

CLASIFICADOR DE IMÁGENES

RED NEURONAL

Grupo 41

Carlos Luis Miranda Hernández
Alba Ramos Quintana
Adrián Robaina Mejías

ÍNDICE

1. INTRODUCCIÓN

2. DATASET

3. DESARROLLO DEL REPOSITORIO

4. ANÁLISIS DE ENSAYOS

4.1. ENSAYO 4

4.2. ENSAYO 3

4.3. ENSAYO 2

4.4. ENSAYO 1

4.5. ENSAYO MODELO PREENTRENADO

4.6. CONCLUSIONES

5. COMPARATIVA Y EVALUACIÓN DE
RESULTADOS

5.1. OPTIMIZADORES

5.2. MÉTRICAS RELEVANTES

6. WEBGRAFÍA

1. INTRODUCCIÓN

El propósito fundamental de este trabajo es desarrollar una red neuronal que, usando lo aprendido, sea capaz de clasificar imágenes en diferentes categorías. El objetivo de la práctica es aplicar los principios teóricos relacionados con las redes neuronales convolucionales (CNN), el preprocesamiento de datos y la interpretación de resultados.

Para realizar el proyecto, se han utilizado los materiales y apuntes proporcionados, los cuales fueron útiles para organizar el código, establecer el modelo y comprender todo el proceso de entrenamiento y validación de una red neuronal. A partir de estos conceptos, se ha implementado un sistema de clasificación de imágenes en PyTorch.

El trabajo incluye varias fases, que van desde la organización y preparación de los datos hasta el desarrollo del modelo, la evaluación de su rendimiento y desempeño y la comparación de diferentes configuraciones de hiperparámetros. Además, se han ejecutado numerosos ensayos experimentales para mejorar el modelo y alcanzar un grado adecuado/alto de precisión.

2. DATASET

Las imágenes que componen el conjunto de datos utilizado en este trabajo provienen de la plataforma [Kaggle](#) y representan cada una de las letras del alfabeto en lenguaje de signos americano, además de tres clases adicionales (*SPACE*, *DELETE* y *NOTHING*) que complementan la comunicación gestual. En total, el dataset está organizado en **29 carpetas**, una por cada clase (las letras desde la A hasta la Z (menos la ñ), más las tres categorías adicionales mencionadas).

Todas las imágenes tienen un tamaño uniforme de 200x200 píxeles, lo que simplifica su manejo y entrenamiento en la red. La colección de imágenes se divide en 54600 para entrenamiento(*train*), 15600 para validación y 7800 para prueba(*test*).

El dataset seleccionado es apropiado para los objetivos del trabajo, ya que no es un conjunto trivial. Las imágenes muestran diversidad en la condiciones de iluminación, en la posición y orientación de las manos, así como en el fondo de la escena. Además, el problema tiene una motivación práctica en términos de accesibilidad y reconocimiento de gestos.

3. DESARROLLO DEL REPOSITORIO

El repositorio del proyecto está estructurado en módulos y carpetas de ensayos, lo que asegura la organización y permite tanto el analizar como la ejecución de los resultados. Esta estructura modular posibilita la ejecución independiente de cada ensayo y el análisis sistemático de los resultados.

Se ha documentado todo el código a través de *docstrings*, explicando de manera precisa los métodos, sus atributos, parámetros y valores de retorno. Esto asegura que cualquier usuario o miembro del grupo pueda comprender rápidamente la funcionalidad de cada módulo sin tener que revisar el repositorio completo.

MÓDULOS GENERALES

- **showGraph.py:** genera las gráficas de evolución de *accuracy* y *loss* durante el entrenamiento como en la validación. Estas gráficas se guardan automáticamente en la carpeta del ensayo correspondiente, lo que facilita visualizar la convergencia del modelo, detectar overfitting o si algún hiperparámetro está afectando negativamente.
- **metrics_viewer.py:** permite analizar un *checkpoint* de entrenamiento guardado (.pth), calculando métricas promedio de *loss* y *accuracy* para entrenamiento y validación, y genera visualizaciones claras: gráficas de evolución de *loss* y *accuracy*, barras comparativas de medias y líneas de medias.
- **matriz.py:** calcula y visualiza la matriz de confusión del modelo entrenado usando el conjunto de prueba. La matriz se guarda como imagen y sirve como parte del análisis de rendimiento, para analizar qué clases se confunden más entre sí y qué tal está generalizando el modelo más allá del simple valor de *accuracy*.

ENSAYOS

Están organizados en su propia carpeta, y dentro de ella se incluye una subcarpeta llamada "resultados", donde se almacenan todos los outputs generados automáticamente durante el entrenamiento:

- La **matriz de confusión**, generada por matriz.py, que permite visualizar de forma clara qué clases han sido correctamente clasificadas y cuáles presentan confusiones.
- Las **gráficas de entrenamiento**, producidas por showGraph.py, donde se muestra la evolución tanto del *accuracy* como de la *loss* a lo largo de las épocas, facilitando el análisis del comportamiento del modelo (convergencia, *overfitting*, etc.)
- Las **gráficas de métricas promedio**, generadas por metrics_viewer.py.
- Dos **archivo de texto**, uno con información del entrenamiento y evaluación: tiempo total, por época, mejores *accuracy*, épocas sin mejora y *accuracy* final en *test* sus aciertos, generado por train.py. Y otro con las medias de *accuracy* / *loss* de metrics_viewer.py.

Cada ensayo incluye su propio model.py, dataset.py y train.py:

- **model.py:** define la arquitectura y los hiperparámetros específicos del ensayo.
- **dataset.py:** gestiona la carga y las transformaciones de las imágenes.
- **train.py:** controla el entrenamiento, incluyendo la carga del dataset, la construcción de la arquitectura y el bucle de entrenamiento/validación según los parámetros definidos. Tras finalizar, se utiliza *plot_training_history()* de showGraph.py, se guarda el modelo y se evalúa en el conjunto de test.

Gracias a esta organización, es muy sencillo comparar distintos ensayos y analizar cómo los cambios en la arquitectura o en el procesamiento de datos afectan los resultados obtenidos.

