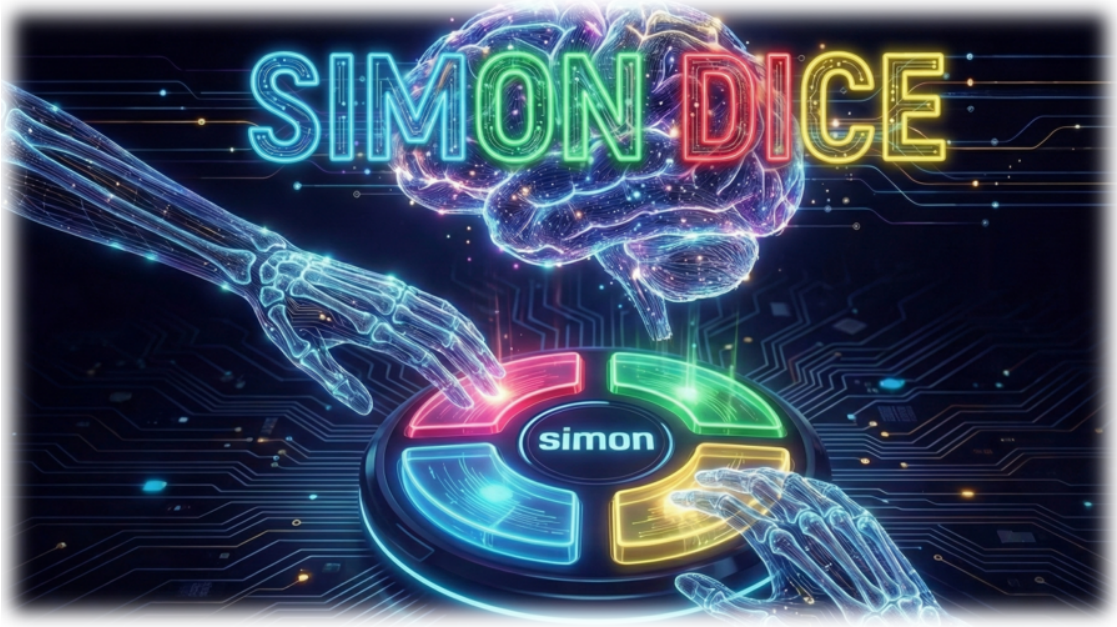
ULPGC
Escuela de Ingeniería Informática

Grado en Ingeniería Informática

Asignatura: Visión por computador

TRABAJO FINAL



Autores:
Laura Herrera Negrín
Dunia Suárez Rodríguez
Ayman Asbai Ghoudan

10 de enero de 2026

Índice

1. Introducción

1.1. Contexto y descripción del proyecto

La interacción persona-ordenador (HCI) ha evolucionado drásticamente en las últimas décadas, transitando desde las abstractas líneas de comandos hasta las interfaces gráficas táctiles que dominan la actualidad. Sin embargo, la frontera actual de la tecnología busca ir un paso más allá: la eliminación total de la barrera física, dando lugar a las **interfaces naturales de usuario (NUI)**. En este contexto, la visión por computador se erige como una herramienta fundamental, permitiendo a las máquinas "ver" e interpretar el comportamiento humano de manera análoga a como lo hacemos las personas.

El presente proyecto, titulado «**Simón Dice - Memorizador**», propone una reimaginación moderna e interactiva del clásico juego de memoria auditiva y visual de los años 80. La propuesta elimina los tradicionales botones de colores y mandos físicos, sustituyéndolos por un sistema de control basado enteramente en el reconocimiento de gestos faciales y manuales en tiempo real.

El sistema utiliza una cámara web estándar como único sensor de entrada. A través de ella, captura y procesa las acciones del jugador, quien debe replicar secuencias de comandos cada vez más complejas utilizando su propio cuerpo: desde gestos manuales (como levantar el pulgar, cerrar el puño o hacer el signo de la victoria) hasta expresiones faciales (cerrar los ojos, girar la cabeza). Esta mecánica transforma la experiencia de juego pasiva en una actividad física e inmersiva.

1.2. Motivación y justificación

La elección de este proyecto no es arbitraria, sino que responde a una serie de motivaciones técnicas, académicas y sociales que buscan explorar los límites de la computación accesible.

- **Exploración de interfaces naturales (NUI)**: Existe un interés creciente en desarrollar sistemas que entiendan el lenguaje corporal humano. La motivación principal es demostrar que es posible crear interfaces fluidas y reactivas sin necesidad de hardware costoso o sensores de profundidad (como Kinect o LiDAR), utilizando únicamente algoritmos eficientes de inteligencia artificial sobre imágenes RGB convencionales.
- **Accesibilidad y usabilidad**: Los periféricos tradicionales (teclado, ratón, gamepads) pueden suponer una barrera para personas con movilidad reducida en los dedos o dificultades motrices finas. Un sistema basado en gestos amplios

y reconocimiento facial abre la puerta a nuevas formas de ocio digital más inclusivas y accesibles.

