

Instituto Tecnológico de Culiacán



**TECNOLÓGICO
NACIONAL DE MÉXICO**



Carrera: Ingeniería en Sistemas Computacionales

Materia: Inteligencia Artificial

Profesor: Zuriel Dathan Mora Felix

Tarea 2 – Reconocedor Facial

Grupo:

11:00 AM – 12:00 PM

Alumnos:

Ojeda López Luis Enrique

Saucedo Rodríguez Roberto Carlos

Detector de emociones con la arquitectura CNN

Para este proyecto decidimos utilizar una arquitectura de red neuronal convolucional (CNN), ya que es especialmente efectiva para el procesamiento de imágenes debido a su capacidad de extraer patrones espaciales relevantes. Antes de empezar a trabajar con el modelo, investigamos diversos tutoriales y documentos relacionados con la clasificación de imágenes y el uso de CNNs en PyTorch. Estos recursos nos ayudaron a entender mejor las prácticas recomendadas y a resolver problemas que surgieron durante la implementación.

Organización del dataset

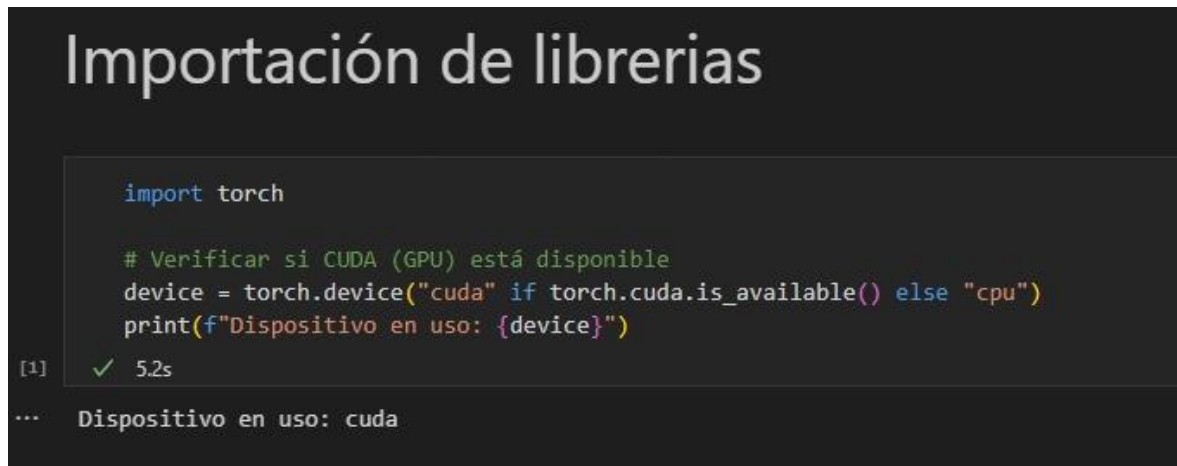
Este proyecto se centra en entrenar un modelo de red neuronal convolucional (CNN) para reconocimiento facial usando PyTorch. Iniciamos con un dataset de imágenes procesadas y organizadas en carpetas para entrenamiento (70%), validación (15%) y prueba (15%), todas con un tamaño reducido a 48x48 píxeles para optimizar el rendimiento, siendo aproximadamente 5,000 imágenes.

Durante el entrenamiento, se aplicaron transformaciones como rotaciones y volteos para mejorar la capacidad de generalización del modelo. Empleamos CrossEntropyLoss como función de pérdida y Adam como optimizador. Para evitar el sobre entrenamiento, incluimos técnicas como EarlyStopping y ReduceLROnPlateau, y se guardó automáticamente el mejor modelo durante el proceso.

Además, usamos barras de progreso y gráficas para monitorear la evolución de la pérdida y la precisión. Finalmente, evaluamos el modelo con la matriz de confusión. Así, obtuvimos un modelo sólido y listo para su uso en reconocimiento facial.

