Proyecto Fin de Carrera

Definición y aplicación de un entorno de programación para Deep Learning aplicado al procesado de imágenes.

Autor

Mikel Esparza Andrés

Director

Emiliano Bernués del Río

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Universidad de Zaragoza
2018

Definición y aplicación de un entorno de programación para DeepLearning aplicado al procesado de imágenes

Resumen

En este proyecto de fin de carrera se define un entorno de trabajo para el desarrollo de algoritmos de DeepLearning especializados en imagen.

Primero se estudian las bases que definen los algoritmos del Deep Learning profundizando mas en las redes neuronales. Después se realiza un trabajo de investigación de las diferentes librerías y lenguajes de programación óptimos para el desarrollo de este tipo de algoritmos. Con estos datos, definiremos un entorno de trabajo.

Para finalizar, se comprueba que este entorno de trabajo es funcional implementando distintas redes neuronales especializadas en la clasificación de imagen basándonos en técnicas del estado del arte actual.

Definition and application of a programming framework of Deep Learning for image processing

Abstract

In this Project we have defined an environment for developing Deep Learning algorithms specialized in image processing.

First we study the basic theory that defines Deep Learning algorithms looking more deeply in neural networks. Then, we do a research looking for the best frameworks, libraries and programming languages for developing the algorithms mentioned. With the data obtained we will define a working environment.

At the end of the project we will prove the utility of the environment defined by developing different architectures of neural networks specialized in image classification using state of art techniques.

Índice

1. Introducción

La reciente popularidad que esta adquiriendo la inteligencia artificial se basa en el éxito del Deep Learning. La disponibilidad de grandes cantidades de datos así como la mejora de las tarjetas gráficas han provocado que los sistemas de Deep Learning comiencen a superar a los humanos en tareas como el reconocimiento de caras o la clasificación de imágenes. Esto hace que empresas y universidades se encuentren diariamente trabajando en proyectos basados en este ámbito.

Gracias también a esta popularidad, grandes compañías han desarrollado librerías que facilitan la implentación de estos algoritmos para los distintos lenguajes de programación.

Lo cierto es que la mayoría de librerías se desarrollan en primer lugar en sistemas operativos Linux y luego se realiza la portabilidad a sistemas Windows. Sin embargo, lo habitual es que los usuarios finales utilicen el sistema operativo Windows.

A lo largo de este proyecto vamos a indagar y profundizar en aquellos lenguajes, entornos y librerías que nos facilitan realizar un buen trabajo en proyectos relacionados con el Deep Learning en Windows.

Después, definiremos un entorno de trabajo basado en nuestra investigación y comprobaremos su funcionalidad creando y entrenando una red neuronal especializada en la clasificación de imágenes, basándonos en técnicas del estado del arte actual.

Realizaremos este proyecto finalidad de que futuros alumnos que se quieran adentrar en el mundo del DeepLearning tengan definido un robusto entorno de trabajo en Windows optimizado para la creación de estos algoritmos.

2. Introducción al Deep Learning

2.1. ¿Qué es una red neuronal?

2.1.1. Introducción

Los humanos son increíbles máquinas de reconocimiento de patrones. Nuestros cerebros procesan "entradas" del mundo, las categorizan (eso es una araña, eso es helado), y luego generan una "salida" (huyen de la araña, prueban el helado). Y lo hacemos de forma automática y rápida, con poco o ningún esfuerzo. Lo mismo pasa cuando detectamos si alguien está enfadado con nosotros, o cuando involuntariamente pronunciamos el punto al final de la frase a medida que vamos leyendo.

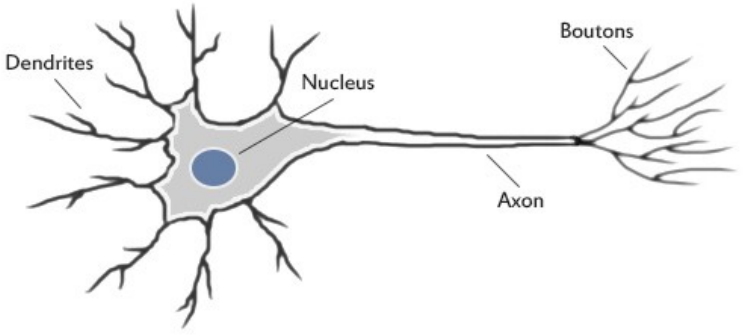


Figura 1: Neurona de cerebro

Nuestro cerebro utiliza una red extremadamente grande de neuronas interconectadas para procesar información y modelar el mundo que nos rodea. En definitiva, una neurona recoge las entradas de otras neuronas usando las dendritas. La neurona suma todas las entradas, y si el valor resultante es mayor que un umbral, se activa. La señal activa se envía a otras neuronas conectadas a través del axón.

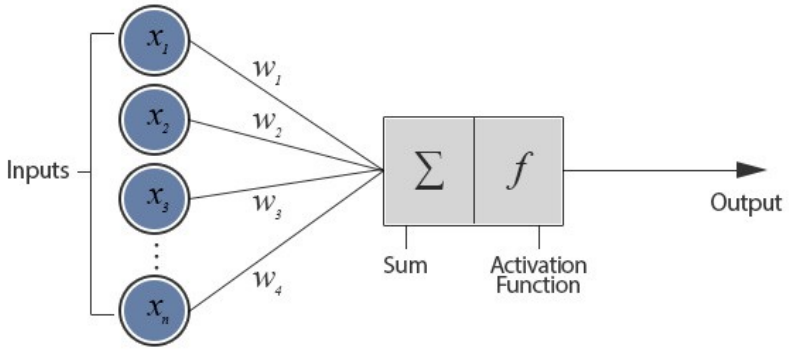


Figura 2: Red neuronal básica

La figura 2 representa una neurona de la red neuronal. Esta neurona esta conectada a la salida de otras n neuronas y por lo tanto, recibe n entradas (x1, x2, ... xn). Esta configuración

se llama Perceptron.

Las entradas (x1, x2, ... xn) y los pesos (w1, w2, ... Wn) son números reales y pueden ser positivos o negativos.

El Perceptrón se compone por un sumador y una función de activación.

Todas las entradas se ponderan individualmente, se suman y se pasan a la función de activación. Estas funciones de activación por lo general son funciones no lineales, que dado un valor de entrada, devuelven un valor a la salida en un rango entre 0 y 1 o entre -1 y 1. Básicamente, deciden si una neurona debe activarse o no, dependiendo de si la información que está recibiendo la neurona es relevante o debe ser ignorada.

ReLU (Rectified Liner Unit) es la función de activación mas usada hoy en día ya que ofrece muy buenos resultados en prácticamente cualquier ámbito del Deep Learning.
Ecuación: $\phi(z) = max(0, z)$

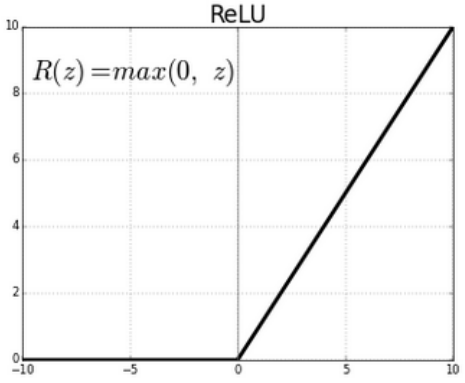


Figura 3: Función de activación ReLU

Con todo lo mencionado anteriormente, ya podríamos definir cual sería la salida de la neurona.
$Salida = \phi(X_1 * W_1 + X_2 * W_2 + ... + X_n * W_n + sesgo)$

El sesgo se utiliza para desplazar la función de activación hacia la derecha o hacia la izquierda. Es imprescindible en algunos casos para que la red pueda encontrar la solución.

Por ejemplo: Si consideramos una red neuronal simple en la que solo hay una neurona con un parámetro de entrada. La salida de la red sera: $salida = \phi(X * W)$ donde ϕ es la función de activación ReLU.
Si queremos que la red devuelva un 1 cuando $X = 0$, no existe un valor de W que cumpla la condición. Por ello existe el sesgo. Si la salida de la red es: $output = \phi(X * W + sesgo)$ Con $W = 1$ y $sesgo = 1$ podemos cumplir esa condición.

2.1.2. Entrenamiento

Una vez que se ha diseñado una red neuronal, es el momento de entrenarla. El entrenamiento consiste en garantizar que los pesos $w_{i,j}$ que corresponden con las entradas de cada neurona se configuren correctamente, de modo que toda la red nos proporcione un correcto valor a la salida.

Si intentamos enseñar a un niño a reconocer un autobús, lo único que hacemos es enseñarle ejemplos y decirle, ."Esto es un autobús. Esto no es un autobús"hasta que el niño aprende el concepto "Autobús". Una vez que el niño ha aprendido este concepto, podemos esperar que si ve nuevos objetos, sea capaz de reconocer si es o no un autobús.

Las redes neuronales son como niños. Imitan la forma en que nuestros cerebros resuelven el problema: tomando en cuenta las entradas, procesándolas y generando una salida. Al igual que nosotros, aprenden a reconocer patrones, entrenando con conjuntos de datos etiquetados.

El entrenamiento de la red se realiza con una técnica llamada "Back Propagation". No vamos a profundizar en la implementación matemática de este concepto. Para saber mas sobre el tema hay varios enlaces interesantes en la bibliografía. [1, 2].

Es importante saber que para el entrenamiento será necesario definir una función de error o función de coste. Esta función define cuánto difiere la salida de la red con respecto a la salida requerida. Existen diferentes funciones que definen el error y dependiendo cual elijamos, la red convergerá de un modo u de otro. Un ejemplo de esta función podría ser la función Mean-Squared Error (MSE) pero mas adelante revisaremos otras funciones de error que se utilizan hoy en día en la implementación de las redes neuronales.

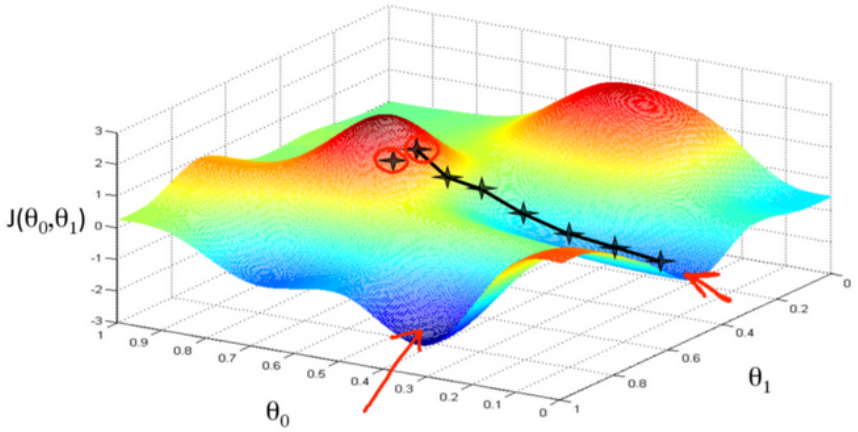


Figura 4: Función de error de un modelo de Deep Learning

Otro parámetro fundamental que influye en el entrenamiento es el Learning Rate. Determina qué tan rápido o lento nos moveremos hacia los pesos óptimos.

Para que la red converja hacia el mínimo de la función de error, debemos establecer el learning rate en un valor apropiado, que no sea ni muy bajo ni demasiado alto.

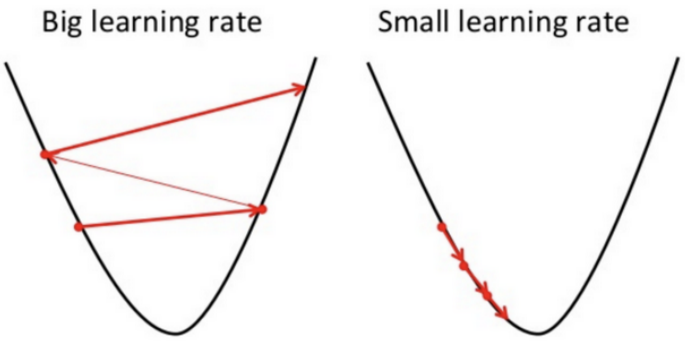


Figura 5: Learning rate

Como podemos ver en la imagen 5 si tomamos un valor de Learning Rate muy grande, es posible que no se alcance el mínimo global, ya que el algoritmo rebotará hacia adelante y hacia atrás en la función de error. Si se establece el valor de Learning Rate demasiado pequeño, la red se puede quedar estancada en un mínimo local de la función de error o si no, tardar demasiado tiempo en converger.

2.1.3. Capas ocultas

Hasta ahora, hemos explorado la arquitectura del Perceptrón, el modelo de red neuronal más simple.

Una vez entendido el Perceprón nos queda interconectar múltiples perceptrones entre sí para hacer redes neuronales mas complejas. En todos los diagramas de redes neuronales, la capa en el extremo izquierdo es la capa de entrada (es decir, los datos que alimenta), y la capa en el extremo derecho es la capa de salida (la predicción / respuesta de la red). Cualquier cantidad de capas entre estas dos se conoce como capas ocultas. Cuanta más cantidad de capas, más matizada puede ser la toma de decisiones.

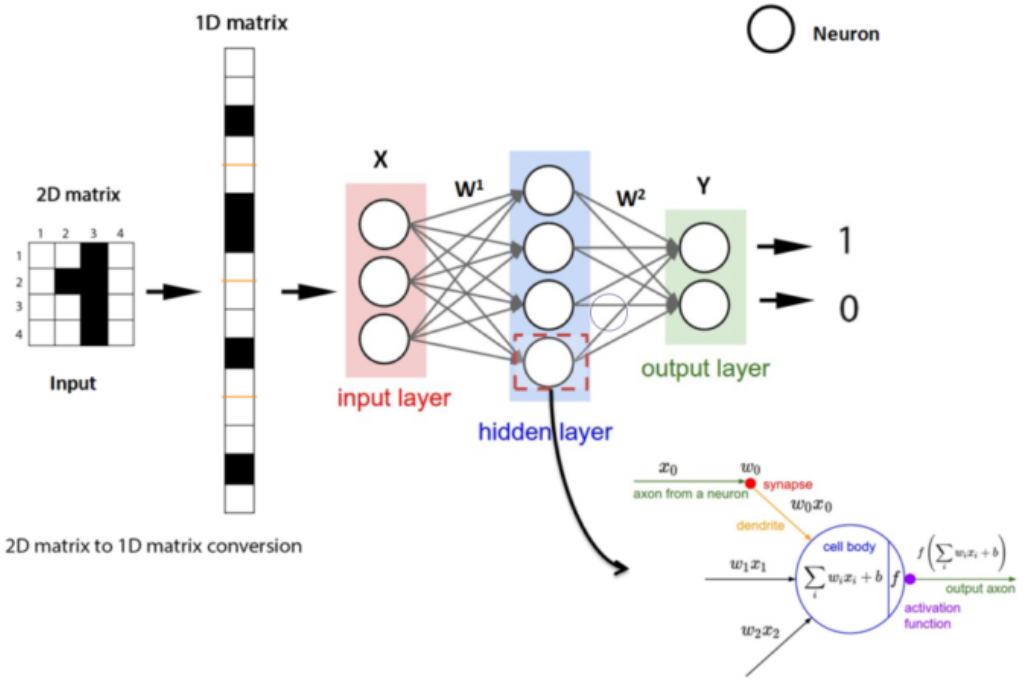


Figura 6: Deep FeedForward net

Existen múltiples configuraciones en las redes neuronales y todas ellas difieren en como cada capa esta interconectada con las anteriores y las posteriores. Por ejemplo, la red de la imagen 6 es la llamada red Deep Feedforward. Se caracteriza porque la salida de la función de la primera capa es la entrada a la segunda capa y esa salida es la entrada a la tercera capa y así sucesivamente ... La longitud de esa cadena da la profundidad del modelo, de esto es de donde proviene el término "profundo" en DeepLearning

2.1.4. Redes Neuronales Convolucionales (CNN)

Una de las arquitecturas de redes neuronales mas usadas para clasificación de imagen son las CNN.

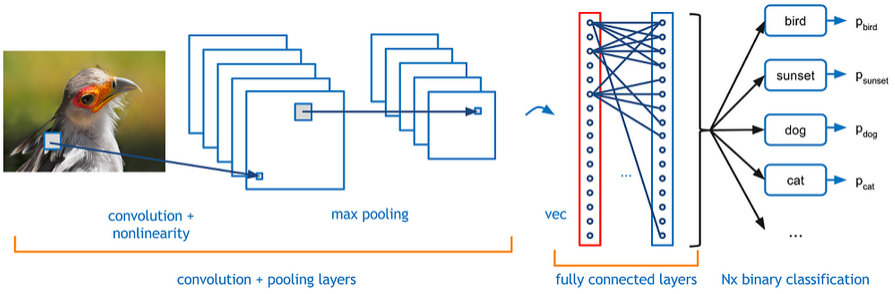


Figura 7: Red neuronal Convolucional

Las CNN, al igual que las redes neuronales, están formadas por neuronas con pesos y sesgos que se van modificando para reducir la función de coste. Cada neurona recibe varias entradas,

realiza una suma ponderada sobre ellas, la pasa a través de una función de activación y responde con una salida. Toda la red tiene una función de pérdida y todos los conceptos básicos de las redes neuronales aún se aplican en las CNN.

Entonces, ¿Cual es la diferencia entre las CNN y las redes neuronales convencionales?

A diferencia de las redes neuronales, donde la entrada es un vector, aquí la entrada es una imagen multicanal (3 canales en el caso de una imagen a color).

La idea principal de una capa convolucional, es que los pesos son filtros. Es decir, vectores multidimensionales que se aplican a la imagen de entrada realizando la operación de convolución.

Se toma un filtro y se desliza sobre la imagen completa. Según va avanzando por la imagen, se realiza el producto punto a punto entre el filtro y los trozos de la imagen de entrada. Luego se suma la salida de la multiplicación y se le añade el sesgo. Finalmente, se aplica la función de activación. Una capa de la red que realiza este proceso, es la llamada capa convolucional. Una red que tiene capas convolucionales se le llama red neuronal convolucional (CNN).