- **Desafío del tiempo real**: El proyecto plantea el reto técnico de procesar vídeo de alta resolución, realizar inferencias con modelos de Machine Learning y gestionar la lógica de un videojuego multimedia, todo ello en intervalos de tiempo inferiores a 33 milisegundos (para mantener 30 FPS). Esta exigencia de optimización es una motivación constante para refinar el código y elegir las arquitecturas de red más eficientes.
- **Aplicación práctica de la inteligencia artificial**: Más allá de la teoría, este trabajo busca aterrizar conceptos abstractos como las redes neuronales (MLP) y la extracción de características (Feature Extraction) en un producto tangible, lúdico y funcional, cerrando la brecha entre los modelos académicos y las aplicaciones del mundo real.

1.3. Objetivos

El propósito del proyecto se desglosa en un objetivo general y varios objetivos específicos técnicos que guían el desarrollo.

1.3.1. Objetivo general

Diseñar e implementar un sistema de entretenimiento interactivo basado en visión artificial que sea capaz de interpretar, con alta precisión y baja latencia, un conjunto predefinido de gestos humanos para controlar la lógica del juego "Simón Dice", validando así la viabilidad de las interfaces gestuales en ordenadores de consumo estándar.

1.3.2. Objetivos específicos

Para alcanzar la meta global, se han establecido los siguientes hitos técnicos:

1. **Desarrollo de un pipeline de visión robusto**: Implementar un flujo de procesamiento de imágenes que integre la captura, preprocesamiento y extracción de puntos clave (*landmarks*) mediante la librería MediaPipe, garantizando la detección estable de manos y rostro bajo diferentes condiciones de iluminación.
2. **Creación y entrenamiento de modelos de clasificación**: Generar un *dataset* propio, diverso y equilibrado de gestos faciales y manuales. Diseñar, entrenar y validar perfiles de redes neuronales (Perceptrón Multicapa - MLP) capaces de clasificar estos gestos con una precisión superior al 95 %.

3. **Implementación de lógica de juego y máquina de estados**: Programar una arquitectura de software modular que gestione los estados del juego (menú, turno del sistema, turno del jugador, evaluación), asegurando la sincronización entre las instrucciones audiovisuales y la respuesta del usuario.
4. **Integración multimedia y feedback de usuario**: Desarrollar un sistema de retroalimentación claro que informe al jugador de sus aciertos y errores mediante efectos sonoros y visuales en pantalla, minimizando la latencia percibida.
5. **Optimización del rendimiento**: Asegurar que el coste computacional del sistema permita su ejecución fluida en CPUs convencionales, evitando el uso obligatorio de GPUs dedicadas para garantizar la portabilidad del software.

2. Tecnologías y herramientas

Para el desarrollo de este proyecto se ha utilizado un stack tecnológico basado en Python, aprovechando su extenso ecosistema para ciencia de datos, visión artificial y desarrollo de videojuegos.

2.1. Lenguaje y entorno

- **Python 3.11**: Lenguaje principal del proyecto.
- **Conda**: Gestor de entornos utilizado para aislar las dependencias.
- **Jupyter Notebooks**: Utilizados para el prototipado, generación de datasets y entrenamiento de modelos.

2.2. Librerías principales

- **OpenCV (cv2)**: Herramienta fundamental para la captura de vídeo desde la webcam, preprocesamiento de imágenes (flip, conversión de color) y visualización de la interfaz básica.
- **MediaPipe**: Framework desarrollado por Google. Se utilizan los módulos:
  - `FaceMesh`: Para generar una malla facial de 468 puntos y detectar gestos de la cabeza y ojos.
  - `Hands`: Para detectar 21 puntos clave en las manos y reconocer posturas complejas.
- **Scikit-Learn**: Librería de ML utilizada para implementar el `MLPClassifier` (Perceptrón Multicapa), responsable de clasificar los vectores de características en gestos concretos.
- **NumPy**: Esencial para operaciones matemáticas vectoriales, normalización de coordenadas y manipulación de arrays.
- **Pickle**: Para la serialización (guardado) y carga de los modelos entrenados (`.pkl`).
- **Pygame**: Utilizada para la gestión de efectos de sonido (feedback de acierto/error) en el juego y narración de instrucciones.
- **Pillow (PIL)**: Utilizada para renderizar texto UTF-8. Esto es crítico ya que OpenCV utiliza por defecto fuentes tipo **Hershey**, las cuales carecen de soporte para glifos extendidos (tildes, eñes), lo que comprometería la calidad de la interfaz en castellano.

3. Estructura del repositorio

El proyecto se organiza en la siguiente estructura de directorios, separando el código fuente (`src`), los datos (`dataset`) y la documentación:

```
/
|-- dataset/                                # Imágenes para entrenamiento
|   |-- gestos_cara/                        # Dataset de gestos faciales
|   |-- gestos_manos/                       # Dataset de gestos manuales
|
|-- src/                                    # Código fuente del proyecto
|   |-- game/                               # Lógica del juego (Simon Says)
        |-- fonts/                          # Fuentes para renderizar texto en Open
        |-- instructions/                   # Audios para la narración de instrucci
        |-- sounds/                         # Sonidos para feedback de acierto/erro
        |-- game_multijugador.ipynb         # Jupyter Notebook principal
|   |-- generate_images/                    # Scripts para captura de datos
|   |-- memory/                             # Documentación LaTeX
|   |-- scripts/                            # Utilidades auxiliares
|   |-- train/                              # Notebooks de entrenamiento (MLP)
|
|-- README.md                               # Documentación general y setup
```

4. Metodología y desarrollo

El desarrollo se ha estructurado en tres fases secuenciales: generación del dataset, entrenamiento de los modelos clasificadores y desarrollo de la lógica del juego.

