Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones

Máster Oficial en Ingeniería Informática

Curso 2020/2021

PRÁCTICA DE REDES NEURONALES

Inteligencia Computacional

Breve descripción

Reconocimiento óptico de caracteres MNIST

Autor

Álvaro de la Flor Bonilla

RESUMEN

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales. Deberá evaluar el uso de varios tipos de redes neuronales para resolver un problema de OCR: el reconocimiento de dígitos manuscritos de la base de datos MNIST (http://yann.lecun.com/exdb/mnist/).

1 ÍNDICE

ÍNDICE DE ILUSTRACIONES

1 INTRODUCCIÓN

En primer lugar, todo el código de la implementación desarrollado se encuentra disponible en GitHub, al que puede acceder a través de este enlace (https://github.com/alvarodelaflor/ic/). Durante todo el proyecto se ha trabajado sobre él, por lo que por otro lado también puede ver la evolución del proyecto.

1.1 Objetivos e implementación

Para la realización de esta práctica se ha optado por la utilización de una implementación externa, en este caso *"Keras" (puede obtener más información en el enlace anterior).*

Se propone como ejercicio entrenar distintas redes neuronales artificiales sobre el conjunto de entrenamiento indicado en el resumen de este documento y posteriormente evaluar los modelos obtenidos a partir de los datos del conjunto de prueba.

En este caso, se han desarrollado cuatro estrategias de implementación, aunque nos hemos centrado en dos. En concreto, se ha realizado:

1. Red Neuronal Simple
2. **Red Neuronal Multicapa**
3. **Red Convolucional**
4. Autoencoders.

Durante el desarrollo de esta práctica, se ha puesto un especial interés en el desarrollo de la implementación de una *"Red Neuronal Multicapa"* y de una *"Red Convolucional"*. El objetivo de los siguientes apartados es explicar cómo se llevó a cabo cada uno de sus desarrollos y cuáles son los aspectos fundamentales de cada una de ellas.

1.2 Patrón de desarrollo

Para todas las implementaciones realizadas se ha estructurado el desarrollo en un mismo patrón de cuatro métodos.

1.2.1 Crear modelo

En este método en primer lugar se recogen y leen los datos necesarios para entrenar la red. Una vez obtenidos los datos necesarios se crea el modelo (que varía en función de la estrategia elegida). Finalmente se comienza el entrenamiento y una vez concluido se realiza una evaluación inicial utilizando los datos de entrenamiento.

1.2.2 Guardar modelo

Simplemente es un método utilizado para dar consistencia al modelo creado en el método anterior, evitando tener que entrenar de nuevo a la red.

1.2.3 Evaluate

Una vez almacenado el modelo, este método permite volver a reevaluar el modelo creado, utilizando para ello tanto los datos de entrenamiento como los datos de evaluación (ambos descritos en el resumen de este documento).

1.2.4 Aplicar modelo

Este método permite hacer un uso gráfico de la predicción del desarrollo realizado, es decir, utiliza el modelo diseñado para predecir el resultado de manera aleatoria de algún ejemplo del conjunto de prueba. El usuario puede ver por consola la imagen que se analiza, el resultado que predice el modelo y la respuesta correcta indicada por la etiqueta relacionada a esta imagen.

2 USO DE LA IMPLEMENTACIÓN

Para el uso de la implementación desarrollada se ha realizado un sencillo script para su uso.

2.1 Prerrequisitos

Para el desarrollo de esta implementación se ha utilizado Python, en su versión "3.8".

Por consecuencia a lo anterior, es necesario crear un entorno virtual acorde a esta versión del lenguaje. Además, se tendrán que instalar las dependencias que se indican en el archivo *"requirement.txt", que se encuentra disponible en la carpeta "Implementación".*

2.2 Ejecución del código

Una vez satisfechos los requisitos establecidos en el punto anterior, basta con ejecutar el archivo *"main.py"*, el cual de manera iterativa le mostrará al usuario las distintas opciones disponibles.

2.2.1 Elección de la implementación

En primer lugar, siempre y cuando se hayan realizado los pasos previos descritos en el punto *"2.1 Prerrequisitos"*, el script solicitará al usuario que elija la implementación que desea utilizar. En concreto se han desarrollado cuatro estrategias:

1. Red Neuronal Simple
2. Red Neuronal Multicapa
3. Red Convolucional
4. Autoenconders

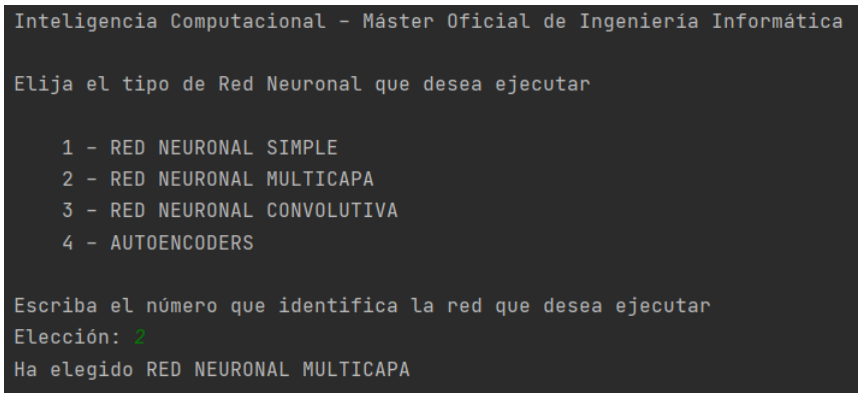

Ilustración 1 - Elección de la estrategia

En el ejemplo de la imagen anterior muestra el inicio del script. En este caso el usuario ha decidido elegir la opción 2 (Red Neuronal Multicapa), para ello lo único que debe hacer es escribir *"2"* en el input de la consola.

2.2.2 Entrenamiento de la red

Tras elegir la estrategia deseada por el usuario, el sistema le solicitará al usuario que le indique si desea entrenar la red. El código que se entrega posee un modelo ya entrenado, en el caso que solicite volver a entrenar la red este modelo será sobrescrito por la nueva red entrenada.

Si se decide no entrenar la red, se pasará al proceso de entrenamiento, en caso contrario se comenzará el proceso de las sucesivas iteraciones de entrenamiento.

Ilustración 2 - Entrenamiento de la red I

El proceso comienza imprimiendo un valor aleatorio del conjunto de imágenes y de etiquetas para comprobar que se tiene acceso a ellas.

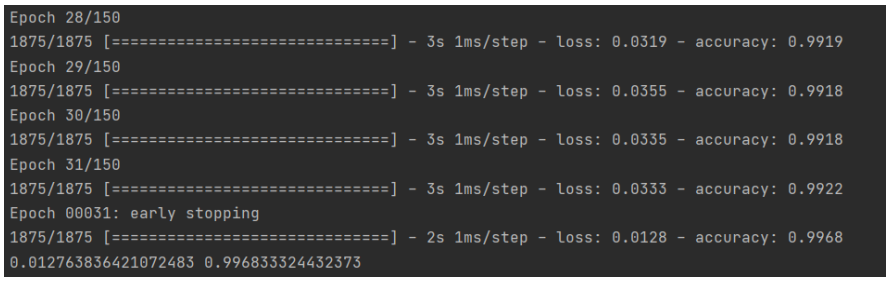

Ilustración 3 - Entrenamiento de la red II

Posteriormente se inicia un proceso de sucesivas iteraciones que concluyen con un modelo entrenado y con la muestra de resultados de la exactitud obtenida en relación con los datos de entrenamiento.

2.2.3 Evaluar la red

Cuando se termina el proceso de entrenamiento, se le permite elegir al usuario entre evaluar la red o no. Si finalmente elige que si desea evaluar la red (escribiendo *"s"* en el input de consola), tendrá que elegir entre utilizar para esta evaluación los datos de entrenamiento y los datos de test (escribiendo *"e"* o *"t"* respectivamente). Lo más lógico para estas evaluaciones es utilizar los datos de test.

```
¿Desea evaluar la red? (s/n)
Elección: s
¿Desea utilizar datos de entrenamiento (e) o de test (t)? (e/t)
Elección: t
313/313 [==============================] - 0s 1ms/step - loss: 0.1188 - accuracy: 0.9821
0.11877938359975815 0.9821000099182129
```

Ilustración 4 - Evaluación de una red

En la imagen anterior se ha evaluado una red, utilizado los datos de test y obteniendo una exactitud del 98,21 %.

2.2.4 Ejemplo de la predicción

Por último, el sistema permite que, si se dispone de un modelo almacenado se pueda aplicar de forma gráfica.