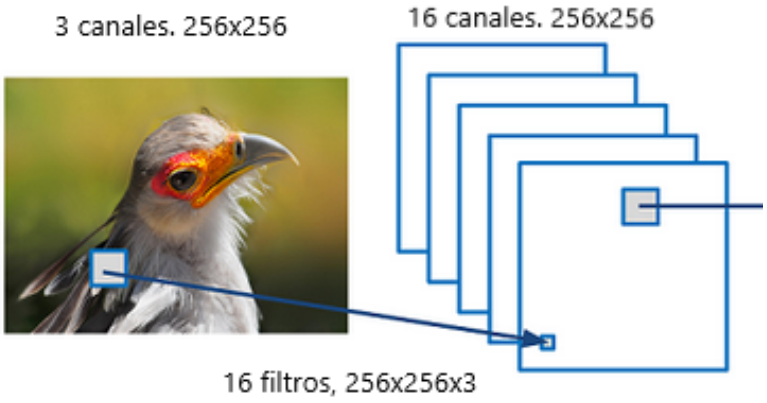


Figura 8: Convolución + bias + función de activación

Normalmente, una capa convolucional, tiene varios filtros y estos se aplican de manera independiente a la imagen de entrada. Por lo tanto, si tenemos una imagen de entrada de dimensiones 256x256x3 y le aplicamos 16 filtros de tamaño 5x5x3. A la salida obtendremos 16 imágenes de tamaño 256x256. A cada una de estas imágenes se le suele denominar "mapa de características".

Como vemos, la operación implica un aumento de dimensiones y por lo tanto un mayor número de operaciones a realizar. Una de las cosas que se suelen hacer para reducir las dimensiones de la red es aplicar una capa de "max pooling.[en]tre las capas convolucionales.

Esta capa actúa de manera independiente para cada mapa de características. A continuación expondremos un ejemplo del funcionamiento de esta operación.

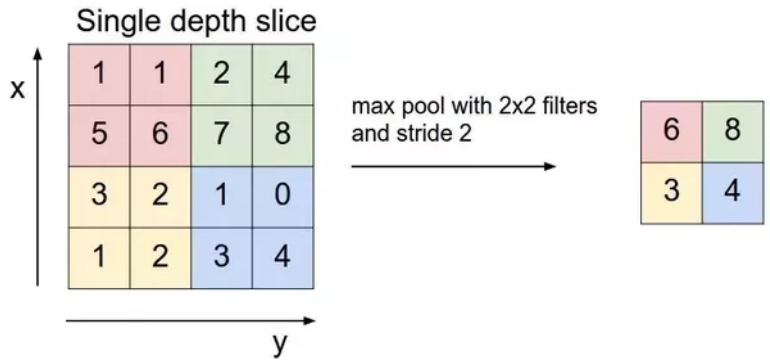


Figura 9: max pooling

Supongamos que tenemos una matriz 4x4 como la de la figura 9 que representa nuestra entrada inicial.
Digamos, también, que tenemos un filtro 2x2 que será deslizado sobre nuestra entrada aplicando un paso de 2 en el eje x y 2 en el eje y. Es decir, no se superpondrán las regiones.

Para cada una de las regiones representadas por el filtro, se tomará el máximo de esa región y se creará una nueva matriz de salida donde cada elemento es el máximo de una región en la entrada original.

Como vemos, esta operación implica una reducción de dimensiones y usarla entre las capas convolucionales de la red nos ayudará a reducir el número de operaciones a realizar.

Ahora que sabemos como se forman las capas convolucionales, ¿Qué pasa si juntamos varias CNN?

Vamos a verlo con un ejemplo. En la figura 10 podemos ver los filtros de una red neuronal que ya se encuentra entrenada. Esta red debe ser capaz de predecir que en la imagen existe un coche. Como vemos, los filtros de la primera capa se han ajustado para convertirse en manchas de piezas y bordes de colores. A medida que profundizamos en otras capas de convolución, los filtros se van aplicando a las salidas de las capas de convolución anteriores. Por lo tanto, van construyendo características mas complejas y mas representativas de la imagen original.

En la salida de la última capa convolucional, ya podemos ver como la red ha identificado una serie partes representativas de la imagen como son las ruedas del coche, las ventanas, etc. Estas características que han extraido las capas convolucionales son las que se pasan a una red FeedForward para que esta clasifique finalmente la imagen.

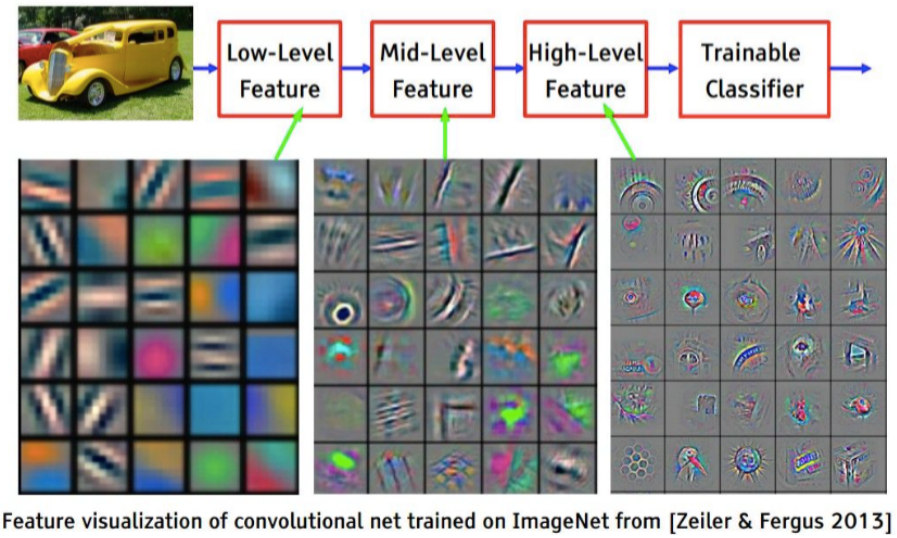


Figura 10: Filtros de una red convolucional

3. Definición de la plataforma de trabajo para Deep Learning

3.1. ¿Por qué Python?

Python [3] sin duda alguna es uno de los lenguajes de programación que está creciendo más rápidamente en la última década, gracias a su sintaxis, su programación es muy intuitiva además de fácilmente legible. Python es un lenguaje multiparadigma, es decir, soporta orientación a objetos, programación imperativa y en menor medida programación funcional. Si indagamos un poco por Internet, nos daremos cuenta de su popularidad rápidamente. El motor de búsqueda de trabajos Indeed [4], por ejemplo, nos permite ver una estadística del número de veces que aparece cada lenguaje de programación relacionado con un trabajo acerca del Deep learning, todo ello en un eje de tiempo. Este es el resultado:

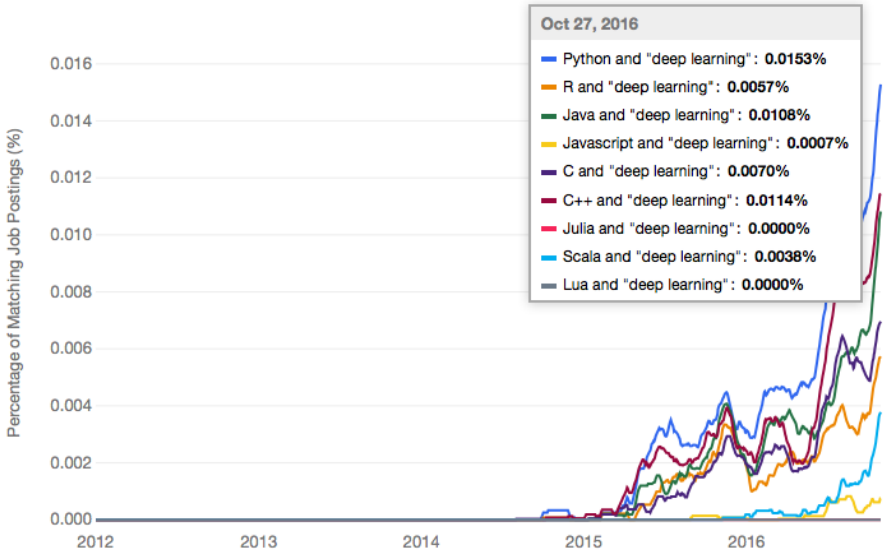


Figura 11: Número de trabajos de los distintos lenguajes de programación para Deep Learning

Como podemos observar, Python lidera en este ámbito seguido de C++ y Java.

Sin embargo, este no es el único ejemplo que podemos encontrar en Internet. También podemos hacer uso de las herramientas que Stackoverflow [5] nos proporciona para la visualización del número de visitas que se realizan por ciertos términos en su página web. En la siguiente gráfica 12 vemos una predicción para los próximos años sobre el número de visitas que Stackoverflow espera recibir sobre los principales lenguajes de programación.

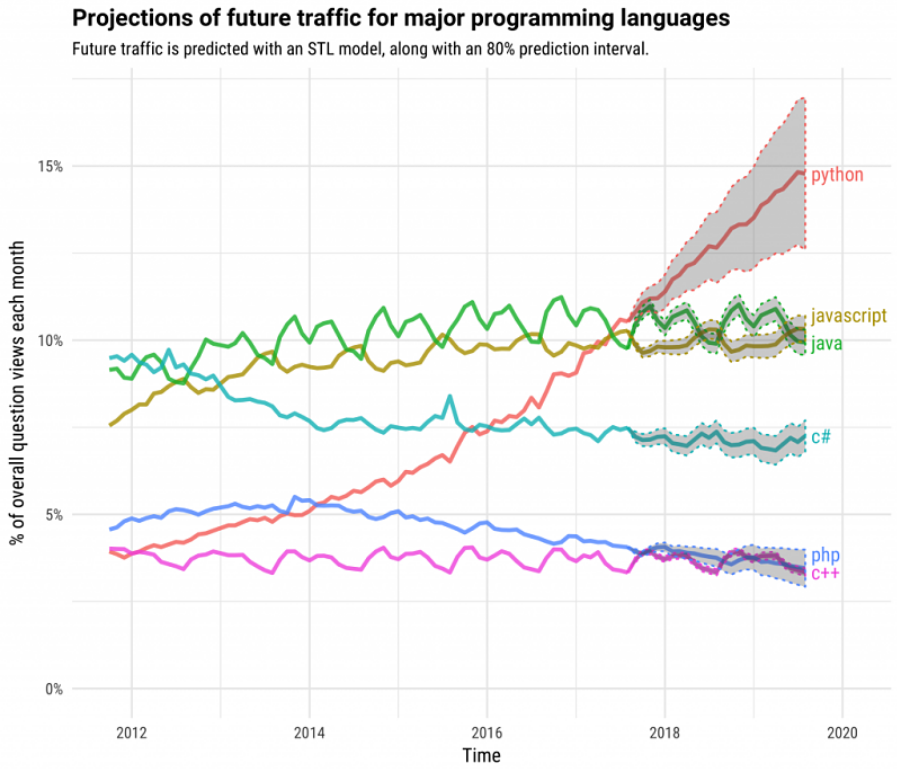


Figura 12: Número de visitas predichas para los próximos años

Otro índice muy relevante es el PYPL(PopularitY of Programming Language Index)[6] el cual está basado en la popularidad de los lenguajes de programación buscados en Google. Este sitúa a Python como el segundo con un crecimiento del 10 % en los últimos 5 años.

Worldwide, Jan 2018 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Java	22.9 %	-0.9 %
2		Python	21.0 %	+5.6 %
3		PHP	8.6 %	-1.9 %
4	↑	Javascript	8.4 %	+0.3 %
5	↓	C#	8.2 %	-0.6 %
6		C	6.7 %	-1.1 %
7	↑	R	4.0 %	+0.3 %
8	↓	Objective-C	3.9 %	-1.1 %
9		Swift	3.2 %	-0.3 %
10		Matlab	2.3 %	-0.7 %

Figura 13: PYP de 2018

Otros rankings como el del IEEE [7] a mediados de 2017, también sitúan a Python en los

primeros puestos.

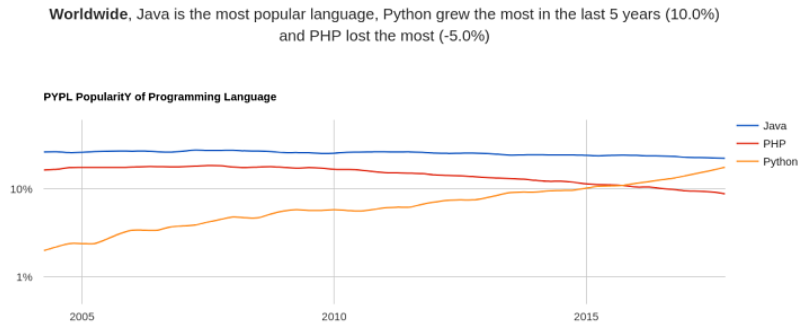


Figura 14: Estadísticas del IEEE

Si hablamos específicamente del área de datos, las comparativas son aún más demoledoras. Recientemente, Kaggle [8], la plataforma para Machine Learning y Data Science de Google, ha realizado un estudio recopilando información sobre las preferencias de los profesionales del sector. Un dato muy significativo es la respuesta a la pregunta: ¿Qué lenguaje de programación recomendaría a los nuevos científicos de datos aprender primero? Más del 63 % de los encuestados respondió Python.

Como vemos, Python es el lenguaje de programación que lidera hoy en día las encuestas. Además, gracias a los datos que hemos visto por parte de PYPL, intuimos que no se trata de una moda pasajera. Por todo esto, seleccionaremos Pyton como lenguaje de programación para el desarrollo de este proyecto.

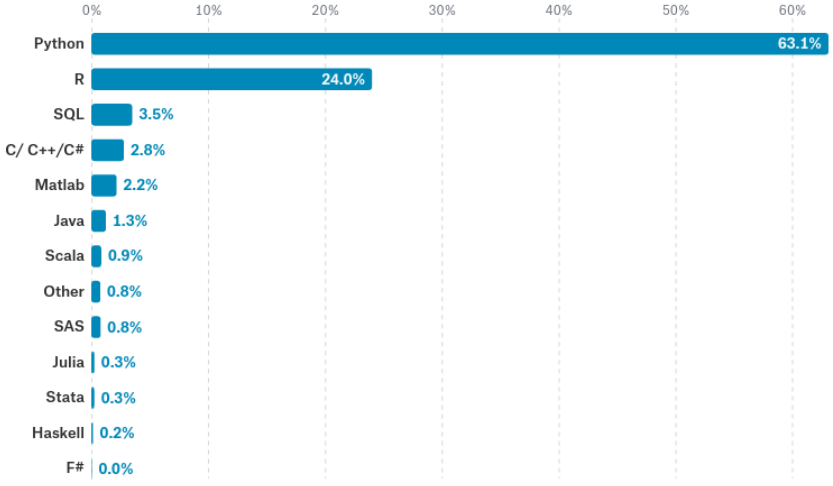


Figura 15: Encuesta de Kaggle por el lenguaje mas popular para empezar a estudiar

3.2. Versión de Python

Una vez seleccionado el lenguaje de programación a usar, nos toca decidir qué versión del mismo vamos a instalar.
En esta sección revisaremos el contexto que hay tras esta pregunta y daremos una explicación de por qué consideramos que la mejor elección se trata de Python 3.x.

El 3 de diciembre de 2008, Python lanzó la versión 3.0. Lo especial de esto fue que era una versión incompatible con versiones anteriores. Como resultado, para cualquiera que estuviera usando Python 2.x en ese momento, la migración de su proyecto a 3.x requirió grandes cambios. Esto no solo incluyó proyectos y aplicaciones individuales, sino también todas las bibliotecas que forman parte del ecosistema de Python.

El cambio fue visto como extremadamente controvertido, y muchos proyectos resistieron realizar la portabilidad, especialmente en la comunidad de Scientific Python. La biblioteca numérica principal NumPy [9] tardó dos años en lanzar su primera versión 3.x, después de lo cual otros proyectos comenzaron a lanzar versiones compatibles con 3.x en los años posteriores.

En 2012, muchas bibliotecas ya tenían soporte para 3.x, pero la mayoría aún se escribía en 2.x. Con el tiempo, se lanzaron herramientas que facilitaban el uso del código de transferencia, pero todavía existía una gran resistencia a dar el paso.

Originalmente, Python había programado la fecha de "fin de vida" para Python 2.x para 2015, pero en 2014 anunciaron que extenderían esto por 5 años hasta 2020, en parte para aliviar las preocupaciones de aquellos usuarios que aún no pueden migrar a Python 3.

Hoy en día, hay muy pocas bibliotecas que no son compatibles con Python 3. La página oficial de preparación Python 3, muestra que 348 de las 360 librerías más importantes para Python admiten 3.x.

Figura 16: Soporte de librerías para Python 3.x

Desde nuestra perspectiva, podemos decir que hemos tenido suerte en entrar en el mundo de Python 3.x. Como vemos prácticamente todas las principales librerías ya se encuentran actualizadas. Además, sabemos que la versión de Python 2.x tiene los días contados. Por todo lo comentado, en este proyecto trabajaremos siempre con la versión más actualizada hasta la fecha de Python 3.x la versión 3.6.

3.3. Librerías

En esta sección vamos a hacer un repaso de las principales librerías disponibles para Python en el ámbito de las redes neuronales. Tras ello, mencionaremos también aquellas librerías que no están relacionadas directamente con el Deep Learning pero que son imprescindibles en cualquier entorno científico para la visualización y representación de datos y resultados. Finalmente, repasaremos una herramienta proporcionada por NVIDIA [10] para la optimización de estas librerías.

1. **Caffe** [11]: La librería de Caffe fue desarrollada originalmente en UC Berkeley. Aunque se trate de una librería escrita en C ++ posee interfaces tanto de Python como de Matlab para trabajar con ella. La interfaz de Python no está muy bien documentada y requiere escribir código en c++ y en CUDA para modificar parámetros de las redes neuronales. Por ello, no se usa habitualmente en en el desarrollo de los proyectos. Sin embargo se usa mucho en la fase de producción. Una característica distintiva importante que Caffe posee, es que se pueden entrenar e implementar modelos sin escribir nada de código. Se pueden definir modelos con una serie de archivos de configuración y ponerlos a entrenar. Caffe también posee una gran cantidad de modelos preentrenados con la mayoría de arquitecturas del estado del arte actual optimizados para distintas tareas de clasificación de imagen.

