ProyectoIA Hands

EQUIPO MICHOACÁN

2024B - Programación para Internet

29/10/2024

219516075| Maldonado Cardenas Diego Arath

222791214 | Navarrete Plancarte José Gael

222790927| García Castañeda Aldo

Resumen del proyecto reconocimiento de manos utilizando machine learning

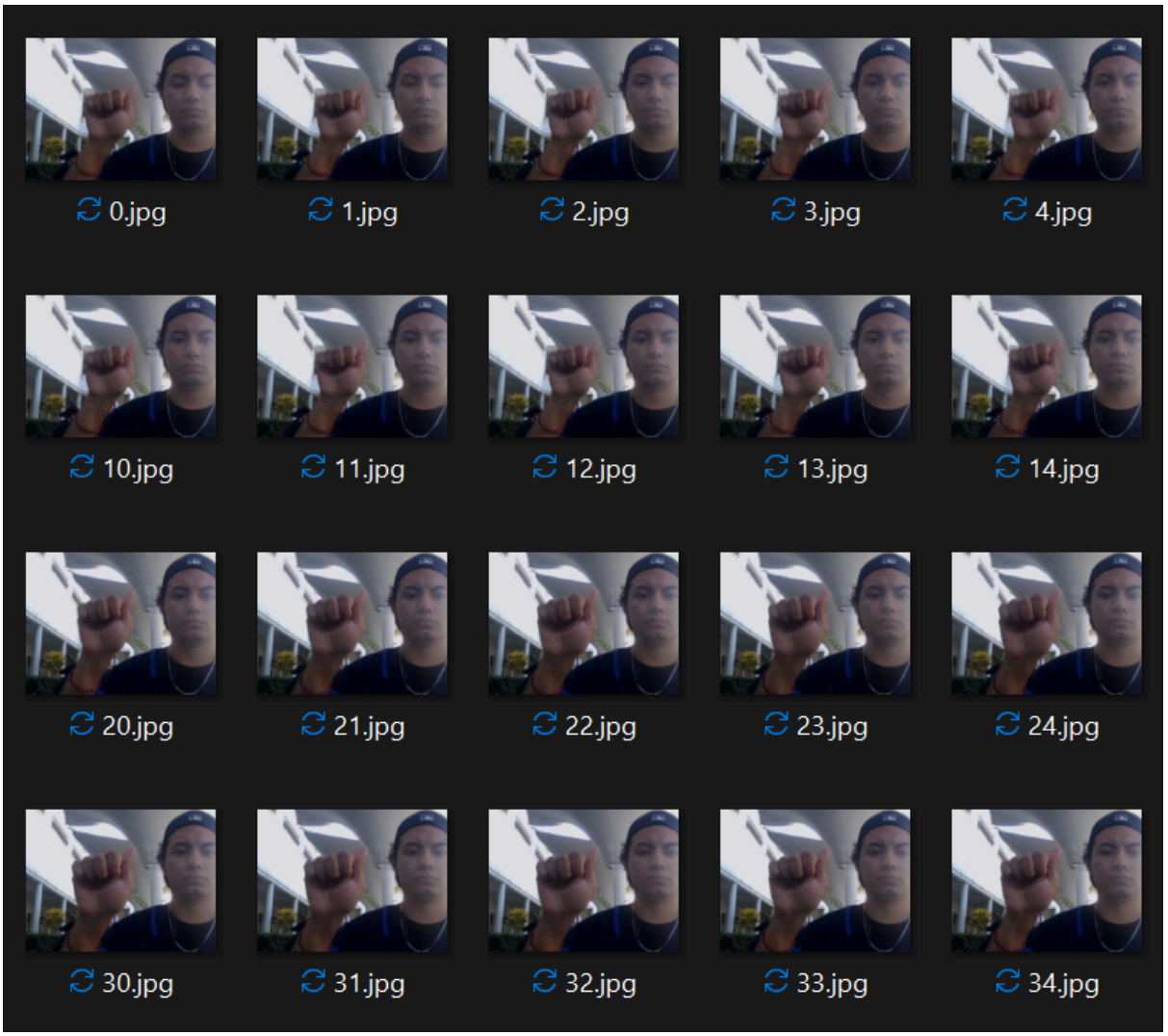
Como proyecto realizamos un traductor y lector con voz de lenguaje de señas en tiempo real utilizando visión artificial y Machine Learning para el reconocimiento de cada seña.

Lo primero que hicimos fue consumir varios vídeos de machine learning y cómo es que funcionaba este tipo de inteligencia artificial para poder enseñarla a reconocer todo tipo de señas. Una vez consumida toda la información necesaria empezamos a programar en Python.