4.1. Fase 1: Recolección del dataset

Para entrenar un modelo robusto, es esencial contar con datos de calidad. Por ello, para construir el textitdataset, inicialmente se realizó una búsqueda de imágenes en repositorios públicos como Kaggle. Se identificaron conjuntos de datos relevantes, tales como:

- CEW Dataset (Closed Eyes in the Wild), útil para la detección de ojos cerrados.
- HaGRID (HAnd Gesture Recognition Image Dataset), para el reconocimiento de posturas de la mano.

Sin embargo, para adaptar el sistema a las necesidades específicas del juego y mejorar la robustez ante el entorno real de uso, se desarrollaron dos scripts en Python que permiten capturar imágenes de la webcam sistemáticamente para enriquecer el conjunto de datos.

- El sistema captura ráfagas de imágenes mientras el usuario realiza un gesto específico.
- Las imágenes se guardan automáticamente en carpetas etiquetadas con el nombre del gesto (ej. `dataset/gestos_manos/4_Pulgar_Arriba`).
- Se capturaron múltiples variaciones de cada gesto (diferentes distancias, manos izquierda/derecha, ligeras rotaciones) para mejorar la generalización.

Las clases definidas incluyen:

- **Faciales**: Neutro, Ojos Cerrados, Cabeza Derecha, Cabeza Izquierda.
- **Manuales**: Mano Arriba, Puños Cerrados, Pulgar Arriba, Victoria, Rock, Llamada, OK.

4.2. Fase 2: Entrenamiento (método y cómo)

El entrenamiento se llevó a cabo en los notebooks:

- `EntrenoCara.ipynb`
- `EntrenoManoCuerpo.ipynb`

La metodología empleada se basa en el aprendizaje supervisado sobre características geométricas.

4.2.1. Extracción de características

En lugar de utilizar redes neuronales convolucionales (CNN) que procesan la imagen completa (pixel a pixel), se optó por un enfoque basado en **landmarks** (puntos clave).

1. Se procesa cada imagen del dataset con MediaPipe.
2. Se extraen las coordenadas (x, y, z) de los puntos de interés (468 para cara, 21 para mano).
3. **Normalización**: Crucial para que el modelo funcione independientemente de la posición del usuario en la pantalla. Se centra el gesto restando el centroide y se escala dividiendo por la máxima distancia absoluta desde el centro.

4.2.2. Pipeline de procesamiento de datos

El flujo de transformación de los datos sigue un proceso secuencial diseñado para maximizar la eficiencia. En la Figura 1 se detalla este esquema.

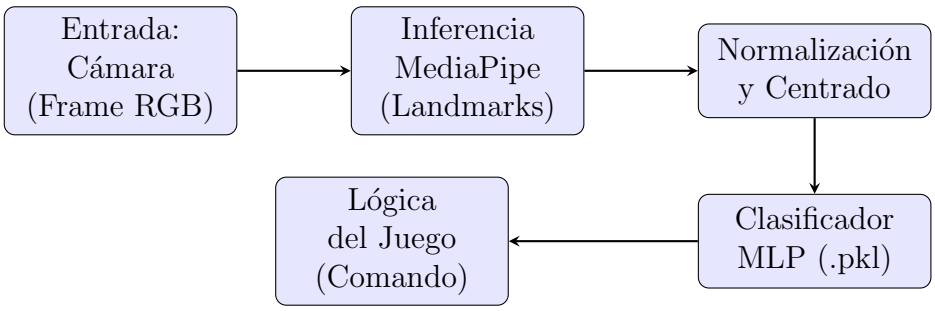


Figura 1: Esquema del flujo de datos del sistema.

Como se observa, el proceso consta de las siguientes etapas:

- **Captura e Inferencia**: MediaPipe localiza los *landmarks* espaciales.
- **Preprocesamiento**: Se extraen las coordenadas (x, y) y se aplican técnicas de centrado y escalado unitario.
- **Vectorización**: La matriz de puntos se aplana en un vector unidimensional apto para la entrada de la red neuronal.
- **Inferencia MLP**: El Perceptrón Multicapa asigna una probabilidad a cada clase de gesto.

4.2.3. Aumento de datos (data augmentation)

Dado que el dataset propio es limitado en tamaño, se aplicaron técnicas de aumento de datos sintético sobre los vectores de características (no sobre las imágenes), lo cual es muy eficiente:

- **Ruido**: Se añade ruido gaussiano aleatorio a las coordenadas para simular el "jitter"de la cámara.
- **Escalado y rotación**: Se aplican transformaciones matriciales 2D para simular que el usuario está más cerca/lejos o inclina la mano.

4.2.4. Clasificación con perceptrón multicapa (MLP)

El núcleo de decisión del sistema se basa en un **perceptrón multicapa (MLP)**, una red neuronal artificial de tipo *feedforward*. A diferencia de las redes convolucionales (CNN) que extraen características directamente de los píxeles de la imagen, nuestro enfoque se basa en la **ingeniería de características** (*Feature Engineering*) previa realizada por MediaPipe.

