UNIVERSIDAD DE GRANADA

PRÁCTICA 2

Deep Learning para Clasificación

Sistemas Inteligentes para la Gestión en la Empresa

Autor

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

ÍNDICE

Introducción

En esta segunda sesión práctica de la asignatura, exploraremos el desarrollo de un modelo de clasificación mediante el empleo de redes neuronales profundas. Nos enfocaremos en el entrenamiento de una red convolucional, donde aplicaremos diversas configuraciones y técnicas para optimizar su rendimiento. Utilizaremos una variante del conjunto de imágenes CUB (enlace al dataset original), el cual ha sido subdividido en dos conjuntos más reducidos: x_20 y x_200.

Para llevar a cabo este proceso, implementaremos la práctica en Python, aprovechando la potencia de la biblioteca PyTorch. Esta librería está especialmente diseñada para la clasificación de imágenes, lo que nos permitirá explorar de manera eficiente y efectiva las capacidades de las redes neuronales en este contexto.

Tareas realizadas

En esta sección, detallaremos las actividades llevadas a cabo durante el desarrollo del proyecto, destacando los hitos alcanzados, los métodos empleados. Los resultados obtenidos serán comentados en la sección correspondiente.

Exploración de datos

Durante la fase de exploración de datos hemos llevado a cabo varias tareas para comprender mejor la naturaleza del conjunto de datos. En primer lugar, se determinaron las dimensiones medias de las imágenes (**500x375**). Posteriormente, se llevó a cabo la visualización de una muestra representativa de las imágenes disponibles aplicando una redimensión a 256x256 (ver figura 1).

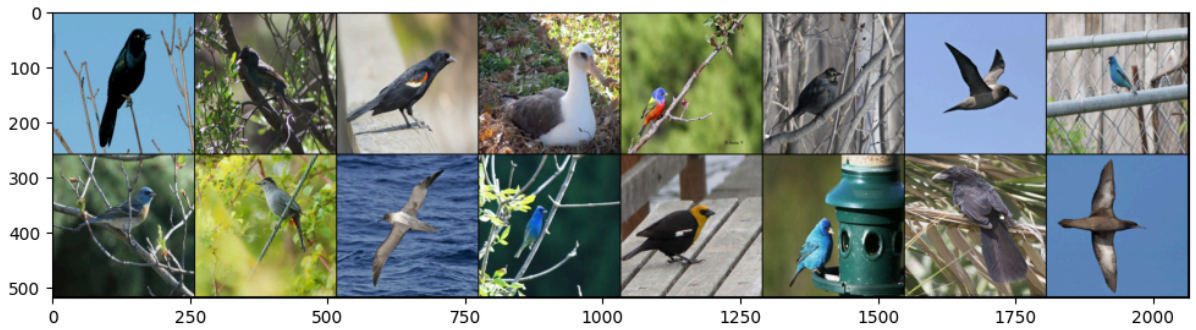

Figura 1. Muestra del conjunto

Como parte final de la exploración, hemos elaborado una gráfica de barras en la que se puede observar la distribución de las clases del conjunto. Como se puede ver en la figura siguiente, la distribución va desde **41** ejemplos para la clase menos representativa hasta los **60** ejemplos para las clases más representadas. A priori, el número de imágenes puede considerarse pequeño teniendo en cuenta que hay que dividir el conjunto para entrenar y validar.

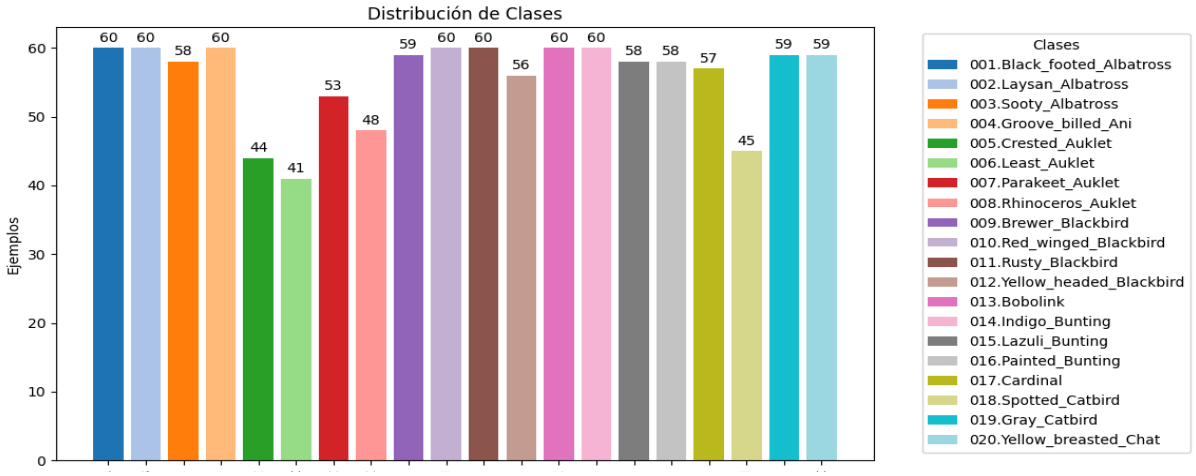

Figura 2. Distribución de clases.

Particionamiento de datos

Como hemos mencionado anteriormente, es necesario subdividir el conjunto de imágenes en dos conjuntos que usaremos para entrenar y evaluar el rendimiento del modelo.

Empezaremos definiendo las transformaciones a usar para cada conjunto. Para el conjunto de **entrenamiento** (ver figura 3):

- Aplicación de un redimensionado **256x256**. Como se puede ver en la figura 1, el objetivo de clasificación suele estar en el centro, y este redimensionado no lo distorsiona en absoluto.
- Aplicación de **volteos horizontales aleatorios** en 50%.
- Aplicación de **volteos verticales aleatorios** en 50%.

Y para el conjunto de **evaluación** solo aplicaremos el redimensionado **256x256**.

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

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

# transformaciones para el dataset de validacion
test_transforms = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(*imagenet_stats)
])
```

Figura 3. Transformaciones para los conjuntos de entrenamiento y validación.

Después de definir las transformaciones, procedemos a dividir el conjunto de datos en un **80% para entrenamiento** y un **20% para pruebas** (ver figura 4). En la figura 5 se puede ver el tamaño resultante de esta división para ambos conjuntos.

