

Modelo de clasificación de imágenes: piedra, papel, tijeras

Domingo Parrales de la Cruz

Agosto 2024

Índice

1. Introducción	2
2. Metodología	2
2.1. Carga de datos mediante la clase Sequence	2
2.2. Carga de datos	3
2.3. Construcción y entrenamiento de los modelos	4
2.3.1. Transfer Learning	4
2.3.2. Modelos propios	10
3. Resultados	14
3.1. Medidas de rendimiento de los modelos	14
3.1.1. Modelo propio con regularización Ridge	14
3.1.2. Modelo de Transfer Learning con la VGG16	15
3.2. Casos incorrectamente clasificados	17
3.3. Casos correctamente clasificados	18
3.3.1. Modelo de Transfer Learning con la VGG16	18
3.3.2. Modelo propio con regularización Ridge	19
3.4. Visualización de las activaciones de los filtros convolucionales en- trenados	21
3.4.1. Modelo de Transfer Learning con la VGG16	21
3.4.2. Modelo propio con regularización Ridge	23
3.5. Kernels y mapas de características de la primera capa convolucional	29
3.5.1. Modelo de Transfer Learning con la VGG16	29
3.5.2. Modelo propio con regularización Ridge	34
4. Conclusiones	39
5. Referencias	40

1. Introducción

El objetivo de este trabajo es emplear técnicas de Deep Learning para clasificar imágenes con tres posibles categorías: piedra, papel o tijeras. Para ello, empezamos construyendo modelos mediante "Transfer Learning" y seguimos con el diseño de un modelo desde cero, al que añadiremos regularización posteriormente.

No obstante, al trabajar con imágenes como entrada, es probable que encontremos problemas relacionados con la falta de memoria, ya que estaremos usando la GPU y, en mi caso, trabajaremos en una instalación local, por lo que este problema es más notable que si usaramos Google Colab, por ejemplo. La solución que desarrollaremos más adelante consistirá en el uso de una clase Sequence, que, a grandes rasgos, se encarga de suministrar las imágenes en demanda, en lugar de tenerlas todas cargadas en memoria simultáneamente.

Por último, vamos a hacer un estudio visual de los pesos de las capas convolucionales resultantes del entrenamiento de los modelos. Esto incluirá el uso de mapas de calor para interpretar las activaciones de la capa y la generación de patrones a partir de una imagen de ruido.

2. Metodología

El trabajo se basa principalmente en el uso de tensorflow y numpy. A partir de ellas hemos creado los modelos (`redes_neuronales.py`) y algunas funciones auxiliares (`funciones_estudio_modelo.py`), que se utilizan mayormente en la visualización de los resultados.

Concretamente, hemos trabajado con una instalación local en Windows, por lo que la versión de tensorflow con la que hemos trabajado es la 2.10.1, la más alta que se permite en este sistema operativo, sin usar dockers [B].

2.1. Carga de datos mediante la clase Sequence

Al estar trabajando en local, la memoria vram se convierte en un factor limitante para la construcción de modelos. Es por esto que vamos a diseñar una clase que nos permita "inyectar" las imágenes en demanda y trabajar en lotes, a los que nos referiremos como "batches". Esta clase hereda los atributos de la clase "Sequence" de tensorflow, por lo que, como mínimo debe contar con un atributo "`__init__`" para inicializar la clase, un atributo "`__getitem__`" que cargue las imágenes y un atributo "`__len__`" que calcule el número de batches por época [1].

La idea intuitiva es que, en lugar de cargar las imágenes en memoria, vamos a mantener un diccionario con la ruta de cada imagen y su categoría. Esta será una de las entradas de la clase que, además, contendrá todos los posibles tratamientos que vayamos a necesitar:

- **Desordenar imágenes al finalizar cada época (y antes de empezar la primera):** con esto buscamos disminuir el sesgo en el entrenamiento. Para hacer esto, tomamos una lista de índices y la desordenamos aleatoriamente que, después, usaremos para acceder a cada pareja “ruta-categoría” del diccionario.
- **Agrupación de imágenes en batches:** podemos hacerlo fácilmente con la lista de índices desordenados.
- **Aumento de datos:** en el caso de que se indique, es posible aplicar modificaciones a la imagen antes de inyectarla al modelo. [3] Estas incluyen, por ejemplo, rotaciones de hasta 90°, zoom y cambios en el brillo de la foto.
- **Cambio del tamaño de la imagen:** esto, además de ser necesario como método adicional para reducir el uso de memoria, es un parámetro que puede afectar notablemente al aprendizaje de la red neuronal.

A esta clase que hemos creado nos referiremos como “**RockPaperScissorsSequence**”.

2.2. Carga de datos

Para hacer un tratamiento correcto de los modelos, hemos optado por hacer 3 particiones:

- **Conjunto de entrenamiento** (80 % de los datos) y **conjunto de validación** (6 % de los datos): para entrenar el modelo.
- **Conjunto de test** (14 % de los datos): lo mantenemos al margen del entrenamiento y, después de hacerlo, lo usamos para medir el rendimiento del modelo.

Uno de los problemas que podemos encontrar al repartir de esta manera los datos es la inestabilidad de la función loss debido a la escasa cantidad de datos en el conjunto de validación [4]. No obstante, como veremos más adelante, este no es un problema significativo, por lo que vamos a mantenerlo de esta manera. Por contra, una de las virtudes de esta repartición es que estamos reservando una gran cantidad de datos para la evaluación final del modelo, mediante el conjunto de test.

El primer paso que hay que tomar tras cargar los datos es transformarlos a la estructura de diccionario que necesita la clase Sequence. Para ello, tomamos como clave la ruta de la imagen y como valor la clase (piedra, papel o tijeras). Después, el diccionario formado por todas las imágenes disponibles, hacemos la partición train-validation-test utilizando la librería sklearn. Y, finalmente, creamos tres objetos de la clase RockPaperScissorsSequence, uno para cada partición, para lo que tendremos que especificar el tamaño de los batches y las dimensiones de las imágenes.

Además, cabe destacar que cada uno de los tres objetos RockPaperScissorsSequence se comportará de manera diferente dependiendo de su función:

- **Conjunto de entrenamiento:** incluye aumento de datos, para dar generalidad al modelo, y desorden de las imágenes, para evitar el sesgo.
- **Conjunto de validación:** no se usa aumento de datos, para evaluar sobre los casos reales, pero sí se desordenan los casos.
- **Conjunto de test:** ni se aumentan los datos ni se desordenan los casos.

2.3. Construcción y entrenamiento de los modelos

Tenemos varias técnicas para construir redes neuronales que predigan con precisión los datos. La primera que vamos a ver en este trabajo es el “Transfer Learning”, que consiste en cargar los pesos de una red neuronal previamente entrenada con otro conjunto de datos. A esta, le quitaremos las capas superiores para añadir las que requiere nuestro conjunto de datos, que, al menos, será una capa densa de 3 neuronas.

Por otro lado, hemos construido desde cero un modelo siguiendo las pautas de [5], que podemos resumir en que debemos empezar por el modelo más simple posible y, a partir de él, construir uno más complejo si nuestro sistema lo necesita.

En cada caso, veremos dos aproximaciones al problema.

2.3.1. Transfer Learning

Vamos a construir dos modelos mediante Transfer Learning: uno basado en una VGG16 y otro en una ResNet. Ambos son modelos relativamente simples y sus pesos se obtienen del mismo conjunto de datos, pero esperamos que haya diferencias notables entre ambos debido al cambio de arquitectura.

El procedimiento que hemos seguido puede verse explícitamente en el fichero “redes_neuronales.py”, pero, a modo de resumen, lo que se ha hecho para ambos es lo siguiente:

1. Cargamos la red con los pesos del conjunto “imagenet”, especificando además: el tamaño de las imágenes de entrada, que no se incluya la parte superior y que estas capas no sean entrenables.
2. Construimos el modelo usando la clase “tf.keras.models.Sequential()” y su atributo “Add()”.
3. Añadimos una capa “Input” al fondo, para controlar las entradas del modelo.
4. Añadimos sobre esta la red cargada anteriormente.

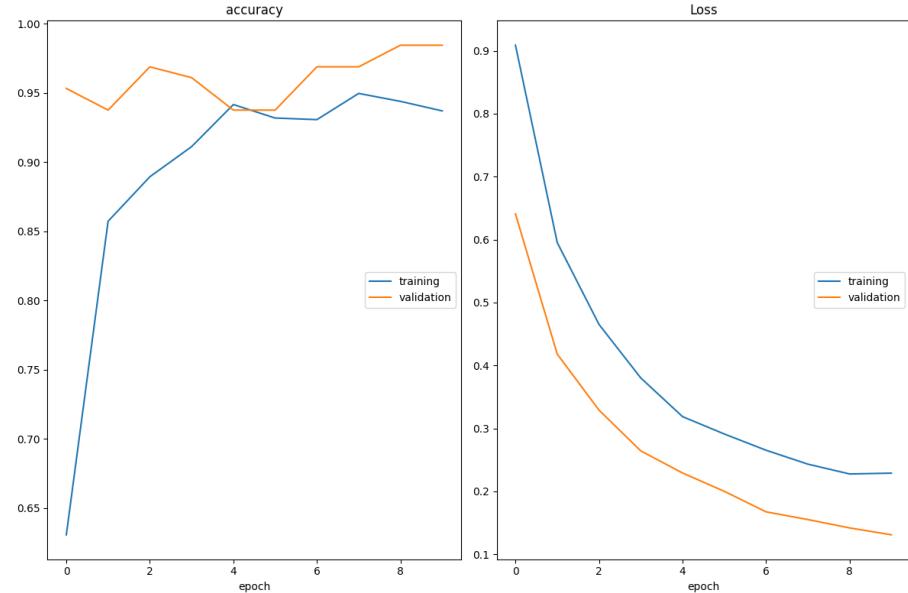
5. Añadimos las capas superiores necesarias para dar la salida de nuestro problema. En nuestro caso, hemos elegido un “GlobalAveragePooling2D” seguido de una capa densa de 3 neuronas con activación “softmax”.

También tenemos capas “Dropout”, que se encargan de desactivar algunas neuronas de la capa anterior, con el objetivo de que existan varias neuronas capaces de abordar un mismo problema, reduciendo la probabilidad de que se dé sobreajuste.

VGG16: nuestra primera elección de red para hacer transfer learning ha sido la VGG16, ya que, además de ser la que hemos trabajado en la asignatura [Práctica 5], es relativamente sencilla comparada con otras como la Inception o la ResNet [7]. Podemos ver su estructura [8][Práctica 6] en la Figura 1.

Podemos ver que es lineal y está mayoritariamente compuesta por capas convolucionales.

Ahora pasamos al entrenamiento, que lo dividiremos en dos partes. En la primera mantendremos las capas de la VGG16 “congeladas”, por lo que solo modificaremos los pesos de las capas superiores que hemos añadido. Veamos los resultados:



Podemos ver que desde las primeras épocas el modelo obtiene unos resultados excepcionales. Esto se debe por un lado a los pesos heredados del conjunto “imagenet”, pero también a la arquitectura de la red.

Para la segunda parte del entrenamiento vamos a “descongelar” algunas de las capas superiores de la red VGG16 para hacer Fine Tuning. Para ello, te-

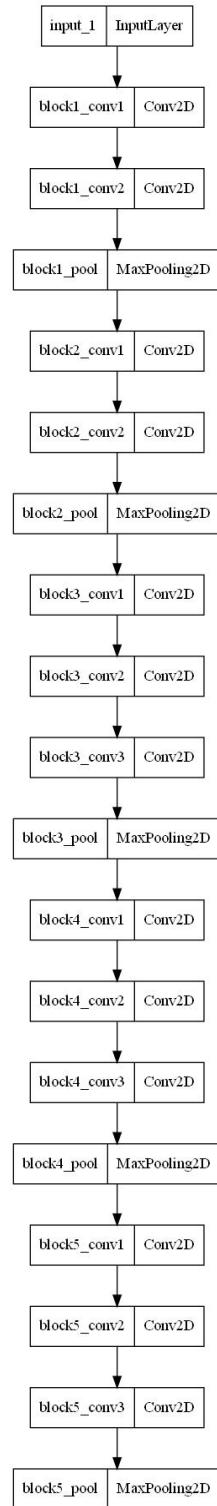


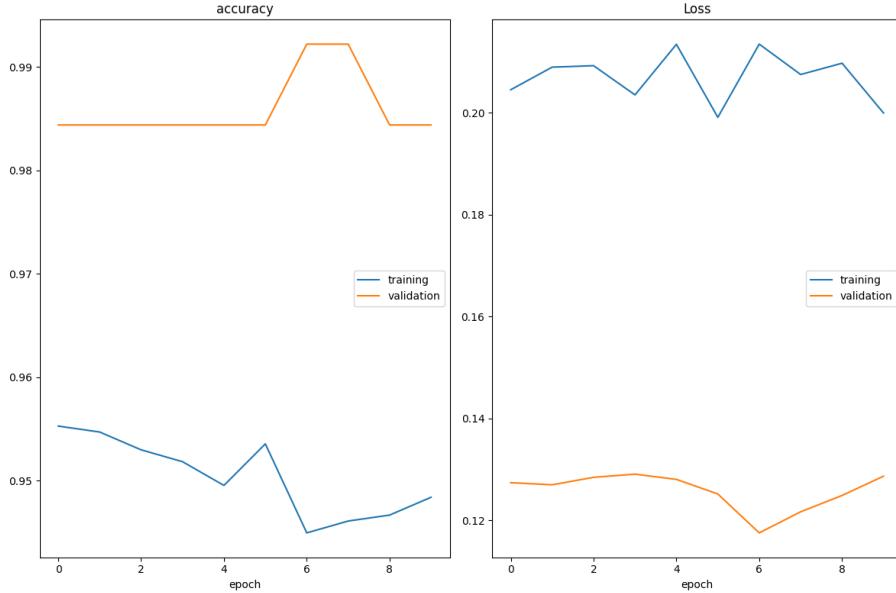
Figura 1: Arquitectura de la red VGG16.

nemos que reducir la **tasa de aprendizaje**, aunque es fundamental ajustarla correctamente, ya que puede tener varios efectos sobre la función loss durante el entrenamiento:

- **Tasa de aprendizaje demasiado grande:** la función loss será inestable y, posiblemente, no se aprecie una disminución clara.
- **Tasa de aprendizaje demasiado pequeña:** la función loss disminuye lentamente, por lo que necesitamos aumentar el número de épocas. Además, es posible que se quede en un mínimo local, dando resultados peores que los que el modelo es capaz de dar en las condiciones óptimas. Aun así, tenemos formas de evitar esto último, como el momento, que viene incorporado por defecto en el optimizador Adam (el que usamos en este trabajo).