4. ANÁLISIS DE ENSAYOS

Para evaluar el rendimiento de las distintas configuraciones probadas, se han realizado varios ensayos cuyos resultados pueden ordenarse de peor a mejor de la siguiente manera: **ensayo 4** → **ensayo 3** → **ensayo 2** → **ensayo 1** → **ensayo Preentrenado**. Cada experimento presenta diferencias tanto en la arquitectura como en los hiperparámetros utilizados, lo que permite analizar cómo afectan estos cambios al comportamiento de la red.

Antes de analizar cada uno de los ensayos, es importante destacar un aspecto que ha tenido un impacto muy significativo en el entrenamiento: la elección del **learning rate (LR)** y del **scheduler** utilizado en cada caso. Durante la realización del proyecto, se ha podido comprobar que pequeñas variaciones en este parámetro producen resultados completamente distintos, desde convergencia estable hasta divergencia total del modelo.

Un **LR demasiado alto** provoca oscilaciones en la función de pérdida e impide que el modelo aprenda patrones útiles, mientras que un **LR demasiado bajo** genera un entrenamiento extremadamente lento, llegando incluso a estancarse sin mejorar la precisión. Debido a esta sensibilidad, varios ensayos han incorporado distintos **learning rate schedulers**, cuyo propósito es ajustar dinámicamente la tasa de aprendizaje durante el entrenamiento.

En este proyecto se han utilizado schedulers diferentes según el ensayo, con el objetivo de analizar cómo afectan al rendimiento y a la estabilidad del modelo. Algunos reducen el LR cuando la validación deja de mejorar, otros aplican una reducción progresiva por épocas, y otros utilizan ciclos suaves que permiten escapar de mínimos locales. Estos mecanismos han sido especialmente importantes porque hubo ensayos que divergían o se quedaban estancados si no se controlaba correctamente la evolución del LR.

En definitiva, la utilización de schedulers es una parte esencial del análisis comparativo, ya que permite identificar qué combinación de arquitectura, optimizador y dinámica de LR resulta más adecuada para el reconocimiento del alfabeto ASL.

A continuación, se describe en detalle cada uno de los ensayos y las mejoras que aportan respecto a los anteriores.

ENSAYO 4

Este es el modelo constituye la arquitectura **más sencilla** de todos los ensayos realizados. Está diseñado como una red ligera, con únicamente dos bloques convolutivos. Que nos servirá como base para los otros ensayos.

➤ ARQUITECTURA

La red neuronal contiene:

- **Dos bloque convolutivos:**
 - Cada bloque incluye una convolución (**LazyConv2d**), seguida de **Batch Normalization** y la función de activación **SiLU (Sigmoid Linear Unit)**.

Tras cada bloque se aplica **MaxPooling 2x2** para reducir la resolución espacial y controlar el coste computacional.

- Dropout moderado (0.2) después de la capa totalmente conectada, con el objetivo de reducir el sobreajuste sin afectar excesivamente la capacidad de aprendizaje, dado que el modelo solo tiene dos bloques convolutivos.
- AdaptiveAvgPool2d que reduce automáticamente cada mapa de activación a un tamaño fijo (2 x 2), permitiendo que la red sea independiente del tamaño exacto de entrada y simplificando el diseño del clasificador.
- Clasificador final:
 - Una capa lineal intermedia de 128 neuronas con activación SiLU.
 - Una capa lineal final que genera las num_classes salida para clasificación.

➤ FUNCIONES DE ACTIVACIÓN Y REGULARIZACIÓN

La activación **SiLU** se eligió por su mejor rendimiento respecto a ReLU en redes pequeñas, gracias a su suavidad y propiedades no lineales más estables.

➤ ENTRENAMIENTO

El entrenamiento está configurado con:

- **CrossEntropyLoss** como función perdida estándar para clasificación multiclas.
- **RMSprop** como optimizador con:
 - Tasa de aprendizaje **lr** = 5e-4
 - Parámetro **alpha** = 0.99, responsable de controlar la ventana de suavizado del gradiente.

RMSprop es adecuado para redes pequeñas y datasets con variabilidad moderada, ya que estabiliza la actualización de cada parámetro sin requerir una tasa de aprendizaje muy precisa.

➤ TRANSFORMACIONES DE ENTRENAMIENTO (TRAIN_TRANSFORM)

Durante el preprocesado se aplica un conjunto mínimo de aumentos de datos, diseñado para evitar sobreajuste sin alterar demasiado la imagen original ni añadir distorsiones que puedan confundir al modelo.

El **train_transform** incluye:

- **Resize a 96×96** píxeles para un tamaño uniforme en toda la red.
- **RandomHorizontalFlip (p = 0.5)**

Este aumento es seguro para el dataset ASL Alphabet, ya que muchos signos son aproximadamente simétricos y el volteo horizontal no altera su significado. Con ello se aumenta la diversidad de ejemplos y se mejora la capacidad de generalización.

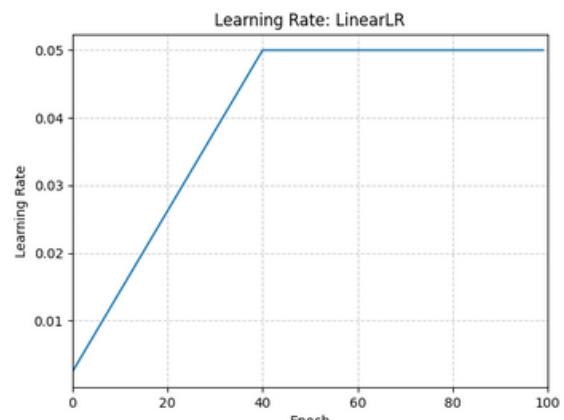
- **ToTensor** para convertir la imagen a formato manejado por PyTorch.
- **Normalización** con media y desviación estándar 0.5 en cada canal, lo que estabiliza el entrenamiento y acelera la convergencia.