```python
# Dividimos el dataset en train y test
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])
```

Figura 4. Partición de los conjuntos de entrenamiento y validación.

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

Figura 5. Tamaño de los conjuntos de entrenamiento y validación.

Clasificación

Se implementó una red neuronal convolucional con las siguientes características:

- Cuatro capas convolucionales, cada una seguida de una función de activación, con tamaños de **salida** de **32**, **64**, **128** y **256** respectivamente. Para cada capa, se utilizó un **kernel** de tamaño **3x3** y un **padding** de **1** para mantener el tamaño de las características.
- Una capa de **pooling** de **2x2** para reducir las dimensiones de las características.
- Dos capas totalmente conectadas, con la primera capa con **512** neuronas y la segunda capa con un número de neuronas correspondiente a las clases del problema (en este caso **20**).

La figura siguiente muestra la configuración completa de la red.

```python
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)

        # Capas 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 para evitar sobreajuste
        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()
```

Figura 6. Configuración de la red neuronal convolucional.

Además se utilizó la función de pérdida **CrossEntropyLoss**. Es comúnmente utilizada en problemas de clasificación multiclase, ya que compara la distribución de probabilidad predicha por el modelo con la distribución real de las etiquetas, buscando minimizar la divergencia entre ellas.

Como función de optimización usaremos **Adam Optimizer**. Este algoritmo de optimización combina las ventajas de los métodos de descenso de gradiente estocástico con momentum y adaptación de la tasa de aprendizaje, lo que lo hace eficiente y efectivo en la optimización de redes neuronales.

```
# Definimos la función de pérdida y el optimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Figura 7. Función de pérdida y optimizador.

Se incluyeron múltiples imágenes (Figuras 8 y 9) que muestran el código de la función de entrenamiento. Esta función no solo se encarga de entrenar el modelo, sino que también realiza la validación del mismo en cada época.

```
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 esta disponible
        optimizer.zero_grad()               # resetea los gradientes
        outputs = model(inputs)             # calcula las salidas para las entradas del lote
        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_loss / (i + 1):.3f}')
```

Figura 8. Entrenamiento del modelo.

```
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 predicciones del modelo
        total += labels.size(0)
        correct += (predicted == labels).sum().item()   # predicciones correctas

        tqdm_val.set_description(f'  --> Epoch {epoch + 1} accuracy: {(100 * correct / total):.2f}%')
```

Figura 9. Validación del modelo.

Ajuste de hiperparametros

Hemos realizado algunos ajustes en los hiperparámetros del modelo, incluyendo funciones de pérdida, optimizadores y la arquitectura de la red neuronal. Estos cambios se llevaron a cabo con el objetivo de mejorar el rendimiento y la precisión del modelo en la tarea de clasificación.

Ajuste 1

En este ajuste hemos tocado el parámetro "**weight decay**", o decaimiento de peso, del optimizador Adam. Cuando se aplica, se añade un término adicional a la actualización de los pesos durante el entrenamiento. Este término es proporcional al propio peso y a una constante de decaimiento, y tiene el efecto de reducir gradualmente el valor de los pesos durante el entrenamiento. Esto ayuda a prevenir el sobreajuste, ya que penaliza los modelos que tienen pesos demasiado grandes.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)
```

Figura 10. Ajuste de hiperparámetros 1.

Ver resultados.

Ajuste 2

Para el segundo ajuste, hemos utilizado el optimizador **SGD**, que funciona actualizando iterativamente los pesos de la red en dirección al gradiente negativo de la función de pérdida.

En él probaremos los parámetros **lr**, para controlar la tasa de aprendizaje; y **momentum**, controla la influencia de la historia pasada de los gradientes en la actualización de los pesos

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Figura 11. Ajuste de hiperparámetros 2.

Ver resultados.

Ajuste 3

En el ajuste 3, hemos aumentado el tamaño del **kernel** a **6x6** de todas las capas convolucionales. De esta manera forzamos a que aprenda características espaciales más complejas, pero como contraparte, podemos aumentar la carga computacional y el sobreajuste.

Como función de pérdida y optimizador se usarán los originales (ver figura 7)

```
# Capas convolucionales
self.conv1 = nn.Conv2d(3, 32, kernel_size=6, padding=1)
self.conv2 = nn.Conv2d(32, 64, kernel_size=6, padding=1)
self.conv3 = nn.Conv2d(64, 128, kernel_size=6, padding=1)
self.conv4 = nn.Conv2d(128, 256, kernel_size=6, padding=1)
```

Figura 12. Ajuste de hiperparámetros 3.

Ver resultados.

Ajuste 4

En el último ajuste, se han reducido las **capas convolucionales** a **dos**, manteniendo la configuración original en los kernels. Así, obtendremos un modelo más simple y eficiente, pero también limitaremos la capacidad del modelo para capturar la complejidad de los datos, lo que puede resultar en un peor rendimiento.

Este ajuste también mantiene la función de pérdida y optimizador originales (ver figura 7)