Se ha optado por una **arquitectura unificada y ligera** para ambos modelos (facial y corporal), priorizando la velocidad de procesamiento en tiempo real:

- **Arquitectura de la red (topología):**
  - **Capa de entrada:** Variable según el modelo.
    - *Manos*: 42 neuronas (21 puntos $\times$ 2 ejes).
    - *Cara*: 936 neuronas (468 puntos $\times$ 2 ejes).
  - **Capas ocultas:** $(64, 32)$. Se utiliza una estructura de ."embudo".
  - **Capa de salida:** Neuronas correspondientes al número de clases a predecir (9 para manos, 4 para cara).

Justificación de la arquitectura (64, 32): A pesar de que la malla facial cuenta con 468 puntos (936 valores de entrada), se demostró que una primera capa oculta de solo **64 neuronas** es suficiente para extraer las características relevantes.

Hiperparámetros del entrenamiento: Para ambos modelos se utilizó la implementación `MLPClassifier` de la librería *Scikit-Learn* con la siguiente configuración:

- **Función de Activación**: `ReLU` (Rectified Linear Unit), elegida por su eficiencia computacional.

- **Solver (Optimizador)**: `Adam`, un algoritmo de optimización estocástica eficiente.
- **Iteraciones máximas**: `max_iter=1500`. Se estableció un límite alto para asegurar la convergencia total del modelo, dado que el tamaño reducido de la red minimiza el riesgo de sobreajuste (*overfitting*) y el tiempo de cómputo sigue siendo bajo.
- **Semilla aleatoria**: Se fijó `random_state=42`. Esto anula la aleatoriedad en la inicialización de los pesos sinápticos y en la mezcla de datos (*shuffling*), garantizando la **reproducibilidad** exacta de los resultados y métricas presentados en este documento.

Esta arquitectura ligera permite que la inferencia del modelo tarde menos de 1 milisegundo en una CPU estándar, siendo ideal para mantener una tasa de FPS alta en el videojuego.

4.2.5. Evaluación mediante validación cruzada (K-Fold)

Para garantizar que el modelo no solo memorice los datos (overfitting) sino que sea capaz de generalizar ante nuevos usuarios, no se ha utilizado un único reparto de entrenamiento y prueba. En su lugar, se ha implementado una estrategia de **validación cruzada estratificada (Stratified K-Fold)** con $k = 5$.

Este proceso funciona de la siguiente manera:

1. El dataset total se divide en 5 bloques o "pliegues"(folds) que mantienen la proporción de cada gesto.
2. El sistema realiza 5 iteraciones de entrenamiento. En cada una, utiliza 4 bloques para entrenar y el bloque restante para evaluar la precisión.
3. Se calcula la media de los resultados para obtener una **precisión estimada real**.

Una vez obtenida una precisión satisfactoria en esta fase (generalmente superior al 95 %), se procede al entrenamiento final utilizando el 100 % de los vectores de características. Esto asegura que el archivo `.pkl` resultante contenga el máximo conocimiento posible derivado del dataset.

4.2.6. Análisis de error: matriz de confusión

Para profundizar en el rendimiento del modelo más allá del porcentaje de precisión, se generó una **matriz de confusión**. Esta herramienta es fundamental para identificar qué gestos específicos pueden inducir a error al clasificador.

Al utilizar la técnica de `cross_val_predict`, cada predicción reflejada en la matriz proviene de un subconjunto de datos que el modelo no utilizó para su entrenamiento en esa iteración particular. Esto permite observar:

- **Falsos positivos**: Casos donde un gesto neutro se confunde con una acción.
- **Confusiones geométricas**: Por ejemplo, si el modelo confunde Çabeza Derechaçon "Neutro"debido a una inclinación insuficiente.
- **Sensibilidad del parpadeo**: Verificar si el cierre de ojos se distingue correctamente de un gesto facial neutro.

Este análisis permitió ajustar los hiperparámetros del MLP y mejorar el proceso de normalización de los *landmarks* en las clases con mayor tasa de error.

A continuación se presentan las matrices de confusión resultantes del entrenamiento final:

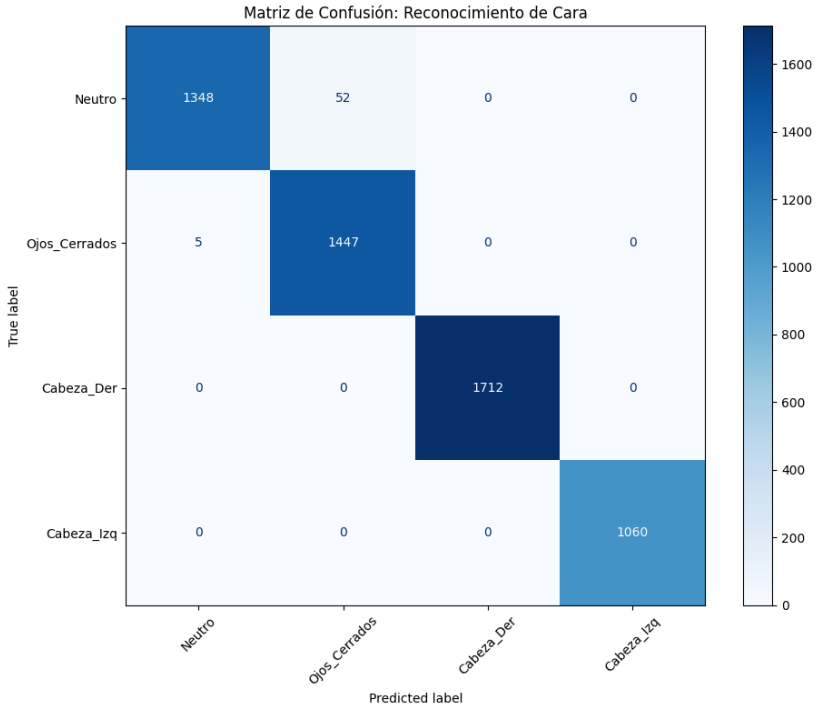


Figura 2: Matriz de confusión del modelo de gestos faciales.

