Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Navegación mediante realimentación visual de un robot móvil basado en técnicas de aprendizaje automático Deep Learning

Autor: Víctor Fernández Calderón

Tutor: Carlos Vivas Venegas

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Navegación mediante realimentación visual de un robot móvil basado en técnicas de aprendizaje automático Deep Learning

Autor:
Víctor Fernández Calderón

Tutor:
Carlos Vivas Venegas

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado: Navegación mediante realimentación visual de un robot móvil basado en técnicas de aprendizaje automático Deep Learning

Autor: Víctor Fernández Calderón
Tutor: Carlos Vivas Venegas

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Resumen

El objetivo de este Trabajo de Fin de Grado consiste en construir un robot móvil capaz de recorrer espacios cerrados evitando su colisión con los obstáculos que encuentre, usando un único sensor visual.

Para lograrlo, se han empleado algoritmos de *Deep Learning*, en concreto, redes neuronales artificiales, para la clasificación de imágenes. Para entender su funcionamiento se ha realizado una profunda revisión sobre las mismas, así como su desarrollo, desde las más simples, como son las *FeedForward Networks*, hasta las más complejas, como las *Convolutional Neural Networks*, que son las que se han usado en este proyecto ya que permiten una elevada tasa de precisión en la clasificación de imágenes.

Por otro lado, se ha usado Python como lenguaje de programación debido a su sencillez y uso extendido, y la librería TensorFlow de Google, que fue creada especialmente para diseñar redes neuronales de forma eficiente.

Para el entrenamiento de la red neuronal, se ha creado un *dataset* de entrenamiento formado por 16700 imágenes tomadas desde el propio robot móvil equipado con tres cámaras que permitieron la clasificación instantánea de dichos ejemplos.

Con la metodología descrita, se ha obtenido un 99.55 % de precisión en un *dataset* con 225 imágenes desconocidas por la red neuronal.

Como conclusión, podemos decir que para obtener buenos resultados en el tratamiento de imágenes, es necesario el uso de *Convolutional Neural Networks*, la creación de un *dataset* de entrenamiento variado y con muchos ejemplos, así como un buen *dataset* de testeo que permita validar el correcto entrenamiento de la red y por último, el control de los distintos parámetros de entrenamiento.

Prefacio

Los métodos de aprendizaje profundo o *Deep Learning* [1] y en concreto, las redes neuronales, son un campo de la Inteligencia Artificial que han tenido gran auge en los últimos años, sobre todo al mejorar la velocidad de procesamiento de los ordenadores más modernos. Debido a ello, se ha venido mostrando el gran potencial que tienen dichas redes en cualquier campo científico, ya que permiten obtener resultados que no se podrían alcanzar con otros algoritmos.

Como futuro ingeniero robótico que soy, esto también llamó mi atención para poder aplicarlo en el campo de la robótica, con la idea que el control de cualquier robot puede facilitarse respecto a las técnicas de control de robots tradicionales.

En este proyecto se ha llevado a cabo un trabajo de investigación en el que se estudia en profundidad las redes neuronales, para, finalmente, aplicar los conocimientos aprendidos a un robot con locomoción diferencial. A través de él veremos el importante potencial del *Deep Learning* y cómo se puede emplear en muchos campos para resolver ciertas tareas, obteniendo elevadas tasas de éxito.

[1] Nota: en este proyecto, los términos técnicos se han consignado en su denominación inglesa en cursiva al ser éstos ampliamente admitidos y usados en el área de conocimiento del aprendizaje automático (*Machine Learning*).

Índice

Resumen I
Prefacio III

1 Teoría de la Redes Neuronales Artificiales 1
1.1 Introducción 2
1.2 Deep Learning 2
1.3 Redes Neuronales Artificiales 5
1.4 FeedForward Neural Networks 8
1.4.1 Forward Propagation 8
1.4.2 Funciones de activación 10
1.4.3 Softmax Layer 13
1.5 Entrenamiento de Redes Neuronales 14
1.5.1 Gradient Descent 14
1.5.2 Backpropagation 16
1.5.3 Demostración de las ecuaciones del algoritmo Backpropagation 21
1.5.4 Delta Rule 23
1.5.5 Entrenamiento Eficiente: Batches 25
1.5.6 Overfitting 27
1.6 Conclusión 30

2 Introducción a TensorFlow 31
2.1 ¿Qué es TensorFlow? 32
2.2 Python, TensorFlow y otros módulos 33
2.3 Conceptos básicos de TensorFlow 34
2.4 FeedForward Network en TensorFlow 39
2.5 Conclusión 45

3 Convolutional Neural Networks 47
3.1 Introducción 48
3.2 Conceptos de Convolutional Neural Networks 50
3.2.1 Introducción 50
3.2.2 Filtros y Convoluciones 50
3.2.3 Max Pooling 55
3.3 Convolutional Neural Networks en TensorFlow 56
3.4 Conclusión 60

4 Material y Método 63
4.1 Introducción 64
4.2 El Robot 64
4.3 Captura de imágenes 67

4.4 Datasets de Entrenamiento y Testeo 69
4.5 La Red Neuronal 69
4.6 Funcionamiento de la Red Neuronal a tiempo real 72
4.7 Resultados y Conclusión 73
Bibliografía 77

5 Anexos 79
5.1 Anexo 1 79
5.2 Anexo 2 87
5.3 Anexo 3 90
5.4 Anexo 4 94
5.5 Anexo 5 96
5.6 Anexo 6 104
5.7 Anexo 7 107
5.8 Anexo 8 112
5.9 Anexo 9 117

Teoría de la Redes Neuronales Artificiales

Introducción

La finalidad de este proyecto consiste en guiar un robot de forma autónoma para que tenga la capacidad de moverse por espacios cerrados (como los pasillos de un edificio) empleando únicamente realimentación visual. Mediante el procedimiento descrito en el proyecto, se tendrá la posibilidad de usar un solo sensor visual sin la necesidad de otros sensores comunes en los distintos enfoques de navegación de robots, como los ultrasonidos.

Para poder cumplir con dicha finalidad, se procederá a diseñar una red neuronal que tenga la capacidad de clasificar imágenes tomadas en un espacio cerrado en tres categorías distintas: (1) según si el camino está libre por la ausencia de obstáculos, (2) si los obstáculos están a la izquierda y, por último, (3) si éstos se encuentran a la derecha. Mediante esta clasificación, se podrá controlar el robot móvil para que sea capaz de moverse sin colisionar de forma automática.

El uso de redes neuronales se ha popularizado en los últimos años, principalmente debido al potencial que tienen, pudiendo obtener resultados más satisfactorios que los conseguidos con otros algoritmos. Una de las principales tareas de las redes neuronales hoy en día es la clasificación de imágenes, ya que se puede elaborar un algoritmo que sea capaz de clasificarlas con un alto porcentaje de aciertos, haciendo que la programación de algoritmos de procesamiento de imágenes digital se vean limitados. Sin embargo, existen redes neuronales para un sinfín de tareas que, dependiendo de cuáles sean, el modelo de red que procesará los datos tendrá una estructura u otra.

En los próximos apartados de este capítulo, entenderemos los conceptos básicos de *Deep Learning* y de las redes neuronales. Específicamente, se estudiará la red neuronal más básica y simple, las *FeedForward Networks*, en cuyo funcionamiento se basan el resto de redes neuronales existentes.

Hay que destacar, que en el campo de la Inteligencia Artificial y, sobre todo, de las redes neuronales, es esencial el uso del inglés, ya que la mayoría de términos específicos se encuentran en este idioma; por ello, en este proyecto nos encontraremos con frecuencia con una jerga anglosajona para referirnos a dichos conceptos que, aunque probablemente tengan traducción al castellano, son términos que pueden inducir a error, pudiendo referirse a otros conceptos no relacionados con la Inteligencia Artificial y las redes neuronales.

Deep Learning

El concepto de *Deep Learning* es un concepto difícil de entender ya que su significado se ha ido modelando conforme han ido pasando los años. Hoy en día, para entender qué es exactamente, es necesario entender otros conceptos que la engloban, como es la Inteligencia Artificial y el *Machine Learning*.

La Inteligencia Artificial o IA (en inglés AI de *Artificial Intelligence*) es el estudio que trata ”dispositivos inteligentes”, es decir, dispositivos que perciben su entorno y toman decisiones que maximizan las posibilidades de éxito a la hora de conseguir sus metas. Estos dispositivos pueden ser cualquier entidad que use sensores para captar el mundo que le rodea y actuadores para realizar las acciones necesarias para conseguir sus objetivos. Un ejemplo muy claro de este tipo de dispositivos, son los robots, que en la mayoría de los casos son programados para hacer determinadas tareas, pero, sobre todo hoy en día debido a la evolución de la IA y a la complejidad de las tareas que deben hacer, se les ”enseña” para que realicen dichas tareas con la mayor tasa de éxito posible.

De hecho, en la actualidad, cuando usamos el concepto de Inteligencia Artificial, nos solemos referir a una máquina que es capaz de imitar funciones cognitivas que se suelen asociar a la mente humana, como el hecho de ”aprender” o ”resolver problemas complejos”. Algunos de los grandes logros de las IA en el año 2017 implican acciones como interpretar el lenguaje humano, jugar a juegos complejos como el ’Go’, coches autónomos... entre otros muchos más. Ha sido en el siglo XXI cuando la Inteligencia Artificial ha experimentado un gran resurgimiento, debido principalmente a la gran mejora de la potencia de computación y a la gran cantidad de datos e información existentes (gracias a Internet, por ejemplo), siendo ahora un

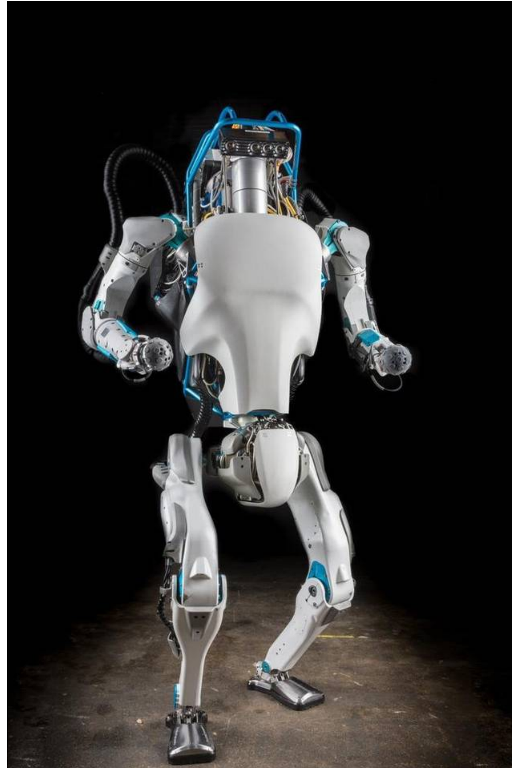


Figura 1.1 *Atlas*, de *Boston Dynamics*, es un gran ejemplo de cómo cada vez las tareas que se les encomiendan a los robots son más complejas.

campo esencial en muchos ámbitos tecnológicos.

El concepto de Inteligencia Artificial no es moderno; ya a mitad del siglo XX, Alan Turing empezó a popularizar el concepto de máquinas inteligentes y el campo de la Inteligencia Artificial. De hecho, las metas tradicionales de este campo incluyen hacer que una máquina sea capaz de razonar, planear, aprender, entender el lenguaje natural, percibir visualmente e incluso tener la capacidad de moverse y manipular objetos. Es por ello que la Robótica y la Inteligencia Artificial se unen cada vez más, ya que son campos que se complementan entre ellos, permitiendo conseguir con mayor facilidad la existencia de dispositivos que se acerquen al comportamiento humano, tanto cognitiva como fisicamente y así alcanzar una de las principales finalidades por las que se fundaron estos campos: imitar lo mejor posible a los seres vivos, no sólo humanos.

La evolución de la Inteligencia Artificial, tal y como la conocemos hoy día, se debe principalmente al desarrollo de otros subcampos de ésta, en concreto, del *Machine Learning*, y sobre todo gracias al *Big Data*. *Machine Learning* se ocupa del estudio de técnicas que permiten a las máquinas ”aprender” a partir de datos e información, sin ser explícitamente programados. Continuamente hemos estado hablando de ”aprender” y no aprender, y esto se debe a que las máquinas no poseen mentes, es decir, las máquinas siguen sin comprender, sin saber cuáles son sus propósitos, no saben pensar. Con ”aprender” nos referimos principalmente a que la máquina es capaz, por medio de ejemplos, refuerzos positivos y negativos o incluso reglas, mejorar su rendimiento y su precisión a la hora de realizar determinadas tareas con éxito a partir de la información que es capaz de recoger de su entorno.

Un ejemplo muy claro es la humanoide Sophia, una robot que posee reconocimiento facial y de voz, y es capaz de entender el lenguaje humano gracias al uso de la Inteligencia Artificial, y, en 2018, se le ha dotado con la capacidad de andar. Gracias a ello, Sophia, ha conseguido gran popularidad en el año 2017, ya que cuando se establecen conversaciones con ella, se puede tener la sensación de estar delante de un ser humano, debido a que posee expresiones faciales que se ajustan a la temática, es capaz de contestar con frases coherentes, de mantener la mirada sobre la persona con la que mantiene la conversación... y otras muchas caracteríticas que dotan a esta humanoide de un parecido asombroso al de un ser humano. Sin embargo, en realidad, se trata de un ejemplo algo más complejo de un *chatbot*, como lo es

Siri de Apple o Google Assistant de Google. Esto quiere decir que ella no entiende lo que escucha o ve, simplemente posee acciones predeterminadas y pre-programadas que usa según la situación en la que se encuentre, por lo que ella realmente no ha aprendido tal y como los humanos lo hacemos. Sin embargo, no deja de ser un gran avance tecnológico, porque esta humanoide es capaz de clasificar palabras y acciones cada vez con más exactitud, pudiendo elegir de forma más precisa qué expresión facial poner o qué frase decir.

Figura 1.2 Sophia, una robot humanoide con Inteligencia Artificial que se asemeja al comportamiento humano.

Los grandes avances en *Machine Learning* se deben principalmente a la necesidad de resolver problemas donde el diseño y programación de algoritmos explícitos con una buena precisión se hace extremadamente difícil o imposible. Por ello, conforme avanza la tecnología, se dan las condiciones perfectas para poder construir algoritmos que mediante entrenamiento por medio de grandes cantidades de información, son capaces de modelar el sistema y hacer predicciones con una alta precisión. Precisamente, los avances que se han producido en este campo se deben principalmente al desarrollo de *Deep Learning*, que fue a principios de la segunda década de este siglo cuando los métodos que se desarrollaron en este campo empezaron a tener una tasa de precisión muy superior a cualquier otro algoritmo de *Machine Learning*, estando en muchas ocasiones por encima de las capacidades humanas.

Expresamente, *Deep Learning* es un método de *Machine Learning* que se ocupa de modelar redes neuronales artificiales que están inspiradas en el funcionamiento del cerebro humano, aunque, irónicamente, los humanos seguimos sin comprender realmente cómo trabaja este órgano. *Deep Learning* hace uso de técnicas capaces de detectar de forma automática patrones en las características que diferencian los distintos datos de entrada para, finalmente, poder clasificarlos.