```
# Capas convolucionales
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
```

Figura 13. Ajuste de hiperparámetros 4.

Ver resultados.

Ampliación a 200 clases

Se ha llevado a cabo una ampliación del problema al conjunto de 200 clases. Dado que no se cuenta con una tarjeta Nvidia para el entrenamiento, se optó por ejecutar el proceso en la plataforma **Google Colab**, donde se pueden utilizar recursos de hardware más potentes sin coste adicional [1].

En cuanto a la configuración de la red neuronal, las funciones de pérdida y el optimizador, se utilizaron los mismos parámetros que en la **configuración original** (ver figuras 6 y 7). La única modificación realizada fue el ajuste del tamaño de los **lotes** (batch size) a **128**, lo que influyó en la velocidad de entrenamiento en la plataforma citada anteriormente.

Ver resultados.

Resultados

Resultados ajuste estándar

Épocas: 40
Tiempo de ejecución: 40 minutos
Dispositivo: CPU

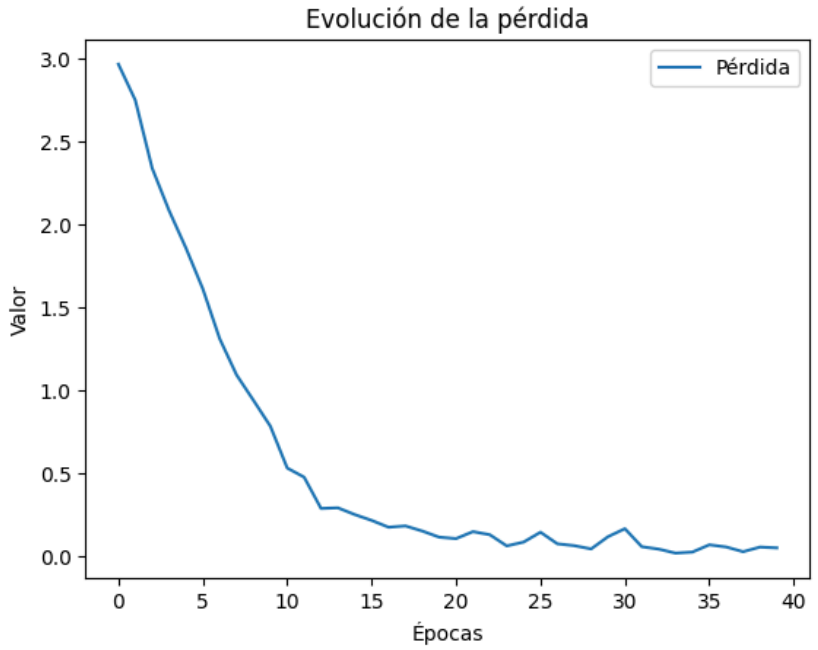

Figura 14. Gráfica de la evolución de la pérdida sobre el ajuste estándar.

En base a la información proporcionada por la gráfica anterior, parece que el modelo alcanza su rendimiento óptimo en términos de pérdida alrededor de la época 20 de 40. Después de esta época, la pérdida se estabiliza y no muestra una mejora significativa ni un deterioro notable, lo que sugiere que el modelo ha convergido a un mínimo local.

En siguientes pruebas se tomará como límite las 20 épocas.

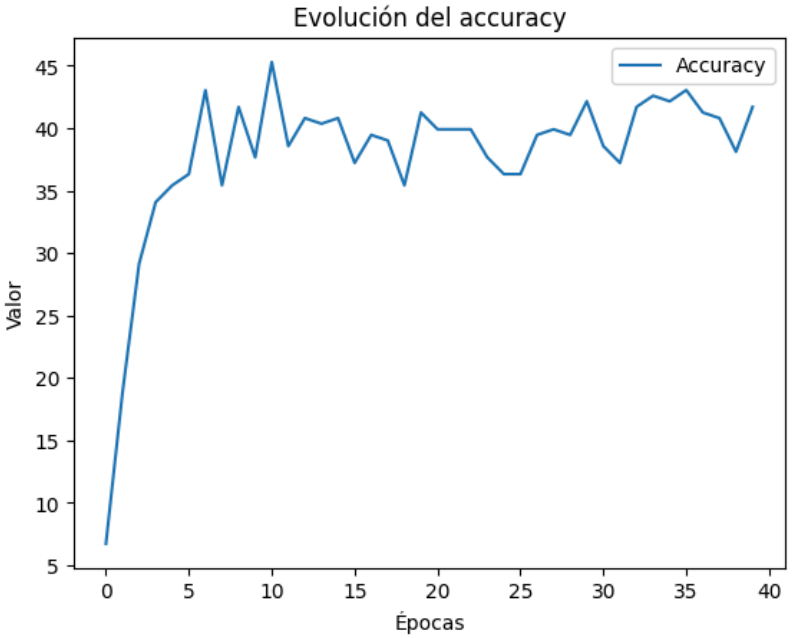