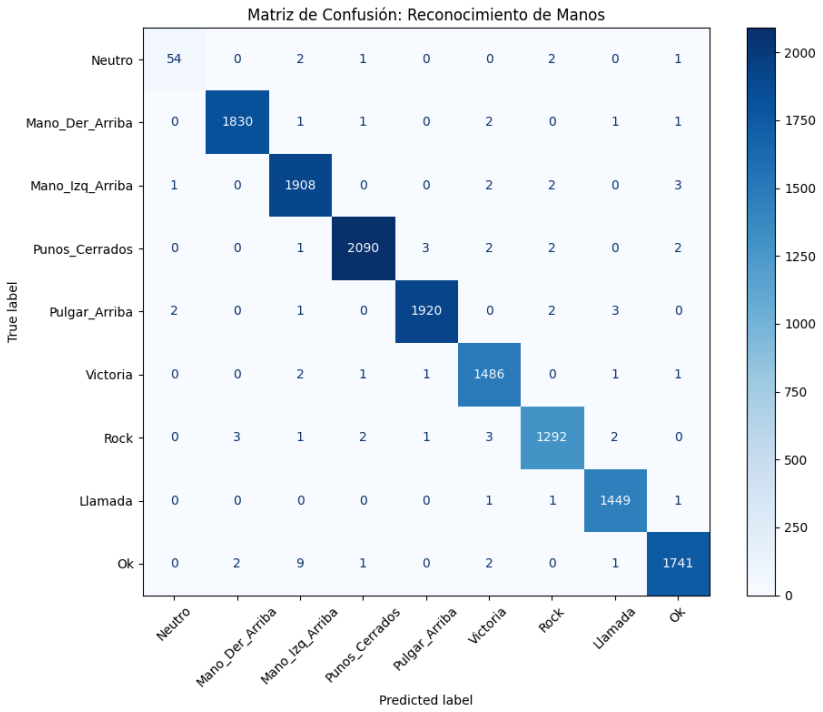


Figura 3: Matriz de confusión del modelo de gestos manuales.

Como se observa en las Figuras 2 y 3, la mayor concentración de predicciones se encuentra en la diagonal principal, lo cual indica un alto nivel de acierto en la clasificación.

Para el **modelo facial**, se obtuvo una precisión estimada mediante validación cruzada del **98.99 %**. Los errores son mínimos, lo que confirma que el modelo distingue correctamente entre ojos abiertos/cerrados y los giros de cabeza, a pesar de la sutileza de estos movimientos.

Por su parte, el **modelo manual** alcanzó una precisión del **99.47 %**. Esto demuestra que la normalización de los puntos de la mano es efectiva para distinguir entre gestos complejos como «Victoria», «Rock» o «Ok», incluso con variaciones en la ejecución por parte del usuario.

La baja tasa de falsos positivos y negativos valida la estrategia de utilizar *Data Augmentation* sintético, permitiendo que el modelo generalice correctamente sin necesidad de un dataset masivo de imágenes reales.

4.3. Fase 3: Desarrollo del juego

La lógica principal del juego reside en el archivo principal (`src/game/game_multijugador.ipynb`). El juego está implementado utilizando programación orientada a objetos para permitir la escalabilidad hacia múltiples jugadores.

4.3.1. Arquitectura y clase Jugador

Se ha introducido la clase `Player` para gestionar el estado individual de cada participante. Esta clase encapsula:

- **Identificación**: ID, nombre y límites de pantalla asignados (bounds) para dividir el área de detección en el modo dos jugadores.
- **Estado del juego**: Vidas restantes, estado de eliminación y progreso en la ronda actual.
- **Detección**: Gesto actual detectado, contadores de estabilidad (frames consecutivos) y tiempos de reacción.

La clase `Player` encapsula el estado del participante. Un aspecto técnico crítico es el **contador de estabilidad**. Para evitar falsos positivos debidos al ruido de la cámara o detecciones efímeras, el sistema solo valida un gesto si este se mantiene idéntico durante un umbral de fotogramas consecutivos (habitualmente $N = 5$).

Esta estructura permite que el juego gestione dos versiones:

1. **Un jugador**: El usuario ocupa toda la pantalla. La lógica se centra en superar la secuencia creciente.
2. **Dos jugadores (VS)**: La pantalla se divide en dos secciones. Ambos jugadores compiten simultáneamente para completar la misma instrucción. El sistema penaliza individualmente los fallos o la falta de rapidez.