Entrenamiento del modelo:

Importación de librerías:



```
Importación de librerías

import torch

# Verificar si CUDA (GPU) está disponible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Dispositivo en uso: {device}")

[1] ✓ 5.2s
... Dispositivo en uso: cuda
```

Este bloque de código importa la librería PyTorch y verifica el entorno de ejecución para determinar si se empleará la GPU la cual se realiza mediante CUDA o la CPU para las operaciones de cómputo. La línea `import torch` carga la biblioteca de PyTorch, la cual es fundamental para el desarrollo y entrenamiento de modelos de aprendizaje profundo. Después tenemos la variable `device` la cual se inicializa utilizando `torch.device`, eligiendo "cuda" si se detecta una GPU compatible con CUDA y en caso de ser que no esté disponible se usa "cpu" en caso contrario. Esto nos permite aprovechar la aceleración de la GPU si está disponible, mejorando considerablemente el rendimiento. Finalmente, el código imprime el dispositivo seleccionado.

Importación y configuración inicial:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, datasets
from torch.utils.data import DataLoader
from keras.preprocessing.image import load_img, img_to_array

# Configuración de GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando: {device}")

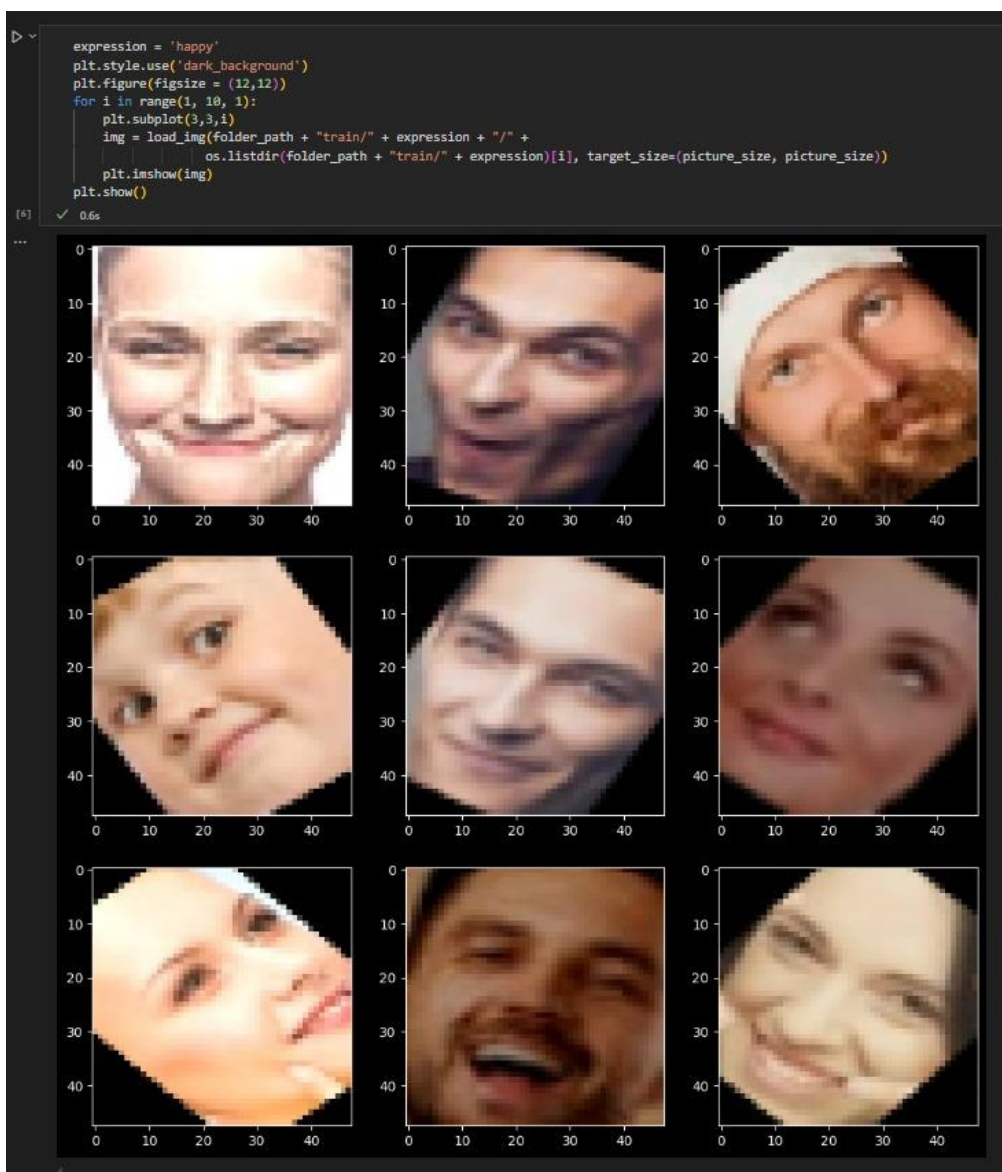
# Hiperparámetros
batch_size = 128
epochs = 38 # Definimos los epochs
learning_rate = 0.0001
```

[4] ✓ 43s

... Usando: cuda

Este bloque de código importa las bibliotecas necesarias para el proyecto, como TensorFlow y PyTorch, que se usan para trabajar con redes neuronales, y otras como Matplotlib, Seaborn, Pandas y NumPy, que ayudan a visualizar y manejar datos. Después, se configura el uso de la GPU si está disponible, lo que mejora la velocidad de procesamiento, o de la CPU en caso de que no haya GPU. Por último, se definen algunos parámetros importantes para el entrenamiento del modelo: el tamaño del grupo de datos que se procesarán juntos (`batch_size = 128`), el número de veces que el modelo verá todos los datos durante el entrenamiento (`epochs = 38`), y la tasa de aprendizaje (`learning_rate = 0.0001`) que controla cuánto se ajustan los valores internos del modelo en cada paso.

Visualización de las imágenes:



Primero, se define el tamaño al que se ajustarán las imágenes, estableciendo `picture_size` en 48 píxeles en lugar de 128 para evitar que el modelo sea demasiado pesado y lento. También se especifica la ruta de la carpeta donde están almacenadas las imágenes que se usarán para el entrenamiento.

Después, se configura el estilo de los gráficos y se selecciona la expresión facial que se desea mostrar. Luego, se cargan y visualizan algunas imágenes de ejemplo en una cuadrícula iterando un ciclo para tener una idea clara de los datos con los que se está trabajando.

Preparando los datos:

```
# Transformaciones para las imágenes (con data augmentation solo en entrenamiento)
train_transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((picture_size, picture_size)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

test_transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((picture_size, picture_size)),
    # Normalizar: valores medios y desviaciones estándar para imágenes en escala de grises
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.ImageFolder(
    folder_path + "train",
    transform=train_transform
)
test_dataset = datasets.ImageFolder(
    folder_path + "validation",
    transform=test_transform
)

# DataLoaders (optimizados para GPU)
train_loader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True,
    pin_memory=True, num_workers=4
)
test_loader = DataLoader(
    test_dataset, batch_size=batch_size, shuffle=False,
    pin_memory=True, num_workers=4
)
```

En este bloque se preparan las transformaciones que se aplicarán a las imágenes antes de pasarlas al modelo. Para las imágenes de entrenamiento, además de cambiar el tamaño y ponerlas en escala de grises, se hacen algunas modificaciones como voltearlas horizontalmente o rotarlas un poco. Esto ayuda a que el modelo aprenda mejor y no se quede pegado solo con las imágenes que tiene. Para las imágenes de validación, solo se cambian a escala de grises y tamaño, sin hacer esas modificaciones extras, porque queremos ver cómo funciona el modelo con datos que parezcan más reales.

Después, se crean los conjuntos de datos con las imágenes organizadas en carpetas según su etiqueta. Para cargar estos datos durante el entrenamiento, se usan los DataLoaders, que permiten leer las imágenes en grupos para que el entrenamiento sea más rápido y eficiente. Además, se usan algunas opciones para aprovechar mejor la memoria y el procesamiento en la GPU.

Construcción del modelo:

Construcción del Modelo

```
class EmotionCNN(nn.Module):
    def __init__(self, num_classes=4):
        super(EmotionCNN, self).__init__()

        # Bloques convolucionales
        self.features = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            nn.Conv2d(64, 128, kernel_size=5, padding=2),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            nn.Conv2d(128, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),

            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.25),
        )

        # Capas fully connected
        self.classifier = nn.Sequential(
            nn.Linear(512 * 3 * 3, 256), # Ajusta según el tamaño final de tus features
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.25),