Figura 15. Gráfica de la evolución del accuracy sobre el ajuste estándar.

En cuanto al accuracy, parece que alcanza su máximo alrededor de las épocas 10-12 y luego se estabiliza o incluso disminuye ligeramente, pero sin mostrar una mejora clara hasta el final de las épocas. Este comportamiento puede indicar que el modelo ha aprendido correctamente los patrones en los datos de entrenamiento, pero tiene dificultades para generalizar a datos no vistos o está alcanzando un límite en su capacidad de aprendizaje.

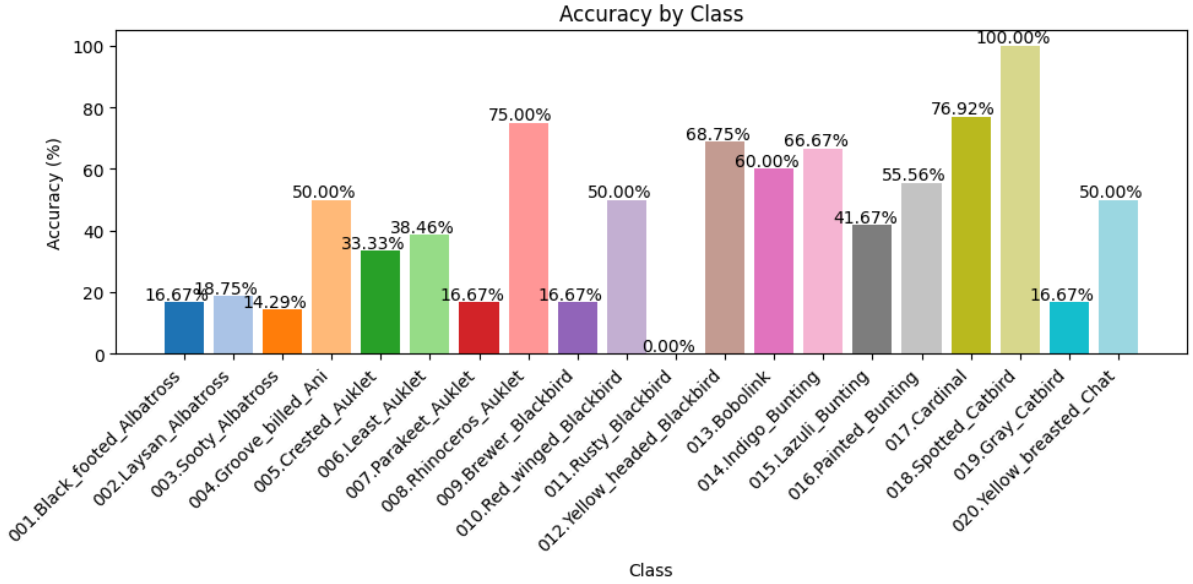

Figura 16. Gráfica de la distribución del acierto en las clases sobre el ajuste estándar.

Resultados ajuste 1

Épocas: 20
Tiempo de ejecución: 20 minutos
Dispositivo: CPU

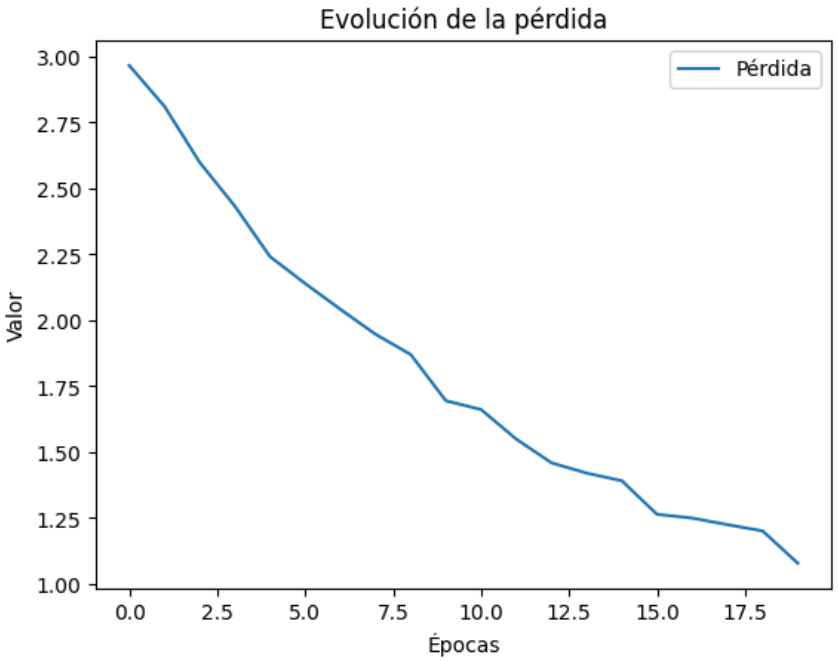

Figura 17. Gráfica de la evolución de la pérdida sobre el ajuste 1.

La gráfica de pérdida muestra una convergencia más lenta y menos efectiva en comparación con el ajuste anterior. El "weight decay" podría estar afectando negativamente la capacidad del modelo para aprender los patrones en los datos de entrenamiento, lo que resulta en una pérdida más alta durante el entrenamiento.

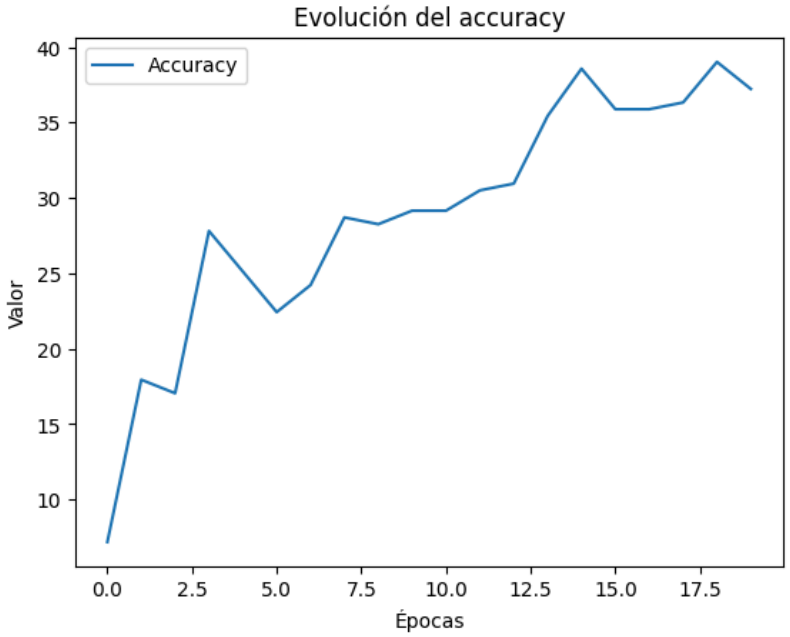

Figura 18. Gráfica de la evolución del accuracy sobre el ajuste 1.

Por otro lado, la gráfica de precisión también muestra un rendimiento inferior en comparación con el ajuste anterior. Aunque la precisión puede variar según la métrica y el problema específico, en este caso, es probable que el modelo esté teniendo dificultades para generalizar a datos no vistos debido a la convergencia menos efectiva de la pérdida durante el entrenamiento.