Principalmente, en *Deep Learning* se usa el método de aprendizaje del ser humano que consiste en aprender a base de ejemplos. Entonces, a estos algoritmos se les da un modelo a partir del cual pueden resolver ejemplos, además de un pequeño conjunto de instrucciones que les permite reajustar dicho modelo para resolver los problemas con mayor precisión. Sin embargo, éste no será el único método de aprendizaje

de los algoritmos de *Deep Learning*, aunque sí el más importante y el más usado.

Uno de los rasgos más importantes de *Deep Learning*, es que hace uso de múltiples capas que procesan la información de entrada, y así extrae las particularidades que distinguen los distintos datos, los cuales pueden tener una relación no lineal. Esto permite a las redes neuronales artificiales modelar sistemas muy complejos y no lineales a partir de solo ejemplos de entrenamiento con una eficacia y eficiencia que ningún otro sistema de *Machine Learning* es capaz de hacer. Es pues *Deep Learning* uno de los campos de Inteligencia Artificial más importantes e interesantes en la actualidad.

Redes Neuronales Artificiales

Como ya se ha mencionado, los modelos de *Deep Learning* se basan en redes neuronales artificiales, que no son más que sistemas computacionales inspirados en el funcionamiento del cerebro de animales, y por tanto, de humanos. Básicamente, una red neuronal artificial se forma cuando se comienzan a unir varias neuronas artificales a otras, donde las entradas de unas son las salidas de otras, hasta que finalmente las últimas neuronas corresponden con la respuesta de la red neuronal ante un problema concreto.

Para entender una red neuronal artificial, es útil entender cómo funciona el cerebro humano y más en concreto, las redes neuronales biológicas. Una red neuronal biológica está formada por neuronas las cuales se encuentran optimizadas para recibir información de otras neuronas, procesarla y enviarla a las siguientes. Para ello, éstas reciben la información por medio de sus dendritas, que dependiendo de cuántas veces se use cada dendrita, la información que fluye por ella se refuerza o debilita, de forma que cada entrada a la neurona influirá más o menos en su salida. Luego, en el cuerpo celular, todas esas entradas se suman para obtener un único valor que es transferido al axón de la neurona, que es la parte de ella que permite la conexión con otras neuronas mediante la sinapsis. La sinapsis no es más que una aproximación entre neuronas a partir de la cual se lleva a cabo la transmisión de impulsos nerviosos que transmiten la información de unas neuronas a otras. Estos impulsos nerviosos son provocados por la segregación de neurotransmisores (compuestos químicos) que son los encargados de excitar o inhibir la otra neurona con la que conecta dependiendo de la información que se ha transmitido por el axón. Mediante la unión de millones de estas células interconectadas, se consigue tener una red neuronal biológica.

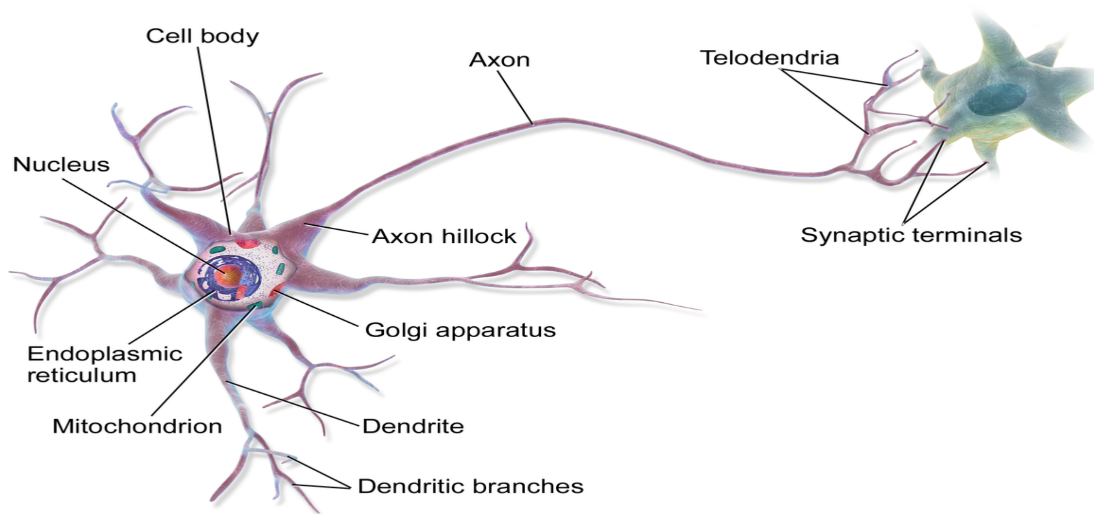


Figura 1.3 Representación gráfica de una neurona biológica.

Sabiendo el funcionamiento de las neuronas del cerebro, se ha realizado un modelo artificial para así poder formar una red neuronal artificial que pueda ser programada mediante un ordenador. Las neuronas artificiales funcionan matemáticamente de la misma forma: tienen unas entradas las cuales son ponderadas según su importancia, se suman todas esas entradas ponderadas para obtener un solo valor que pasa por una función de activación, y, finalmente, la salida de dicha función será la salida de la propia neurona que se transmitirá a

las siguientes neuronas.

Podemos deducir que la función matemática que modela una neurona es la siguiente:

$$a = g\Big(\sum_{i=1}^{m}(x_i w_i) + b\Big), \tag{1.1}$$

donde a representa la salida de la neurona, b es un valor de *bias* que se suele añadir a la suma de entradas, $g()$ es la función de activación de la neurona, x_i la i-ésima entrada de la neurona y w_i el peso, o más técnicamente llamado *weight*, de la conexión i-ésima. Además, hay que destacar que $\sum_{i=1}^{m}(x_i w_i) + b$ se suele definir como la entrada o la señal de activación de la neurona, representado como z.

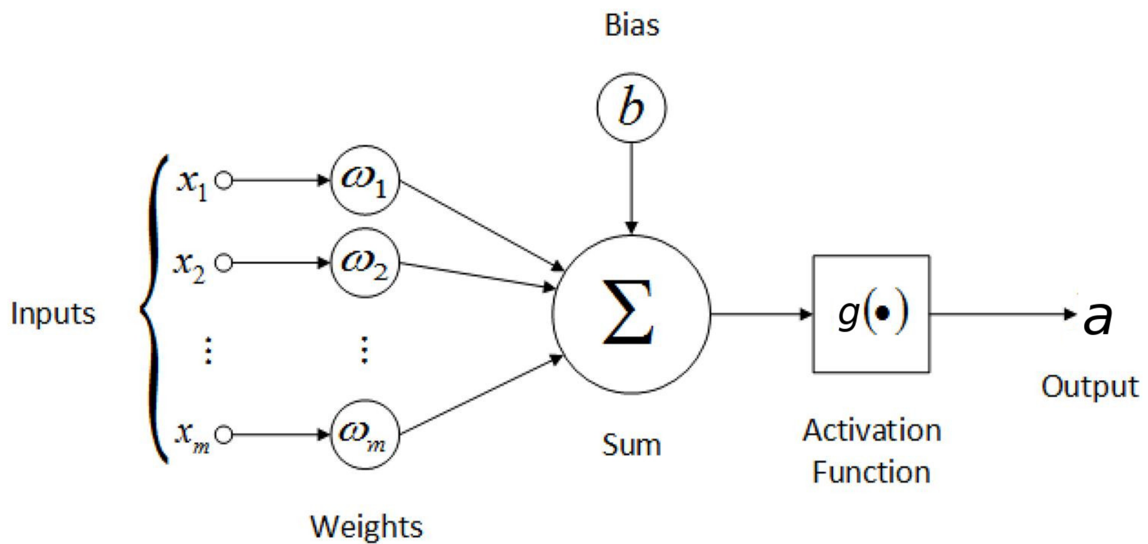


Figura 1.4 Representación gráfica de una neurona artificial.

Por tanto, una red neuronal artificial se basa en un conjunto de neuronas artificiales distribuidas en capas que se encuentran interconectadas entre ellas. Se puede deducir que en una red neuronal, quien capta las interacciones, características o patrones de los datos de entrada, son las neuronas, y cuantas más tenga, más interacciones podrá capturar dicha red. Como ya se ha mencionado, estas neuronas se encuentran distribuidas en una red neuronal artificial por medio de capas: la capa de entrada, la capa de salida y las intermedias, comunmente llamadas *hidden layers*. Cada una de las neuronas que conforman una capa se conectan con todas las neuronas de la siguiente capa, proceso que se repite hasta llegar a la capa final, la de salida, que nos aportará un resultado, dependiendo de los datos de entrada de esa red.

Cuando hablamos de neuronas en redes neuronales, se suelen denominar como ”nodos”, debido a que en los grafos computacionales como el de la Figura 1.5, las neuronas se representan como nodos interconectados.

Hoy en día, se han conseguido hacer redes neuronales de millones de neuronas (un número nada comparable con las que tiene el cerebro humano) por lo que se han conseguido tareas muy complejas en las que se ha llegado incluso a superar la precisión alcanzada por seres humanos, y por ello, las redes neuronales han podido abrirse hueco entre los mejores algoritmos de *Machine Learning*. Sin embargo, para hacer redes neuronales tan complejas, no es suficiente una red que posea sólo tres capas de neuronas (*input*,*hidden* y *output layers*), por ejemplo, el *cortex* cerebral humano posee seis capas formadas por millones de neuronas cada una. Este tipo de redes neuronales se denominan *Deep Neural Networks* cuya característica principal es que poseen más de una *hidden layer*, de forma que son capaces de construir internamente representaciones de los patrones que caracterizan los datos de entrada, y cuanto más nos adentramos a capas más profundas, más específica y compleja es la característica o patrón que modela. De ahí, que el uso de *Deep Neural Networks* se haya popularizado tanto, porque son capaces de modelar sistemas realmente complejos, y más importante, sistemas no lineales. Es común usarlas en tareas de clasificación complejas, como la clasificación de imágenes o sonido. Para ello, es usual construir redes neuronales con capas cuyo número de neuronas va

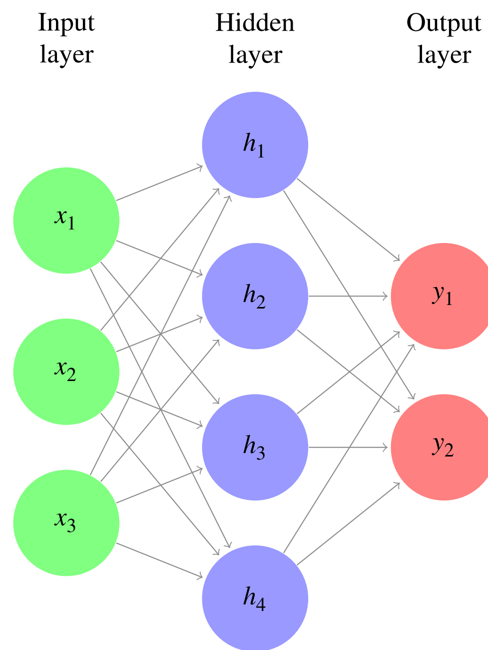


Figura 1.5 Red neuronal artificial. Podemos observar las tres capas: *input layer*, *hidden layer* y *output layer*.

en disminución conforme nos acercamos a la capa de salida, normalmente, para comprimir la información que va obteniendo cada capa hasta reducirla al número de clases o categorías a clasificar.

Sin embargo, las redes neuronales artificiales hay que entrenarlas, y para ello, en *Deep Learning* nos basaremos la mayoría de las veces en el ejemplo. El entrenamiento consiste en encontrar los valores óptimos de los parámetros que forman la red neuronal, principalmente *weights* y *biases*, aunque pueden existir más parámetros entrenables. El conjunto de parámetros que pueden ser entrenados se denomina vector de parámetros θ. Para entrenar estos parámetros, lo podemos hacer por tres vías:

- **Entrenamiento supervisado:** se basa en dar ejemplos a la red neuronal compuestos por las entradas a ésta y por las respectivas salidas deseadas de la red para cada entrada. En tareas de clasificación, se le aporta a la red neuronal ejemplos ya clasificados (*labeled data*), es decir, cada ejemplo estará clasificado por medio de una etiqueta (*label*). De esta forma, la red neuronal podrá calcular el error que tiene en sus predicciones respecto la salida ideal, y modificar sus parámetros para disminuir dicho error.
- **Entrenamiento no supervisado:** a diferencia del entrenamiento supervisado, en este caso se entrena a la red neuronal con datos no clasificados, con lo que ahora no hay forma de calcular la precisión y el error, haciendo el problema de entrenamiento mucho más complejo.
- **Entrenamiento por refuerzo:** se basa en enseñar a la red neuronal sin datos de ejemplos, por lo que la red neuronal hace una predicción a partir de datos que captura del entorno, y por medio de observación se genera un valor de error instantáneo que permite saber si la predicción de la red ha sido correcta o errónea.

Lo más común es el entrenamiento supervisado, ya que con cálculos de error y el algoritmo de *Backpropagation* que permite calcular el error introducido por cada nodo de la red, se pueden ajustar los valores de las variables que modelan la red neuronal por medio de otros algoritmos, como *Gradient Descent*, posiblemente el más sencillo de todos. Hay que destacar, que las variables que modelan una red neuronal, en la gran mayoría de casos son los valores de los *weights* de todas las conexiones existentes y los valores de los *bias* de todos los nodos de la red, sin incluir evidentemente los de entrada. Por ello, uno de los mayores retos a la hora de hacer redes neuronales eficientes, es conseguir grandes cantidades de datos ya clasificados para poder entrenarla.

Según el tipo de dato que se vaya a usar en la red neuronal o cómo se vaya a entrenar, es necesario elegir el tipo de *Deep Neural Network* acorde a ello. Existen muchas *Deep Neural Networks*, como la *Autoenconders* que es habitual para entrenamientos de tipo no supervisado; *Recurrent Neural Network* que suele ser usada

para reconocimiento de caracteres escritos o de habla; y una de las más famosas, la *Convolutional Neural Network*, que veremos durante el proyecto, ya que es la red neuronal más eficiente en el tratamiento de imágenes. Sin embargo, la primera red neuronal y más simple es la llamada *FeedForward Network*, que estudiaremos más profundamente en el siguiente apartado.

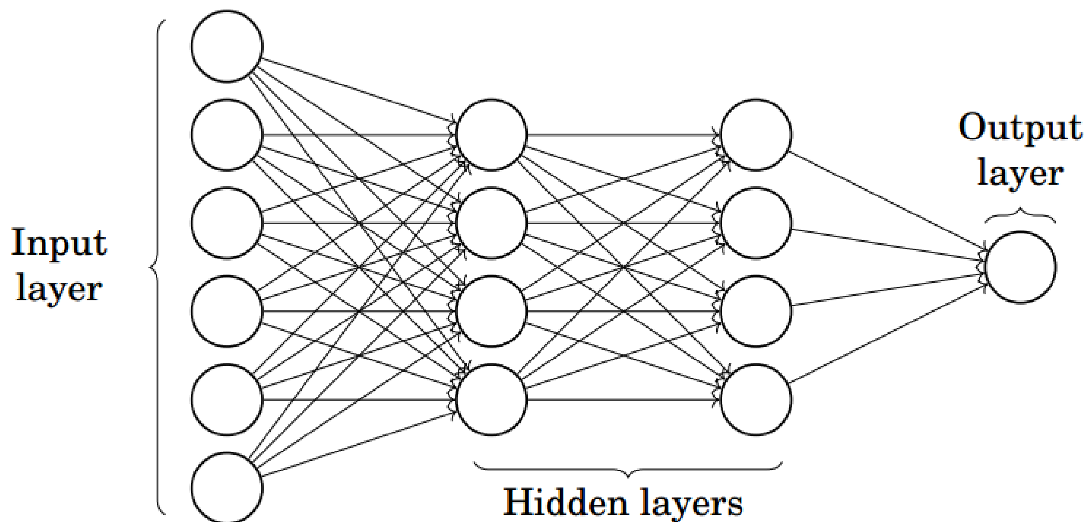


Figura 1.6 Esquema de una *FeedForward Network* que posee varias *hidden layers*.