Nuestro repositorio en GitHub contiene 4 archivos indispensables para crear un dataset, entrenar un modelo de IA y posteriormente usar este modelo para hacer predicciones en las señas que se muestran en cámara

-recolectar_imagenes.py:

Es el primer archivo a ejecutar, al hacerlo veremos que se abre una ventana que usa la cámara para recolectar las imágenes de cada clase (cada clase siendo una seña en el lenguaje de señas), en el código está señalada que linea hay que modificar para elegir el número de imágenes a guardar, y el número de clases (señas) a registrar. Automáticamente están especificadas 26 clases y una recolección de 150 imágenes por cada una.



Cómo podemos apreciar nosotros dividimos por carpeta cada letra, dentro de las carpetas tenemos 150 fotos para entrenar a la inteligencia artificial.

Una vez que la cámara está abierta, hay que presionar la letra 'Q' para comenzar la detección de imágenes, mientras se está en esta fase hay que mantener a vista de la cámara la seña que queremos guardar. Este proceso se repetirá el número de veces que clases hayamos especificado en el código. Una vez terminado el proceso obtendremos una carpeta llamada 'data' con subcarpetas para cada clase, y dentro de cada una de estas carpetas tendremos todas las imágenes recién captadas por cada seña.

-crear_dataset.py:

Una vez creada la carpeta 'data' con las imágenes dentro, no hace falta más que ejecutar este nuevo script, que tomará toda la información de la carpeta para crear un archivo binario llamado 'data.pickle', este contiene toda la información necesaria para entrenar a nuestro modelo.

-entrenar_modelo.py:

Ahora contamos con el archivo 'data.pickle', por lo que solo hace falta ejecutar este script para entrenar y testear el modelo encargado de la detección de señas. Al ejecutar el código terminaremos viendo un mensaje en consola que indica el porcentaje de datos detectado correctamente. Ahora explicaremos un poco más sobre el data.pickle:

El dataset utilizado para este proyecto fue cargado desde un archivo data.pickle. Este archivo contiene dos componentes importantes:

data: Un arreglo con las características de las muestras.

labels: Un arreglo con las etiquetas correspondientes a cada muestra.

Ambos elementos se cargaron utilizando la función pickle.load(). Los datos fueron convertidos en arreglos de numpy para facilitar su manipulación.

El dataset se dividió en dos subconjuntos:

Conjunto de Entrenamiento (80% de los datos): Utilizado para entrenar al modelo.

Conjunto de Prueba (20% de los datos): Utilizado para evaluar el rendimiento del modelo sobre datos no vistos.

Para garantizar una representación adecuada de las etiquetas en ambos subconjuntos, se utilizó la opción stratify=labels en la función train_test_split. Además, se activó el parámetro shuffle=True para barajar las muestras y evitar sesgos en la selección.

El modelo utilizado fue RandomForestClassifier, un algoritmo de aprendizaje basado en múltiples árboles de decisión. Se utilizó la implementación por defecto de scikit-learn, sin especificar hiperparámetros adicionales.

El RandomForestClassifier es un algoritmo de aprendizaje automático basado en un método de ensamble que utiliza múltiples árboles de decisión para realizar tareas de clasificación. Pertenece a la familia de los random forests (bosques aleatorios), los cuales combinan las predicciones de varios árboles individuales para mejorar la precisión y reducir el riesgo de sobreajuste (overfitting).

El modelo fue entrenado con el conjunto de entrenamiento mediante la función fit():

```
model = RandomForestClassifier()
model.fit(x_train, y_train)
```

Después de entrenar el modelo, se realizaron predicciones sobre el conjunto de prueba con la función predict():

```
y_predict = model.predict(x_test)
```

