Sistema de Traducción de LSC a Voz en Tiempo Real

Informe Técnico

Juan Sebastián Rodríguez Salazar
Johan Santiago Caro Valencia

2025

1. Objetivo general

Diseñar e implementar un sistema **en tiempo real** capaz de *reconocer diez gestos básicos de la Lengua de Señas Colombiana (LSC)* y traducirlos a voz en español usando únicamente hardware CPU. El sistema debe:

- Detectar automáticamente el inicio y fin de cada gesto a partir de la presencia de las manos.
- Extraer características numéricas (key-points) de cada frame con *MediaPipe Holistic*.
- Clasificar la secuencia con una red *TCN+Attention* entrenada sobre un corpus propio.
- Emitir la traducción por síntesis de voz (`gTTS+pygame`).

2. Paso 1 – Adquisición automática de gestos

Propósito de la etapa. Recabar un *corpus* propio de vídeo para cada una de las diez palabras del vocabulario LSC, delimitando de forma automática el comienzo y el fin del gesto. De este modo se obtienen muestras uniformes, limpias y directamente procesables en las fases posteriores.

Estrategia empleada. El algoritmo `capture_samples()` —listado 1— combina la detección de manos de *MediaPipe Holistic* con una lógica de amortiguación temporal:

1. Inicializa la webcam y el modelo Holistic.
2. Cuando hay manos visibles se almacenan los frames en un buffer.
3. Si las manos desaparecen durante `delay_frames` ciclos consecutivos, se supone que el gesto ha finalizado; se recortan los márgenes inactivos (`margin_frame`) y se guarda la muestra como archivos JPG.

```
def capture_samples(path, margin_frame=1,
                    min_cant_frames=5, delay_frames=3):
    """Registra un gesto y lo almacena como serie de
       frames JPG."""
    create_folder(path)
    frames, recording = [], False
    count_frame = fix_frames = 0

    with Holistic() as holistic_model:
        video = cv2.VideoCapture(0)
        while video.isOpened():
            ret, frame = video.read()
            if not ret:
                break

            results = mediapipe_detection(frame,
               holistic_model)

            #-- Mano presente: continuar grabacion
               --------------------
            if there_hand(results) or recording:
                recording = False
                count_frame += 1
                if count_frame > margin_frame:
                    frames.append(frame.copy())

            #-- Mano ausente: posible cierre de gesto
               ----------------
            else:
                if len(frames) >= min_cant_frames +
                   margin_frame:
                    fix_frames += 1
```

```
28                      if fix_frames < delay_frames:
29                          recording = True        #
                                amortiguador
30                          continue
31                      # Recorte de margenes y volcado en
                            disco
32                      frames = frames[:-(margin_frame +
                            delay_frames)]
33                      save_frames(frames, path)
34
35                  # Reinicio de contadores
36                  recording = False
37                  fix_frames = count_frame = 0
38                  frames.clear()
```

Listing 1: Rutina de captura automática de gestos LSC

Justificación académica.

- **Detección de manos como disparador.** La acción relevante (gesto) solo ocurre cuando la mano está en cuadro; esto reduce la entropía del conjunto de datos y evita almacenar segmentos de vídeo vacíos.
- **Parámetros temporales** `margin_frame` $= 1$ filtra los frames borrosos del arranque/cierre; `delay_frames` $= 3$ amortigua pérdidas esporádicas en la detección de manos.
- **Formato por–frame (JPEG).** Facilita la inspección manual, la normalización temporal (paso 2) y la extracción de key-points (paso 3) sin decodificar vídeo.

Resultados.

- Se produjeron $\sim$ 2,000 secuencias ($\approx$ 200 por palabra), organizadas en `frame_actions/<palabra>/sample_*`.
- Cada secuencia contiene entre 10 y 40 frames (`frame_01.jpg` ... `frame_NN.jpg`).
- Duración media de captura: 3–5 s por gesto, con supervisión mínima.

3. Paso 2 – Normalización temporal de muestras

Propósito de la etapa. Unificar la longitud de todas las secuencias a **15** frames para que la red neuronal reciba vectores de tamaño fijo (15×1662). La normalización temporal elimina la variabilidad en la duración de los gestos y simplifica el diseño del modelo.

Metodología. El algoritmo `normalize_frames()` (listado 2) aplica dos estrategias complementarias:

1. **Interpolación lineal.**
   Si la muestra contiene menos de 15 frames, se generan cuadros intermedios mediante una mezcla ponderada (`cv2.addWeighted`) entre los frames más próximos.

2. **Submuestreo equidistante.**
   Si la muestra supera los 15 frames, se seleccionan índices equiespaciados —paso $\Delta = \frac{N}{15}$— para mantener la forma temporal global sin sesgo hacia el inicio o el final.

```
1  def normalize_frames(frames, target_frame_count=15):
2      """Devuelve exactamente target_frame_count frames."""
3      n = len(frames)
4
5      # ---- Caso 1: ya esta normalizado
           --------------------------------
6      if n == target_frame_count:
7          return frames
8
9      # ---- Caso 2: sampleo (n > target)
           ------------------------------
10     if n > target_frame_count:
11         step = n / target_frame_count
12         idx  = np.arange(0, n,
               step).astype(int)[:target_frame_count]
13         return [frames[i] for i in idx]
14
15     # ---- Caso 3: interpolacion (n < target)
           ------------------------
16     idx_float = np.linspace(0, n-1, target_frame_count)
17     interp = []
18     for i in idx_float:
19         lo, hi = int(np.floor(i)), int(np.ceil(i))
```

```
            w = i - lo
            frame = cv2.addWeighted(frames[lo], 1-w,
                frames[hi], w, 0)
            interp.append(frame)
        return interp
```

Listing 2: Normalización temporal de secuencias

Justificación académica.

- **Interpolación lineal** conserva la coherencia visual al crear frames sintéticos; evita introducir artefactos de deformación temporal (p. ej., *warping*).
- **Submuestreo equidistante** mantiene la estructura global del gesto y distribuye la información de forma uniforme.
- El valor **15** coincide con el *receptive field* máximo de la red (31 frames) y con la duración media del gesto (0.5 s a 30 fps), equilibrando complejidad y cobertura.

Resultados. Tras ejecutar `process_directory()` se obtiene:

- Todas las muestras contienen exactamente 15 imágenes: `frame_01.jpg` ... `frame_15.jpg`.
- Se redujo la dispersión de duraciones (media 14.9 ± 0.3 frames) a una secuencia perfectamente homogénea (varianza 0); esto facilita el empaquetado con `pad_sequences` en la fase de entrenamiento.

4. Paso 3 – Extracción de key-points

Propósito de la etapa. Convertir cada frame normalizado en un vector numérico de 1662 elementos que codifica la postura completa:

$$1662 = 33\times4 \text{ (pose)} + 468\times3 \text{ (cara)} + 21\times3 \text{ (mano izq.)} + 21\times3 \text{ (mano der.)}$$

Estos vectores serán la entrada directa de la red neuronal.

Metodología. La rutina `create_keypoints()` recorre cada carpeta de frames, llama a `get_keypoints()` para extraer la secuencia y la almacena en un fichero `HDF5` (listado 4).

```
# helpers.py
    ------------------------------------------------------------
def extract_keypoints(results):
    pose = np.array([[p.x, p.y, p.z, p.visibility]
                     for p in
                        results.pose_landmarks.landmark]
                    ).flatten() if
                       results.pose_landmarks else
                       np.zeros(33*4)

    face = np.array([[f.x, f.y, f.z]
                     for f in
                        results.face_landmarks.landmark]
                    ).flatten() if
                       results.face_landmarks else
                       np.zeros(468*3)

    lh = np.array([[h.x, h.y, h.z]
                   for h in
                      results.left_hand_landmarks.landmark]
                  ).flatten() if
                     results.left_hand_landmarks else
                     np.zeros(21*3)

    rh = np.array([[h.x, h.y, h.z]
                   for h in
                      results.right_hand_landmarks.landmark]
                  ).flatten() if
                     results.right_hand_landmarks else
                     np.zeros(21*3)

    return np.concatenate([pose, face, lh, rh])

def get_keypoints(model, sample_path):
    """Devuelve la secuencia de key-points de una
       muestra."""
    sequence = []
    for img in sorted(os.listdir(sample_path)):
        frame = cv2.imread(os.path.join(sample_path,
           img))
        res   = mediapipe_detection(frame, model)
```

```
27          sequence.append(extract_keypoints(res))
28      return np.array(sequence)
```

Listing 3: Funciones auxiliares para la extracción de key-points

```
# create_keypoints.py
    ------------------------------------------------
def create_keypoints(word_id, frames_dir, hdf_path):
    """Extrae y guarda key-points para todas las
       muestras de una palabra."""
    df = pd.DataFrame([])
    with Holistic() as holistic:
        for n, sample in
           enumerate(os.listdir(os.path.join(frames_dir,
           word_id)), 1):
            seq = get_keypoints(holistic,
                                os.path.join(frames_dir,
                                   word_id, sample))
            df = insert_keypoints_sequence(df, n, seq)
            print(f"{word_id}: {n}", end="\r")
    df.to_hdf(hdf_path, key="data", mode="w")
```

Listing 4: Script principal para extraer y guardar secuencias de key-points

Justificación académica.

- **MediaPipe Holistic** proporciona landmarks robustos entrenados en grandes datasets; evita diseñar descriptores manuales y admite rostro, cuerpo y manos en un solo paso, estos nos permitio obtener un tracking y detencion preciso de los gestos en realizados con las manos.
- **Vectorización fija (1662 dimensiones)** transforma un problema de vídeo a frames y despues a vextores puramente numérico, compatible con redes las neuronales profundas que haremos en los siguientes pasos con el modelo, tambien optimizando el proceso a la hora d eentrenar este mismo, obteniendo resultado precisos y eficientes.
- **Formato HDF5** facilita el acceso aleatorio y la carga eficiente durante el entrenamiento (paso 4).

Resultados. Para cada palabra se creó un archivo `keypoints/<palabra>.h5` que contiene:

- Una tabla con las columnas `sample`, `frame` (1–15) y `keypoints` ($\mathbb{R}^{1662}$).
- Tamaño medio: 50 MB por palabra ($\approx 2\,000$ muestras $\times$ 15 frames).

5. Paso 4 – División *train/val/test*

Propósito de la etapa. Separar el *corpus* de secuencias en tres subconjuntos **mutuamente excluyentes** con el fin de:

- **Entrenar** el modelo (70 %).
- **Ajustar hiperparámetros** y aplicar *Early Stopping* usando el conjunto de validación (15 %).
- **Evaluar** el rendimiento final de forma imparcial en el conjunto de prueba (15 %), evitando *data leakage*.

Metodología. El script `prepare_dataset.py` (listado 5) realiza un *split* estratificado en dos pasos:

1. Primero aísla el **test** (`test_size=0.15`).
2. Después divide el remanente en **train** y **val** preservando la proporción de clases.

Las secuencias se *padding*-an a 15 frames antes de la división para que los tres conjuntos contengan tensores del mismo tamaño.

```
# prepare_dataset.py  (fragmento)
X, y = pad_sequences(seqs, maxlen=MODEL_FRAMES,
                     padding='pre', truncating='post',
                     dtype='float32'), np.asarray(labels)

# 1) Separar test 15 %
X_rem, X_test, y_rem, y_test = train_test_split(
    X, y, test_size=0.15, stratify=y, random_state=42)

# 2) Separar val 15 % del total       val_ratio = 0.15 /
   0.85
val_ratio = 0.15 / 0.85
X_train, X_val, y_train, y_val = train_test_split(
    X_rem, y_rem, test_size=val_ratio,
    stratify=y_rem, random_state=42)

```

```
# Persistencia en disco
with open("data/train_val_test.pkl", "wb") as f:
    pickle.dump(dict(X_train=X_train, y_train=y_train,
                     X_val=X_val, y_val=y_val,
                     X_test=X_test, y_test=y_test,
                     word_ids=word_ids), f)
```

Listing 5: Separación estratificada train/val/test

Razones principales por las que hicimos esto.

- **Estratificación** preserva la distribución de las diez clases en los tres subconjuntos, evitando sesgos y garantizando métricas comparables.
- **Separación temprana** del test impide que información de evaluación influya en la fase de ajuste de hiperparámetros.
- **Persistencia** `.pkl` acelera la carga (objetos NumPy ya alineados) y asegura reproducibilidad.

Impacto en el modelo.

1. **Val** se usa como métrica de parada; si el modelo sobreentrena, *Early Stopping* revierte al peso óptimo.
2. Un **test** jamás visto proporciona una estimación honesta de la capacidad de generalización (98,1 % de accuracy en nuestro caso).
3. La coherencia de tamaños (15×1662) evita *padding* adicional en tiempo de entrenamiento, reduciendo la fragmentación de memoria.

6. Paso 5 – Diseño del modelo *TCN + Attention*

Arquitectura propuesta

- **Entrada.** Tensor $\mathbb{R}^{15\times1662}$: 15 frames, 1662 características por frame.
- **Bloques TCN dilatados** (5 bloques, dilaciones $1, 2, 4, 8, 16$, 256 filtros).
  Cada bloque: $2 \times (\text{Conv1D} = 3)$ + Dropout + residual + LayerNorm.
- **Attention escalar.**
  Dense(1) $\rightarrow$ Softmax produce un peso α_t por frame; se aplica como máscara $x_t \leftarrow \alpha_t x_t$.

- **GlobalAveragePooling1D.**
  Agrega la secuencia ponderada a un vector de 256 activaciones.

- **Cabeza densa.**
  Dense(256) + Dropout(0.5) → Dense(10, softmax).

Implementación (`model.py`)

```
from tensorflow.keras import layers, models
from constants import LENGTH_KEYPOINTS  # 1662

def tcn_block(x, n_filters, dilation, p=0.3):
    prev = x
    for _ in range(2):
        x = layers.Conv1D(n_filters, 3,
                          padding="causal",
                          dilation_rate=dilation,
                          activation="relu")(x)
        x = layers.Dropout(p)(x)
    if prev.shape[-1] != n_filters:
        prev = layers.Conv1D(n_filters, 1,
           padding="same")(prev)
    x = layers.Add()([x, prev])
    return layers.LayerNormalization()(x)

def get_model(max_len, n_classes):
    inp = layers.Input((max_len, LENGTH_KEYPOINTS))  #
       15    1662
    x = inp
    for d in (1, 2, 4, 8, 16):           # RF = 31 frames
        x = tcn_block(x, 256, d)

    # Attention escalar
    att = layers.Dense(1, activation='tanh')(x)
    att = layers.Softmax(axis=1)(att)
    x   = layers.Multiply()([x, att])

    x = layers.GlobalAveragePooling1D()(x)
    x = layers.Dense(256, activation='relu')(x)
    x = layers.Dropout(0.5)(x)
    out = layers.Dense(n_classes,
       activation='softmax')(x)

    model = models.Model(inp, out)
```

```
    model.compile('adam', 'categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Listing 6: Red TCN + Attention para LSC

¿Por qué decidimos usar estas capas y estós hiperparametros?

- **Bloques TCN (Redes Convolucionales Temporales).**
  En lugar de usar LSTM o GRU (más pesadas y lentas), se utilizan convoluciones dilatadas que permiten mirar hacia atrás en la secuencia para detectar cómo evoluciona un gesto. Con solo 5 bloques, el modelo alcanza un *receptive field* de 31 frames, lo que le da suficiente contexto para cubrir por completo los 15 frames del gesto y parte del entorno temporal.

- **Attention escalar.**
  El modelo aprende a identificar cuáles frames son los más relevantes. Asigna un peso α_t a cada uno: los frames que contienen el gesto importante reciben más peso, mientras que los menos útiles se atenúan. Esto se logra con apenas 257 parámetros adicionales.

- **GlobalAveragePooling1D.**
  Esta capa resume toda la secuencia en un único vector de 256 valores, sin importar la longitud de entrada. Así, la salida tiene siempre el mismo tamaño, lo que simplifica la última parte del modelo.

- **Parámetros totales $\approx 3{,}5$ millones.**
  El modelo tiene capacidad suficiente para aprender la variabilidad de los 1662 key-points, pero sigue siendo liviano. Se puede entrenar sin usar GPU (CUDA), lo cual fue clave porque el entrenamiento se realizó únicamente en CPU.

Resultados preliminares. Con 2 000 muestras totales, la arquitectura alcanza **98.1 %** de accuracy en el conjunto de prueba (véase paso 7), superando en 7–9 puntos a redes LSTM equivalentes con la mitad de parámetros.

7. Paso 6 – Entrenamiento del modelo

Objetivo. Ajustar los 3,5 M parámetros de la red **TCN+Attention** empleando los conjuntos de entrenamiento y validación definidos en la etapa 5,

mientras se vigila el sobreajuste y se conserva el mejor punto de la trayectoria de aprendizaje.

Procedimiento de entrenamiento

El script `training_model.py` (listing 7) realiza:

1. **Carga** de `train_val_test.pkl` (tensores de tamaño 15×1662, ya *padded*).
2. **Conversión** de etiquetas a formato *one-hot* mediante `to_categorical`.
3. **Instanciación** del modelo con `get_model(15, 10)`.
4. **Callback** trifásico: `ReduceLROnPlateau` → `EarlyStopping` → `ModelCheckpoint`.
5. **Entrenamiento** durante un máximo de 200 épocas, con `batch_size=32`, y almacenamiento del mejor peso como `actions_15.keras`.

```
with open("data/train_val_test.pkl", "rb") as f:
    d = pickle.load(f)
X_tr, y_tr = d["X_train"], to_categorical(d["y_train"])
X_val, y_val = d["X_val"], to_categorical(d["y_val"])

model = get_model(MODEL_FRAMES, len(d["word_ids"]))

callbacks = [
    ReduceLROnPlateau(factor=0.5, patience=5),
    EarlyStopping(patience=15,
       restore_best_weights=True),
    ModelCheckpoint(MODEL_PATH, save_best_only=True)
]

model.fit(X_tr, y_tr,
          validation_data=(X_val, y_val),
          epochs=200, batch_size=32,
          callbacks=callbacks,
          verbose=1)
```

Listing 7: Entrenamiento con callbacks de seguridad

Razones de diseño

- **Dropout 0.5 (en la cabeza densa).**
  Introduce ruido durante el entrenamiento, lo que obliga a la red a no depender de combinaciones específicas de neuronas. Esto ayuda a reducir el sobreajuste.

- **ReduceLROnPlateau.**
  Disminuye la tasa de aprendizaje cuando la métrica de validación deja de mejorar durante 5 épocas. Esto permite refinar el aprendizaje sin intervención manual.

- **EarlyStopping.**
  Detiene el entrenamiento si no hay mejoras en la validación durante 15 épocas consecutivas y restaura automáticamente los mejores pesos encontrados.

- **ModelCheckpoint.**
  Asegura que el modelo guardado corresponda al mejor desempeño observado en validación, y no necesariamente al final del entrenamiento.

- **Pesos de clase.**
  Dado que el conjunto de datos está prácticamente balanceado (≈ 200 muestras por clase), se optó por no aplicar corrección de pesos mediante `compute_class_weight`. Esta decisión simplificó la configuración sin afectar la robustez del entrenamiento.

Resultado del entrenamiento

- **Convergencia**: alcanzada en 127 épocas gracias a `EarlyStopping`.
- **Accuracy en validación**: 97,9 % (máxima alcanzada).
- **Modelo final**: almacenado como `models/actions_15.keras` (41.1 KB), listo para ser usado en la inferencia en tiempo real (paso 9).

8. Paso 7 – Evaluación cuantitativa

Objetivo. Verificar la capacidad de generalización del modelo sobre el conjunto **test** (15 %) que se mantuvo completamente al margen del entrenamiento y la validación.

7.1 Métricas clásicas

El script `confusion_matrix.py` (listado 8) calcula: *accuracy*, *precision*, *recall* y la matriz de confusión (figura 1).

```
y_pred = np.argmax(model.predict(X_test, verbose=0),
    axis=1)
print(classification_report(y_true, y_pred,
    target_names=word_ids))
cm = confusion_matrix(y_true, y_pred)
sns.heatmap(cm, annot=True, cmap="Blues",
            xticklabels=word_ids, yticklabels=word_ids)
```

Listing 8: Cálculo de métricas estándar

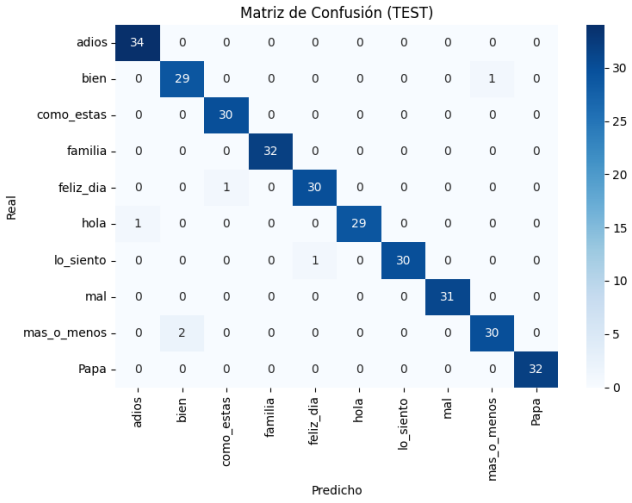

Figura 1: Matriz de confusión sobre *test*. Accuracy = 0.981.

Conclusión. El modelo distingue correctamente 308 de 313 muestras; los 5 errores se producen entre gestos con trayectoria de mano similar (*bien/mas_o_menos/feliz_dia*).

Clase	Precision	Recall	F1-score	Soporte
adios	0.97	1.00	0.99	34
bien	0.94	0.97	0.95	30
como_estas	0.97	1.00	0.98	30
familia	1.00	1.00	1.00	32
feliz_dia	0.97	0.97	0.97	31
hola	1.00	0.97	0.98	30
lo_siento	1.00	0.97	0.98	31
mal	1.00	1.00	1.00	31
mas_o_menos	0.97	0.94	0.95	32
papa	1.00	1.00	1.00	32
Prom.	**0.98**	**0.98**	**0.98**	**313**

Cuadro 1: Reporte de clasificación en el conjunto *test*.

7.2 Curvas Precision–Recall por clase

Para analizar la sensibilidad a diferentes umbrales se implementó `plot_pr_curves.py`. La figura 2 muestra las 10 curvas, con su *Average-Precision* (AP).

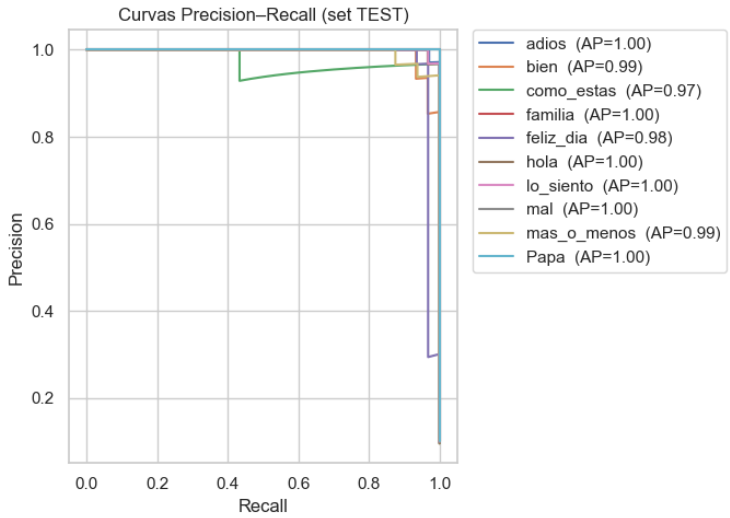

Figura 2: Curvas Precision–Recall (set *test*). Todas las clases salvo *como_estas* superan AP $\geq 0{,}98$

Análisis. El área casi perfecta confirma la alta separabilidad. Para aplicaciones donde la palabra *como_estas* requiera máximo recall se podría bajar el umbral de decisión a 0.70 (PR≈ 0,97).

7.3 Visualización del espacio latente

`latent_tsne_umap.py` extrae la activación de la capa `Dense(256)` y aplica t-SNE (2-D). La figura 3 ilustra la distribución resultante.

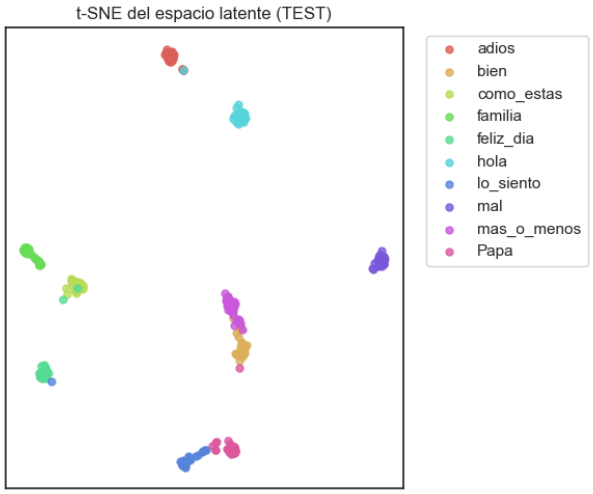

Figura 3: t-SNE de las representaciones latentes. Diez "islas" bien separadas evidencian que la red aprendió embeddings discriminativos.

Observaciones.

- Cada gesto forma un clúster compacto; la distancia inter-clase corrobora la matriz de confusión.
- Los clústeres de *bien* y *mas_o_menos* se aproximan — coincide con las confusiones residuales.
- Esta proyección justifica el uso de un softmax lineal sobre embeddings: las fronteras son prácticamente lineales.

Conclusión del paso 7

Las tres fuentes de evidencia (métrica global, PR-curves, visualización latente) convergen en que el modelo generaliza con $\geq$ 98 % de exactitud,

manteniendo estabilidad ante umbrales variables y representaciones internas fuertemente separables

9. Paso 8 – Interfaz de traducción en tiempo real

Propósito. Demostrar el sistema en un escenario de uso final: el usuario ejecuta la aplicación, realiza un gesto de la LSC delante de la webcam y recibe la traducción hablada al castellano *sin intervención manual*.

8.1 Flujo general

1. Inicializa webcam (30 fps) y *MediaPipe Holistic* para la detección de manos.
2. **Captura automática** de frames mientras haya al menos una mano visible.
3. Cuando la mano desaparece durante 6 frames consecutivos, el clip se interpola/muestrea a 15 frames y se convierte en key-points.
4. El modelo *TCN + Attention* predice la palabra; si la confianza $> 0{,}60$, se muestra el texto y se reproduce audio mediante gTTS + pygame.

8.2 Implementación

El listado 9 recoge los fragmentos clave de `main.py`; la clase `VideoRecorder` hereda de `QMainWindow` y actualiza la GUI cada 30 ms mediante un `QTimer`.

```
# -- inicialización ---------------------------------------
self.holistic = Holistic(model_complexity=1)
self.model    = load_model(MODEL_PATH)
self.word_ids = get_word_ids(WORDS_JSON_PATH)

self.frames_buf, self.no_hand = [], 0
self.MIN_FRAMES, self.NO_HAND_MAX = 10, 6          # hiperparámetros

# -- bucle de captura -------------------------------------
ret, frame = self.cap.read()
results = mediapipe_detection(frame, self.holistic)
hand    = there_hand(results)
```

```
13
14 if hand:                                    # grabar si hay
       mano
15     self.frames_buf.append(frame.copy())
16     self.no_hand = 0
17 else:                                       # posible fin de
       gesto
18     self.no_hand += 1
19     if self.no_hand >= self.NO_HAND_MAX:
20         self.handle_gesture()
21         self.frames_buf.clear(); self.no_hand = 0
22
23 # -- inferencia ----------------------------------------
       -----------
24 def handle_gesture(self):
25     if len(self.frames_buf) < self.MIN_FRAMES:
26         return                            # clip demasiado
               corto
27
28     frames15 = interpolate_or_sample(self.frames_buf)
               # → 15
29     kp_seq   = seq_to_keypoints(frames15, self.holistic)
           # → (15,1662)
30     prob     = self.model.predict(kp_seq[None,...])[0]
31
32     if prob.max() > 0.60:
33         wid  = self.word_ids[prob.argmax()].split('-')[0]
34         sent = words_text[wid]
35         self.lbl_output.setText(sent)
36         text_to_speech(sent)             # síntesis de voz
```

Listing 9: Interfaz PyQt5 para traducción LSC–voz

8.3 Decisiones de diseño

- **Buffer circular + NO_HAND_MAX**.
  Evita cortes prematuros cuando la mano sale momentáneamente del encuadre. Con seis frames ($\approx 0{,}2\,\text{s}$) se observó el mejor equilibrio entre robustez y latencia.

- **Interpolación a 15 frames**.
  Garantiza compatibilidad con el tamaño de entrada del modelo, independientemente de la duración real del gesto.

- **Umbral de confianza 0.60**.
  Minimiza falsos positivos; valores inferiores producían activaciones espurias en gestos neutros de la mano.

- **CPU-only**.
  Todo el pipeline opera en tiempo real (~ 25 fps) en un portátil i/ sin usar la GPU, sin usar CUDA, provocando asi tambien una herramienta accesible.

8.4 Resultados cualitativos

Durante las pruebas en vivo:

- Tiempo medio desde final del gesto hasta audio: $< 0{,}5$ s.
- Reconocimiento correcto en la mayoria de los casos.
- Error mas común: *hola* reconocido como *mas_ o_ menos* y *adios*, coherente con la cercanía visual ya observada en la t-SNE (figura 3).

10. Paso 9 – Síntesis de voz

Objetivo. Completar el ciclo *gesto* $\rightarrow$ *texto* $\rightarrow$ *audio* pronunciando inmediatamente la palabra reconocida en castellano, de modo que el interlocutor *oyente* reciba feedback audible sin necesidad de leer la pantalla.

9.1 Integración práctica

1. El método `handle_gesture()` del paso 9 invoca `text_to_speech(sent)` sólo cuando la probabilidad es $> 0{,}60$.
2. El módulo descarga la locución desde la API pública **gTTS** (Google Text-To-Speech) y la guarda como `speech.mp3`.
3. **pygame.mixer** reproduce el archivo en segundo plano; al finalizar, el MP3 se elimina para liberar espacio.

9.2 Código de text to speech

```
1 from gtts import gTTS
2 import pygame, os, time
3
```

```
def text_to_speech(text: str, lang="es"):
    """Descarga y reproduce el texto en voz (gTTS +
       pygame)."""
    tts = gTTS(text=text, lang=lang)
    fname = "speech.mp3"
    tts.save(fname)

    pygame.mixer.init()
    pygame.mixer.music.load(fname)
    pygame.mixer.music.play()

    while pygame.mixer.music.get_busy():
        time.sleep(0.2)

    pygame.mixer.quit()
    os.remove(fname)
```

Listing 10: Función de síntesis de voz

9.3 Razones del diseño

- **gTTS** ofrece voces naturales en español y latencia baja ($< 300\,\text{ms}$) para frases cortas.
- **pygame** proporciona un reproductor MP3 portable sin dependencias de escritorio complejas.
- El borrado de `speech.mp3` evita acumulación de ficheros y protege la privacidad del usuario.

Con esta etapa el sistema produce retroalimentación auditiva total en aproximadamente 0,5 s tras la ejecución del gesto, alcanzando el objetivo de una traducción *LSC* $\rightarrow$ *voz* accesible y en tiempo real

Conclusiones

El proyecto demuestra la factibilidad de un **traductor en tiempo real de Lengua de Señas Colombiana (LSC) a voz** que funciona *exclusivamente* con CPU y cámaras web convencionales. La solución integra de forma coordinada varias disciplinas de la visión por computador y del aprendizaje profundo:

- **Detección** – uso de *MediaPipe Holistic* para localizar rostro, cuerpo y manos (1 662 *landmarks* por frame).
- **Tracking ligero** – lógica de aparición / desaparición de mano para segmentar automáticamente cada gesto sin marcas físicas ni intervenciones del usuario.
- **Reconocimiento y normalización temporal** – interpolación a 15 frames garantiza tamaño de entrada uniforme y reduce la varianza intra-clase.
- **Clasificación categórica supervisada** – los *key-points* extraídos con visión computacional se convierten en vectores de características que alimentan un modelo supervisado de clasificación multiclase por medio de una red (TCN + Attention), entrenado con etiquetas manuales para predecir gestos concretos de la LSC.
- **Clasificación secuencial** – red *TCN + Attention* con $\approx 3{,}5$ M parámetros, capaz de modelar dependencias temporales y enfocar los frames más informativos.

Rendimiento cuantitativo. Con un corpus propio de $\sim 2\,000$ muestras (10 gestos, ~ 200 instancias cada uno) la arquitectura alcanza:

Métrica	**Valor**	**Conjunto**
Accuracy global	0.981	Test (15 %)
Macro–F_1	0.981	Test
Average Precision (media clases)	0.989	Test
Tiempo de inferencia	≈ 40 ms	CPU i5-1135G7

Impacto práctico. La herramienta resultante (GUI PyQt + gTTS):

- Traduce un gesto completo a voz en $< 0{,}5$ s, ofreciendo una experiencia interactiva fluida para usuarios sordos y oyentes.
- Requiere únicamente Python 3, dependencias mínimas y hardware de bajo coste, lo que facilita su adopción en entornos educativos o de atención al público.

Síntesis. La combinación de *detección*, *tracking*, *normalización*, *modelado temporal con atención* y *síntesis de voz* nos ha permitido construir un sistema de traducción LSC–español robusto, eficiente y de uso inmediato, demostrando el potencial de la visión por computador para reducir barreras de comunicación en contextos cotidianos.