FeedForward Neural Networks

Estas redes neuronales son las más simples y las primeras que se idearon. Tienen la característica de que la información sólo fluye en una dirección, de la capa de entrada a la capa de salida. Cuando hablamos de *FeedForward Neural Network*, nos solemos referir a la *Deep Neural Network*, que como ya se ha mencionado, este tipo de redes neuronales se caracteriza por poseer dos o más *hidden layers*.

Las *FeedForward Neural Networks* que se van a estudiar en los próximos apartados, se caracterizan porque todas sus capas, excepto la de entrada, poseen neuronas con funciones de activación no lineales, además, el tipo de entrenamiento usado es el supervisado por medio de ejemplos, utilizando el algoritmo de *Backpropagation* para el cálculo de errores, y el de *Gradient Descent* para optimizar la red. Por tanto, uno de los objetivos de este tipo de redes neuronales es la capacidad de separar información que es dificilmente clasificable debido a su naturaleza no lineal.

En los próximos apartados aprenderemos todos los conceptos de este tipo de redes neuronales y a programarlas mediante Python y una de sus librerías más esenciales, NumPy, que permite el uso de arrays y matrices de una forma sencilla.

Forward Propagation

En apartados anteriores hemos estudiado el comportamiento de una neurona artificial, que como ya sabemos la operación matemática que realizan intrínsecamente viene dada por la ecuación 1.1. Mediante esta fórmula matemática, podremos relacionar los valores de una capa de la red neuronal con la siguiente, de forma que podemos crear fácilmente una *FeedForward Neural Network* y hallar la salida o predicción de dicha red neuronal uilizando el algoritmo de *Forward Propagation*. Para ello, comencemos con una red neuronal sencilla como la que aparece en la Figura 1.7, con sólo dos *hidden layers*.

Tal y como vemos en dicha Figura, el hecho de contener varios nodos por capa hace inviable e ineficiente hallar individualmente el valor de cada uno de los nodos que forman la red neuronal. Por ello, es necesario el uso de NumPy, que nos permitirá utilizar matrices y arrays de forma sencilla, de manera que podemos

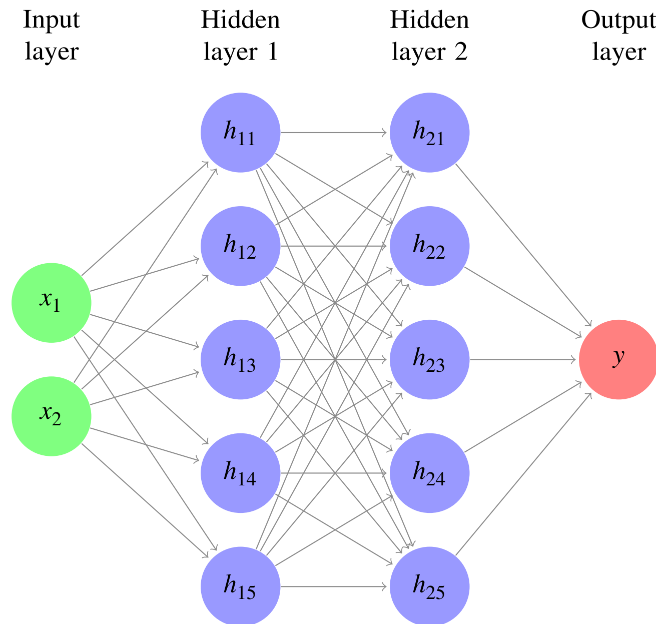


Figura 1.7 Esquema de una *Deep Neural Network* de tipo *FeedForward*.

tratar la red por medio de capas y no de nodos. Para esto, en primer lugar, hay que establecer los tamaños y formatos de las matrices que intervendrán en el cálculo de los valores de cada uno de los nodos de cada capa.

Supondremos que el valor de cada capa de la red estará dado por un array de valores, es decir, por una matriz de tamaño $(1 \text{ x } n^l)$, siendo n^l el número de nodos que forman dicha capa. Por ejemplo, para la capa de entrada en nuestro caso, tendría el siguiente formato:

$$\mathbf{a^{in}} = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$$

Teniendo esto, podremos hallar los formatos de las matrices que representen los valores de las siguientes capas para que la ecuación 1.1 se cumpla. Para multiplicar matrices, lo más común es el uso del producto escalar, lo que nos permitirá hallar prácticamente de inmediato los valores de cada uno de los nodos de la capa, ya que poseemos el mismo proceso de suma-multiplicación que tiene la ecuación 1.1. Para ello, la matriz de *weights* de cada capa deberá tener un tamaño de $(n^{l-1} \text{ x } n^l)$, es decir, cada columna de la matriz corresponderá con los *weights* para las entradas de un nodo de la capa de la que queremos calcular sus valores, y cada fila, corresponderá con los *weights* aplicados al valor de cada nodo de la capa anterior. Por ejemplo, para *hidden layer 1* de la Figura 1.7 tendría el siguiente formato:

$$\mathbf{w^{l1}} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{pmatrix}$$

De esta forma cuando multipliquemos mediante el producto escalar los datos de entrada con los *weights* de la primera capa, obtendremos una matriz con formato $(1 \text{ x } n^{l1})$, tal y como se deben representar los valores de todos los nodos de una capa. Por último, nos faltaría añadir el valor de *bias* para cada nodo de la capa. Dicha matriz es sencilla, ya que el valor de *bias* se debe añadir al final, por lo que deberá tener un formato de $(1 \text{ x } n^{l1})$. Finalmente, podríamos calcular la salida de una capa de la siguiente forma:

$$a^l = g^l(a^{l-1}w^l + b^l), \tag{1.2}$$

donde l es la capa actual y $l-1$ la capa anterior.

De este modo se puede usar una nueva notación para hallar la salida, no de una neurona, sino de una capa al completo usando matrices y arrays. Por ello, se puede proceder a realizar un código que permita el cálculo de una salida de una red neuronal de forma eficiente:

```
import numpy as np
```

```
# Se define el numero de entradas y de salidas
input_data_size = 2
n_classes = 1

# Se define el modelo de red neuronal
layer1_nodes = 5
layer2_nodes = 5

# Se inicializan los parametros
weights_layer1 = np.random.rand(input_data_size , layer1_nodes)
biases_layer1 = np.random.rand(1,layer1_nodes)

weights_layer2 = np.random.rand(layer1_nodes , layer2_nodes)
biases_layer2 = np.random.rand(1,layer2_nodes)

weights_output = np.random.rand(layer2_nodes , n_classes)
biases_output = np.random.rand(1, n_classes)

# Se define la funcion de la red neuronal
def neural_network(input_data):

    layer1_out = np.add(np.dot(input_data , weights_layer1) , biases_layer1)

    layer2_out = np.add(np.dot(layer1_out , weights_layer2) , biases_layer2)

    prediction = np.add(np.dot(layer2_out , weights_output) , biases_output)

    return prediction

# Se crea un array de datos de entrada de ejemplo
data = [[1 , 2]]

# Prediccion de la red neuronal
print('Prediction: ' + str(neural_network(data)))
```

Hay que destacar varios puntos del código anterior. En primer lugar, si nos fijamos, los valores de las variables *weights* y *biases* se han inicializado de forma aleatoria. Esto se debe a que en un inicio, no nos importa los valores que posean dichas variables, ya que la intención en el entrenamiento de la red neuronal, es ajustar dichos valores para que, ante determinados datos de entrada, tengamos una salida determinada. El hecho de empezar con valores aleatorios, también implica que según qué valores posean estas variables, a veces la red neuronal se entranará mejor y en otras ocasiones peor, aunque esto lo estudiaremos más tarde mediante el método de *Gradient Descent*.

En segundo lugar, hay que destacar que en el código anterior no se está haciendo uso de funciones de activación, de hecho, el tipo de neuronas del que se está haciendo uso son las llamadas neuronas lineales. Sin embargo, el uso de este tipo de neuronas puede tener serias limitaciones. Se ha demostrado que una *FeedForward Network* que sólo posea neuronas lineales se puede expresar como una red neuronal sin *hidden layers*, es decir, como una red neuronal con sólo dos capas: la capa de entrada y la capa de salida. Como ya se comentó, son las *hidden layers* las que permite a la red neuronal descubrir y captar las características complejas de los datos de entrada, por lo que es esencial el uso de otros tipos de neuronas no lineales para que la red neuronal pueda ”aprender” relaciones complejas.

Funciones de activación

Existen una gran cantidad de funciones de activación, sin embargo, hay sólo tres tipos que son comunmente utilizados según la aplicación y necesidades de la red neuronal. Estas funciones de activación se basan en el

impulso eléctrico de las neuronas biológicas para transportar la información de unas a otras.

En primer lugar nos encontramos con las neuronas *logistic* o más comunmente llamadas sigmoidales, que se caracterizan porque la función de activación tiene forma de ”S”. Este tipo de neuronas se basa en la teoría probabilística y ha sido una de las más usadas. La función de activación es la siguiente:

$$f(z) = \frac{1}{1+e^{-z}}, \tag{1.3}$$

donde z corresponde con la salida de la neurona lineal. Esta función se hace denominar *logistic function*, de ahí el nombre de este tipo de neurona.

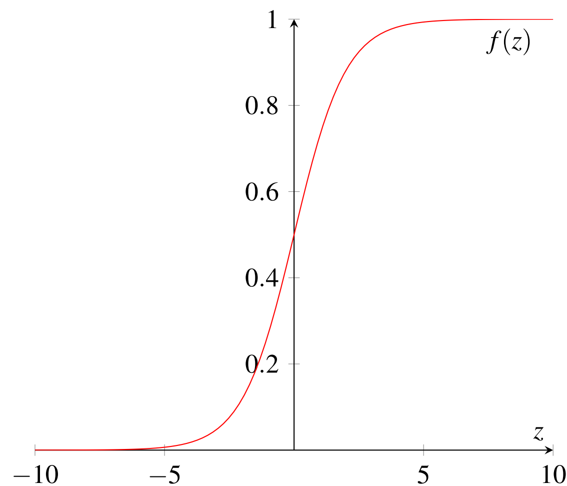


Figura 1.8 Salida de una neurona sigmoidal según varía z.

Otro tipo de neurona con una función con forma de ”S” como la sigmoidal, es la neurona *tanh* (tangente hiperbólica), pero en este caso, la función se encuentra centrada en cero. Normalmente, este tipo de neuronas es más utilizado que las sigmoidales debido a que estas funciones, que son antisimétricas respecto del origen, causan una convergencia mucho más rápida con métodos de entrenamientos basados en *Backpropagation*. La función usada por este tipo de neuronas es:

$$f(z) = tanh(z) \tag{1.4}$$

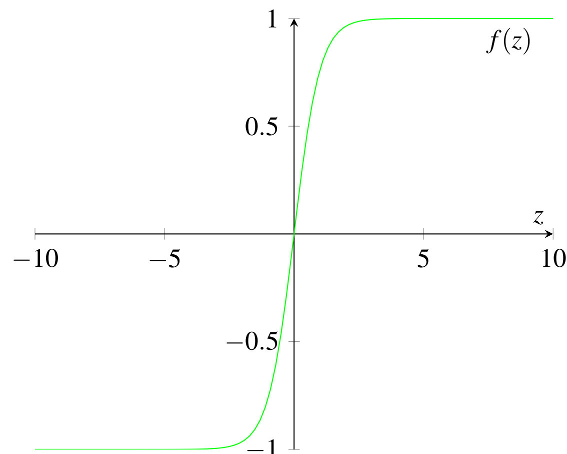


Figura 1.9 Salida de una neurona *tanh* según varía z.

Por último, nos encontramos con las neuronas de tipo ReLU (*Restricted Linear Unit*). Son las más

utilizadas en los últimos años en redes neuronales, ya que en 2011 se demostró que permiten a la red un mejor entrenamiento en casos de *Deep Neural Networks* comparado con las neuronas de tipo sigmoidal o de tangente hiperbólica. La función que rige el comportamiento de este tipo de neuronas es la siguiente:

$$f(z) = max(0,z) \tag{1.5}$$

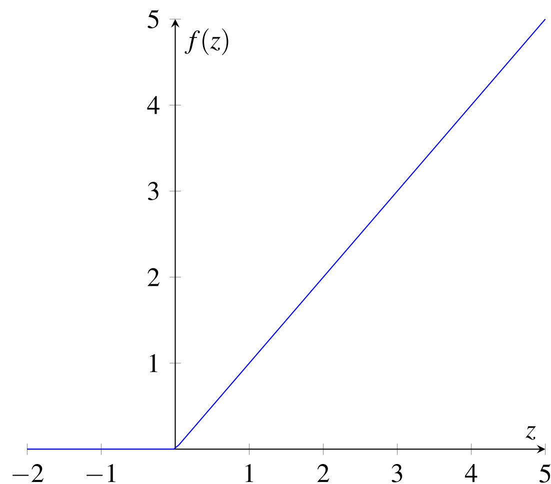


Figura 1.10 Salida de una neurona ReLU según varía z.

Como ya veremos más adelante, existe la posibilidad de que las neuronas que posean este tipo de función de activación, mueran, es decir, se vuelven inservibles impidiendo ser entrenadas y teniendo siempre un valor de salida nulo, y habría que evitarlas para impedir que éstas estropeen el entrenamiento de la red. Para ello se suelen promover distintas soluciones, pero la más básica es el uso de *Leaky ReLUs*, cuya función se modifica de la siguiente manera:

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ mz & \text{if } z \leq 0 \end{cases} \tag{1.6}$$

en el que m es un valor que dependiendo del método puede ser constante con un valor positivo, o un parámetro de aprendizaje que se ajusta junto con el resto de parámetros de la red neuronal.

Siguiendo con el ejemplo que se propuso en la Figura 1.7, se va a realizar el código para calcular la predicción de una red neuronal que posea neuronas de tipo ReLU:

```
import numpy as np

# Se define el numero de entradas y de salidas
input_data_size = 2
n_classes = 1

# Se define el modelo de la red neuronal
layer1_nodes = 5
layer2_nodes = 5

# Se inicializan los parametros del modelo
weights_layer1 = np.random.rand(input_data_size , layer1_nodes)
biases_layer1 = np.random.rand(1,layer1_nodes)

weights_layer2 = np.random.rand(layer1_nodes , layer2_nodes)
biases_layer2 = np.random.rand(1,layer2_nodes)

weights_output = np.random.rand(layer2_nodes , n_classes)
```

```
biases_output = np.random.rand(1, n_classes)

# Se define la funcion de activacion ReLU
def relu(input):
    n_elements = np.size(input)

    out = np.empty([1 , n_elements])
    i=0

    # Se calcula el valor de la salida de la neurona ReLU
    for _ in range(n_elements):
        out[0,i] = max(0, input[0,i])
        i+=1

    return out

# Se define la funcion de la red neuronal
def neural_network(input_data):

    z1 = np.add(np.dot(input_data , weights_layer1) , biases_layer1)
    layer1_out = relu(z1)

    z2 = np.add(np.dot(layer1_out , weights_layer2) , biases_layer2)
    layer2_out = relu(z2)

    prediction = np.add(np.dot(layer2_out , weights_output) , biases_output)

    return prediction

# Se crea un array de entrada
data = [[1 , 2]]

# Prediccion del modelo
print('Prediction: ' + str(neural_network(data)))
```

Se puede ver en el código superior la implementación de *hidden layers* con neuronas que tienen funciones de activación, sin embargo, se observa que en la *output layer* se han implementado neuronas lineales, algo que suele ser común en las redes neuronales sobre todo en tareas de clasificación.

