Respuestas sobre el ejemplo de clasificación de imágenes con PyTorch y MLP

1. Dataset y Preprocesamiento

- ¿Por qué es necesario redimensionar las imágenes a un tamaño fijo para una MLP?
  - Porque una MLP no puede procesar entradas de forma variable. Necesita vectores de entrada de longitud fija. Las imágenes deben tener la misma cantidad de píxeles para poder ser "aplanadas" y procesadas por las capas lineales.
- ¿Qué ventajas ofrece Albumentations frente a otras librerías de transformación como `torchvision.transforms`?
  - Albumentations es más rápido, flexible y permite transformaciones más avanzadas como `GridDistortion`, `ElasticTransform` o `Cutout`. Es especialmente eficiente para imágenes, soportando transformaciones simultáneas en imágenes y máscaras.

- ¿Qué hace `A.Normalize()`? ¿Por qué es importante antes de entrenar una red?
  - Normaliza las imágenes (los valores de los píxeles) restando la media y dividiendo por la desviación estándar. Esto mejora la convergencia, ayudando a que el entrenamiento sea más estable y rápido.
- ¿Por qué convertimos las imágenes a `ToTensorV2()` al final de la pipeline?
  - Porque convierte la imagen (HWC, np.array) a (CHW, tensor float32) compatible con PyTorch, que espera tensores como entrada. `ToTensorV2()` devuelve el tensor listo para la red.

```
transform = A.Compose([
        A.Resize(64, 64),
        A.Normalize(mean=(0.5,), std=(0.5,)),
        ToTensorV2()
])
```

2. Arquitectura del Modelo

- ¿Por qué usamos una red MLP en lugar de una CNN aquí? ¿Qué limitaciones tiene?
  - Se usa por simplicidad. Una MLP es más sencilla de implementar pero tiene la limitación de que no captura relaciones espaciales (patrones locales) como una CNN, por lo que suele tener peor desempeño en imágenes.
- ¿Qué hace la capa `Flatten()` al principio de la red?
  - Convierte el tensor de imágen 3D (canales, alto, ancho) en un vector 1D para poder pasarla por capas lineales (a diferencia de lo que ocurría con PyTorch y `ToTensorV2()`).
- ¿Qué función de activación se usó? ¿Por qué no usamos `Sigmoid` o `Tanh`?
  - Se usó `ReLU` porque es más eficiente, acelera el entrenamiento y evita el problema del gradiente desvanecido, común en `Sigmoid` y `Tanh`.
- ¿Qué parámetro del modelo deberíamos cambiar si aumentamos el tamaño de entrada de la imagen?
  - El número de `in_features` (parámetro `input_size`) de la primera capa `Linear`. Debe ajustarse a la nueva cantidad de píxeles.

```
class MLPClassifier(nn.Module):
        def __init__(self, input_size=N*N*3, num_classes=10):
                super().__init__()
                self.model = nn.Sequential(
```

```
                nn.Flatten(),
                nn.Linear(input_size, 512),
                nn.ReLU(),
                nn.Linear(512, 128),
                nn.ReLU(),
                nn.Linear(128, num_classes)
            )

    def forward(self, x):
            return self.model(x)
```

3. Entrenamiento y Optimización

- ¿Qué hace `optimizer.zero_grad()`?
  - Reinicia los gradientes acumulados antes de calcular los nuevos en cada batch y llamar a `loss.backward()`.
- ¿Por qué usamos `CrossEntropyLoss()` en este caso?
  - Porque es adecuada para clasificación multiclase con salidas softmax implícitas. Es la función estándar para estos casos, combinando softmax y log-loss.
- ¿Cómo afecta la elección del tamaño de batch (`batch_size`) al entrenamiento?
  - Tamaños grandes de batch pueden hacer el entrenamiento más rápido y estabilizan el gradiente, pero usan más memoria y pueden ser menos precisos. Tamaños pequeños agregan ruido pero pueden generalizar mejor.
- ¿Qué pasaría si no usamos `model.eval()` durante la validación?
  - Algunas capas (como `Dropout` o `BatchNorm`) se comportarían como en entrenamiento, dando resultados inconsistentes en validación.