➤ SCHEDULER

En este ensayo se empleó el scheduler **LinearLR**, cuyo papel es realizar una **transición suave del learning rate** al valor final definido en el optimizador. En lugar de comenzar directamente con un LR fijo, LinearLR permite que el modelo alcance ese valor de forma gradual, evitando saltos bruscos en la magnitud de las actualizaciones durante las primeras iteraciones.

Este comportamiento resulta útil cuando la arquitectura parte desde pesos sin ningún tipo de preentrenamiento y la red todavía no ha estabilizado la distribución de sus activaciones. Un incremento lineal controlado ayuda a que:

- El optimizador comience con pasos más estables y no demasiado agresivos.
- Las capas iniciales ajusten sus estadísticas sin grandes variaciones en los gradientes.
- La pérdida descienda de forma más regular durante la primera fase del entrenamiento.
- Se reduzca el riesgo de que el modelo quede atrapado en oscilaciones tempranas del entrenamiento..



La elección de LinearLR en este primer ensayo permite, además, disponer de un punto de comparación frente a otros schedulers utilizados posteriormente, donde se aplican estrategias más complejas de reducción de LR según el comportamiento de la validación o la progresión de las épocas.

ENSAYO 3

Este modelo corresponde a un **versión ampliada** de la arquitectura mínima, incorporando cinco bloques convolutivos en lugar de dos. El objetivo de este ensayo es analizar cómo mejora el rendimiento al aumentar la profundidad de la red manteniendo tamaños de canal muy reducidos de la red (1, 2, 4, 8, 16 filtros por capa), lo cual permite conservar un coste computacional no muy alto.

A diferencia del Ensayo 4, este utiliza un esquema de aumento de datos más completo y un optimizador destino, lo que permite estudiar su impacto en la capacidad de generalización.

➤ ARQUITECTURA

La red neuronal incluye:

- Cinco bloques convolutivos progresivos como se comentó anteriormente (**1 → 2 → 4 → 8 → 16**):
- Cada bloque está formado por:
 - Una capa convolutiva (LazyConv2d)
 - Batch Normalization
 - Función de activación GELU, más expresiva y suave que ReLU y adecuada para arquitecturas con activaciones pequeñas.
 - Una operación MaxPooling 2 x 2, encargada de reducir gradualmente la resolución espacial.

➤ REGULARIZACIÓN

Se aplica **Dropout** (0.2) tras la capa lineal intermedia del clasificador, con el fin de reducir el sobreajuste.

➤ ADAPTIVE AVGPOOL2D

Reduce automáticamente cada mapa de activación a 1×1 , creando un vector compacto con una única activación por canal. Esto simplifica notablemente la capa lineal posterior.

➤ CLASIFICADOR FINAL

- Una capa lineal intermedia de **128 unidades** con activación **GELU**.
- Una capa lineal final que genera las **num_classes** salidas para clasificación.

➤ FUNCIONES DE ACTIVACIÓN Y REGULARIZACIÓN

- La activación **GELU** se escogió debido a su capacidad para conservar más información útil que ReLU en arquitecturas profundas y estrechas.
- Combinada con el **Dropout moderado**, proporciona un buen equilibrio entre expresividad y control del sobreajuste.

➤ ENTRENAMIENTO

La configuración de entrenamiento incluye:

- **CrossEntropyLoss** como función de pérdida estándar para clasificación multiclas.
- **Adamax** como optimizador, configurado con:
 - **Tasa de aprendizaje 1e-3.**

Adamax es especialmente estable en redes con gradientes pequeños o irregulares, como ocurre en arquitecturas profundas de baja dimensionalidad, por lo que resulta útil para este tipo de red minimalista.

➤ TRANSFORMACIONES DE ENTRENAMIENTO (TRAIN-TRANSFORM)

En este ensayo se utilizó un esquema de aumento de datos más completo que en el [Ensayo 4](#). El objetivo es incrementar la robustez del modelo frente a pequeñas variaciones geométricas del gesto.

El train_transform incluye:

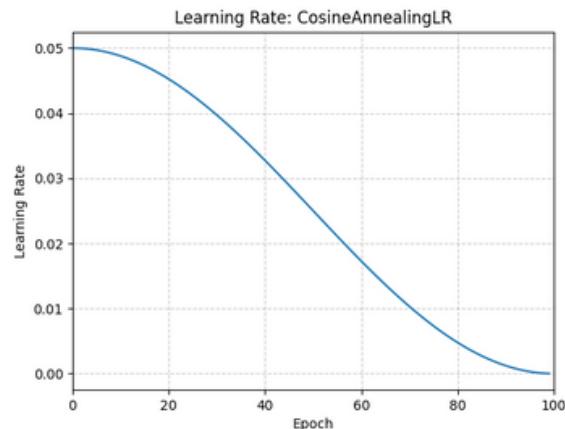
- **Resize** a 96×96 píxeles.
- **RandomCrop (padding = 4)**: Simula ligeros desplazamientos y recortes, permitiendo que el modelo aprenda a ser más invariante respecto a la posición de la mano en la imagen.
- **RandomHorizontalFlip (p = 0.5)**: Útil en el ASL Alphabet debido a la simetría de muchos signos, incrementando la variabilidad del conjunto sin afectar la etiqueta.
- **RandomRotation (± 5 grados)**: Añade pequeñas variaciones angulares que aparecen de forma natural al capturar imágenes de manos, y que el modelo debe aprender a ignorar.
- **ToTensor** para convertir las imágenes a formato PyTorch.
- **Normalización** con media y desviación estándar **0.5** por canal, lo que estabiliza la propagación del gradiente.

➤ SCHEDULER

En este ensayo se empleó CosineAnnealingLR, un scheduler que ajusta el learning rate siguiendo una curva coseno decreciente. A diferencia de una reducción lineal o basada en métricas externas, este método introduce una **variación suave y continua** del LR a lo largo de todas las épocas. El valor de la tasa de aprendizaje disminuye progresivamente hasta un mínimo predefinido, evitando reducciones abruptas que puedan interrumpir el proceso de optimización.