Softmax Layer

Como se ha comentado anteriormente, es frecuente encontrar las capas de salida de las redes neuronales sin función de activación. Sin embargo, existe un tipo especial de *output layer* para casos de clasificación: la llamada *softmax layer*.

Esta capa es útil en casos de clasificación en los que las etiquetas de cada uno de los ejemplos se encuentran en un formato especial: *one-hot encoding*, es decir, cuando cada clase a la que pertenece cada uno de los ejemplos excluye al resto. Por ejemplo, si se trata de clasificar imágenes que poseen números del 0 al 9 (como es el caso de clásico *dataset* MNIST), tendremos 10 posibles clases a clasificar, sin embargo, estas clases son mutuamente excluyentes, ya que si se clasifica la imagen como un '5', ya no puede ser posible que sea ninguno de los 9 números restantes. La salida de la red neuronal (y por tanto las etiquetas de cada clase) tendrían la siguiente forma:

$$
\begin{aligned}
0 &\rightarrow 1000000000 \\
1 &\rightarrow 0100000000 \\
2 &\rightarrow 0010000000 \\
3 &\rightarrow 0001000000 \\
&\vdots
\end{aligned}
$$

Normalmente, no se obtendrá como salida los resultados expuestos arriba, ya que con este tipo de capas se obtiene un resultado probabilístico de las posibilidades de que la imagen sea un número concreto, de forma $\sum_{i=0}^{9} p_i = 1$, por lo que las salidas que se han mostrado pertenecerían a una red neuronal perfecta que sería capaz de clasificar sin error alguno todas las imágenes. Por tanto, como la salida de una neurona de una *softmax layer* depende de las predicciones del resto de neuronas, podremos calcular la predicción de la neurona k como:

$$a_k = \frac{e^{z_k}}{\sum_j e^{z_j}}, \tag{1.7}$$

donde j corresponde al resto de neuronas que forman la capa de salida, y z es la señal de activación de las neuronas.

Como veremos más tarde, este tipo de capas son muy usadas ya que el objetivo principal de las redes neuronales consiste en la clasificación de información, por lo que la notación de tipo *one-hot* ayuda mucho a lograr dicha meta.

Entrenamiento de Redes Neuronales

El entrenamiento de una red neuronal es esencial para encontrar los valores óptimos del vector de parámetros θ, formado principalmente por *weights* y *biases*, aunque siempre podrán existir más parámetros que permitan el aprendizaje de la red neuronal. En este apartado abordaremos el problema del aprendizaje supervisado, es decir, para entrenar la red neuronal se le presentarán a ésta una serie de ejemplos de entrenamiento y de forma iterativa se modificarán los parámetros de la red para minimizar el error que está cometiendo en dichos ejemplos.

Existen muchos algoritmos que permiten entrenar a una red neuronal mediante ejemplos, aunque la mayoría de ellos se basan en el algoritmo de entrenamiento *Gradient Descent* y en el algoritmo de *Backpropagation* para el cálculo de errores, los cuales veremos en los próximos apartados. Algunos de estos algoritmos son *Momentum*, *NAG*, *Adagrad*, *Adadelta*, *Adam* o *Rmsprop*, que además de ser mucho más eficientes y rápidos, son capaces de evitar algunos problemas que posee el algoritmo de *Gradient Descent*, como el estancamiento en puntos de silla.

A pesar de ello, en los próximos apartados vamos a estudiar el método básico de *Gradient Descent* para entender el funcionamiento principal del entrenamiento de redes neuronales.

Gradient Descent

En primer lugar, para entrenar la red neuronal, debemos saber cuál es el error que ha tenido a la hora de hacer una predicción para un ejemplo de entrenamiento. En estos casos, sabemos la salida real que se ha obtenido de la red neuronal y la salida que se desea tener, con lo que se puede calcular fácilmente el error. Dicho error se calcula mediante unas funciones específicas llamadas *loss functions* o *cost functions*, las cuales devuelven un único valor sobre el ”coste” o error de la red neuronal ante un determinado evento. Debido a que nos encontramos ante un problema de optimización, el objetivo del entrenamiento de la red neuronal es minimizar el valor de la *loss function*, es decir, del error. Existen muchas *loss function*, como *Cross Entropy,*

Hinge, Kullback Leibler, Poisson... aunque la función más usada es aquella que utiliza la función del error cuadrático como medida de coste, llamada *Quadratic Loss Function*:

$$E = C(t-y)^2, \tag{1.8}$$

donde C es una constante que se suele establecer en un valor de $1/2$ (para simplificar más tarde su derivada), aunque dicho valor no hace modificaciones ante una decisión; t es el valor esperado de la salida en el ejemplo de entrenamiento (*targets*) e y es el valor predicho por la red neuronal. Esta *loss function* lo que permite es calcular el error cuadrático ante los ejemplos de entrenamiento y así, obtener un sólo valor que indique el error o coste de la red neuronal en un único ejemplo.

Por tanto, cuando el error cuadrático se hace nulo para un ejemplo de entrenamiento, quiere decir que el modelo de red neuronal ha sido capaz de obtener la salida deseada con total precisión. Normalmente, no es posible conseguir tal precisión en todos los ejemplos de entrenamiento, pero, a partir de la optimización del vector de parámetros θ, se puede llegar a una precisión muy alta con buenos modelos de redes neuronales.

En primer lugar, supongamos que nuestro vector de parámetros para una neurona lineal está formado por solamente dos *weights* y ningún *bias*. De esta forma, podremos imaginar cómo el error que introduce dicha neurona cambia según los valores de dichos *weights*.

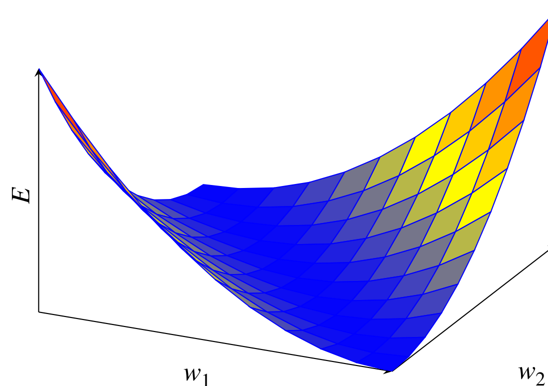


Figura 1.11 Error cuadrático de una neurona lineal con dos *weights* y sin *bias*.

Como se puede observar en la Figura 1.11, el error cuadrático que añade una neurona lineal es mínimo en determinados valores de los parámetros. Es sencillo pensar que, empezando con cualquier valor de las variables w_1 y w_2, implica que se empezará con un determinado error, pero, en ese punto en el que nos encontramos, será posible calcular la pendiente de la función del error cuadrático y por tanto ir modificando ambas variables para movernos por la superficie hasta hacer el error mínimo.

En la Figura 1.12 se puede observar claramente las líneas de contorno del error cuadrático de la neurona lineal y que su valor mínimo se encuentra en las zonas interiores de los contornos. Esta Figura es mucho más representativa ya que se puede observar la pendiente de la función de error cuadrático mediante las distancias entre los contornos, de forma que, cuanto más cerca esté un contorno de otro, mayor será la pendiente. Además, la dirección que indica la mayor pendiente de la función es aquella que es perpendicular a los contornos tal y como se ve en dicha figura mediante los vectores de color azul. Esta dirección la indica un vector llamado gradiente, que se calcula mediante las derivadas parciales de la función respecto las variables que la componen. Por ejemplo, si calculamos el gradiente de la función $f(x,y)$, el gradiente resultará como un vector de 2 elementos:

$$\bigtriangledown f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{1.9}$$

Con esto, podremos desarrollar una estrategia que nos permita disminuir el error en una red neuronal. Por ejemplo, supongamos que empezamos en cualquier punto de la función de error con valores aleatorios de los parámetros. Si se evalúa el gradiente respecto a ambos *weights* en la posición en la que nos encontramos,

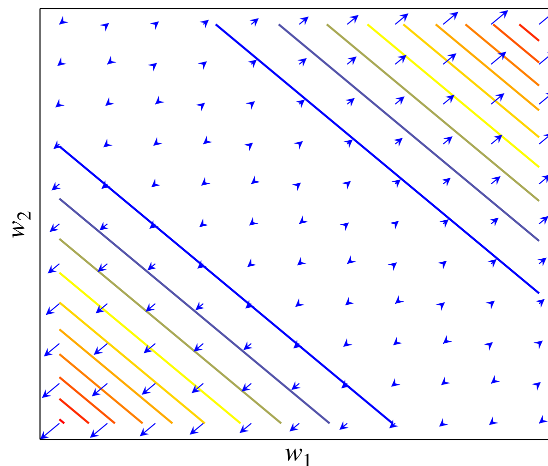


Figura 1.12 Contornos del error cuadrático de una neurona lineal junto con su gradiente.

tendremos la dirección más óptima a la que nos debemos mover para disminuir el error, ya que con el gradiente podremos modificar dichos parámetros la cantidad necesaria para desplazarnos en aquella dirección en la que la pendiente de la función de error sea mayor. Luego, nos encontraremos en otro punto, en el que realizaremos el mismo proceso para encontrar de nuevo la dirección con una mayor pendiente, hasta que, finalmente, lleguemos a un punto en el cual no se podrá disminuir más el error. Este algoritmo se denomina *Gradient Descent*, que aunque en este ejemplo se ha razonado para disminuir el error de una sola neurona, se podrá usar de la misma forma para entrenar redes neuronales al completo en un entorno multidimensional.

Backpropagation

El algoritmo de *Gradient Descent* es quizás el algoritmo de entrenamiento más utilizado y sencillo que se puede encontrar. Sin embargo, el cálculo de derivadas respecto cada uno de los parámetros que intervienen en la red neuronal no es eficiente, ya que normalmente se tienen miles de estos parámetros con derivadas muy complejas, es decir, nos encontramos en un espacio multidimensional en el que el error cuadrático depende de una gran cantidad de parámetros. Por ello, el problema de entrenar *Deep Neural Networks* lo resolvieron David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams en 1986 mediante el algoritmo de *Backpropagation*, algoritmo que se ha convertido en la base de los algoritmos de aprendizaje en redes neuronales artificiales.

El algoritmo de *Backpropagation* nos permitirá calcular el gradiente de la función del error de una forma sencilla y rápida. La finalidad de este algoritmo es poder calcular el error que introducen todas las neuronas que forman la red neuronal para, así, hallar el gradiente de la función del error y la dirección con mayor pendiente, para disminuir lo más rápidamente posible dicho error mediante el cambio de valor del vector de parámetros θ.

En concreto, debido a que el valor de una sola neurona puede afectar al valor de muchas neuronas de las capas que la prosiguen, el error que introduce una de estas neuronas también afectará al error final. Por ello, sabiendo el error que tenemos en la salida se puede ir transmitiendo hacia capas anteriores para hallar así el error que introduce cada una de ellas, es decir, propagamos el error hacia atrás, de ahí el concepto de *Backpropagation*. Una vez se tiene el error que introduce cada nodo de cada capa, es sencillo obtener cuánto se debe cambiar cada uno de los parámetros de la red para disminuir el error en cada capa.

Debido a que normalmente el vector de parámetros θ está formado por *weights* y *biases*, al final de este apartado, seremos capaces de entender cómo se pueden hallar los gradientes respecto cualquiera de estos parámetros que intervenga en nuestra red neuronal. Por ello, lo que queremos calcular es lo siguiente:

$$\frac{\partial E}{\partial w} \quad , \quad \frac{\partial E}{\partial b}, \tag{1.10}$$

expresiones que indican cuán rápido cambia el coste cuando cambiamos el valor de *weights* o *biases*.

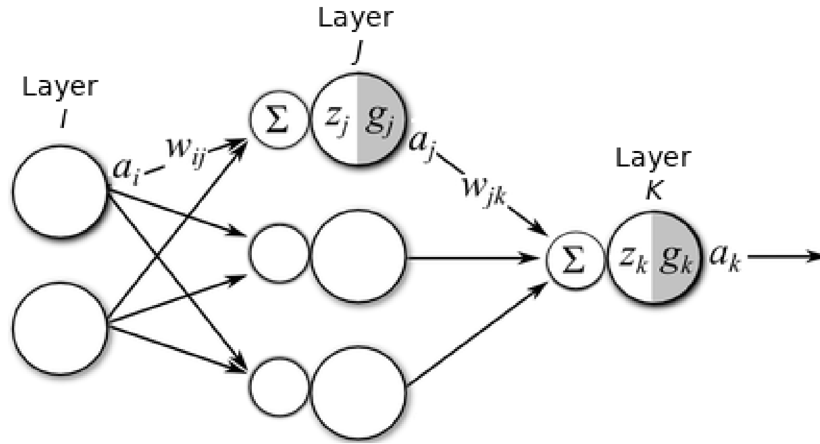


Figura 1.13 Red neuronal con una única *hidden layer* con la notación por elementos usada (los nodos de la capa de entrada se nombran con el subíndice i, los de la *hidden layer* con j y los de la capa de salida con k).

Antes de continuar con la explicación, se va a destacar los dos tipos de notación que se van a usar, ya que se puede usar una notación por elementos tal y como se observa en la figura 1.13, que puede ser útil en determinadas ocasiones para entender las ecuaciones pero puede resultar también un tanto ambigua. Por ello, se va a usar también una notación matricial tal y cómo se empezó a usar en el apartado 1.4.1, que no sólo va ayudar a simplificar las ecuaciones, sino que también va a ser útil para implementar dichas ecuaciones en un código eficiente. Por ejemplo, para expresar la salida de una capa de la red neuronal, se puede usar la notación matricial de la siguiente forma:

$$a^l = g(z^l) \quad ,\text{donde } z^l = (a^{l-1}w^l) + b^l \tag{1.11}$$

donde l se refiere a la capa actual de la cual queremos calcular el valor de sus nodos, y $l-1$ es la capa anterior. Si esta misma ecuación la representamos en una notación de elemento por elemento tal y cómo aparece representado en la Figura 1.13, la ecuación se complica pudiendo representarse de la siguiente forma:

$$a_k = g_k(z_k) \quad ,\text{donde } z_k = \sum_{j\in J}\left(a_j w_{jk}\right) + b_k \tag{1.12}$$

donde k representa el nodo k-ésimo de la capa K (en este caso la capa de salida) y j representa el nodo j-ésimo de la capa J (en este caso es la *hidden layer*). Si nos fijamos, esta ecuación es equivalente a la ecuación 1.1 que se explicó en en el apartado 1.3, sólo que esta vez las entradas de las neuronas, al no estar aisladas sino unidas a una red de neuronas conectadas, corresponden con las salidas de las neuronas de la capa anterior.