4.3.2. Máquina de estados

Para gestionar el flujo del juego, se implementó una máquina de estados finitos mejorada:

1. **MENU**: Pantalla de selección de modo (1 o 2 jugadores).
2. **CHECK_PLAYERS**: (Solo multijugador) Verifica que ambos jugadores estén en posición y listos antes de comenzar.
3. **SHOW_NEW_STEP**: El turno de la CPU. El sistema:
   - Añade un nuevo paso a la secuencia.
   - Reproduce el archivo de audio correspondiente a la instrucción.
   - Muestra texto en pantalla usando **Pillow**.
4. **WAIT_NEUTRAL**: Estado de sincronización que obliga a todos los jugadores activos a volver a una posición neutra antes de la siguiente acción, evitando

falsos positivos.

5. **PLAYER_TURN**: Fase de ejecución. El sistema valida en tiempo real si el gesto detectado coincide con el esperado para cada jugador activo.
6. **GAME_OVER / SUCCESS_SEQUENCE**: Estados finales de ronda o partida.

4.3.3. Niveles de dificultad y trampas

Una característica clave es la lógica de "trampas", inspirada en el juego real:

- **Nivel básico**: El juego siempre dice "Simón dice...".
- **Nivel intermedio**: Aparecen comandos como "Modesto dice..." El jugador **no** debe realizar el gesto. Si se mueve, pierde.
- **Nivel avanzado**: Se omiten palabras clave ("Simón...", "Dice...").

Esta lógica se gestiona mediante la función `add_sequence_step`, que genera aleatoriamente instrucciones válidas o trampas según la puntuación actual.

4.3.4. Sistema de audio personalizado

Inicialmente se contempló el uso de librerías de síntesis de voz (TTS) para narrar las instrucciones. Sin embargo, se detectaron problemas de bloqueo (freezing) al ejecutar el motor TTS en un hilo paralelo al procesamiento de vídeo, lo que afectaba a la fluidez del juego.

Como solución, se implementó un sistema de audio basado en clips pre-grabados:

- Se generaron archivos de audio (.mp3) individuales para cada instrucción posible (ej. "simon_1_Mano_Der_Arriba.mp3").
- Se utiliza `pygame.mixer` para la reproducción asíncrona de estos sonidos, garantizando que el bucle de vídeo no se detenga.
- Este enfoque permite además personalizar la voz del narrador, añadiendo variedad con voces para "Simón", "Modesto.º instrucciones genéricas.

5. Justificación de decisiones de diseño

En el desarrollo de sistemas de visión por computador e interacción en tiempo real, la elección de las herramientas y arquitecturas determina la viabilidad y usabilidad del proyecto. A continuación se detallan y justifican las decisiones técnicas más relevantes adoptadas durante el desarrollo.

5.1. Estrategia de validación y entrenamiento final

A diferencia de los flujos de trabajo convencionales de Machine Learning que dividen el dataset en tres conjuntos estáticos (entrenamiento, validación y prueba), en este proyecto se ha optado por una estrategia basada en **Validación Cruzada Estratificada (Stratified K-Fold Cross-Validation)** seguida de un re-entrenamiento total.

Esta decisión se fundamenta en los siguientes puntos:

- **Optimización del dataset**: Al trabajar con un dataset generado específicamente para este juego, la validación cruzada permite utilizar cada muestra tanto para entrenar como para evaluar en diferentes iteraciones. Esto proporciona una estimación de la precisión mucho más robusta y menos sesgada que un simple *split* de 80/20, especialmente valioso cuando el volumen de datos es moderado.
- **Garantía de generalización**: El uso de `StratifiedKFold` asegura que cada "pliegue"(fold) mantenga la proporción de clases original, evitando que el modelo se sesgue hacia los gestos con más muestras durante la evaluación de su rendimiento.
- **Modelo de producción de máximo aprendizaje**: Una vez validada la arquitectura del MLP y confirmada su alta precisión (superior al 95 %), se procedió a entrenar el modelo final utilizando el 100 % de los datos disponibles. En entornos de producción, esta práctica garantiza que el clasificador haya "visto"todas las variaciones posibles de los gestos, maximizando su capacidad de respuesta ante el usuario final.

  Nota sobre el Data Augmentation: Se es consciente de que la aumentación de datos se realiza de forma previa a la validación. Aunque esto puede optimizar los resultados de la métrica, la robustez final se comprueba mediante las pruebas unitarias en tiempo real descritas en la fase de desarrollo, donde el modelo demuestra su capacidad de generalización ante nuevos usuarios.

5.2. Extracción de características vs. redes convolucionales

Una decisión fundamental fue optar por una arquitectura de dos etapas: extracción de *landmarks* (puntos clave) con MediaPipe seguida de clasificación con un Perceptrón Multicapa (MLP), en lugar de utilizar una Red Neuronal Convolucional (CNN) *end-to-end* que procese la imagen completa.

- **Eficiencia computacional (latencia)**: Una imagen estándar de webcam (1280×720) contiene casi un millón de píxeles. Procesar esta matriz con una CNN profunda en cada frame requiere una potencia de cálculo considerable (GPU), lo que limitaría el juego a ordenadores potentes.
  Al utilizar MediaPipe, delegamos la detección espacial a un modelo altamente optimizado para CPU, y nuestro clasificador MLP solo necesita procesar un vector de entrada muy pequeño (42 valores para manos, 936 para cara). Esto garantiza una tasa de fotogramas estable (> 30 FPS) incluso en equipos portátiles estándar.
- **Robustez y generalización**: Las CNNs entrenadas con pocas imágenes tienden a aprender patrones irrelevantes, como el color de fondo o la ropa del usuario (overfitting). Al trabajar exclusivamente con coordenadas geométricas normalizadas, nuestro modelo se vuelve "ciego" al entorno. Esto significa que el sistema funciona igual de bien si el usuario está en una habitación oscura, luminosa o con un fondo complejo, ya que solo "ve" la posición relativa de los puntos de la mano y la cara.