2. **Theano** [12]: Theano es un proyecto de código abierto desarrollado por Montreal Institute for Learning Algorithms group at University of Montreal. Se trata de una librería de Python de bajo nivel usada para definir, optimizar y evaluar expresiones matemáticas que envuelven arrays multidimensionales. Todo esto lo hace incluyendo una buena integración con NumPy y un uso transparente de la GPU.
   Optimiza el uso de la CPU y de la GPU haciendo que el uso intensivo de datos sea aun más rápido.
   En noviembre de 2017, Youshua Bengio (jefe del laboratorio MILA) anunció que dejarían de dar soporte a esta librería. La razón es que la mayoría de las innovaciones que Theano introdujo a lo largo de los años han sido adoptadas y perfeccionadas por otras librerías.

3. **TensorFlow** [13]: TensorFlow es un framework de deepLearning desarrollado y mantenido por Google. Está escrito en C ++ / Python y proporciona APIs de Python, Java, Go y JavaScript. TensorFlow usa gráficos computacionales estáticos, es decir, primero se define la arquitectura de red en un grafo y luego lo ejecutamos tantas veces como queramos para entrenar la red. Recientemente, TensorFlow ha agregado agregado soporte para gráficos dinámicos.

   En la actualidad, TensorFlow ha reunido a la comunidad de aprendizaje profundo más grande a su alrededor, por lo que hay muchos videos, cursos en línea, tutoriales, etc. Ofrece soporte para ejecutar modelos en múltiples GPU e incluso puede dividir un único grafo computacional en varias máquinas para poder entrenar en clusters.

   Además de las funciones puramente computacionales, TensorFlow proporciona una gran extensión llamada TensorBoard. Esta extensión nos permite ver el gráfo computacional,

trazar métricas cuantitativas sobre la ejecución del modelo de entrenamiento y básicamente proporcionar todo tipo de información necesaria para depurar y ajustar las redes de manera más sencilla.

4. **Keras** [14]: Keras es una librería de redes neuronales de alto nivel escrita en Python por Francois Chollet, actualmente miembro del equipo de Google Brain. Funciona como interfaz para librerías de mas bajo nivel como TensorFlow, Microsoft Cognitive Toolkit, Theano o MXNet.

   Keras tiene una sintaxis muy simple e intuitiva por lo que es fácil de aprender y de usar. Posee ademas de una muy buena documentación, una comunidad muy grande y activa. La motivación principal de Keras es que deberías ser capaz de a partir de una idea, llegar a su implementación lo más rápido posible. Esto tiene como contra que en algunos casos no es demasiado flexible y por lo tanto no es recomendable su uso en modelos especialmente complicados.

5. **Pytorch** [15]: PyTorch fue lanzado por el grupo de investigación de inteligencia artificial de Facebook. Esta basado en Torch (librería que Facebook creó anteriormente para el lenguaje de programación "Lua"). Pytorch es el principal referente en el uso de grafos dinámicos (el grafo computacional se genera de manera secuencial a medida que se ejecuta el programa).

   Esta librería sigue el estilo de los comandos de Python y por lo tanto es fácil de utilizar para aquellos que tienen experiencia en este lenguaje. Además, el uso de la memoria en PyTorch es muy eficiente para cualquier red neuronal.

   Tiene una comunidad muy activa y posee muy buena documentación con muchos ejemplos y tutoriales, sin embargo, no se encuentra aún a la altura de TensorFlow en este ámbito.

3.3.1. Acelerador de procesado de NVIDIA

El módulo computacional principal en un ordenador es la Unidad Central de Procesamiento (más conocida como CPU). Está diseñado para hacer cálculos rápidamente en una pequeña cantidad de datos. Por ejemplo, multiplicar unos pocos números en una CPU es increíblemente rápido. Pero tiene problemas cuando se opera con una gran cantidad de datos. Por ejemplo, multiplicar matrices de decenas o cientos de miles de números. Por lo tanto, las GPU se desarrollaron para manejar muchos cómputos paralelos utilizando miles de núcleos. Además, tienen un gran ancho de banda de memoria para manejar los datos de estos cálculos. Esto los convierte en el hardware básico ideal para hacer Deep Learning.

NVIDIA conoce este uso para los grafos de computación y por ello nos aporta un SDK(Software Development Kit) con el que nos permite acelerar el procesado de todas las librerías mencionadas anteriormente.

3.3.2. Librerías adicionales

1. **Numpy** [9]: Es una librería irreemplazable para operaciones científicas en Python. El objeto fundamental de esta librería es el array. Con él podemos realizar operaciones matemáticas como sumas, restas o multiplicaciones. Estos vectores están muy bien optimizados para realizar estas operaciones mucho más rápido que usando simples listas en Python.

2. **Pandas** [16]: Esta librería es realmente útil para el uso de grandes cantidades de datos, algo que implícitamente encontramos en el Deep learning. Nos ayuda en el uso de estructuras de datos así como con herramientas para el análisis de los mismos.

3. **Matplotlib** [17]: Matplotlib es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python. Será esencial para el tratamiento de los datos que necesitamos para entrenar nuestras redes neuronales.

4. **Scipy** [18]: Se trata de una librería complementaria a NumPy. Esta librería nos proporciona herramientas para realizar operaciones estadísticas. También contiene herramientas de procesado de señal en las que se incluye la convolución y la transformada de Fourier.

3.3.3. Librerías finalmente escogidas

Tras investigar por Internet cuales son las librerías más conocidas y populares, por su versatilidad y funcionalidad para proyectos en Deep Learning, podemos decir lo siguiente.

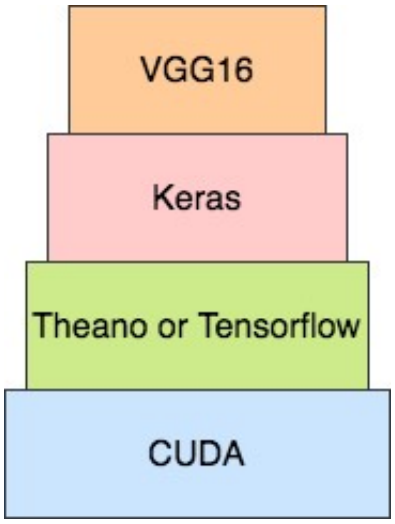


Figura 17: Niveles de abstracción DeepLearning

En la figura 17 podemos ver una idea de la representación de las diferentes capas de abstracción que corresponden con los diferentes módulos de DeepLearning. El nivel inferior, el llamado Cuda, corresponde directamente con el elemento computacional que se encargará de hacer los cálculos. Preferiblemente será la GPU pero también puede ser la CPU.
Por encima encontramos el nivel básico de programación, en él se definen las operaciones más básicas para el deep Learning como son las operaciones con matrices. Este nivel ya es completamente funcional y podríamos crear sin problemas redes neuronales en él. Se trata de un nivel

muy bajo de programación por lo que la creación de redes neuronales será algo mas tediosa teniendo que definir prácticamente todas las operaciones necesarias. Sin embargo, y como consecuencia de lo anterior, se trata de un nivel de programación muy flexible y versátil. Una vez aprendida las sintaxis y la forma de trabajo, nos permitirá configurar las redes neuronales a nuestro gusto sin tener prácticamente limitaciones.
Para aquellos que se inician en el Deep Learning, es recomendable empezar en el nivel superior de abstracción, el que corresponde con Keras. Su sintaxis es mucho mas sencilla de aprender y permite crear una gran variedad de redes neuronales con muy pocas líneas de código. Como ya hemos mencionado, tiene la contra de que no es tan flexible en algunos casos como nos gustaría.
Por encima de Keras encontramos finalmente la arquitectura de la red. Es decir, las diferentes librerías de los niveles inferiores, ya sea a nivel de TensorFlow o a nivel de Keras, nos proporcionan herramientas útiles para la creación de redes neuronales. En este caso, la red VGG [22] se trata de una red que se caracteriza por la utilización de varias capas convolucionales con filtros de tamaño 3x3 que funciona especialmente bien en imágenes.

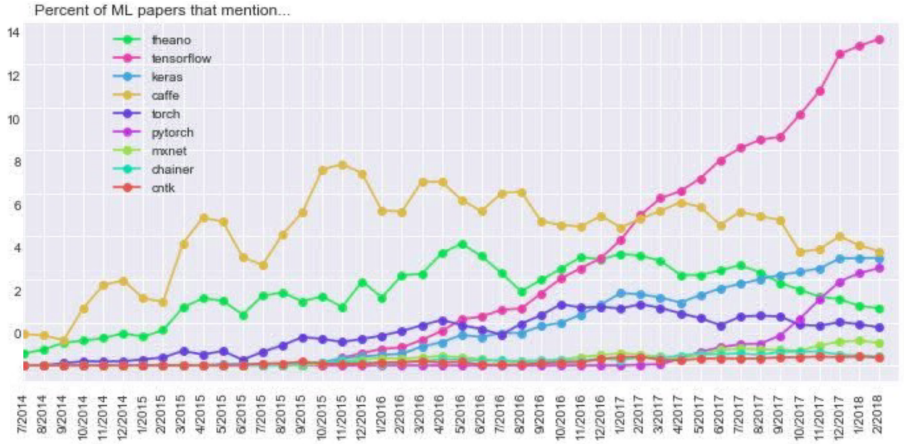


Figura 18: Porcentaje de artículos publicados por cada librería de DeepLearning

En la figura 18 se muestra el porcentaje de artículos publicados que mencionan las distintas librerías de DeepLearning. Como vemos, TensorFlow es el que lidera en esta gráfica.

Debido a su gran popularidad, su extensa documentación, su continuo desarrollo por parte de Google y su flexibilidad a la hora de programar cualquier tipo de red, escogeremos TensorFlow para la creación de algoritmos orientados al DeepLearning.

3.4. Instalación y configuración del entorno y lenguaje de programación

Para la instalación y configuración del entorno utilizaremos Anaconda [19]. Anaconda es una Suite de código abierto que abarca una serie de aplicaciones, librerías y conceptos diseñados para el desarrollo de aplicaciones científicas con Python. En líneas generales Anaconda Distribution es una distribución de Python que funciona como un gestor de entornos y un gestor de librerías. Todo esto con una potente interfaz que nos evita tener que manejarnos mediante consola de comandos. Además, contiene la instalación de la última versión de Python. Gracias a esta Suite, no nos tendremos que preocupar de la instalación de las principales librerías ya que vienen preinstaladas.

La gestión de entornos es una característica importante de Anaconda. Nos permite desarrollar diferentes aplicaciones en paralelo variando las versiones de Python y sus librerías.

Con la Suite de Anaconda viene incluido Spyder [20]. Se trata de un IDE(Integrated development environment) muy ligero desarrollado para la programación científica con Python. Posee funciones avanzadas de edición, pruebas interactivas y depuración. Este será el IDE que utilizaremos en el desarrollo de nuestras redes neuronales.

El proceso de Instalación y configuración de Anaconda en Windows está descrito en los Anexos.

4. Deep Learning en clasificación de imagen

Antes de pasar a la implementación de las redes neuronales en el entorno de trabajo que hemos definido, vamos a hacer un repaso de los algoritmos de Deep Learning para clasificación de imagen a lo largo de su historia. En esta sección analizaremos estos algoritmos y veremos como fueron mejorando año tras año hasta llegar al estado del arte actual.

4.1. Arquitecturas y evolución

Gracias a la aparición de grandes bases de datos categorizadas y a la mejora en las tarjetas gráficas, las grandes compañías y universidades se pusieron a competir entre ellas para ver quien podía conseguir los mejores resultados. Esto incentivó el desarrollo de nuevos y mejorados algoritmos de Deep Learning.

Usaron como dataset de referencia ImageNet y así poder medir sus progresos. Para 2012, ImageNet tenía casi 1.3 millones de imágenes de entrenamiento con 1000 categorías diferentes. Se trataba de un dataset complicado debido a la gran variedad de clases que pretendía clasificar.

4.1.1. AlexNet

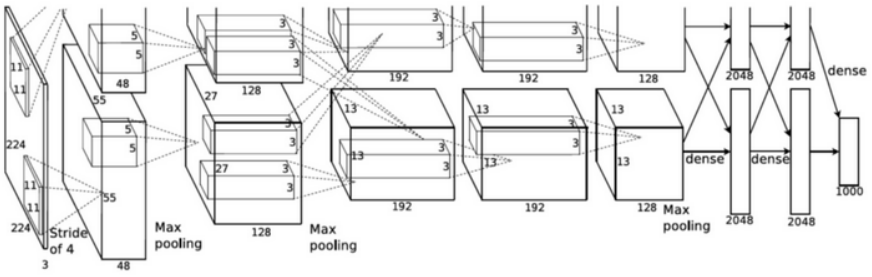


Figura 19: Arquitectura Alexnet

En 2012, la Universidad de Toronto publicó un artículo llamado "ImageNet Classification with Deep Convolutional Network" [21]. Este artículo se convertiría en uno de los artículos más influyentes en el campo, después de lograr una reducción de casi el 50 % en la tasa de error en el desafío de ImageNet, el cual fue un progreso sin precedentes en ese momento.

El artículo propuso utilizar una red neuronal convolucional (CNN) para la tarea de clasificación de imágenes. Era relativamente simple comparado con aquellos que se usan hoy en día. Las principales contribuciones que vinieron de este artículo fueron:

1. Fue el primero en implementar redes convolucionales de manera exitosa para la clasificación de imágenes a gran escala. Esto fue posible debido a la gran cantidad de datos etiquetados de ImageNet, así como el entrenamiento del modelo uilizando cálculos paralelos en dos GPU.

2. Utilizaron ReLU para las funciones no lineales de activación , encontrando que funcionaban mejor y que disminuían el tiempo de entrenamiento relativo a la función tanh. La no linealidad ReLU ahora tiende a ser la función de activación predeterminada para redes neuronales.

3. Utilizaron técnicas de aumento de datos que consistían en rotaciones de imágenes, reflexiones horizontales y la supresión de la media. Hoy en día, este tipo de técnicas se usan para la mayoria de tareas relacionadas con imagen.

4. Utilizaron capas "dropout" para combatir el problema del sobreentrenamiento. El sobreentrenamiento se produce cuando la tasa de error del set de entrenamiento es menor que la tasa de error del subset de validación. Es decir, la red esta aprendiendo las características propias del set de entrenamiento y no esta obteniendo conceptos generales de las imágenes. Este problema es común cuando tenemos redes neuronales con una enorme cantidad de neuronas. En estos casos, las neuronas desarrollan una codependencia entre ellas. Una de las soluciones que se aplican a este problema son las capas de dropout. Estas capas ponen a 0 de manera aleatoria las salidas de algunas neuronas. De esta manera, eliminamos esa codependencia y forzamos a la neurona que no hemos ignorado a obtener mas informacion.

5. Su estilo propuesto de tener sucesivas capas convolucionales y finalizar con capas "fully connecte" sigue siendo la base de muchas redes de hoy en día.

Básicamente, AlexNet estableció el estándar, y la base de la utilización de las redes convolucionales para la clasificación de imagen.

4.1.2. VGGNet

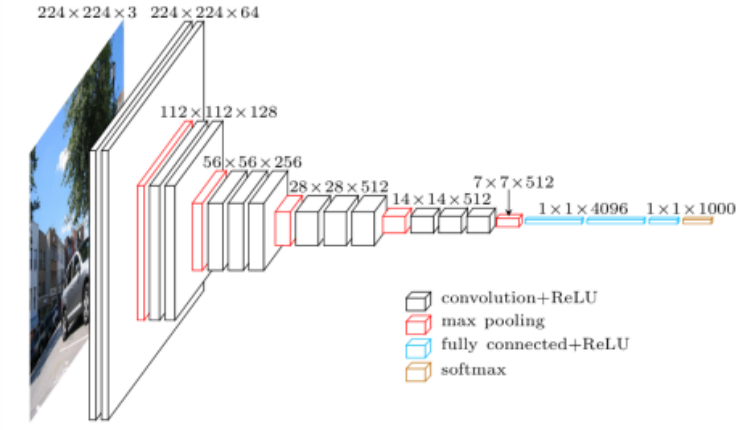


Figura 20: Arquitectura VGG

El artículo de VGGNet "Very Deep Convolutional Neural Networks for Large-Scale Image Recognition" [22] salió en 2014, ampliando aún más las ideas de utilizar redes convolucionales con la función de activación ReLU. Este artículo fue publicado por la universidad de Oxford y

su idea principal era que solo bastaba con añadir muchas capas convolucionales con filtros de 3x3 y ReLU, la red haría el resto.

Las principales contribuciones de VGGNets son:

1. El uso de filtros de solo 3x3 en lugar del 11x11 utilizado en AlextNet. Demostraron que la sucesión de varios filtros de 3x3 es equivalente al uso de un filtro de 7x7. El primer beneficio de los filtros más pequeños es una disminución en el número de parámetros. El segundo es poder utilizar una función ReLU entre cada convolución. Esto aumenta la no linealidad en la red, lo que hace que la función de decisión sea más discriminatoria.

2. Uno de los motivos por los que VGGNet mejoró sobre AlexNet es que debido al uso de las capas de Max Pooling, las dimensiones de las imágenes de entrada disminuyen. Sin embargo, el número de canales aumenta tras las capas convolucionales. Para que la red pueda converger al mínimo de la función de error, las neuronas deben obtener características más discriminatorias y así es como obtienen una clasificación mas precisa.