Para evitar futuras aclaraciones, se puede observar que en la notación matricial se utilizan superíndices que indican la capa con la que se está trabajando, mientras que con la notación de elementos, se utiliza subíndices para referirse a una neurona concreta de una capa en concreto, siempre haciendo referencia a la Figura 1.13.

Sabiendo esto, hay que continuar aclarando más conceptos. En concreto, se debe entender bien cómo funciona la *Quadratic Loss Function* que se vió en el apartado anterior. En primer lugar, hay que destacar que esta función debe devolver un único número que indique el error total de la red neuronal ante un ejemplo de entrenamiento, sin embargo, cabe la posibilidad de que exista más de un nodo de salida. Por ello, la *loss function* pasa a ser la siguiente:

$$E = \frac{1}{2}\sum_{k\in K}\left(t_k - a_k\right)^2, \tag{1.13}$$

donde K es la capa de salida. Por tanto, el coste de una red neuronal se establece como la suma de los errores cuadráticos de cada nodo de salida para así obtener un único dato numérico.

Por último, otro concepto a aclarar de la *loss function* y el algoritmo de *Backpropagation*, es que la *loss function* se suele aplicar a un conjunto de ejemplos de entrenamiento, llamado *training dataset*, sin embargo, el algoritmo de *Backpropagation* sólo se puede utilizar para un único ejemplo de entrenamiento. Debido a ello, la *cost function* para un *dataset* completo se suele representar como:

$$E = \frac{1}{n}\sum_{x}^{n}(E_x), \tag{1.14}$$

donde n es el número de ejemplos de entrenamiento. Es decir, simplemente se suele hacer la media aritmética para hallar el error en un conjunto de ejemplos, es decir, se utiliza el error cuadrático medio. Como ya veremos más adelante, esto nos resultará útil a la hora de entrenar de forma eficiente la red neuronal mediante *minibatches*.

Con todos estos conceptos aclarados, podemos volver a la explicación del algoritmo de *Backpropagation*. Antes de calcular las derivadas presentadas en la ecuación 1.10, es necesario presentar otro valor, δ^l, que es el error que introduce la capa l. Este valor no es más que un array de valores de cada uno de los errores que introduce cada nodo que forma dicha capa. *Backpropagation* lo que nos permitirá es calcular ese valor para cada una de las capas que forman la red neuronal de una forma sencilla, y a partir de este valor podremos calcular el gradiente de la *cost function* respecto el vector de parámetros.

Ahora bien, hay que entender cómo se halla el error de cada nodo que interviene en la red. Para ello, se puede pensar que el error que introduce un nodo se debe al valor de salida de dicho nodo. De esta forma, se puede deducir que el cambio en el coste de la red neuronal depende de las salidas de cada nodo en cada capa:

$$\delta_j = \frac{\partial E}{\partial a_j} \tag{1.15}$$

donde δ_j es el error que introduce la neurona j de la capa J. Entonces, en esta ecuación estamos definiendo cuánto cambia el error cuando cambia la salida de la neurona j, es decir, qué error introduce dicha neurona. Por ello, el hecho de que la neurona introduzca un error en su salida a_j, hace que este error se propague por las neuronas que la prosiguen, provocando un error en la salida de la red neuronal.

Por tanto, el error que introduce una capa depende de las salidas de los nodos, definiéndose el error de una capa como el gradiente de la *loss function* respecto las salidas de los nodos de la capa. A partir del error que introduce cada capa se puede hallar el gradiente de la *loss function* respecto cada uno de los parámetros de entrenamiento de dicha capa.

Para ello, en primer lugar debemos calcular δ^{out}, que no es más que el error que introduce la capa de salida de la red neuronal. Esto se hace porque sólo podremos calcular el error cuadrático en la capa de salida, ya que, de los ejemplos de entrenamiento, sólo tendremos qué salida deseamos tener para una determinada entrada en la red neuronal. Empecemos en primer lugar hallando el error de una única neurona de la capa de salida, pudiéndose representar por:

$$\delta_k = \frac{\partial E}{\partial a_k}, \tag{1.16}$$

sin embargo, la salida a_k del nodo de salida también depende de la entrada al nodo, z_k. Por ello, la ecuación anterior se puede definir finalmente como:

$$\delta_k = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial z_k} = \frac{\partial E}{\partial a_k} g'_k(z_k) \tag{1.17}$$

Teniendo esto, ya podemos obtener el error en cualquier nodo de salida, ya que la derivada parcial de la *loss function* respecto la salida del nodo es sencillo de computar:

$$\frac{\partial E}{\partial a_k} = (t_k - a_k) \tag{1.18}$$

Y la derivada de la función de activación también suele ser bastante fácil de calcular, por ejemplo, para las neuronas tipo ReLU, la derivada será nula cuando z_k sea menor o igual que cero (en el caso de *Leaky ReLUs*, tendrá el valor de m de la ecuación 1.6), y tendrá valor unitario cuando z_k sea mayor que cero.

Si pasamos estas ecuaciones a una notación matricial, llegamos a la ecuación final:

$$\delta^{out} = \bigtriangledown_{a^{out}} E \odot g'(z^{out}), \tag{1.19}$$

donde $\bigtriangledown_{a^{out}} E$ se refiere al gradiente de la *cost function* respecto el valor de cada uno de los nodos de salida, y $\odot$ es el llamado producto matricial de Hadamard. Sabiendo la derivada de la *cost function* respecto el valor de los nodos de salida, la ecuación anterior se reduce a:

$$\delta^{out} = (a^{out} - t) \odot g'(z^{out}), \tag{1.20}$$

A partir de aquí, empieza lo que se denomina *backpropagation*, debido a que tenemos el error en la salida de la red neuronal, pero no sabemos el error de cada una de las capas de la red. El proceso que se realiza ahora es el mismo que el que se utilizó para hallar las predicciones de la red neuronal en el apartado 1.4.1, sólo que esta vez vamos de la última capa (la capa de salida) a la primera (capa de entrada). Esto se debe, a que el error también va ponderado por los parámetros *weights*, por tanto es sencillo deducir que el error en cualquier capa es:

$$\delta^{l} = (\delta^{l+1} (w^{l+1})^{T}) \odot g'(z^{l}), \tag{1.21}$$

donde claramente se utiliza la traspuesta de la matriz de *weights* de la capa siguiente multiplicado por el error de dicha capa. El uso de la traspuesta en la matriz w^{l+1} es necesario y se puede intuir que eso es lo que permite transmitir el error hacia capas anteriores, es decir, hacer el proceso de *backpropagation*. Luego, al igual que en el resto de derivadas, hay que hallar el producto de Hadamard respecto la derivada de la función de activación. Con esto ya podremos aplicar el algoritmo *Backpropagation* para hallar el error introducido por cualquier capa que forma la red neuronal.

Utilizando el algoritmo de *Backpropagation*, ya se puede calcular los gradientes de la *cost function* respecto los parámetros de entrenamiento de la red: *weights* y *biases*.

En primer lugar, se puede calcular la derivada parcial de la *cost function* respecto cualquier *weight* de la red tal y como se representa en la siguiente ecuación:

$$\frac{\partial E}{\partial w_{jk}} = a_j \delta_k \tag{1.22}$$

Si esta ecuación se pasa a notación matricial, podremos encontrar el gradiente de la *loss function* respecto todos los *weights* de una capa:

$$\bigtriangledown_{w^l} E = (a^{l-1})^T \delta^l \tag{1.23}$$

Como vemos, esta ecuación permite calcular el gradiente de la *loss function* respecto los *weights* de la capa l. Para ello se utiliza el error de la capa calculado mediante el algoritmo de *Backpropagation* junto con las salidas de la capa anterior, lo cual parece lógico, ya que si las salidas de la capa anterior son cercanas a 0, los pesos que ponderan dichos valores no intervendrán mucho en la salida, y por tanto, apenas influirán en el error.

También es interesante observar que, cuando z^l es menor que cero, la derivada de la función de activación en el caso de usar neuronas de tipo ReLU, será nula. Esto implica, que el error introducido por dichas neuronas también lo es y por tanto la derivada parcial de la *cost function* respecto los *weights* que intervienen en dicha neurona será 0. Cuando sucede esto, dichos parámetros no cambiarán de valor y por tanto dejarán de "aprender"; esto es algo que hay que evitar para impedir un bloqueo en el proceso de aprendizaje de la red neuronal, y para ello es común el uso de *Leaky ReLUs* tal y como se mencionó en apartados anteriores. De igual forma sucede en otro tipo de neuronas como las sigmoidales o tanh, ya que en ciertos valores la pendiente es nula, resultando en la saturación de dichas neuronas.

Por último, se va a presentar la ecuación que rige el cambio de la *cost function* respecto cualquier *bias* de la red neuronal:

$$\frac{\partial E}{\partial b_j} = \delta_j \tag{1.24}$$

Exactamente, el error que aporta la neurona es equivalente a dicha derivada parcial. Pasado a notación matricial:

$$\bigtriangledown_{b^l} E = \delta^l \tag{1.25}$$

De esta forma, ya tenemos todas las ecuaciones necesarias para hacer nuestro algoritmo de *Gradient Descent* de una forma sencilla y rápida computacionalmente. Por ello, se puede proceder a realizar un código en Python que permita entrenar una red neuronal dados los ejemplos de entrenamiento, es decir, dados una entrada y una salida deseada. El código puede tener la siguiente forma:

```
#Derivada de la funcion de activacion ReLU
def relu_slope(input):

    n_elements = np.size(input)

    output = np.empty([1 , n_elements])
    i=0
    for _ in range(n_elements):
        if input[0,i] > 0:
            output[0 , i] = 1
        else:
            #Pendiente Leaky ReLUs
            output[0 , i] = 0.01
        i+=1

    return output

#Calculo del gradiente de una hidden layer
def gradient_hidden_layer(next_layer , actual_layer , previous_layer):
    '''
    El formato de gradiente debe tener el mismo formato en el
    que se definen los weights de cada layer: las columnas
    corresponderan con cada uno de los nodos que forman el
    hidden layer del cual queremos cambiar los weights que lo
    alimentan, mientras que las filas corresponderan al numero
    de nodos de la layer anterior, ya que cada uno de estos
    nodos conecta con un weight a cada uno de los nodos de la
    output layer
    '''

    #Si actual hidden layer tiene funcion de activacion (ReLU):
    g_dot = relu_slope(actual_layer['z'])

    actual_layer['error'] = np.multiply(np.dot(next_layer['error'] , (next_
        layer['weights']).T) , g_dot)

    actual_layer['weights_gradient'] = (np.dot((previous_layer['a']).T , actual
        _layer['error']))
    actual_layer['bias_gradient'] = actual_layer['error']

#Calculo del gradiente de la capa de salida
def gradient_output_layer(final_layer , previous_layer , output_targets):
```

```
    #Array con los errores de todos los nodos de output layer
    errors = final_layer['a'] - output_targets

    #Si output layer tiene funcion de activacion (ReLU):
    g_dot = relu_slope(final_layer['z'])
    final_layer['error'] = np.multiply(errors , g_dot)

    final_layer['weights_gradient'] = (np.dot((previous_layer['a']).T , final_
        layer['error']))
    final_layer['bias_gradient'] = final_layer['error']

#Algoritmo de Backpropagation para calcular el error de todas las capas
def backpropagation(targets):

    gradient_output_layer(out_layer,layer2,targets)

    gradient_hidden_layer(out_layer,layer2,layer1)

    gradient_hidden_layer(layer2,layer1,input_layer)
```

Del código hay que destacar el uso de diccionarios de Python, que permiten agrupar variables de distinto tipo en un mismo conjunto. En este caso se han supuesto definidas fuera de las funciones los diccionarios de: `out_layer`, `input_layer`, `layer1` y `layer2`, que contienen distintas variables como los *weights*, *bias*, sus respectivos gradientes, los valores de sus salidas a^l y de sus entradas z^l, variables que han tenido que ser inicializadas o calculadas como se hizo en el apartado 1.4.1. En los casos en los que existe una gran cantidad de variables y parámetros, el uso de diccionarios ayuda a comprender mejor el código y a tenerlo más organizado.

Demostración de las ecuaciones del algoritmo Backpropagation

En el apartado anterior, se explicó de forma rápida las ecuaciones que intervienen en este eficiente método de entrenamiento, pero en este apartado, analizaremos los conceptos y se verá claramente cómo aparecen dichas ecuaciones y cómo deben interpretarse.

En este momento dejaremos de usar la notación matricial que se utilizó en gran parte del apartado anterior y nos centraremos en la notación por elementos que permita derivar y hallar las ecuaciones de una forma más sencilla e intuitiva. De este modo, tomaremos de nuevo como referencia el diagrama de la red neuronal que se ve en la Figura 1.13.

En primer lugar, se va a calcular el gradiente para los *weights* de la capa de salida. Para ello, realizaremos la derivada parcial de la *cost function* respecto cualquier *weight* que conecta la capa de salida con la capa anterior:

$$\begin{aligned}\frac{\partial E}{\partial w_{jk}} &= \frac{\partial}{\partial w_{jk}}\Big(\frac{1}{2}\sum_{k\in K}(a_k-t_k)^2\Big)\\ &= (a_k-t_k)\frac{\partial}{\partial w_{jk}}(a_k-t_k) = (a_k-t_k)\frac{\partial}{\partial w_{jk}}a_k\end{aligned} \tag{1.26}$$

El sumatorio se elimina porque w_{jk} sólo afecta al nodo k y por tanto a la salida a_k, aunque en este caso al haber sólo un nodo de salida es indiferente. Hasta ahora ha sido sencillo, ya que simplemente se ha aplicado la regla de la cadena y se ha derivado como usualmente se suele hacer. Ahora sabemos que:

$$a_k = g_k(z_k) = g_k\Big(b_k + \sum_{j\in J}(a_j w_{jk})\Big) \tag{1.27}$$

Si lo añadimos a la ecuación 1.26 y se deriva a_k, obtenemos lo siguiente:

$$\frac{\partial E}{\partial w_{jk}} = (a_k - t_k) g_k'(z_k) a_j \quad \text{, donde} \tag{1.28}$$

$$\delta_k = (a_k - t_k) g_k'(z_k) \tag{1.29}$$

Al igual que antes, el sumatorio se elimina porque w_{jk} sólo afecta al nodo j, ya que, como ya se ha mencionado, w_{jk} es la conexión del nodo j con el nodo k.

Como podemos ver, el resultado de las ecuaciones 1.28 y 1.29 son equivalentes a las ecuaciones 1.17 y 1.22 deducidas en el apartado anterior. Este resultado ha sido bastante intuitivo y fácil de llegar debido principalmente a que se está utilizando la capa de salida, cuyas derivadas son mucho más simples y directas.

