Artificial Neurotransmitter

María del Carmen Llano Carmona
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
marllacar1@alum.us.es - maricarmen_-10@hotmail.com

El objetivo principal en el que nos embarcamos en este trabajo es el estudio de las llamadas redes neuronales convolucionales, concretando aún más, describiremos cómo funcionan estos algoritmos y qué tipo de arquitectura pueden dotar. Explicaremos varios modelos diseñados y entrenados desde cero con distintitas técnicas de regularización con el fin de seleccionar aquel modelo que mejor generalice nuestro conjunto de imágenes, para ello usaremos como *backend* TensorFlow, la API de Keras, y como cuadernos de ejecución Google Colab y Kaggle.

Palabras Clave—Inteligencia artificial, aprendizaje profundo, redes neuronales, capas convolucionales, sobreajuste.

I. INTRODUCCIÓN

¿Qué es la Inteligencia Artificial?, con esta pregunta presentamos este trabajo, dado que se describe a partir del concepto de la propia inteligencia humana que, hoy en día tiene múltiples interpretaciones. Una de sus definiciones podría ser: subdisciplina de la Informática que busca imitar comportamientos inteligentes a través de máquinas. Estos comportamientos podrían ser: jugar a juegos, conducir coches o reconocer imágenes entre muchos otros. Dependiendo del tipo de comportamiento que presenten se pueden clasificar en: Robótica, Procesamiento del Lenguaje Natural, Voz o Visión, sin embargo, hay algo que nos dota de agentes inteligentes y es la capacidad de aprender y aquí es donde introducimos al *Machine Learning*. Esta última podría ser el núcleo de la Inteligencia Artificial dado que mediante este último campo podemos conseguir aprender cualquiera de las habilidades que se consigue programando en los distintos campos anteriores. Por ejemplo, podemos programar el movimiento de un robot, pero ¿qué tal suena que el propio robot aprenda a interaccionar con el entorno? Para ello existen distintas técnicas de aprendizaje como son: Árboles de Decisión, Modelos de Regresión y Clasificación, y 'Clustering', pero sin duda la que ha alzado a la fama al *Machine Learning* han sido las Redes Neuronales [1].

Las Redes Neuronales, como modelo computacional existen desde mediados del siglo pasado, pero no ha sido hasta hace unos años, con la mejora de la tecnología, cuando se logran grandes hallazgos como la conducción autónoma (Waymo) o verificación facial (DeepFace) pertenecientes a las compañías de Google y Facebook respectivamente.

Estos modelos inspirados en el cerebro humano reciben una serie de valores de entrada y cada una de estas entradas llega a un nodo llamado neurona. Las neuronas de la red están a su vez agrupadas en capas que forman la red neuronal. Cada una de las neuronas de la red posee a su vez un peso, un valor numérico, con el que modifica la entrada recibida. Los nuevos valores obtenidos salen de las neuronas y continúan su camino por la red. Una vez que se ha alcanzado el final de la red se obtiene una salida que será la predicción calculada por la red Este funcionamiento puede observarse de forma esquemática en la siguiente imagen [3]:

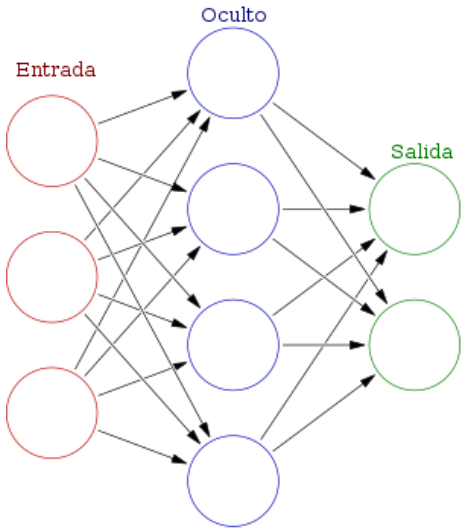


Fig. 1. Ejemplo de la estrucutra de una red neuronal

En este trabajo realizaremos un enfoque en un tipo de red neuronal denominadas: Redes Neuronales Convolucionales.

Las Redes Neuronales Convolucionales (CNN) son un algoritmo de *Deep Learning* usado para trabajar con imágenes, donde las primeras capas son capaces de detectar los elementos más simples como son líneas, curvas… y poco a poco se van especializando hasta reconocer formas complejas como rostros, objectos, siluetas, etc. Su uso mas frecuente es la clasificación de imágenes.

Para la extracción de características se usa la composición de capas convolucionales, capas de agrupación y capas totalmente conectadas.

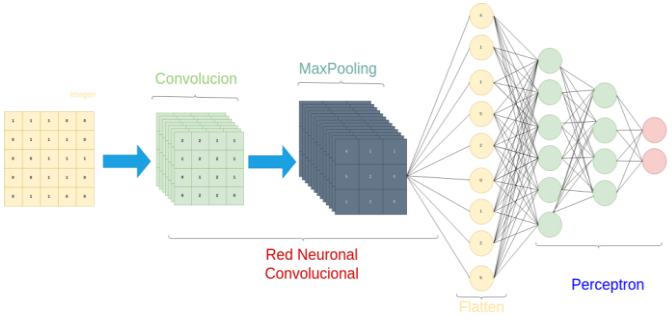


Fig. 2. Arquitectura de una CNN

Las capas de convolución realizan una operación lineal en la que se multiplican los datos de la matriz de entrada (por ejemplo, los pixeles de una imagen) y una matriz bidimensional de ponderaciones denominada filtro. El uso de un filtro con dimensiones más pequeñas que la matriz de entrada, permite el desplazamiento de dicho filtro de izquierda a derecha y de arriba a abajo otorgando el poder de detectar un tipo específico de característica en cualquier parte de la matriz de entrada. Con esta operación obtenemos como resultado una matriz bidimensional denominada *feature map*. Es muy importante resaltar que, durante el entrenamiento del modelo, nuestra red neuronal aprenderá cuales son los valores mas adecuados para los filtros, ajustando así los pesos. [4]

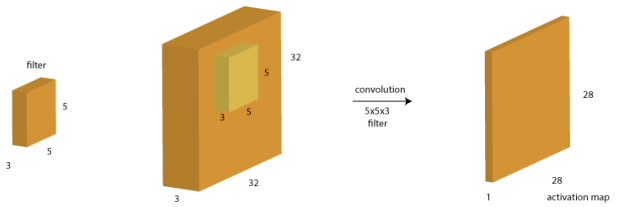


Fig. 3. Ejemplo de convolución: Matriz de entrada con dimensiones 32x32x3 y filtro de 5x5x3. Ambas con profundidad 3 (número de canales de color)

Las capas de agrupación o *pooling layers* son usadas para reducir las dimensiones espaciales (ancho y alto) para la entrada de la siguiente capa convolucional. Una de las ventajas de esta operación es la reducción de sobrecarga de cálculo, ya que si tuviéramos una CNN con muchas capas el coste computacional para procesar todos los parámetros sería muy elevado. Aunque reducir el tamaño nos conlleve a pérdida de información, ayudará para reducir el sobreajuste. [5]

Las dos capas de agrupación más importantes son: *Max Pooling* y *Average Pooling*, en las que en cada subregión nos quedaremos con el valor máximo o medio respectivamente.