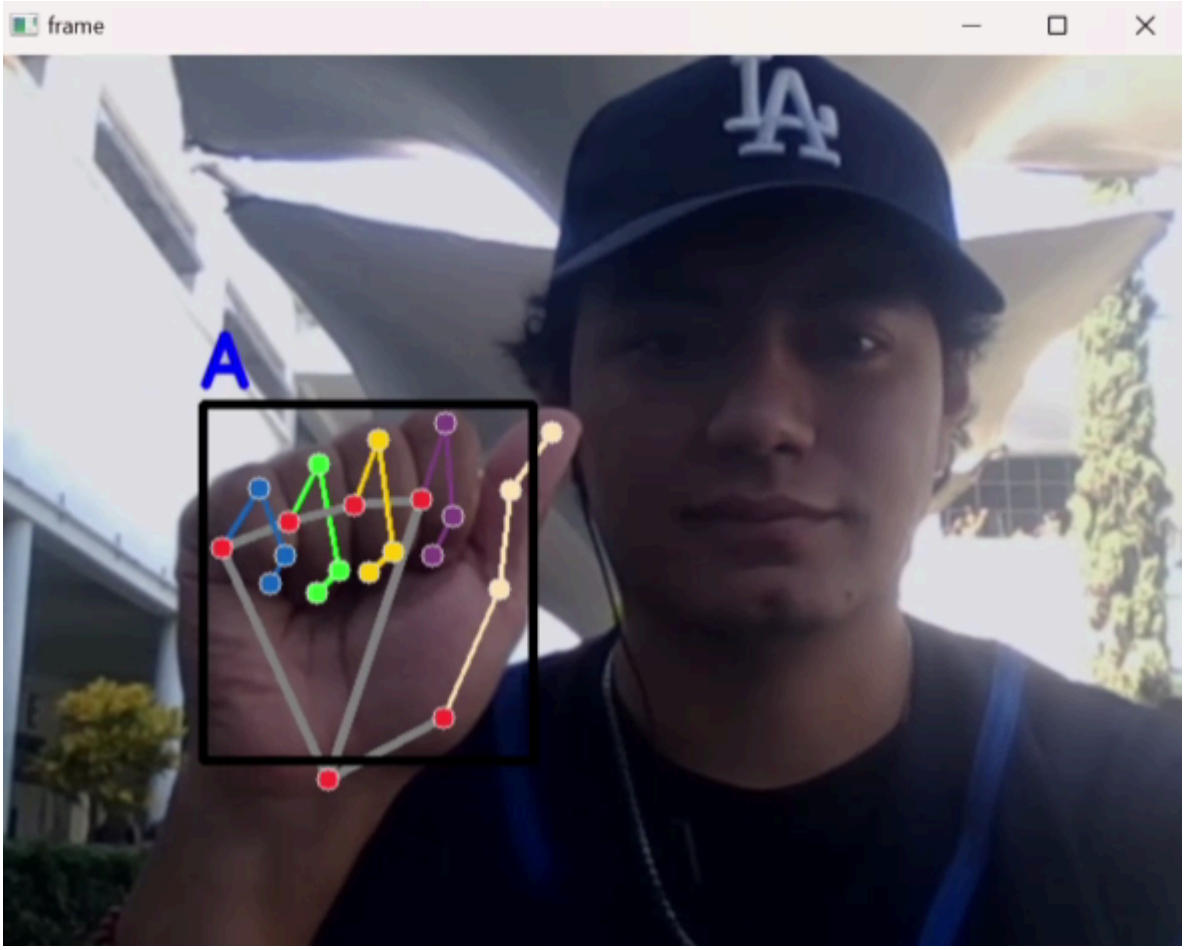
El rendimiento del modelo se evaluó utilizando la métrica accuracy score (precisión), la cual mide el porcentaje de muestras correctamente clasificadas:

score = accuracy_score(y_predict, y_test)

print('{}% of samples were classified correctly !'.format(score * 100))

-traductor_main.py:

¡Es momento de ejecutar el último script!, este archivo ejecutará de nueva cuenta una ventana que activa la cámara integrada de tu computador y está listo para detectar las señas que le enseñaste al modelo incluso reproducidas con voz



Ahora para el primer archivo de python generado esta es la explicación del código:

```
import pickle
import cv2
import mediapipe as mp
import numpy as np
import pyttsx3
import time
import threading

model_dict = pickle.load(open('./model.p', 'rb'))
model = model_dict['model']
```

Aquí mismo se carga el modelo entrenado que en este caso es el archivo “model.p”, este archivo es el más importante ya que tiene toda la información necesaria para poder reconocer cada seña.

```
cap = cv2.VideoCapture(0)

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)
```

Aquí en este apartado es donde tenemos las funciones de la cámara, se inicializan las herramientas de MediaPipe para la detección de manos y el dibujo de las conexiones de la mano.

```
labels_dict = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K', 11: 'L',
               12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W',
               23: 'X', 24: 'Y', 25: 'Z'}
```

Se crea un diccionario que mapea cada número de clase a una letra del abecedario.

```python
while True:

    data_aux = []
    x_ = []
    y_ = []

    ret, frame = cap.read()

    H, W, _ = frame.shape

    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    results = hands.process(frame_rgb)
```

En este bucle, se capturan frames de la cámara en tiempo real, se convierten a RGB para que MediaPipe pueda procesarlos correctamente y results contiene los puntos de referencia de la mano detectada.

```python
if results.multi_hand_landmarks:
    for hand_landmarks in results.multi_hand_landmarks:
        mp_drawing.draw_landmarks(
            frame,  # image to draw
            hand_landmarks,  # model output
            mp_hands.HAND_CONNECTIONS,  # hand connections
            mp_drawing_styles.get_default_hand_landmarks_style(),
            mp_drawing_styles.get_default_hand_connections_style())
```