Sabiendo todo esto, se puede proceder a calcular la derivada parcial de la *cost function* respecto un *weight* de la *hidden layer*. El proceso en un primer instante es el mismo que con la capa de salida:

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \Big(\frac{1}{2} \sum_{k \in K} (a_k - t_k)^2 \Big) \\ &= \sum_{k \in K} \Big((a_k - t_k) \frac{\partial}{\partial w_{ij}} a_k \Big) \\ &= \sum_{k \in K} \Big((a_k - t_k) g_k'(z_k) \frac{\partial z_k}{\partial w_{ij}} \Big) \end{aligned} \tag{1.30}$$

Hay que destacar que en este caso el sumatorio no es eliminado porque w_{ij} interviene en el valor de todos los nodos de salida de la red neuronal como vemos en la ecuación 1.31. En este punto nos encontramos quizás en la parte más complicada de la demostración debido a la gran cantidad de términos que hay que derivar. Para continuar, debemos expandir la ecuación 1.27 para que aparezcan los términos w_{ij}:

$$z_k = b_k + \sum_{j \in J} \Big(g_j \big(b_j + \sum_{i \in I} (a_i w_{ij}) \big) w_{jk} \Big) \tag{1.31}$$

Si derivamos el término z_k respecto w_{ij}, sabemos que ambos sumatorios pueden eliminarse ya que el parámetro w_{ij} sólo afecta a los nodos i y j. Sabiendo esto, se puede proceder a realizar la derivada de z_k respecto w_{ij} aplicando de nuevo la regla de la cadena para poder hacer dicha derivada de una forma mucho más sencilla:

$$\begin{aligned} \frac{\partial z_k}{\partial w_{ij}} &= \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} \\ &= \frac{\partial}{\partial a_j} a_j w_{jk} \frac{\partial a_j}{\partial w_{ij}} \\ &= w_{jk} g_j'(z_j) \frac{\partial z_j}{\partial w_{ij}} \\ &= w_{jk} g_j'(z_j) a_i \end{aligned} \tag{1.32}$$

Si la ecuación anterior la añadimos a la ecuación 1.30, tendremos como resultado lo siguiente:

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \sum_{k \in K} \Big((a_k - t_k) g_k'(z_k) w_{jk} \Big) g_j'(z_j) a_i \\ &= \sum_{k \in K} \Big(\delta_k w_{jk} \Big) g_j'(z_j) a_i \end{aligned} \tag{1.33}$$

Si nos fijamos atentamente en la ecuación 1.33, lo que estamos haciendo es hacer el proceso de *Backpropagation*, ya que cada uno de los errores δ_k de los nodos de salida son ponderados por los parámetros

weights que conectan los nodos de salida con el nodo j que se está evaluando. Por tanto, dicha ecuación se puede reducir a lo siguiente:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j a_i \quad \text{, donde} \tag{1.34}$$

$$\delta_j = \sum_{k \in K} \left(\delta_k w_{jk} \right) g'_j(z_j) \tag{1.35}$$

Con esto ya tenemos demostrado cómo aparecen las ecuaciones de *Backpropagation* para los diferentes *weights* que intervienen en la red neuronal, que como vemos, son equivalentes a las ecuaciones presentadas en el apartado anterior.

Sabiendo esto, es fácil demostrar tanto para *hidden layers* como para la capa de salida, cómo cambia la *cost function* respecto los *bias*. Para los *bias* de la capa de salida obtenemos lo siguiente:

$$\begin{aligned}\frac{\partial E}{\partial b_k} &= \frac{\partial}{\partial b_k} \left(\frac{1}{2} \sum_{k \in K} \left(a_k - t_k \right)^2 \right) \\ &= (a_k - t_k) \frac{\partial}{\partial b_k} a_k \\ &= (a_k - t_k) g'_k(z_k) = \delta_k \end{aligned} \tag{1.36}$$

Como vemos, la demostración de los *bias* se hace mucho más sencilla ya que se trata de términos que van sumados, y por tanto, son independientes en las derivadas.

Por último, hallaremos lo mismo pero para la *hidden layer*:

$$\begin{aligned}\frac{\partial E}{\partial b_j} &= \frac{\partial}{\partial b_j} \left(\frac{1}{2} \sum_{k \in K} \left(a_k - t_k \right)^2 \right) \\ &= \sum_{k \in K} \left(a_k - t_k \right) \frac{\partial}{\partial b_j} a_k \\ &= \sum_{k \in K} \left(\delta_k w_{jk} \right) g'_j(z_j) = \delta_j \end{aligned} \tag{1.37}$$

Ahora, hemos obtenido de nuevo exactamente los mismos resultados que en el apartado anterior, quedando demostradas todas las ecuaciones que participan en el algoritmo de *Backpropagation*.

Delta Rule

Una vez conocemos el gradiente del error cuadrático respecto el vector de parámetros, se puede proceder a modificar en dichos parámetros una cantidad equivalente al valor del gradiente para así desplazarnos a lo largo de la curva del error cuadrático y disminuir el coste de la red neuronal, y de esta forma comenzar con el proceso de entrenamiento de la red.

Por tanto, en práctica, en cada iteración en la que entrenamos la red neuronal, nos moveremos en una dirección perpendicular al contorno de la *loss function*, pero necesitamos calcular cuánto debemos movernos antes de calcular la nueva dirección. En un principio, sabemos que la distancia que nos desplazaremos debe depender de la pendiente del error en el punto actual, y por tanto de los gradientes calculados, ya que si nos encontramos en un punto donde la pendiente es elevada, tendremos que modificar más nuestros parámetros porque nos encontramos lejos de un punto donde la pendiente sea nula y por tanto el error sea mínimo; y ocurrirá lo contrario cuando nos encontremos cerca del error mínimo. Pero en tal caso, la distancia que nos movemos a lo largo de la curva del error, no será controlable, ya que dependerá sólo de los gradientes. Por ello usaremos lo que se denomina *Delta Rule*, que no es más que una regla de entrenamiento para el algoritmo de *Gradient Descent* que permite actualizar los valores del vector de parámetros θ en una capa de la red neuronal. Esta regla lo que hará, será añadir un parámetro más para modificar los valores del vector θ,

y se trata del *learning rate*. Este parámetro ponderará el gradiente de la *loss function* respecto el vector de parámetros θ calculado por un valor determinado para modificar dichos parámetros una cantidad deseada.

Con *Delta Rule* tendremos la posibilidad de controlar cuánto nos desplazamos por la *loss function*. De esta forma, si se requiere un entrenamiento más rápido, se elevará el valor del *learning rate*; sin embargo, esto puede llevar a errores, ya que si se elige un valor muy alto, existirá el peligro de que continuamente alternemos entre valores extremos de los parámetros que impedirán la convergencia en un punto de error mínimo. Este suceso es lo que se donmina *overshooting*. Sin embargo, si se elige un valor demasiado pequeño, el aprendizaje de la red neruonal será muy lento, aunque finalmente convergerá en un punto de error mínimo.

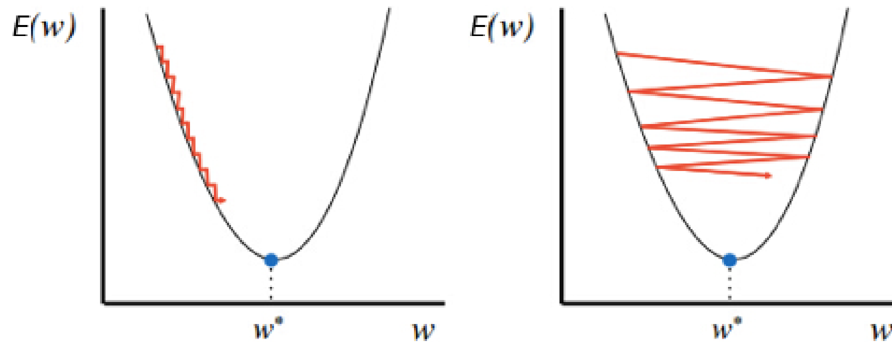


Figura 1.14 Representación gráfica de elección de un *learning rate* bajo en el que converge lentamente a la izquierda; y a la derecha, un *learning rate* alto que llega incluso a divergir.

No obstante, la elección de un valor óptimo del *learning rate* es muy complicado, aunque existen métodos en el que ésta variable pasa a ser un parámetro entrenable cuyo valor depende de la situación de apredizaje de la red neuronal. Estos métodos se denominan *adaptative learning rate methods*, que prácticamente nos permiten automatizar el proceso de elección de dicho valor. Hay que destacar que aunque existan métodos para calcular el valor del *learning rate*, es común fijarlo a un único valor constante, que suele ser de 0.001, aunque siempre modificable dependiendo de la situación en la que nos encontremos.

Teniendo en cuenta los códigos anteriores expuestos y explicados, una forma simple de modificar los parámetros de cada capa que forma la red neuronal, sería la siguiente:

```
def update_variables(learning_rate):

    out_layer['weights'] = out_layer['weights'] - out_layer['weights_gradient']
         * learning_rate
    layer2['weights'] = layer2['weights'] - layer2['weights_gradient'] *
        learning_rate
    layer1['weights'] = layer1['weights'] - layer1['weights_gradient'] *
        learning_rate

    out_layer['biases'] = out_layer['biases'] - out_layer['bias_gradient'] *
        learning_rate
    layer2['biases'] = layer2['biases'] - layer2['bias_gradient'] * learning_
        rate
    layer1['biases'] = layer1['biases'] - layer1['bias_gradient'] * learning_
        rate
```

Como vemos, es sencillo actualizar los valores, ya que simplemente hay que restar a los parámetros sus respectivos gradientes ponderados por el *learning rate*. De esta forma, se comenzará a actualizar los valores y por tanto a entrenar la red neuronal.

Entrenamiento Eficiente: Batches

El entrenamiento de una red neuronal es un proceso que consume gran cantidad de recursos de un ordenador. De hecho, gracias a las mejoras tecnológicas de los últimos años, el campo de *Deep Learning* comenzó a desarrollarse a pasos agigantados.

Esto se debe, principalmente al uso de la GPU del ordenador, la cual permite que los procesos de aprendizaje se puedan realizar de forma paralela y mucho más eficiente que usando los procesadores del ordenador. Las GPUs se pueden definir como procesadores especializados que fueron desarrollados inicialmente para llevar a cabo procesamiento de imágenes complejo, como procesamiento 3D. Sin embargo, el nuevo uso de las GPUs es para el entrenamiento de *Deep Neural Networks* debido a que éstas se encuentran especialmente diseñadas para trabajar de forma eficiente con matrices. Además, debido a que están formadas por cientos o incluso miles de núcleos, son capaces de procesar grandes cantidades de información de forma paralela y simultánea, y por ello, se ajustan a la perfección a la capacidad de computación que se puede llegar a necesitar en el entrenamiento de redes neuronales. Por ejemplo, la librería *TensorFlow* permite una instalación que posibilita el uso de la GPU como herramienta para realizar todas las operaciones sobre los tensores, incluído el entrenamiento.

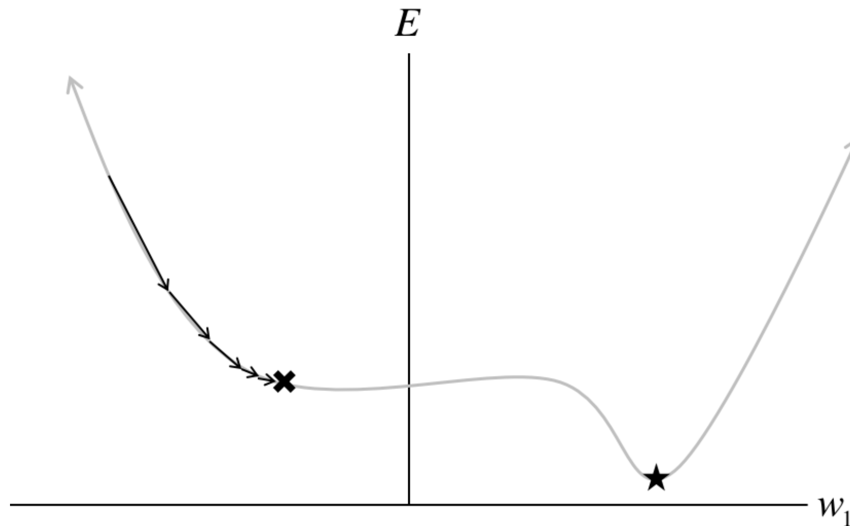
A pesar de ello, en bastantes ocasiones es necesario procesar mucha información para entrenar la red neuronal. Esto se debe principalmente a *Big Data*, *training datasets* tan grandes y complejos que se necesitaría mucho tiempo para entrenar una red neuronal a la perfección. Por ello, existen varios métodos de *Gradient Descent* que permiten entrenar la red neuronal de una forma mucho más eficiente aunque también puede implicar que el entrenamiento no sea óptimo, consiguiendo unos resultados deficientes.

En los apartados anteriores, cuando se explicó el proceso de *Gradient Descent* y *Backpropagation*, en realidad se utilizó de uno de los métodos de *Gradient Descent* existentes, el denominado *Stochastic Gradient Descent*. Este método consiste en que si tenemos un *dataset* con una gran cantidad de ejemplos, procesaríamos y entrenaríamos la red neuronal ejemplo a ejemplo. Una de las ventajas importantes a la hora de usar este método de entrenamiento es, que al procesar en cada ejemplo el error cuadrático, se puede evitar los llamados puntos de silla, ya que estos puntos nos impedirán encontrar el mínimo absoluto del error cuadrático, quedándonos estancados en un punto con pendiente nula, pero no mínimo.

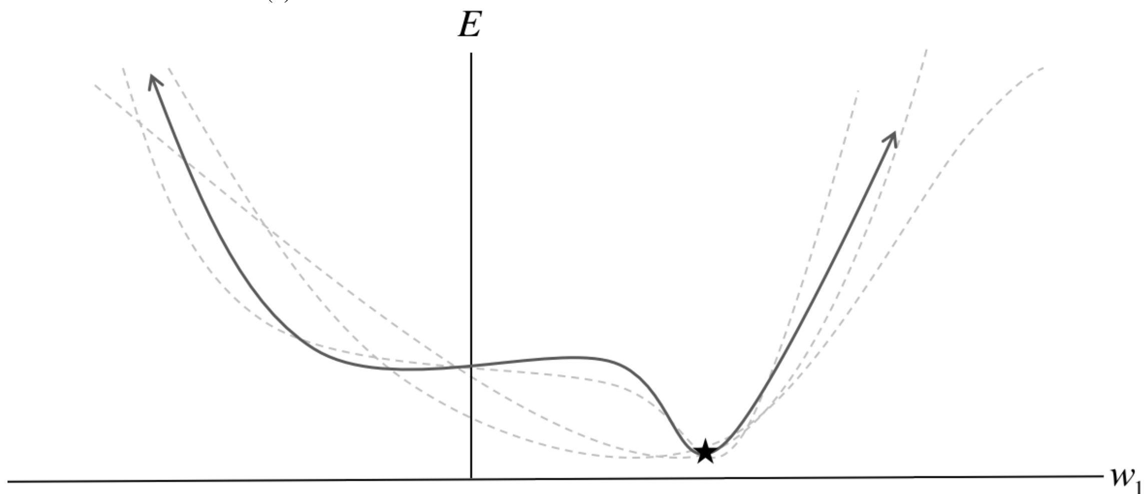
Sin embargo, también existe el método opuesto, el llamado *Batch Gradient Descent*. En este caso la aparición de puntos de silla se hace notable, ya que dicho método consiste en calcular el error cuadrático para el *dataset* completo, que funciona muy bien cuando se trata de errores sencillos, pero en cuanto se complica algo, la aparición de estos tipos de puntos se hace frecuente, impidiendo el correcto entrenamiento de la red neuronal.