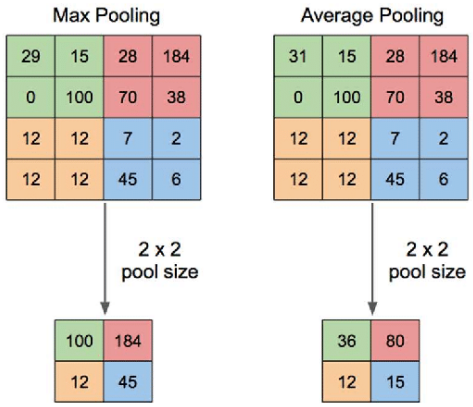


Fig. 4. Ejemplo de *Max Pooling* y *Average Pooling* con agrupaciones de 2x2

Las capas totalmente conectadas o *fully-connected layers*, servirán para aplanar la matriz en una sola dimensión (*flatten*), conectando con la capa de salida que tendrá la cantidad de neuronas correspondientes con las clases que estamos clasificando (función *softmax)*, prediciendo así la etiqueta correcta.

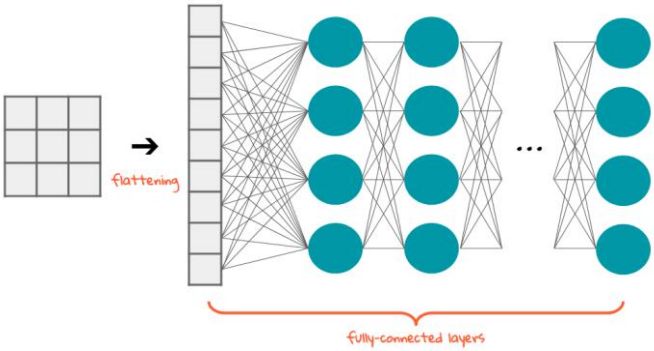


Fig. 5. Ejemplo de operación *flatten* y capas totalmente conectadas

El enfoque de este trabajo con respecto a las Redes Neuronales Convolucionales es comparar algunas de las técnicas de regularización más usadas para mejorar la generalización y reducir el sobreajuste, y las distintas arquitecturas que estos algoritmos pueden dotar. Finalmente hablaremos de algunos de los modelos más importantes de hoy en día usado para la clasificación de imágenes.

Trabajaremos con un conjunto de imágenes sobre distintas escenas naturales clasificadas como: *buildings*, *glacier*, *forest*, *mountain*, *sea* y *street*. Dicho conjunto esta dividido en tres subconjuntos: entrenamiento (*train*), validación (*val*) y prueba (*test*), con un total de 11224, 2810 y 3000 respectivamente.

También es importante fijarnos en las cantidades de imágenes que existen de cada clase por eso es bueno analizar que categorías están más desbalanceadas ya que podría repercutir en como nuestro modelo aprende. Podríamos decir que nuestro conjunto de imágenes está balanceado tras analizar los porcentajes de cada clase.

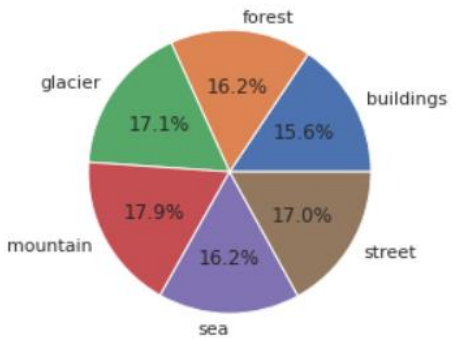


Fig. 6. Diagrama porcentual de imágenes por clases

Aunque nuestro conjunto está ya preparado para procesarlo, usaremos la técnica de *Data Augmentation* para mejorar el sobreajuste y aprendizaje de nuestra red neuronal. Aunque esta técnica suele ser usada para aumentar el *dataset* en caso de que tengamos un conjunto de imágenes muy pequeño, la usaremos para analizar el aprendizaje de nuestro modelo y así evitar que nuestra red neural aprenda patrones irrelevantes. ¿Y, en qué consiste dicha regularización?

Data Augmentation se define como la generación artificial de datos por medio de perturbaciones en los datos originales. [6]

La biblioteca de redes neuronales de aprendizaje profundo de Keras proporciona la clase *ImageDataGenerator*, capaz de aplicar transformaciones espaciales a las imágenes. Como tal, está claro que la elección de las técnicas de aumento de datos específicas utilizadas para un conjunto de datos de entrenamiento debe elegirse con cuidado y dentro del contexto del conjunto de datos de entrenamiento y el conocimiento del dominio del problema. [7]

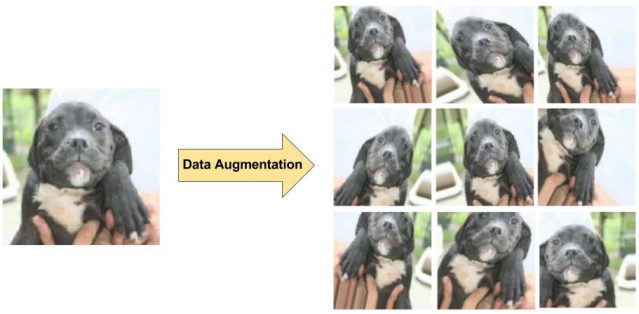


Fig. 7. Ejemplo de *Data Augmentation*

Para nuestro caso de estudio hemos elegido las siguientes modificaciones a aplicar:

- *Shear range*: aplica transformaciones de corte al azar.
- *Zoom_range*: aplica zoom aleatorio dentro de las imágenes.
- *Horizontal_flip*: voltea aleatoriamente las imágenes horizontalmente (en nuestro caso de estudio no tiene sentido voltear verticalmente las imágenes).

```
train_datagen = ImageDataGenerator(
    rescale= 1./ 255, # Reescalamos los pixeles con valores entre 0 y 1
    shear_range= 0.2,
    zoom_range= 0.2,
    horizontal_flip= True
)
```

Fig. 8. Ejemplo de la clase *ImageDataGenerator*

II. PRELIMINARES

Sabemos que construir modelos de redes neuronales supone una gran complejidad para los nuevos desarrolladores del aprendizaje automático, por ello existen varias propuestas de APIs de alto nivel mejoradas y simplificadas para construir modelos de redes neuronales. Por ello dedicaremos una breve presentación a tres herramientas fundamentales para la iniciación del *Machine Learning*: TensorFlow, Keras y Python. [8]

1. **TensorFlow**: Es una biblioteca desarrollada por Google Brain para sus aplicaciones de aprendizaje automático y las redes neuronales profundas.