            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.25),

            nn.Linear(512, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.Flatten(x, 1)
        x = self.classifier(x)
        return x

model = EmotionCNN(num_classes=4).to(device) # Mueve el modelo a GPU
```

En este bloque se define la arquitectura de la red neuronal convolucional llamada EmotionCNN. Primero, en el método `init`, se crean varias capas convolucionales agrupadas en `self.features`, que se encargan de extraer características importantes de las imágenes. Cada bloque tiene una capa convolucional, seguida de normalización, una función de activación ReLU, un “max pooling” para reducir dimensiones y un “dropout” para evitar que el modelo se sobreajuste. Luego, están las capas completamente conectadas (`self.classifier`), que toman las características extraídas y las usan para clasificar la imagen en una de las cuatro categorías. Estas capas también usan normalización, activación y dropout para mejorar el entrenamiento. Finalmente, en el método `forward`, se define cómo pasan los datos por la red: primero por las capas convolucionales, luego se aplanan los datos para pasar a las capas `fully connected` y se obtiene la predicción final. Al final, se crea una instancia del modelo y se envía al dispositivo adecuado (GPU o CPU) para que esté listo para entrenar.

Optimización:

```
# Optimizador y función de pérdida
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Callbacks en PyTorch
from torch.optim.lr_scheduler import ReduceLROnPlateau

# 1. EarlyStopping
class EarlyStopping:
    def __init__(self, patience=10, delta=0):
        self.patience = patience
        self.delta = delta
        self.best_loss = np.inf
        self.counter = 0

    def __call__(self, val_loss):
        if val_loss < self.best_loss - self.delta:
            self.best_loss = val_loss
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                return True # Detener entrenamiento
        return False

early_stopping = EarlyStopping(patience=10)

# 2. ModelCheckpoint (guardar el mejor modelo)
best_val_accuracy = 0.0

# 3. ReduceLROnPlateau
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.2, patience=10, verbose=True)
```

Se define la función de pérdida `CrossEntropyLoss`, que es adecuada para problemas de clasificación, y se utiliza el optimizador `Adam` con la tasa de aprendizaje establecida previamente para ajustar los parámetros del modelo. Para mejorar el proceso de entrenamiento, se crea una clase `EarlyStopping` que detiene el entrenamiento si la pérdida de validación no mejora después de varias épocas seguidas, evitando así el sobreentrenamiento. También se mantiene una variable para registrar la mejor precisión en validación, lo que facilita guardar el modelo con mejor rendimiento. Finalmente, se configura un scheduler llamado `ReduceLROnPlateau` que reduce la tasa de aprendizaje cuando el progreso del entrenamiento se estanca, permitiendo ajustes más finos en los pesos del modelo.

Entrenamiento y validación:

```
from tqdm import tqdm # Importar la librería
import time

# Listas para métricas
train_loss = []
val_loss = []
train_accuracy = []
val_accuracy = []

for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    # Barra de progreso para el entrenamiento
    train_loader_tqdm = tqdm(train_loader, desc=f'Epoch {epoch+1}/{epochs} [train]', leave=False)

    for images, labels in train_loader_tqdm:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

    # Actualizar la barra de progreso con métricas en tiempo real
    train_loader_tqdm.set_postfix({
        'loss': running_loss / (total_train / batch_size),
        'acc': 100 * correct_train / total_train
    })

    # Métricas de entrenamiento
    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct_train / total_train
    train_loss.append(train_loss)
    train_accuracy.append(train_accuracy)

    # Validación (con barra de progreso)
    model.eval()
    val_loss = 0.0
    correct_val = 0
    total_val = 0

    val_loader_tqdm = tqdm(val_loader, desc=f'Epoch {epoch+1}/{epochs} [val]', leave=False)

    with torch.no_grad():
        for images, labels in val_loader_tqdm:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            val_loss += criterion(outputs, labels).item()
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

        val_loader_tqdm.set_postfix({
            'val loss': val_loss / (total_val / batch_size),
            'val acc': 100 * correct_val / total_val
        })

    # Métricas de validación
    val_loss /= len(val_loader)
    val_accuracy = 100 * correct_val / total_val
    val_loss.append(val_loss)
    val_accuracy.append(val_accuracy)

    # Scheduler
    scheduler.step(val_loss)

    # ModelCheckpoint
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        torch.save(model.state_dict(), 'best_model.pth')
        print(f"¡Mejor modelo guardado con val_accuracy: {val_accuracy:.2f}!")