4.1.3. GoogLeNet y el módulo Inception

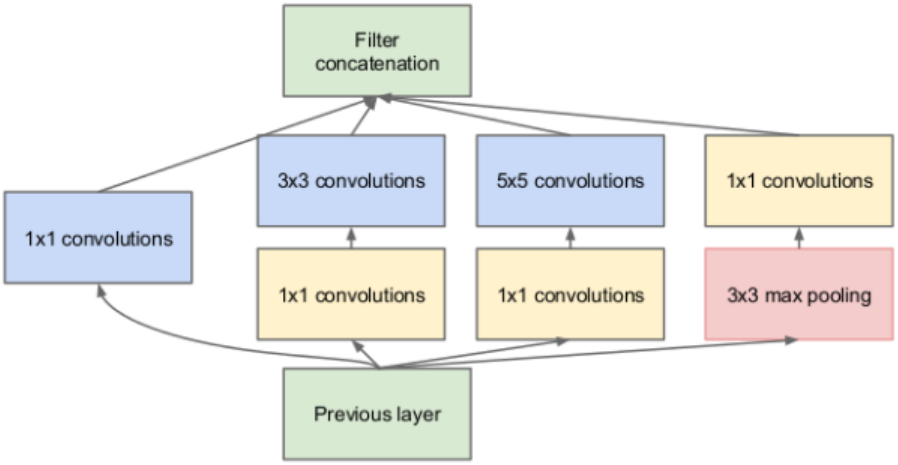


Figura 21: Módulo inception

La arquitectura de GoogLeNet fue la primera en abordar el problema de los altos recursos que consumía entrenar una red convolucional. Esta aqruitectura se presentó con el artículo "Going Deeper with Convolutions" [23]. A medida que la redes neuronales se iban haciendo más y más profundas, se llegó a un punto en el que se estaba consumiendo mucha memoria. Por otro lado, la elección del tamaño del filtro era complicada. Se habían propuesto diferentes tamaños desde 1x1 a 11x11; ¿cómo decidimos qué tamaño elegir? El módulo de inception y GoogLeNet abordan todos estos problemas con las siguientes contribuciones:

1. Mediante el uso de convoluciones de 1x1 antes de cada 3x3 y 5x5 se reduce la cantidad de mapas de características que pasan a través de cada capa, lo que reduce los cálculos y el consumo de memoria.

2. El módulo de inception tiene convoluciones 1x1, 3x3 y 5x5 todas en paralelo. La idea detrás de esto era dejar que la red decidiera, a través del entrenamiento, qué información se aprendería y usaría.

3. GoogLeNet fue una de las primeras arquitecturas que introdujo la idea de que las capas de CNN no siempre tenían que apilarse de manera secuencial. Los autores del documento mostraron que se puede mejorar el rendimiento de la red no solo aumentando su profundidad si no que también su anchura.

4.1.4. ResNet

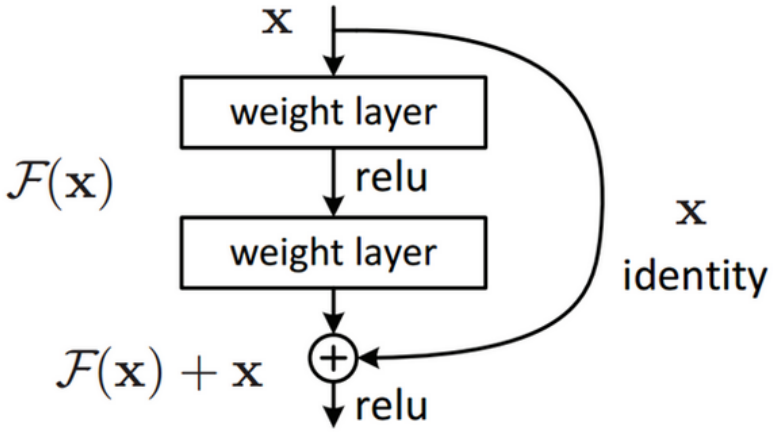


Figura 22: Bloque residual

Desde su publicación inicial en 2015 con el artículo "Deep Residual Learning for Image Recognition" [24] por parte de la universidad de Cornell, ResNets (Residual Neural Network) han implementado importantes mejoras en la precisión de muchas tareas de visión artificial. La arquitectura ResNet fue la primera en mejorar el rendimiento de un humano en ImageNet, y su principal contribución de aprendizaje residual a menudo se utiliza por defecto en muchas redes de última generación en la actualidad:

1. Demostró que apilar de manera descontrolada capas para hacer que la red sea muy profunda no siempre ayuda e incluso puede empeorar las cosas.

2. Para abordar el problema anterior, introducen el concepto de aprendizaje residual con omisiones (skip connection). La idea es pasar a la funcion de activación la salida de una neurona mas la suma de la entrada de la misma neurona sin ser modificada. De esta forma, las capas profundas tienen acceso directo a las características de las capas anteriores. Esto permite que la información de las funciones de activación se propague más fácilmente a través de la red.

3. Crearon la primera red ültra profunda", donde ya usaban mas de 100 capas convolucionales.

4.1.5. DenseNet

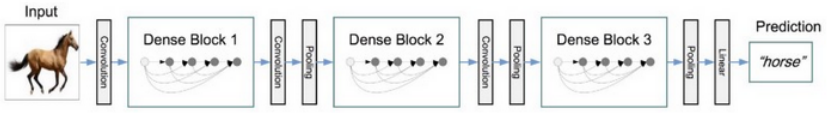


Figura 23: DenseNet

El concepto introducido por las ResNets de "skip connections" es llevado al extremo con la introducción de DenseNets del artículo "Densely Connected Convolutional Networks" [25].

1. DenseNets aplica el mismo concepto de "skip connections" pero con una pequeña diferencia. Conecta la salida de la neurona con la entrada mediante una concatenación en lugar de la adición utilizada en ResNets, de este modo, las características en bruto se pasan a través de las capas directamente. Esto permite que cada capa use todos los mapas de características de todas las capas precedentes como entradas, y sus propios mapas de características se utilizan como entradas en todas las capas posteriores.

2. Muestra que tiene un mejor rendimiento que ResNets. DenseNets ayuda a aliviar el problema del desvanecimiento gradiente, al fortalecer la propagación de características. Además, al fomentar la reutilización de características reduce sustancialmente la cantidad de parámetros.

Esas son las principales arquitecturas que han formado la columna vertebral del progreso en la clasificación de imágenes en los últimos años.

5. Implementación

Una vez estudiadas las bases del Deep Learning y definido un entorno de trabajo óptimo para el desarrollo de este tipo algoritmos, vamos a pasar a la implementación. En esta sección revisaremos el hardware del que disponemos para trabajar. Tras ello, escogeremos una base de datos para entrenar nuestras redes. Finalmente explicaremos las redes que han sido implementadas a lo largo del desarrollo de este proyecto detallando cada uno de los parámetros que las componen.

5.1. Hardware utilizado

Como hemos visto en la sección 3.3.1 las gráficas cumplen un gran papel en la ejecución de algoritmos relacionados con el Deep Learning debido intensivo uso de matrices. Por ello,Las grandes compañías utilizan varios ordenadores en paralelo conmúltiples GPUs para la creación y entrenamiento de las redes. Sin embargo nosotros no disponemos de dicho hardware.

El desarrollo de este proyecto se realizará sobre ordenador con una gráfica NVIDIA GeForce GTX 1060 de 6Gb de memoria RAM de video. Por otro lado el PC utilizado tiene 16Gb de memoria RAM ddr4 y una CPU Intel Core i7-6700k que posee 8 núcleos los cuales funcionan a 4GHz. Como vemos, no se trata de un Hardware muy potente para la implementación de redes neuronales.

Por lo tanto plantearemos la nuestras redes desde una perspectiva mas conservadora. No crearemos redes demasiado profundas puesto que tardarían semanas en ser entrenadas. Vamos a buscar arquitecturas y utilizar conceptos que nos ofrezcan buenos resultados sin una gran demanda de recursos.

5.2. Base de datos

El DeepLearning ha dominado absolutamente en la informática en los últimos años. Esto es debido a la creación de grandes datasets de fotos clasificadas por categorías. Gracias a estos datasets, las redes neuronales han comenzado a sacar muy buenos resultados. Uno de los datasets mas importantes es “Places” [26].

El conjunto de datos de Places está diseñado siguiendo los principios de la cognición visual humana. Se ha creado por el Instituto de Tecnología de Massachusetts (MIT) con el motivo de entrenar sistemas artificiales para tareas de comprensión visual de alto nivel, como el contexto de la escena, el reconocimiento de objetos y la predicción de acciones y eventos.

En total, Places contiene más de 10 millones de imágenes que comprenden más de 400 categorías de escenas únicas. El conjunto de datos presenta de 5.000 a 30.000 imágenes de entrenamiento por clase, consistentes con las frecuencias de ocurrencia en el mundo real.

Places esta dividido en 4 subsets de datos.

1. Places365-Estándar tiene 1.803.460 imágenes de entrenamiento con el número de imágenes

por clase que varía desde 3.068 a 5.000. El conjunto de validación tiene 50 imágenes por clase y el conjunto de prueba tiene 900 imágenes por clase.

2. Places365-Desafío contiene las mismas categorías que Places365-Estándar, pero el conjunto de entrenamiento es significativamente mayor con un total de 8 millones de imágenes de entrenamiento.
   El conjunto de validación y el conjunto de prueba son los mismos que Places365-Estándar. Este subconjunto fue lanzado para el Places Challenge 2016 celebrada conjuntamente con el Conferencia europea sobre la visión artificial (ECCV) 2016, como parte del Desafío ILSVRC.

3. Places205, tiene 2.5 millones imágenes de 205 categorías de escenas. El número de imágenes por clase varía de 5.000 a 15.000. El conjunto de entrenamiento tiene 2.448.873 imágenes en total, con 100 imágenes por categoría para el conjunto de validación y 200 imágenes por categoría para el conjunto de prueba.

4. Places88 contiene las 88 categorías que son comunes entre los datasets: ImageNet, SUN y Places205. Se utiliza principalmente para obtener el rendimiento entre los diferentes datasets.

Debido al hardware del que se dispone para la realización del proyecto, no se puede optar por escoger una base de datos excesivamente grande ya que el entrenamiento sería demasiado costoso. Por ello, para la implementación de la red, se escogerán de manera aleatoria un total de 50 clases de places365-Estándar.

De esta forma, nuestro dataset se compone de un total de 250.000 imágenes de entrenamiento (5.000 imágenes por clase) y 5.000 imágenes para validación (100 imágenes por clase).

El uso de este dataset nos presenta dos grandes desafíos:

1. **Gran variedad de clases**: Debido a la gran variedad de clases que este dataset posee, existen casos en los que la diferencia entre las clases es mínima.

   Queremos maximizar la variabilidad entre clases es decir, si tenemos dos imágenes, que visualmente son parecidas, pero pertenecen a diferentes categorías, nuestro modelo debe ser capaz de separarlas lo máximo posible y así poder de diferenciarlas.

   En la figura 24 podemos apreciar dos ejemplo de imágenes de nuestra base de datos que pertenecen a distintas categorías pero que son visualmente muy similares.

(a) Isla

(b) Faro

(c) campo de fútbol

(d) campo de golf

Figura 24: Variedad de clases

2. **Variabilidad dentro de la misma clase**: Este es otro desafío de nuestro dataset, los objetos de la misma clase pueden ser visualmente muy diferentes. Veamos las imágenes a continuación 25. Las dos primeras imágenes, corresponden con la categoría "ball˙pit" y las dos siguientes son ambas de la categoría "classroom". Sin embargo, cada par de imágenes se ve muy diferente. Como humanos, podemos ver que en un caso la piscina de bolas se ve desde lejos y en el otro se le ha hecho zoom; del mismo modo, sabemos diferenciar entre las sillas y mesas de una clase así como el profesor que se encuentra escribiendo en la pizarra de la clase. Esto se llama variabilidad intraclase. Queremos minimizar esta variabilidad y así, dos imágenes de la misma clase se verán muy similares para nuestro modelo y no tendrá problemas en clasificarlas en la misma categoría.

(a) ball˙pit (b) ball˙pit

(c) classRoom (d) classRoom

Figura 25: Variedad dentro de la misma clase

5.3. Implementación de las Redes neuronales

A lo largo de esta memoria hemos estudiado los principios del Deep Learning y de las redes neuronales. Después, hemos profundizado en los diferentes lenguajes de programación y librerías mas útiles para su implementación. Hemos realizado un repaso de las diferentes arquitecturas de redes neuronales y como ha sido su evolución a lo largo del tiempo. También, hemos presentado el equipo del que disponemos para trabajar y en base a ello, hemos seleccionado una base de datos. Hemos realizado un pequeño análisis de la base de datos y hemos analizado los posibles problemas que nos vamos a encontrar

Ahora vamos a pasar a la implementación. En esta sección se explicarán las diferentes redes neuronales implementadas comentando su arquitectura y los parámetros necesarios para su correcto funcionamiento.

A la hora de implementar redes neuronales, existen una serie de características y parámetros que todas ellas comparten. A continuación se darán a conocer de manera mas detallada cuales son estas características.

5.3.1. Parámetros generales

Parámetros de entrada a la red

De la introducción al deep Learning, recordamos que la redes para aprender a clasificar las imágenes necesitan: Una imagen y la categoria a la que pertenece dicha imagen. Sin embargo, a las redes no podemos enseñarles como a los niños señalándoles un autobús y diciéndoles la palabra "Autobús". Las redes operan con números y por lo tanto necesitan entradas y salidas numéricas.

Las imágenes ya son matrices de números pero ¿como convertimos las diferentes categorías a números? Aquí entra en juego la codificación "One Hot"

La codificación "One Hot" transforma las categorías en vectores con ceros y unos. La idea es generar un vector de tantas posiciones como categorías tienen los datos. Este vector se rellena con 0 en todas las posiciones excepto en la posición que corresponde con la categoría que queremos clasificar donde pondremos un 1. Por ejemplo: Si tenemos 3 categorías que son: Rojo, Verde y Azul. La codificación "One Hot" será la siguiente:

Rojo	Verde	Azul
1	0	0
0	1	0
0	0	1

Utilizaremos esta codificación para representar cada una de las 50 categorías en las que se pueden clasificar nuestras imágenes de la base de datos.

Batch size

Como comentamos en las bases del DeepLearning, la red debe ir aprendiendo las características de las imágenes para poder luego clasificarlas a la categoría que pertenecen. Este proceso se realiza durante el entrenamiento de la red. En cada iteración le pasamos un subconjunto de imágenes del dataset que representa lo mejor posible al conjunto global. Con estas imágenes, la red obtiene una predicción y calcula el error que ha tenido en el proceso. Gracias a este error, la red es capaz de cambiar sus pesos para obtener mejores predicciones en el futuro. Al número de iteraciones que necesita la red para que revise el dataset por completo se le denomina Epoch.

El conjunto de imágenes que se utilizan en cada iteración del entrenamiento se denomina "mini batch". El número que representa cuantas imágenes componen un "mini batch" se corresponde con el "Batch size".

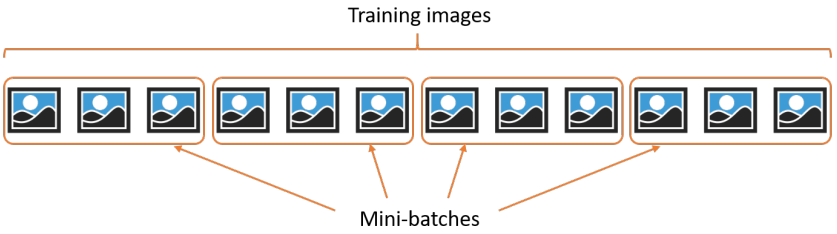


Figura 26: Mini Batches

Como es lógico pensar, cuantas mas muestras le pasemos en cada iteración a la red, mejor será su calculo del error de predicción y por lo tanto modificará de manera mas precisa sus pesos. Sin embargo, un número alto de Batch Size, implica una gran cantidad de operaciones de la red en cada iteración y por lo tanto, en algunos casos nos puede suponer un problema de insuficiencia de memoria RAM. Además, el realizar el cálculo del error de predicción basándonos en muchas imágenes provoca que este error varíe ligeramente entre iteraciones. Esta es una situación que queremos evitar, si la red siempre avanza en la misma dirección, no explorará nuevos caminos y por lo tanto es posible que jamás converja al mínimo global de la función de error.

Por otro lado, un valor extremadamente pequeño de Batch Size hace que el subconjunto de datos no sea una representación global del dataset y por lo tanto, el error difiera de una iteración a otra provocando que la red diverja.

Es importante recalcar que el subconjunto de datos que le pasamos en cada iteración, debe ser una representación del conjunto global del dataset. Si en cada iteración le pasamos todas las imágenes de una misma categoría, la red convergerá para que los pesos se adapten únicamente a dicha categoría, y como consecuencia, empeorará en la predicción del resto de imágenes.

Para evitar esta situación, en nuestra implementación, al inicio de cada Epoch, escogeremos el conjunto de imágenes que compone el dataset, lo desordenaremos de manera aleatoria y realizaremos agrupaciones con un Batch Size de 32 imágenes.

Learning Rate

El learning rate es uno de los parámetros mas importantes que hay que modificar a la hora de entrenar una red neuronal. Este parámetro determina cuanto de grandes serán los pasos que tome la red en dirección al óptimo en cada iteración.

El entrenamiento debe comenzar con un learning rate relativamente grande porque, al principio, los pesos que han sido inicializados de manera aleatoria, están lejos de ser los óptimos. Según la red vaya aprendiendo, se necesitan modificaciones mas precisas y por lo tanto, el learning rate debe ir disminuyendo.

Existen muchas funciones que definen como el learning rate va disminuyendo a lo largo del entrenamiento. En la arquitectura ResNet [24] por ejemplo, se utiliza un learning rate del tipo escalón. Este, comienza siendo de 0.1, es disminuido un factor 10 en la iteración 32.000

y de nuevo en la iteración 48.000. El entrenamiento se finaliza tras realizar un total de 64.000 iteraciones.

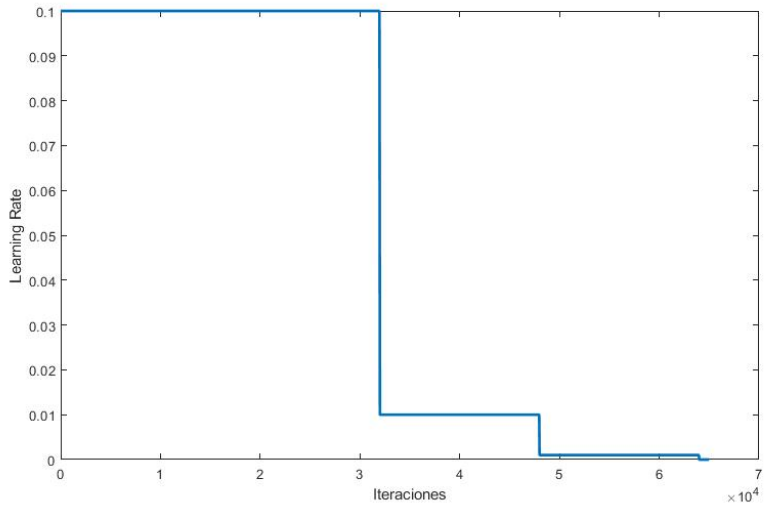


Figura 27: Función Learning Rate ResNet

El uso de esta función para el learning rate, implica la noción a priori del número de iteraciones que se van a realizar. Este dato no lo sabemos para nuestra red. Podríamos establecer un número muy elevado de iteraciones y esperar que la red terminara de entrenar al cabo de unos días. Pero, a parte del coste energético que esto supone, si la red no es lo suficientemente robusta, una cantidad elevada de iteraciones puede dar lugar a un sobreentrenamiento de la misma. Es decir, la red aprende las características propias del set de entrenamiento y por lo tanto sufre a la hora de analizar nuevas imágenes nunca vistas anteriormente (set de validación). Este dato se puede detectar fácilmente cuando el error del set de entrenamiento es menor que el del set de validación.