2. **Keras**: Es una de las API de redes neuronales de alto nivel. Está escrita en **Python** y es compatible con varios motores de cálculo de redes neuronales de *back-end*. Aunque Keras admite múltiples motores de *back-end*, su motor principal (y predeterminado) es TensorFlow.
   El modelo es la estructura de datos central de Keras, hay dos tipos principales de modelos disponibles en Keras: el modelo *Sequential* y la clase *Model* utilizada con la API funcional. Para nuestro trabajo nos centraremos en el uso de la clase *Sequential*, creando de manera simple una pila lineal de capas

III. METODOLOGÍA

Esta sección está dedicada a la descripción de los distintos modelos empleados para la clasificación de las escenas naturales pertenecientes a nuestro conjunto de datos, explicaremos sus arquitecturas adentrándonos en las distintas capas que sostienen la arquitectura, entrenamiento, evaluación y predicción. Explicaremos cuatro modelos distintos, de los cuales dos están formados por una misma arquitectura, donde usaremos algunas de las técnicas de regularización y

contrastaremos sus resultados. El tercer modelo estará compuesto por una arquitectura distinta a los dos anteriores con el fin de comparar los distintos resultados obtenidos tras el proceso de entrenamiento. Finalmente, en el cuarto modelo usaremos la metodología de *Transfer Learning* empleando como punto de partida el modelo ya entrenado VGG16, modificando el bloque de salida para adaptarlo a nuestro problema de clasificación.

Modelo-I (Base)

Este primer modelo está basado en una arquitectura simple, sin ninguna implementación de regularización. Nos referiremos a él como Modelo-I (Base).

A. **Datos:** Para importar los datos, en nuestro caso, imágenes, al cuaderno de ejecución, usaremos la clase *ImageDataGenerator* que nos proporciona Keras, donde aprovecharemos para reescalar los pixeles entre los valores 0 y 1. Dicha clase nos ofrece un método denominado *flow-from-directory* al que le pasamos los siguientes parámetros: la ruta de nuestro directorio, el tamaño de nuestras imágenes (150 x 150), los canales de color (3 canales, RGB), el tamaño de los lotes o '*batch size'* (32) y el tipo de clase ('*categorical'*). Este último valor aplica a nuestro '*dataset'* el llamado 'One-hot encoding', que asigna la etiqueta correspondiente a cada imagen en lugar de un valor numérico. Este proceso se llevará a cabo tanto en las imágenes de entrenamiento como en las de validación y prueba.

B. **Arquitectura:** El tipo de modelo que usaremos es 'Sequential', mencionado precedentemente, ya que nos permite construir el modelo capa por capa. Para agregar las capas empleamos la función '*add'*, pasando como parámetro el tipo de capa que queremos agregar y sus distintas características.

   Comenzamos agregando cuatro bloques formados por pares de capas de tipo 'Conv2D' y 'MaxPooling2D'. Para la agrupación usamos siempre ventanas de 2 x 2, mientras que las capas de convolución tienen 32, 64, 64 y 128 filtros respectivamente, con tamaño de 3 x 3, todas ellas con la función de activación 'ReLU', que devuelve el valor proporcionado como entrada directamente, o el valor 0 si la entrada es 0 o menos.

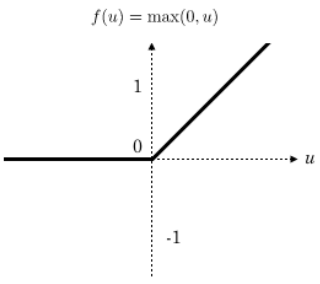


Fig. 9. Función ReLU

En el siguiente bloque agregamos una capa de tipo 'Flatten', en la que como ya se comentó previamente aplanamos la matriz con un tamaño de array de una sola dimensión.

Ultimamos nuestro modelo con tres capas de tipo 'Dense', donde las dos primeras tienen 256 y 128 neuronas con función de activación 'ReLU', y la última con 6 neuronas y función de activación 'Softmax' que, asigna probabilidades decimales a cada clase.

Finalmente, compilamos el modelo donde usaremos como optimizador 'Adam' que, como su propio nombre indica optimiza los valores de los parámetros para reducir el error cometido por la red. ¿Y por qué 'Adam'?, Adam o 'Adaptative Moment Optimization' combina la metodología de 'Momentum' y 'RMSProp', calculando una combinación lineal entre el gradiente y el incremento anterior, y considera los gradientes recientemente aparecidos en las actualizaciones para mantener diferentes tasas de aprendizaje por variable [9].

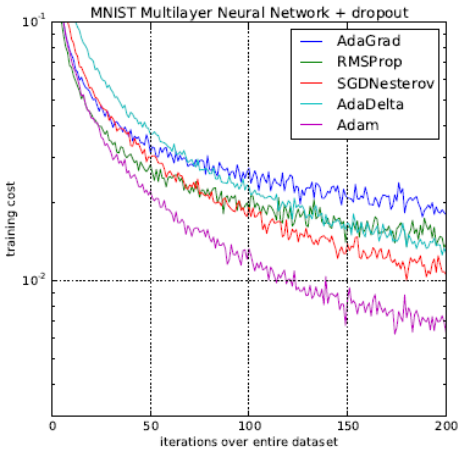


Fig. 9. Comparación de diferentes optimizadores mediante el entrenamiento de redes neuronales multicapa en imágenes MNIST [10]

El siguiente parámetro es la función de coste, que determinará el error entre el valor estimado y el valor real, con el fin de optimizar los parámetros de la red neuronal. Elegimos para este parámetro el valor '*categorical_crossentropy'* que aumenta a medida que la probabilidad predicha diverge de la etiqueta real. Por lo tanto, nuestro objetivo es minimizar el valor de la función de coste para así, conseguir mayor predicción en nuestro clasificador de imágenes.

Como último parámetro y no menos importante nos encontramos con la elección de las métricas para nuestro modelo, que en nuestro caso elegimos '*accuracy'*. Esta métrica mide la precisión de nuestro modelo de la siguiente forma:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Esta métrica será usada posteriormente para la matriz de confusión.