La elección de CosineAnnealingLR se fundamenta en que este tipo de programación favorece:

- Un descenso del learning rate más natural.
- Actualizaciones progresivamente más precisas a medida que el modelo converge.
- Menor riesgo de caer en oscilaciones al final del entrenamiento.
- Una fase final más estable en la que el modelo refina mejor los patrones aprendidos.



Además, este scheduler es especialmente útil cuando se entrena una red desde cero y se espera que las primeras épocas exploren el espacio de parámetros con un LR relativamente alto, mientras que las últimas requieren ajustes más finos. Su comportamiento suave y no monótono permite una convergencia más controlada en comparación con otros métodos de reducción por saltos o factores fijos.

Gracias a esta dinámica, CosineAnnealingLR proporciona un buen equilibrio entre exploración y precisión, lo cual es relevante para modelos con mayor profundidad o con mayor variabilidad en los gradientes, como ocurre en este ensayo.

ENSAYO 2

Este modelo corresponde a una arquitectura **CNN ligera**, diseñada para mantener una buena capacidad de aprendizaje sin un coste computacional elevado. A diferencia del Ensayo 3 y 4, este ensayo utiliza cuatro bloques convolutivos, aunque con un número de filtros más amplio (**16 → 32 → 32 → 64**).

El objetivo es evaluar el rendimiento de una arquitectura sencilla pero bien equilibrada, aprovechando una función de activación más moderna (**Mish**) y optimizador robusto como **AdamW**.

➤ ARQUITECTURA

La red neuronal está formada por cuatro bloques convolutivos, cada uno compuesto por:

- Una convolución **LazyConv2d** como en los anteriores ensayos.
- **Batch Normalization** para estabilizar el aprendizaje.

- Activación **Mish**, una función suave y no lineal más expresiva que **ReLU**.
- **MaxPooling 2 x 2**, responsable de la reducción progresiva de la dimensión espacial.
- **Dropout2D (0.1)** tras cada bloque, reduciendo el sobreajuste especialmente cuando las activaciones presentan correlación espacial.

El numero de filtros por bloque sigue la progresión: **16 → 32 → 32 → 64**

➤ REGULARIZACIÓN

La arquitectura combina:

- **Dropout2D (0.1) comentando anteriormente.**
- **Dropout (0.3)** en el clasificador final, ayudando a reducir el sobreajuste tras el *flattening* de los mapas comprimidos.

Este esquema de regularización es adecuado para redes ligeras, donde la capacidad es suficiente para aprender, pero existe riesgo de memorizar parte del conjunto de entrenamiento.

➤ ADAPTIVEAVGPOOL2D

Antes de la fase lineal, se utiliza un:

- **AdaptiveAvgPool2d(1×1)**

Esta capa condesa cada canal en una única actuación, resultando en un vector de 64 valores (uno por filtro de último bloque). De esta forma, el clasificador posterior se simplifica, reduciendo parámetros y aumentando la eficiencia.

➤ CLASIFICADOR FINAL

El clasificador consiste en:

- **Dropout (0.3)** tras el *flatten*.
- Una única **capa lineal** que proyecta el vector de 64 canales hacia las **num_classes** del dataset.

Ideal para evitar sobreajuste y comprobar si la parte convolutiva es suficiente.

➤ FUNCIONES DE ACTIVACIÓN Y REGULARIZACIÓN

La activación **Mish** se eligió por su capacidad para:

- Mejorar la suavidad en el flujo de gradientes
- Opera mejor en arquitecturas con un número intermedio de filtros.

Combinada con el esquema de **Dropout2D + Dropout**, la red mantiene un equilibrio adecuado.

➤ ENTRENAMIENTO

La configuración de entrenamiento incluye:

- **CrossEntropyLoss** como función de pérdida estándar para clasificación multiclas.
- **AdamW** como optimizador, con:
 - **learning rate = 1e-3.**
 - **weight decay = 1e-4.**

AdamW es una mejora respecto al Adam clásico, separando la decaída de pesos de la estimación de gradientes, lo que lo hace especialmente robusto en arquitecturas **CNN ligeras** como esta.

➤ TRANSFORMACIONES DE ENTRENAMIENTO (TRAIN-TRANSFORM)

En este ensayo se emplea el mismo esquema de aumentos definido previamente, basado en pequeñas perturbaciones geométricas destinadas a mejorar la capacidad de generalización.

El pipeline incluye redimensionado a **96x96**, recortes aleatorios suaves (**padding=4**), volteo horizontal y una ligera rotación, junto con la conversión a tensor y normalización estándar.

Este conjunto de transformaciones proporciona diversidad sin alterar la estructura esencial de los signos, manteniendo la coherencia del gesto.

➤ SCHEDULER

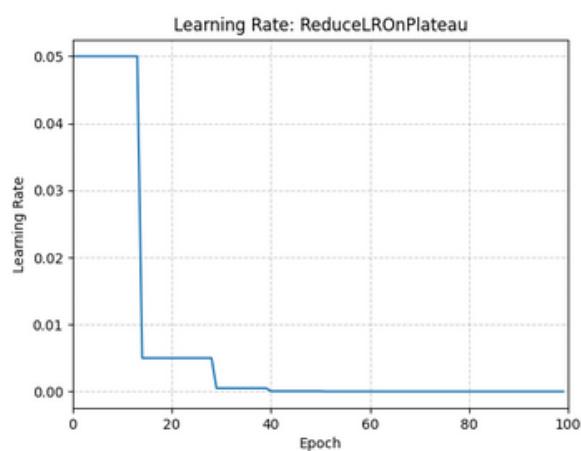
En este último ensayo se empleó **ReduceLROnPlateau**, un scheduler que ajusta la tasa de aprendizaje únicamente cuando el rendimiento del modelo deja de mejorar. A diferencia de LinearLR y CosineAnnealingLR, que reducen el LR siguiendo una función prefijada independientemente del estado del entrenamiento, este método toma decisiones en función del comportamiento real de la métrica monitorizada (en este caso, la pérdida de validación).