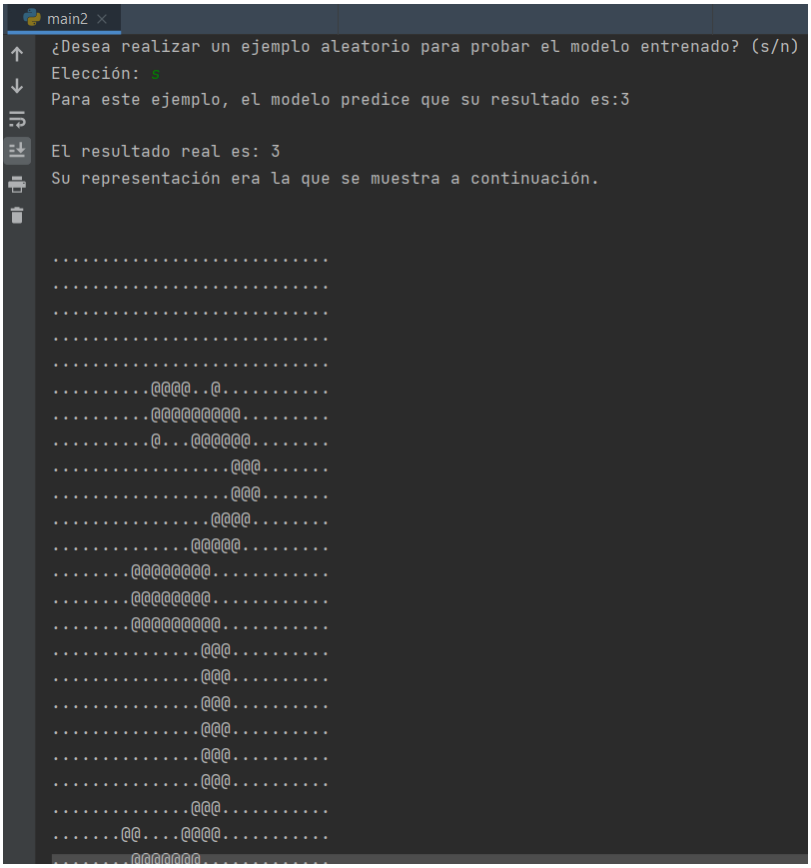

Ilustración 5 - Aplicar modelo

Es decir, tal y como se muestra en la ilustración anterior, la última acción interactiva consiste en que, si el usuario lo desea, el sistema elige una imagen de forma aleatorio, solicita al sistema que prediga el resultado que evalúa como correcto y posteriormente se indica el resultado real asociado con la etiqueta de la imagen evaluada.

Centrándonos en el ejemplo anterior el modelo ha acertado, ya que ha indicado que el resultado de la imagen que podemos ver es tres, tal y como indica la etiqueta asociada.

3 IMPLEMENTACIONES

En este apartado nos centraremos, para cada una de las distintas implementaciones, en la forma en la que se ha construido el modelo, teniendo en cuenta que para todas ellas se ha utilizado la librería *"Keras"*.

3.1 Red Neuronal Simple

No nos vamos a centrar mucho en la descripción de esta implementación ya que la siguiente, *"Red Neuronal Multicapa"* es una ampliación que toma como base esta. Básicamente utilizaremos esta sección para explicar las bases de *"Keras"* y como funciona su configuración, centrándonos en el significado de los parámetros más importantes que debemos configurar.

3.1.1 ¿Qué se ha hecho?

En primer lugar, se han **normalizado los datos**, tanto de las imágenes como de las etiquetas. *"Keras"* requiere de la normalización tanto de las imágenes como de los datos de las etiquetas.

Por ejemplo, en el caso de las imágenes cada uno de los puntos tiene valores que van de 0 a 255. Gracias a la normalización conseguimos que estos datos vayan de 0 a 1. A medida que el rango de los valores de las características se reduce a un rango particular debido a la normalización, es fácil realizar cálculos en un rango de valores más pequeño. Por lo tanto, normalmente el modelo se entrena un poco más rápido, es por ello por lo que *"Keras"* requiera de la normalización de los datos.

Una vez obtenidos los datos normalizados se comienza con la **construcción del modelo**. Vamos a replicar el ejemplo que se nos propone en el enunciado de la práctica, es decir, una red neuronal simple con una capa de entrada y otra de salida de tipo *"softmax"*.

La primera acción que realizar es constituir el modelo, para ello se inicia con la llamada a *"Sequentials"*, la cual nos permite crear el esqueleto sobre el que añadir las sucesivas capas y por consiguiente el modelo completo.

Como dijimos que en esta ocasión nos proponemos a imitar el ejemplo que se describe en el enunciado, la siguiente acción es añadir una capa de entrada, en este caso se va a utilizar una matriz aplanada con 784 elementos (el tamaño de las imágenes es 28x28, ahí este número). Para realizar esto en *"Keras"* basta con añadir al modelo creado anteriormente un *"InputLayer"* parametrizado con el tamaño de la matriz aplanada.

Finalmente añadimos al modelo una capa de salida de 10 unidades lógicas con una función de activación de tipo *"softmax"* ("*la función softmax transforma las salidas a una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas de 1*").

A partir de este momento, el modelo ya esta construido, el siguiente paso (ya que estamos usando *"Keras"*) consiste en la **compilación del modelo**. Para la compilación solo necesitamos configurar en esta ocasión 3 parámetros.