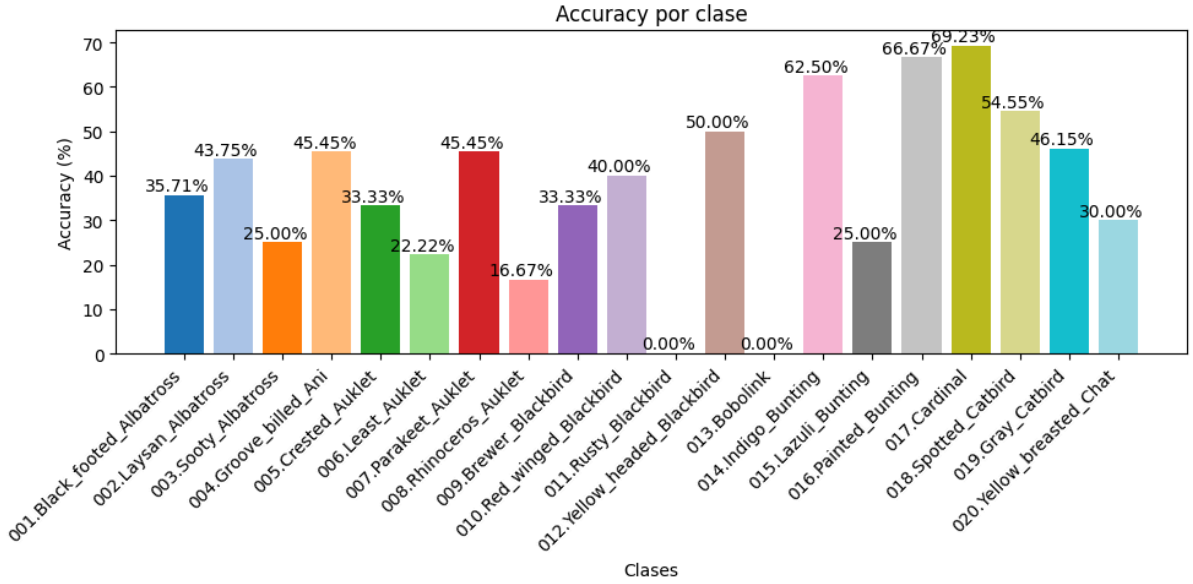

Figura 19. Gráfica de la distribución del acierto en las clases sobre el ajuste 1.

Resultados ajuste 2

Épocas: 20
Tiempo de ejecución: 18 minutos
Dispositivo: CPU

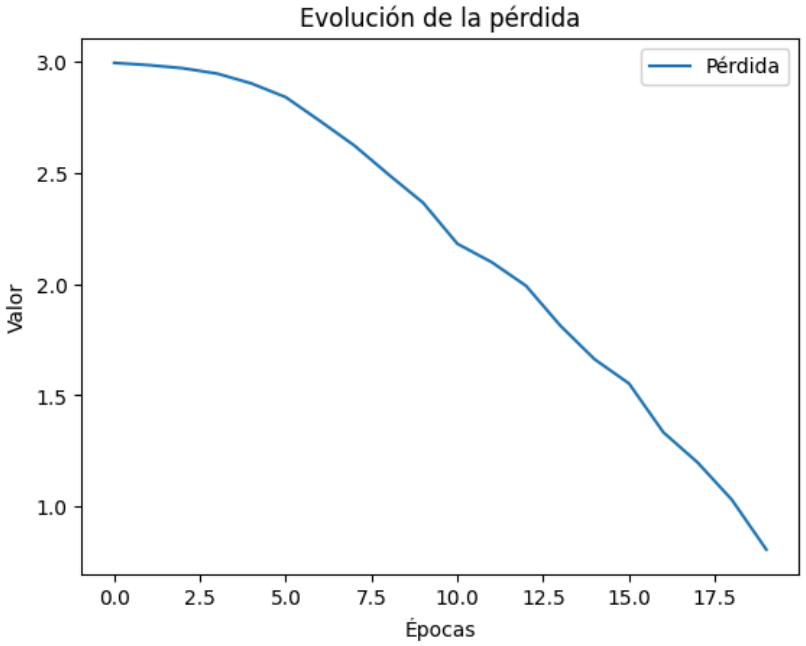

Figura 20. Gráfica de la evolución de la pérdida sobre el ajuste 2.

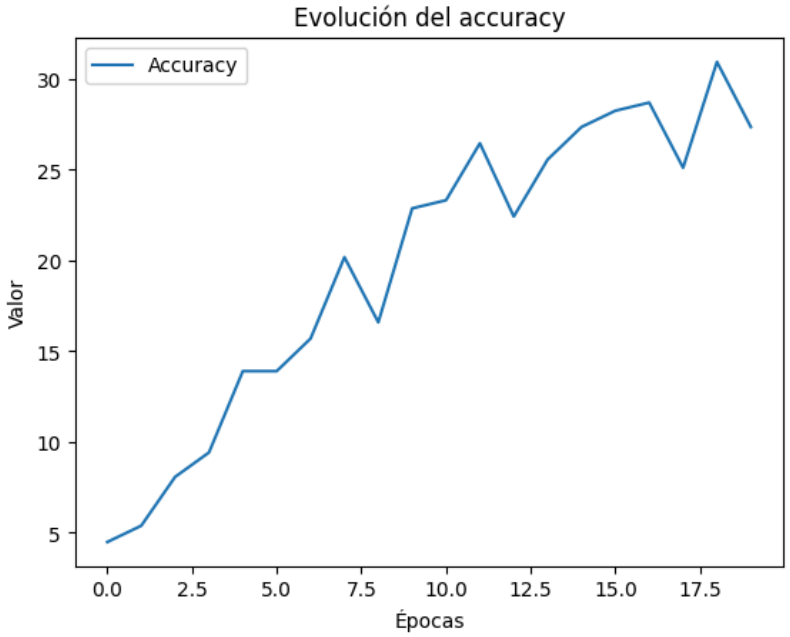

Figura 21. Gráfica de la evolución del accuracy sobre el ajuste 2.

Los resultados obtenidos muestran una evolución de la pérdida y la precisión similares a los ajustes anteriores (ajuste 1). La pérdida disminuye gradualmente de valores altos a valores bajos, lo que indica que el modelo está mejorando durante el entrenamiento, pero posiblemente a un ritmo más lento que en el ajuste original. La precisión también muestra un aumento modesto durante el entrenamiento, pero no alcanza el mismo nivel que en el ajuste original.

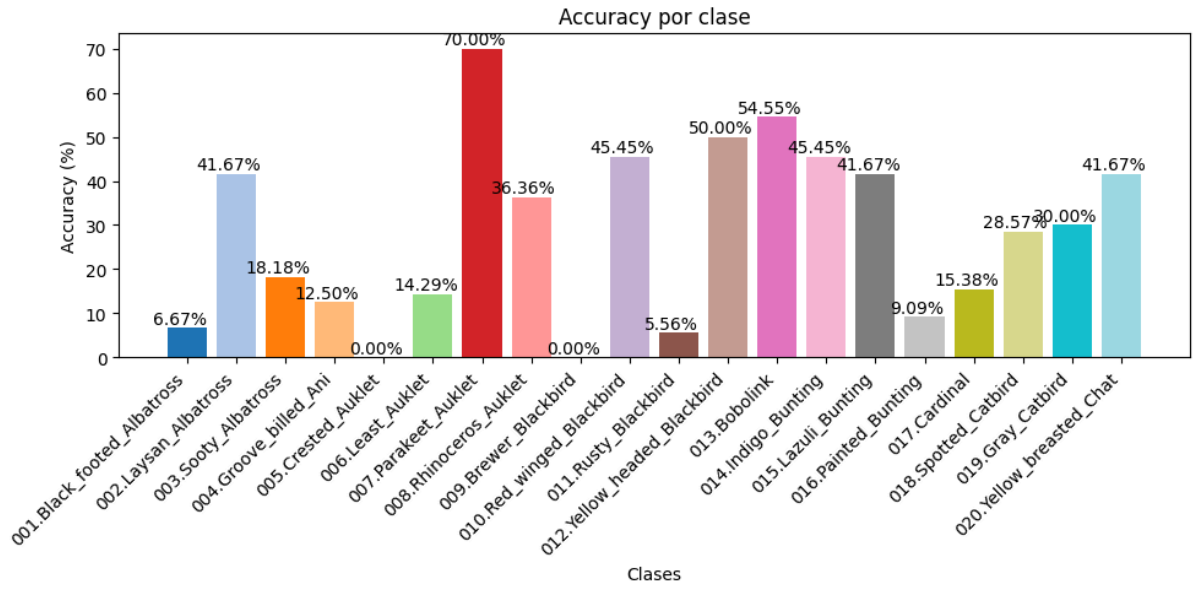

Figura 22. Gráfica de la distribución del acierto en las clases sobre el ajuste 2.

Resultados ajuste 3

Épocas: 20
Tiempo de ejecución: 29 minutos
Dispositivo: CPU

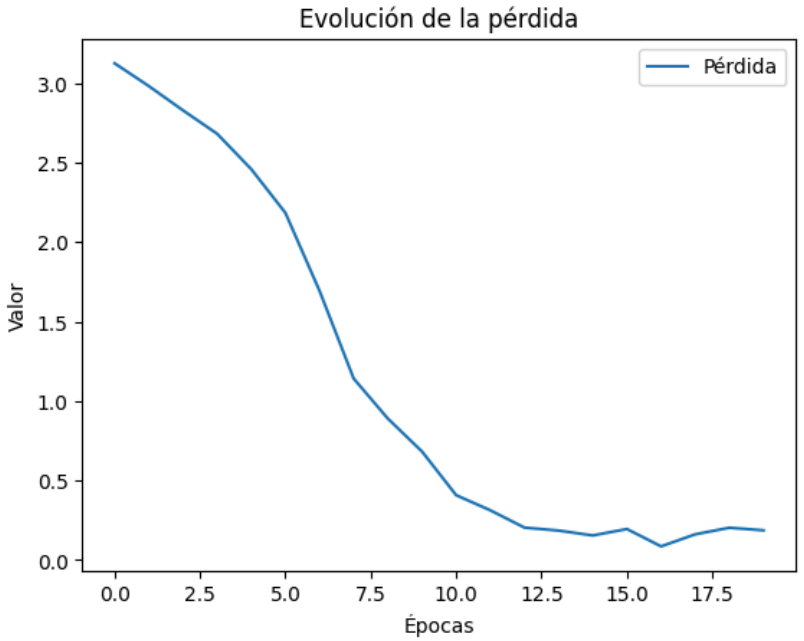

Figura 23. Gráfica de la evolución de la pérdida sobre el ajuste 3.

El aumento del tamaño del kernel a 6x6 en todas las capas convolucionales ha dado lugar a una mejora en comparación con los dos ajustes anteriores. La pérdida converge a valores bajos en torno a la época 10, lo que indica que el modelo está mejorando durante el entrenamiento y alcanzando una buena generalización en una cantidad relativamente baja de épocas.