    # EarlyStopping
    if early_stopping(val_loss):
        print("EarlyStopping activado!")
        break

# Resumen de la época (se hace para mejor legibilidad)
print(f"Epoch {epoch+1}/{epochs}")
print(f"Train loss: {train_loss[-1]} | Train acc: {train_accuracy[-1]}")
print(f"Val loss: {val_loss[-1]} | Val acc: {val_accuracy[-1]}")
print("-" * 50)
```

En la parte final del entrenamiento, se está definiendo el ciclo de entrenamiento y validación del modelo utilizando la librería tqdm para mostrar barras de progreso y métricas en tiempo real. Al inicio, se crean listas para guardar las métricas de pérdida y precisión tanto en entrenamiento como en validación. Para cada época, el modelo se entrena pasando por todo el conjunto de entrenamiento: primero se reinicia el gradiente, luego se calcula la predicción y la pérdida, se hace la retro propagación y se actualizan los pesos con el optimizador. Además, se calculan la precisión y la pérdida promedio de la época, y se actualizan las barras de progreso para ver cómo va el entrenamiento.

Al terminar la parte de entrenamiento, el modelo se evalúa con los datos de validación sin actualizar los pesos. En esta validación también se registran las métricas, y se muestra el progreso con

otra barra. Después de cada época, se ajusta la tasa de aprendizaje con el scheduler y, si la precisión de validación mejora, se guarda el mejor modelo. Por último, se usa EarlyStopping para detener el entrenamiento si la pérdida de validación no mejora en varias épocas, evitando entrenar de más. Así, se tiene un mejor control sobre el entrenamiento del modelo y se asegura que funcione bien con los datos de validación.

Demonstración del entrenamiento:

```
print(f"\nEpoch {epoch+1}/{epochs}")
print(f"Train Loss: {train_loss:.4f} | Train Acc: {train_accuracy:.2f}%")
print(f"Val Loss: {val_loss:.4f} | Val Acc: {val_accuracy:.2f}%")
print("-" * 50)
```

[10] 9m 17.8s

...

Mejor modelo guardado con val_accuracy: 25.48%

Epoch 1/38
Train Loss: 1.3959 | Train Acc: 32.16%
Val Loss: 1.4771 | Val Acc: 25.48%

Mejor modelo guardado con val_accuracy: 30.17%

Epoch 2/38
Train Loss: 1.3175 | Train Acc: 37.56%
Val Loss: 1.4076 | Val Acc: 30.17%

Mejor modelo guardado con val_accuracy: 32.09%

Epoch 3/38
Train Loss: 1.2865 | Train Acc: 39.20%
Val Loss: 1.3723 | Val Acc: 32.09%

Mejor modelo guardado con val_accuracy: 32.17%

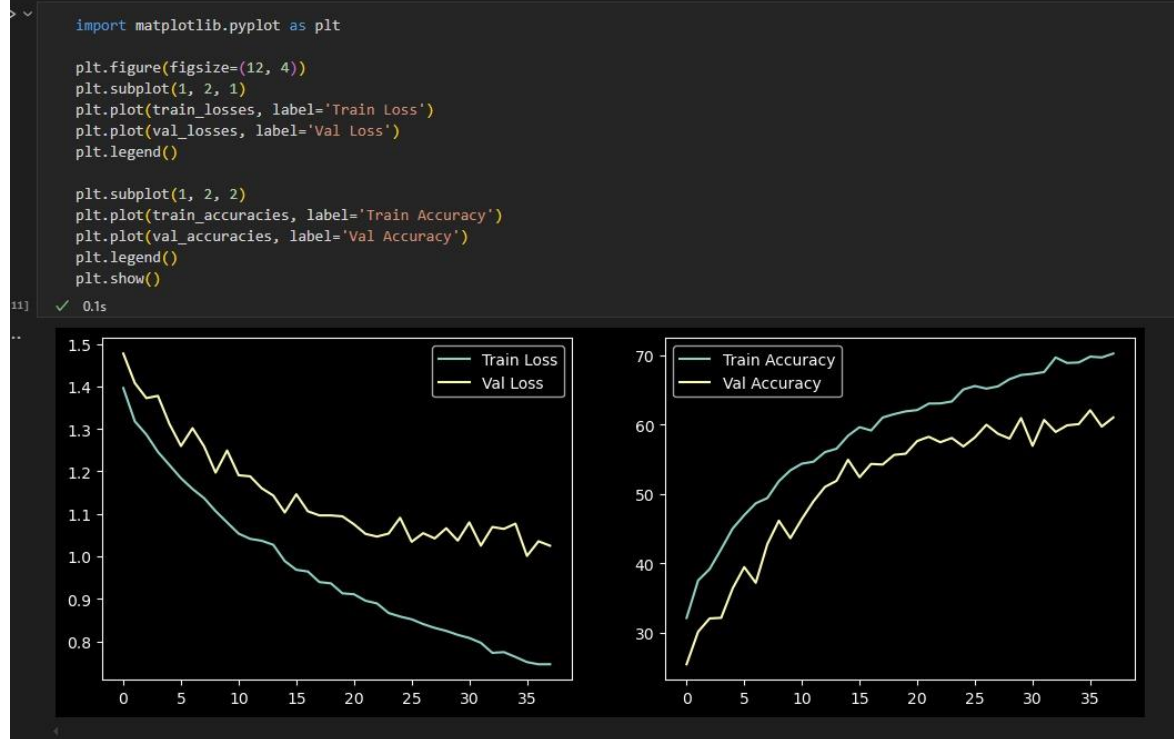
Epoch 4/38
Train Loss: 1.2456 | Train Acc: 42.08%
Val Loss: 1.3775 | Val Acc: 32.17%

Mejor modelo guardado con val_accuracy: 36.43%

Epoch 5/38
Train Loss: 1.2147 | Train Acc: 45.05%
Val Loss: 1.3113 | Val Acc: 36.43%

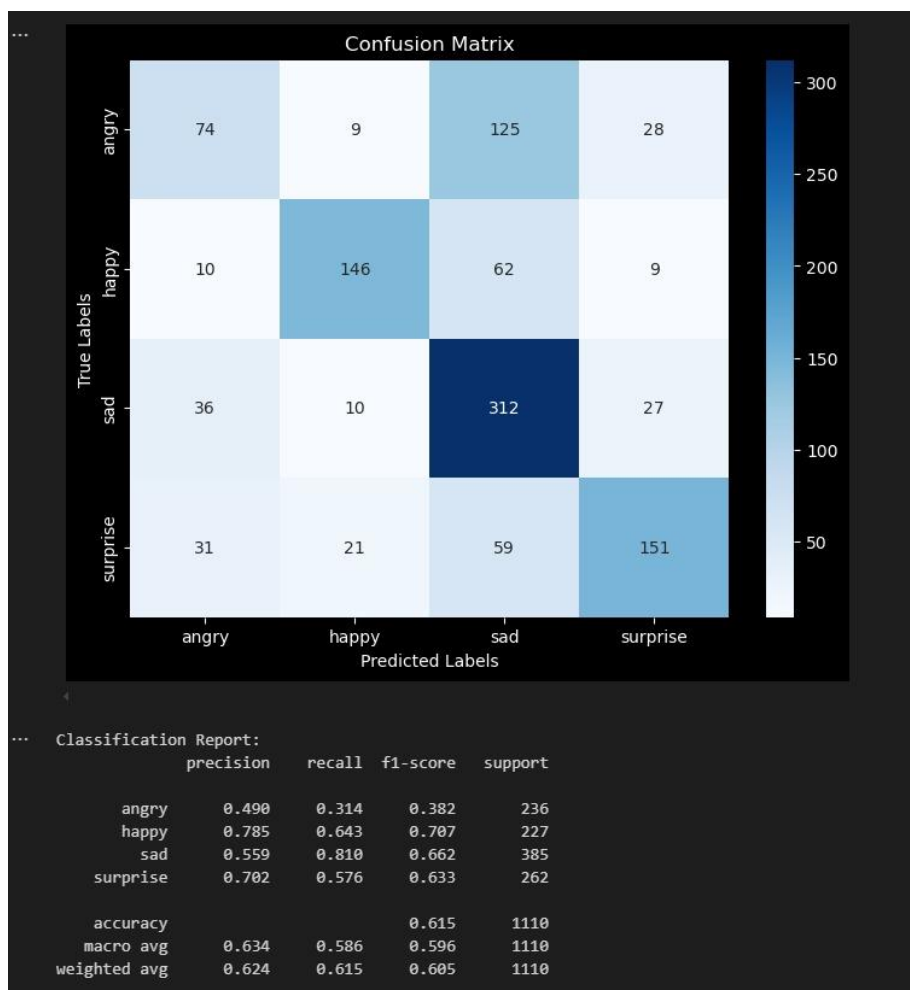
Visualización de métricas y guardado final:

Trazando la precisión y pérdidas



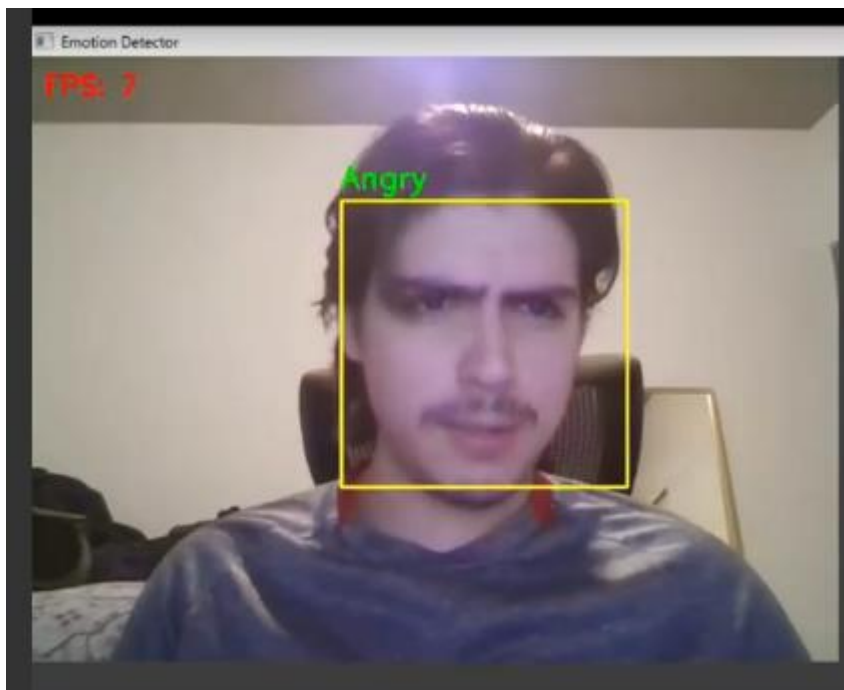
Por último, se crea una gráfica para visualizar cómo evolucionaron las pérdidas y las precisiones durante el entrenamiento y la validación, usando la librería matplotlib. Se muestran dos gráficos: uno para comparar las pérdidas y otro para comparar la precisión