1. **Optimizer**. Cada vez que una red neuronal termina de pasar un lote por la red y de generar resultados de predicción, debe decidir cómo utilizar la diferencia

entre los resultados que obtuvo y los valores que sabe que son verdaderos para ajustar los pesos de los nodos de modo que la red camine hacia una solución. El algoritmo que determina ese paso se conoce como el algoritmo de optimización.
En este caso se ha utilizado *"adam",* que es el que mejor resultado ha mostrado tras las diversas pruebas. Para esta implementación también se han utilizado *"adadelta"*, *"adamax"* y *"nadam"* pero han mostrado peores resultados.

2. **Loss**. En cada iteración, el error del estado actual del modelo debe ser estimado repetidamente, por lo cual es necesario la elección de una función de error (o pérdida) del modelo para que los pesos puedan actualizarse y reducir su pérdida en la siguiente iteración.
   Se han utilizado en esta ocasión también diversas configuraciones como *"mean_squared_error"*, *"mean_squared_logarithmic_error"*, *"mean_absolute_error"*, *"binary_crossentropy", "hinge"*, *"kullback_leibler_divergence"* y *"sparse_categorical_crossentropy"*. Tras probar todas estas variantes finalmente *"sparse_categorical_crossentropy"* ha mostrado mejores resultados.

3. **Metrics**. Una métrica es una función que se utiliza para juzgar el rendimiento de su modelo. Las funciones métricas son similares a las funciones de pérdida, excepto que los resultados de la evaluación de una métrica no se utilizan al entrenar el modelo. En esta ocasión se utilizará la variante *"accuracy"* la cual calcula la frecuencia con la que las predicciones son iguales a las etiquetas.

Cuando contamos con el modelo compilado puede comenzarse a **entrenar la red**. Como utilizamos *"Keras"* lo realizaremos convocando a la función *"fit"*. Esta función necesitará una vez más como en el caso anterior de tres parámetros que serán las imágenes que se van a predecir, las etiquetas que dan solución a esas imágenes y por tanto permiten realizar un entrenamiento supervisado y finalmente el número de *"epochs"*. El número de *"epochs"* es un hiperparámetro que define el número de veces que el algoritmo de aprendizaje funcionará a través de todo el conjunto de datos de entrenamiento.

Una vez entrenada la red se habrá concluido todo el proceso de aprendizaje y el único paso restante será la **evaluación de la red**. Para hacer esto *"Keras"* lo permite realizar de una forma muy sencilla, tan solo hay que convocar al método *"evaluate"* desde el modelo, y pasarle como parámetros el conjunto de imágenes y etiquetas que se utilizarán para realizar este examen. Cabe destacar que es muy importante que este conjunto sea diferente al utilizado para el entrenamiento, de otra forma no obtendríamos datos reales ya que durante el entrenamiento se podría haber realizado un sobreajuste y la red solo habría memorizado su conjunto de entrenamiento.

3.1.2 Resultados

Al utilizar una capa tan simple y con tan poca configuración, los resultados obtenidos son muy malos.

Finalizado el entrenamiento (15 epochs) se tiene una precisión final del 91,38%, sin embargo, si se realiza esta evaluación sobre el conjunto de prueba la exactitud baja

hasta el 10,17%. La red no ha sido capaz de realizar un buen entrenamiento y básicamente lo que ha hecho es memorizar el conjunto utilizado para entrenarse.

En la siguiente implementación se comentará las estrategias seguidas para evitar que ocurran estas situaciones.

3.2 Red Neuronal Multicapa

Esta red es una en la que hemos centrado nuestros esfuerzos de desarrollo. Parte de la base de una red neuronal simple, la explicada en el caso anterior. La diferencia principal es el número de capas ocultas que se le han añadido al modelo, y la conexión que mantienen entre ellas.

A lo largo de este apartado pasaremos a comentar las estrategias que se han llevado a cabo para intentar mejorar el rendimiento y comentar los resultados obtenidos.

3.2.1 ¿Qué se ha hecho?

De igual forma que en el caso anterior, el primer paso necesario ha sido la normalización de los datos. Se ha realizado exactamente igual, haciendo uso de la librería *"numpy"* y *"keras normalize"*.

Ya que disponemos de los datos normalizados comenzaremos con la construcción del modelo, una vez más haciendo uso de la función *"Sequentials"* que permite construir el esqueleto sobre el que se irán añadiendo las sucesivas capas.

Antes de comenzar con la descripción de la construcción del modelo vamos a comenzar un poco lo que pretendemos hacer.

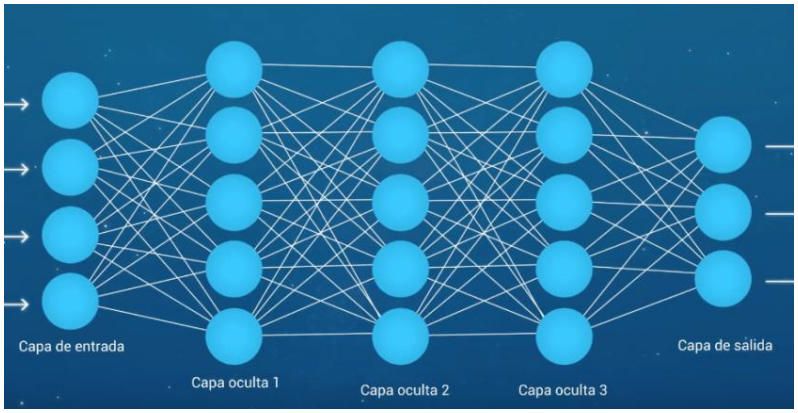

Ilustración 6 - Red Neuronal Multicapa

En definitiva, la estructura de una red neuronal multicapa sigue la estrategia de la ilustración de arriba. La red esta conformada por una capa de entrada y otra de salida, pero entre ellas existe una anidación de capas ocultas (o intermedias) interconectadas entre todas ellas. Para cada una de estas capas se puede configurar una estrategia diferente, que adecúan el modelo construido hasta conseguir una basta red capaz de resolver problemas que no son linealmente separables.

En nuestro caso, la primera capa que se ha añadido es *"Flatten"*. Utilizaremos una imagen para que se entienda un poco mejor su funcionalidad.

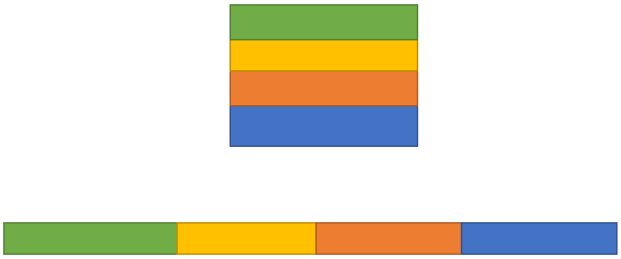

Ilustración 7 - Keras Flatten

Básicamente, el objetivo que pretende Flatten es convertir un matriz en un simple array. Recordemos que nuestros datos de entrada son una matriz de valores y es con estos datos con los que trabaja Flatten.

Tras el proceso anterior, nuestra red esta preparada para comenzar a configurar las capas ocultas que se van a utilizar.

Nuestra configuración para esta red se ha basado en realizar lo siguiente:

DENSE

Es la forma de añadir capas completamente conectadas en *"Keras"*. Cada una permite realizar una configuración especial, con el uso de tres parámetros.

1. **Unidades lógicas**. Establecen la dimensionalidad de la salida de la capa. Por ejemplo, en la ilustración 6, la capa oculta 1 tiene cinco unidades lógicas.
2. **Input Dim**. Establece el valor de la entrada unidimensional. Si tuviese más de una dimensión se utilizaría *"input_shape"*.
3. **Activation**. Las funciones de activación son ecuaciones matemáticas que determinan la salida de una red neuronal. La función se adjunta a cada neurona de la red y determina si debe ser activada ("disparada") o no, en función de si la entrada de cada neurona es relevante para la predicción del modelo. Las funciones de activación también ayudan a normalizar la salida de cada neurona en un rango entre 1 y 0 o entre -1 y 1.
4. **Kernel Constraint**. Impone una regla dura sobre los pesos. Un ejemplo común es la norma máxima que obliga a la norma vectorial de los pesos a estar por debajo de un valor, como 1, 2, 3. Una vez superada, todos los pesos del nodo se hacen lo suficientemente pequeños para cumplir la restricción.

Para nuestro proyecto se han creado 5 capas ocultas y una se salida:

Capa 1

Configurada con 256 unidades lógicas y una función de activación de tipo *"Relu"*.

Capa 2

Configurada con 128 unidades lógicas y una función de activación de tipo *"Relu"*.

Capa 3

Configurada con 64 unidades lógicas, una función de activación de tipo *"Relu"* y un *"kernel constraint"* con *"maxnorm" 6*.

Capa 4

Configurada con 32 unidades lógicas y una función de activación de tipo *"Relu"* y un *"kernel constraint"* con *"maxnorm" 6*.

Capa 5

Configurada con 16 unidades lógicas y una función de activación de tipo *"Relu"* y un *"kernel constraint"* con *"maxnorm" 3*.

Capa Salida

Configurada con 10 unidades lógicas y una función de activación de tipo *"softmax"*.

En cuanto a la función de activación *"Relu",* de entre todas las escogidas ha sido la que mejor desempeño ha demostrado. Se han probado las mismas que se detallaron en el caso de la red neuronal simple del caso anterior, pero todas han mostrado un rendimiento inferior.