Como no se sabe a priori el número de iteraciones que va a necesitar la red para que converja por completo, en la implementación de nuestras redes, se ha aplicado la técnica de regulación "Early Stopping". Esta regulación consiste en analizar eventualmente el error de predicción para el set de validación. Si este ha dejado de disminuir tras varias iteraciones, significa que la red ha convergido y se debe dar por finalizado el entrenamiento.

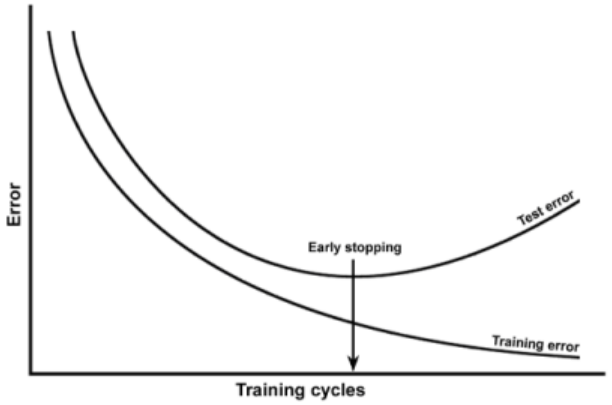


Figura 28: Sobreentrenamiento y "'Early Stopping'

En nuestro caso, realizaremos el analisis del error de predicción del set de validación tras finalizar una "Epoch". Si el error deja de disminuir en 3 "Epoch" consecutivas, se detiene el entrenamiento.

Ahora que ya sabemos cuando vamos a analizar el error de predicción del set de validación, vamos a establecer un Learning rate que tenga un valor alto (10^{-4}) al inicio de la "Epoch". De este modo, los pesos converjerán más rápidamente y explorarán nuevos caminos. Al final de la "Epoch" estableceremos un valor bajo de Learning Rate ($10^{-}7$). Así, la red hará modificaciones mas precisas de los pesos para reducir lo mejor posible el error de predicción.

Para la elección de la función que defina nuestro Learning Rate, nos basaremos en el artículo "Stochastic gradient desdent with warm restarts" [27]. En este artículo nos demuestran que el uso de un Learning Rate basado en la función coseno con reinicios ayuda a una rápida convergencia de la red. Por lo tanto la ecuación de nuestro Learning Rate tendrá la siguiente forma:

$$lr = minLR + \frac{1}{2} * (maxLR - minLR) * (1 + \cos(\frac{iter}{maxIter} * \pi))$$

Donde $maxIter$, corresponde con la iteración en el que el Learning Rate será mínimo. Este valor coincide con la última iteración de la Epoch es decir $\frac{Size\ del\ dataset}{Batch\ Size} = \frac{50000}{32} = 1563$

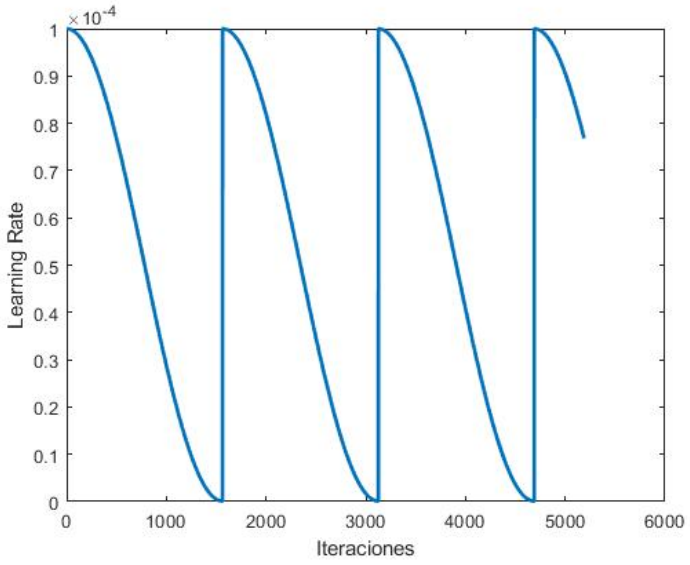


Figura 29: Función Learning Rate Coseno

Inicialización de las variables y normalización de datos

La inicialización de las variables es otro factor importante en las redes neuronales. Como hemos mencionado, la función de activación mas usada en el ámbito de las redes neuronales es la función ReLU. Esta será la función que usemos en prácticamente todas las capas de nuestra red. Hay que recordar que esta función no limita la salida a ningún rango de valores concreto. Si la entrada es infinito, la salida también lo será.

Para evitar que al inicio del entrenamiento se generen salidas desproporcionadas de las funciones de activación, aplicaremos tres técnicas de normalización distintas:

1. **Inicialización de los pesos**: Inicializaremos los pesos de manera aleatoria siguiendo una distribución normal de media 0 con una desviación estándar de 0.1.

2. **Normalización de los datos**: Las imágenes de entrada de la base de datos están cuantificadas en valores del rango 0,255. Lo que haremos es dividir la imagen de entrada entre 255, de esta forma, su rango de valores pasará a ser de 0,1.

3. **Regularización L2**: Otra forma de evitar que los pesos de la red converjan a valores extremadamente altos es aplicando la Regularización L2. Esta consiste en aplicar una penalización a la función de error que dependa del valor de los pesos de la red. Para cada filtro y neurona de la red calculamos esta penalización mediante la siguiente función:

   $L2\_loss_k = \frac{\sum W_k^2}{2}$

   Una vez calculada esta penalización se la sumamos al error de predicción. La red debe aprender a reducir el valor de los pesos y así disminuir el error de predicción.

$$Error\_prediccion = E + L2\_loss$$

Como vemos, esta regulación no solo depende del valor de los pesos sino que también del número de neuronas que la red posee. A mayor número de neuronas mayor es la penalización. Esto nos ayudará a disminuir la probabilidad de sobreentrenamiento de la red. Como sabemos, el sobreentrenamiento aparece por un excesivo uso de neuronas en la red. Cada neurona es capaz de aprender una característica propia de cada imagen de entrenamiento y así activarse cuando esta imagen pase por la red. Como estamos penalizando el valor de los pesos, las neuronas tendrán salidas con valores pequeños y la red tendrá que tener en cuenta la salida de múltiples neuronas a la hora de clasificar una imagen.

5.3.2. CNN con inception y skip connections

En base al estudio del estado del arte que hemos realizado, hemos visto que las CNN han dominado en el ámbito de la clasificación de imagen. Por ello, se ha implementado una CNN que posee los diferentes conceptos que hemos aprendido de las arquitecturas del estado del arte actual.

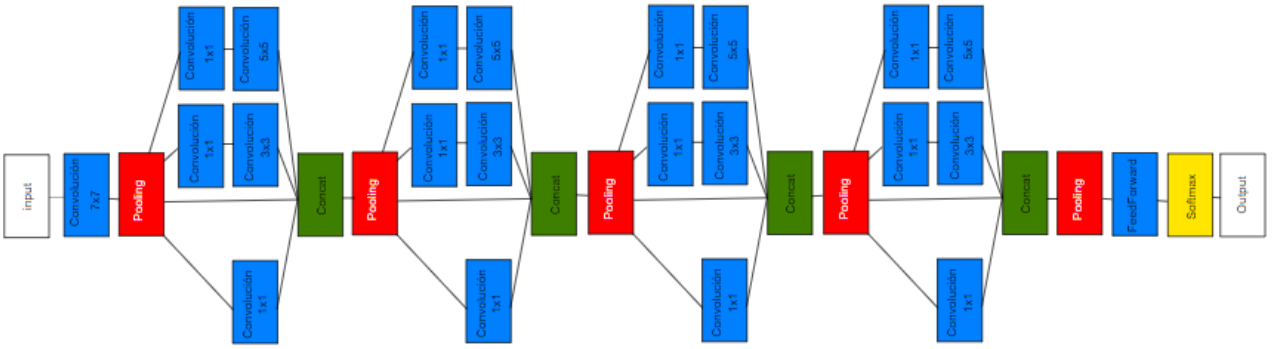


Figura 30: CNN con inception y skip connections

En la imagen 30 podemos ver la arquitectura final de la red neuronal implementada. La red posee 4 capas convolucionales con el módulo de inception. En cada capa convolucional realizamos la convolución con 3 filtros de distinto tamaño (1x1, 3x3, 5x5). La salida de la capa convolucional es la concatenación de las salidas de los filtros mas la entrada de la capa sin modificar siguiendo el estilo propuesto por DenseNet. La salida de la última capa convolucional se corresponde con 8x8x1024 características de la imagen original. Estas características pasan por una red Deep FeedForward que mediante un paso intermedio convierte estas 1024x8x8 características en 1024 y finalmente en 50.

En la tabla 1 podemos observar mas detalladamente los filtros que se han utilizado para cada capa y las dimensiones de salida de las mismas. Todas las capas convoucionales incluidas las que se encuentran en los módulos inception usan la función de activación ReLu. Las columnas "3x3 reduce " y "5x5 reduce" indican el número de filtros 1x1 que se han utilizado para reducir las dmensiones antes de aplicar las convoluciones de los filtros 3x3 y 5x5.

tipo	Tamaño del filtro	dimensiones salida	1x1	3x3 reduce	3x3	5x5 reduce	5x5
Entrada		256x256x3					
convolución	7x7	256x256x32					
maxpool	2x2	128x128x32					
inception		128x128x128	64		16		16
maxpool	2x2	64x64x128					
inception		64x64x256	64	16	32	16	32
maxpool	2x2	32x32x256					
inception		32x32x512	128	32	64	32	64
maxpool	2x2	16x16x512					
inception		16x16x1024	256	64	128	64	128
maxpool	2x2	8x8x1024					
FeedForward		1x1x1024					
dropout (60 %)		1x1x1024					
FeedForward		1x1x50					
Softmax		1x1x50					

Tabla 1: Detalle CNN con inception y skip connections

A continuación, profundizaremos de manera mas detallada en los parámetros mas relevantes de la red.

Modulo inception y skipp connection y DenseNet

Uno de los conceptos que aprendimos de la arquitectura GoogLeNet [23] y su módulo inception es que para mejorar el rendimiento no solo podemos aumentar la profundidad de la red si no que también su anchura. En cada capa convolucional, podemos aplicar filtros de diferentes tamaños y sumarlos a la salida. De este modo, dejaremos que la red decida a que filtros les dará mas peso. Por otro lado, vimos en la arquitectura ResNet [24] que podemos saltarnos conexiones entre capas convolucionales. Podemos sumar la salida de la primera capa convolucional con la salida de la segunda y pasarla a la entrada de la tercera sin realizar ninguna modificación. Así, lograríamos que las características que obtienen las diferentes capas circulen a lo largo de toda la red. Por último, de la arquitectura DenseNet [25] descubrimos que si realizamos esta conexión concatenando las salidas en lugar de sumarlas, logramos que las características de las capas fluyan de una manera mas directa a lo largo de toda la red.

Todos estos conceptos han sido aplicados directamente en la implementación de la red.

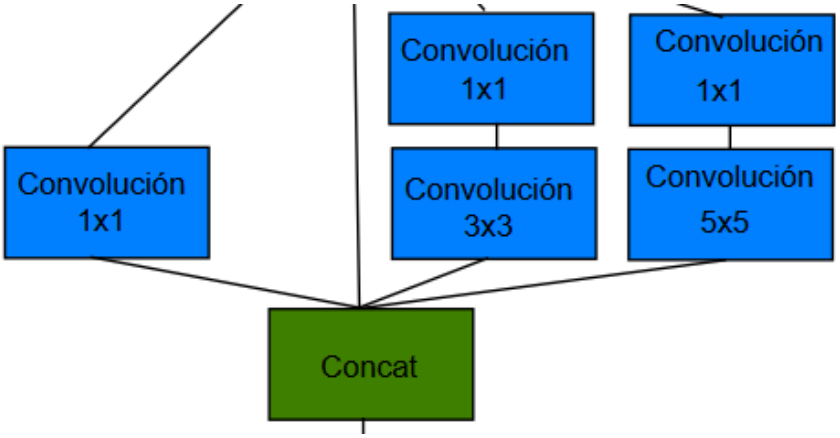


Figura 31: Módulo inception

En la figura 31 podemos ver una capa convolucional de la CNN. En esta, se aplican 3 filtros de diferentes tamaños (1x1,3x3,5x5) y se concatena la salida con la entrada sin modificar. Si nos fijamos en la imagen, antes de realizar las convoluciones de 3x3 y de 5x5 realizamos una convolución con un filtro 1x1. Esta convolución previa se realiza por dos motivos. El primero es que añade un nivel mas de no linealidad a la red y por lo tanto la hace algo mas compleja. El segundo motivo es para conseguir una reducción previa de dimensiones. El enlace [28] se trata de un vídeo de un curso de la plataforma coursera [29] en el que se explica gráficamente este concepto.

Red FeedForward

Tras la obtención de los mapas de características por parte de los módulos inception, es el turno de la red FeedForward. Esta red como sabemos necesita vectores unidimensionales a la entrada. Por ello, tras la última capa convolucional aplicamos una capa "flatten". Esta capa convierte los 1024 mapas de características de dimensiones 8x8 en un único vector unidimensional de tamaño 1024x8x8.

La red FeedForward es la que finalmente nos dará una predicción de las imágenes de entrada. Esta lo hará mediante el vector de características que los módulos inception previos han obtenido. La arquitectura de la red FeedForward es la que podemos ver en la figura 32

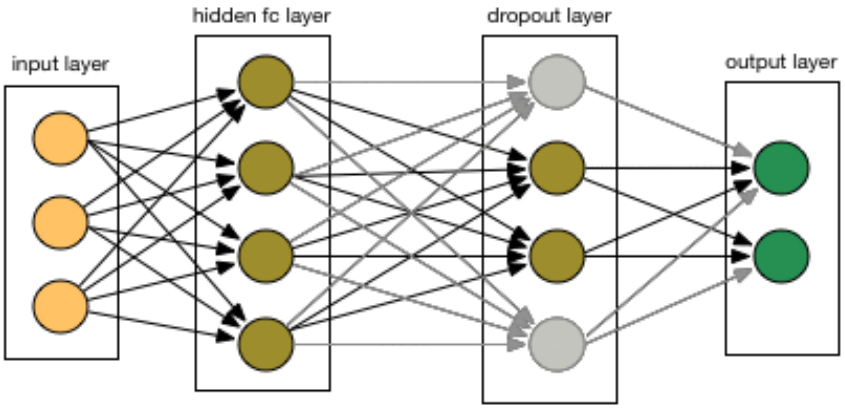


Figura 32: Red FeedForward

La red tiene una única capa oculta. Convierte las 1024x8x8 caracteristícas de entrada en 1024 y finalmente en 50. Las redes FeedForward son las que mas problemas provocan en general en el sobreentrenamiento. Estas redes son las que mas parámetros concentran en la red. Por ejemplo en nuestro caso, la capa oculta posee 1024 neuronas. Cada una de estas neuronas tiene 8x8x1024 entradas y por lo tanto el mismo número de pesos. Si hacemos la cuenta, da lugar a un total de 67.108.864 parámetros en la capa oculta.

Para evitar que esta inmensa cantidad de parámetros provoque un sobreentrenamiento en la red hemos aplicado una capa que se denomina "Dropout". Esta capa actua durante el entrenamiento. Elige con una cierta probabilidad en cada iteración que neuronas se deben tener en cuenta y cuales no. De esta forma, obligamos que las neuronas no creen codependencias entre ellas e intenten sacar el máximo de información por si solas.

La probabilidad con la que esta capa descarta a las neuronas es del 60 %. Es decir, en cada iteración de entrenamiento solo el 40 % de neuronas estarán activas. Cabe destacar que esta capa solo se usa durante el entrenamiento. Cuando midamos la tasa de acierto del clasificador queremos que se usen todas las neuronas y asi obtener la mejor predicción. Por lo tanto, en ese caso, la probabilidad con la que se descartarán las neuronas será del 0 %.

Salida de la red

Pasamos ya a la última capa de la red, la que contiene la predicción de la imagen de entrada. Esta capa contiene 50 neuronas y por lo tanto devuelve 50 valores. Estos 50 valores corresponden a la codificación "One hot" de la imagen de entrada. Queremos que esta salida se encuentre en términos de probabilidad por lo tanto usaremos la función de activación Softmax únicamente en esta capa.

La función Softmax tiene la siguiente formula:

$$\phi(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Donde z_j corresponde a la salida de la neurona j.

Esta función de activación es muy diferente de la ReLU. La salida de Softmax está acotada a un rango de valores de entre 0 y 1. Además su función depende del resto de neuronas de la misma capa.

Si nos fijamos en la fórmula, nos daremos cuenta de que la función Softmax devolverá valores altos para aquellas neurona que predominen sobre el resto. Además, la suma de las salidas de todas las neuronas siempre será uno. Por ello, la salida de Softmax la podemos interpretar como la probabilidad de que una categoría sea cierta.

La red durante el entrenamiento, tendrá que converger y modificar los pesos para maximizar la probabilidad que corresponde con la categoría de cada imagen.

Función de error

Como hemos mencionado en las bases del DeepLearning, necesitamos definir una función de error para el entrenamiento de la red. La función de error que hemos utilizado en este caso es la función "Cross Entrophy Loss". Esta función se define con la siguiente fórmula:

$H_{y'}(y) := -\sum_i y_i \log(y'_i)$

Donde y'_i es la probabilidad que devuelve la red para la clase i. y_i es la probabilidad real de la clase i.