A lo largo de los ensayos previos se observó que los schedulers basados en un calendario fijo (LinearLR y CosineAnnealingLR) presentaban una limitación importante: **modificaban el learning rate incluso cuando el modelo todavía estaba aprendiendo de forma efectiva**. Esto provocaba, en algunos casos, que el LR se redujera demasiado pronto o en momentos poco adecuados, dificultando la capacidad del modelo para avanzar y obteniendo resultados menos consistentes.

Reducir el LR “a ciegas”, sin tener en cuenta la evolución real del entrenamiento, no resultó óptimo para los modelos base utilizados en este proyecto.

Por ello, ReduceLROnPlateau se adapta mejor a este escenario. Sus ventajas principales son:

- **La reducción del LR depende del estancamiento real del modelo**, no de una función temporal fija.
- **Evita reducciones prematuras** cuando el modelo todavía está mejorando.
- **Actúa como un mecanismo de estabilización**, permitiendo que el optimizador refine los parámetros solo cuando es necesario.
- **Es especialmente útil en redes entrenadas desde cero**, donde la dinámica de aprendizaje es más sensible y menos predecible que en modelos preentrenados.



Para ello se utiliza una variable llamada **patience** que es el número de épocas que se deben esperar para reducir el lr.

ENSAYO 1

Este ensayo corresponde al cierre de todos los ensayos realizados y representa la síntesis de aquellos que han demostrado funcionar mejor a lo largo de los ensayos previos. Después de evaluar arquitecturas extremadamente mínimas, variantes profundas con canales reducidos y redes ligeras pero equilibradas, este ensayo final consolida los aprendizajes obtenidos para construir una arquitectura robusta, estable y equilibrada, adecuada para un uso más práctico del reconocimiento del alfabeto ASL.

Mientras los ensayos anteriores tenían un propósito principalmente comparativo o exploratorio, este ensayo final combina características que demostraron tener impacto positivo en la generalización sin comprometer de manera significativa el coste computacional.

➤ ARQUITECTURA

La arquitectura final se fundamenta en tres bloques convolutivos progresivos ($16 \rightarrow 32 \rightarrow 64 \rightarrow 128$ filtros).

Esta configuración se establece como un punto medio ideal entre:

- Es **más potente y expresiva** que la arquitectura mínima del [Ensayo 4](#), gracias a un mayor número de filtros por bloque y un extractor de características más profundas.
- Es **menos profunda** que la arquitectura del [Ensayo 3](#), lo que reduce el coste computacional y mejora la estabilidad del entrenamiento, evitando los problemas observados en redes muy estrechas y profundas.
- Recupera elementos del [Ensayo 2](#) (como la combinación de **Dropout2D**, **Batch Normalization** y **bloques convolutivos amplios**) pero aumentando la capacidad del modelo hasta obtener un extractor robusto sin llegar a sobreajuste de forma severa.

Cada bloque incluye:

- **LazyConv2d** para mantener adaptabilidad al tamaño de entrada.
- **Batch Normalization** para asegurar estabilidad durante el entrenamiento.
- **ReLU** como activación (más eficiente en redes de esta escala que GELU o SiLU).
- **MaxPooling 2×2** para reducir gradualmente la resolución espacial.
- **Dropout2D** en los bloques intermedios, trasladado desde el [Ensayo 2](#) debido a su eficacia para reducir sobreajuste en representaciones espaciales.

Esta arquitectura se posiciona como una red generalista capaz de equilibrar precisión, velocidad de entrenamiento y facilidad de implementación.

➤ REGULARIZACIÓN

El modelo adopta un esquema de regularización que ha demostrado ser consistente en los ensayos previos:

- **Dropout2D** (0.1) tras bloques convolutivos.
- **Dropout** (0.3) en el clasificador final.

Esta combinación ya había mostrado mejores resultados en redes ligeras ([Ensayo 2](#)), y su inclusión aquí responde a la necesidad de mantener la capacidad de generalización sin aumentar innecesariamente la profundidad o complejidad del modelo.

➤ ADAPTIVE FLATTENING Y CLASIFICADOR FINAL

La capa convolutiva final se aplana para generar un vector que pasa a través de un clasificador denso compuesto por:

- Capa lineal de **1024** unidades **BatchNorm1D** y **ReLU**.
- Capa de **256** unidades con normalización y activación.
- Capa de salida con **num_classes** neuronas.

Este diseño, a diferencia del clasificador minimalista del [Ensayo 2](#) (solo una capa lineal), permite capturar relaciones abstractas adicionales entre los patrones de entrada, compensando la menor profundidad respecto al [Ensayo 4](#).

➤ FUNCIONES DE ACTIVACIÓN Y REGULARIZACIÓN

Este ensayo final se actualiza **ReLU**, retomando la conclusión derivada de los ensayos anteriores:

- **SiLU** ([Ensayo 4](#)) es muy útil en redes muy pequeñas.
- **GELU** ([Ensayo 3](#)) destaca en redes muy profundas.
- **Mish** ([Ensayo 2](#)) ofrece suavidad en redes ligeras, pero con mayor coste computacional.

ReLU, por tanto, se posiciona como la opción más eficiente y suficientemente expresiva para este modelo final de complejidad intermedia.

➤ ENTRENAMIENTO

El modelo utiliza la configuración que mostró mayor estabilidad a lo largo de todos los ensayos:

- **CrossEntropyLoss** como función de pérdida estándar.
- **Adam** con **weight decay 1e-4**, debido a:
 - Su mayor robustez que **Adam** puro ([Ensayo 1 y 2](#)),
 - Que permite un entrenamiento estable incluso con aumentos fuertes y **Dropout**.

La tasa de aprendizaje de **1e-3** equilibra rapidez y estabilidad, evitando la sensibilidad excesiva que se observó en **RMSprop** ([Ensayo 4](#)) y siendo más adecuado para arquitecturas no extremadamente profundas como esta.

➤ ETRANSFORMACIONES DE ENTRENAMIENTO (TRAIN-TRANSFORM)

El esquema de aumentos utilizado en este ensayo final es el más completo y agresivo, combinando las conclusiones de los ensayos anteriores:

- De [Ensayo 4](#) se retoma el flip horizontal moderado.
- De [Ensayo 3](#) se adopta el **RandomCrop** y la rotación suave.
- De [Ensayo 2](#) se mantiene la normalización estándar y el tamaño de entrada consistente.