Para elegirlo correctamente no hay un método infalible. Por ejemplo, en nuestro caso, hemos probado valores de tasa de aprendizaje entorno a 10^{-3} para el entrenamiento inicial y entorno a 10^{-5} para el Fine Tuning. No obstante, existen métodos como el que se muestra en la referencia [9] que pueden ser de ayuda, aunque trabaja sobre la suposición de que un loss bajo en una época concreta se traduce en un loss bajo para todas las épocas, lo que puede no ser cierto.

Los resultados obtenidos de hacer Fine Tuning son los siguientes:



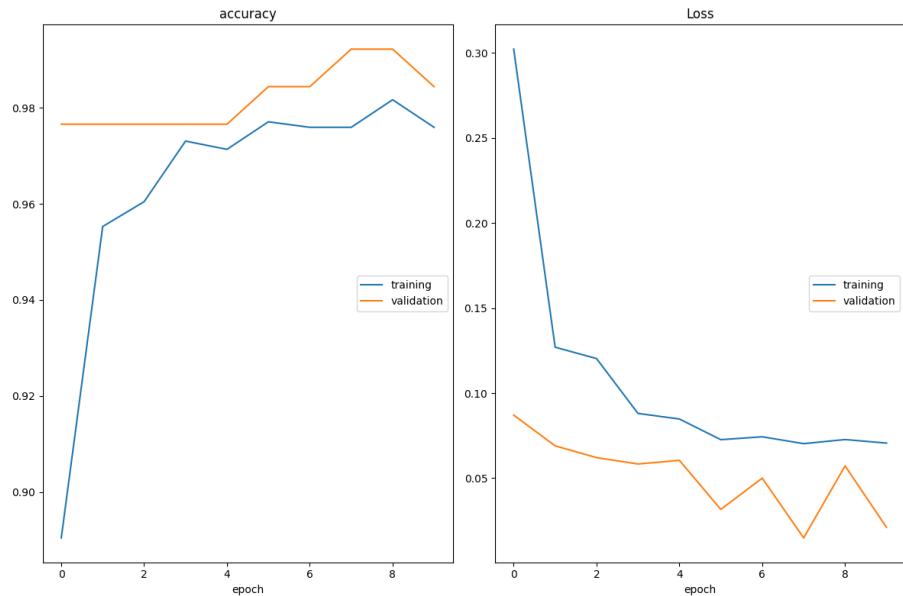
ResNet: ahora vamos a comparar el desempeño de la VGG16 con el de la red ResNet, utilizando el mismo procedimiento y la misma estructura de las capas

superiores. En primer lugar, fijémonos en las diferencias en la arquitectura de la red tomando como referencia la Figura 2.

Lo más llamativo de la ResNet, en comparación con la VGG16, es que es mucho más profunda. Sin embargo, por esto mismo es de esperar que haya problemas como el desvanecimiento del gradiente, lo que ralentiza o incluso detiene el entrenamiento debido a los valores tan bajos que alcanza.

Es por esto que también podemos ver bifurcaciones en la red, que funcionan como bloques residuales [10]. Es decir, al tener un segundo camino con un número muy inferior de capas o incluso ninguna, el efecto sobre el gradiente es menor, por lo que estamos permitiendo que el gradiente alcance profundidades mayores sin desvanecerse.

Pasemos al entrenamiento del modelo con las capas de la ResNet congeladas:



En este caso se han alcanzado valores excepcionales incluso más rápido que la VGG16, lo que es de esperar, ya que puede realizar un análisis más detallado al contar con una red más profunda.

Los resultados del Fine Tuning de este modelo son:

Zona cercana a la entrada de la red ResNet

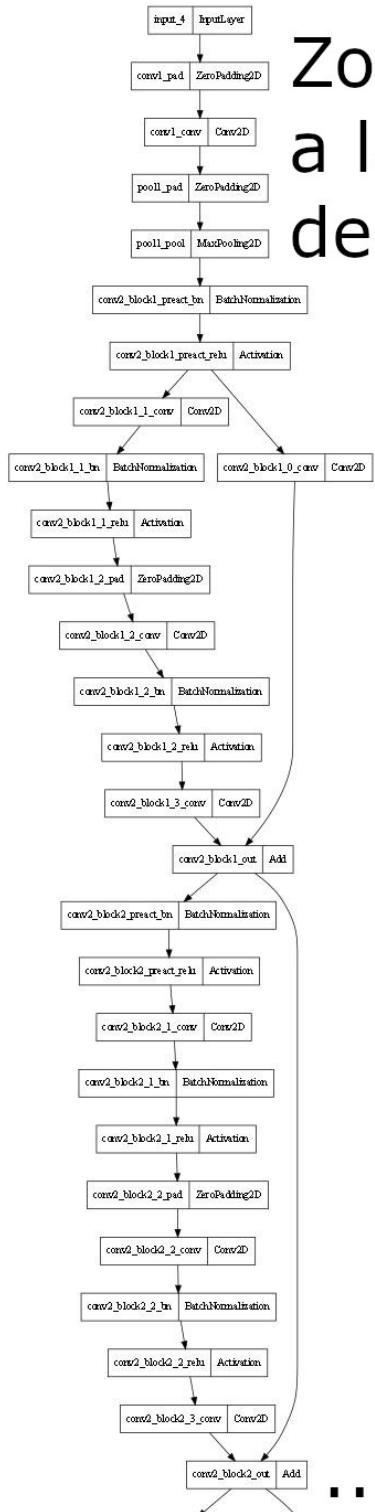
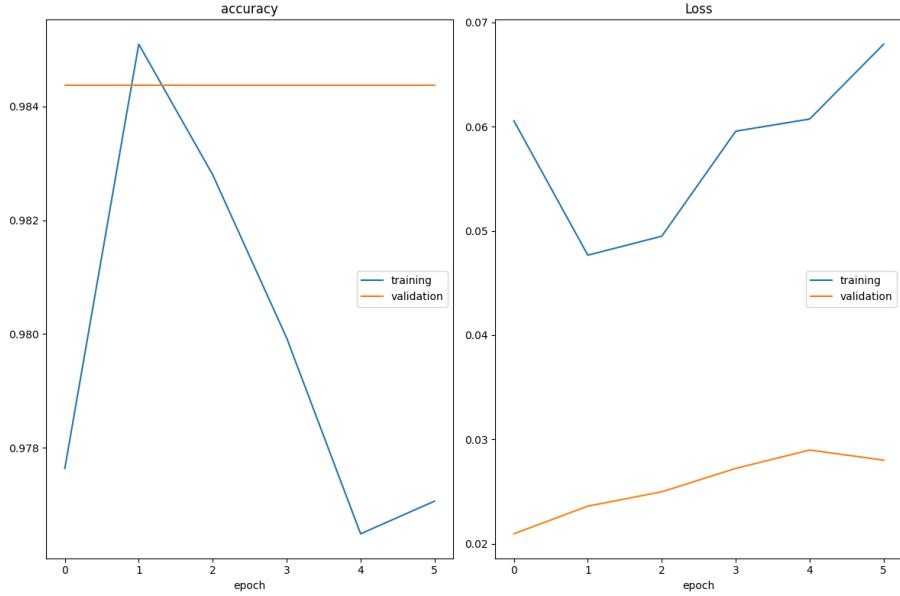


Figura 2: Arquitectura de la red ResNet. Se extiende mucho más, pero se ha representado el inicio únicamente, ya que contiene suficiente información para hacer la comparación.



2.3.2. Modelos propios

Para el estudio del modelo que hemos construido desde cero [5], vamos a tomar una aproximación diferente. En vez de plantear dos arquitecturas diferentes, como en el apartado anterior, vamos a estudiar el efecto que tiene la regularización Ridge (L2) sobre exactamente el mismo modelo, que tiene la estructura que mostramos en la Figura 3.

Como podemos ver, se sigue una idea similar a la de la VGG16, salvo porque hemos incluido una bifurcación poco antes de empezar con las capas densas. No obstante, estas no se comportan como bloques residuales como en la ResNet, sino que analizan la imagen en dos escalas diferentes, como ocurre en las redes “Inception” [11]. Es por esto que la única diferencia entre ambos caminos es la capa convolucional, siendo 7×7 en una y 3×3 en la otra.

Modelo sin regularización Ridge: la primera versión del modelo solo incluye el Dropout como técnica de regularización. Veamos cómo desempeña con una tasa de aprendizaje de 10^{-3} :

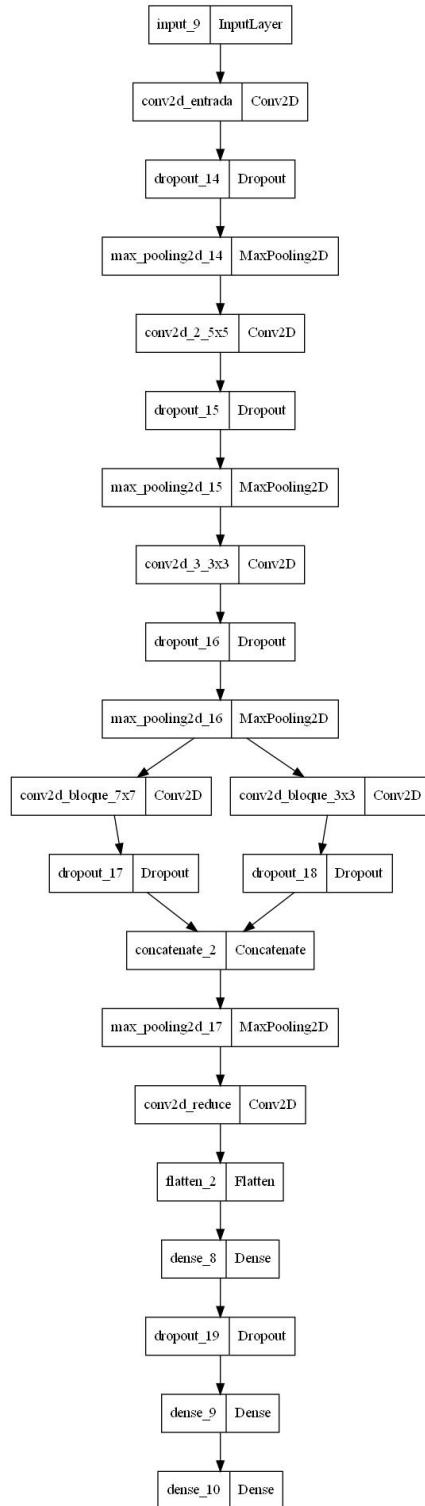
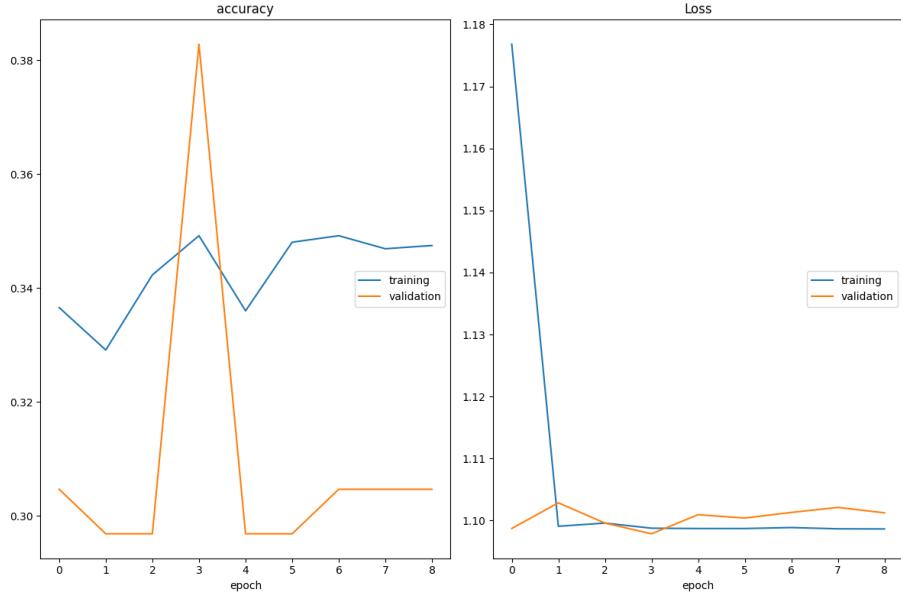
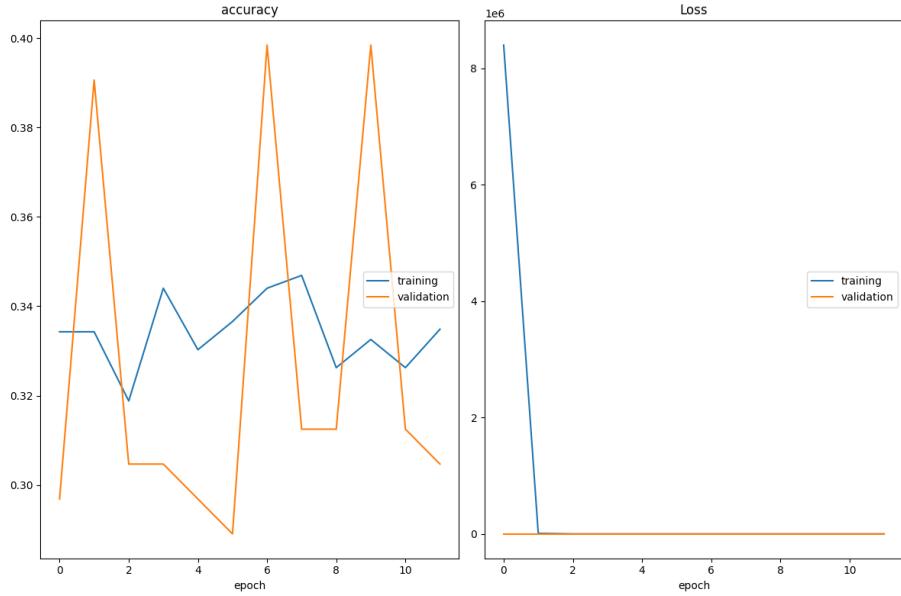


Figura 3: Arquitectura diseñada desde cero. Presenta un bloque similar al Inception.