4. Validación y Evaluación

- ¿Qué significa una accuracy del 70% en validación pero 90% en entrenamiento?
  - Que el modelo está sobreajustando (overfitting): aprende bien el entrenamiento porque el modelo lo memorizó pero no generaliza.
- ¿Qué otras métricas podrían ser más relevantes que accuracy en un problema real?
  - `precision`, `recall`, `f1-score`, AUC, especialmente en datasets desbalanceados. Va a depender del problema y el balance de clases.
- ¿Qué información útil nos da una matriz de confusión que no nos da la accuracy?

    - Muestra qué clases se confunden entre sí, no solo el total correcto, permitiendo detectar errores sistemáticos.
- En el reporte de clasificación, ¿qué representan `precision`, `recall` y `f1-score`?
    - `precision`: $\frac{TP}{TP+FP}$. Proporción de verdaderos positivos sobre los predichos positivos.
    - `recall`: $\frac{TP}{TP+FN}$. Proporción de verdaderos positivos sobre los reales positivos.
    - `f1-score`: promedio armonico entre `precision` y `recall`.

5. TensorBoard y Logging

- ¿Qué ventajas tiene usar TensorBoard durante el entrenamiento?
    - Permite monitorear métricas, visualizar imágenes, histogramas, comparar experimentos, etc. en tiempo real.
- ¿Qué diferencias hay entre loguear `add_scalar`, `add_image` y `add_text`?
    - `add_scalar`: valores numéricos (loss, accuracy).
    - `add_image`: visualización de imágenes (inputs, resultados).
    - `add_text`: notas, logs o texto (reportes, hiperparámetros).
- ¿Por qué es útil guardar visualmente las imágenes de validación en TensorBoard?
    - Permite inspeccionar visualmente cómo está funcionando el modelo y detectar errores o cambios visuales en predicciones.
- ¿Cómo se puede comparar el desempeño de distintos experimentos en TensorBoard?
    - Usar diferentes carpetas de logs (`log_dir`) y luego combinarlas en TensorBoard para comparar curvas y resultados.

6. Generalización y Transferencia

- ¿Qué cambios habría que hacer si quisiéramos aplicar este mismo modelo a un dataset con 100 clases?
    - Cambiar el parámetro `num_classes` en la última capa `Linear` del modelo.
- ¿Por qué una CNN suele ser más adecuada que una MLP para clasificación de imágenes?
    - Porque las CNN capturan patrones espaciales y locales, y son más eficientes, mientras que una MLP trata todos los píxeles como

independientes.

- ¿Qué problema podríamos tener si entrenamos este modelo con muy pocas imágenes por clase?
  - Riesgo de sobreajuste (overfitting) y mala generalización. Se podría usar data augmentation o preentrenamiento.

```
from collections import defaultdict
import numpy as np

def undersample_dataset(dataset, max_per_class=5):
        class_indices = defaultdict(list)
        for idx, label in enumerate(dataset.labels):
                class_indices[label].append(idx)
        selected_indices = []
        for indices in class_indices.values():
                np.random.shuffle(indices)
                selected_indices.extend(indices[:max_per_class])
        return torch.utils.data.Subset(dataset, selected_indices)

# Para pocas imágenes por clase:
train_dataset_small = undersample_dataset(train_dataset, max_per_class=5)
train_loader_small = DataLoader(train_dataset_small,
batch_size=batch_size, shuffle=True)
```

- ¿Cómo podríamos adaptar este pipeline para imágenes en escala de grises?
  - Cambiar la carga de imágenes a un solo canal modificando `in_channels` (ajustando el `input_size` del modelo) y adaptar la normalización:

```
# En CustomImageDataset.__getitem__:
image = np.array(Image.open(self.image_paths[idx]).convert("L"))  # "L"
para escala de grises

# En el modelo:
input_size = 64*64*1  # Cambio a 1 canal

# En las transformaciones (train, val):
A.Normalize(mean=(0.5,), std=(0.5,))
```

7. Regularización

Preguntas teóricas:

- ¿Qué es la regularización en el contexto del entrenamiento de redes neuronales?