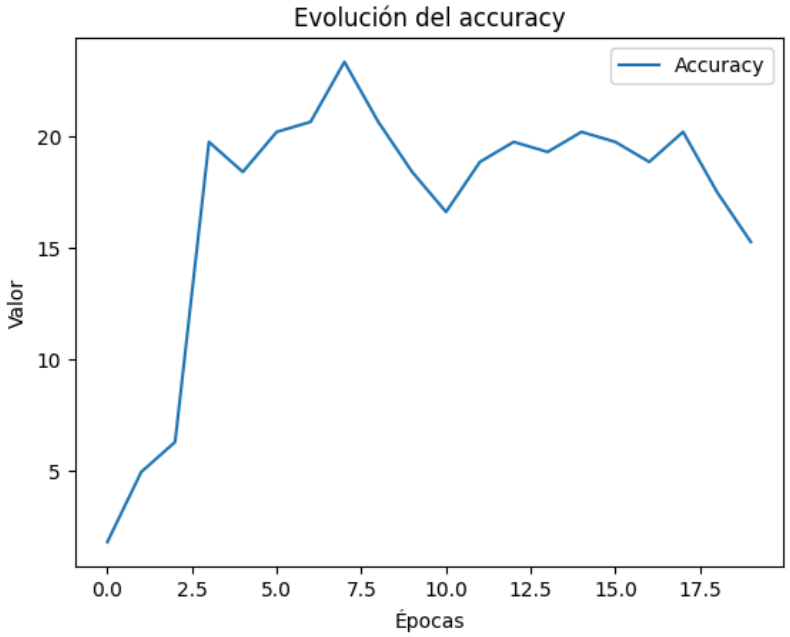

Figura 24. Gráfica de la evolución del accuracy sobre el ajuste 3.

El accuracy alcanza su máximo en la época 8, pero no llega al nivel del ajuste original. Después de esta época, no muestra una mejora significativa y a veces empeora, lo que sugiere que el modelo está teniendo dificultades para mejorar su precisión más allá de cierto punto.

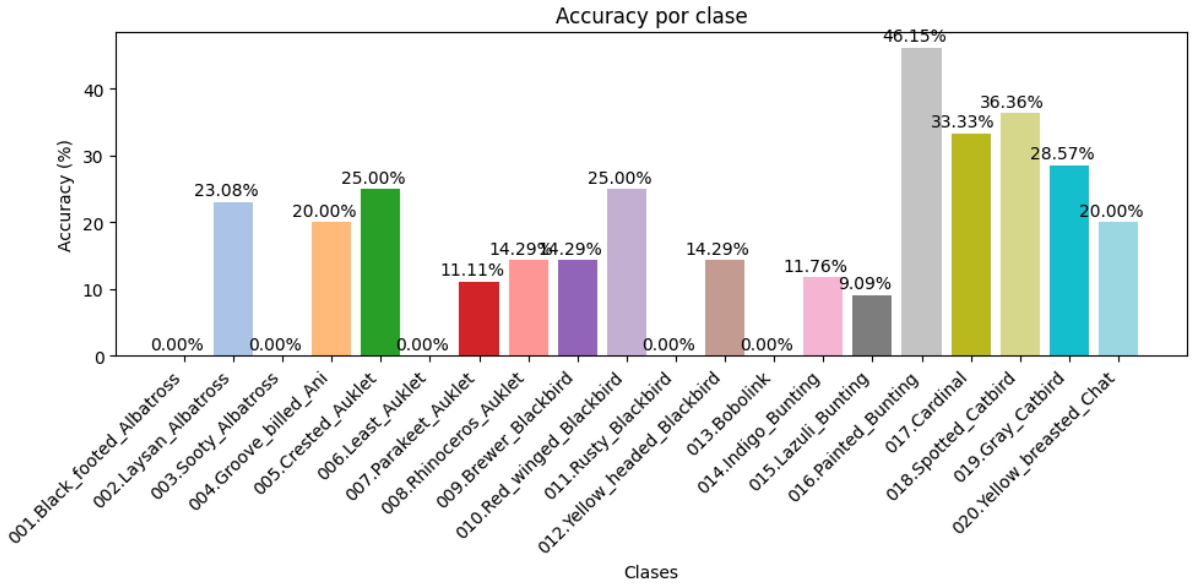

Figura 25. Gráfica de la distribución del acierto en las clases sobre el ajuste 3.

Resultados ajuste 4

Épocas: 20
Tiempo de ejecución: 40 minutos
Dispositivo: CPU

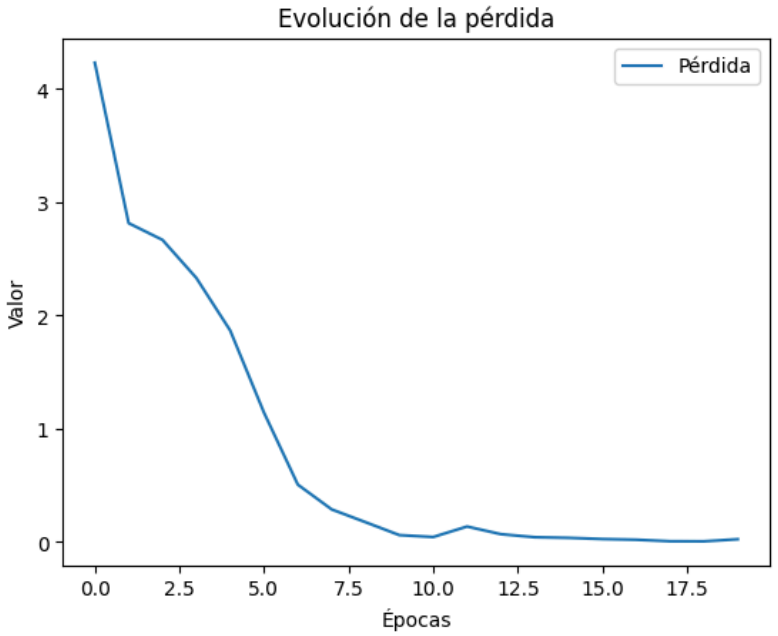

Figura 26. Gráfica de la evolución de la pérdida sobre el ajuste 4.

La reducción de las capas convolucionales a dos ha dado lugar a una convergencia más rápida de la pérdida, con valores cercanos a 0 alrededor de la época 8. Esto sugiere que el modelo está mejorando durante el entrenamiento y alcanzando una buena generalización en menos épocas en comparación con ajustes anteriores.

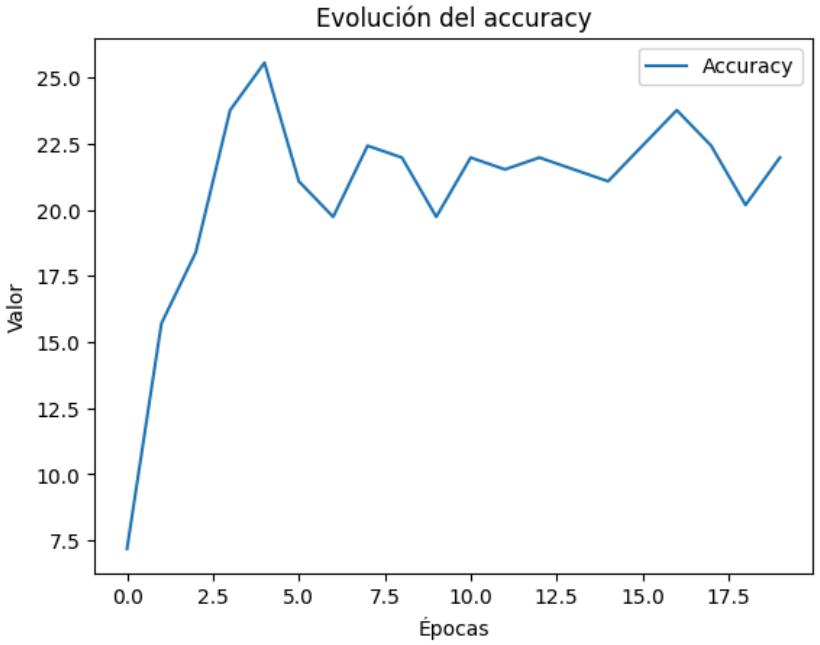

Figura 27. Gráfica de la evolución del accuracy sobre el ajuste 4.

Sin embargo, el accuracy muestra un rendimiento inferior en comparación con el modelo original. Aunque alcanza su máximo alrededor de la época 5, este valor es significativamente menor que el del modelo original. Además, el accuracy empeora después de la época 5, lo que sugiere que el modelo tiene dificultades para mejorar su precisión más allá de cierto punto.