Si se detectan manos, se dibujan los puntos de referencia en el frame.

```python
for hand_landmarks in results.multi_hand_landmarks:
    for i in range(len(hand_landmarks.landmark)):
        x = hand_landmarks.landmark[i].x
        y = hand_landmarks.landmark[i].y

        x_.append(x)
        y_.append(y)

    for i in range(len(hand_landmarks.landmark)):
        x = hand_landmarks.landmark[i].x
        y = hand_landmarks.landmark[i].y
        data_aux.append(x - min(x_))
        data_aux.append(y - min(y_))
```

Se recopilan las coordenadas (normalizadas) de los puntos de referencia de la mano y se almacenan en data_aux para que el modelo pueda utilizarlas.

```python
    cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
    cv2.putText(frame, predicted_character, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 1.3, (255, 0, 0), 3,
                cv2.LINE_AA)

cv2.imshow('frame', frame)
if cv2.waitKey(1) & 0xFF == ord('q'):
    break
```

Se dibuja un recuadro alrededor de la mano y se muestra el carácter detectado y cv2.imshow muestra el frame en una ventana, y el bucle se interrumpe cuando se presiona la tecla 'q'.

Pero para que este código de python pueda desplegarse a una página web y poder utilizarse, tomamos en cuenta FastApi que fue algo que se nos fue enseñado durante las sesiones de clase.

Por lo que agarramos el primero documento de Python junto con el archivo que tiene la maquina ya entrenada, y los combinamos para realizar una API, de la siguiente manera:

```python
from fastapi import FastAPI, File, UploadFile
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import pickle
import cv2
import mediapipe as mp
import numpy as np

# Configuración de la API
app = FastAPI()

# Configuración de CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Cargar el modelo de lenguaje de señas
model_dict = pickle.load(open('model.p', 'rb'))
model = model_dict['model']
```

Importamos las librerías y configuramos el CORS para que no tengamos problemas accediendo a la API desde nuestro despliegue de la aplicación, por eso en allow_origins tenemos un asterisco, permitiendo todos los orígenes, también volvemos a cargar el archivo donde está entrenada la máquina.

```python
# Configuración de MediaPipe para detección de manos
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)

# Diccionario de etiquetas
labels_dict = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K', 11: 'L',
               12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W',
               23: 'X', 24: 'Y', 25: 'Z'}
```

También nos traemos la configuración ya mencionada anteriormente de MediaPipe y el diccionario de etiquetas mapeado.

```python
# Endpoint para procesar la imagen y reconocer el signo
@app.post("/recognize-sign", response_model=SignResponse)
async def recognize_sign(file: UploadFile = File(...)):
    # Lee la imagen recibida
    image_data = await file.read()
    # Convierte la imagen para OpenCV
    np_img = np.frombuffer(image_data, np.uint8)
    image = cv2.imdecode(np_img, cv2.IMREAD_COLOR)

    # Convertir a RGB y procesar con MediaPipe
    frame_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = hands.process(frame_rgb)

    if results.multi_hand_landmarks:
        data_aux = []
        x_ = []
        y_ = []

        for hand_landmarks in results.multi_hand_landmarks:
            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y
                x_.append(x)
                y_.append(y)

            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y
                data_aux.append(x - min(x_))
                data_aux.append(y - min(y_))

        # Predecir el carácter
        prediction = model.predict([np.asarray(data_aux)])
        predicted_character = labels_dict[int(prediction[0])]

        return {"recognized_text": predicted_character}

    return {"recognized_text": "No se detectaron manos en la imagen"}
```

Creamos nuestro endpoint, y dentro de aquí mandamos la imagen que capturemos en nuestro frontend.

```python
# Para correr la API en local
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=5500)
```

Y ya para tenerlo necesitamos configurar el puerto de uvicorn.

```javascript
    try {
        const response = await fetch("https://proyecto-manos.onrender.com/recognize-sign", {
            method: "POST",
            body: formData
        });
        const data = await response.json();
        output.innerText = `Carácter Reconocido: ${data.recognized_text}`;
    } catch (error) {
        console.error("Error al enviar el fotograma:", error);
    }

    setTimeout(captureFrameAndSend, 1000); // 1 segundo entre capturas
}
```

Ya después dentro de nuestro frontend, hacemos un try n catch para que se pueda conectar nuestra frontend con la API, que por cierto fue hosteada en render.

Para poder hostearla en render lo único que necesitamos fue subirlo a github y tener un archivo de requerimientos.txt donde tengamos las librerías que utiliza nuestra APi, así se ve nuestro archivo de requerimientos:

File Edit View

```
fastapi
opencv-python
mediapipe
numpy
scikit-learn
python-multipart
uvicorn
```

Y pues por el momento con el 80% de nuestro proyecto completado, tenemos desplegado nuestro frontend con github pages y nuestro backend con render, y así es como se luce nuestro programa: