Universitat d'Alacant
Universidad de Alicante

INGENIERÍA ROBÓTICA

VISIÓN POR COMPUTADOR

Proyecto

Reconocimiento de acciones

Autores: Miguel Fernández Lorenzo, Ekaitz Duque Fernández y Daniel Azorín Martínez

Índice

Lista de Figuras

Lista de Códigos

1 Introducción

Este proyecto tuvo distintos enfoques que se fueron explorando durante su desarrollo. Inicialmente, se partió con la ambiciosa tarea de implementar un traductor en tiempo real del lenguaje de señas. Sin embargo, tras observar la complejidad que eso supone se decidió reconducir el enfoque hacia la evaluación comparativa del rendimiento de diferentes modelos. En este contexto, se optó por emplear un único conjunto de datos para entrenar modelos de redes neuronales recurrentes (RNN) y modelos ocultos de Markov (HMM).

Durante la experimentación, se observó que la aplicación del HMM generaba muchos errores. Tras un largo periodo de pruebas se decidió centrarse exclusivamente en el desarrollo y entrenamiento del modelo basado en RNN, ya que proporcionaba resultados más prometedores y consistentes en términos de precisión y eficacia, además de la comodidad que suponía trabajar únicamente con este modelo.

2 Dataset

El Dataset usado se puede encontrar en [4]. El conjunto de datos fue capturado utilizando el sensor *Leap Motion*, que extrae datos en 3D relacionados con las manos, muñecas y antebrazos, generando así nubes de puntos. Todos los datos obtenidos para cada gesto están normalizados en un rango de [0,1]. La normalización es un paso común de preprocesamiento para llevar todas las características a una escala similar. El Dataset está dividido en dos carpetas principales:

- **"separated_gestures":** Contiene 91 gestos diferentes, cada uno con 40 muestras en formato *.csv*, sumando un total de 3640 archivos. Corresponde con el conjunto de datos utilizado en este proyecto.
- **"consecutive_gestures":** Contiene gestos consecutivos en lenguaje de señas español, con 274 archivos.

Además, cada archivo contiene 42 columnas que representan variables dependientes del tiempo para ambas manos (21 variables para cada mano). Las variables incluyen componentes de dirección, ángulos de inclinación, ángulos de giro, y direcciones específicas de los dedos, cuando se realiza un gesto con una sola mano (siempre la mano derecha), los datos de una mano se duplican, para obtener las 42 columnas totales.

Cabe destacar que el número de filas de cada muestra varía en función de si el gesto es más largo o más corto y de la rapidez en realizarlo, esto es importante debido a que en el entrenamiento se necesita indicar el tamaño de entrada de los datos a la red neuronal, por lo que se ha tenido que realizar un procesamiento de los datos previo al entrenamiento.

2.1 Procesamiento de datos

El procesamiento de los datos previo al entrenamiento fue clave, para ello se mejoró la estructura del Dataset organizando las 40 muestras correspondientes a cada acción en carpetas con los respectivos

nombres de las acciones. Posteriormente se elaboró el Código 1, cuyo funcionamiento se va a explicar detalladamente, comenzando por las librerías usadas:

- **os:** Proporciona una interfaz para interactuar con el sistema operativo, en este caso, se utiliza para manejar rutas de archivos.
- **pandas:** Librería para manipulación y análisis de datos. Se usa para leer archivos CSV y almacenar datos en estructuras de datos llamadas DataFrames.
- **numpy:** Librería para realizar operaciones matriciales y numéricas eficientes.
- **train_test_split:** Función de scikit-learn para dividir conjuntos de datos en conjuntos de entrenamiento y prueba.
- **LabelEncoder:** De scikit-learn, se utiliza para convertir etiquetas de texto en números.

A continuación, en la línea 7 se creó la lista de acciones que corresponden con las 91 carpetas contenedoras de las 40 muestras de datos. En la línea 9 se encuentra la ruta de acceso al Dataset y, por último, en las líneas 11 y 12 se crean las listas X_list e y_list respectivamente, las cuales contendrán los siguientes datos:

- **X_list:** los datos de las acciones.
- **y_list:** las etiquetas de las acciones.

El bucle siguiente recorre cada acción y cada uno de los 40 archivos *.csv* asociados a esa acción y guarda los datos (matriz) en **X_list** y la etiqueta en **y_list**.

Como se ha comentado en la explicación del Dataset, es necesario conseguir que los datos tengan el mismo tamaño, para ello se ha obtenido el tamaño de la muestra de datos con más filas, es decir, de la muestra con más datos de todas y se ha guardado en la variable **max_length**. Después, se rellena cada muestra de datos guardada en **X_list** con valores nulos hasta que tenga el mismo número de datos que **max_length**, para, finalmente, convertirlas en matrices numpy.

Seguidamente se crea un objeto LabelEncoder y se utiliza para transformar las etiquetas de texto en valores numéricos y, por último, se dividen los datos en conjuntos de entrenamiento (**X_train, y_train**) y prueba (**X_test, y_test**) con una proporción del 80% entrenamiento y 20% prueba.

```
1 import os
2 import pandas as pd
3 import numpy as np
4 from sklearn.model_selection import train_test_split
5 from sklearn.preprocessing import LabelEncoder
6
```

```
actions = np.array(['a', 'adios', 'amarillo', 'apellido', 'aprender', 'azul', 'bien', '
    buenasnoches', 'buenastardes', 'buenosdias', 'camisa', 'catorce', 'cinco', 'color', '
    cuatro', 'dar', 'de', 'decir', 'denada', 'diecinueve', 'dieciocho', 'dieciseis', '
    diecisiete', 'diez', 'doce', 'donde', 'dos', 'el', 'ella', 'ensenar', 'entender', '
    espana', 'estudiar', 'euro', 'gracias', 'gris', 'gustar', 'haber', 'hastaluego', '
    hermano', 'hola', 'madre', 'mal', 'marron', 'morado', 'mucho', 'muchos', 'nacer', 'negro
    ', 'nino', 'no', 'nohaber', 'nombre', 'nosaber', 'nosotras', 'nosotros', 'nueve', 'ocho'
    , 'ojos', 'olvidar', 'once', 'padre', 'perdon', 'poder', 'porfavor', 'preguntar', '
    quetal', 'quince', 'regular', 'repetir', 'rojo', 'saber', 'seis', 'si', 'siete', 'signar
    ', 'signo', 'su', 'trabajar', 'trece', 'tres', 'tu', 'universidad', 'uno', 'veinte', '
    verbos', 'verde', 'vivir', 'vosotras', 'vosotros', 'yo'])

data_directory = "Dataset/separated_gestures"

X_list = []
y_list = []

for action in actions:
    for i in range(40):
        filepath = os.path.join(data_directory, action, action + '{}.csv'.format(i + 1))
        df = pd.read_csv(filepath)

        X_list.append(df.values)
        y_list.append(action)

max_length = max(len(seq) for seq in X_list)

X_padded = np.zeros((len(X_list), max_length, X_list[0].shape[1]))
for i, seq in enumerate(X_list):
    X_padded[i, :len(seq), :] = seq

X = np.array(X_padded)
y = np.array(y_list)

label_encoder = LabelEncoder()
y_numerical = label_encoder.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y_numerical, test_size=0.2,
    random_state=42)
```

Código 1: Procesamiento de los datos

3 Modelos

El reconocimiento de acciones se basa en el reconocimiento de patrones puesto que una acción supone emplear repeticiones en el desplazamiento, es decir, trayectorias. Un gesto tiene asociada una trayectoria que varía entre usuarios. De ahí la utilidad de emplear redes neuronales, estas son capaces de generalizar dichas trayectorias y predecir con suficiente precisión el gesto recibido. En consecuencia, la red se entrenará para poder adaptarse a distintos usuarios.

3.1 RNN

Las Redes Neuronales Recurrentes, tal como se explica en [2], tienen la característica de emplear una "memoria". De este modo la salida de la red depende de la entrada actual y/o la entrada o salida anterior, de modo que los preceptrones multicapa conforman este tipo de redes. Para poder entrenar la red en el seguimiento de trayectorias, es necesario conocer las posiciones de los puntos registrados en cada frame del video. Por lo tanto, la red aprende a seguir las trayectorias y predecir el gesto que se ha querido expresar.

La configuración o diseño de la red implementada es bastante simple. Se trata de 3 capas, tan solo la oculta tiene realimentación de recurrencia.

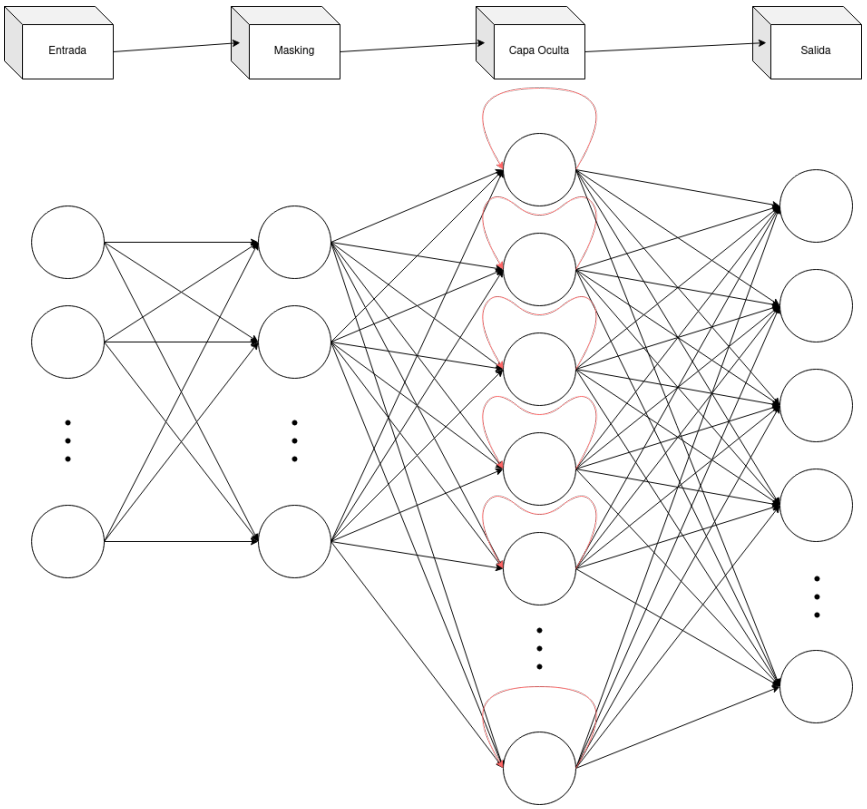


Figura 1: Topología de la red neuronal recurrente modelada.

Como se puede observar en la línea 5 del Código 2, se decidió hacer uso de un modelo secuencial, que es una pila lineal de las siguientes 3 capas:

- La primera capa de entrada es de enmascaramiento, ya que ignora las entradas con un determinado valor, en este caso 0.0. Esta capa es útil cuando se tienen secuencias de longitud variable y se quiere que el modelo ignore las partes de la secuencia llenas de valores de relleno, como se da en este caso.
- La segunda capa es de tipo SimpleRNN con 32 unidades y función de activación ReLU. SimpleRNN es una capa de red neuronal recurrente simple que toma en cuenta la información temporal en las secuencias de datos.
- La tercera y última capa es densa con un número de unidades igual al número de acciones, 91 en este caso, y utiliza la función de activación softmax. Esto se utiliza comúnmente en problemas de clasificación para obtener probabilidades normalizadas para cada clase.

Por último, se compila el modelo con el optimizador Adam, debido a su eficacia en la adaptación de la tasa de aprendizaje, la función de pérdida sparse_categorical_crossentropy (utilizada para problemas de clasificación con etiquetas enteras) y la métrica de precisión (accuracy) para evaluar el rendimiento del modelo durante el entrenamiento.

```
1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import SimpleRNN, Dense, Masking, InputLayer, LSTM, Dropout
3  from tensorflow.keras.optimizers import Adam
4
5  model = Sequential()
6  model.add(Masking(mask_value=0.0, input_shape=(max_length, X.shape[2])))
7  model.add(SimpleRNN(units=32, activation='relu'))
8  model.add(Dense(units=len(actions), activation='softmax'))
9
10 model.compile(optimizer="adam", loss='sparse_categorical_crossentropy', metrics=['accuracy'
       ])
```

Código 2: Modelo RNN creado

3.2 HMM

Los Modelos Ocultos de Markov (Hidden Markov Models), tal como se explica en [3], son una herramienta estadística utilizada para modelar sistemas dinámicos, especialmente aquellos que evolucionan a lo largo del tiempo. Los HMMs son particularmente útiles en el campo del reconocimiento de patrones y señales, lo que incluye aplicaciones como el reconocimiento de acciones en videos.

Como son efectivos para modelar secuencias de datos, como series temporales o secuencias de eventos, aplicado al reconocimiento de acciones, las acciones pueden considerarse como secuencias de poses o movimientos a lo largo del tiempo, y los HMMs pueden capturar la naturaleza dinámica de estas secuencias.

El modelo que se decidió implementar es el modelo basado en distribuciones gaussianas, debido a que las distribuciones gaussianas podrían capturar la variabilidad en los gestos. Debido a esta elección comenzaron los errores, ya que la clase GaussianHMM de la librería hmmlearn usada espera una entrada bidimensional con cada fila representando una observación, y cada columna una caraceterística. Para conseguir esto se utilizó la instrucción reshape como se puede observar en la línea 3 del Código 3.

Tras dividir los datos y las etiquetas en conjuntos de entrenamiento y de prueba tal como se hizo en el modelo RNN se definieron 3 estados ocultos de Markov, ya que un número muy bajo de estados ocultos podría no ser suficiente para modelar la complejidad del comportamiento y un número muy alto podría llevar a un sobreajuste. Ahora se crea el modelo y este caso, se utiliza una matriz de covarianza diagonal y 100 iteraciones además del número de estados ocultos ya comentado.

Por último se ajusta el modelo utilizando los datos de entrenamiento y se evalúa el rendimiento del mismo calculando el logaritmo de la verosimilitud del modelo en los datos de prueba. En el contexto de un modelo de Markov oculto, un logaritmo de verosimilitud más alto en los datos de prueba indica un mejor rendimiento del modelo en la tarea de reconocimiento de gestos, en este caso se obtuvo un número muy alto, tanto que hace sospechar sobre el correcto funcionamiento del mismo.

```
1  from hmmlearn import hmm
2
3  X_flatten = X.reshape((X.shape[0], -1))
4
5  X_train, X_test, y_train, y_test = train_test_split(X_flatten, y, test_size=0.2,
       random_state=42)
6
7  n_states = 3
8  model = hmm.GaussianHMM(n_components=n_states, covariance_type="diag", n_iter=100)
9  model.fit(X_train)
10
11 log_likelihood = model.score(X_test)
12 print(f'Log-verosimilitud del modelo HMM en el conjunto de prueba: {log_likelihood:.2f}')
```

Código 3: Código de HMM

4 Resultados

El repositorio del proyecto puedes encontrarlo en [1]. En él se encuentra todo el código junto al dataset para recrear el experimento. En este apartado se explicará como se han obtenido los resultados con los cuales se va evaluar el rendimiento del modelo creado. Para la evaluación se ha obtenido la precisión del modelo a lo largo de sus épocas, tanto la precisión durante el entrenamiento como la de validación, y la matriz de confusión.

4.1 Precisión

Para obtener la precisión se utilizó la instrucción evaluate, la cual devuelve los factores loss y accuracy de los resultados obtenidos sobre los datos y las etiquetas de prueba usando el modelo entrenado. Esto se realizó con modelos entrenados en 10, 20 y 30 épocas. Obviamente a mayor número de épocas, mejor resultado, pero más tarda el entrenamiento, aunque este modelo lleva a cabo el entrenamiento muy rápidamente.

```
import matplotlib.pyplot as plt

history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
    y_test))

print("Evaluando Modelo...")
accuracy = model.evaluate(X_test, y_test)[1]
print(f'Accuracy on test set: {accuracy}')

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Código 4: Cálculo de la precisión

Además, para representar la mejora de la precisión mientras se llevaba a cabo el entrenamiento y la validación, se ha hecho uso de la librería matplotlib. Para ello se ha guardado el historial del entrenamiento en la variable **history** y posteriormente se mostraron ambas precisiones.

En la Figura 2 se puede observar la gráfica que muestra el porcentaje de acierto según avanzan las épocas tanto del entrenamiento como de la validación. Por un lado, la precisión del entrenamiento se refiere a la precisión del modelo en el conjunto de datos de entrenamiento. Es la medida de qué tan bien la red está realizando las predicciones en los datos que ha visto durante el proceso de entrenamiento y se calcula comparando las predicciones de la red con las etiquetas reales en el conjunto de entrenamiento.

Por otro lado, está la precisión de la validación que se refiere a la precisión del modelo en un conjunto de datos que no ha usado durante el entrenamiento, conocido como conjunto de

validación, que generalmente se mantiene aparte del conjunto de entrenamiento. Ésta proporciona una evaluación del rendimiento de la red en datos no usados y ayuda a detectar si la red está aprendiendo patrones generales o si está sobreajustando los datos de entrenamiento.

En resumen, mientras que **train accuracy** evalúa el rendimiento durante el entrenamiento en sí, **validation accuracy** proporciona una evaluación independiente del rendimiento en datos no vistos y es crucial para asegurar una buena generalización del modelo para nuevos datos.

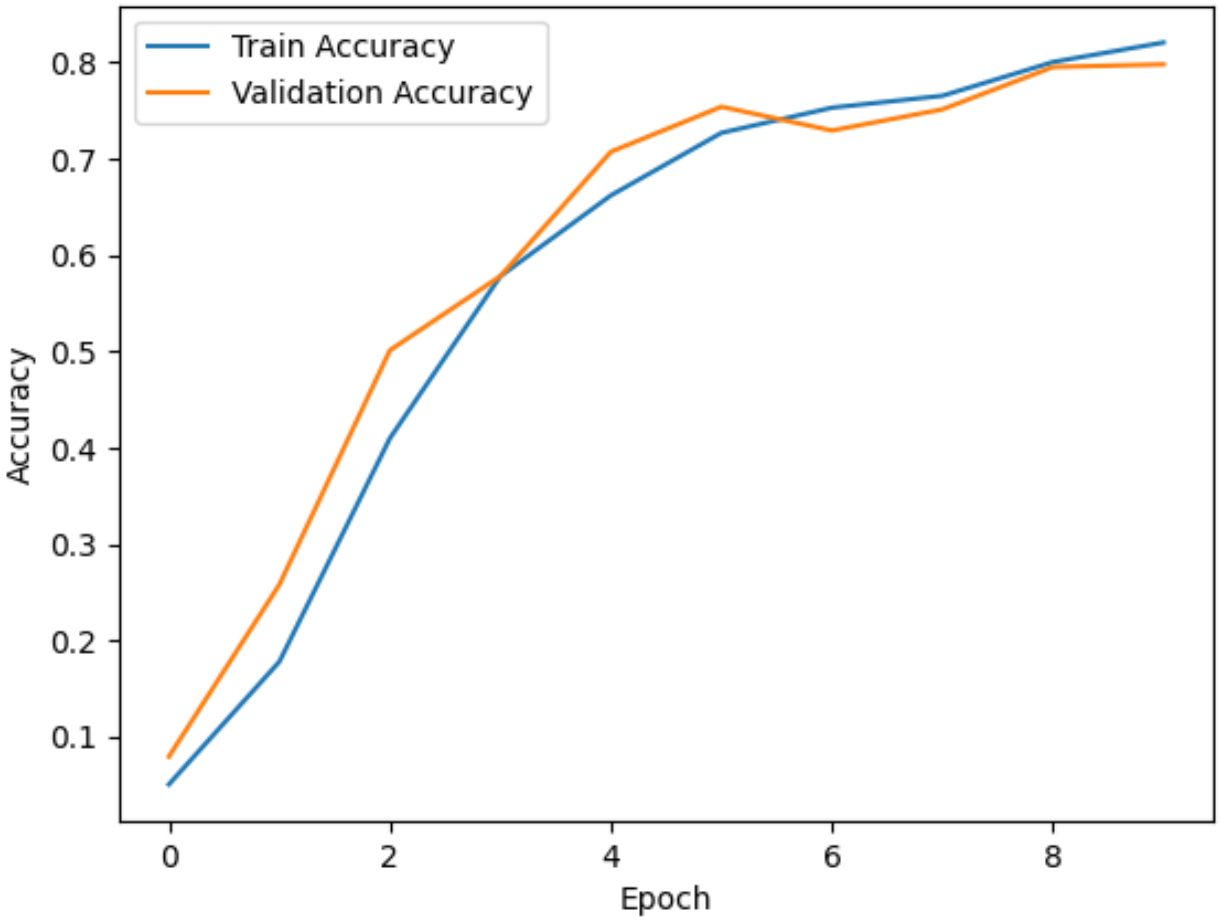


Figura 2: Precisión del entrenamiento y de la validación a lo largo de las épocas

4.2 Matriz de confusión

La matriz de confusión es una herramienta que se utiliza en el campo de la clasificación, como en el caso de modelos entrenados para reconocer gestos. Esta matriz permite evaluar el rendimiento de un modelo al comparar las predicciones que realiza con respecto a las clases reales de los datos. En la matriz:

- Los elementos en la diagonal principal representan el número de instancias correctamente clasificadas. Es decir, la intersección de la fila i y la columna i indica cuántas veces el modelo predijo correctamente la clase i.
- Los elementos fuera de la diagonal principal representan las instancias que fueron clasificadas incorrectamente. Por ejemplo, la intersección de la fila i y la columna j indica cuántas veces el modelo predijo la clase j cuando la clase real era i.

La matriz de confusión te permite evaluar el rendimiento del modelo de reconocimiento de gestos de manera más detallada que simplemente observar la precisión. Puedes calcular métricas adicionales, como la sensibilidad (tasa de verdaderos positivos), especificidad (tasa de verdaderos negativos), precisión, recall, entre otras, utilizando los valores de la matriz de confusión. Estas métricas proporcionan información valiosa sobre cómo se comporta el modelo en cada clase y pueden ayudarte a ajustar y mejorar tu modelo.

Para poder visualizar la matriz de confusión del modelo anterior se ha importado la librería seaborn ya que se utiliza comúnmente para hacer gráficos estadísticos atractivos, como mapas de calor. Como se puede observar en las líneas 3 y 4 del Código 5 se han generado las predicciones del modelo **y_probs** y se han obtenido las etiquetas predichas.

A partir de esto, se ha creado la matriz de confusión mediante la instrucción confussion_matrix con las etiquetas verdaderas y las predichas. Por último, se ha normalizado y se ha creado un DataFrame con la matriz de confusión normalizada para crear y visualizar un mapa de calor de la misma con Seaborn.

```
1  import seaborn as sns
2
3  y_probs = model.predict(X_test)
4  y_pred = np.argmax(y_probs, axis=1)
5
6  cm = confusion_matrix(y_test, y_pred)
7
8  cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
9
10 classes = unique_labels(y_test, y_pred)
11
12 cm_df = pd.DataFrame(cm, index=classes, columns=classes)
13
14 plt.figure(figsize=(10, 8))
15 sns.heatmap(cm_df, annot=False, cmap='Blues', cbar=True)
16 plt.title('Matriz de Confusion Normalizada')
17 plt.xlabel('Predicciones')
18 plt.ylabel('Etiquetas Verdaderas')
19 plt.show()
```

Código 5: Matriz de confusión

La matriz de confusión obtenida con el modelo entrenado anterior se puede observar en la Figura 3. Por ejemplo, la etiqueta 47 se confunde con la etiqueta 8, o la etiqueta 29 se confunde con la 85.

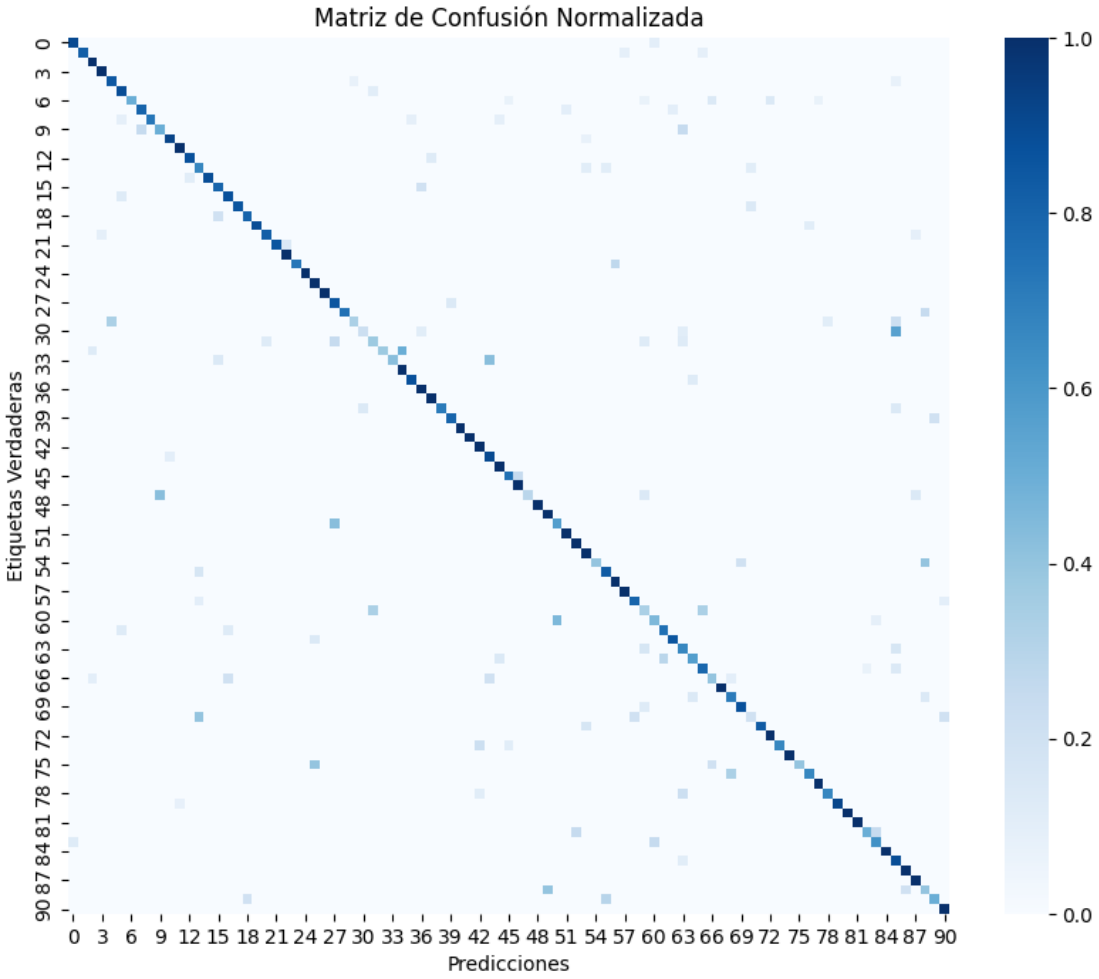
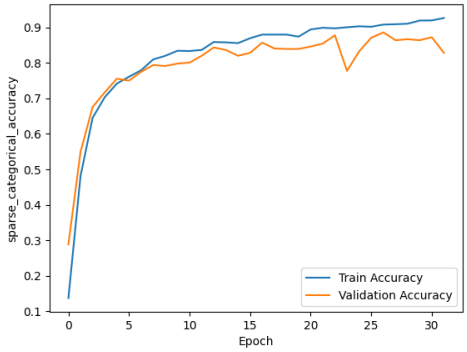


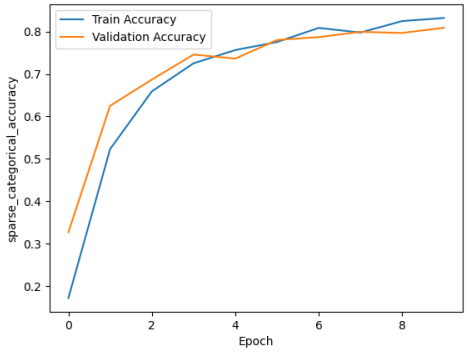
Figura 3: Matriz de confusión

5 Conclusiones

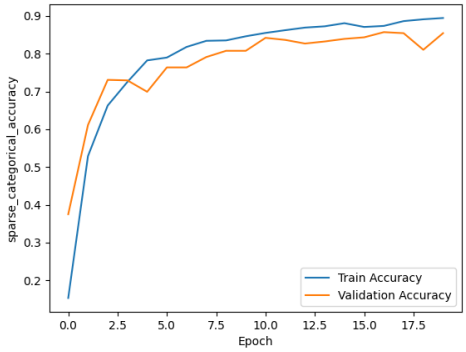
Como se ha comentado anteriormente, se realizaron pruebas de como mejora el rendimiento del modelo al aumentar las pruebas, con 10, 20 y 30 épocas. Estos son los resultados obtenidos:



(a) 10 epochs.



(b) 20 epochs.



(c) 30 epochs.

Figura 4: Comparación de resultados entre 10, 20 y 30 epochs

- **10 épocas:** 82.16%
- **20 épocas:** 87.28%
- **30 épocas:** 89.71%

Como se puede deducir, a mayor número de épocas mejores resultados se obtienen. Esto no significa que cuantas más épocas mejor será el modelo, ya que éste llegará a un punto donde no podrá mejorar más a pesar de que se aumente el número de épocas. Además, a mayor número de épocas, más tarda el entrenamiento del modelo, dato a tener en cuenta si se desea entrenar un modelo que tiene que realizar muchos más cálculos, debido a que tenga más capas o que tenga capas con mucha cantidad de neuronas, ya que tardará mucho más.

En este caso, el modelo obtenido es muy rápido computacionalmente, debido a que tiene pocas

capas de neuronas. El hecho de que con 10 épocas de entrenamiento ya se consiga una precisión del 78.16% significa que el modelo es eficiente.

Volviendo al enfoque inicial del proyecto, si se deseara implementar un traductor de las señas del dataset con las que ha sido entrenado el modelo, se necesitaría el mismo equipamiento que se tuvo cuando se creó el dataset, ya que tienen que ser datos acordes a los que se han usado para entrenar el modelo. Además, para poder implementarlo en tiempo real habría que tener en cuenta el coste computacional del modelo para realizar las predicciones. En este caso, en ese contexto no habría ningun problema ya que debido a las pocas capas usadas, y a las pocas neuronas que contiene cada capa, el modelo no tiene que realizar una cantidad enorme de cálculos.

Bibliografía

[1] M Fernández Lorenzo. Repositorio. https://github.com/MiguelFernandezLorenzo/UAGIR31VC23, 2023.

[2] L Múgica Ramón. Reconocimiento de la lengua de signos española mediante redes neuronales profundas. Universidad de Alicante, Diciembre 2023.

[3] Zuzanna Parcheta and Carlos-D. Martínez-Hinarejos. Sign language gesture classification using neural networks. In *In proceedings of the X Jornadas en Tecnologías del Habla and VI Iberian SLTech Workshop (iberSPEECH 2018)*, pages 127–131, 2018.

[4] Zparcheta. Dataset usado. https://github.com/zparcheta/spanish-sign-language-db, 2017.