5.3. Gestión de audio: pre-grabado vs. síntesis (TTS)

La experiencia de usuario requiere instrucciones auditivas claras. Inicialmente se implementó síntesis de voz en tiempo real (*Text-to-Speech*), pero fue descartada en favor de clips de audio pre-grabados.

- **Bloqueo del hilo principal (Thread Blocking)**: Las librerías de TTS en Python, como `pyttsx3`, suelen operar de forma síncrona o tienen una gestión de hilos compleja que entra en conflicto con el bucle principal de OpenCV ('while True'). Esto provocaba micro-congelamientos ("stuttering") en el vídeo cada vez que el juego hablaba.
- **Consistencia estética**: Los motores TTS dependen de las voces instaladas en el sistema operativo del usuario, lo que hace que el juego suene diferente en cada ordenador (a veces con voces robóticas de baja calidad). El uso de archivos `.mp3` garantiza que todos los jugadores escuchen la misma voz, con la entonación y emoción correctas para el contexto del juego.

6. Desafíos técnicos y soluciones

Durante el desarrollo del proyecto surgieron varios retos inherentes a la aplicación de visión por computador en entornos no controlados:

6.1. Variabilidad en la iluminación

Los algoritmos de detección visual son sensibles a las condiciones de luz extremas (contraluz o oscuridad).
Solución: La normalización de las coordenadas de los landmarks hace que el modelo dependa de la geometría relativa de los puntos y no de los valores de intensidad de los píxeles, mitigando parcialmente este problema.

6.2. Oclusiones parciales

En ocasiones, al realizar gestos cerca de la cara, las manos pueden ocultar partes del rostro, confundiendo al detector facial.
Solución: Se implementó una lógica de priorización en el bucle principal del juego y se ajustaron los umbrales de confianza (`min_detection_confidence`) para reducir falsos positivos.

6.3. Renderizado de texto

OpenCV no soporta caracteres UTF-8 nativamente, lo que impedía escribir "Simón" o "Puño" correctamente.
Solución: Se creó la función auxiliar `poner_texto_utf8` que convierte el frame a formato PIL Image, dibuja el texto con una fuente TrueType (.otf) personalizada y lo reconvierte a OpenCV.

6.4. Latencia en tiempo real

El procesamiento de vídeo requiere un alto rendimiento para no afectar a la jugabilidad.
Solución: El uso de clasificadores MLP (redes neuronales simples) sobre vectores de características ligeros, en lugar de redes convolucionales profundas (CNN) sobre imágenes completas, garantiza una tasa de fotogramas (FPS) alta incluso en equipos con hardware modesto.

6.5. Bloqueo por librerías de audio (pyttsx3)

El uso inicial de la librería `pyttsx3` para la síntesis de voz en tiempo real provocaba bloqueos en el hilo principal de ejecución, congelando el vídeo durante la narración de instrucciones.

Solución: Se sustituyó la síntesis en tiempo real por el uso de clips de audio pregrabados generados mediante una Inteligencia Artificial de voz neuronal. Se implementó una función asíncrona basada en `pygame.mixer` y una gestión de cola de reproducción para integrar estos audios sin afectar a los FPS del juego.

7. Manual de usuario

7.1. Requisitos previos

Es necesario disponer de una webcam funcional y un entorno con las librerías instaladas:

```
pip install numpy==2.2.6 matplotlib==3.10.7 opencv-contrib-python
   ==4.12.0.88 mediapipe==0.10.14 scikit-learn==1.7.2 pygame==2.6.1
    sounddevice==0.5.3 pillow jax jaxlib ipykernel
```

7.2. Ejecución

1. Abrir la terminal en la carpeta `src/game/`.
2. Ejecutar el archivo principal del juego en Jupyter o Python.
3. Seleccionar el modo de juego: [1] Un Jugador o [2] Dos Jugadores.
4. Seguir las instrucciones en pantalla y audio.

7.3. Cómo jugar

1. **Atención**: El juego te hablará y mostrará texto.
2. **Órdenes válidas**: Si dice "Simón dice: [Acción]", debes imitar el gesto.
3. **Trampas**:
   - Si dice "Modesto dice..." o no menciona a "Simón", ¡No te muevas!
   - Mantén la posición neutra hasta que pase la ronda.
4. **Secuencia**: Debes memorizar y repetir todos los pasos acumulados.

8. Propuestas de ampliación y trabajo futuro

Aunque el sistema actual presenta una experiencia de juego completa y funcional, el campo de la visión por computador y las interfaces naturales (NUI) está en constante evolución. A continuación se detallan varias líneas de investigación y desarrollo que podrían enriquecer significativamente el proyecto en futuras iteraciones.

8.1. Reconocimiento de gestos dinámicos

A pesar de la solidez del sistema actual, existe una limitación inherente a su diseño: el modelo clasifica **posturas estáticas**. El sistema entiende si una mano está abierta o cerrada en un instante concreto (t), pero carece de contexto temporal para entender el movimiento que llevó a esa postura.