Sin embargo, ninguno de los dos métodos es infalible. En primer lugar, con *Batch Gradient Descent* en un entorno multidimensional, es muy difícil no estancarse en un punto de silla, con lo que nunca los entrenamientos serán óptimos. En segundo lugar, con *Stochastic Gradient Descent* evitaríamos dicho problema, pero nunca es conveniente calcular en cada ejemplo el error ya que no es una buena aproximación del error al completo que tendremos en el conjunto del *dataset*, lo que ocasionaría una estimación poco precisa de los gradientes de los parámetros, resultando en un entrenamiento bastante peor por fluctuar los gradientes. Además, este tipo de entrenamiento tiene otro problema ya mencionado anteriormente, la velocidad de procesamiento, que será mucho mayor ya que en cada ejemplo estaremos calculando el error y modificando los parámetros de la red neuronal.

Por ello, a la hora de entrenar redes neuronales, es común utilizar un método intermedio, *Mini-batch Gradient Descent*, que consiste en dividir dichos *datasets* en *minibatches*, es decir, se agrupan los ejemplos de entrenamiento en conjuntos más pequeños para procesar dichas agrupaciones por separado. Este método permite evitar en gran medida puntos de silla ya que no estamos utilizando un *dataset* al completo, y por tanto, el entrenamiento de la red neuronal resultará más satisfactorio. Además, el uso de *minibatches* implica una mejora en el tiempo de computación en el entrenamiento de redes neuronales, puesto que el procedimiento consiste en calcular el gradiente medio de la *loss function* de un único *minibatch*, y una vez lo tengamos, se actualizan los parámetros de la red neuronal según la media del gradiente obtenido, y se repite el procedimiento con el siguiente *minibatch*. Es decir, el método de entrenamiento sería el siguiente:



(a) Punto de silla con un entrenamiento *Batch Gradient Descent*.



(b) Entrenamiento de tipo *Stochastic Gradient Descent*, en el que cada curva punteada corresponde con el error calculado en cada ejemplo.

Figura 1.15 Mientras que en el caso a) no se puede encontrar el mínimo absoluto, en el caso b) sí, ya que cada ejemplo de entrenamiento tiene su propio error, que en conjunto forma un punto de silla pero no por separado, teniendo en este caso un entrenamiento eficiente.

1. Se divide el *dataset* de entrenamiento en *minibatches*, preferiblemente todos los *minibatches* deberían tener el mismo número de ejemplos de entrenamiento.
2. Se evalúa la salida para cada ejemplo de entrenamiento de un *minibatch* y con ello el error cometido por la red neuronal y los gradientes de la *loss function*.
3. Los gradientes obtenidos de cada capa en cada ejemplo de entrenamiento del *minibatch* se suman para obtener, una vez procesado todo el *minibatch*, la media de los gradientes obtenidos.
4. Mediante la media obtenida de cada gradiente, se actualizan los parámetros de la red neuronal.
5. Se coge el siguiente *minibatch* y se repite el proceso desde punto 2.

Una vez se hayan procesado todos los *minibatches* que forman el *dataset* al completo, se dice que se ha realizado un *epoch*. Es usual hacer varios *epochs* para repetir el mismo *dataset* varias veces y así perfeccionar el entrenamiento de la red neuronal.

Con esto, podremos entrenar cualquier red neuronal de una forma más o menos eficiente, dependiendo de los valores que cojamos para el tamaño del *minibatch*, tamaño en el que se debe encontrar el equilibrio entre

la eficiencia de *Batch Gradient Descent* y la evitación de los puntos de silla de *Stochastic Gradient Descent*, y para el número de *epochs* a realizar, que lo veremos en el próximo apartado.

A pesar de todo esto, hay que destacar que existen muchos algoritmos de entrenamientos basados en *Gradient Descent* que impiden problemas como los puntos de silla e incluso son capaces de encontrar un mínimo absoluto muy rápidamente (algunos de estos algoritmos se mencionaron en el apartado 1.5).

Overfitting

La construcción y diseño de modelos de *Deep Neural Networks* es uno de los mayores retos a los que nos debemos enfrentar si queremos conseguir resultados óptimos.

Es común contruir modelos muy complejos que permiten en el entrenamiento de las redes neuronales que se ajusten a la perfección a todos los ejemplos dados porque le damos a nuestro modelo suficientes grados de libertad para que pueda ajustarse a todas las observaciones realizadas.

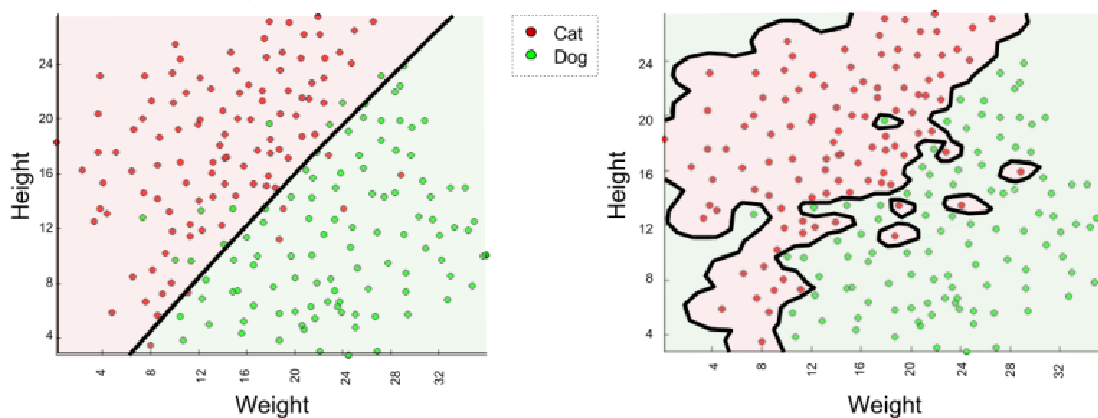


Figura 1.16 A la izquierda, clasificación de una red neuronal muy sencilla; a la derecha, clasificación de una red neuronal muy compleja.

Sin embargo, cuando se evalúa estos modelos con nuevos ejemplos, la red neuronal no es capaz de obtener buenos resultados, es decir, no es capaz de generalizar los datos correctamente. Este fenómeno se denomina *overfitting* y es uno de los mayores retos a la hora de diseñar y entrenar redes neuronales. Este problema es especialmente difícil en el caso de *Deep Learning*, debido a que tenemos múltiples capas con una gran cantidad de neuronas. Por supuesto, la complejidad de la red neuronal será mayor cuantas más capas y neuronas posea, ya que cuantas más conexiones tenga, más parámetros pueden ser entrenados, y con ello, se pueden desarrollar modelos mucho más complejos.

Por tanto, si nuestro modelo no es suficientemente complejo, no se llegará a capturar toda la información importante de los datos para resolver el problema. Sin embargo, si el modelo es muy complejo, especialmente cuando tenemos un limitado número de ejemplos de entrenamiento, nos arriesgamos a que la red neuronal no pueda generalizar bien.

Para evitar que suceda todo esto, nos vamos a basar en unas soluciones simples que nos permiten evaluar nuestra red neuronal en el proceso de entrenamiento, y de esta forma, podremos evitar el fenómeno de *overfitting*. Para ello, además del *dataset* de entrenamiento, es usual tener otro *dataset* (normalmente con menos ejemplos) para poder evaluar nuestra red neuronal. Dicho *dataset* se denomina *test data*, que normalmente, por simplificar, lo que se hace es dividir el propio *dataset* que ya tenemos en dos conjuntos: *trainig data* y *test data*. Este conjunto de ejemplos, nos permitirá evaluar el modelo midiendo directamente el porcentaje de aciertos (precisión) que ha obtenido ante un conjunto de ejemplos nuevos para la red neuronal, es decir, que no ha sido entrenada con dichos ejemplos. Debido a que conseguir *dataset* es una de las tareas más difíciles en el entrenamiento de redes neuronales, es un fallo común usar los mismos ejemplos tanto para entrenarla como para validarla, es algo que hay que evitar ya que es esencial la validación de nuestra red neuronal y

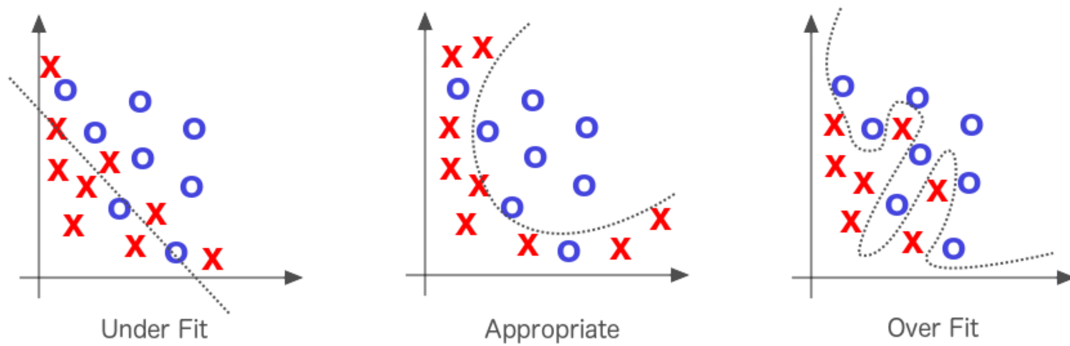


Figura 1.17 Para que la red neuronal pueda generalizar, hay que encontrar un punto medio entre redes muy simples y otras muy complejas.

llegar a conclusiones que no estén falseadas.

Durante el entrenamiento de la red neuronal, se llega a un punto donde, en vez de seguir aprendiendo la información importante y útil que caracterizan los datos de entrenamiento, aparece el fenómeno de *overfitting*. Para evitarlo, debemos ser capaces de parar el proceso de entrenamiento tan pronto como empiece dicho fenómeno. Para hacerlo, se divide el proceso de entrenamiento en *epochs*, que como ya mencionamos, un *epoch* consiste en haber completado totalmente el entrenamiento sobre todos los ejemplos que disponemos de nuestro *trainig dataset*. Esto implica, que si disponemos un *training dataset* de d ejemplos de entrenamiento, y utilizamos el método de *Mini-batch Gradient Descent* con un tamaño de *minibatch* de b, en un *epoch* completaremos $\frac{d}{b}$ actualizaciones de los parámetros de la red neuronal. Al final de cada *epoch*, debemos evaluar cuán bien entrenada está la red neuronal y si es capaz de generalizar lo suficiente con nuevos datos. Por ello, se suele dividir nuestro *datatset* en otro conjunto de ejemplos, llamado *validation dataset*, que debe estar compuesto por ejemplos que nunca se le ha presentado a la red. Por ello, si la precisión tras el entrenamiento de un *epoch* con los datos del *validation dataset* aumenta, se podrá seguir entrenando la red neuronal; pero si la precisión se mantiene o incluso disminuye, es un buen signo para dejar de entrenarla, ya que implica que está apareciendo el fenómeno de *overfitting*.

También, en el caso de que tengamos algún método para cambiar los valores de otros parámetros de entremaiento, como *learning rate* o el tamaño del *minibatch*, el *validation dataset* es un buen indicativo para cambiar el valor de dichos parámetros. Por ejemplo, un buen método, sería teniendo un conjunto de valores predefinido para los parámetros, entrenar la red neuronal con todos los posibles valores de los parámetros, y dependiendo de la precisión que se obtenga en el *validation dataset*, se escogen los valores que obtengan los mejores resultados.

Sabiendo todo esto, volvamos al problema de evitar el fenómeno de *overfitting*. En este caso se va a proponer una solución que consiste en un proceso de entrenamiento de red neuronal que nos permite finalmente obtener buenos resultados. Este procedimiento se encuentra descrito en la Figura 1.18.

El procedimiento consiste en, una vez que tengamos la red neuronal diseñada, debemos recolectar la mayor cantidad de datos para crear nuestros *datasets*. Es frecuente e importante mezclar los ejemplos de entrenamiento para evitar cualquier relación lineal que haya entre ellos, y una vez hecho, creamos nuestro *training*, *test* y *validation datasets*. Con esto, estamos listos para entrenar la red neuronal mediante *Gradient Descent*. Al final de cada *epoch*, nos aseguraremos que tanto en el *training dataset* como en el *validation dataset* el error está disminuyendo, y una vez que alguno de los errores deje de disminuir, paramos el proceso de entrenamiento y nos aseguramos que se obtienen buenos resultados en el *test dataset*. Si no se obtienen los resultados deseados, se debe volver a pensar la arquitectura del modelo de red neuronal o reconsiderar si el *dataset* obtenido es el óptimo para que la red pueda captar toda la información importante, o incluso que el *dataset* no contenga los suficientes ejemplos y por ello la red neuronal falla a la hora de procesar ejemplos nunca vistos.

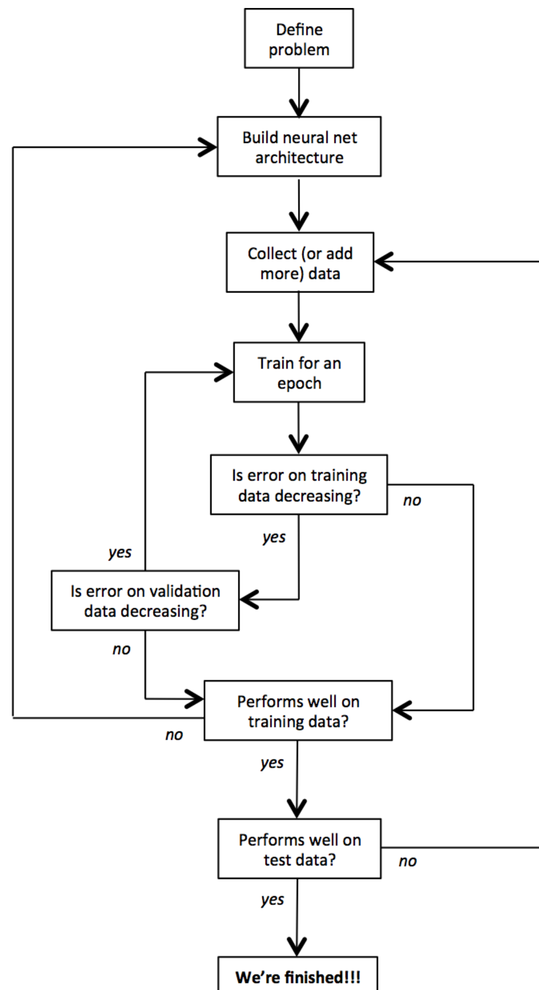


Figura 1.18 Procedimiento idóneo para conseguir una red neuronal eficiente y bien entrenada.

Con ello, tenemos un procedimiento bastante estricto que nos permitirá crear y entrenar una red neuronal que se ajuste a la perfección a los objetivos que queremos.

Igualmente, hay que destacar que existen muchos métodos que permiten un buen entrenamiento de la red neuronal y sobre todo para combatir *overfitting*. Algunos métodos existentes son *Regularization* y *Max norm constraints*, los cuales tienen como objetivo evitar que el vector de parámetros θ tenga algunos valores muy elevados. Por último nos encontramos *Dropout*, uno de los métodos más utilizados para prevenir *overfitting* en *Deep Neural Networks*. Este método obliga a la red neuronal a usar todas sus neuronas por igual a partir de la activación y desactivación de algunas de éstas durante el entrenamiento. De esta manera, fuerza a la red a que sea precisa incluso con la ausencia de alguna de sus neuronas, y por tanto, ante la ausencia de cierta información. Así, evita que la red neuronal se vuelva muy dependiente de ciertas neuronas y hace que aprenda a usar todas las que posee.

