# Informe Final: Detección de enfermedades en la hoja de tomate

## 1. Introducción

En el presente trabajo, se exploraron diversas arquitecturas de deep learning para abordar el desafío de detectar enfermedades en la agricultura de precisión, específicamente en cultivos de tomate. El objetivo fue desarrollar modelos capaces de identificar de manera automatizada las enfermedades que afectan a los cultivos, utilizando tecnologías de visión artificial basadas en imágenes digitales.

El documento está estructurado de la siguiente manera:

- Descripción de la estructura de los notebooks entregados: En este capítulo se detalla la organización de los notebooks utilizados en el proyecto, brindando una visión general de cada uno de ellos y su función en el desarrollo de la solución.

- Descripción de la solución (arquitectura, preprocesado, etc.): En este capítulo se presenta en detalle la arquitectura del sistema propuesto, incluyendo las diferentes capas y componentes utilizados. Además, se describen los pasos de preprocesado de las imágenes y cualquier otro aspecto relevante de la solución implementada.

- Descripción de las iteraciones realizadas: Este capítulo describe las diferentes iteraciones y mejoras que se realizaron en el proyecto. Se explican los cambios y ajustes realizados en las arquitecturas, así como las decisiones tomadas en cada etapa del proceso de desarrollo.

- Descripción de los resultados: En este capítulo se presentan y analizan los resultados obtenidos a partir de la evaluación de los modelos desarrollados. Se discuten las métricas utilizadas, se comparan los rendimientos y se destacan los logros alcanzados en la detección de enfermedades en los cultivos de tomate.

En resumen, este trabajo se enfoca en explorar y proponer soluciones basadas en deep learning para la detección de enfermedades en la agricultura de precisión, específicamente en cultivos de tomate. Se presenta una estructura clara y organizada que abarca desde la descripción de los notebooks hasta los resultados obtenidos, proporcionando una visión integral del desarrollo del proyecto.

## 2. Descripción de la estructura de los notebooks entregados

A continuación, se listan y explican los notebooks entregados y que se pueden encontrar en el repositorio de github *https://github.com/keonij1/Tomato-Leaf-Disease-Detection* .

- *01 - Exploración y preprocesado de datos.ipynb*
-

Este notebook esta dividido en tres bloques, los cuales se listan y explican a continuación:

1) *Importamos los recursos necesarios:* En este bloque se importan todas las librerías necesarias para implementar la solución y se clonan los datos a usar. Estos datos se clonan del Github al entorno de Colab
2) *Exploración y Visualización de Datos:* En este bloque se leen y se muestran 4 imágenes de cada una de las 10 clases a usar.
3) *Procesamiento de Datos:* En este punto se procesan los datos, se realiza aumentación con el fin de tener mas datos y mejorar el entrenamiento de los modelos. Se separan los datos para entrenamiento y para validación

- *02 - evaluación de 4 arquitecturas.ipynb*
-

En este notebook se evalúan 4 arquitecturas, con el fin de seleccionar las 2 arquitecturas que presenten mejores resultados, con el fin de realizar iteraciones con ellas. Este notebook está dividido en los siguientes 6 bloques:

*1)Exploración de recursos y datos:* En este bloque se importan todos los recursos y datos.
*2) ARQUITECTURA # 1:* Implementación de la primera arquitectura.
*3) ARQUITECTURA # 2:* Implementación de la segunda arquitectura.
*4) ARQUITECTURA # 3:* Implementación de la tercera arquitectura.
*5) ARQUITECTURA # 4:* Implementación de la cuarta arquitectura.
*6) Resumen de resultados:* Se listan los resultados de la precisión del modelo con los datos de validación y se seleccionan las mejores dos arquitecturas.

- *03 - Iteraciones-Arquitectura 3.ipynb*
-

En este Notbook se realizan 4 iteraciones en la arquitectura # 3. Estas iteraciones consisten en realizar 4 modificaciones a la arquitectura y evaluar su desempeño.

- *04 - Iteraciones-Arquitectura 4.ipynb*

En este Notbook se realizan 4 iteraciones en la arquitectura # 4. Se utiliza el mismo concepto de los cambios realizados a la arquitectura # 3.

## 3. Descripción de la solución (arquitectura, preprocesado, etc.)

### a. Datos

El Dataset utilizado fue tomado de la web https://www.kaggle.com/ y consta de 11.000 imágenes (199 MB) de planta de tomate distribuidas de la siguiente forma:

- *Datos de entrenamiento:* 10.000 imágenes distribuidas en 10 clases, donde cada clase tiene asociada 1.000 imágenes.
- *Datos de Validación:* 1.000 imágenes distribuidas en 10 clases, donde cada clase tiene asociada 100 imágenes.

Estos datos se pueden obtener y hacer disponibles en los notebooks, clonando los datos que se encuentran en el repositorio en el entorno de cola. Esto se hace con el código mostrado en la Figura 1.

```
# URL del repositorio de GitHub
repo_url = 'https://github.com/keonij1/Tomato-Leaf-Disease-Detection'

# Directorio de destino para clonar el repositorio
destination_folder = '/Data'

# Verificar si el directorio de destino ya existe
if not os.path.exists(destination_folder):
    # Clonar el repositorio si el directorio no existe
    git.Repo.clone_from(repo_url, destination_folder)
else:
    print("El repositorio ya ha sido clonado anteriormente.")
```

*Figura 1 : Código para clonar los datos de un github a un entorno Colab*

### b. Preprocesado

En el procesamiento de datos se realizaron 5 pasos descritos en el notebook y resumidos a continuación:

- *Paso 1:* Se define una función llamada convert_image_to_array que toma la ruta de un archivo de imagen como entrada y devuelve la imagen convertida en forma de matriz numpy.

- *Paso 2:* Se carga imágenes de enfermedades de plantas en una lista (image_list) y las correspondientes etiquetas de enfermedades en otra lista (label_list).

- *Paso 3:* Se crea un objeto LabelBinarizer que se utilizará para codificar las etiquetas de las imágenes. Aplicamos la transformación de etiquetas a través del método fit_transform() del objeto LabelBinarizer para obtener una representación binaria de las etiquetas

- *Paso 4:* Separamos los datos en datos de entrenamiento y de validacion.

- *Paso 5:* En este paso se crea un objeto aug de la clase ImageDataGenerator de Keras, el cual se utiliza para aplicar técnicas de aumento de datos (data augmentation) en las imágenes. Estas técnicas se utilizan para aumentar la diversidad y la cantidad de datos de entrenamiento, lo que puede ayudar a mejorar el rendimiento y la generalización del modelo.

### c. Arquitecturas Usadas

Se evaluaron 4 arquitecturas diferentes y se describe en la Tabla 1 y la Tabla 2.

*Tabla 1 : Estructura de las arquitecturas 1 y 2.*

| ARQUITECTURA # 1 | ARQUITECTURA # 2 |
|---|---|
| Capa convolucional (Conv2D) con 16 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. |
| Capa BatchNormalization (axis=chanDim). | Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. |
| Capa convolucional (Conv2D) con 16 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa de MaxPooling2D con un tamaño de ventana de (2, 2). |
| Capa BatchNormalization | Capa de Dropout con una tasa de 0.25. |
| Capa de MaxPooling2D con un tamaño de ventana de (2, 2). | Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. |
| Capa de Dropout con una tasa de 0.25. | Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. |
| Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa de MaxPooling2D con un tamaño de ventana de (2, 2). |
| Capa BatchNormalization | Capa de Dropout con una tasa de 0.25. |
| Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa de aplanamiento (Flatten) para convertir los mapas de características en un vector unidimensional. |
| Capa BatchNormalization) | Capa densa (Dense) con 512 neuronas. Activación ReLU. |
| Capa de MaxPooling2D con un tamaño de ventana de (2, 2). | Capa de Dropout con una tasa de 0.5. |
| Capa de Dropout con una tasa de 0.25. | Capa densa (Dense) con el número de clases como salida. Activación softmax |
| Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | |

| | |
|---|---|
| Capa BatchNormalization | |
| Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | |
| Capa BatchNormalization | |
| Capa de MaxPooling2D con un tamaño de ventana de (2, 2). | |
| Capa de Dropout con una tasa de 0.25. | |
| Capa de aplanamiento (Flatten) | |
| Capa densa (Dense) con 256 neuronas. Activación ReLU. | |
| Capa BatchNormalization | |
| Capa de Dropout con una tasa de 0.5. | |
| Capa densa (Dense) con el número de clases como salida. Activación Softmax. | |

*Tabla 2 Estructura de las arquitecturas 3 y 4.*

| **ARQUITECTURA # 3** | **ARQUITECTURA # 4** |
|---|---|
| Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3). Activación ReLU. |
| Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa de pooling (MaxPooling2D) con un tamaño de ventana de (2, 2). |
| Capa convolucional (Conv2D) con 32 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3). Activación ReLU. |
| Capa de MaxPooling2D con un tamaño de ventana de (2, 2). | Capa de pooling (MaxPooling2D) con un tamaño de ventana de (2, 2). |
| Capa de Dropout con una tasa de 0.25. | Capa de aplanamiento (Flatten) para convertir los mapas de características en un vector unidimensional. |
| Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa completamente conectada (Dense) con 128 neuronas. Activación ReLU. |
| Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | Capa de salida (Dense) con 10 neuronas. Activación softmax" |
| Capa convolucional (Conv2D) con 64 filtros, tamaño del filtro (3, 3) y relleno (padding) "same". Activación ReLU. | |
| Capa de MaxPooling2D con un tamaño de ventana de (2, 2). | |
| Capa de Dropout con una tasa de 0.25. | |
| Capa de aplanamiento (Flatten) para convertir los mapas de características en un vector unidimensional. | |
| Capa densa (Dense) con 512 neuronas. Activación ReLU. | |
| Capa de Dropout con una tasa de 0.5. | |

| | |
|---|---|
| Capa densa (Dense) con el número de clases como salida. Activación softmax. | |
| | |

## 4. Descripción de las iteraciones realizadas

Esta sección explicaremos las iteraciones realizadas. Inicialmente se evaluaron as 4 arquitecturas base descritas anteriormente. De esta evaluación se tomaron las dos arquitecturas con mejor desempeño y en ambas se realizaron las mismas 4 iteraciones. Estas iteraciones se describen a continuación:

- *Iteración 1:* Se duplica el número de filtros en las capas convolucionales.
- *Iteración 2:* Se cambia el tamaño del kernel en las capas convolucionales, en este caso se usa (5, 5) en lugar de (3, 3).
- *Iteración 3:* Se agregan capas de normalización por lotes (Batch Normalization) después de las capas de convolución.
- *Iteración 4:* Se agrega una capa de regularización L2 (Weight Decay) a las capas totalmente conectadas (Dense).

## 5. Descripción de los resultados.

En las figuras 2, 3, 4 y 5 se presentan los resultados de la evaluación de las arquitecturas 1, 2, 3, y 4 respectivamente. De estos resultados observamos que la arquitectura 1 presenta indicios de sobre entrenamiento, con una precisión de 55.1% en validación. Esto mismo se presenta en menor medida en la arquitectura 2, que presento una precisión de 63.9% en validación. Este sobre entrenamiento no se presento en las arquitecturas 3 y 4, que presentaron una precisión en validación de 82.83% y 83.83% respectivamente. Debido a esto se seleccionan las arquitecturas 3 y 4 para realizar las iteraciones propuestas y descriptas en las secciones anteriores.

### *a. Resultados de las 4 Arquitecturas*

i. *Arquitectura # 1 --> Test Accuracy: 54.1 %*
ii. *Arquitectura # 2 --> Test Accuracy: 63.99%*
iii. *Arquitectura # 3 --> Test Accuracy: 82.83%*
iv. *Arquitectura # 4* **-->** *Test Accuracy: 83.83%*

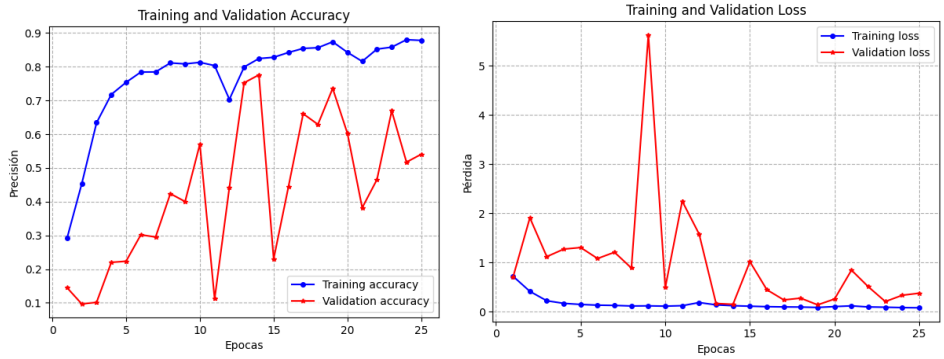


*Figura 2 : Resultados Arquitectura # 1*

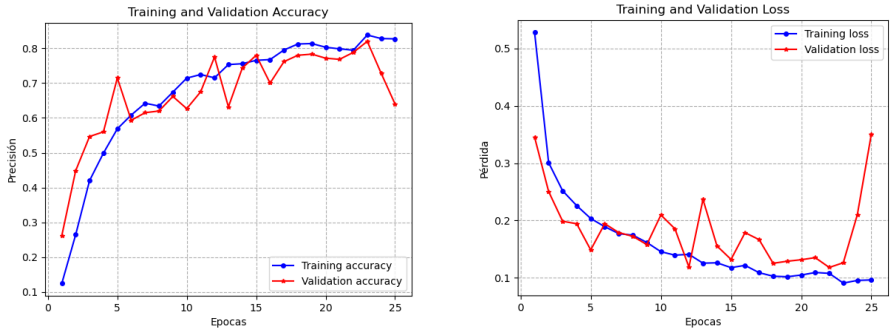


*Figura 3: Resultados Arquitectura # 2*

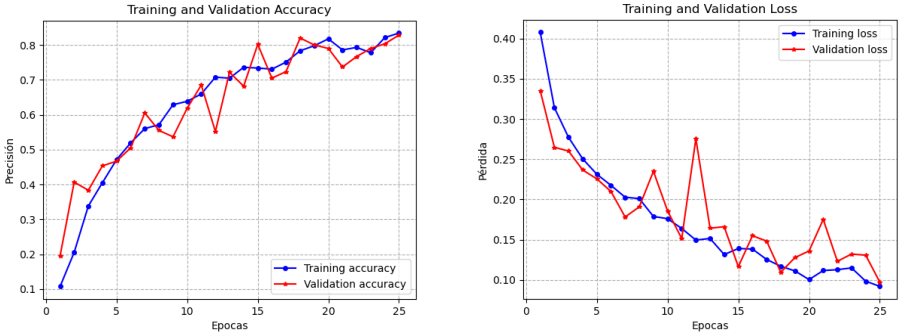


*Figura 4: Resultados Arquitectura # 3*

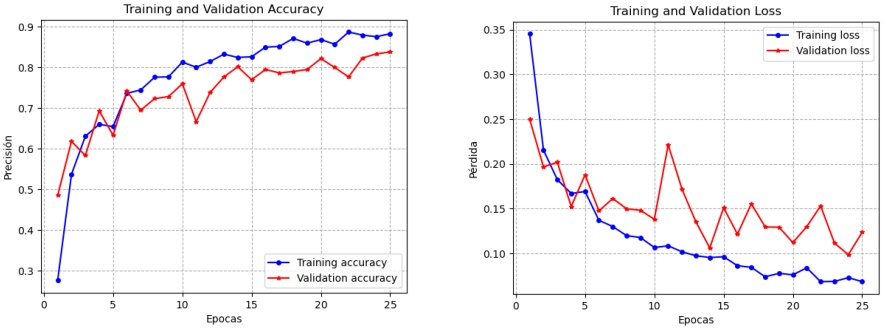


*Figura 5 : Resultados Arquitectura #*

En las figuras 6-9 se presentan los resultados de la evaluación de la iteraciones 1, 2, 3, y 4 de la arquitectura 3. En este caso podemos observar que ninguna de las iteraciones logro obtener mejor desempeño a la arquitectura base. Destacando que la iteracion 3, que corresponde a agregar capaz de normalización, logro un desempeño aceptable, estando 1% por debajo de la precisión de la arquitectura base. Por otro lado, la iteración 4, correspondiente al agregar una capa de regularización L2 (Weight Decay) a las capas totalmente conectadas (Dense), hizo que el modelo perdiera estabilidad y empeorara los resultados.

### ***b. Iteraciones en la Arquitectura 3***

i. *Iteración # 1 --> Test Accuracy: 70.49%*
ii. *Iteración # 2 --> Test Accuracy: 66.00%*
iii. *Iteración # 3 --> Test Accuracy: 81.83 %*
iv. *Iteración # 4* **-->** *Test Accuracy: 9.49%*

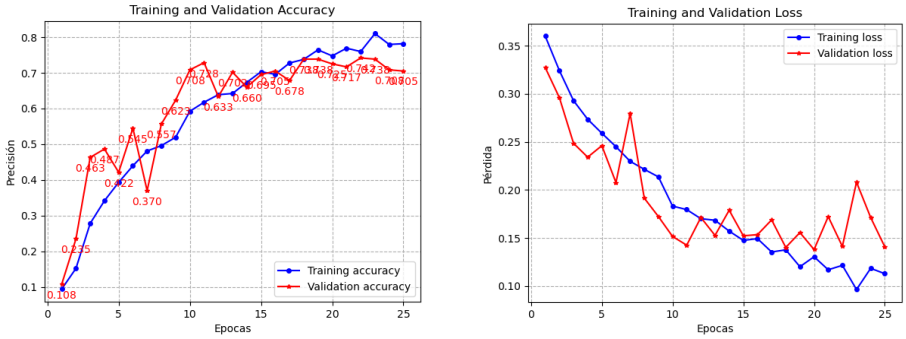


*Figura 6: Resultados iteración # 1 en la arquitectura # 3.*

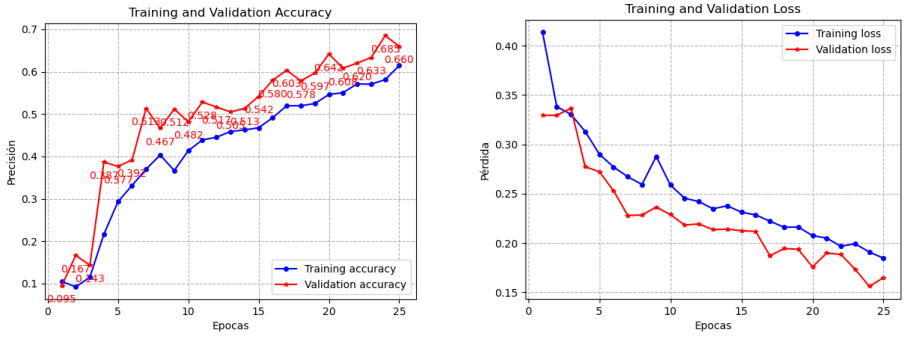


*Figura 7: Resultados iteración # 2 en la arquitectura # 3.*

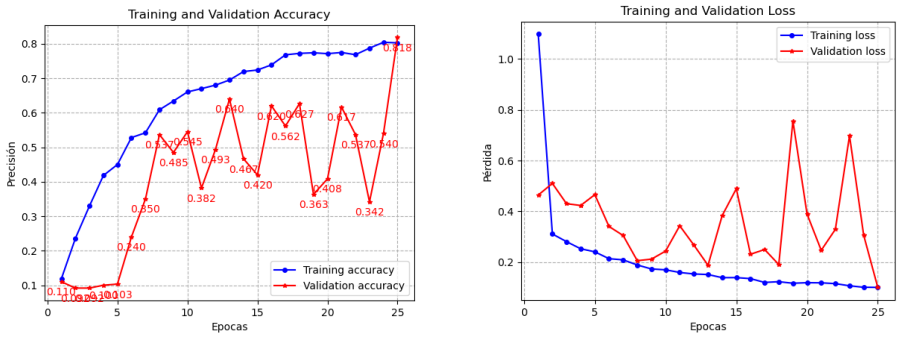


*Figura 8: Resultados iteración # 3 en la arquitectura # 3.*

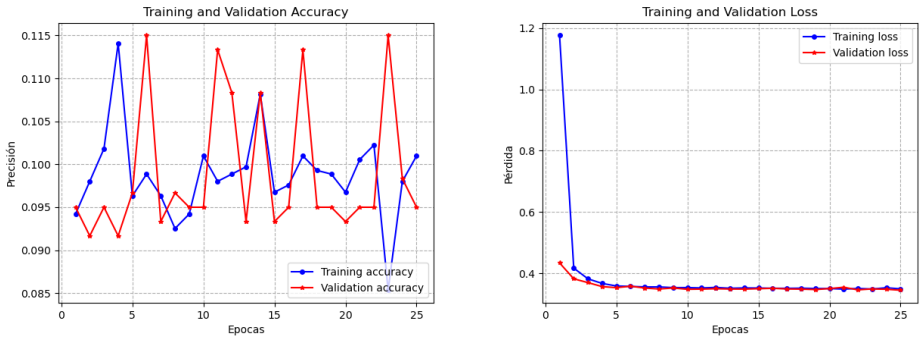


*Figura 9: Resultados iteración # 4 en la arquitectura # 3.*

En las figuras 10-12 se presentan los resultados de la evaluación de las iteraciones en la arquitectura 4. En este caso podemos observar en el caso de la iteración 2, correspondiente a cambiar el tamaño del kernel, presento una mejora en los resultados. Con esta iteración se logró pasar de *83.83% de precisión a 84.33 %. Al igual que en la arquitectura 3, la iteración 4 hizo que el modelo perdiera estabilidad y los resultados empeoraran considerablemente.*

### ***c. Iteraciones en la Arquitectura 4***

i. *Iteración # 1 --> Test Accuracy: 70.16%*
ii. *Iteración # 2 --> Test Accuracy: 84.33 %*
iii. *Iteración # 3 --> Test Accuracy: 66.63 %*
iv. *Iteración # 4* **-->** *Test Accuracy: 40.16 %*

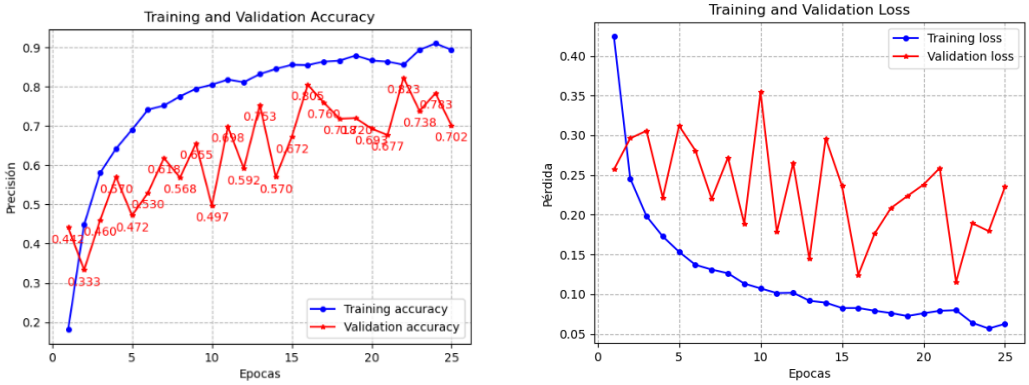


*Figura 10: Resultados iteración # 1 en la arquitectura # 4.*

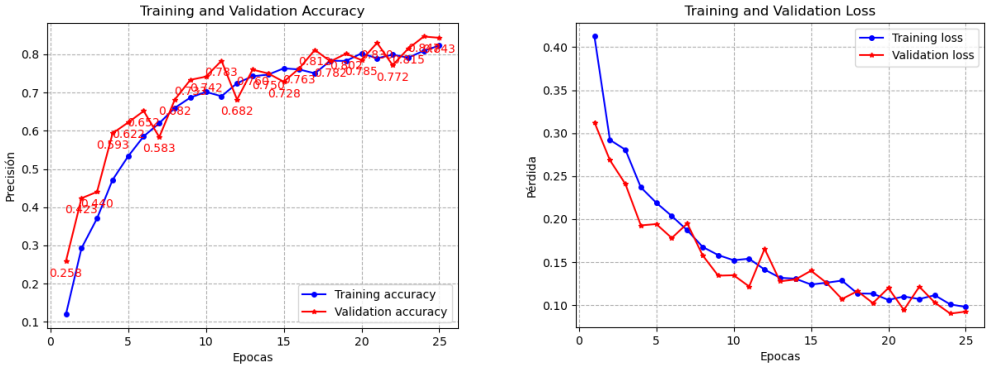


*Figura 11: Resultados iteración # 2 en la arquitectura # 4.*

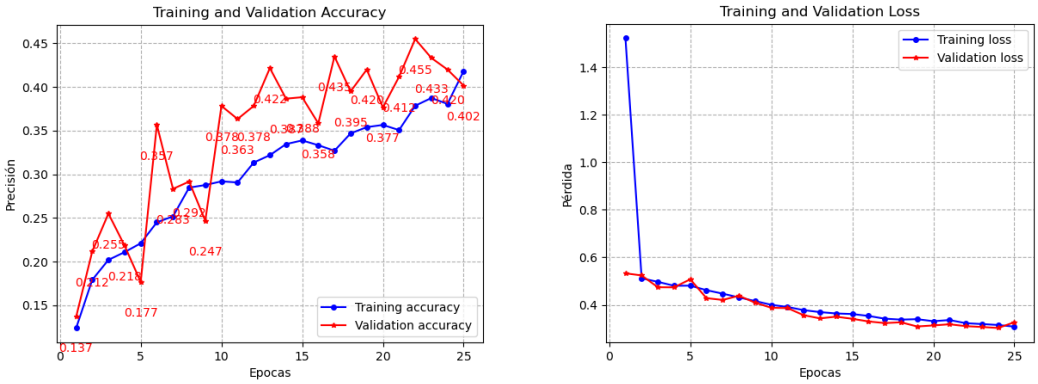


*Figura 12: Resultados iteración # 4 en la arquitectura # 4.*