El pipeline incluye:

- Resize a **128x128 píxeles**.
- **RandomCrop** con padding 8.
- **RandomHorizontalFlip** (0.5).
- **RandomRotation** $\pm 15^\circ$.
- **ColorJitter** suave para simular variaciones reales de iluminación.
- **ToTensor** y normalización estándar.

Este conjunto de transformaciones no solo diversifica el dataset, sino que refleja el aprendizaje acumulado sobre cuáles perturbaciones favorecen la estabilidad y generalización del modelo.

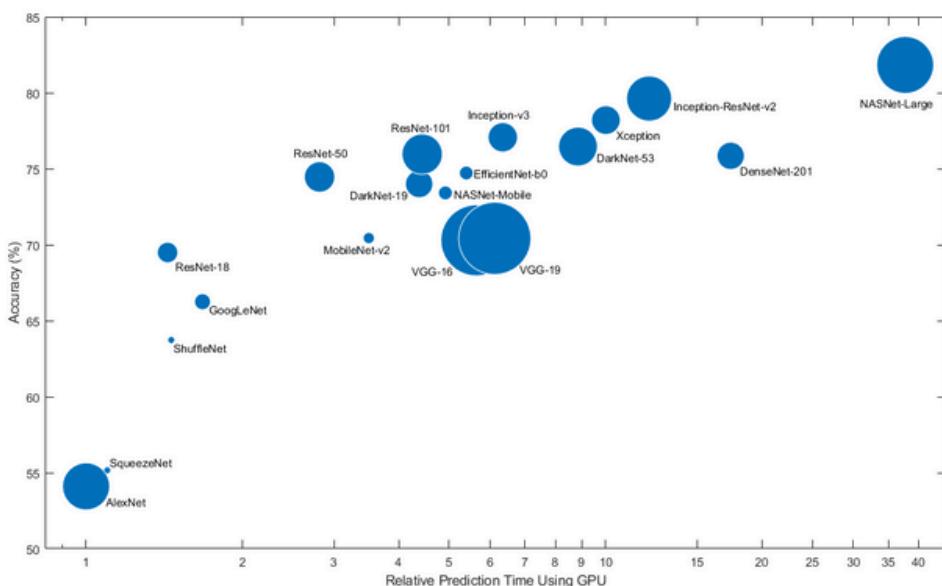
➤SCHEDULER

En este ensayo se ha decidido continuar con **ReduceLROnPlateau** porque, en resumidas cuentas, aunque **LinearLR** y **CosineAnnealingLR** son técnicas interesantes y ampliamente utilizadas, en este proyecto resultaron menos adecuados debido a la variabilidad del aprendizaje en las arquitecturas personalizadas. **ReduceLROnPlateau** ofreció un comportamiento más estable y coherente con la dinámica de mejora del modelo, adaptándose al rendimiento real durante el entrenamiento y proporcionando un ajuste más óptimo del *learning rate*.

ENSAYO - MODELO PREENTRENADO

Este último ensayo cambiará el enfoque inicial de usar como base el anterior ensayo, y se hará uso de modelo ya entrenados.

En nuestro caso se utilizará el modelo preentrenado **EfficientNet-B0**, una red convolutiva diseñada mediante *neural architecture search* y basada en un escalado compuesto de profundidad, anchura y resolución. Este diseño permite alcanzar una relación muy eficiente entre capacidad representativa y número de parámetros, lo que la hace especialmente adecuada para tareas con imágenes relativamente pequeñas, como ocurre en este dataset.



Además, EfficientNet-B0 destaca por su **compacidad y eficiencia computacional**. Con aproximadamente **5.3 millones de parámetros** (unos **20 MB** en FP32) ofrece un rendimiento competitivo utilizando aproximadamente la **mitad de los recursos** que arquitecturas tradicionales como ResNet50. El tamaño de entrada recomendado para este modelo es **224x224**, aunque en este proyecto se emplean imágenes de **128x128** para adaptarse a las restricciones de hardware disponibles, manteniendo un equilibrio adecuado entre coste computacional y rendimiento.

Estas características convierten a EfficientNet-B0 en una opción especialmente ventajosa en entornos con recursos limitados, como el utilizado en este proyecto.

En este ensayo se utiliza la versión preentrenada en **ImageNet**, lo que permite transferir características visuales generales (bordes, texturas, formas) ya aprendidas.

>ARQUITECTURA INTERNA: MBConv Y COMPOUND SCALING

A diferencia de las arquitecturas tradicionales diseñadas manualmente, EfficientNet-B0 se basa en el bloque **MBConv** (Mobile Inverted Bottleneck Convolution) con mecanismos de atención **Squeeze-and-Excitation** (SE).

Esta estructura permite reducir drásticamente el coste computacional mediante el uso de convoluciones separables en profundidad (*depthwise separable convolutions*), que descomponen la operación de filtrado espacial y la combinación de canales en dos pasos más eficientes.

Además, los bloques SE actúan como un mecanismo de atención, permitiendo a la red recalibrar dinámicamente la importancia de cada canal de características, enfatizando las informaciones relevantes (como la forma de la mano) y suprimiendo el ruido de fondo.

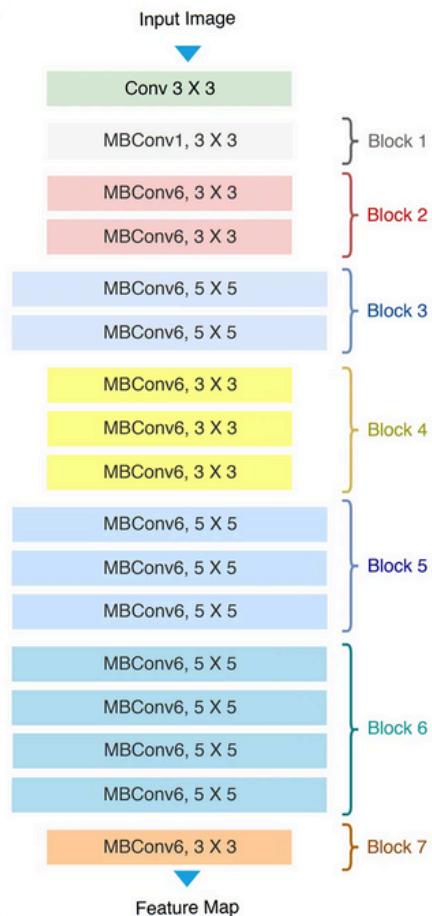
Por último, la red utiliza la función de activación **Swish** ($x * o(x)$) en lugar de la tradicional ReLU, lo que favorece un flujo de gradientes más suave y mejora la convergencia durante el entrenamiento.