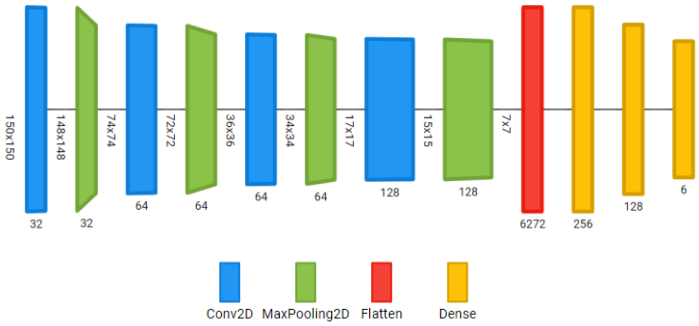


Fig. 9. Diagrama de la arquitectura del Modelo-I (Base) usando la herramienta Net2Vis [11]

C. **Entrenamiento:** Keras nos ofrece una función llamada *fit*, que nos permite entrenar nuestro modelo con los siguientes parámetros: primero agregamos el generador de imágenes de entrenamiento, seguido del parámetro '*steps_per_epoch'*, definido como el número de pasos por lotes ('*batch size'*) en cada época, agregamos también nuestro generador del conjunto de validación con su respectivo parámetro 'validation steps' equivalente a '*steps_per_epoch'* con el conjunto de validación, y finalmente el parámetro '*epochs'*, que indica el número de pasadas de todo el conjunto de datos de entrenamiento que ha completado el algoritmo, asignamos el valor 20.

Detallamos el proceso de entramiento mediante las siguientes dos gráficas. En la primera, vemos la evolución del '*accuracy'* según los datos de entrenamiento y validación, mientras que en la segunda se muestra la función de coste o '*loss'*.

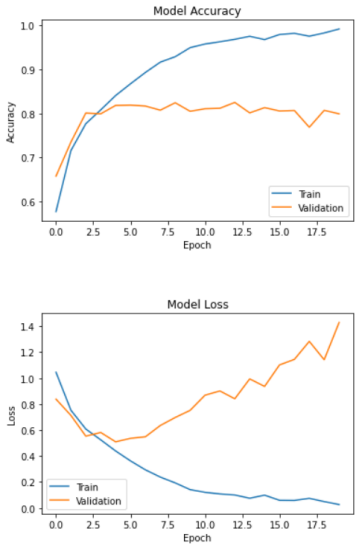


Fig. 10. Comparativa del Modelo-I (Base) desde el punto de vista de la precisión y función de pérdida durante las 20 épocas

Si nos fijamos en el comportamiento de las gráficas, vemos como se da el famoso concepto de '*overfitting'*. Por una parte, la precisión de los datos de entrenamiento crece durante las 20 épocas hasta llegar 99%, mientras que la precisión en los datos de validación crece y se estanca a partir de la tercera época, tomando valores entre 80% y 82%.

La función de pérdida de los datos de validación alcanza su mínimo después de pocos '*epochs'* y luego empieza a subir, mientras que la de los datos de entrenamiento disminuye linealmente hasta llegar a casi 0.

D. **Evaluación:** Es hora de evaluar nuestro modelo con la función '*evaluate'* que nos proporciona Keras. Para la evaluación usaremos los datos de prueba, imágenes nuevas para el modelo. Los resultados que obtenemos son: 78.7% de *accuracy* y 1.4 de *loss*. Como ya podriamos predecir, la clasificación de nuestro modelo es mejor con datos vistos anteriormente ya que en el entrenamiento se ha producido un sobreajuste en los valores de los pesos.

E. **Predicción:** Para medir las predicciones de algunas imágenes, haremos uso de la función '*predict'* y pasaremos como parámetro una imagen preprocesada. La llamada a la función nos devolverá un vector de probabilidades, con valores entre 0 y 1, siendo el argumento mayor la etiqueta predicha. Se usará un método auxiliar para imprimir por pantalla a que etiqueta pertenece el índice con mayor probabilidad llamado '*getcode'* al que le pasaremos como parámetro el índice indicado. El siguiente ejemplo muestra la predicción que nos retorna tras probar con una imagen que pertenece a la clase '*buildings'*:

```
Predicción de la imagen: buildings
```

Fig. 11. Resultado de la predicción

En este caso, el índice del valor máximo en el vector de probabilidades se corresponde con el índice de la etiqueta '*buildings'* . Nuestro modelo ha acertado. Sin embargo no deberiamos de olvidar que la tasa de acierto de nuestro modelo cuando evaluamos imágenes no vistas anteriormente fue de 78% aproximadamente y esto nos hizo pensar que sería buena idea trazar una matriz de confusión ya que nos proporcionaría bastante información sobre que clases del modelo falla, ya que ni yo misma sabria diferenciar algunas de las imágenes pertenecientes a las clases '*buildings'* y '*street'*.

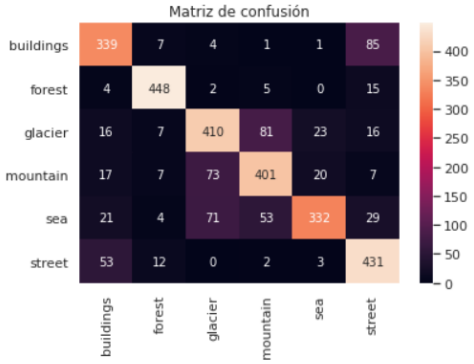


Fig. 12. Matriz de confusión del Modelo-I (Base)

En la izquierda se muestran las etiquetas verdaderas o '*true labels'* y bajo la matriz, las etiquetas que ha predecido nuestro modelo o '*predict labels'*. Ponemos un ejemplo de como deberiamos de leer la matriz: nos fijamos en la clase '*buildings'* del eje de las '*true labels'*, donde la predicción de 339 imágenes ha sido correcta, sin embargo 85 de las imágenes catalogadas como 'buildings' han sido clasificadas como '*street'*. Según la matriz, las clases con mas tasas de error son '*mountain'* – '*glacier'* y '*buildings'* – '*street'*. A continuación veremos como podemos mejorar estos valores con algunas técnicas de regularización.

Modelo-I (DataAug + Dropout)

En este segundo modelo emplearemos la misma arquitectura que en el Modelo-I (Base), solo que usaremos dos técnicas de regularización: *Dropout* y *Data Augmentation.*

A. **Datos:** Para la importación de los datos usaremos de nuevo la clase *ImageDataGenerator*, que al igual que el primer modelo, reescalaremos los pixeles de las imágenes que en principio toman valores entre 0 y 255 a valores entre 0 y 1. Además, aplicaremos algunas transformaciones y veremos cómo afectan al nivel de generalización de nuestro modelo. Esta técnica se denomina *Data Augmentation* y se explica con más detalle en el punto de Introducción.

B. **Arquitectura:** Como ya hemos introducido, la arquitectura de este segundo modelo está basado en la misma arquitectura que el Modelo-I (Base), con una variación, se agregan dos capas de tipo *Dropout* en el último bloque. Esta capa consiste en desactivar aleatoriamente un porcentaje de las neuronas acorde a una probabilidad de descarte que definiremos en los parámetros de Keras. Con esto se consigue que ninguna neurona memorice patrones. En nuestro caso, la tasa de abandono se corresponde con un 30% y 50% respectivamente.

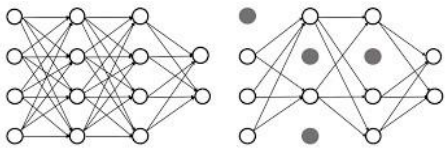


Fig. 13. Ejemplo de capa *Dropout*

Como se muestra en la figura 13, en la primera imagen podemos ver como todas las neuronas están interconectadas, mientras que, en la segunda imagen las neuronas de color gris no participan en la fase de entrenamiento.