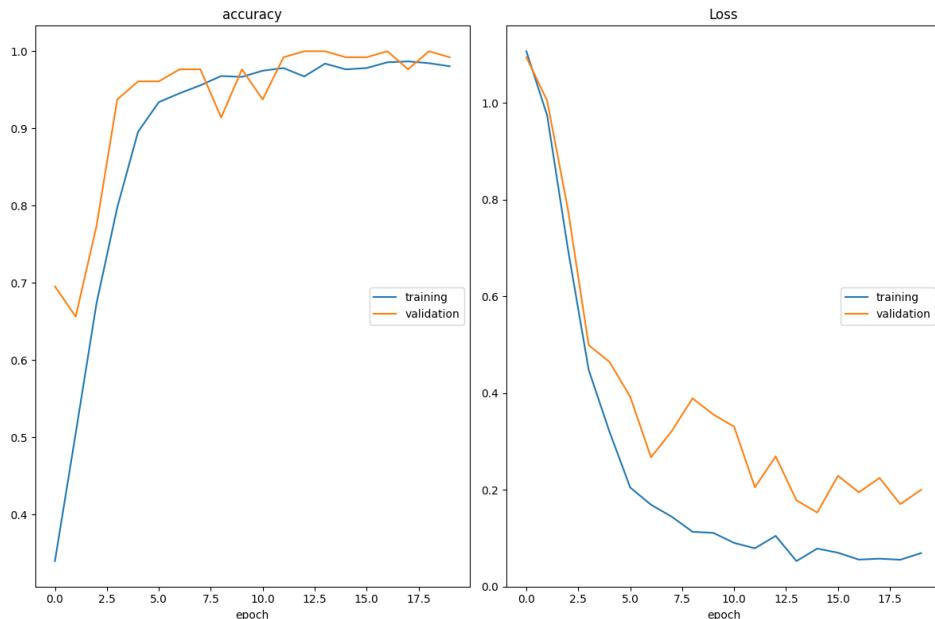


Podemos ver que se “estanca” en valores altos de la función loss, lo que puede deberse a que haya quedado atrapado en un mínimo local. Una posible solución es aumentar la tasa de aprendizaje, por ejemplo, pasando a 10^{-2} . Sin embargo, este valor sigue sin ser suficiente y aumentarlo más nos provoca inestabilidades. Por ejemplo, vemos qué ocurre cuando establecemos la tasa de aprendizaje como $5 \cdot 10^{-2}$:



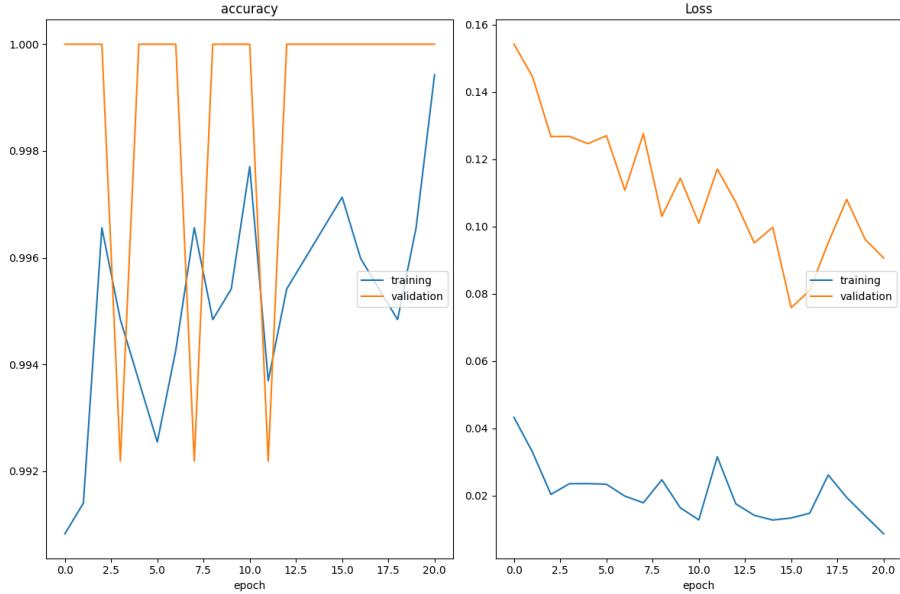
Podemos ver como la precisión cambia drásticamente entre épocas y, además, no se han recorrido las 30 épocas establecidas en el código, debido a que el “EarlyStopping” ha detenido la ejecución al no haber mejoras significativas en la función loss.

Modelo con regularización Ridge: ante el problema anterior, nos planteamos utilizar regularización Lasso, regularización Ridge o ambas. Basándome en [12], elegí la regularización Ridge, ya que la penalización que aplica previene que los pesos tomen valor cero, aunque se mantengan pequeños. Teniendo esto en mente, aplique sobre la capa “conv2d_entrada” la regularización, ya que es la convolucional más profunda de mi modelo. De esta forma, y con una tasa de aprendizaje de 10^{-4} , se obtuvo el siguiente resultado:



Como vemos, los resultados son mucho mejores respecto al caso anterior. De hecho, se llega a alcanzar el 100% de acierto en validación, aunque también cuenta con un mayor número de épocas. También, comparando con los resultados del transfer learning, se aprecia que la función loss empieza en valores mucho más altos, que se debe a que tenemos que entrenar los pesos desde cero.

Vamos a tratar de mejorar los resultados mediante Fine Tuning:



Hemos conseguido reducir la función loss ligeramente, aunque el proceso se ha detenido mucho antes de lo esperado por el EarlyStopping de nuevo. No obstante, la función loss parecía ir en aumento en las últimas épocas, por lo que puede haberse evitado una situación de sobreajuste, aunque los resultados de entrenamiento todavía no parecían mejorar a costa de que empeoren los de la validación.

3. Resultados

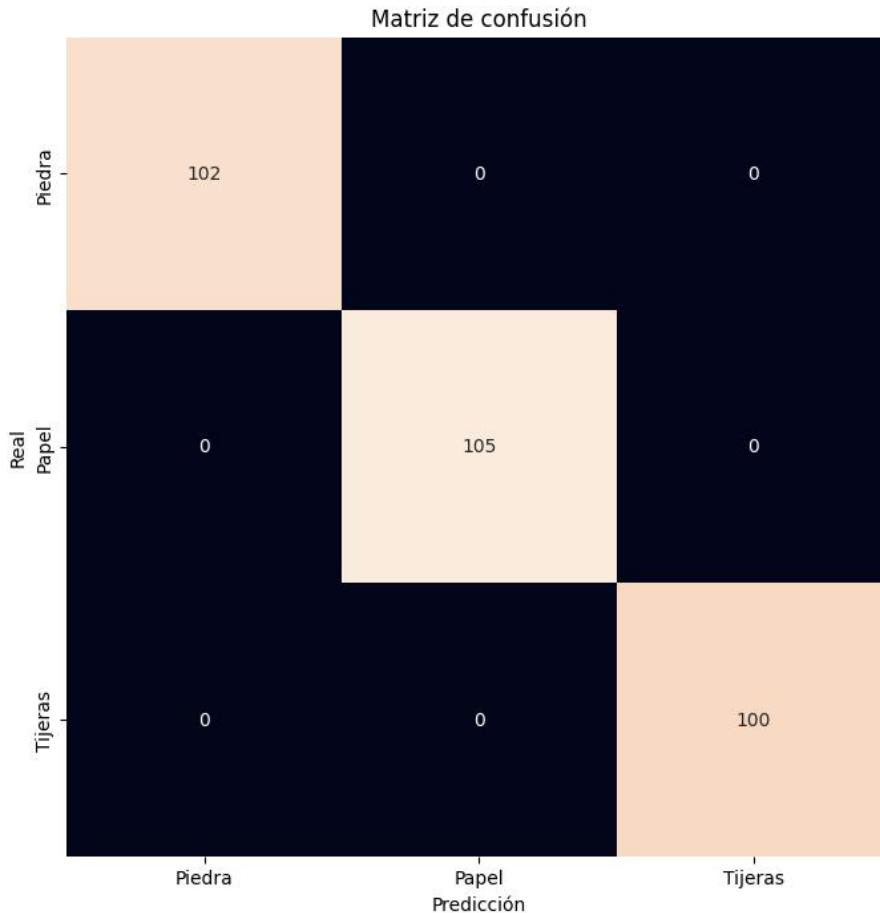
Nos centraremos, a continuación, en los resultados que surgen del modelo que hemos diseñado (con regularización Ridge) y del construido mediante transfer learning con la VGG16.

En primer lugar, veremos las medidas de rendimiento hechas a partir del conjunto de test, que no ha sido expuesto a los modelos en ningún momento. Después, analizaremos heatmaps correspondientes a las activaciones de las capas convolucionales entrenadas sobre algunas de las imágenes. Por último, visualizaremos los filtros de estas capas haciéndolos actuar sobre imágenes de ruido y aplicándolos sobre una imagen.

3.1. Medidas de rendimiento de los modelos

3.1.1. Modelo propio con regularización Ridge

Comenzando por nuestro modelo, podemos ver que también se consigue un 100 % de acierto en test:

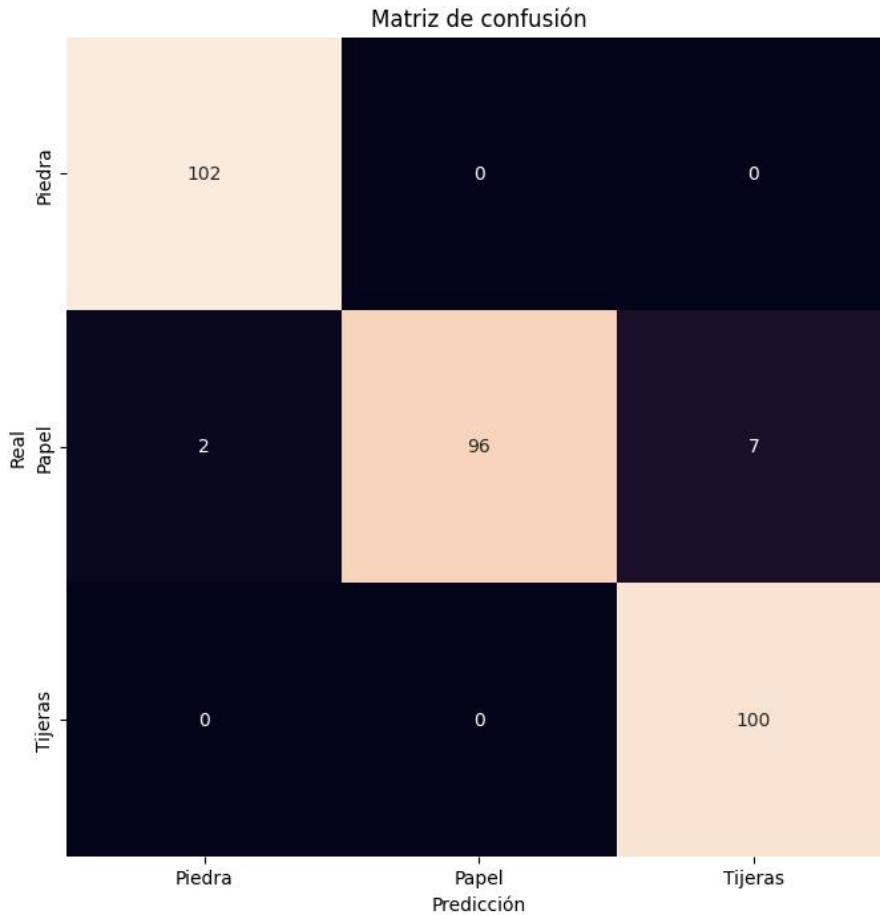


Esto también podemos visualizarlo en la siguiente tabla:

Medidas de rendimiento					
Accuracy (sobre test): 100.0%					
	Precision	Recall	F1-score	Número de casos	
Piedra	1.0	1.0	1.0	102.0	
Papel	1.0	1.0	1.0	105.0	
Tijeras	1.0	1.0	1.0	100.0	

3.1.2. Modelo de Transfer Learning con la VGG16

Ahora podemos ver un caso en el que no tenemos un acierto del 100 %, aunque esto se deba principalmente a que no le hemos dado épocas suficientes al entrenamiento:



Podemos ver que la única categoría en la que se cometan errores es en papel, cosa que analizaremos en el siguiente apartado. Veamos los resultados con mayor detalle:

Medidas de rendimiento				
Accuracy (sobre test): 97.07%				
	Precision	Recall	F1-score	Número de casos
Piedra	0.980769	1.000000	0.990291	102.0
Papel	1.000000	0.914286	0.955224	105.0
Tijeras	0.934579	1.000000	0.966184	100.0

Podemos ver que no se cometan errores al clasificar una imagen como “Papel”, pero la tasa de acierto no es tan buena como la de las otras dos categorías.

3.2. Casos incorrectamente clasificados

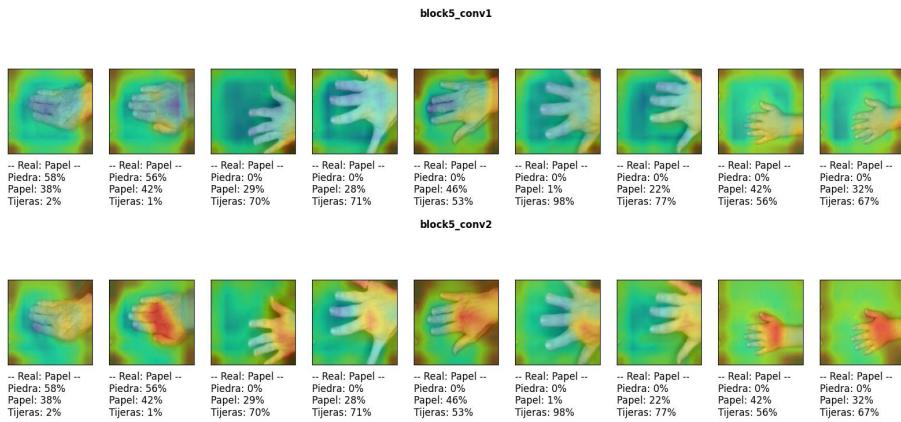
Vamos a tratar de interpretar el motivo de los errores de clasificación cometidos en el modelo que hacemos con la VGG16 (a falta de clasificaciones incorrectas en el modelo propio). Para ello, vamos a empezar por visualizar las imágenes erróneas con la probabilidad que ha estimado el modelo de que pertenezcan a cada clase:

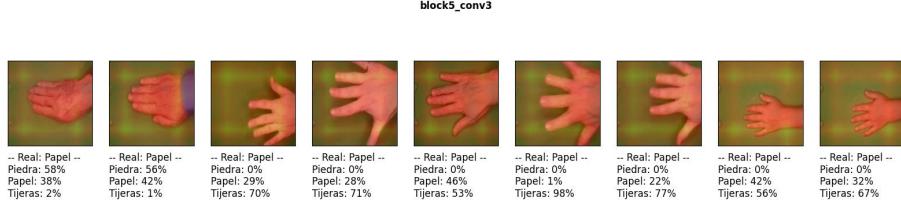


A simple vista, podemos identificar varios puntos importantes:

- Los casos en los que se confunde la imagen por “Piedra” tienen una probabilidad muy baja de ser “Tijera” y viceversa, lo que nos indica que el modelo es capaz de diferenciar entre ambos correctamente.
- Cuando la imagen no contiene por completo la mano, es decir, si por ejemplo un dedo queda fuera, la probabilidad de ser papel baja drásticamente, llegando hasta el 1% en uno de los casos.
- Tenemos dos formas de hacer papel y cada una se confunde con una categoría diferente: con la mano abierta se confunde con tijeras y con la mano cerrada se confunde con piedra.

Con este entendimiento del problema, vamos a tratar de identificar estos problemas en la representación de heatmaps sobre estas mismas imágenes, aunque cabe destacar que solo representaremos las convolucionales descongeladas de la VGG16 (las únicas que se han entrenado con fine tuning):





De las tres convolucionales descongeladas de la VGG16, la más interesante es la “block5_conv2”, ya que en ella vemos cómo el modelo se centra en la zona de los nudillos de la mano e incluso en el pulgar, por lo que puede estar tratando de conocer si se está usando la mano derecha o la izquierda. Además, parece que, de entre los casos con la mano abierta, se tiene una activación de la capa menos intensa cuando los dedos quedan fuera de la imagen, que puede deberse a que el pulgar es irreconocible.

Con estas capas, no podemos localizar todos los problemas que hemos propuesto anteriormente, pero es posible que haya capas más profundas que se encarguen de cosas como reconocer la forma de la mano, lo que podría llevar a confundir una mano cerrada con “Piedra”.

3.3. Casos correctamente clasificados

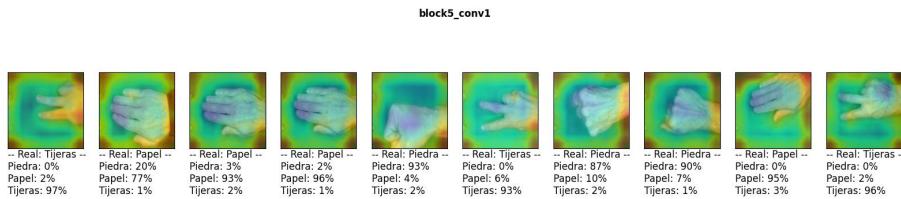
Repitamos ahora el análisis anterior con imágenes correctamente clasificadas. En este caso, nos fijaremos en la seguridad con la que se predice cada caso.

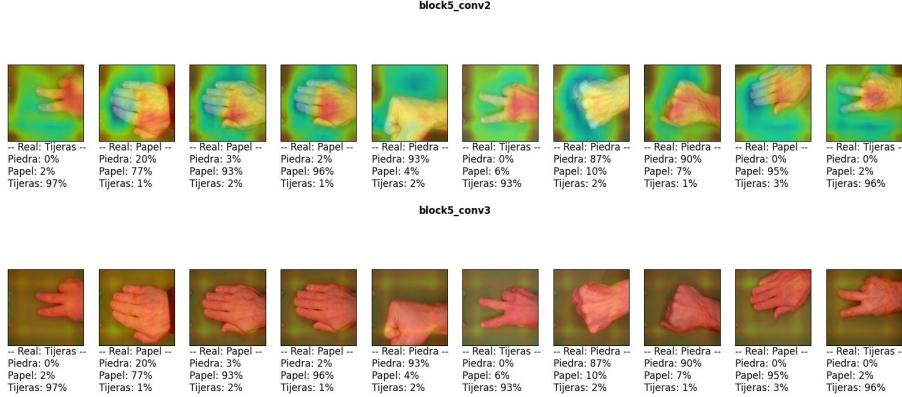
3.3.1. Modelo de Transfer Learning con la VGG16



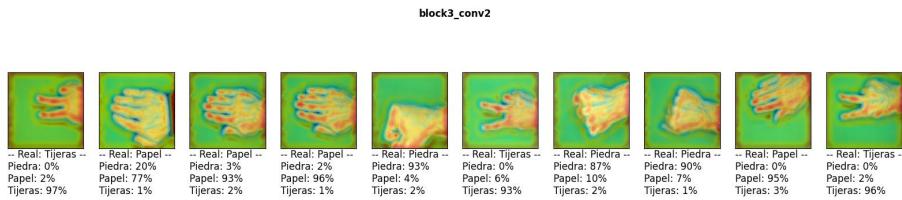
En general tenemos un acierto de la clase con una probabilidad superior al 90 % salvo en la segunda imagen. En esta, al tener la mano cerrada, se da una probabilidad relativamente alta de poder ser piedra.

Estudiemos esto también con los heatmaps:





No parece haber una diferencia clara. No obstante, revisando las otras capas convolucionales (las congeladas) encontramos este heatmap:



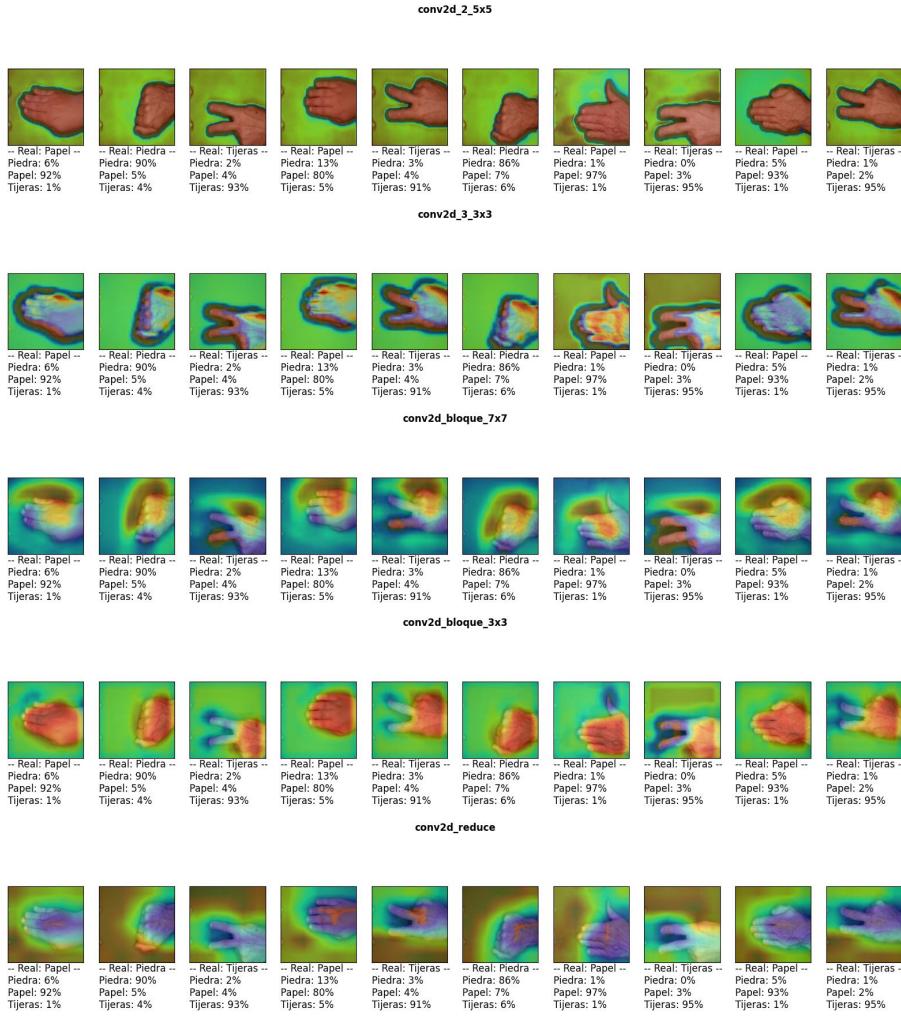
En este caso, podemos apreciar como en la segunda imagen no se marcan tan claramente como en los otros casos de “Papel” los dedos, lo que podría haber llevado a esta sobreestimación de la categoría “Piedra”.

3.3.2. Modelo propio con regularización Ridge



En esta muestra podemos ver que la clase correcta se asigna con una seguridad entorno al 90 % de nuevo. El caso con probabilidad más baja es la primera imagen, pero sigue siendo un valor bastante elevado. No obstante, cabe destacar que el motivo parece ser que los dedos se han separado ligeramente, dado al modelo a entender que también podría ser “Tijeras”.





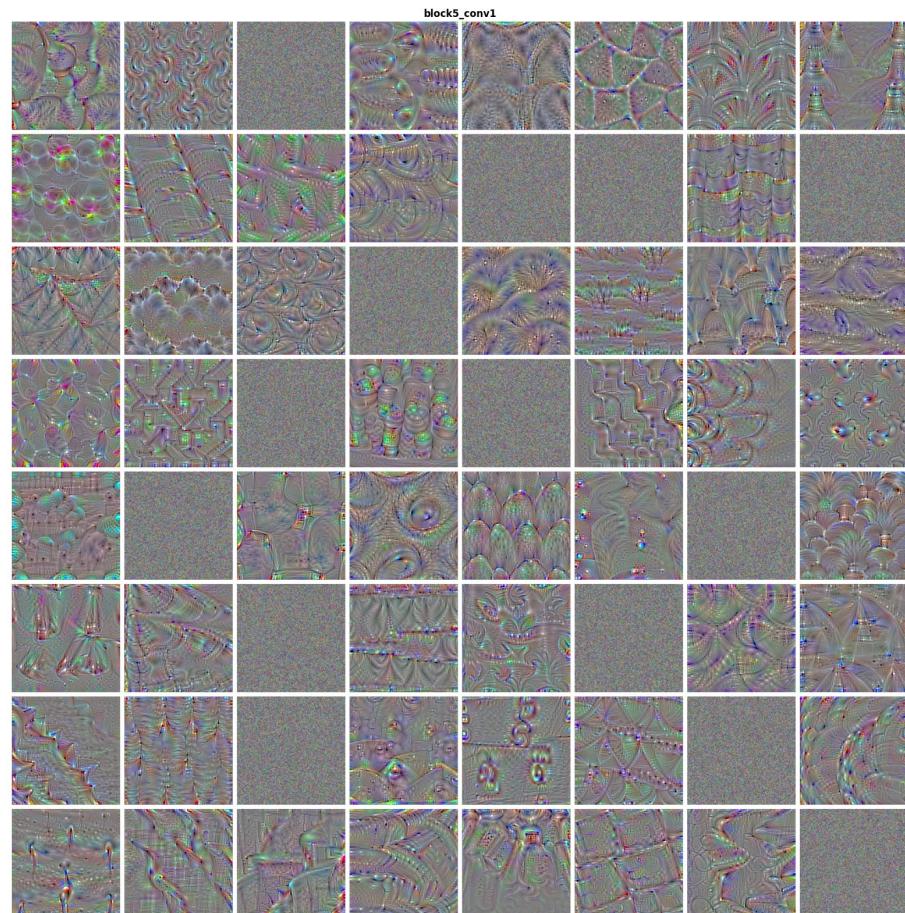
Las características más llamativas de estos resultados son:

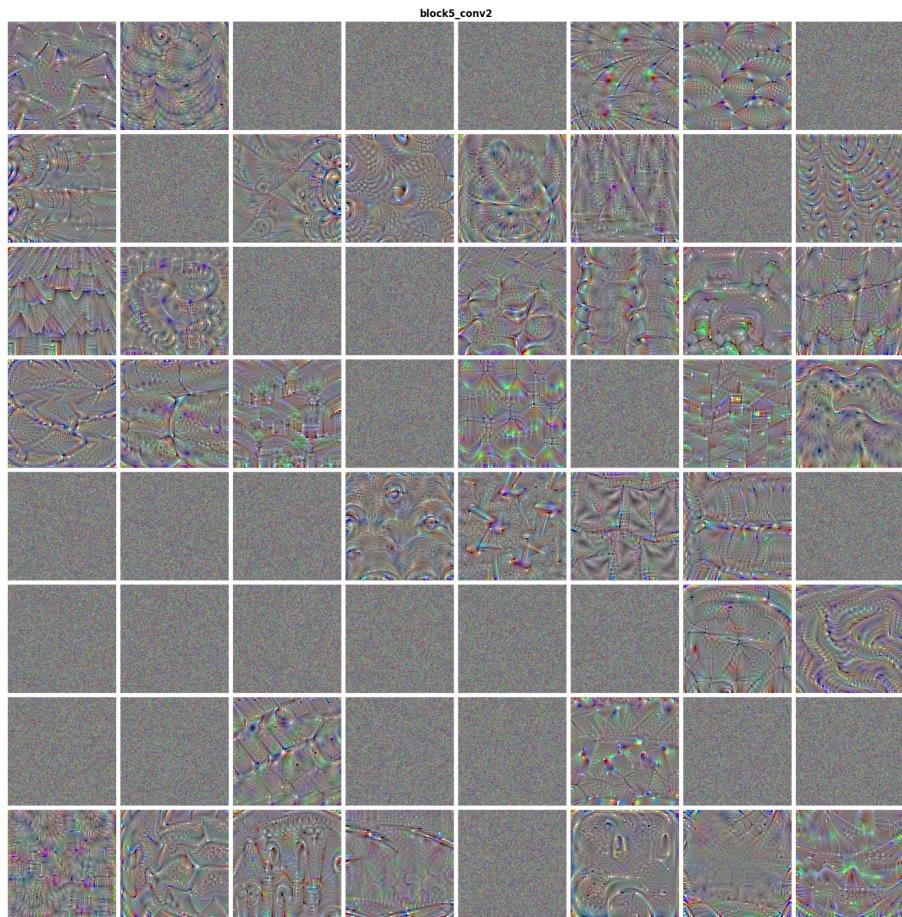
- En la capa conv2d_entrada tenemos ligeras activaciones en las imágenes de la categoría “Tijeras”. Concretamente, podemos verlo entre los dedos y cerca del brazo, en el borde de la imagen.
- Mientras que la capa conv2d_2_5x5 parece tratar de identificar la forma de la mano, la capa conv2d_3_3x3 se centra en su sombra.
- El bloque con la convolucionaria 7×7 y la 3×3 busca también la forma de la mano, pero ignora los dedos separados.

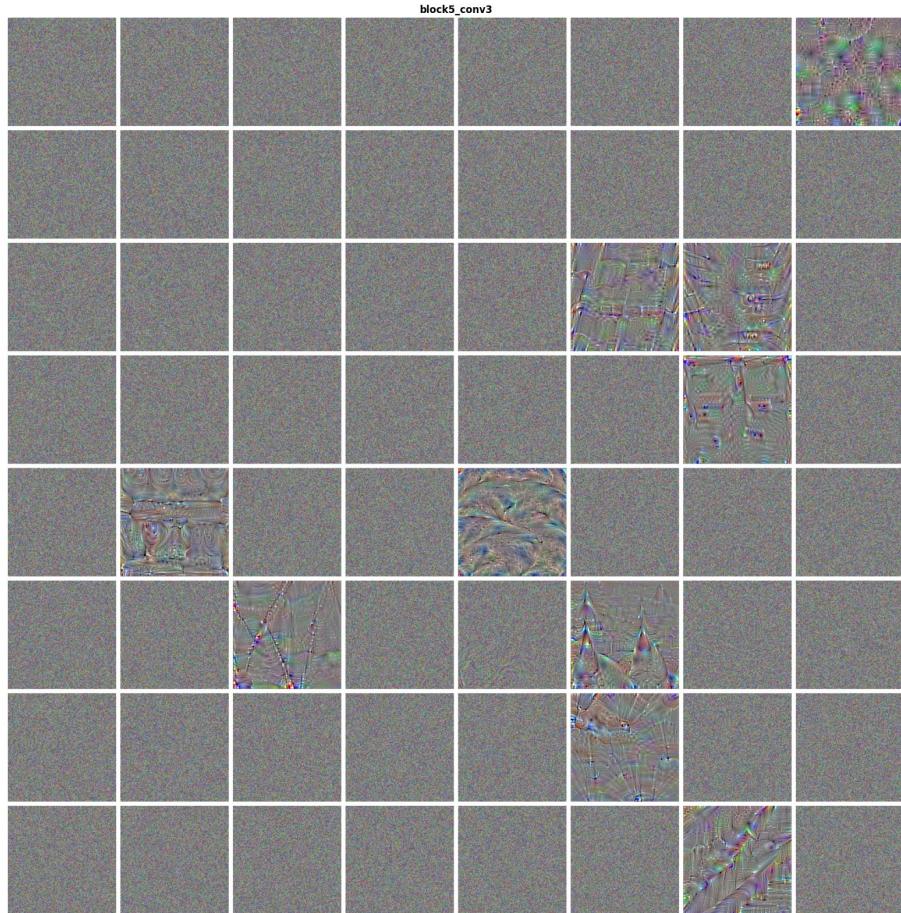
3.4. Visualización de las activaciones de los filtros convolucionales entrenados

Ahora, en lugar de ver qué partes de las imágenes activan las capas convolucionales, vamos a generar patrones introduciendo imágenes de ruido varias veces a los kernels (o filtros) individualmente. No obstante, para evitar unos tiempos de ejecución mucho mayores, hemos decidido representar un máximo de 64 kernels. De esta forma, podemos pasar por ellos la imagen de ruido más veces, dando unos patrones más visibles.

3.4.1. Modelo de Transfer Learning con la VGG16



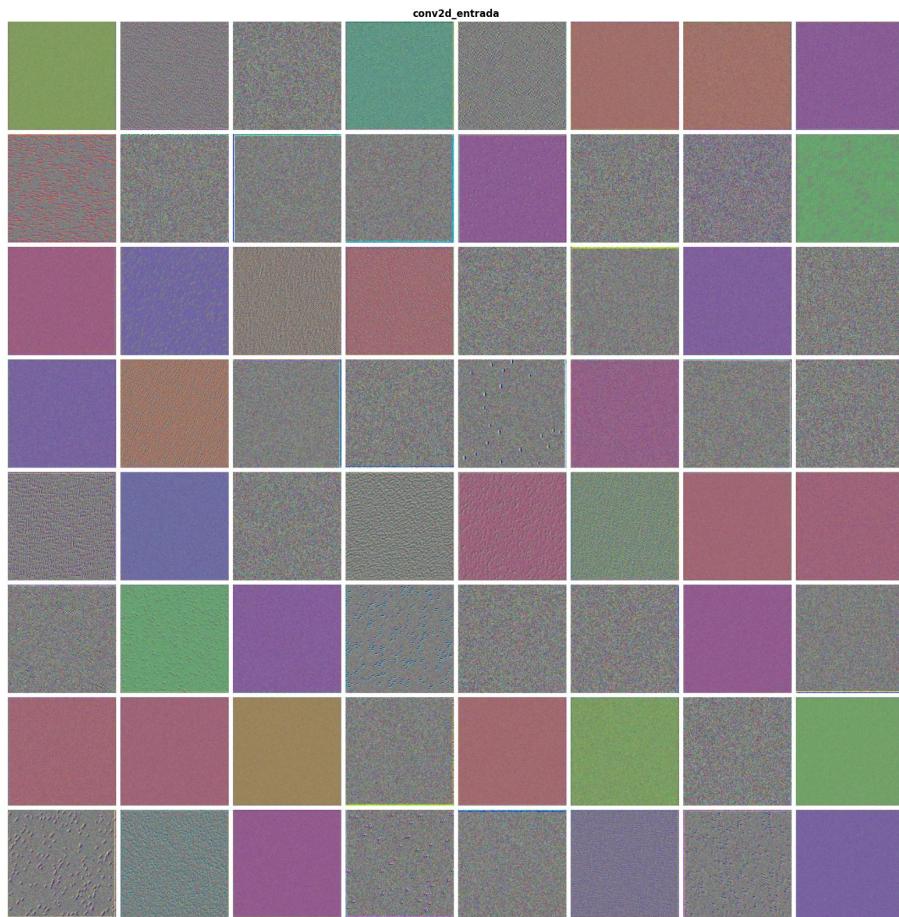




Como era de esperar, las convolucionales descongeladas de la VGG16 tienen patrones marcados que, además, parecen corresponderse a texturas de ropa. Esto se debe a que los pesos provienen del conjunto “imagenet” y, aunque les hayamos hecho fine tuning, queda demostrado que no es un efecto lo suficientemente grande como para modificar el comportamiento fundamental de la red.

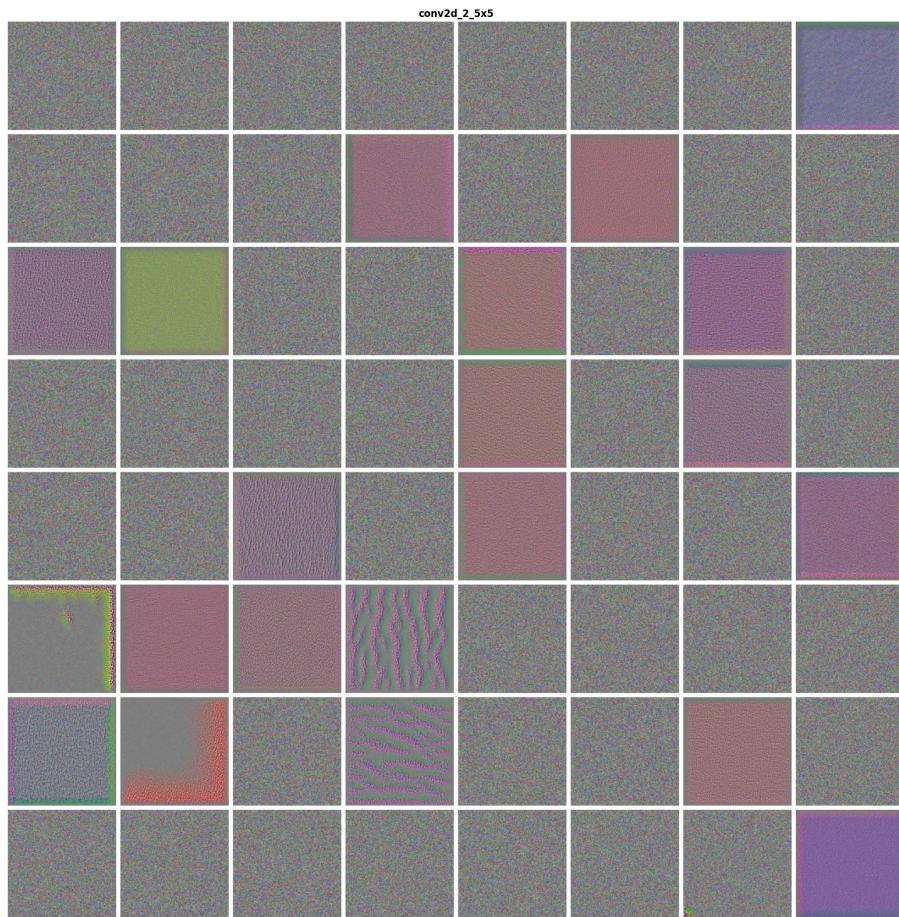
3.4.2. Modelo propio con regularización Ridge

Para este caso, vamos a ir analizando capa por capa los resultados. Esto lo haremos tomando también como referencia lo que podemos ver en los heatmaps anteriores.

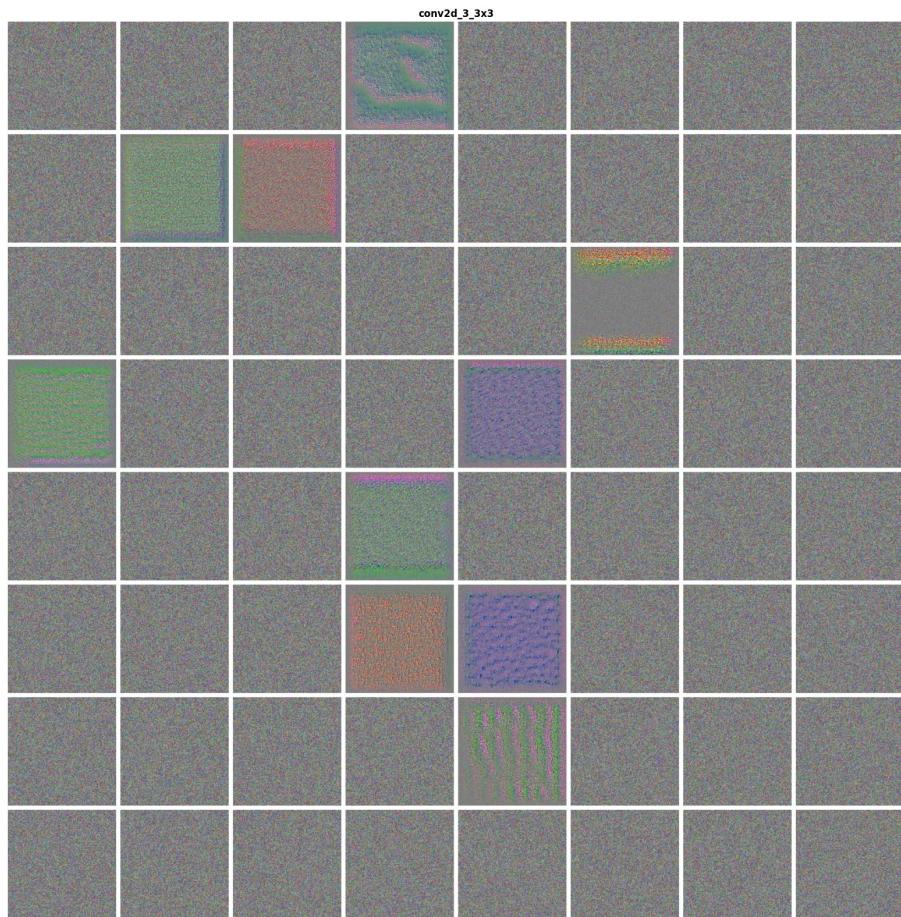


En esta capa principalmente se están reconociendo colores y texturas generales. Los kernels más llamativos son:

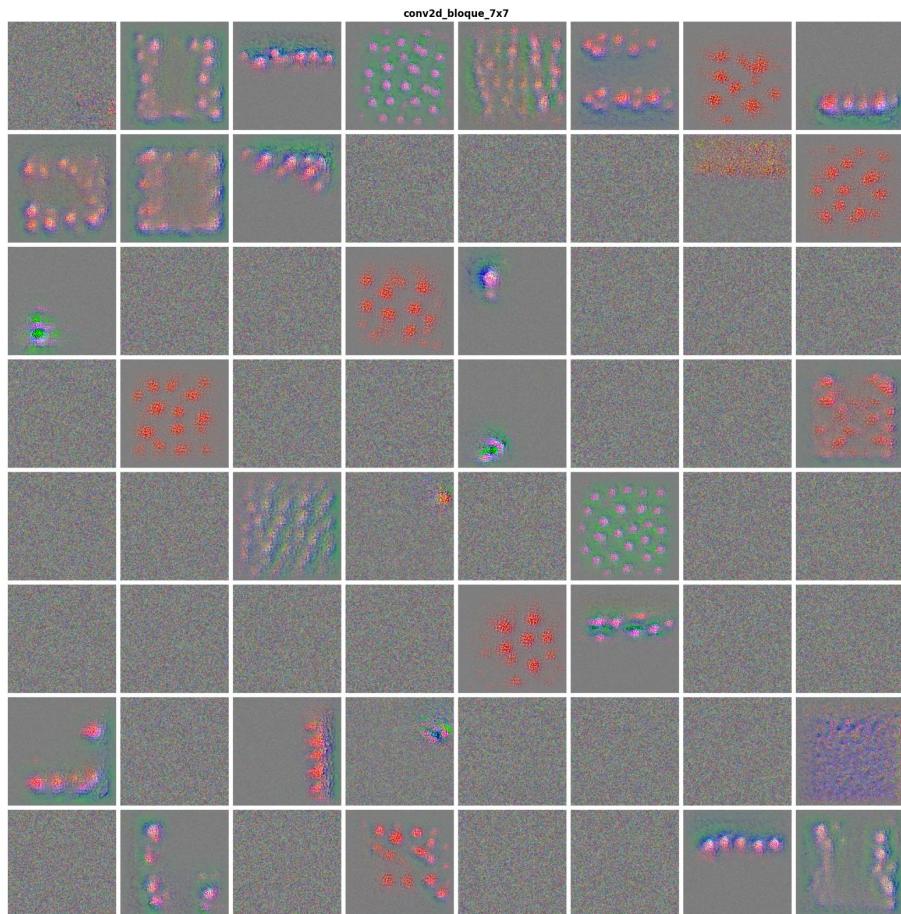
- Los que reconocen los bordes de la imágenes, que además los hay que buscan bordes verdes y bordes en tonos cálidos (igual que las manos).
- Patrones de manchas verdes, que podrían ser los que reconocen el hueco entre los dedos de las imágenes de la categoría “Tijeras”, como hemos visto antes en los heatmaps.



Por los heatmaps, sabemos que esta capa se centra principalmente en las manos, por lo que es razonable que veamos principalmente tonos cálidos en estos kernels. De hecho, los únicos patrones verdes que apreciamos son líneas en los bordes de la imagen o bien patrones de líneas que podrían usarse para localizar los dedos.

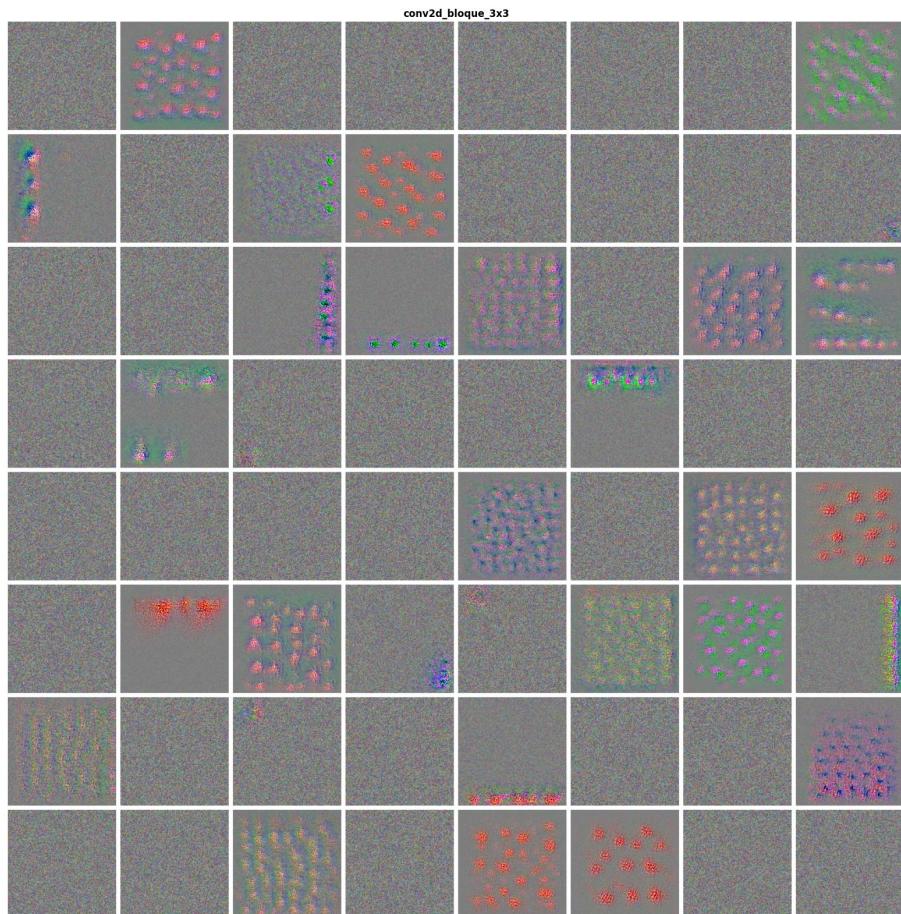


Esta capa se centraba en localizar las sombras de las manos y, como era de esperar, vemos más patrones en tonos verdes y azules.

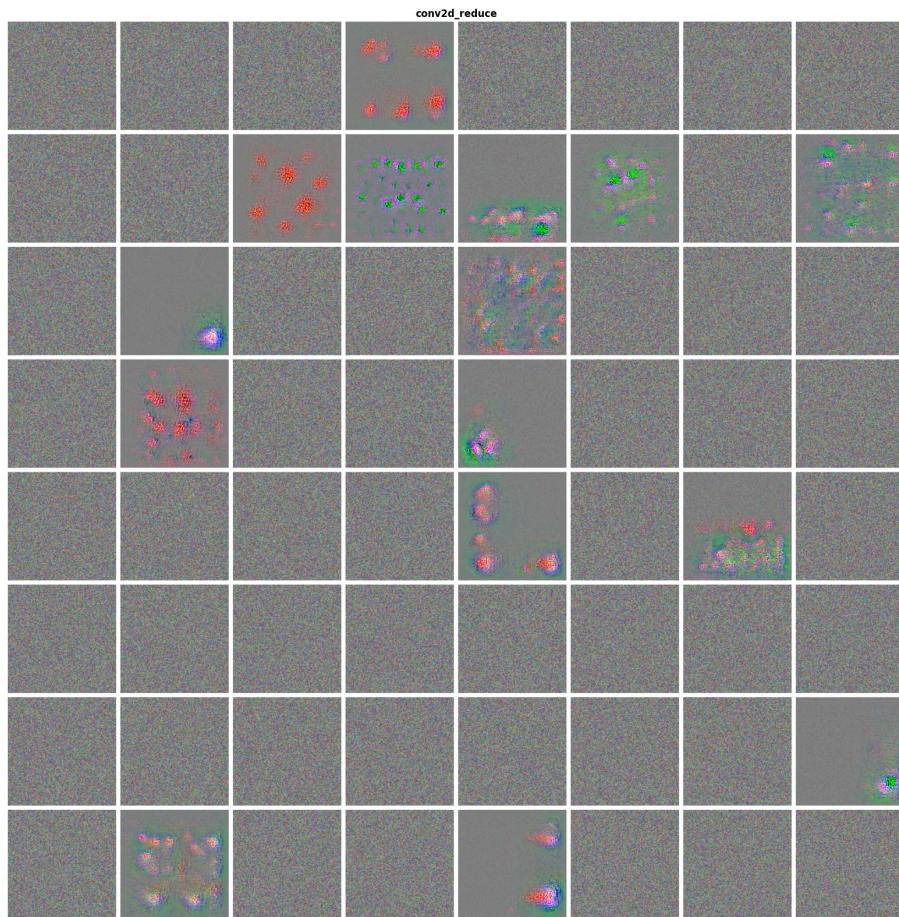


Los patrones generados por el bloque 7×7 y 3×3 resulta más interesante, ya que estos se centran en detalles más concretos de las imágenes.

Empezando por la 7×7 , podemos ver principalmente patrones de “bultos”. A priori, y tomando como referencia los heatmaps, parece razonable pensar que podrían ser los responsables de localizar características de la mano como nudillos o huesos que quedan especialmente marcados al hacer por ejemplo “Tijeras”.



En el caso de la convolucional 3×3 vemos algo muy similar con la diferencia de que, en general, los colores son más cálidos en estos filtros que en los de arriba y que se tienen más casos de patrones en bordes de la imagen o en esquina.



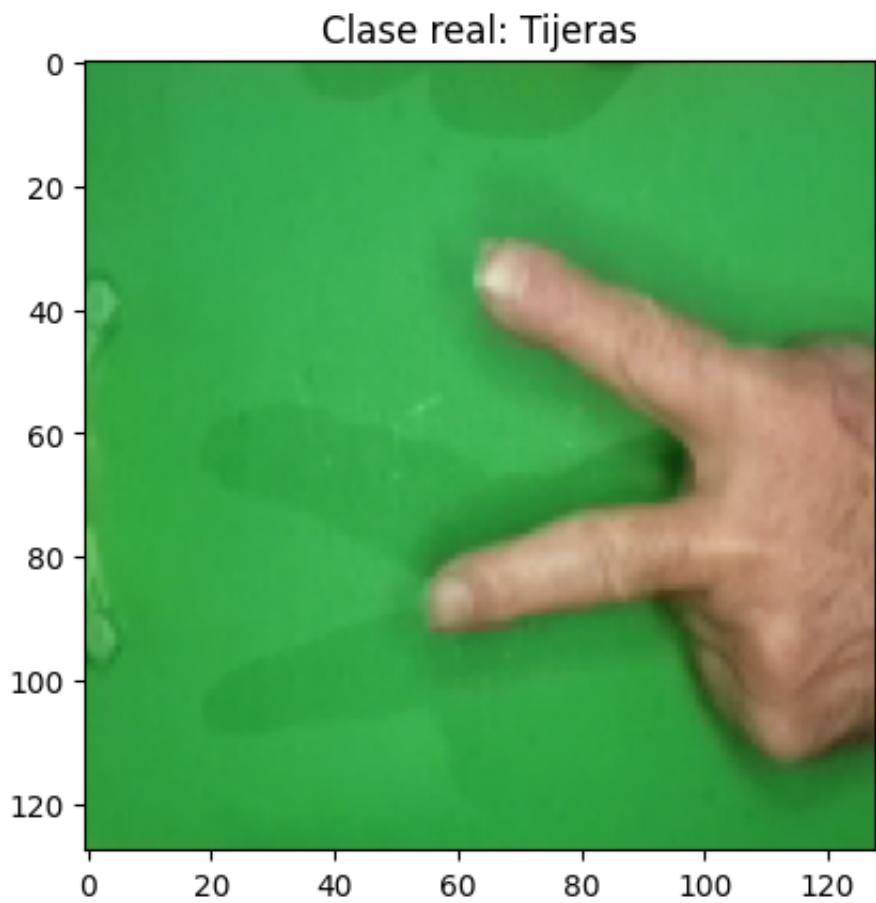
En la última capa, tenemos también un patrón de “bultos”, pero estos son más diversos y de un tamaño ligeramente mayor.

3.5. Kernels y mapas de características de la primera capa convolucional

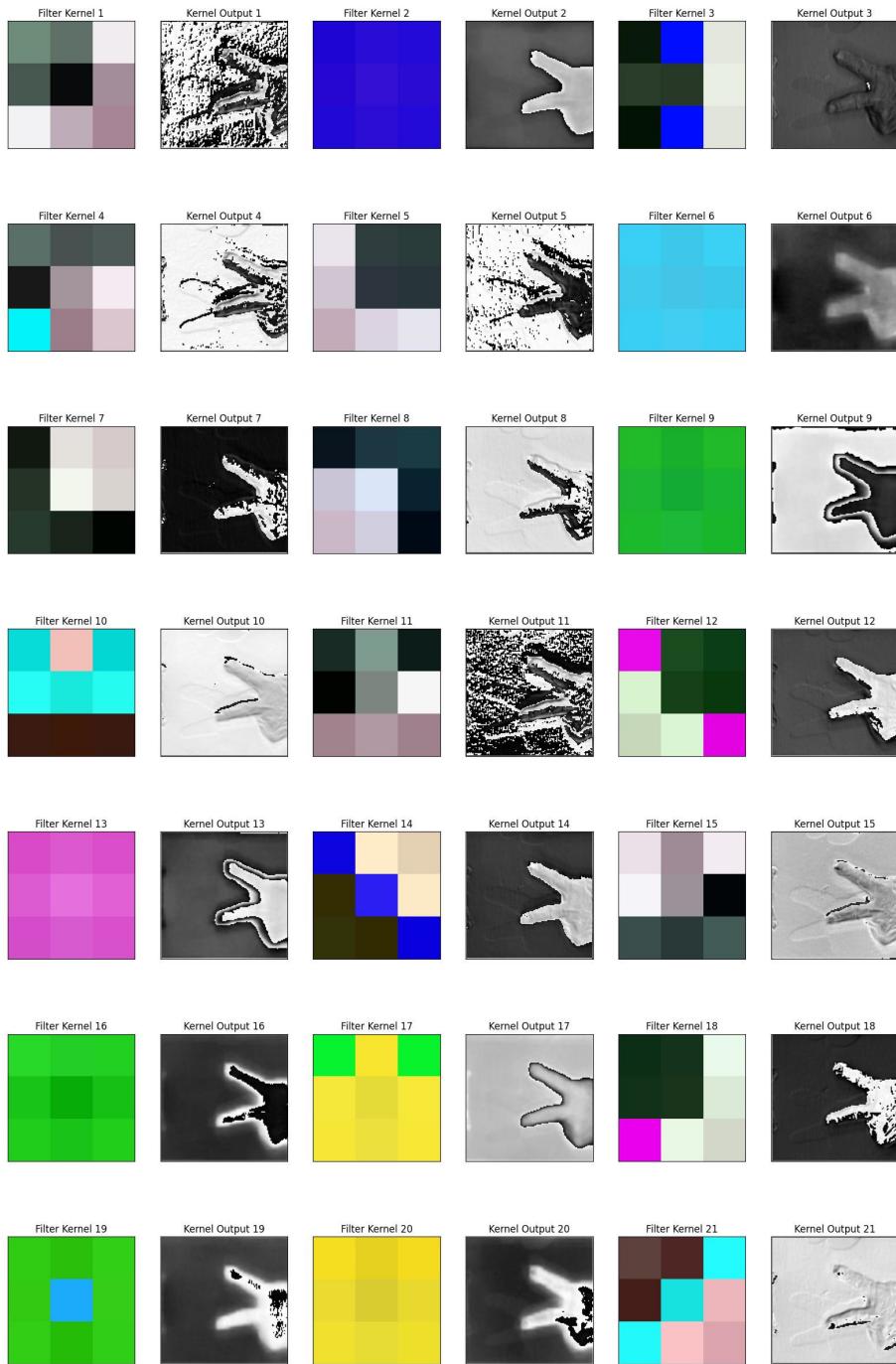
Finalmente, vamos a tomar una imagen correctamente clasificada y vamos a visualizar el resultado de aplicar sobre ella cada kernel de la primera capa convolucional.

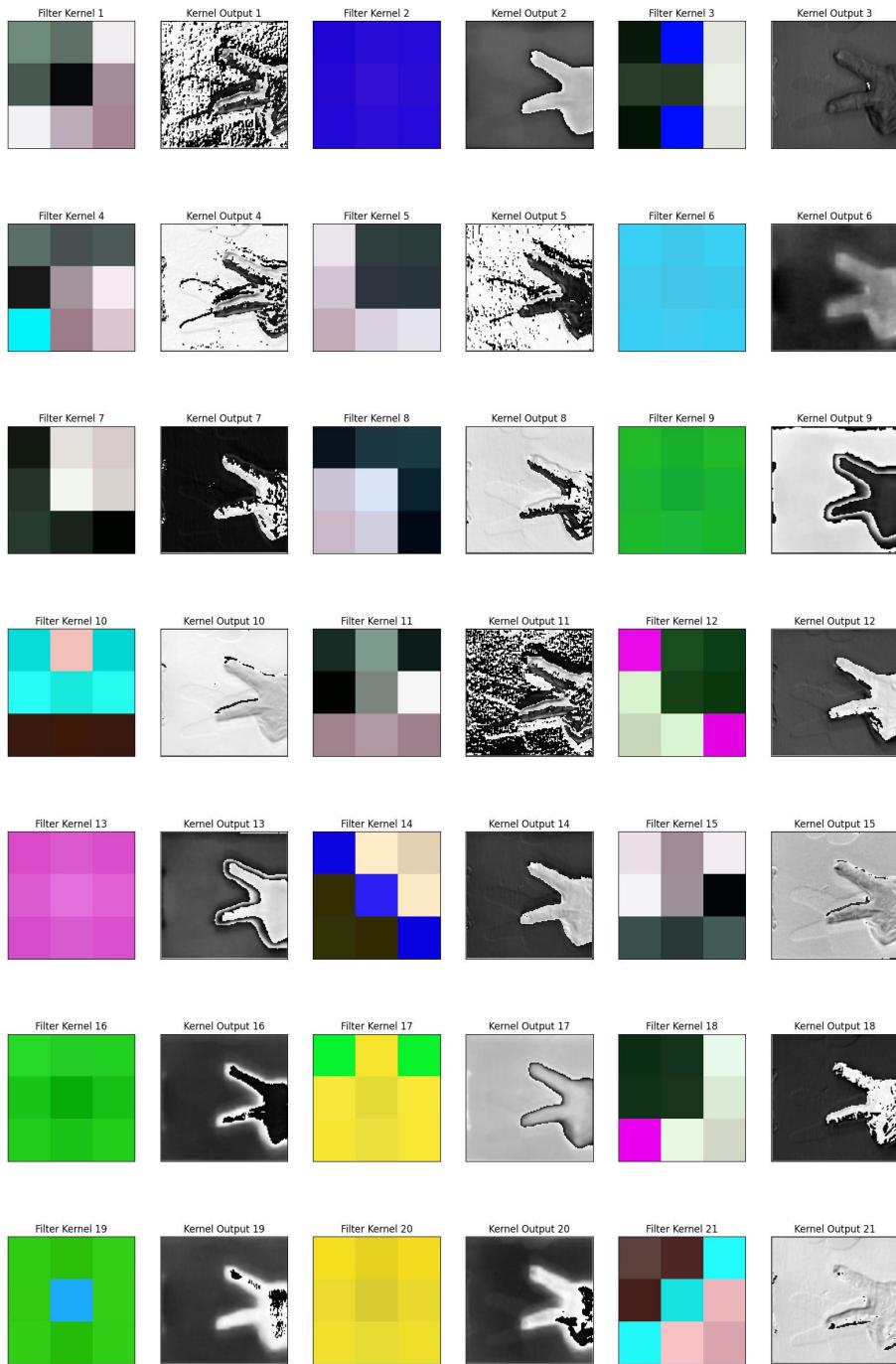
3.5.1. Modelo de Transfer Learning con la VGG16

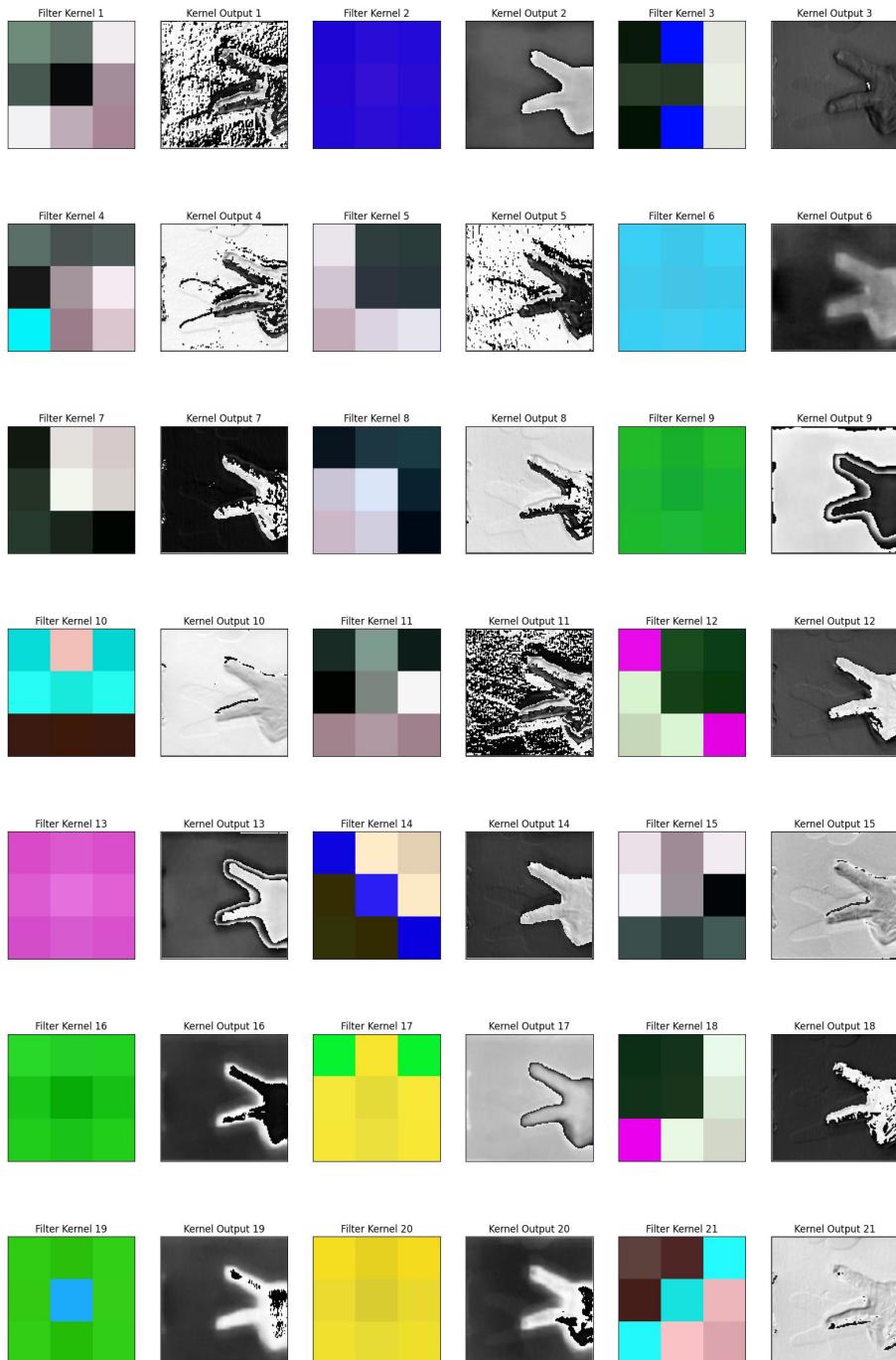
La imagen de referencia que se ha seleccionado aleatoriamente del conjunto de las VGG16 clasificadas correctamente es la siguiente:



Pasemos a aplicarle los filtros de la primera capa convolucional de la VGG16:







Algunas de las características más destacables son:

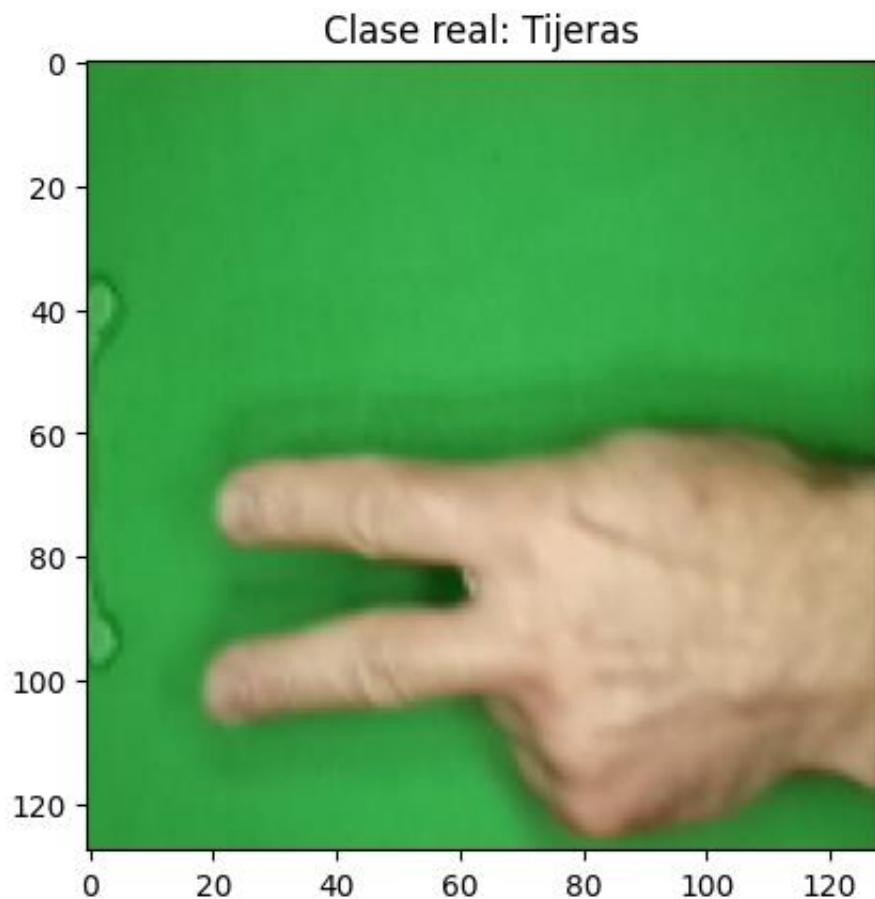
- Los kernels tienen colores asociados. Es decir, cada kernel tiene 3 versiones,

al igual que ocurre con las imágenes RGB.

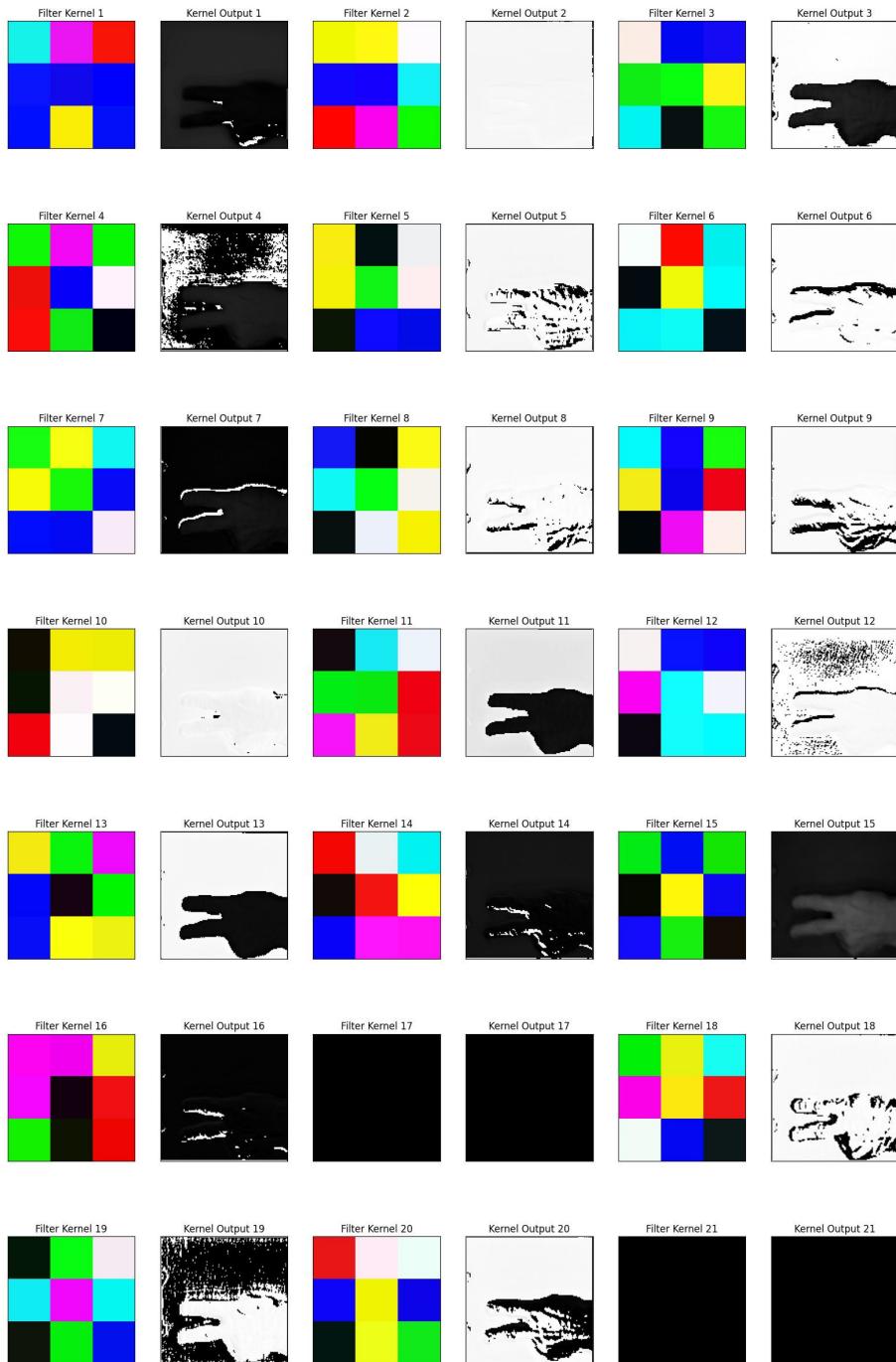
- En el kernel 9 podemos ver que el filtro es principalmente verde. Esto hace que el efecto del kernel recaiga principalmente en lo verde de la imagen, el fondo.
- Podemos ver otro ejemplo de kernel principalmente verde en el kernel 19 por ejemplo. En este caso se ha perdido la mayoría del fondo, dejando la mano, pero esta queda difuminada, imagino que por no haber filtros que apliquen sobre ella.
- El kernel 42 actúa sobre la mayoría de aspectos de la imagen, lo que coincide con los colores del kernel.

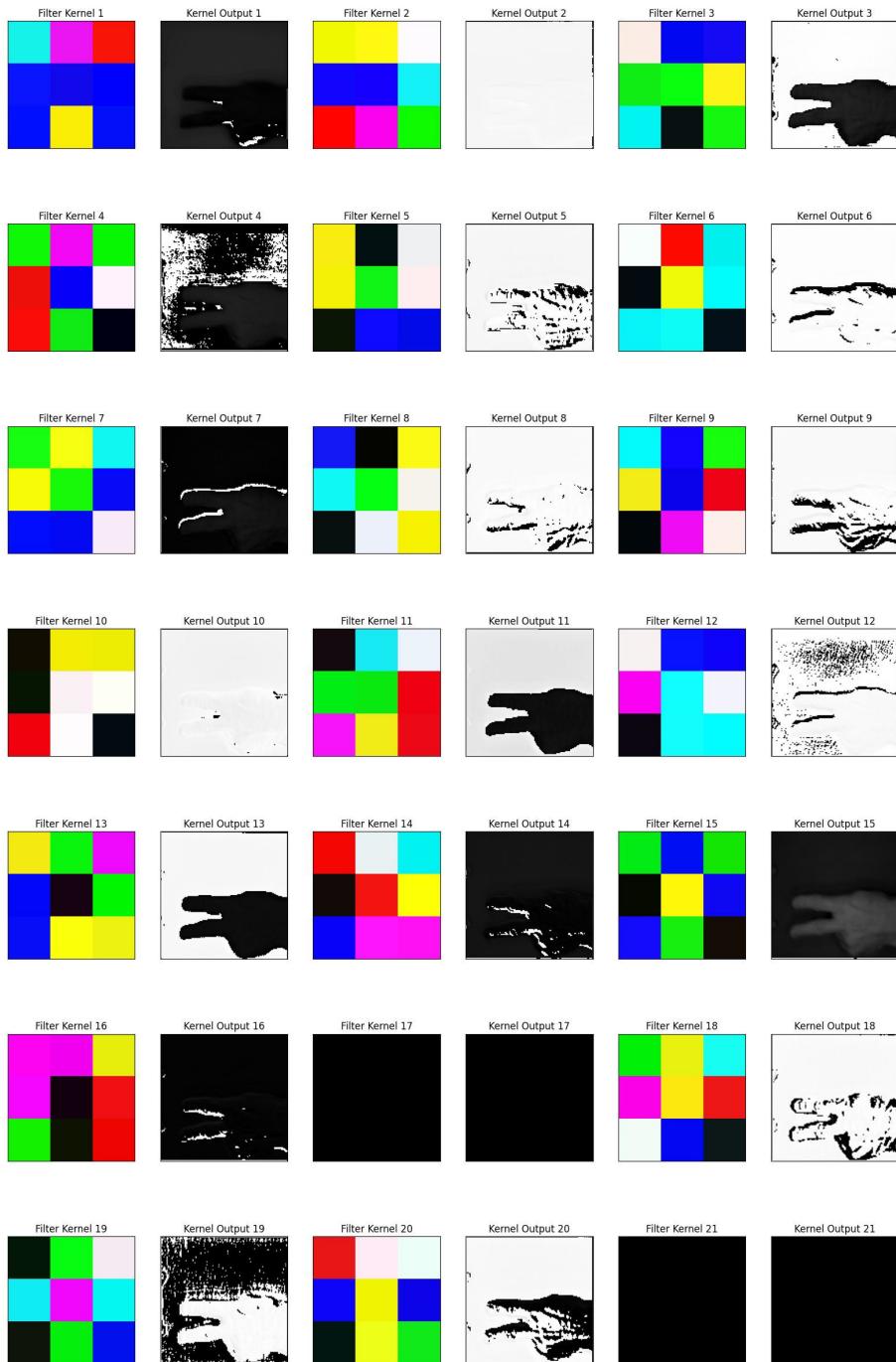
3.5.2. Modelo propio con regularización Ridge

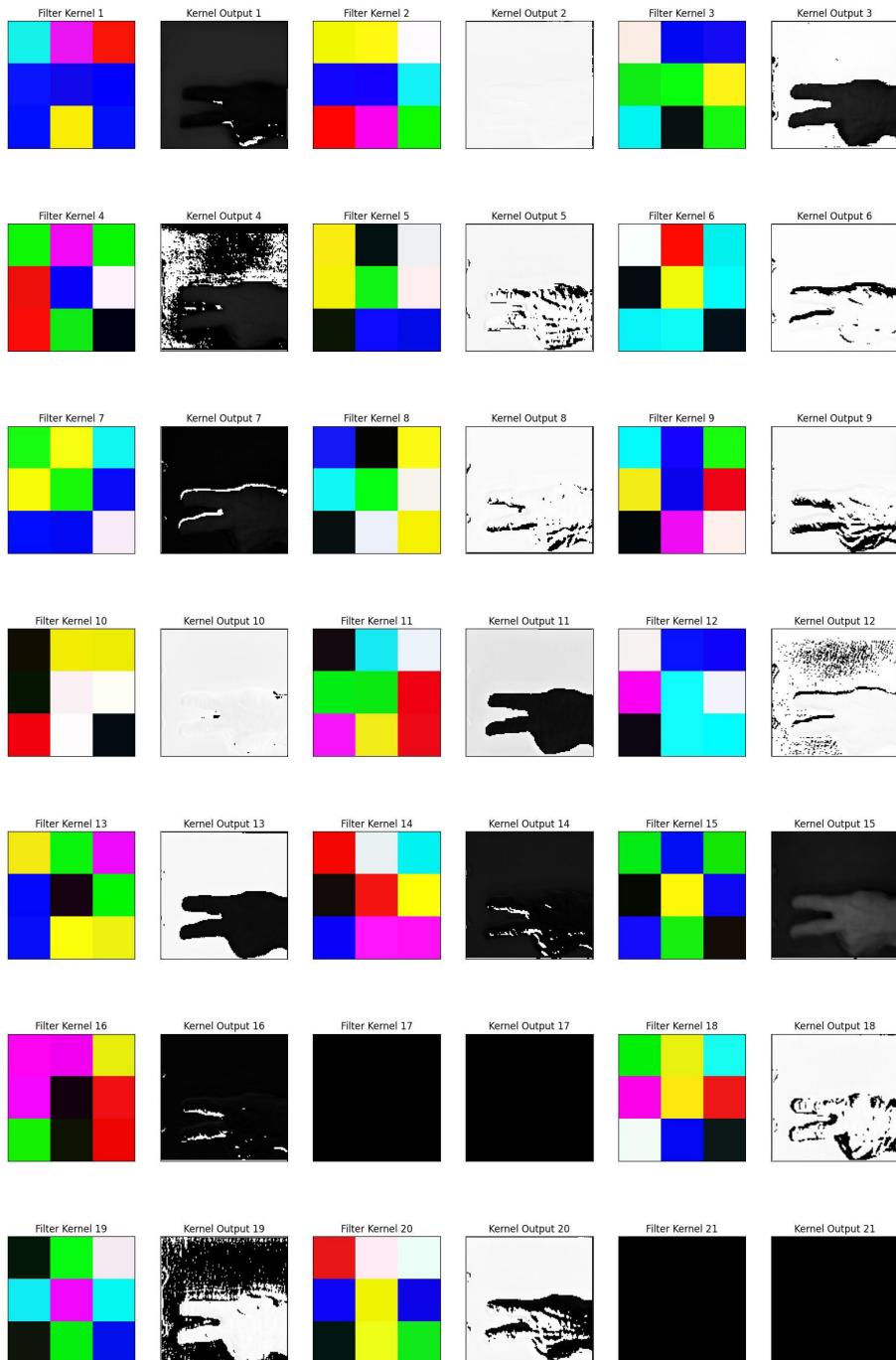
La imagen que se ha seleccionado aleatoriamente de entre las que se clasifican correctamente por este modelo es la siguiente:



Veamos el efecto de los kernels de la primera capa convolucional de este modelo sobre la imagen:







Vemos algo muy similar a lo del otro modelo. Por destacar alguno de los kernels, podemos ver en el séptimo que al tomar colores azules y verdes se actúa sobre

la sombra de la mano.

4. Conclusiones

En este trabajo hemos visto aproximaciones diferentes al problema de clasificar imágenes del juego “Piedra, papel y tijeras” y diversas técnicas en la construcción del modelo. En base a los resultados obtenidos podemos decir que:

- El uso de “Transfer learning”, frente a entrenar una red neuronal desde cero, nos ayuda a alcanzar valores de precisión altos en menos tiempo.
- No es necesario utilizar una red profunda como la ResNet para solucionar este problema. Esto es especialmente importante en el entrenamiento de una red desde cero.
- El uso de regularización en las capas adecuadas puede solucionar problemas de desvanecimiento del gradiente. Sin embargo, utilizar regularización en todas las capas o incluso en algunos puntos de la red en los que no es necesaria, puede llevar a una gran disminución del rendimiento, cosa que hemos experimentado al tratar de construir un modelo desde cero.
- Otro factor que se ha observado mientras hacíamos el trabajo es que los modelos con menores valores de la función loss alcanzaba el modelo en un conjunto de datos daban activaciones más notables sobre las imágenes y patrones generados más visibles en la imagen de ruido.

En conclusión, si tuviésemos que elegir qué técnica a usar para construir un modelo para este problema y lo fundamental fuera la precisión, no tendríamos un camino claro que elegir. No obstante, si planteamos requisitos adicionales, como una limitación de espacio o un límite en el tiempo de entrenamiento, elegiría construir un modelo de cero y usar Transfer Learning respectivamente.

5. Referencias

En general, las referencias se indican en la parte concreta del código donde se ha utilizado, pero también las mostraré a continuación:

Prácticas de la asignatura: se han tomado de referencia en varias ocasiones, por lo que indico en cada caso la práctica concreta. Las más utilizadas fueron la 5 (regularización de redes) y la 6 (visualización de información).

Construcción de la clase Sequence:

1. Ejemplo funcional de un objeto Sequence
2. Usar array de numpy como índices
3. Aumento de datos

Carga de datos:

4. Inestabilidad de la función loss

Construcción del modelo:

5. Pautas para construir una red neuronal
6. Atributo loss en la compilación del modelo
7. Modelos pre-entrenados
8. Visualización de redes
9. Elección de learning rate
10. Bloques residuales
11. Redes Inception
12. Técnicas de regularización

Medidas de rendimiento del modelo:

13. Matriz de confusión

Visualización de resultados

14. Mostrar color correcto usando imshow de matplotlib
15. Mantener rango de valores de la imagen entre 0 y 255 para la correcta representación de imagen RGB

Otras referencias:

- A. Crear índice en Jupyter notebook
- B. Instalación de tensorflow en Windows
- C. Fijar semilla aleatoria en tensorflow

Referencias del archivo .py con funciones externas:

- Prácticas 5 y 6 de la asignatura de Deep Learning del Máster en Data Science y Big Data.
- Mostrar color correcto usando imshow de matplotlib

Referencias del archivo .py con los modelos:

- Prácticas 5 y 6 de la asignatura de Deep Learning del Máster en Data Science y Big Data.
- Indicaciones generales sobre la regularización
- Usar regularización para evitar sobreajuste
- Redes implementadas en tensorflow