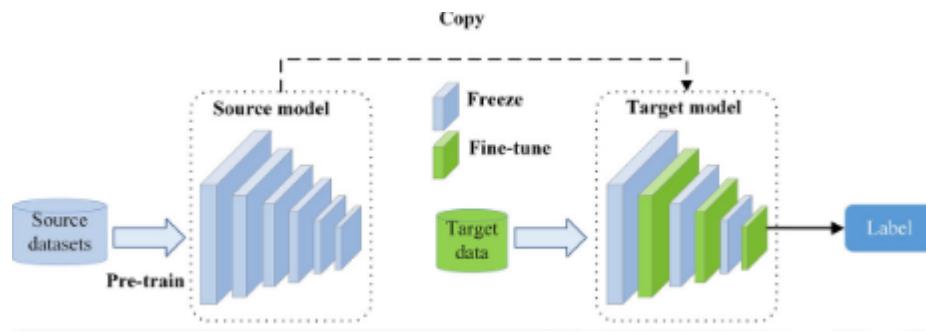
>ESTRATEGIA DE FINE-TUNING:

Para maximizar el rendimiento del modelo preentrenado EfficientNet-B0 sin empezar el aprendizaje desde cero, se ha implementado una técnica de **Fine-Tuning** selectivo. El proceso consiste en aprovechar las capacidades de extracción de características generales que la red ya aprendió del dataset ImageNet, mientras se re-entrenan específicamente las capas más profundas para reconocer los matices particulares del lenguaje de signos.

En la práctica, esto se traduce en la siguiente configuración de entrenamiento:

1. **Congelación de la Base:** Se mantienen congelados (no entrenables) los bloques iniciales e intermedios de la red.
2. **Descongelación Selectiva:** Se descongelan únicamente los últimos bloques convolucionales (bloques 7 y 8 en EfficientNet-B0) y el clasificador final.
3. **Adaptación del Clasificador:** Se sustituye la capa final original por una nueva capa lineal ajustada a las 29 clases.





Esta estrategia equilibra dos factores esenciales:

1. Aprovechar el conocimiento previo del modelo.
2. Especializar las etapas finales en el problema concreto sin incrementar el coste computacional.

➤RELACIÓN CON LA ESTRUCTURA DE LOS ENSAYOS ANTERIORES

Aunque **EfficientNet-B0** es una arquitectura externa, la estructura experimental sigue la misma lógica utilizada en los ensayos previos:

1. DIFERENTES NIVELES DE COMPLEJIDAD ARQUITECTÓNICO

Los ensayos anteriores progresaban desde CNN muy sencilla ([Ensayo 3](#)), seguida por una versión más profunda ([Ensayo 4](#)) y otro más ancha y con activaciones modernas ([Ensayo 2](#)).

EfficientNet-B0 se integra como el nivel superior en este escala, proporcionando un referencia de rendimiento para una arquitectura altamente optimizada.

2. CRITERIOS DE REGULARIZACIÓN

Los ensayos del 2 al 4 utilizaban **Dropout**, **BatchNorm** y **funciones de activacion suaves (Mish, SiLU, GELU)** como mecanismo de regularización.

EfficientNet-B0 incorpora su propio esquema de regularización interna ([MBConv](#), [SE blocks](#), [swish activations](#)), y en este ensayo se complementa con el uso de:

- **ColorJitter**
- **RandomCrop**
- **RandomRotation**
- **RandomHorizontalFlip**

Ofreciendo un aumento de datos más rico que los empleados en los ensayos anteriores.

Con ello, este ensayo mantiene la coherencia metodológica y amplia el conjunto de técnicas de regularización.

3. TRANSFER LEARNING COMO EXTENSIÓN NATURAL DEL ENFOQUE EXPERIMENTAL

Los modelos anteriores demostraron que las CNN diseñadas desde cero pueden aprender patrones del ASL, pero requieren más tiempo de entrenamiento y son más sensibles al sobreajuste.

➤ ENTRENAMIENTO Y OPTIMIZACIÓN

Para ajustarse a la estructura de experimentación previa, se mantiene la función pérdida estándar **CrossEntropyLoss**.

El optimizador elegido es **AdamW**, debido a su estabilidad y su mejor gestión de la regularización mediante decaimiento de pesos, especialmente importante al reentrenar una capa final recién inicializada sobre una red parcialmente congelada.

➤ CONCLUSIONES PRELIMINARES DEL ENSAYO FINAL

El uso de EfficientNet-B0 representa un salto respecto a los ensayos anteriores en términos de expresividad arquitectónica. Su capacidad para aprovechar el conocimiento adquirido en ImageNet, junto con un conjunto de aumentos de datos más completo, permite explorar el límite superior del rendimiento posible para este proyecto.

Este ensayo final no sustituye el trabajo realizado con las CNN personalizadas, sino que las complementa, ofreciendo una visión completa que va desde modelos construidos manualmente hasta una arquitectura moderna preentrenada.

CONCLUSIONES

Este ensayo final se presenta como la **integración de todo el proceso experimental**. No persigue la mínima complejidad ([Ensayo 3](#)), ni la máxima profundidad controlada ([Ensayo 4](#)), ni el balance experimental de funciones modernas ([Ensayo 2](#)). En cambio:

Busca combinar aquello que funcionó mejor a lo largo de todos los ensayos, resultando en un modelo práctico, robusto y aplicable a un escenario real.