    - Es un conjunto de técnicas para evitar el sobreajuste (overfitting) y mejorar la capacidad de generalización del modelo a datos no vistos.
- ¿Cuál es la diferencia entre `Dropout` y regularización `L2` (weight decay)?
    - `Dropout`: desactiva neuronas aleatoriamente durante el entrenamiento, forzando a la red a no depender de rutas específicas.
    - `L2`: penaliza pesos grandes agregando su norma cuadrada a la función de pérdida.
- ¿Qué es `BatchNorm` y cómo ayuda a estabilizar el entrenamiento?
    - BatchNorm normaliza las activaciones de cada batch, manteniendo la media y varianza estables, lo que acelera y estabiliza el entrenamiento.
- ¿Cómo se relaciona `BatchNorm` con la velocidad de convergencia?
    - `BatchNorm` permite usar tasas de aprendizaje (learning rates) más altas y reduce la sensibilidad a la inicialización, acelerando la convergencia.
- ¿Puede `BatchNorm` actuar como regularizador? ¿Por qué?
    - Sí, porque introduce ruido en el entrenamiento al depender de la estadística del batch cuando lo normaliza, ayudando a evitar el sobreajuste.
- ¿Qué efectos visuales podrías observar en TensorBoard si hay overfitting?
    - La loss de entrenamiento baja mucho, pero la de validación se estanca o sube. La accuracy de validación puede empeorar.
- ¿Cómo ayuda la regularización a mejorar la generalización del modelo?
    - Evita que el modelo se adapte demasiado a los datos de entrenamiento, forzándolo a aprender patrones más generales.

Actividades de modificación:

1. Agregar Dropout en la arquitectura MLP:

- Insertar capas `nn.Dropout(p=0.5)` entre las capas lineales y activaciones.
    - Comparar los resultados con y sin `Dropout`.

2. Agregar Batch Normalization:

    - Insertar `nn.BatchNorm1d(...)` después de cada capa `Linear` y antes de la activación:

```
self.net = nn.Sequential(
        nn.Flatten(),
        nn.Linear(in_features, 512),
        nn.BatchNorm1d(512),
```

```
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
    )
```

3. Aplicar Weight Decay (L2):
    - Modificar el optimizador:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,
weight_decay=1e-4)
```

4. Reducir overfitting con data augmentation:
    - Agregar transformaciones en Albumentations como `HorizontalFlip`, `BrightnessContrast`, `ShiftScaleRotate`.
5. Early Stopping (opcional):
    - Implementar un criterio para detener el entrenamiento si la validación no mejora después de N épocas.

Preguntas prácticas:

- ¿Qué efecto tuvo `BatchNorm` en la estabilidad y velocidad del entrenamiento?
    - Hizo que la loss baje más rápido y de forma más estable, por lo que permitió usar learning rates más altos.
- ¿Cambió la performance de validación al combinar `BatchNorm` con `Dropout`?
    - Muchas veces mejoró la generalización, pero usando dataset es pequeño, demasiada regularización empeoró el resultado a veces. Lo mejor sería probar ambas combinaciones.
- ¿Qué combinación de regularizadores dio mejores resultados en tus pruebas?
    - Generalmente, BatchNorm + Dropout + data augmentation fue una combinación muy buena. El `weight_decay` también ayudó, pero depende del problema.
- ¿Notaste cambios en la loss de entrenamiento al usar `BatchNorm`?

- Sí, la loss suele bajar más rápido y de forma más suave, y el modelo es menos sensible a la inicialización y al learning rate.

8. Inicialización de Parámetros

Preguntas teóricas:

- ¿Por qué es importante la inicialización de los pesos en una red neuronal?
  - Mejora la estabilidad. Una mala inicialización puede hacer que el entrenamiento sea muy lento o que no converja (que la red no aprenda), o que los gradientes se desvanezcan o exploten.
- ¿Qué podría ocurrir si todos los pesos se inicializan con el mismo valor?
  - La red no aprende, todos los gradientes son iguales. Todas las neuronas aprenderían lo mismo (simetría), y la red no podría aprender representaciones y patrones útiles.
- ¿Cuál es la diferencia entre las inicializaciones de Xavier (Glorot) y He?
  - Xavier está pensada para capas con activaciones lineales o `tanh`, y He es ideal para `ReLU`. La diferencia está en la varianza inicial de los pesos.
- ¿Por qué en una red con ReLU suele usarse la inicialización de He?
  - Porque está diseñada para mantener la varianza de las activaciones estable través de las capas cuando se usa `ReLU`.
- ¿Qué capas de una red requieren inicialización explícita y cuáles no?
  - Las capas lineales y convolucionales suelen requerir inicialización explícita; capas como `BatchNorm` y de activación no la necesitan porque inicializan sus parámetros automáticamente.

Actividades de modificación:

1. Agregar inicialización manual en el modelo:
   - En la clase `MLP`, agregar un método `init_weights` que inicialice cada capa:

```
def init_weights(self):
        for m in self.modules():
                if isinstance(m, nn.Linear):
                        nn.init.kaiming_normal_(m.weight)
                        nn.init.zeros_(m.bias)
```

2. Probar distintas estrategias de inicialización:
    - Xavier (`nn.init.xavier_uniform_`)
    - He (`nn.init.kaiming_normal_`)
    - Aleatoria uniforme (`nn.init.uniform_`)
    - Comparar la estabilidad y velocidad del entrenamiento.
3. Visualizar pesos en TensorBoard:
    - Agregar esta línea en la primera época para observar los histogramas:

```
for name, param in model.named_parameters():
        writer.add_histogram(name, param, epoch)
```

Preguntas prácticas:

- ¿Qué diferencias notaste en la convergencia del modelo según la inicialización?
    - Inicializaciones adecuadas (He para `ReLU`, Xavier para `tanh`) hacen que la loss baje más rápido y el modelo sea más estable. Inicializaciones malas pueden hacer que la loss no baje o que aparezcan NaNs.
- ¿Alguna inicialización provocó inestabilidad (pérdida muy alta o NaNs)?
    - Sí, inicializar todos los pesos iguales o usar una varianza muy alta causó inestabilidad/NaNs.
- ¿Qué impacto tiene la inicialización sobre las métricas de validación?
    - Una buena inicialización ayuda a alcanzar mejores métricas más rápido y evita que el modelo quede "atrapado" en malas soluciones.
- ¿Por qué `bias` se suele inicializar en cero?
    - Porque el bias no afecta la simetría de la red y empezar en cero no introduce problemas de aprendizaje. No aporta información inicial y es más estable.

Se implementaron y compararon distintas variantes de modelos MLP y CNN para la clasificación de imágenes de piel. El pipeline fue mejorado con técnicas de regularización (`Dropout`, `BatchNorm`, `weight_decay`), búsqueda de hiperparámetros, logging avanzado y análisis exploratorio de datos.

`BatchNorm` demostró mejorar la estabilidad y velocidad de convergencia del entrenamiento, permitiendo usar learning rates más altos y obteniendo curvas de loss

más suaves. Además, actuó como regularizador, ayudando a reducir el overfitting en combinación con `Dropout`.

La **inicialización de pesos** resultó fundamental: inicializaciones adecuadas (He para `ReLU`, Xavier para `tanh`) aceleraron la convergencia y evitaron problemas de pérdida explosiva o NaNs. Visualizar los histogramas de pesos en TensorBoard permitió detectar rápidamente si la inicialización era problemática.

El uso de `weight_decay` (`L2`) ayudó a mejorar la generalización, especialmente en datasets pequeños o con muchas clases, penalizando los pesos grandes y evitando que el modelo memorice los datos de entrenamiento.

La **data augmentation** fue clave para combatir el overfitting, generando mayor variedad de ejemplos y mejorando la robustez del modelo. El early stopping permitió evitar el sobreentrenamiento, deteniendo el entrenamiento cuando la métrica de validación dejaba de mejorar.

En escenarios con **pocas imágenes por clase** o **muchas clases**, el modelo tendió a sobreajustar rápidamente, lo que se reflejó en la divergencia entre la loss de entrenamiento y validación, y en la matriz de confusión. En estos casos, la regularización y el aumento de datos fueron aún más importantes.

Finalmente, la infraestructura de logging y comparación de experimentos (MLflow, TensorBoard) facilitó enormemente el análisis y la toma de decisiones, permitiendo comparar objetivamente el impacto de cada técnica y configuración.