoch 20 accuracy: 34.08%: 100%|██████████| 14/14 [00:01<00:00,  7.71it/s]
```

```
df_metrics = pd.DataFrame(metrics)
```

```
df_metrics = pd.DataFrame(metrics)

df_loss = df_metrics[["loss"]].plot()
plt.xlabel('Épocas')
plt.ylabel('Valor')
plt.title('Evolución de la pérdida')
plt.legend(['Pérdida'])
plt.show()

df_acc = df_metrics[["accuracy"]].plot()
plt.xlabel('Épocas')
plt.ylabel('Valor')
plt.title('Evolución del accuracy')
plt.legend(['Accuracy'])
plt.show()
```








```
# Guarda el modelo
torch.save(model.state_dict(), MODEL_x20)
# torch.save(model.state_dict(), MODEL_x200)
```

TEST

```
# Carga del modelo
test_model = BirdCNN()
test_model.load_state_dict(torch.load(MODEL_x20))
# test_model.load_state_dict(torch.load(MODEL_x200))
```

```
<All keys matched successfully>
```

```
# Función para evaluar el modelo
def evaluate_model(model, test_loader):
    model.eval()
    correct_global = 0
    total_global = 0
    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total_global += labels.size(0)
            correct_global += (predicted == labels).sum().item()
            for label, prediction in zip(labels, predicted):
                if label == prediction:
                    correct_pred[classes[label]] += 1
                total_pred[classes[label]] += 1

    print(f'Accuracy: {np.round((100 * correct_global / total_global),2)}%')
```

```
    # for classname, correct_count in correct_pred.items():
    #     accuracy = 100 * float(correct_count) / total_pred[classname]
        # print(f'Accuracy de: {classname:5s} --> {accuracy:.2f}%')

    plot_class_accuracy(correct_pred, total_pred)

def plot_class_accuracy(correct_pred, total_pred):
    classes = list(correct_pred.keys())
    accuracy = [(correct_pred[classname] / total_pred[classname]) * 100 for classname in c


    colors = plt.cm.get_cmap('tab20', len(class_names)).colors

    plt.figure(figsize=(10, 5))
    bars = plt.bar(classes, accuracy, color=colors)

    for bar, acc in zip(bars, accuracy):
        plt.text(bar.get_x() + bar.get_width() / 2 - 0.15, bar.get_height() + 1, f'{acc:.2

    plt.xlabel('Clases')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy por clase')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()
    plt.show()


# Evaluar el modelo
evaluate_model(test_model, testloader)
```

```
Accuracy: 34.08%
<ipython-input-49-e07b144a344a>:34: MatplotlibDeprecationWarning: The get_cmap funct:
  colors = plt.cm.get_cmap('tab20', len(class_names)).colors
```