Utilicemos una comparativa entre la función de activación *"relu"* y *"sigmoid"* que son las funciones no lineales que mejor resultado han dado en nuestros entrenamientos.

```
¿Desea utilizar datos de entrenamiento (e) o de test (t)? (e/t)
Elección: t
313/313 [==============================] - 0s 954us/step - loss: 0.1143 - accuracy: 0.9819
0.11433009058237076 0.9818999767303467
```

Ilustración 8 - Evaluación con la función de activación Sigmoid

En primer lugar, han sido necesarios la realización de 86 epochs (nuestra red posee una función de parada que posteriormente será comentada). Como resultado, se ha obtenido una exactitud del 98,19% (un error del 1,81%).

```
2020-12-13 13:01:18.558924: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1257] Device interc
2020-12-13 13:01:18.557094: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1263]
313/313 [==============================] - 0s 926us/step - loss: 0.1275 - accuracy: 0.9828
0.12746568024158478 0.9828000068664551
```

Ilustración 9 - Evaluación con la función de activación Relu

En cambio, utilizando la función de activación *"Relu"* solo han sido necesarios realizar 39 epochs (46% respecto al caso anterior). Además, se ha incrementado la exactitud del modelo generando, alcanzando una precisión del 98,28% (un error del 1.72 %).

En cuanto al uso de *"kernel_constraint"* he podido comprobar que, aunque sea una mejora ínfima, ayuda a un mejor rendimiento en el entrenamiento de la red. Los valores se han ido modificado aleatoriamente hasta encontrar la configuración óptima para nuestra red.

DROPOUT

Esta funcionalidad de *"Keras"* consiste en desactivar un número de nuronas de una red neuronal de forma aleatoria. Es decir, en cada iteración de la red neuronal se desactivan diferentes neuronas las cuales no se toman en cuenta ni si quiera para el "***backward propagation"*** por lo que se le obliga a las neuronas cercanas a no depender tanto de las neuronas desactivadas.

En cuanto al *"backward propagation"* no hemos comentado su implementación ya que la librería que utilizamos (*"Keras"*) hace su uso internamente sin necesidad de que tengamos que hacer implícita la decisión de usarlo.

Respecto a nuestra red, hemos hecho uso de esta técnica en cuatro ocasiones, siguiendo un enfoque reductivo. En las primeras capas ocultas hemos forzado con mayor nivel a la desactivación de estas neuronas, reduciendo este esfuerzo a medida que se avanzas en las distintas capas.

Early Stopping

Uno de los puntos fundamentales que se deben configurar al construir una red es el criterio para elegir el número de iteraciones o epochs. En nuestro caso hemos utilizado una funcionalidad disponible en *"Keras"* que permite un criterio de parada variable en función de la decisión que tome el desarrollador. En nuestro caso hemos decidido que este criterio dependa de los valores de pérdida, estableciéndose que en el caso de que se repita tres veces el mismo valor (o se den tres veces resultados peores) se pare la ejecución de la iteración de entrenamiento.

3.2.2 Resultados

A diferencia del caso anterior en esta ocasión se han obtenido muchísimos mejores resultados.

```
2020-12-13 13:52:46.545735: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1263]
313/313 [==============================] - 0s 930us/step - loss: 0.1275 - accuracy: 0.9828
0.12746568024158478 0.9828000068664551
```

Ilustración 10 - Evaluación de RNM con conjunto de prueba

```
2020-12-13 13:53:57.770354: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1263]
1875/1875 [==============================] - 2s 928us/step - loss: 0.0041 - accuracy: 0.9990
0.004144558683037758 0.9990333318710327
```

Ilustración 11 - Evaluación de RNM con conjunto de entrenamiento

Tras el entrenamiento, el error sobre el conjunto ha sido únicamente del 0,0967% con una pérdida del 0,0041%. Este mismo modelo aplicado sobre el conjunto de prueba ha mostrado un **error del 1,72%** con una pérdida del 12,74%.

Consideramos estas cifras un muy buen desempeño de la red finalmente desarrollada, especialmente teniendo en cuenta las cifras descritas en la sección de la red neural simple del primer apartado.

3.3 Red Neuronal Convolutiva

En primer lugar, hay que aclarar que para la realización de esta implementación ha sido casi obligatorio el uso y configuración de la GPU, ya que para cada iteración (*"epoch"*) se consumían de 12 a 16 segundos. Una vez configurada esta característica los tiempos se han reducido a tiempos entre 3 y cuatro segundos.

Para poder realizar esta configuración es necesario disponer de una GPU Nvidea, e instalar el software *"cuda"*, en concreto su versión 10.1, si desea más información puede visitar este enlace.

Antes de comenzar con la explicación del desarrollo realizaremos una breve introducción a las redes convolutivas.

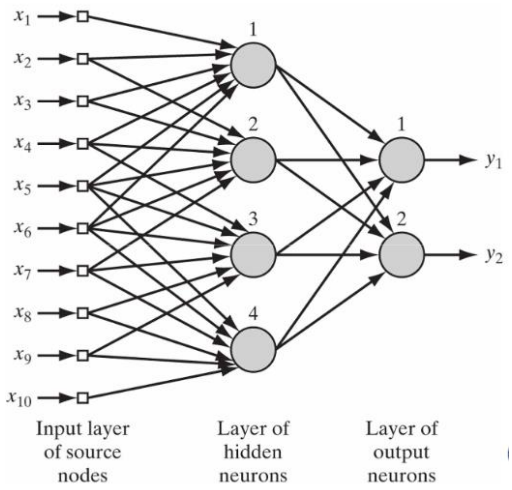

Ilustración 12 - Esquema red Convolutiva

La idea principal de una red convolucional es que se filtren las imágenes antes de entrenar a la red neuronal profunda. Después de filtrar, las características de la imagen pasan a primer plano y podemos detectar las características principales que le diferencian.

3.3.1 ¿Qué se ha hecho?

En primer lugar, como en las ocasiones anteriores ha sido necesario normalizar los datos, sin embargo, debido a las características propias de las redes convolucionales también ha sido necesario realizar un paso adicional, la configuración de las dimensiones.

La librería utilizada (*"Keras"*) espera recibir un input de 4 dimensiones, por lo que antes de trabajar con el modelo tendremos que reajustar nuestros datos a test requisito (fácilmente resuelto utilizando *"reshape"*).

Una vez completado el reajuste de datos podremos comenzar a construir el modelo, como en los casos anteriores, utilizando la función *"Sequence"*.

La construcción de una red neuronal mediante la librearía *"Keras"* se basa en dos fases claramente diferenciadas.

FASE I

Esta fase se basa en la adicción de las tuplas de capas *"Conv2D"* y *"MaxPooling2D"*.

- **Conv2D**. Introduce una capa convolucional en la que se requiere la especificación de las **unidades lógicas** a utilizar, la **función de activación** y el **tamaño del kernel**, que se refiere a la anchura y altura de la máscara del filtro.
- **MaxPooling2D**. Es la función de activación (se podría haber utilizado otras como AvergaPooling). En este caso se toma el máximo sobre los pasos, pero limitado a un **pool_size** para cada zancada.

FASE II

Una vez obtenido los datos filtrados por la red convolutiva se introducen en la red neuronal profunda, que en este caso se compone de dos capas ocultas (iguales que las expresadas en la implementación anterior con 32 y 16 unidades lógicas respectivamente) y una última capa de salida con función de activación *"softmax"*. Además, se han añadido desactivaciones de neuronas con el uso nuevamente de *"Dropout"*.

Una vez construido el modelo, se ha pasada a compilarlo, utilizando en esta ocasión a diferencia de anteriores implementaciones la función de pérdida *"categorical_crossentropy"* que ha resultado ser la que mejor rendimiento ha demostrado para nuestro modelo de red convolutiva, a diferencia del caso anterior que se utilizó la *"sparse_categorical_crossentropy"*. Como algoritmo de optimización una vez más se ha utilizado *"adam"*.

Por último, en esta ocasión se ha vuelto a utilizar la técnica de ***"EarlyStopping"*** como criterio de parada para las sucesivas épocas del entrenamiento. En concreto el criterio elegido ha sido utilizar la función de pérdida. En el momento en el que se produzcan cinco resultado iguales o peores finalizará el entrenamiento.

Como añadido, también a diferencia del caso anterior se ha utilizado en el entrenamiento el parámetro ***"batch_size"*** que representa o define el número de muestras que se propagarán a través de la red.

3.3.2 Resultados

Esta red ha mostrado la menor tasa de error respecto a todas las redes que se han implementado.

```
Epoch 80/150
422/422 [==============================] - 4s 10ms/step - loss: 0.0516 - accuracy: 0.9841 - val_loss: 0.0337 - val_accuracy: 0.9928
Epoch 00080: early stopping
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0096 - accuracy: 0.9977
0.009567183442413807 0.9977166652679443
```

Ilustración 13 - Evaluación RNC conjunto de entrenamiento

```
¿Desea utilizar datos de entrenamiento (e) o de test (t)? (e/t)
Elección: t
313/313 [==============================] - 1s 2ms/step - loss: 0.0322 - accuracy: 0.9924
0.03217673301696777 0.9923999905586243
```

Ilustración 14 - Evaluación RNC conjunto de prueba

Como puede verse, tras 80 etapas se ha conseguido un modelo con una tase de error del 0,76% y con un valor de pérdida del 0,95% sobre el conjunto de prueba.

Respecto al conjunto de prueba, el modelo muestra una exactitud del 99,77% (un error de 0.23%) con un valor de pérdida del 3.21%. Prácticamente como puede verse en los datos el error mostrado en inapreciable y se conforma la mejor red que hemos podido implementar en esta práctica.

3.4 Autoencoders

Esta última implementación se ha realizado con un objetivo más de curiosidad por lo que los resultados ni si quiera se acercan a los obtenidos en el apartado anterior. Antes de comenzar realizaremos una breve explicación de en qué consisten los autoencoders.

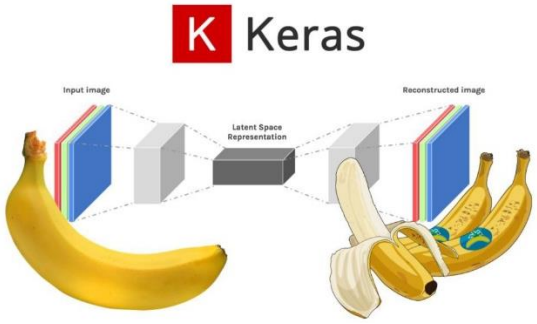

Ilustración 15 - Funcionamiento de los autoencoders

Los autoencoders son un tipo de redes neuronales cuyo objetivo es generar nuevos datos a partir de una entrada de valores que son comprimidos y luego reconstruidos en base a la información de salida adquirida. Esta red cuenta con dos partes claramente diferenciadas:

- **Encoders**. Es la parte de la red que se encarga de comprimir los datos de entrada en base.
- **Decoders**. Se encarga de reconstruir la entrada en base a la información recolectada previamente.

Al final del estudio de esta implementación comprobaremos si un entrenamiento previo utilizando autoencoders es capaz de conseguir extraer las características principales de una imagen y predecir los resultados utilizando una red neuronal simple, que como vimos en el primer caso tiene un rendimiento desastroso (apenas un 10% de exactitud).

3.4.1 ¿Qué se ha hecho?

Como siempre, el primer paso a realizar ha sido la normalización de todos los datos de entrada. A diferencia de ocasiones anteriores en este caso, al ser un aprendizaje **no supervisado** tendremos que utilizar también los datos de prueba, ya que así lo requiere *"Keras"* para evaluar la red.

Una vez se tengan listos los datos se puede comenzar con la construcción del autoencoder. Para ello se divide en dos fases, la construcción del *"encoder"* y la construcción del *"decoder"*.

Ambos casos son muy sencillos y similares a lo realizado en una red neuronal multicapa, de hecho, el esquema mostrado en la ilustración anterior es muy similar.

El *"encoder"* recibe el input de entrada, y trabaja con el añadiendo una sucesión de diversas capas ocultas (en nuestro caso con una densidad de unidades lógicas de 128, 64 y 32 respectivamente, todas ellas con función de activación *"relu"*).

Por otro lado, el *"decoder"* también queda conformado por una sucesión de capas ocultas conformado por el mismo número de unidades lógicas pero ordenadas de forma inversa (también con función de activación *"relu"*) y con una capa de salida con el mimo número de unidades lógicas que el tamaño de la dimensión de nuestra imagen, en nuestro caso 28x28.

El modelo creado se ha entrenado utilizando previamente una compilación con optimización *"Adam"* y una función de pérdida *"binary_crossentropy"*. Este modelo se ha entrenado utilizando un ***"bath_size"*** de 256 además de añadiendo el criterio de parada ***"EarlyStopping"*** visto en las ocasiones anteriores.

Una vez creado y entrenado el modelo se han utilizado las predicciones que realiza para usarlas en una red neuronal simple, y comparar los resultados obtenidos enfrentándola al uso de las imágenes sin tratar.

3.4.2 Resultados

A continuación, enfrentaremos los resultados de una *"Red Neuronal Simple"* en la que se usa imágenes no pretratadas e imágenes pretratadas.

```
313/313 [==============================] - 0s 2ms/step - loss: 0.2980 - accuracy: 0.1017
0.2979752719402313 0.10170000046491623
```

Ilustración 16 - RNS sin el uso de autoencoders

```
313/313 [==============================] - 0s 2ms/step - loss: 0.5818 - accuracy: 0.8734
0.5817819833755493 0.8733999729156494
```

Ilustración 17 - RNS con el uso de autoencoders

Como puede verse en la imagen anterior, la mejoría es claramente notable, ya que se ha pasado de un 10,17% de precisión a un 87,34% de precisión en el segundo caso.