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"
```

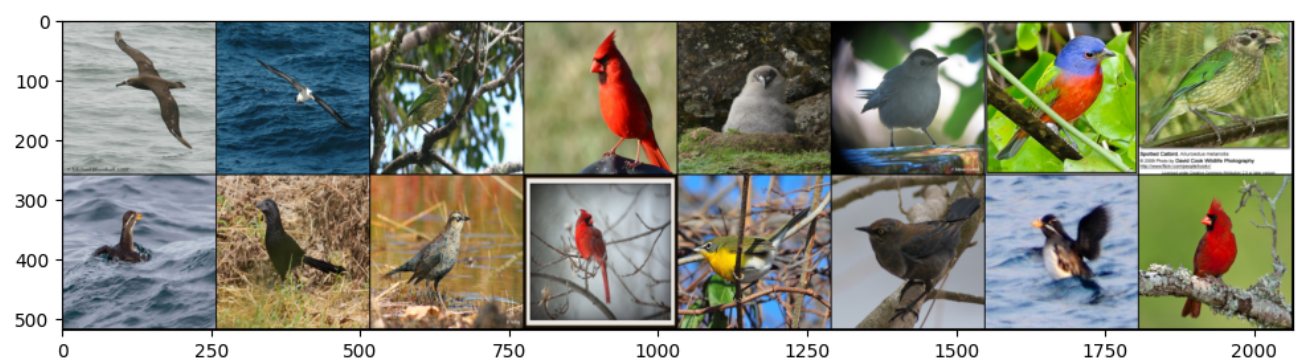
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

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]
```

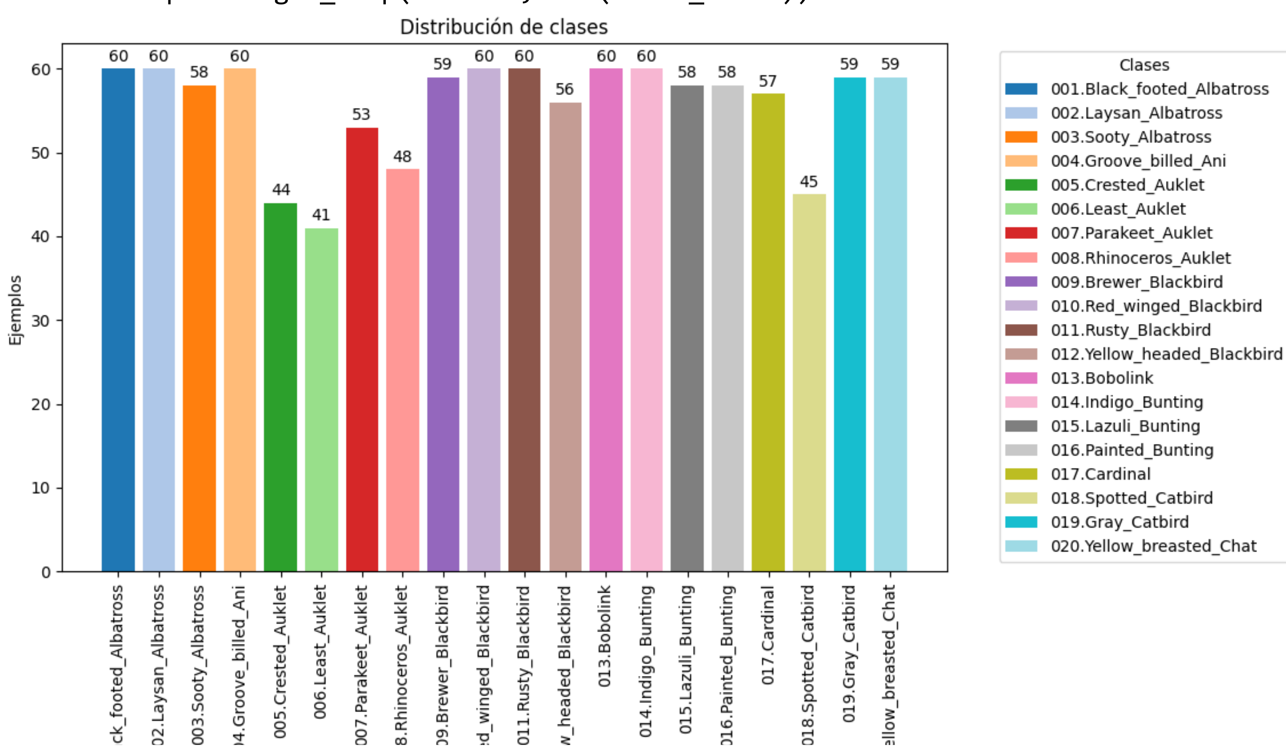
```
exp_transforms = transforms.Compose([
    transforms.Resize((256,256)),
    transforms.ToTensor()
])

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)))
```

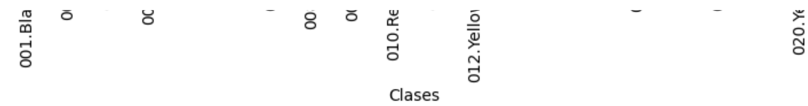



```
001.Black_footed_Albatross 002.Laysan_Albatross 018.Spotted_Catbird 017.Cardinal 003.
```

```
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='bot

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

plt.show()
```

```
<ipython-input-104-13fb55626ea7>: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)
full_dataset = ImageFolder(root=DATA_x200)
classes = full_dataset.classes

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

```
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=128, shuffle=True, num_workers=2)
testloader = DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)
```

```
Tamaño del conjunto de entreno: 9430
Tamaño del conjunto de validación: 2358
```

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)
```

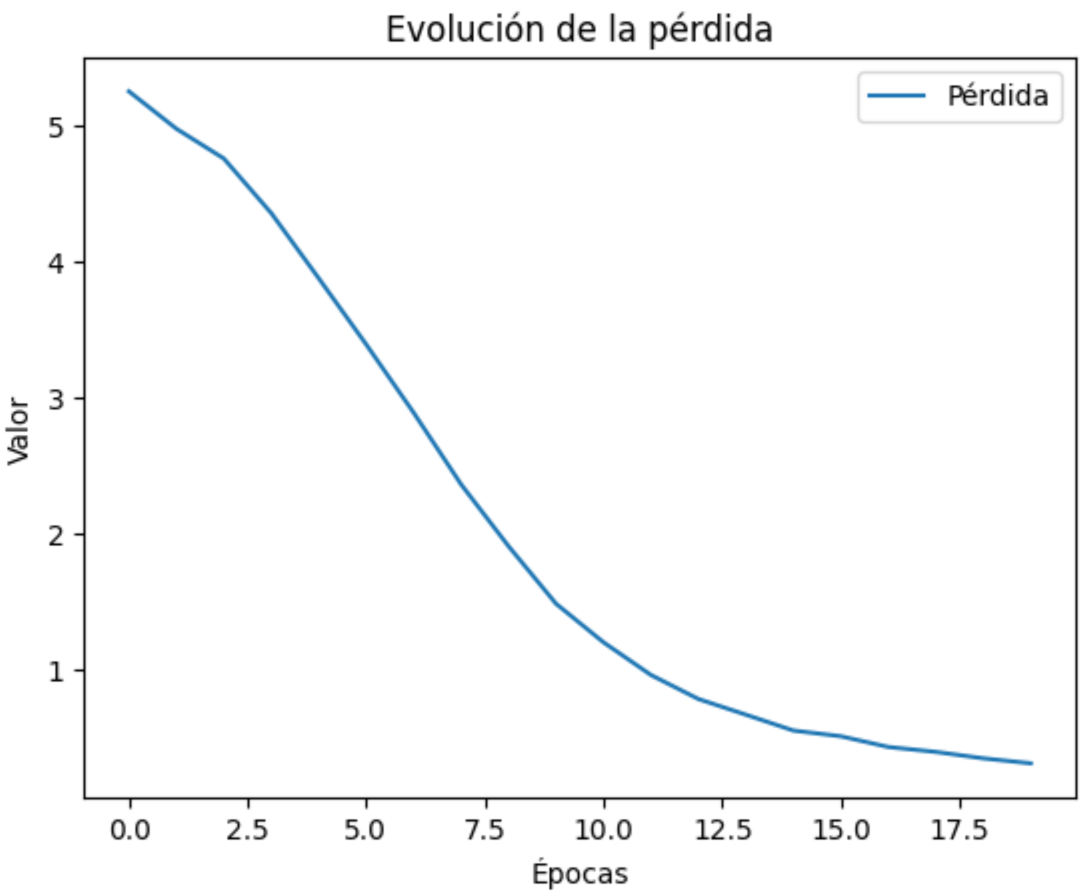
```python
        x = self.fc3(x)
        return x

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 d
            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_l


        ### 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 predicc
                total += labels.size(0)
```

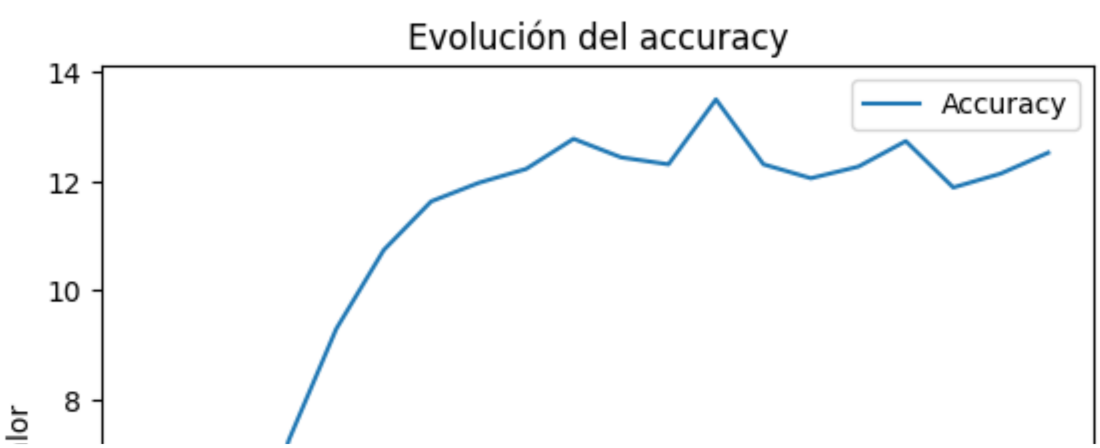
```
                correct += (predicted == labels).sum().item()       # predicciones correc

                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
```

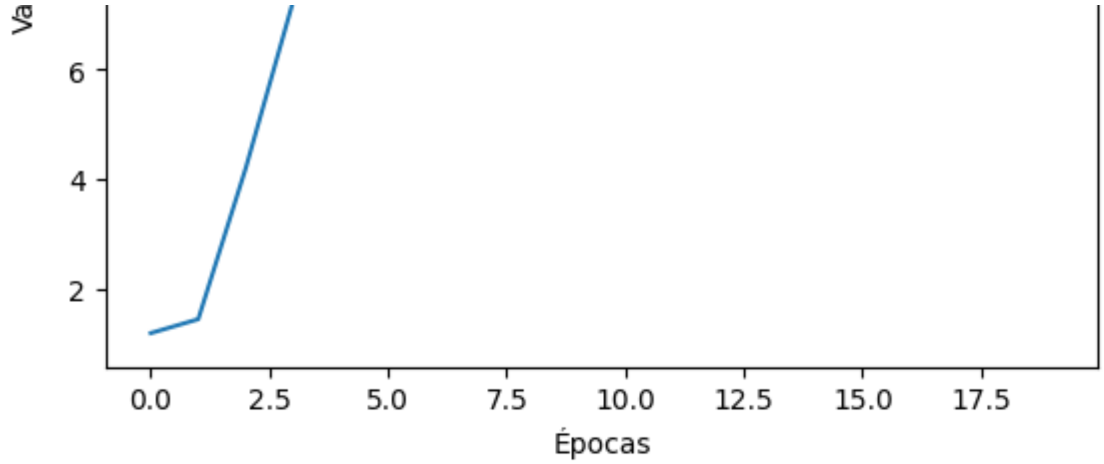
```
Epoch 1/20, Loss: 5.249: 100%|██████████| 74/74 [01:25<00:00,  1.16s/it]
  --> Epoch 1 accuracy: 1.19%: 100%|██████████| 19/19 [06:57<00:00, 21.97s/it]
Epoch 2/20, Loss: 4.975: 100%|██████████| 74/74 [01:28<00:00,  1.19s/it]
  --> Epoch 2 accuracy: 1.44%: 100%|██████████| 19/19 [00:19<00:00,  1.00s/it]
Epoch 3/20, Loss: 4.756: 100%|██████████| 74/74 [01:26<00:00,  1.18s/it]
  --> Epoch 3 accuracy: 4.20%: 100%|██████████| 19/19 [00:19<00:00,  1.01s/it]
Epoch 4/20, Loss: 4.354: 100%|██████████| 74/74 [01:22<00:00,  1.12s/it]
  --> Epoch 4 accuracy: 7.25%: 100%|██████████| 19/19 [00:22<00:00,  1.18s/it]
Epoch 5/20, Loss: 3.876: 100%|██████████| 74/74 [01:29<00:00,  1.21s/it]
  --> Epoch 5 accuracy: 9.29%: 100%|██████████| 19/19 [00:21<00:00,  1.11s/it]
Epoch 6/20, Loss: 3.390: 100%|██████████| 74/74 [01:28<00:00,  1.20s/it]
  --> Epoch 6 accuracy: 10.73%: 100%|██████████| 19/19 [00:20<00:00,  1.08s/it]
Epoch 7/20, Loss: 2.888: 100%|██████████| 74/74 [01:27<00:00,  1.18s/it]
  --> Epoch 7 accuracy: 11.62%: 100%|██████████| 19/19 [00:22<00:00,  1.21s/it]
Epoch 8/20, Loss: 2.361: 100%|██████████| 74/74 [01:24<00:00,  1.14s/it]
  --> Epoch 8 accuracy: 11.96%: 100%|██████████| 19/19 [00:21<00:00,  1.13s/it]
Epoch 9/20, Loss: 1.906: 100%|██████████| 74/74 [01:26<00:00,  1.17s/it]
  --> Epoch 9 accuracy: 12.21%: 100%|██████████| 19/19 [00:20<00:00,  1.06s/it]
Epoch 10/20, Loss: 1.484: 100%|██████████| 74/74 [01:27<00:00,  1.18s/it]
  --> Epoch 10 accuracy: 12.77%: 100%|██████████| 19/19 [00:19<00:00,  1.02s/it]
Epoch 11/20, Loss: 1.200: 100%|██████████| 74/74 [01:25<00:00,  1.16s/it]
  --> Epoch 11 accuracy: 12.43%: 100%|██████████| 19/19 [00:19<00:00,  1.03s/it]
Epoch 12/20, Loss: 0.960: 100%|██████████| 74/74 [01:26<00:00,  1.17s/it]
  --> Epoch 12 accuracy: 12.30%: 100%|██████████| 19/19 [00:20<00:00,  1.07s/it]
Epoch 13/20, Loss: 0.784: 100%|██████████| 74/74 [01:25<00:00,  1.15s/it]
  --> Epoch 13 accuracy: 13.49%: 100%|██████████| 19/19 [00:20<00:00,  1.08s/it]
Epoch 14/20, Loss: 0.668: 100%|██████████| 74/74 [01:25<00:00,  1.15s/it]
  --> Epoch 14 accuracy: 12.30%: 100%|██████████| 19/19 [00:20<00:00,  1.08s/it]
Epoch 15/20, Loss: 0.552: 100%|██████████| 74/74 [01:26<00:00,  1.17s/it]
  --> Epoch 15 accuracy: 12.04%: 100%|██████████| 19/19 [00:19<00:00,  1.04s/it]
Epoch 16/20, Loss: 0.510: 100%|██████████| 74/74 [01:26<00:00,  1.17s/it]
  --> Epoch 16 accuracy: 12.26%: 100%|██████████| 19/19 [00:20<00:00,  1.07s/it]
Epoch 17/20, Loss: 0.431: 100%|██████████| 74/74 [01:23<00:00,  1.13s/it]
  --> Epoch 17 accuracy: 12.72%: 100%|██████████| 19/19 [00:24<00:00,  1.27s/it]
Epoch 18/20, Loss: 0.395: 100%|██████████| 74/74 [01:24<00:00,  1.14s/it]
  --> Epoch 18 accuracy: 11.87%: 100%|██████████| 19/19 [00:20<00:00,  1.07s/it]
Epoch 19/20, Loss: 0.348: 100%|██████████| 74/74 [01:27<00:00,  1.18s/it]
  --> Epoch 19 accuracy: 12.13%: 100%|██████████| 19/19 [00:19<00:00,  1.05s/it]
Epoch 20/20, Loss: 0.311: 100%|██████████| 74/74 [01:32<00:00,  1.25s/it]
  --> Epoch 20 accuracy: 12.51%: 100%|██████████| 19/19 [00:20<00:00,  1.09s/it]
```

```
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:.2f

    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: 12.51%
Accuracy de: 001.Black_footed_Albatross --> 10.00%
Accuracy de: 002.Laysan_Albatross --> 14.29%
Accuracy de: 003.Sooty_Albatross --> 6.67%
Accuracy de: 004.Groove_billed_Ani --> 0.00%
Accuracy de: 005.Crested_Auklet --> 20.00%
Accuracy de: 006.Least_Auklet --> 0.00%
Accuracy de: 007.Parakeet_Auklet --> 18.75%
Accuracy de: 008.Rhinoceros_Auklet --> 20.00%
Accuracy de: 009.Brewer_Blackbird --> 0.00%
Accuracy de: 010.Red_winged_Blackbird --> 22.22%
Accuracy de: 011.Rusty_Blackbird --> 0.00%
Accuracy de: 012.Yellow_headed_Blackbird --> 28.57%
Accuracy de: 013.Bobolink --> 33.33%
Accuracy de: 014.Indigo_Bunting --> 10.00%
Accuracy de: 015.Lazuli_Bunting --> 0.00%
Accuracy de: 016.Painted_Bunting --> 16.67%
Accuracy de: 017.Cardinal --> 61.54%
Accuracy de: 018.Spotted_Catbird --> 40.00%
Accuracy de: 019.Gray_Catbird --> 13.33%
Accuracy de: 020.Yellow_breasted_Chat --> 28.57%
Accuracy de: 021.Eastern_Towhee --> 7.69%
Accuracy de: 022.Chuck_will_Widow --> 20.00%
```

```
Accuracy de: 022.Chuck_will_Widow --> 20.00%
Accuracy de: 023.Brandt_Cormorant --> 14.29%
Accuracy de: 024.Red_faced_Cormorant --> 20.00%
Accuracy de: 025.Pelagic_Cormorant --> 14.29%
Accuracy de: 026.Bronzed_Cowbird --> 0.00%
Accuracy de: 027.Shiny_Cowbird --> 0.00%
Accuracy de: 028.Brown_Creeper --> 15.38%
Accuracy de: 029.American_Crow --> 9.09%
Accuracy de: 030.Fish_Crow --> 0.00%
Accuracy de: 031.Black_billed_Cuckoo --> 14.29%
Accuracy de: 032.Mangrove_Cuckoo --> 0.00%
Accuracy de: 033.Yellow_billed_Cuckoo --> 33.33%
Accuracy de: 034.Gray_crowned_Rosy_Finch --> 18.18%
Accuracy de: 035.Purple_Finch --> 15.38%
Accuracy de: 036.Northern_Flicker --> 0.00%
Accuracy de: 037.Acadian_Flycatcher --> 30.00%
Accuracy de: 038.Great_Crested_Flycatcher --> 6.25%
Accuracy de: 039.Least_Flycatcher --> 0.00%
Accuracy de: 040.Olive_sided_Flycatcher --> 8.33%
Accuracy de: 041.Scissor_tailed_Flycatcher --> 0.00%
Accuracy de: 042.Vermilion_Flycatcher --> 40.00%
Accuracy de: 043.Yellow_bellied_Flycatcher --> 8.33%
Accuracy de: 044.Frigatebird --> 16.67%
Accuracy de: 045.Northern_Fulmar --> 10.00%
Accuracy de: 046.Gadwall --> 0.00%
Accuracy de: 047.American_Goldfinch --> 0.00%
Accuracy de: 048.European_Goldfinch --> 11.11%
Accuracy de: 049.Boat_tailed_Grackle --> 14.29%
Accuracy de: 050.Eared_Grebe --> 12.50%
Accuracy de: 051.Horned_Grebe --> 35.71%
Accuracy de: 052.Pied_billed_Grebe --> 17.65%
Accuracy de: 053.Western_Grebe --> 20.00%
Accuracy de: 054.Blue_Grosbeak --> 27.27%
Accuracy de: 055.Evening_Grosbeak --> 14.29%
Accuracy de: 056.Pine_Grosbeak --> 0.00%
Accuracy de: 057.Rose_breasted_Grosbeak --> 50.00%
```