entre los conjuntos de entrenamiento y validación. Esto es útil para ver si el modelo está aprendiendo bien o si hay sobreentrenamiento. Finalmente, se guarda el modelo entrenado con el método `torch.save`, asegurándonos de tenerlo listo para usarlo más adelante sin necesidad de volver a entrenarlo.

Visualización de la matriz de confusión utilizando sklearn.metrics:

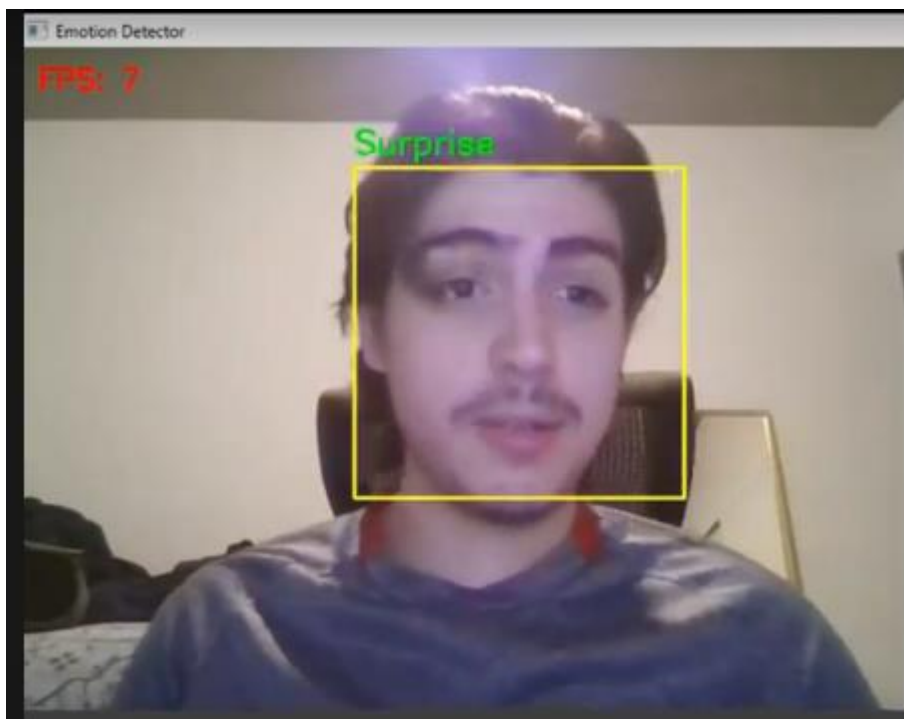


Probando el modelo:

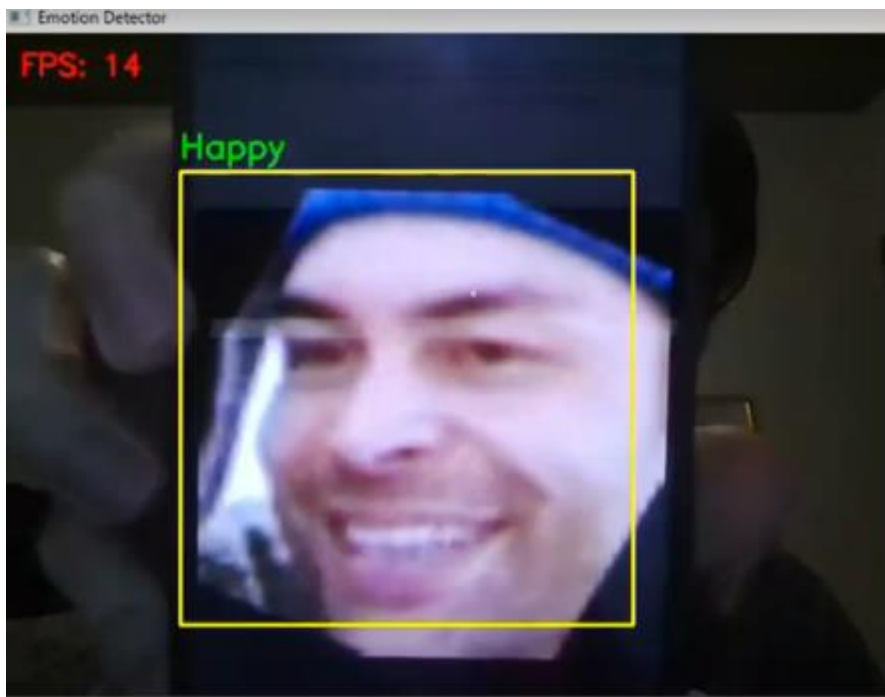
Enojado:



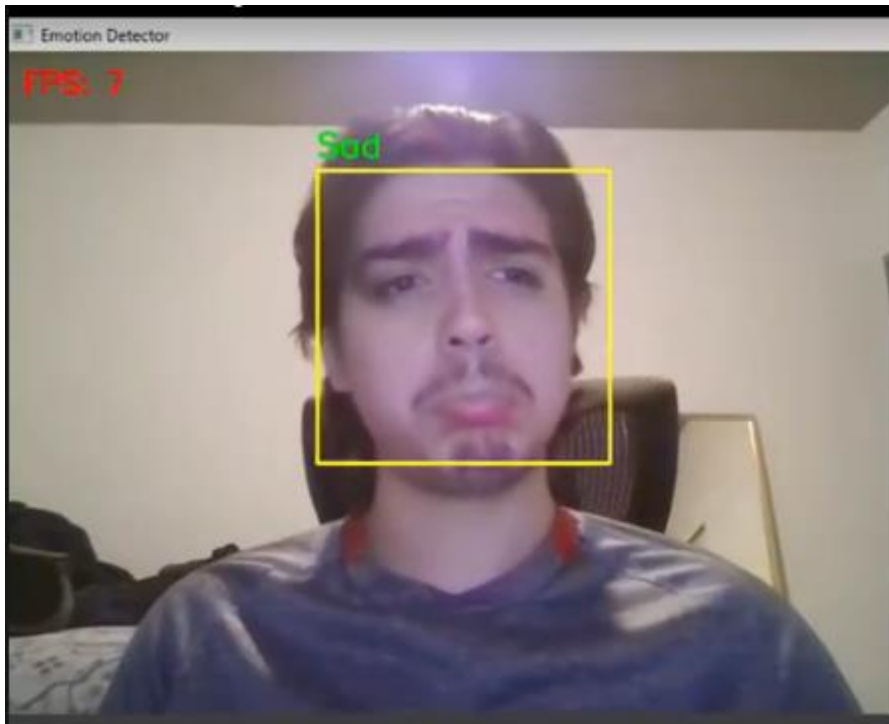
Sorprendido:



Feliz:



Tristeza:



Enlace al video de youtube de la muestra del reconocedor:

<https://www.youtube.com/watch?v=qW7PUhqzx8M>