Para llevar el proyecto al siguiente nivel y permitir una interacción más rica, se propone la siguiente evolución técnica:

8.1.1. De la foto al vídeo: redes recurrentes

La ampliación natural consiste en dotar al sistema de "memoria". Técnicamente, esto implicaría sustituir o complementar el clasificador actual por una **red neuronal recurrente (RNN)**, específicamente una arquitectura **LSTM (Long Short-Term Memory)**.

En lugar de alimentar al modelo con un único *frame*, se utilizaría una ventana deslizante de los últimos N fotogramas (por ejemplo, los últimos 0.5 segundos). Esto permitiría al sistema analizar no solo la posición de los puntos, sino su **trayectoria y velocidad**.

8.1.2. Nuevas mecánicas de juego

Esta capacidad de análisis temporal abriría la puerta a gestos complejos que actualmente son imposibles de detectar:

- **Gestos de trayectoria:** Instrucciones como "Saluda con la mano" o "Dibuja un círculo en el aire".
- **Asentimiento y negación:** Detectar si el usuario mueve la cabeza diciendo "Sí" o "No", en lugar de solo detectar si mira a la derecha o izquierda.
- **Análisis de velocidad:** Mecánicas donde el juego pida realizar una acción "a cámara lenta" o "lo más rápido posible".

Esta evolución requeriría la generación de un nuevo dataset basado en secuencias

de vídeo, aumentando la complejidad del entrenamiento, pero elevando exponencialmente la naturalidad de la interacción hombre-máquina.

8.2. Adaptabilidad emocional y cognitiva

Aprovechando la malla facial de alta densidad de MediaPipe (468 puntos), se podría integrar un módulo de análisis de emociones en tiempo real.

- **Ajuste de dificultad dinámico**: Si el sistema detecta micro-expresiones de frustración o estrés en el jugador, podría reducir la velocidad de la secuencia de "Simón" automáticamente. Por el contrario, si detecta aburrimiento, podría acelerar el ritmo.
- **Feedback empático**: El juego podría reaccionar a las expresiones del usuario, por ejemplo, animándolo si falla o celebrando con él si sonríe tras un acierto.

8.3. Soporte multiplataforma y móvil

Actualmente, el juego se ejecuta en PC. Una línea de trabajo futura es la migración a arquitecturas móviles (Android/iOS).

- **Optimización TFLite**: Convertir los modelos entrenados al formato TensorFlow Lite para permitir la inferencia en dispositivos con recursos limitados (Edge Computing).
- **Sensores inerciales**: En un entorno móvil, se podrían fusionar los datos de la cámara frontal con los del giroscopio y acelerómetro del teléfono para detectar gestos de inclinación de la cabeza con mayor precisión.

8.4. Multijugador online (WebRTC)

Romper la barrera local y permitir que dos jugadores compitan desde ubicaciones remotas.

- **Arquitectura cliente-servidor**: Implementar un servidor de señalización (usando WebSockets) y protocolos WebRTC para transmitir el vídeo y los datos de la partida en tiempo real con baja latencia.
- **Competitividad**: Crear un ranking global de puntuaciones almacenado en la nube.

9. Conclusiones

El desarrollo de *Simón Dice* ha servido para validar una premisa fundamental en la ingeniería de software actual: no siempre es necesario recurrir a la potencia de cálculo bruta para resolver problemas complejos. A veces, un diseño arquitectónico inteligente es mucho más efectivo.

Al finalizar este proyecto, podemos extraer tres conclusiones principales que definen el éxito del sistema:

1. **La eficiencia de la arquitectura híbrida:** La decisión de desacoplar el problema en dos fases —utilizando MediaPipe para la extracción geométrica y un perceptrón multicapa (MLP) para la clasificación— ha demostrado ser una estrategia superior al uso de redes convolucionales profundas (CNN) para este caso de uso. Hemos logrado una inferencia en tiempo real robusta (> 30 FPS) en ordenadores portátiles estándar, democratizando el acceso a la aplicación sin depender de tarjetas gráficas dedicadas.

2. **Las matemáticas como herramienta de robustez:** El preprocesamiento de los datos, específicamente la normalización y centrado de coordenadas, resultó ser la clave técnica del proyecto. Gracias a esto, el sistema es prácticamente "ciego" al entorno: funciona con la misma precisión si el usuario se encuentra cerca o lejos de la cámara, o si cambia su posición en el encuadre, resolviendo uno de los problemas clásicos de la visión artificial: la invarianza a la escala y traslación.

3. **La tecnología al servicio de la experiencia:** Más allá de la precisión algorítmica, la integración de feedback multimodal (instrucciones por voz y respuesta visual inmediata) transformó un detector de gestos técnico en una experiencia lúdica fluida. Esto demuestra el potencial de las interfaces naturales de usuario (NUI) para crear software más accesible e inmersivo.

10. Bibliografía

- Lugares, C., & Zhang, F. (2019). *MediaPipe: A Framework for Building Perception Pipelines*. Google Research.
- Kartynnik, Y., et al. (2019). *Real-time Facial Surface Geometry from Monocular Video on Mobile Devices*. arXiv preprint arXiv:1907.06744.
- Scikit-Learn Documentation: `https://scikit-learn.org/stable/`
- OpenCV Documentation: `https://docs.opencv.org/`