5. COMPARATIVA Y EVALUACIÓN DE RESULTADO

Para poder comparar de forma justa el rendimiento de cada ensayo, todos los experimentos se han ejecutado bajo las mismas condiciones y utilizando exactamente el mismo ordenador. Esto asegura que las diferencias de los tiempos de entrenamiento, así como el comportamiento del modelo, se deban únicamente a los cambios de la arquitectura/hiperparámetros, y no a variaciones externas del hardware.

En concreto, todas las pruebas se realizaron con una **GPU NVIDIA RTX 3050** de portátil y un procesador Intel Core **i7-12700H**, garantizando así un entorno homogéneo para todos los ensayos.

TABLA COMPARATIVA -OPTIMIZADORES

OPTIMIZADOR	¿QUÉ ES?	¿CUÁNDO USARLO?	VENTAJA PRINCIPAL	DESVENTAJA
RMSprop	Adapta la tasa de aprendizaje dividiendo por el promedio de magnitudes recientes de los gradientes.	Redes Recurrentes (RNNs) o cuando se necesita simplicidad.	Muy bueno manejando gradientes ruidosos.	A veces converge más lento que Adam en CNNs modernas.
Adam	Combina las ideas de RMSprop + Momentum.	El estándar por defecto para casi todo (CNNs, NLP).	Convergencia muy rápida al inicio del entrenamiento.	Puede "estancarse" en óptimos locales o generalizar peor que SGD.
AdamW	Es Adam pero con una corrección en cómo aplica el "Weight Decay".	Modelos modernos (Transformers, EfficientNet) o si buscas mejor generalización .	Reduce el overfitting mejor que Adam estándar.	Requiere ajustar bien los hiperparámetros.
Adamax	Una variante de Adam que usa la norma infinito L^∞ en lugar de la norma L2.	Cuando Adam es inestable (ej. embeddings grandes o datos con mucho ruido).	Es más robusto y estable ante gradientes que explotan.	Menos común, a veces converge un poco más lento que Adam.

TABLA COMPARATIVA - MÉTRICAS RELEVANTES

ENSAYO	ARQUITECTURA	REGULARIZACIÓN	MEDIA ACCURACY TRAIN/DEV	MEDIA LOSS TRAIN/DEV	TIEMPO DE ENTRENAMIENTO
4	CNN Simple	Ninguna	76.22% / 78.69%	0.7298 / 0.6764	22m 26s (1m 7s / época)
3	CNN Simple Ampliada	Dropout	50.40% / 56.19%	1.5264 / 1.3553	16m 50s (50s / época)
2	CNN Ligera	Dropout + BatchNorm	74.13% / 87.57%	0.8099 / 0.4781	17m 6s (51s / época)
1	Optimizada	Dropout + L2 + Scheduler	91.32% / 98.34%	0.2654 / 0.0574	21m (1m 3s / época)
Ensayo PreEntrenado	Altamente Optimizada	MBConv + SE blocks + swish activations	98.09% / 99.57%	0.0900 / 0.0203	51m 1s (2m 33s / época)

- ▶ ENSAYO 4 → Evaluación - Total: 8700, Correctos: 7950
- ▶ ENSAYO 3 → Evaluación - Total: 8700, Correctos: 6095
- ▶ ENSAYO 2 → Evaluación - Total: 8700, Correctos: 8460
- ▶ ENSAYO 1 → Evaluación - Total: 8700, Correctos: 8676
- ▶ ENSAYO PREENTRENADO → Evaluación - Total: 8700, Correctos: 8698

- La evolución de los ensayos muestra un progreso constante desde modelos sencillos e inestables hasta arquitecturas mucho más robustas y exactas:

A pesar de que el **Ensayo 4** consigue un *accuracy* aceptable, evidencia un *train* muy irregular, con fuertes oscilaciones en los gráficos. **El Ensayo 3** en su caso, pese a obtener peores métricas, corrige ese comportamiento: añade regularización y aumentos de datos que estabilizan la curva y evitan el zig-zag del ensayo anterior.

El **Ensayo 2** representa el primer progreso significativo en términos de generalización. Su combinación de BatchNorm, Dropout2D y una arquitectura ligera pero equilibrada mejora claramente la validación y reduce la pérdida.

El **Ensayo 1** consolida esta tendencia, al incorporar lo más destacado de los ensayos anteriores y lograr una precisión muy alta tanto en entrenamiento como en validación, con curvas limpias y estables.

El **Ensayo del Modelo Preentrenado** representa el máximo rendimiento del proyecto. Gracias al transfer learning, EfficientNet-B0 aprovecha el conocimiento previo de ImageNet, logrando la menor pérdida y la mayor estabilidad; sin embargo, esto requiere un tiempo de entrenamiento más largo, pero es el más consistente y el que mejor generaliza, reflejando claramente las ventajas de reutilizar conocimiento ya adquirido.

- En la evaluación final (*test*) se confirma esta evolución: los ensayos iniciales tienen muchos errores (sobre todo el Ensayo 3), mientras que los modelos más avanzados -*Ensayo 2, Ensayo 1 y especialmente, el Preentrenado-* logran más de 8600 predicciones acertadas de un total de 8700. Esto demuestra que la mejora no solo ocurre durante el entrenamiento, sino también en la capacidad real del modelo para generalizar.

6. WEBGRAFÍA

https://docs.pytorch.org/vision/main/models/generated/torchvision.models.efficientnet_b0.html#torchvision.models.EfficientNet_B0_Weights

https://docs.pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR.html

<https://es.mathworks.com/help/deeplearning/ug/pretrained-convolutional-neural-networks.html>

<https://docs.pytorch.org/docs/stable/nn.html#loss-functions>

<https://www.kaggle.com/datasets/grassknoted/asl-alphabet>

<https://docs.pytorch.org/vision/main/models.html>

<https://docs.pytorch.org/docs/stable/optim.html>

<https://es.wikipedia.org/wiki/Sobreajuste>

<https://cayetanoguerra.github.io/ia/>

<https://peerj.com/articles/cs-2107/>

2025/2026

Fundamentos de los Sistemas Inteligentes