Para entender mejor esta fórmula volvamos al ejemplo anterior de los colores. Tenemos 3 colores: Rojo, Verde y Azul. Aplicando la codificación "One hot" tenemos:

Rojo: 1 0 0, Verde: 0 1 0 y Azul: 0 0 1

Supongamos que la red, tras pasarle una imagen de la categoría Rojo obtiene la siguiente predicción:

predicción: 0.7 0.2 0.1

El error calculado mediante la función Cross Entrophy será el siguiente:

$Error = -(1 * \log(0{,}7) + 0 * \log(0{,}2) + 0 * \log(0{,}1)) = 0{,}15$

Como podemos ver, el error solo depende del término que corresponde con la categoria correcta de la imagen. De esta forma, la red aprendende que probabilidad tiene que maximizar en cada caso y modifica los pesos para ello.

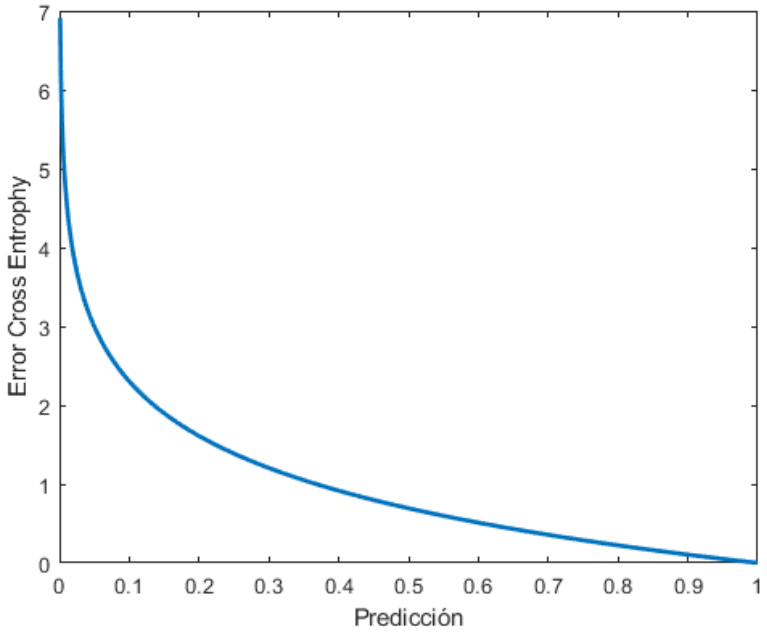


Figura 33: Error Cross Entrophy

En la figura 33 se observa una gráfica del error de predicción. El error es infinito para predicciones con muy baja probabilidad. Por otro lado, si la predicción es 1 quiere decir que la predicción es perfecta y por lo tanto el error que supone es 0.

Cabe recordar que en nuestra red, la función de error se compone de dos términos: el término correspondiente al "Cross Entrophy" mas el correspondiente a la normalización L2. Finalmente la función de error es:

$$Error\_prediccion = -\sum_i y_i \log(y_i')s + L2\_loss$$

Una vez definido el error de predicción, ya solo queda que la red en cada iteración vaya modificando los pesos hasta que finalmente converja. Debido a que estamos usando un Batch size muy pequeño (32), la red en cada iteración obtiene muy poca información de la base de datos y por lo tanto la convergencia es muy lenta. Una de las técnicas que podemos utilizar para mejorar la velocidad de convergencia se llama el "Momentum". Esta técnica consiste en utilizar información de iteraciones previas para modificar los pesos de la iteración actual. Lo hace agregando una fracción λ del vector de actualización de los pesos de la iteración anterior al vector de actualización de los pesos de la iteración actual.

Esta técnica actua como filtro paso bajo a la función de convergencia de la red. El valor de λ que se utilizará en la implementación de la red sera de 0.9.

6. Otras implementaciones

Durante el desarrollo de este proyecto se han ido realizando diferentes pruebas implementando otras arquitecturas de redes neuronales y revisando sus resultados.

6.1. Red neuronal CNN similar a VGG

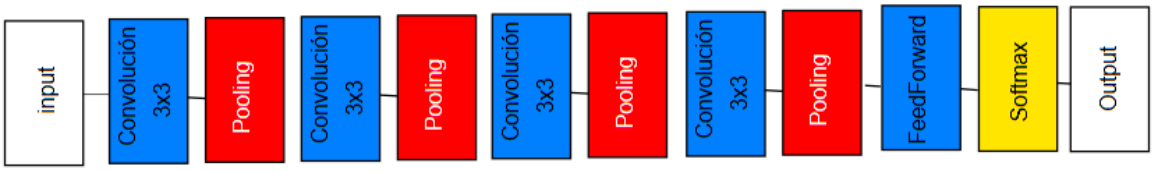


Figura 34: Red CNN similar a VGG

En la figura 34 podemos ver la estructura de la red CNN implementada. Como vimos en las arquitecturas que forman parte del estado del arte actual, la red VGG [22] se trata de una red simple pero que da muy buenos resultados. Basándonos en el hardware del que disponemos es una buena opción implementar una red no demasiado profunda y que utiliza filtros relativamente pequeños (3x3).

La versión de VGG finalmente implementada tiene 4 capas convolucionales que utilizan filtros de tamaño 3x3. La última capa convolucional genera 128 mapas de características de dimensiones 16x16. Al igual que en la CNN con inception, convertimos estos mapas de características a un array unidimensional para que la red FeedForward de una capa oculta finalmente obtenga la predicción.

La función de error de esta red es la misma que se utilizó para la CNN con inception: "Cross Entrophy" con la regulación L2. En esta red también se utiliza el optimizador de momentum para aumentar la velocidad de convergencia de la red.

En la tabla 2 se especifican de manera detallada las diferentes capas de la red con las dimensiones y el número de filtros utilizados.

tipo	Tamaño del filtro	dimensiones salida	3x3
Entrada		256x256x3	
convolución	3x3	256x256x16	16
maxpool	2x2	128x128x16	
convolución	3x3	128x128x32	32
maxpool	2x2	64x64x32	
convolución	3x3	64x64x64	64
maxpool	2x2	32x32x64	
convolución	3x3	32x32x128	128
maxpool	2x2	16x16x128	
FeedForward		1x1x1024	
Dropout 60 %		1x1x1024	
FeedForward		1x1x50	
Softmax		1x1x50	

Tabla 2: Red CNN similar a VGG

6.2. Autoencoder + FeedForward

A lo largo de este proyecto, hemos aprendido que las redes neuronales para clasificación de imagen siguen siempre el mismo procedimiento. Utilizan una serie de capas convolucionales que extraen las características fundamentales de la imagen. Luego, estas características son utilizadas por una red FeedForward para que pueda clasificar la imagen de entrada.

Hasta ahora, la red convolucional mediante el entrenamiento ha decidido cuales son las caracteríticas óptimas que debe extraer de la imagen para que la red FeedForward clasifique lo mejor posible.

En esta implementación hemos cambiado un poco el paradigma. Vamos a crear una red(Autoencoder), que sea capaz de codificar una imagen de dimensiones 256x256 de 3 canales (R,G,B) a un total de 32768 valores de tal forma que luego sea capaz de reconstruirla lo mejor posible. Una vez que ya tenemos una red capaz de hacer esto, utilizamos una segunda red (FeedForward) que coge la salida del codificador e intenta clasificarla del mismo modo que lo hacía en el resto de arquitecturas

La idea de esta arquitectura, es que la red Autoencoder, sea capaz de codificar la imagen con una información que luego pueda ser valiosa ya no solo para reconstruir si no que también para clasificar.

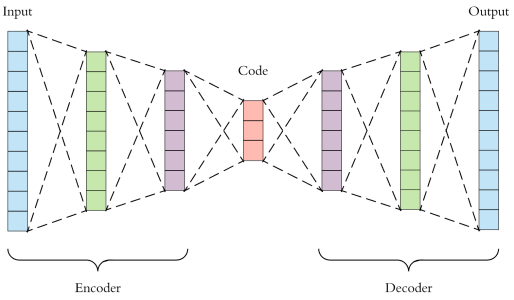
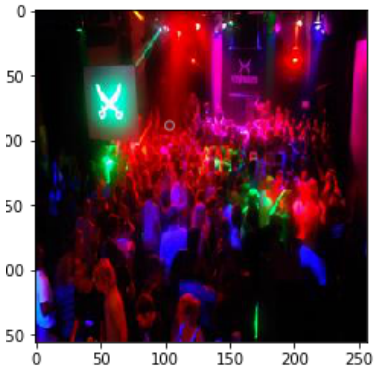
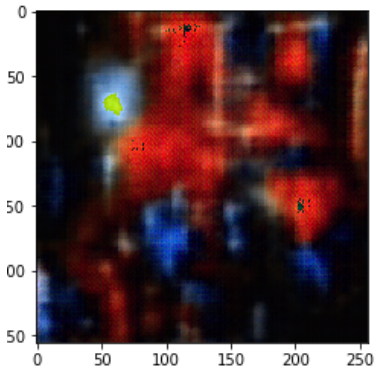


Figura 35: Autoencoder

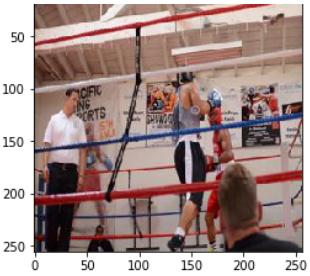
En la figura 35 podemos ver la arquitectura del Autoencoder. Cabe destacar que en todo AutoEncoder, existen dos redes que funcionan como codificador y decodificador. Ambas redes tienen una estructura simétrica entre si. El codificador se trata de una red convolucional de 4 capas que utilizan filtros 3x3. La última capa convolucional del codificador genera 128 mapas de características de dimensiones 16x16. Esta última capa es importante ya que será la entrada de la red FeedForward en la fase de clasificación. El decodificador es una red convolucional simétrica al codificador. La última capa del codificador genera una imagen de dimensiones 256x256 con 3 canales. La salida de esta capa convolucional se trata de la imagen de entrada tras ser codificada y posteriormente decodificada. Si la red funciona correctamente, dicha salida debe ser lo mas similar a la entrada posible.



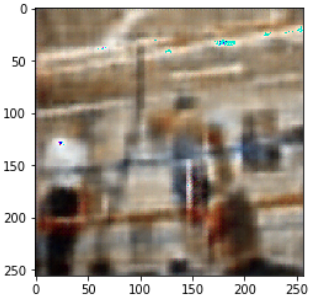
(a) Discoteca Original



(b) Discoteca Reconstruida



(c) Gimnasio Original



(d) Gimnasio Reconstruido

Figura 36: Codificación y decodificación de una imagen

En la figura 36 podemos ver imágenes originales a la izquierda y las imágenes tras pasar por el autoencoder a la derecha. Como vemos, la red ha perdido información puesto que codificamos la imagen con menos valores de los que originalmente tenía.

Para que la red actúe como Autoencoder, es importante definir una función de error que dirija el entrenamiento en esa dirección. La función de error escogida es MSE (Mean Square Error) que sigue la siguiente fórmula:

$$MSE = \frac{1}{MN} \sum_{n=1}^{N} \sum_{m=1}^{M} [Y_{n,m} - \hat{Y_{n,m}}]^2$$

Donde $Y_{n,m}$ es la imagen de entrada y $\hat{Y_{n,m}}$ se trata de la imagen que ha devuelto el Autoencoder tras codificar y decodificar.

Este error expresado en términos de Tensor flow será:

MSE = tf.square(tf.subtract(input,output_pred))

En este caso, no hemos utilizado la normalización L2. La forma en la que es entrenada la red es mediante el algoritmo Adam. Para saber mas acerca del funcionamiento de este algoritmo mirar la siguiente referencia [30].

En la tabla 3 podemos ver en detalle las capas que posee el codificador del Autoencoder. Del mismo modo, en la tabla 4 tenemos la arquitectura que corresponde con el Decoder.

tipo	Tamaño del filtro	dimensiones salida	3x3
Entrada		256x256x3	
convolución	3x3	256x256x16	16
maxpool	2x2	128x128x16	
convolución	3x3	128x128x32	32
maxpool	2x2	64x64x32	
convolución	3x3	64x64x64	64
maxpool	2x2	32x32x64	
convolución	3x3	32x32x128	128
maxpool	2x2	16x16x128	

Tabla 3: Codificador Autoencoder

tipo	Tamaño del filtro	dimensiones salida	3x3
unpool	2x2	32x32x128	
convolución	3x3	32x32x64	64
unpool	2x2	64x64x64	
convolución	3x3	64x64x32	32
unpool	2x2	128x128x32	
convolución	3x3	128x128x16	16
unpool	2x2	256x256x16	
convolución	3x3	256x256x3	3
Salida		256x256x3	

Tabla 4: Decodificador Autoencoder

Una vez que tenemos el Autoencoder funcionando, lo único que queda es entrenar una red FeedForward que utilice las características que obtiene el codificador para poder clasificar las imágenes de entrada. La red FeedForward implementada en esta arquitectura es la misma que se utilizó en la red convolucional sin inception.

Finalmente, la arquitectura de la red encargada de clasificar tiene la forma que se ve en la tabla 5.

tipo	dimensiones salida
Entrada	256x256x3
Codificador	16x16x128
FeedForward	1x1x1024
dropout (60 %)	1x1x1024
FeedForward	1x1x50
Softmax	1x1x50

Tabla 5: Clasificador con Autoencoder

6.3. Variational AutoEncoder + FeedForward

Uno de los problemas que plantea el uso del AutoEncoder básico para clasificación de imagen es que durante la codificación, intenta guardar el máxmimo de información de la imagen original. Debido a esto, codificaciones de dos imágenes que pertenecen a la misma categoría pueden ser completamente diferentes lo cual resulta un problema grave para la red FeedForward ya que clasificará estas imágenes en diferentes categorías.

El VAE(Variational Autoencoder) intenta solucionar este problema de forma que no solo intenta minimizar el error en la reconstrucción de la imagen si no que también minimiza la diferencia entre la distribución de los valores de codificación y una distribución Gaussiana de media 0 y desviación típica 1. De esta forma, el VAE aprende a representar las imágenes de entrada utilizando características que siguen una distribución normal. Al codificar todas las imágenes utilizando una distribución normal, imágenes de la misma categoría obtendrán codificaciones

similares y por lo tanto serán correctamente clasificadas.

La arquitectura del VAE se puede ver la figura 37. El codificador se compone de 4 capas convolucionales que utilizan filtros 3x3. La última capa convolucional genera 32 mapas de caraterísticas de dimensiones 16x16. Estos mapas de características son convertidos en un único vector unidimensional. Este vector será la entrada de 2 redes FeedForward de una sola capa que convierten estas 32x16x16 características en 8192. Por lo tanto, a la salida de estas dos redes, tenemos dos vectores de 8192 valores. Estos vectores serán el vector de medias y el vector de desviaciones típicas que describen la distribución Gaussiana de cada uno de los valores que codifica la imagen original. Para obtener estos valores solo nos queda muestrear dichas distribuciones con un valor aleatorio.

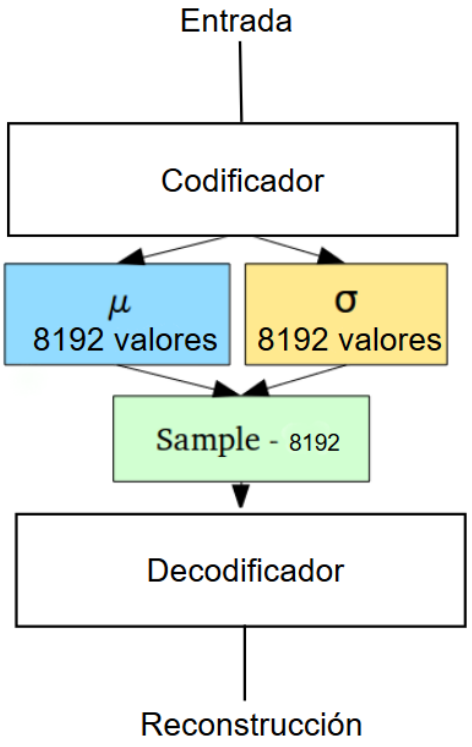


Figura 37: Variational Autoencoder

Una vez que tenemos codificada la imagen en 8192 valores se la pasamos al decodificador para reconstruya e intente obtener una imagen lo mas similar a la original.

La función de error que hemos tenido que diseñar para el VAE tiene dos componentes. El primer componente se trata del error MSE del mismo modo que se implementó en el Autoencoder de la arquitectura anterior. El segundo componente se trata de la distancia de la distribución de los valores de la codificación con respecto a una distribución normal de media 0 y desviación típica 1. Para calcular esta distancia hemos aplicado el algoritmo de KL-divergence [31]. TesorFlow nos ofrece la función tf.contrib.distributions.kl() la cual nos ayuda a calcular esta diferencia.

Entrenar VAEs puede ser un proceso complejo debido a las dos componentes que tiene la función de error. Si el componente que corresponde con las distribuciones Gaussianas tiene demasiado peso, la red tan solo aprende a predecir valores Gaussianos sin preocuparse en la correcta reconstrucción de la imagen. Para solucionar esto se comienza el entrenamiento optimizando únicamente la reconstrucción de la imagen. Tras varias iteraciones, se va imponiendo la condición de la distribución Gaussiana y así la red que ya sabe reconstruir, comienza a codificar la imagen siguiendo esta segunda condición.

Una vez definido el VAE, el resto de la arquitectura es similar a la descrita en el Autoencoder básico. En la tabla 6 podemos el diseño final de esta arquitectura.

tipo	dimensiones salida
Entrada	256x256x3
Codificador	1x1x8192
FeedForward	1x1x1024
dropout (60 %)	1x1x1024
FeedForward	1x1x50
Softmax	1x1x50

Tabla 6: Clasificador con VAE

7. Resultados

En esta sección vamos a comparar los resultados que hemos obtenido de las diferentes redes implementadas. En la tabla 7 podemos observar las tasas de acierto de las redes así como el número de Epoch que han sido necesarias para que la red converja. La columna "Top1-acc" indica la probabilidad con la que la red clasifica correctamente una foto. La columna "Top5-acc" indica la probabilidad de que la red muestre la categoría correcta de una imágen en los primeros 5 candidatos de una predicción.

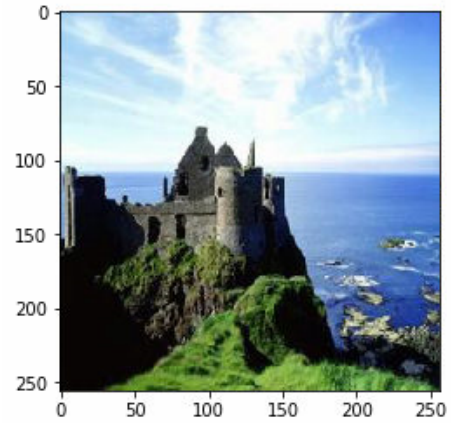
Red	Top5-acc	Top1-acc	NumEpoch
CNN con Inception	84,74 %	**56,94 %**	**32**
CNN similar a VGG	**85,44 %**	56,86 %	45
Autoencoder + FeedForward	71,06 %	37,99 %	47
VAE + FeedForward	63,95 %	31,34 %	41

Tabla 7: Comparativa de resultados entre las redes implementadas

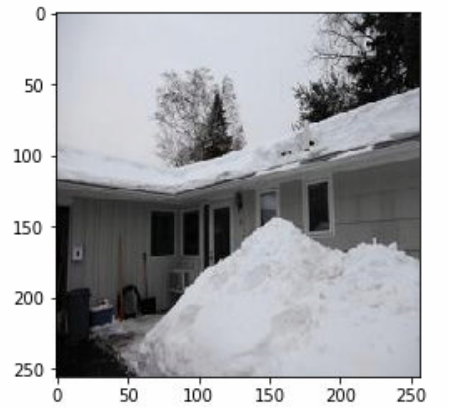
Como podíamos esperar, las redes convolucionales son las que mejor han funcionado en la tarea de clasificación de imagen. Comparando entre la CNN con inception y la CNN básica podemos ver resultados interesantes. Por un lado, la CNN básica ha obtenido una mejor tasa de acierto para el Top-1 sin embargo obtiene peores resultados para el Top-5. Es interesante fijarse en el número de Epochs que ha necesitado cada red para entrenar. Vemos que la CNN con inception ha sido la red que mas rápidamente ha convergido. Este dato corrobora lo visto en la teoría. Gracias a la utilización del módulo inception, la red es capaz de decidir que filtros ponderar mas para disminuir lo mas rápido posible la función de error. Por otro lado, la utilización de skip connections ha permitido que los mapas de características de las diferentes capas fluyan a lo largo de la red ayudando mas aun a la rápida convergencia de la red.

En el lado de los autoencoders, vemos que el autoencoder básico ha tenido mejores resultados. Esto puede ser debido a que el autoencoder básico utiliza un codificador que codifica la imagen en 128x16x16 características mientras que el VAE lo hace en 8192. Al reducir tanto las dimensiones de una imagen de entrada de 256x256x3 ha podido resultar en una falta de información para la red DeepFeedForward y por lo tanto una mala clasificación.

A continuación se pueden ver 2 ejemplos de las predicciones que la red CNN con inception ha obtenido **??**.



(a) Categoria real: castle



(b) Categoría real: ski_resort

Figura 38: Resultados de predicción de la red CNN con inception

Como podemos apreciar a la izquierda, la red ha predicho correctamente que la imagen pertenece a la categoría castillo. Además está bastante seguro de su predicción puesto que tiene una probabilidad del 70 %. En la imagen de la derecha, podemos ver un ejemplo en el que la red se ha confundido. Su predicción con mas peso se trata de hospital. Sin embargo, no se trata de un error muy grave la red no esta segura de la categoría a la que pertenece esta imagen y por ello genera una probabilidad del 23 %.

8. Conclusiones y lineas futuras

En el transcurso de este trabajo se han estudiado las bases de los algoritmos de DeepLearning centrándonos sobretodo en las redes neuronales. Luego, hemos realizado un análisis exhaustivo de los lenguajes de programación y librerías mas útilies para trabajar con estos algoritmos. Después hemos descrito un entorno de trabajo óptimo para el desarrollo de las redes neuronales. En este entorno de trabajo hemos especificado el lenguaje de programación que se va a usar así como las librerías complementarias. Finalmente hemos pasado a estudiar la evolución de las redes neuronales para la clasificación de imágenes hasta llegar al estado del arte actual. Por último, y teniendo en cuenta todo lo estudiado, hemos pasado a la implementación de nuestras redes neuronales. Para ello, primero hemos seleccionado una base de datos con la que trabajar y tras analizar dicha base de datos, hemos pasado a diseñar e implementar nuestras redes. Hemos implementado diferentes arquitecturas. En algunas, hemos utilizado conceptos comunes del estado del arte actual. Sin embargo en otras, hemos explorado nuevos caminos y estudiado diferentes alternativas. Tras la implementación hemos comparado los resultados de todas ellas.

Futuros proyectos podrán utilizar el entorno de trabajo definido en este proyecto además de las arquitecturas vistas para crear nuevas redes neuronales mas complejas con facilidad. Una de las ideas que se plantean para la continuación de este trabajo de fin de grado podrían ser la profundización de los Variational Autoencoders para la clasificación de imagen. Por otro lado, se podrían intentar aplicar estas arquitecturas en distintos datasets para ver si funcionan correctamente con diferentes entornos.

Referencias

[1] Gradient Descent y BackPropagation,
https://medium.com/@lahorekid/a-dummies-guide-to-gradient-descent-and-backpropagation
fecha consulta Agosto 2018

[2] NeuralNetworks y BackPropagation,
https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simp
fecha consulta Agosto 2018

[3] Python,
https://www.python.org/, fecha consulta Agosto 2018

[4] Indeed,
https://www.indeed.es/?r=us, fecha consulta Febrero 2018

[5] StackOverflow
https://stackoverflow.com/, fecha consulta Marzo 2018

[6] PYPL,
http://pypl.github.io/PYPL.html, fecha consulta Febrero 2018

[7] IEEE,
https://www.ieee.org/, fecha consulta Abril 2018

[8] Kaggle,
https://www.kaggle.com/, fecha consulta Marzo 2018

[9] Numpy,
http://www.numpy.org/, fecha consulta Agosto 2018

[10] NVIDIA,
https://www.nvidia.com/en-us/, fecha consulta Junio 2018

[11] Caffe,
http://caffe.berkeleyvision.org/, fecha consulta Julio 2018

[12] Theano,
http://deeplearning.net/software/theano/, fecha consulta Marzo 2018

[13] TensorFlow,
https://www.tensorflow.org/, fecha consulta Agosto 2018

[14] Keras,
https://keras.io/, fecha consulta Agosto 2018

[15] PyTorch,
http://pytorch.org/, fecha consulta Agosto 2018

[16] Pandas,
https://pandas.pydata.org/, fecha consulta Marzo 2018

[17] Matplotlib,
`https://matplotlib.org/`, fecha consulta Agosto 2018

[18] SciPy,
`https://www.scipy.org/`, fecha consulta Agosto 2018

[19] Anaconda,
`https://www.anaconda.com/`, fecha consulta Febrero 2018

[20] Spyder,
`https://pythonhosted.org/spyder/`, fecha consulta Febrero 2018

[21] Alex Krizhevsky, Llya Sutskever, Geoffrey E. Hinton *ImageNet Classification with Deep Convolutional Neural Networks*. [*Advances in Neural Information Processing Systems 25*]. pp:1097–1105, Curran Associates, Inc 2012.

[22] Karen Simonyan and Andrew Zisserman *Very Deep Convolutional Networks for Large-Scale Image Recognition*. [*arXiv*]. abs/1409.1556, 2014.

[23] Christian Szegedy and Wei Liu and Yangqing Jia and Pierre Sermanet and Scott E. Reed and Dragomir Anguelov and Dumitru Erhan and Vincent Vanhoucke and Andrew Rabinovich *Going Deeper with Convolutions*. [*CoRR*]. abs/1409.4842, 2014.

[24] Kaiming He and Xiangyu Zhang and Shaoqing Ren and Jian Sun *Deep Residual Learning for Image Recognition*. [*arXiv*]. abs/1512.03385, 2015.

[25] Gao Huang and Zhuang Liu and Kilian Q. Weinberger *Densely Connected Convolutional Networks*. [*arXiv*]. abs/1608.06993, 2016.

[26] MIT Places
`http://places2.csail.mit.edu/`, fecha consulta Agosto 2018

[27] Ilya Loshchilov and Frank Hutter *SGDR: Stochastic Gradient Descent with Restarts*. [*arXiv*]. abs/1608.03983, 2016.

[28] Convolución 1x1,
`https://www.coursera.org/lecture/convolutional-neural-networks/networks-in-networks-a`
fecha consulta Agosto 2018

[29] Coursera,
`https://www.coursera.org/`, fecha consulta Agosto 2018

[30] Diederik P. Kingma and Jimmy Ba *Adam: A Method for Stochastic Optimization*. [*arXiv*]. abs/1412.6980, 2014.

[31] KL-Divergence
`https://en.wikipedia.org/wiki/Kullback-Leibler_divergence`, fecha consulta Julio 2018

[32] Instalar TensorFlow con GPU,
`https://www.tensorflow.org/install/install_windows`, fecha consulta Abril 2018

ANEXOS

A. Anexo1 - Instalación de Anaconda para Windows

1. Descargue el instalador de Anaconda.

2. Opcional: Verifique la integridad de los datos con MD5 o SHA-256 . Más información sobre hashes

3. Haga doble clic en el instalador para iniciar.

   NOTA: Si encuentra algún problema durante la instalación, deshabilite temporalmente su software antivirus durante la instalación, luego vuelva a habilitarlo después de que finalice la instalación. Si se ha instalado para todos los usuarios, desinstale Anaconda y vuelva a instalarlo solo para su usuario y vuelva a intentarlo.

4. Haga clic en Siguiente.

5. Lea los términos de la licencia y haga clic en “Acepto”.

6. Seleccione una instalación para “Solo yo” a menos que esté instalando para todos los usuarios (que requiere privilegios de administrador de Windows) y haga clic en Siguiente.

7. Seleccione una carpeta de destino para instalar Anaconda y haga clic en el botón Siguiente. Ver FAQ .

   NOTA: instale Anaconda en una ruta de directorio que no contenga espacios ni caracteres Unicode.

   NOTA: No instale como administrador a menos que se requieran privilegios de administrador.

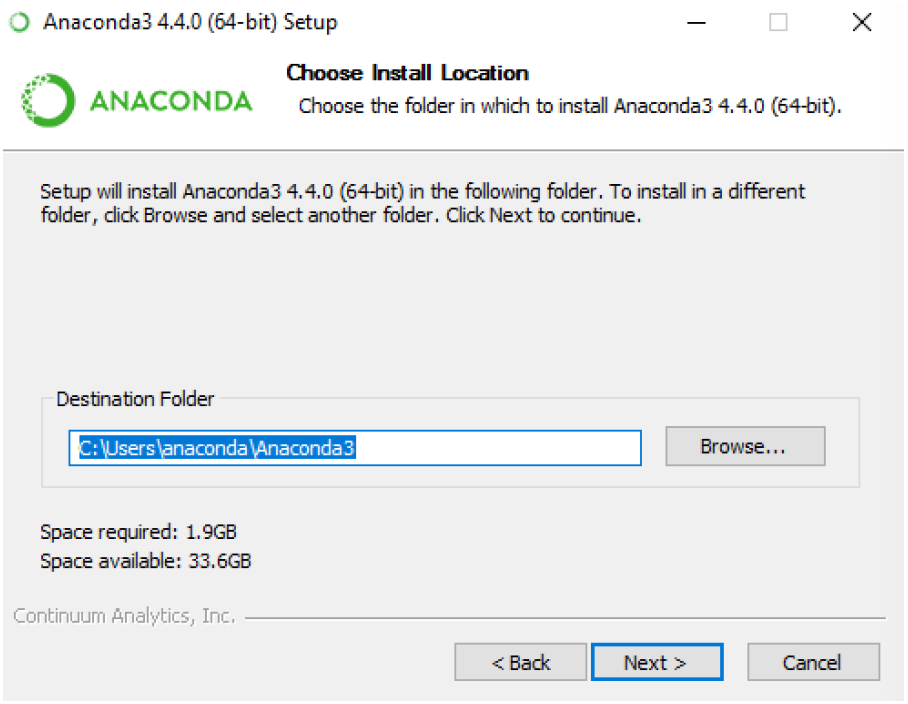


Figura 39: Ruta de instalación de Anaconda

8. Elija si desea agregar Anaconda a su variable de entorno PATH. Recomendamos no agregar Anaconda a la variable de entorno PATH, ya que esto puede interferir con otro software. En su lugar, use el software Anaconda abriendo Anaconda Navigator o Anaconda Prompt desde el Menú de Inicio.

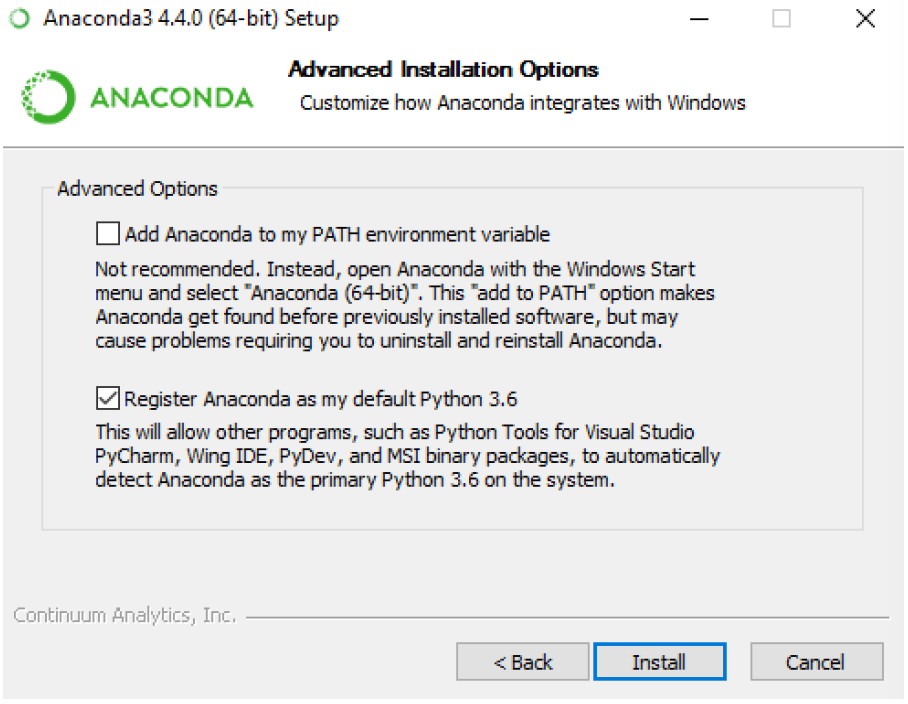


Figura 40: Opciones avanzadas de instalación

9. Elija si desea registrar Anaconda como su Python 3.6 predeterminado. A menos que planee instalar y ejecutar varias versiones de Anaconda, o múltiples versiones de Python, debe aceptar el valor predeterminado y dejar esta casilla marcada.

10. Haga clic en el botón Instalar. Puede hacer clic en Mostrar detalles si desea ver todos los paquetes que Anaconda está instalando.

11. Haga clic en el botón Siguiente.

12. Después de una instalación exitosa, verá el cuadro de diálogo "Gracias por instalar Anaconda":

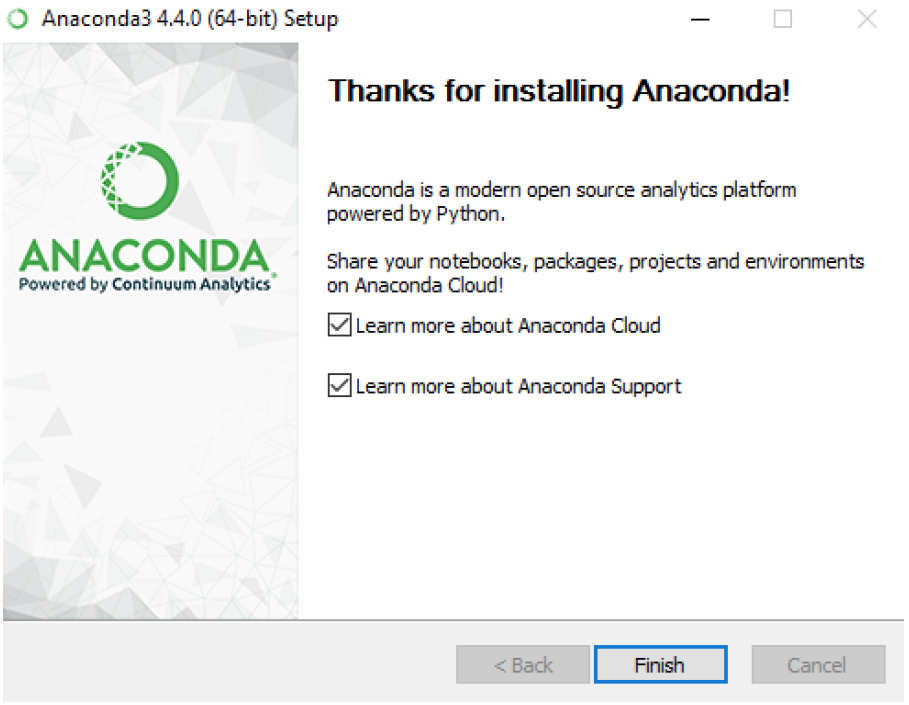


Figura 41: Finalizando la instalación de Anaconda

13. Puede dejar las casillas marcadas "Obtenga más información acerca de Anaconda Cloud" y "Obtenga más información sobre Anaconda Support" si desea leer más acerca de este servicio de administración de paquetes en la nube y el soporte de Anaconda. Haga clic en el botón Finalizar.

14. Una vez completada la instalación, verifíquela abriendo Anaconda Navigator, un programa que se incluye con Anaconda: desde el menú Inicio de Windows, seleccione el acceso directo Anaconda Navigator. Si se abre Navigator, ha instalado Anaconda correctamente. De lo contrario, verifique que haya completado cada paso anterior, luego vea nuestra página de Ayuda.

Como podemos observar, la instalación es sencilla. Le tomará un tiempo al programa instalar todos los paquetes necesarios pero finalmente ya tendremos prácticamente todo preparado para comenzar a programar.

A.1. Primeros pasos con Anaconda Navigator

Después de instalar Anaconda en Windows, inicie Navigator de la siguiente manera:
- Hacer clic en la aplicación de escritorio Anaconda Navigator desde el menú de inicio.

El flujo de trabajo básico para usar Navigator para instalar y ejecutar una librería es:

1. Crea y activa un nuevo entorno para el paquete.
2. Encuentra e instala la librería.
3. Trabaja con el entorno donde puedes acceder a las librerías.

A continuación explicaremos de manera mas detallada los pasos a seguir:

A.2. Crear y activar un nuevo entorno para librerías

1. En Navigator, haga clic en la pestaña Entornos , luego haga clic en el botón Crear.

   Aparecerá el cuadro de diálogo Crear nuevo entorno.

2. En el campo Nombre del entorno, escriba un nombre descriptivo para su entorno.

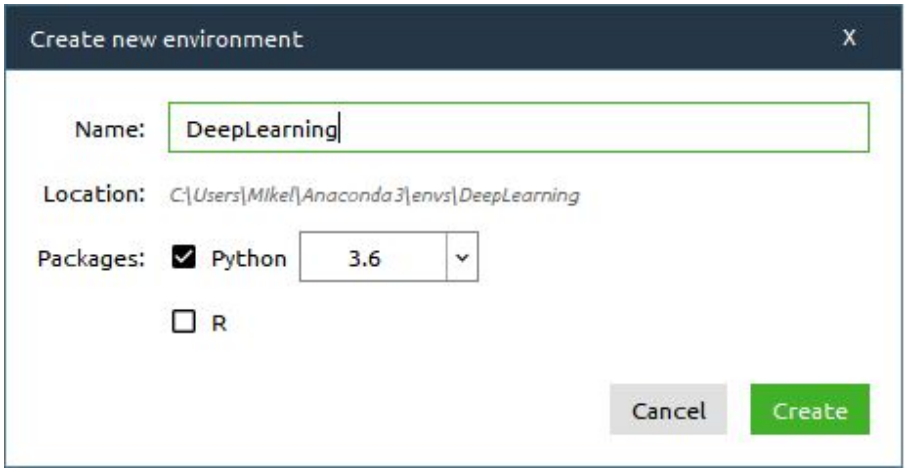


Figura 42: Creando un nuevo entorno

3. En la lista de versiones de Python, seleccione Python 3.6.
4. Haga clic en el botón Crear.

   Navigator crea el nuevo entorno y lo activa, como se muestra en la barra verde resaltada. Todas las acciones tienen lugar en el entorno activo.

Figura 43: Entorno ya creado

A.3. Abrir un terminal del entorno

Anaconda es una interfaz gráfica que nos facilita en muchos casos la instalación y actualización de librerías. Sin embargo todo esto lo podemos hacer también desde el terminal.
En esta sección aprenderemos a abrir un terminal con un entorno seleccionado. Esto es importante porque mas adelante haremos uso del terminal para ejecutar una serie de comandos que solo pueden ser ejecutados desde el aquí.

1. En la parte superior izquierda, debajo de la barra de búsquedas de entornos, hacer clic en el triangulo que se encuentra delante del entorno deseado.

2. Hacer clic en la opción “Abrir Terminal”

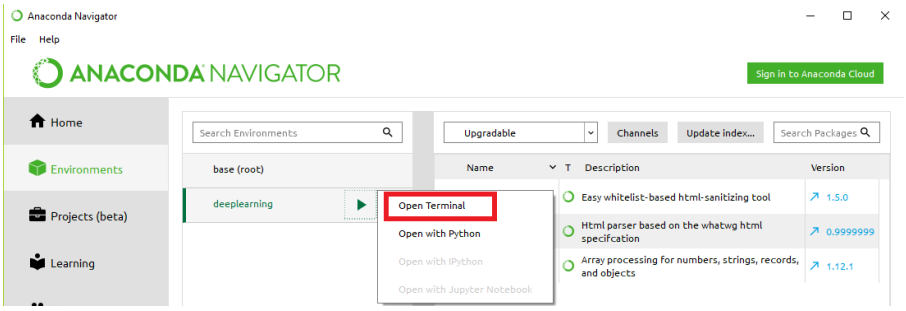


Figura 44: Abrir terminal.

En el terminal que se nos abre, podemos escribir cualquier comando que funcione con las librerías instaladas. En las siguientes secciones, daré algunos comandos útiles para el uso del terminal.

A.4. Encontrar e instalar un paquete

A.4.1. Mediante el terminal

1. Abra el terminal del entorno en el que quiere instalar las librerías.

2. Escriba: *"conda install 'nombre de la librería' "*

3. Revise las librerías que se instalarán, luego escriba "*y*z presione intro para confirmar.

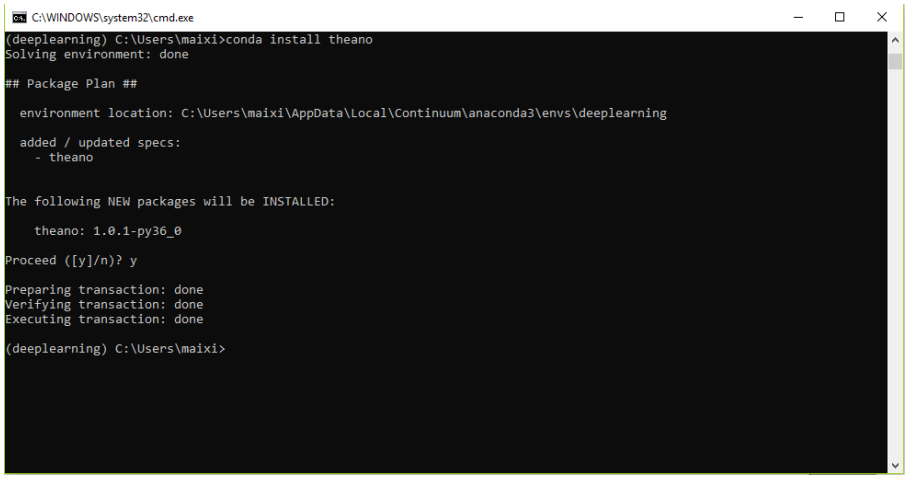


Figura 45

A.4.2. Mediante Anaconda Navigator

1. En la lista, en la parte superior izquierda en la sección de librerías, seleccione Todo.

2. En el cuadro búsqueda de librerías, escriba el nombre de la librería.

3. En los resultados de búsqueda, seleccione la casilla de verificación al lado de la librería.

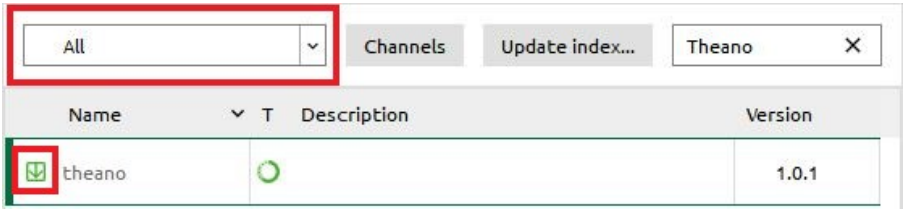


Figura 46: Buscando librerías

4. En la ventana Instalar librerías, revise las librerías que se instalarán y luego haga clic en el botón Aplicar. Luego, en Navigator, haz clic en el botón Aplicar.

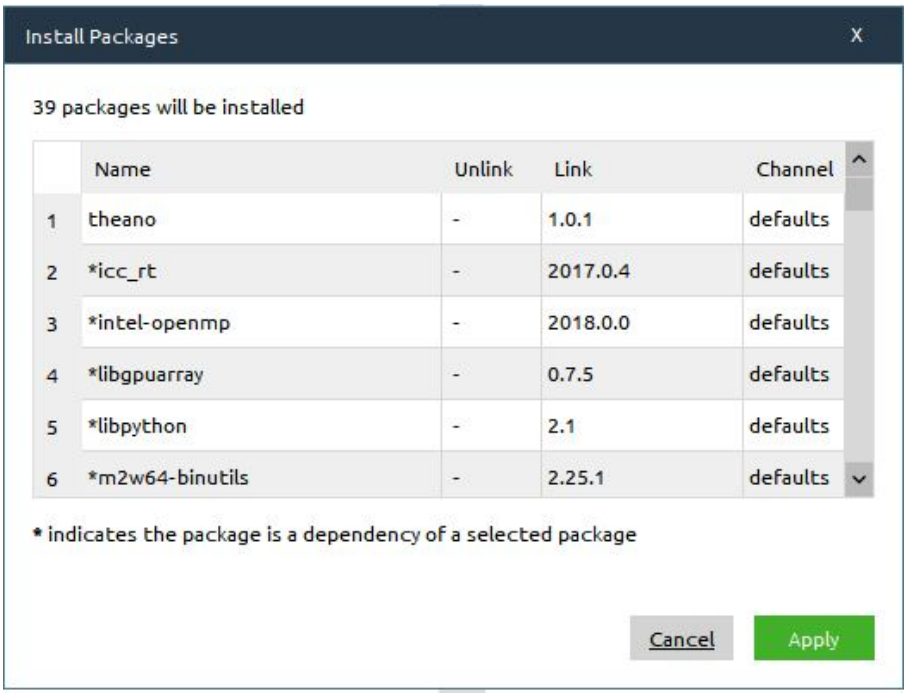


Figura 47: Instalando librerías

5. En el cuadro de diálogo de confirmación, haz clic en el botón Aceptar. Todos los archivos y dependencias de la librería están instalados en su nuevo entorno.

Siguiendo este método podemos ya incluir todas las librerías de una manera muy sencilla y trabajar con todas ellas en nuestro entorno de trabajo. Este gestor de librerías, es muy útil ya que podemos crear entornos en los que incluir diferentes versiones de librerías en diferentes versiones de Python y cambiar entre ellos de una manera muy fácil.

A.5. Actualizar librerías

A.5.1. Mediante el terminal

1. Abra el terminal del entorno en el que quiere actualizar las librerías.
2. Escriba: *"conda update 'nombre de la librería' "* y presione intro

   O

   Escriba: *"conda update −−all"* y presione intro. (Para actualizar todas las librerías a la vez)
3. Revise las librerías que se actualizarán, luego escriba "*y*" y presione intro para confirmar.

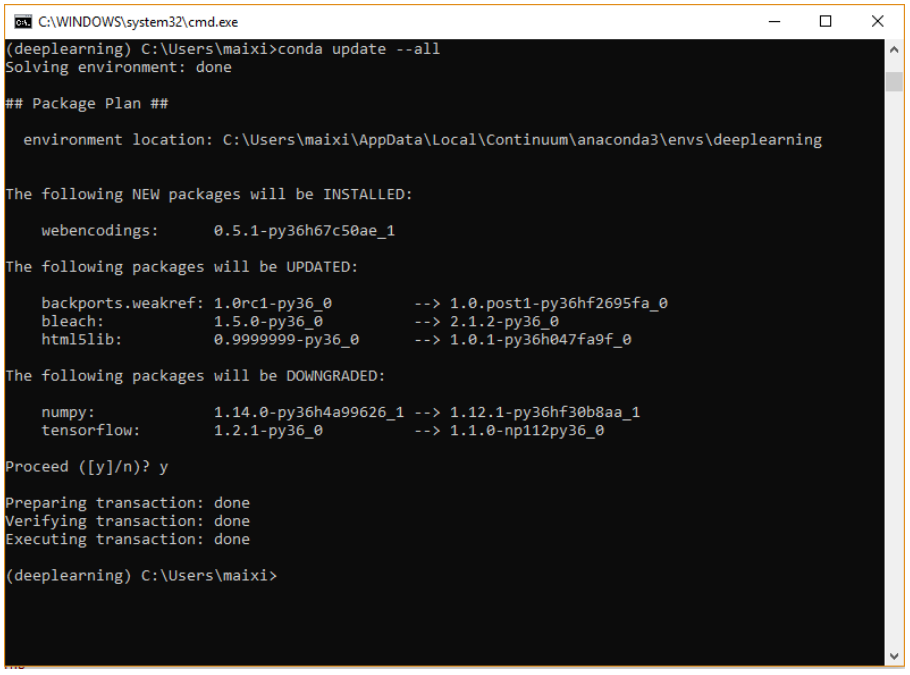


Figura 48: Actualizando librerías

A.5.2. Mediante Anaconda Navigator

1. Seleccione el filtro Actualizable para ver una lista de todas las librerías instaladas que tienen actualizaciones disponibles.

2. Seleccione la casilla de verificación junto a la librería que desea actualizar, luego en el menú que aparece, seleccione Marcar para actualizar.

   O

   En la columna Versión, haga clic en la flecha azul hacia arriba.

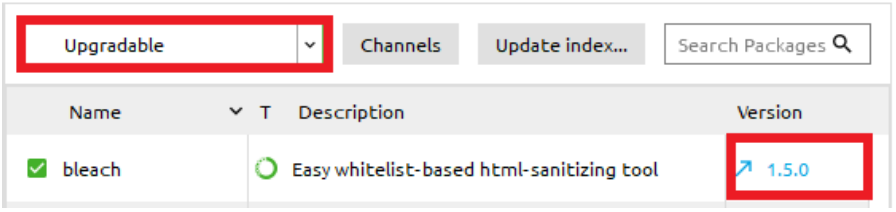


Figura 49: Actualizando librerías.

3. Haga clic en el botón Aplicar.

A.6. Lanzando el entorno de desarrollo

A.6.1. Mediante el Terminal

1. Abra el terminal del entorno en el que quiere abrir el entorno de desarrollo.

2. Escriba: *"nombre del entorno de desarrollo"* y presione intro.

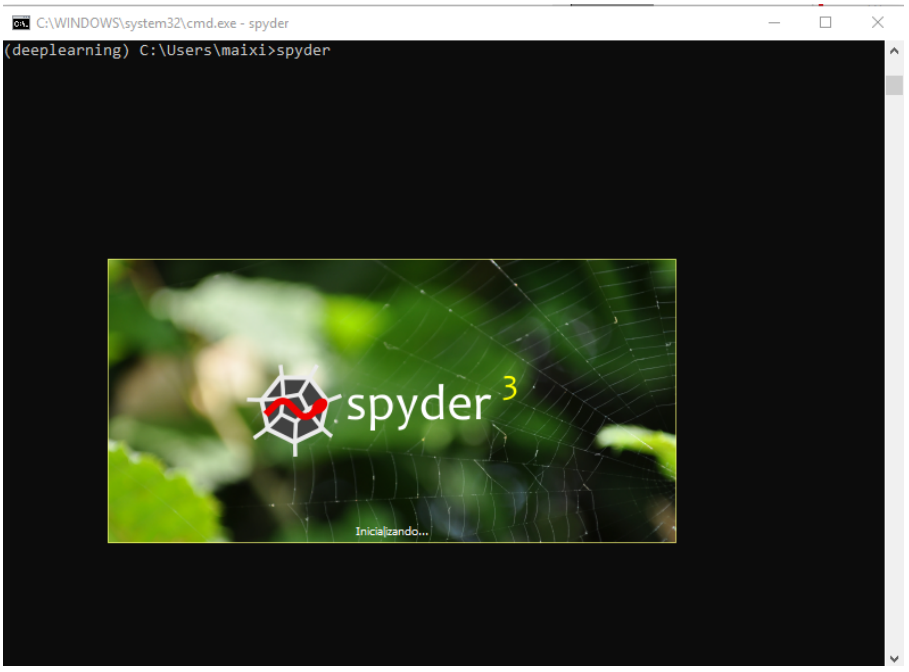


Figura 50: Ejecutando el entorno de desarrollo

A.6.2. Mediante Anaconda Navigator

1. En Navigator, haga clic en la pestaña Home.

2. En la parte superior, seleccione el canal en que tiene instaladas las librerías que quiere utilizar.

3. Haga clic en Launch del entorno de desarrollo que desea ejecutar.

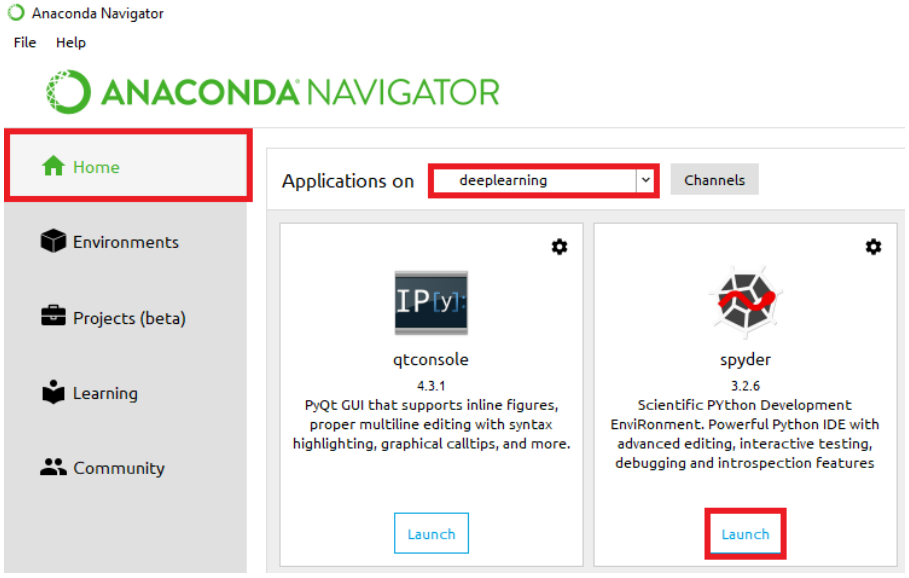


Figura 51: Ejecutando el entorno de desarrollo

NOTA: Si es la primera vez que ejecuta Spyder, le saldrán varias alertas de seguridad de su antivirus. Es importante decirle que es un falso positivo y crear una excepción para cada una de las alertas. Si no lo hace, saldrá un error a la hora de ejecutar programas diciendo: El núcleo dejó de funcionar, reiniciando.

B. Anexo2 - Instalar la compatibilidad de NVIDIA con TensorFlow

Para instalar TensorFlow con soporte para GPU visitar el siguiente enlace y seguir los pasos descritos. [32]