Table 1. Porcentaje de abandono por capa

Capa	Tasa de abandono
1	¼ = 0.25
2	½ = 0.50
3	¼ = 0.25

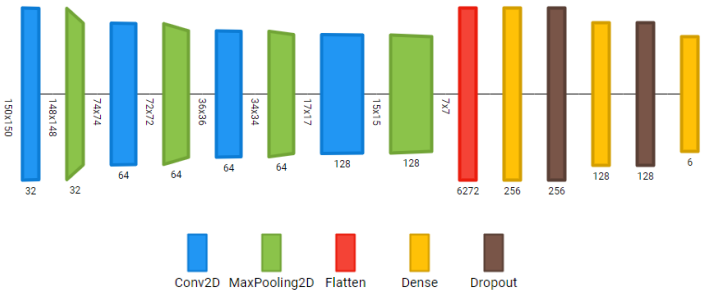
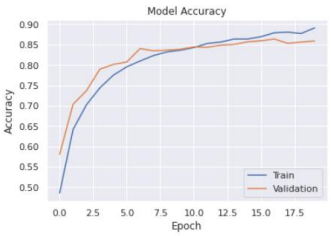


Fig. 14. Diagrama de la arquitectura del Modelo-I (DataAug + Dropout) usando la herramienta Net2Vis [11]

C. **Entrenamiento:** Para entrenar este nuevo modelo usaremos los mismos hiperparámetros que usamos en el Modelo-I (Base) con el fin de estudiar como afecta las nuevas técnicas de regularización.

Detallamos el entrenamiento de este nuevo modelo apoyándonos en las siguientes gráficas:



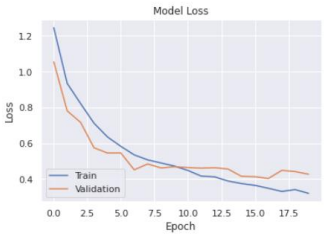


Fig. 14. Comparativa del Modelo-I (DataAug + Dropout) desde el punto de vista de la precisión y función de pérdida durante las 20 épocas

Notoriamente, se puede comprobar que nuestro modelo generaliza mejor e incluso existe cierta linealidad en los valores de '*accuracy'* y '*loss'* de las imágenes de entrenamiento y validación. El entrenamiento esta sujeto al recorrido de 20 épocas llegando a alcanzar un '*accuracy'* de 86% en el conjunto de validación. Con estos nuevos valores, se podría afirmar que hemos reducido el problema del sobreajuste.

D. **Evaluación:** La evaluación del Modelo-I (DataAug + Dropout) se realizará de forma análoga a la evaluación del Modelo-I (Base). Los valores obtenidos del tras evaluar el modelo con las imágenes de prueba son: 85% de '*accuracy'* y 0.4 de '*loss'*, valores mejorados tras aplicar *Data Augmentation* y *Dropout*.

E. **Predicción:** Calculamos la predicción de una imagen que pertenece al directorio de glaciares y obtenemos como resultado la siguiente etiqueta:

```
Predicción de la imagen: glacier
```

En este caso nuestro modelo ha acertado la clase a la que pertenece la imagen correctamente pero al igual que en el modelo anterior, analizaremos la matriz de confusión.

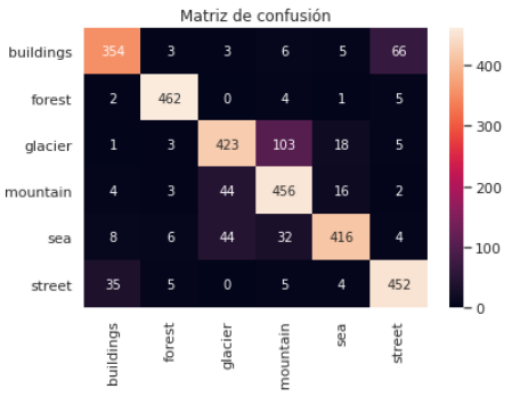


Fig. 15. Matriz de confusión del Modelo-I (DataAug + Dropout)

Si comparamos esta matriz con la anterior vemos como hemos mejorado la clasificación para las clases '*buildings'* – '*street'* en cambio, hay cierta incertidumbre entre las clases '*glacier'* – '*mountain'*.

Modelo-II

En este apartado describimos las características del Modelo-II, centrado en una arquitectura distinta a los modelos anteriores, en la que veremos mayor número de capas convolucionales y la implementación de la técnica de regularización *Early Stopping.*

A. **Datos:** Aplicamos el mismo procedimiento que en el Modelo-I (DataAug + Dropout) para la inserción de las imágenes en el proceso de aprendizaje con la técnica de Data Augmentation.

B. **Arquitectura:** Dividimos la estructura en 7 bloques, el primer bloque se compone de 2 capas de tipo *Conv2D* con los siguientes parámetros: 32 filtros con ventanas de tamaño 3 x 3 y función de activación ReLU para ambas convoluciones seguido de una capa *MaxPooling* con un tamaño de ventana de 2 x 2. El segundo bloque es similar al primero, pero en este caso el número de filtros queda multiplicado por 2 con un total de 64. Con esta misma relación formamos los tres bloques siguientes, compuesto de pares de capas de *Conv2 – MaxPooling*, multiplicando en cada bloque el número de filtros por 2. Los dos últimos bloques conforman el aplanamiento de la matriz en una sola dimensión, es decir una capa de tipo Flatten y nuestra salida de la red compuesta de 3 capas de tipo *Dense* interferidas por capas *Dropout.*

Referente a la compilación del modelo, el único cambio a destacar es la bajada de la tasa de aprendizaje del optimizador '*Adam'* a 0.0001, siendo el valor por defecto de 0.001.

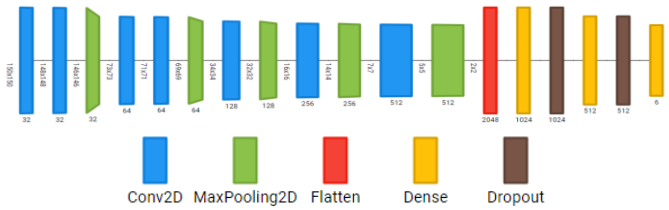


Fig. 10. Arquitectura Modelo-II

C. **Entrenamiento:** En la fase de entrenamiento de este modelo se agrega un nuevo concepto como técnica de regularización: *Early Stopping* que, nos permite especificar una cantidad arbitrariamente grande de épocas de entrenamiento y detener el entrenamiento una vez que el rendimiento del modelo deja de mejorar en el conjunto de datos de validación. El entrenamiento se detendrá cuando la medida elegida deje de mejorar. En nuestro caso hemos elegido monitorizar el valor de *'val_loss'* , agregando el parámetro '*patience'* con valor 3, queremos decir que debe de haber alguna mejora de nuestro indicador en 3 épocas.

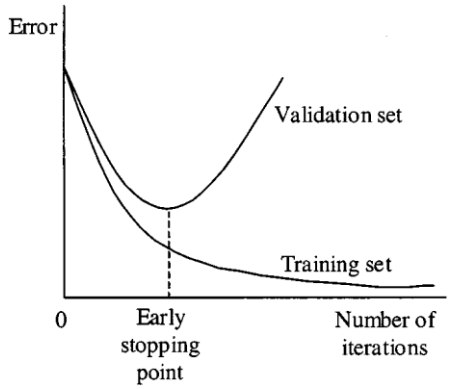


Fig. 11. *Early Stopping* [16]

Inicializamos el entrenamiento con 20 épocas, sin embargo, gracias a la técnica de Early Stopping nuestro modelo finaliza el aprendizaje en la época número 17 tras comprobar que nuestro parámetro a monitorizar no disminuye tras los '*epochs'* 15, 16 y 17.

```
Epoch 14/20
350/350 [==============================] - 106s 302m
- val_loss: 0.4569 - val_accuracy: 0.8423
Epoch 15/20
350/350 [==============================] - 106s 304m
- val_loss: 0.4713 - val_accuracy: 0.8279
Epoch 16/20
350/350 [==============================] - 104s 297m
- val_loss: 0.4850 - val_accuracy: 0.8333
Epoch 17/20
350/350 [==============================] - 104s 297m
- val_loss: 0.4604 - val_accuracy: 0.8441
```

Fig. 12. Captura extraida del entrenamiento con *Early Stopping* del Modelo-II

D. **Evaluación:** Tras evaluar el Modelo-II con los datos de prueba, obtenemos los siguientes resultados: 86.2% de '*accuracy'* y 0.39 de '*loss'*. Hasta el momento ha sido el mejor modelo con respecto a estas dos métricas.

E. **Predicción:** Como en los modelos anteriores, probaremos a usar el método '*predict'* pasandole como parámetro alguna imagen del directorio de pruebas. En este caso pasamos una imagen de la clase 'sea' dando como resultado la siguiente salida:

```
Predicción de la imagen: sea
```

Como podemos comprobar nuestro modelo ha elegido la etiqueta correcta a pesar de hacer varias pruebas con varias imágenes las cuales nuestro modelo no acierta con la clase correcta.Visualizamos la matriz de confusión y vemos como las relaciones de confusión entre '*mountain'* - '*glacier'* y '*street - buildings'* siguen tomando valores altos. Aun asi la generalización referente a este modelo es mejor con respecto a los dos anteriores ya que la tasa de acierto de las etiquetas tras la predicción aumenta.

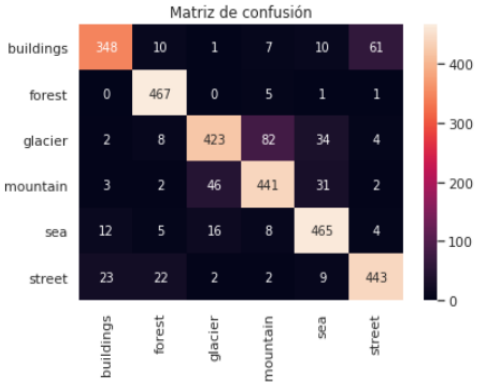


Fig. 15. Matriz de confusión del Modelo-II

Modelo-III (VGG16 – TL)

La implementación de este modelo se ha llevado a cabo mediante el método de aprendizaje automático de '*Transfer Learning'* en el que un modelo desarrollado para una tarea se reutiliza como punto de partida para un modelo en una segunda tarea. El modelo elegido es VGG16. Nos centraremos en los aspectos más relevantes de la arquitectura, entrenamiento y predicción.

A. **Arquitectura:** La arquitectura del modelo VGG16 tiene una entrada a la primera capa convolucional de tamaño 224 x 224 con 3 canales de color. La matriz pasa a través de una pila de capas convolucionales, donde se usan filtros de tamaño 3 x 3 y 1 x 1. La agrupación espacial se construye mediante 5 capas de *MaxPooling* con ventanas de 2 x 2. Como último bloque se compone de 3 capas '*fully connected'* con 4096 neuronas las dos primeras y 1000 la última, coincidiendo con el número de clases pertenecientes al subconjunto de ImageNet [17].

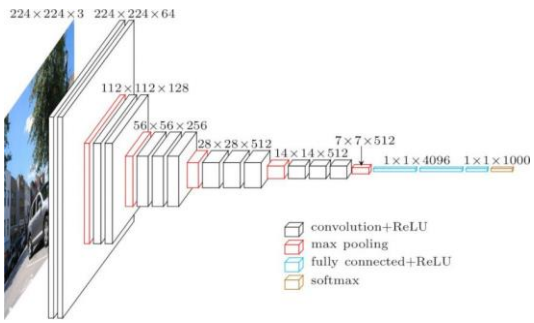


Fig. 13. Arquitectura VGG16

Para importar el modelo VGG16, Keras nos ofrece una librería con los modelos mas relevantes en el mundo del

Machine Learning, pudiendo adaptar dichos modelos a problemas concretos. En este proyecto se importará el modelo con los pesos includios de 'ImageNet', no se entrenará ninguna capa y finalmente agregaremos el siguiente bloque: *Flatten* + *Dense* (256) + *Dropout* (0.5) + *Dense* (6 = número de etiquetas en nuestro trabajo).

En este caso compilaremos el modelo con el optimizador '*Nadam'*, función de coste de tipo '*categorical crossentropy' y* métrica '*accuracy'*.

B. **Entrenamiento:** Entrenamos el modelo durante 15 '*epochs'* y detallamos los resultados en la Fig. 14.

Observamos como el modelo en tan solo 15 'epochs' consigue un 89% de 'accuracy' en los datos de validación. Es bastante ventajoso el uso del '*Transfer Learning'*, ya que nos ahorramos bastante tiempo en crear un modelo y sobre todo en entrenarlo.

La idea clave aquí es simplemente aprovechar las capas ponderadas del modelo previamente entrenado para extraer características, pero no actualizar las ponderaciones de las capas del modelo durante el entrenamiento con nuevos datos para la nueva tarea [18].

C. **Evaluación:** Evaluamos el Modelo-III (VGG16 – TL) y obtenemos los siguientes resultados:

```
100/100 [==============================] - 5s 53ms/step -
loss: 0.3045 - accuracy: 0.8867
Loss: 0.30445656180381775 Accuracy: 88.66666555404663
```

¡No esperabamos menos! Los valores obtenidos tras evaluar el modelo con los datos de pruebas son bastante buenos.

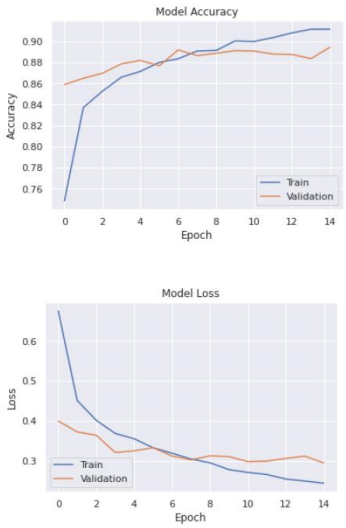
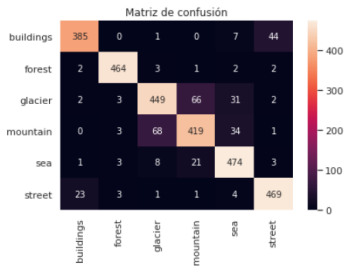


Fig. 14. Comparativa del Modelo-III (VGG16 - TL) desde el punto de vista de la precisión y función de pérdida durante las 15 épocas

D. **Predicción:** Para la predicción pasamos directamente a contrastar la matriz de confusión:



Observamos que a pesar del buen resultado tras evaluar el modelo, persisten los problemas de las clases 'buildings' – 'street' y 'mountain' – 'glacier'.

IV. Resultados

En esta sección presentaremos una comparativa de los modelos previamente descritos y haremos una evaluación de que modelo generaliza mejor, excluyendo el Modelo-III (VGG16-TL).

Table 2. Comparativa de modelos

Nombre del modelo	***Test loss***	***Test accuracy***
Modelo-I (Base)	1.39	78.70%
Modelo-I (DataAug + Dropout)	0.43	85.43%
Modelo-II	0.39	86.23%

Si nos fijamos en los valores anteriores, el Modelo-I (Base) queda descartado de la elección ya que si recordamos cuando hablamos de él, podíamos encontrar un patrón que daba lugar a la presencia de '*overfitting'*. Sin embargo, el Modelo-I con regularización y el Modelo-II tienen métricas muy parecidas por eso desempatamos mediante la matriz de confusión, siendo entonces el modelo elegido: Modelo-II.

V. ANEXO

En esta sección ampliaremos el estudio de las Redes Neuronales Convolucionales y hablaremos sobre la arquitectura de ResNet50, exponiendo los elementos que componen introduciendo el concepto de Redes Neuronales Residuales.

Tras el estudio realizado sobre las Redes Neuronales Convolucionales, podemos afirmar que la dificultad estriba en el diseño de la arquitectura, por ejemplo, cuántas capas de convolución debemos agregar, qué tamaño de filtro será óptimo o que técnicas de regularización debo de someter a mi modelo para que generalice lo mejor posible. Por eso un enfoque útil para aprender a diseñar arquitecturas efectivas es estudiar aplicaciones exitosas [12].

La clave del diseño de ResNet50 es la idea de bloques residuales que hacen uso de conexiones de acceso directo. Un bloque residual es un patrón de dos capas convolucionales con activación ReLU donde la salida del bloque se combina con la entrada al mismo. Esta arquitectura tiene una profundidad de 152 capas la cual permitió el desarrollo del *Deep Learning*.

Como ya sabemos, las redes profundas son muy difícil de entrenar debido a que cuando aplicamos '*Backpropagation'* al comienzo del modelo, reducimos el valor muy cercano a 0, por lo que el cambio en los pesos de esas capas es casi insignificante. Entonces, se puede decir que estas capas no contribuyen tanto en el proceso de aprendizaje. Esta limitación se conoce como '*Gradient vanishing'* [14] .

Para solucionar este problema, Kaiming He propuso una medida denominada 'Conexiones de salto' que consiste en agregar directamente la activación de una capa en particular a la activación de alguna otra capa que está "más profunda" en la red. Gracias a esta técnica pudieron entrenar una red neuronal con una gran cantidad de capas sin dejar de tener una complejidad menor que VGGNet.

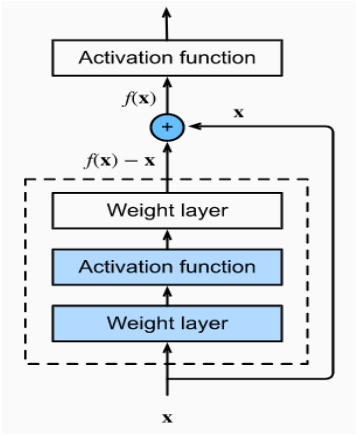


Fig. 15. Esquema de un Bloque Residual

En la **¡Error! No se encuentra el origen de la referencia.** se tiene el input *x* que viajara por la red mediantes las capas de convolución y activación entre otras. El resultado de estas operaciones es f(x). Nuestro objetivo en una red convolucional tradicional es que f(x) y f(x) + x tengan el mismo valor. Con los bloques residuales se calcula el valor que tiene que añadir f(x) a su entrada x. Ahora que ya hemos explicado en que consiste un bloque residual procederemos a describir las capas que componen ResNet50. Existen dos tipos de bloques involucrados en esta arquitectura: Bloque Identidad y Bloque Convolucional.

- Bloque Identidad : consta de 3 capas de convolución con sus respectivas funciones de activación y normalización por lotes. Se usa cuando la entrada y la salida tienen la misma dimensión.

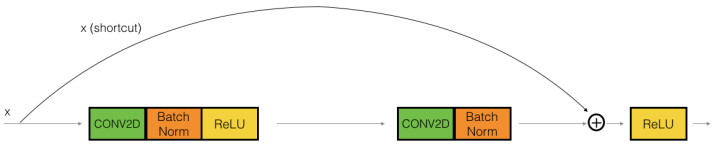


Fig. 16. Ejemplo de Bloque Identidad

- Bloque Convolucional: Se compone de los mismo elementos que los bloques identidad sin embargo, se usan cuando la entrada y la salida no tienen la misma dimensión usando una capa convolucional para reducir la dimensión de la entrada.

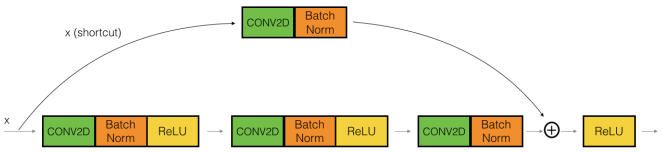
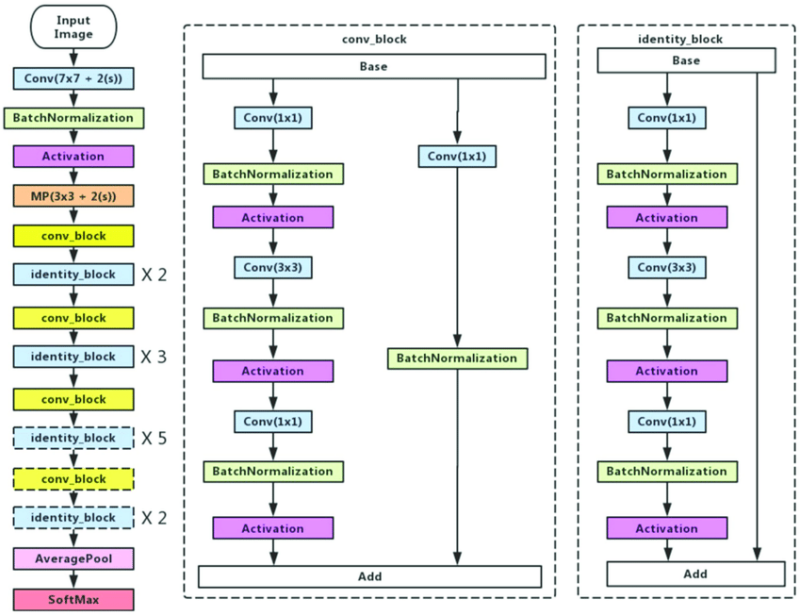


Fig. 17. Ejemplo de Bloque Convolucional

A continuación se muestra un diagrama de la arquitectura de ResNet50, donde podemos observar un bloque principal compuesto de: *Convolution* + *BatchNormalization* + *Activation* + *MaxPooling*, seguido de un conjunto de bloques de identidad y convolución. Finalizando con una capa de *AveragePooling* y *Dense* con función de activación '*Softmax'*

Fig. 18. Arquitectura ResNet50

ResNet se convirtió en el ganador de ILSVRC 2015 en clasificación, detección y localización de imágenes, así como en el ganador de detección y segmentación de MS COCO 2015. ILSVRC utiliza un subconjunto de ImageNet de alrededor de 1000 imágenes en cada una de las 1000 categorías. En total, hay aproximadamente 1,2 millones de imágenes de



formación, 50.000 imágenes de validación y 100.000 imágenes de prueba.

Seguidamente comparamos el error de validación de una red neuronal tradicional frente a ResNet aumentando el número de capas. Cuando se usa una red simple, 18 capas es mejor que 34 capas, debido al problema del gradiente de desaparición. Cuando se usa ResNet, 34 capas es mejor que 18 capas, el problema del gradiente que se desvanece se ha resuelto omitiendo conexiones. Si comparamos la red simple de 18 capas y la ResNet de 18 capas, no hay mucha diferencia. Esto se debe a que el problema del gradiente de desaparición no aparece en redes poco profundas.

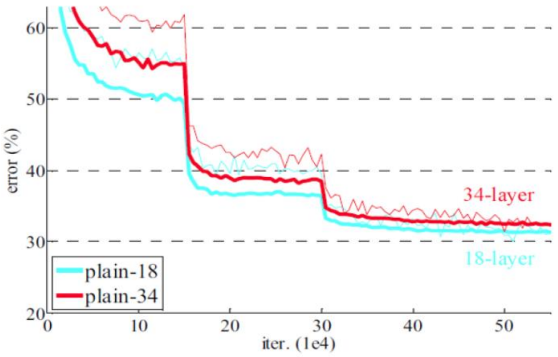


Fig. 19. Error de validación de una red neuronal tradicional

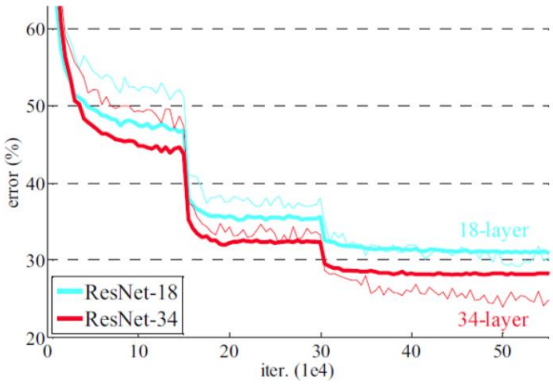


Fig. 20. Error de validación de ResNet

VI. CONCLUSIONES

Finalmente, puedo decir que he aprendido bastante con este trabajo, llegado incluso a probar muchísimas combinaciones de arquitecturas y técnicas de regularización, pero algunas de ellas como '*Batch Normalization'*, *L1* y *L2* no fueron muy efectivas para las arquitecturas e hiperparámetros de mis modelos. Sin embargo, como he leído en muchos artículos sigo sin saber definir cuantos filtros son los óptimos en cada capa convolucional o cuantas capas convolucionales serán necesarias para extraer todas las características mas relevantes de este conjunto de datos. Otro problema que se me alzó fue el límite de GPU en Google Colab y fue entonces cuando conocí el mejor cuaderno de trabajo, Kaggle, con la sorpresa de la velocidad que tenía al entrenar la red, 4 veces mas rápido y eso que dura 12 – 15 minutos.

Pero lo que tengo claro es la aportación tan grande que nos ofrece la Inteligencia Artificial hoy en día, que, visto desde un lado más práctico, gusta más.

REFERENCIAS

[1] https://www.youtube.com/watch?v=KytW151dpqU.

[2] https://blog.techdata.com/ts/latam/la-historia-de-las-redes-neuronales-y-la-ia-parte-3?hs_amp=true

[3] https://www.atriainnovation.com/que-son-las-redes-neuronales-y-sus-funciones/

[4] https://knepublishing.com/index.php/KnE-Engineering/article/view/1462/3528#:~:text=Para%20el%20reconocimiento%20de%20im%C3%A1genes,et%20al.%2C%202012).

[5] https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/#:~:text=La%20capa%20de%20reducci%C3%B3n%20o,dimensi%C3%B3n%20de%20profundidad%20del%20volumen.

[6] https://becominghuman.ai/the-idea-of-preparing-your-own-dataset-for-convolutinal-neural-network-49ec097b8313

[7] https://torres.ai/data-augmentation-y-transfer-learning-en-keras-tensorflow/

[8] https://www.infoworld.com/article/3336192/what-is-keras-the-deep-neural-network-api-explained.html

[9] https://www.interactivechaos.com/manual/tutorial-de-machine-learning/adam

[10] https://www.researchgate.net/figure/Comparison-of-different-optimizer-by-training-of-multilayer-neural-networks-on-MNIST_fig1_324808725

[11] https://viscom.net2vis.uni-ulm.de/bAqMlTCIYiMeW8G5O4n7C0uF7nYbZmLZB6RMKkYXU3zrWXLikl

[12] https://sitiobigdata.com/2019/05/01/innovaciones-arquitectonicas-redes-neuronales-clasificacion-imagenes/

[13] https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33

[14] https://www.wandb.com/articles/exploring-resnets-with-w-b

[15] https://www.researchgate.net/figure/The-proposed-Resnet50-CNN-architecture_fig3_338253578

[16] https://www.researchgate.net/figure/Early-stopping-based-on-cross-validation_fig1_3302948

[17] https://neurohive.io/en/popular-networks/vgg16/

[18] https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a

[19]