En resumen, la reducción de las capas convolucionales a dos ha resultado en una mejora en la convergencia de la pérdida, pero a costa de un rendimiento inferior en términos de precisión. Esto sugiere que el modelo simplificado puede no ser capaz de capturar la complejidad de los datos de la misma manera que el modelo original.

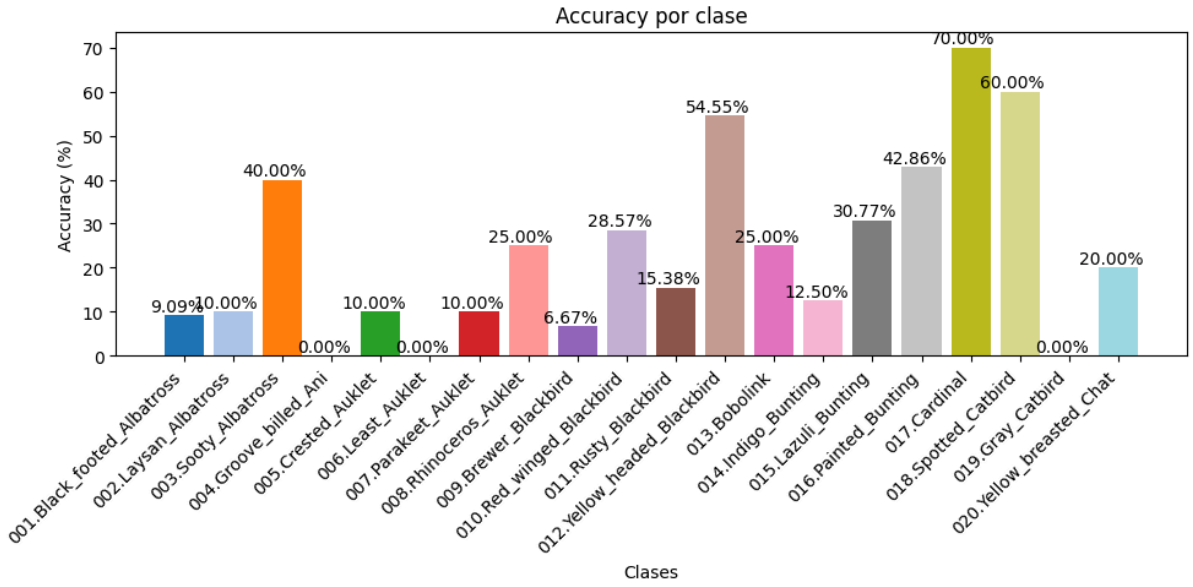

Figura 28. Gráfica de la distribución del acierto en las clases sobre el ajuste 4.

Resultados ampliación a 200 clases

Épocas: 20
Tiempo de ejecución: 45 minutos
Dispositivo: GPU (Colab)

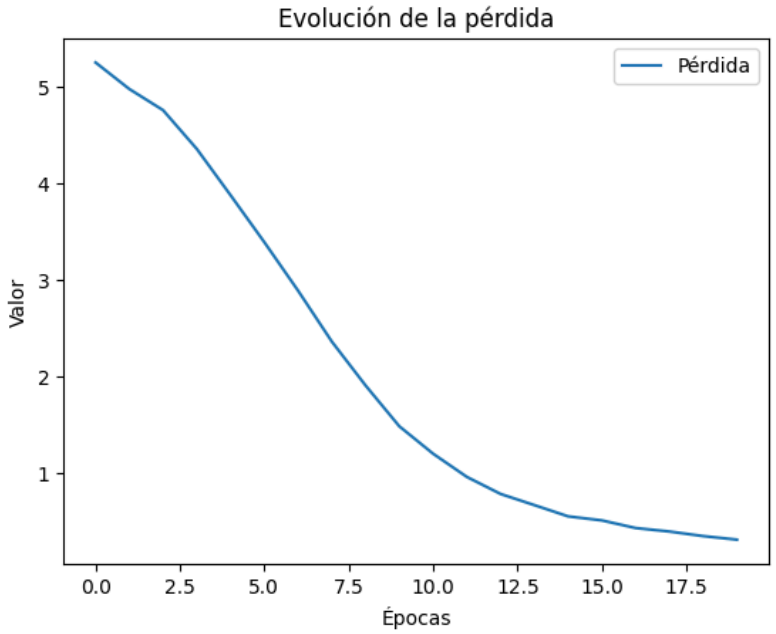

Figura 29. Gráfica de la evolución de la pérdida sobre el ajuste ampliado.

La pérdida converge de manera más suave y gradual en comparación con los ajustes anteriores, alcanzando un valor cercano a 0.3. Esta convergencia más suave puede indicar que el modelo está mejorando de manera más estable a lo largo del entrenamiento.

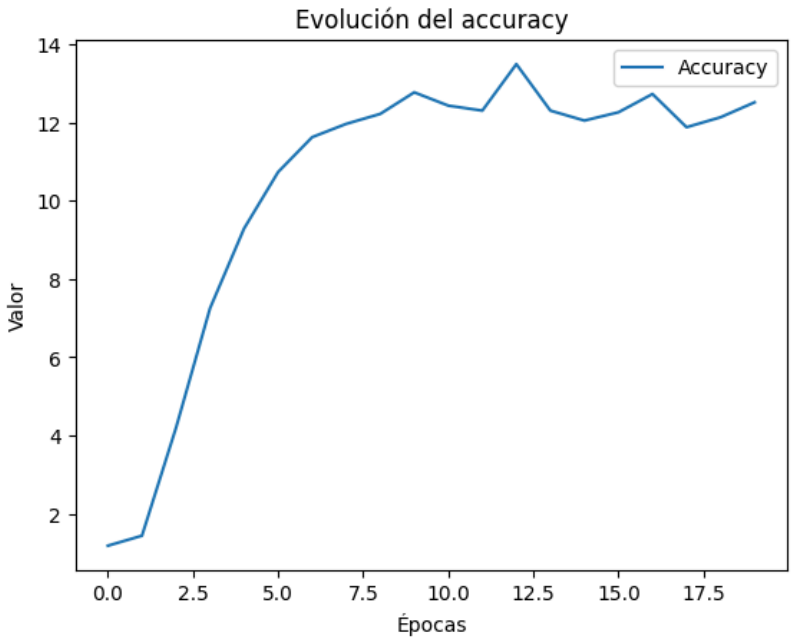

Figura 30. Gráfica de la evolución del accuracy sobre el ajuste ampliado.

Por otro lado, el accuracy muestra una mejora rápida durante las primeras épocas, estabilizandose alrededor de la época 6. Sin embargo, los valores de precisión son significativamente más bajos en comparación con el modelo original. Esto sugiere que el modelo tiene dificultades para aprender y generalizar los patrones en los datos de las 200 clases.

Para ver la distribución de clases acudir al enlace del fichero ejecutado en la plataforma Google Colab [1].

Conclusiones

Durante esta práctica, exploré el proceso de desarrollo de un modelo de clasificación utilizando redes neuronales profundas, centrándome en el uso de una red convolucional. A lo largo del proyecto, implementé diversas estrategias y técnicas con el objetivo de mejorar el rendimiento de la clasificación.

Durante la fase de clasificación, experimenté con diferentes arquitecturas de redes neuronales y opciones de optimización. Desde la configuración de capas convolucionales hasta la elección de funciones de pérdida y optimizadores, cada ajuste tuvo un impacto en el rendimiento del modelo. A través de este proceso, aprendí la importancia de la selección de hiperparámetros y la realización de ajustes iterativos para optimizar el rendimiento del modelo.

Además, amplié el alcance del problema al trabajar con un conjunto de datos más grande que contenía 200 clases. Esta ampliación presentó nuevos desafíos, como la gestión de recursos computacionales y la adaptación de estrategias de entrenamiento para manejar conjuntos de datos más grandes y complejos.

En resumen, esta práctica me proporcionó una comprensión del proceso de desarrollo de modelos de clasificación utilizando redes neuronales profundas.

Bibliografía

[1] Enlace a colab ejecutado, P2SIGE.ipynb

[2] Anexo 1 - p2sige_x20.pdf [Ejecución en Google Colab del conjunto x20]
[3] Anexo 2 - p2sige_x200.pdf [Ejecución en Google Colab del conjunto x200]