Por último, hay que destacar que puede surgir *overfitting* de otras formas. Por ejemplo, es común que en entrenamientos de clasificación sobre clases mutuamente excluyentes, se alimente a la red neuronal para entrenarla con más ejemplos de una clase que de otra, provocando en la red neuronal un *overfitting* sobre dicha clase. Esto provoca que la red neuronal no esté bien entrenada y equilibrada ya que se ha entrenado más para clasificar una clase que el resto, por lo que es recomendable conseguir en estos casos el mismo número de ejemplos para todas las clases a clasificar.

Con esto, ya tendríamos gran parte de los conceptos sobre redes neuronales aprendidas y asimiladas. A pesar de que existe muchos más conceptos importantes e interesantes por mostrar, con esto tendremos suficientes conocimientos para hacer una *Deep Neural Network* funcional y eficiente.

Conclusión

En el Anexo 1, se encuentra adjunto un código completo con una red neuronal *FeedForward* de tres *hidden layers* junto con el código necesario de *Gradient Descent* y *Backpropagation* para un correcto entrenamiento de la red. Expresamente, en dicho código, la red neuronal será entrenada para que sea capaz de sumar dos términos, de forma, que mediante sumas aleatorias con cifras del 0 al 9 aprende las relaciones matemáticas que existen para poder sumar dos términos, y así, finalmente, podrá sumar con cualquier cifra, no sólo en ese rango de 10 números. De esta forma, nos damos cuenta del potencial de las redes neuronales, y que éstas son capaces de encontrar relaciones matemáticas entre unas entradas determinadas y las salidas correspondientes, pudiendo modelar sistemas matemáticos extremadamente complejos y no lineales.

Dicho código está preparado para que se puedan añadir y eliminar capas con relativa facilidad, y cambiar con facilmente el modelo de red neuronal según la complejidad que deseemos. Además, en este código no se han añadido características muy complejas a la red neuronal, con lo que su entrenamiento y el procesamiento de los datos es bastante más ineficiente que con un código que se pueda hacer con librerías específicas, como Keras, Theano o TensorFlow. A pesar de ello, es una buena forma de comprender el principio básico de las redes neuronales, cómo funcionan y cómo son entrenadas, para así poder entender con absoluta claridad cómo se usan librerías de este estilo y hacer redes neuronales mucho más complejas e interesantes, como haremos en los siguientes capítulos.

Introducción a TensorFlow

¿Qué es TensorFlow?

Ahora que tenemos una buena base teórica y comprendemos el funcionamiento de las *Deep Neural Networks*, en los próximos apartados aprenderemos a utilizar una librería de alto nivel que nos permite diseñar redes neuronales con relativa facilidad.

La herramienta que se va a utilizar es TensorFlow, una librería de código abierto creada por Google en 2015, con el fin de ayudar a desarrolladores a diseñar, construir y entrenar modelos de redes neuronales fácilmente. Debido a que se trata de una librería de código abierto y relativamente moderna, se espera que en los próximos años haya mucha mejoría y cambien la funcionalidad del código, ya que, todavía, se encuentra en una etapa bastante temprana de su desarrollo. A pesar de ello, TensorFlow ya ha superado a otras librerías existentes como Keras o Theano que han sido sustituidas por la librería de Google, principalmente gracias a su buen diseño y la facilidad de uso. Además, TensorFlow propociona una interfaz limpia y sencilla, pudiendo expresar una gran cantidad de modelos de redes neuronales en unas pocas líneas de código, asimismo, posee muchas características que sólo han sido desarrolladas en esta librería y que permite el uso de redes neuronales en sistemas reales debido a su eficiencia y eficacia.

Esta librería se encuentra programada en Python, C++ e incluso Java, y permite a los usuarios programar grafos de flujo de datos (*dataflow graphs*), y por tanto, expresar cualquier red neuronal mediante un código simple. Estos grafos están formados por nodos (*nodes*), que representan las operaciones que se realizan sobre los tensores (unidades de cómputo), y por conexiones (*edges*), que representan los tensores que forman cada una de las operaciones y permiten la comunicación de unos nodos con otros (unidades de datos). Por tanto, TensorFlow trabaja con tensores, que no son más que arrays multidimensionales (los vectores son un tensor 1D, las matrices un tensor 2D...). Debido a que se trabaja con grafos cuando se programa con TensorFlow, lo primero que se hace es la creación y construcción del grafo, y luego, se procede a ejecutarlo para obtener valores numéricos en los tensores.

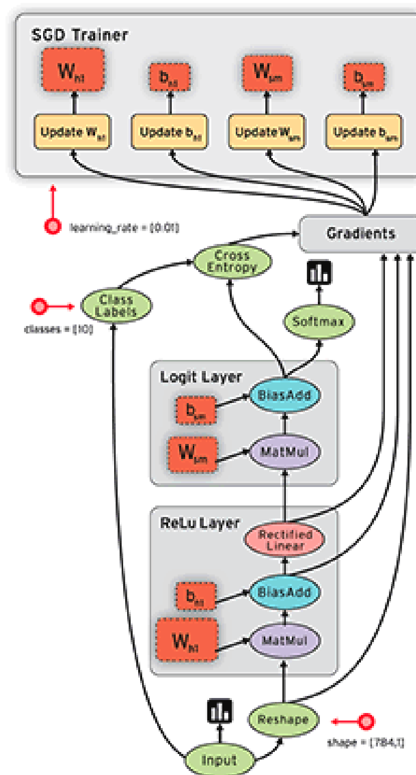


Figura 2.1 Ejemplo de TensorFlow *dataflow graph*.

Sin embargo, el razonamiento de redes neuronales como un conjunto de tensores no es sencillo, aunque en el primer capítulo de este proyecto ya se ayudó a entenderlas como tal. Como ya se comentó, representar las redes neuronales como un conjunto de tensores, ayuda a poder mejorar en gran medida las velocidades de procesamiento y entrenamiento de las redes neuronales, gracias principalmente a la nueva tecnología, como el uso de GPU y su procesamiento paralelo de operaciones con tensores. Para entender bien el uso de esta librería, se procederá a realizar una *FeedForward Network* tal y como se ha hecho en el primer capítulo usando Python sin ninguna librería, cuyo código se puede ver en el Anexo 1.

Python, TensorFlow y otros módulos

Como ya se ha mencionado en apartados anteriores, en este proyecto se va a hacer uso de Python para programar redes neuronales con TensorFlow. Debido a que se utilizará el sistema operativo Ubuntu 16.04, Python deberá estar instalado de forma predeterminada.

Para comprobar la versión instalada se puede introducir en el terminal el siguiente comando:

```
$ python -V
```

En el caso de no tener ninguna versión instalada, se puede hacer facilmente mediante el comando:

```
$ sudo apt-get install python
```

En este caso, la versión usada es Python 2.7.12, la más actualizada hasta la fecha, aunque también se podrá usar Python3 sin ningún inconveniente ya que TensorFlow soporta ambas versiones.

Ya con Python instalado, se puede proceder a la instalación de TensorFlow. Aunque hay varios métodos para poder hacerlo, el más sencillo y rápido es por medio del módulo Pip que se puede instalar a partir de un simple comando:

```
$ sudo apt-get install python-pip python-dev
```

Se recomienda, que si se tiene una GPU Nvidia, se instale la versión de TensorFlow con soporte de GPU, que aunque la instalación sea bastante más complicada, esto permitirá un gran rendimiento y eficiencia a la hora de ejecutar redes neuronales complejas, sobre todo durante su entrenamiento. A pesar de ello, si no puede ser posible, se podrá instalar la versión con sólo soporte de CPU de la siguiente forma:

```
$ pip install tensorflow
```

Una vez se haya terminado de ejecutar el comando sin errores, podremos validar la instalación ejecutando el entorno interactivo de Python en la propia línea de comandos escribiendo sólo

```
$ python
```

y ya simplemente importamos TensorFlow:

```
$ import tensorflow as tf
```

Si tras ejecutar dicho comando no hay error, quiere decir que se ha instalado todo correctamente. Probemos TensorFlow con un pequeño programa:

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

Simplemente ejecutamos dicho código con la línea de comandos y se observará como realmente se imprime por pantalla el mensaje ”Hello, TensorFlow!”.

Por último, se quiere destacar que se han hecho uso de otras muchas librerías y módulos de Python para realizar este proyecto, pero principalmente hay que destacar NumPy, de la cual ya hemos hablado en el capítulo anterior, y OpenCV, una librería muy útil y famosa en Python, que como veremos en los próximos capítulos, nos será necesaria para capturar imágenes y crear los *datasets* de entrenamiento y testeo.

Una vez instalados todos los módulos correctamente, se puede proceder a aprender el funcionamiento básico de TensorFlow para empezar a programar redes neuronales.

Conceptos básicos de TensorFlow

Para poder diseñar una red neuronal con TensorFlow debemos aprender cómo se usa esta librería y cómo funciona, ya que es muy compleja, poseyendo una gran cantidad de funciones que dependiendo nuestra finalidad hay que utilizar unas u otras.

Es muy importante entender el concepto de grafo de flujo de datos, ya que es la forma de trabajar de TensorFlow. Por tanto, los programas de TensorFlow comienzan siempre con la construcción de un grafo, fase en la que se utiliza la API de TensorFlow para llamar a funciones que generan operaciones (`tf.Operation`), y más en concreto, los nodos del grafo; cuando se ejecutan estas funciones, se crearán tensores (`tf.Tensors`), es decir, objetos que representan la unidad de dato que fluye entre operaciones (conexiones de los nodos), por lo que, mediante la concatenación de funciones se termina generando uno o varios ”subgrafos” (pequeños conjuntos de operaciones dependientes unas de otras) que, finalmente, se añadirán a un sólo grafo (`tf.Graph`). Normalmente, debido a que no se van a programar modelos muy complejos, se suele utilizar únicamente el grafo que por defecto usa la API de TensorFlow de forma automática. El conjunto de nodos y conexiones indicarán qué transformaciones (operaciones matemáticas o de otro tipo) se realizarán sobre cada uno de los tensores que participan en el grafo, pero no darán un resultado numérico, ya que para ello habrá que ejecutar el grafo (o subgrafo) mediante una sesión (`tf.Session`).

Esta forma de programar mediante grafos de flujo de datos es un modelo de programación muy común para computar en paralelo. Por tanto, este modelo de programación permite tener muchas ventajas: realización de cálculos de forma paralela (resulta más fácil de esta forma identificar las dependencias en el cálculo de una operación), ejecución distribuida (pudiendo usar al mismo tiempo CPU y GPU para el cómputo de operaciones), compilación eficiente (el compilador puede generar un código mucho más rápido) y portabilidad (te permite programar un modelo en Python pudiendo exportarlo luego a un programa escrito en C++, por ejemplo). Todas estas ventajas hacen de TensorFlow una gran librería que permitirá programar redes neuronales de una forma muy eficiente y sencilla.

Ahora que sabemos qué es un grafo, debemos aprender el concepto de operación, que como ya se ha mencionado, no es más que cualquier transformación que se le realiza a uno o varios tensores, que no serán ejecutadas hasta que no se realice una sesión. TensorFlow posee una grandísima cantidad de operaciones y de muchos tipos que se pueden realizar a los distintos tensores, algunas de ellas se pueden ver en la Tabla 2.1.

Tabla 2.1 En esta tabla están representadas algunas de las operaciones más usadas de TensorFlow. .

Tipo de Operación	Ejemplos
Operaciones por elementos	add, sub, mul, div, exp, greater, less, equal...
Operaciones en arrays	concat, slice, split, rank, shape, shuffle...
Operaciones en matrices	matmul, matrixinverse, matrixdeterminant...
Operaciones específicas de Redes Neuronales	softmax, relu, convolution2d, maxpool...

Cuando se llama a una función que genere una operación, ésta se añade automáticamente al grafo por defecto, con lo que, por ejemplo, si se llama a la función `c=tf.matmul(a,b)`, lo que sucederá es que se generará una operación de tipo ”MatMul” (que se añadirá al grafo por defecto) que coge los tensores `a` y

`b` como entrada, los multiplica, y genera el tensor `c`. Aunque como ya se mencionó, esta función no va a devolver ningún valor numérico ya que simplemente con ella se ha generado un nodo en el grafo, por lo que habrá que ejecutarla con una sesión si se quiere saber el resultado.

Sin embargo, para realizar dichas operaciones es necesario tener un tensor. Por sencillez, TensorFlow permite el uso de otro tipos de objetos, que no sean específicamente de tipo `tf.Tensor`, para poder usarlos como tal. Por ejemplo, se pueden usar arrays de NumPy, listas de Python, escalares de Python (int, bool, float, str...), aunque, además, se podrán usar funciones específicas de TensorFlow para generar tensores con características especiales que serán muy importantes en la creación de redes neuronales, como son las variables y los *placeholders*.

Es muy importante aprender el concepto de las variables de TensorFlow. Estas variables serán tensores que mantienen su estado en los buffers de memoria temporalmente, por lo que, a diferencia de los tensores normales que sólo son instanciados cuando se ejecuta un grafo y después de ello son limpiadas de memoria, las variables se mantienen a lo largo de las ejecuciones. Como cualquier otro objeto de tipo tensor, las variables se pueden usar como entradas a cualquier operación del grafo, de hecho, estas variables son las que se usan para definir los parámetros del modelo de la red neuronal, es decir, definiremos como variables nuestro vector de parámetros θ, como *weights* y *biases*. Esto se debe a que, gracias a que estos tensores se mantienen entre las distintas ejecuciones, se pueden usar los algoritmos de *Gradient Descent* para entrenar dichos tensores tras cada iteración de ejecución del grafo de forma eficiente, para encontrar de esta forma los valores óptimos de dichos parámetros para el modelo. Además, una de las mejores características de las variables es que una vez entrenada la red neuronal por medio de estas variables, se pueden guardar sus valores en un archivo para un posterior uso. Aunque sean objetos parecidos a los tensores, las variables tienen unas características especiales y es necesario tratarlas de forma algo distinta que al resto de tensores.

Las variables deben ser obligatoriamente inicializadas antes de la primera ejecución de un grafo o una operación para usar sus valores. Para inicializar una variable que describa cualquier parámetro del modelo, se pueden usar varias funciones. En una primera opción, se puede usar el propio constructor de la clase *Variable* de TensorFlow, el cual necesita un tensor de cualquier tipo y tamaño que sea el valor inicial que se le quiere dar a la variable, y las propiedades que uno necesite para modificar sus valores por defecto. Este valor inicial, le dará el tipo y tamaño a la variable, por tanto, tras la inicialización no se podrá cambiar, aunque su valor se podrá modificar mediante el método `tf.assign`. Un ejemplo de creación e inicialización de una variable que contiene los *weights* que conectan los nodos de una capa con la siguiente, sería:

```
weights = tf.Variable(tf.random_normal([300,200], stddev = 0.5), trainable =
    True, name = "weights")
```

Para darle un valor inicial a la variable, se ha hecho uso de la función `tf.random_normal()`, que genera un tensor usando una distribución aleatoria Gaussiana, según los argumentos que se impongan a la función, como su distribución estándar `stddev`. Además, del contructor hay que destacar el argumento `trainable` que permite indicar a TensorFlow que los valores de dicha variable se pueden modificar para conseguir optimizar la red neuronal durante el entrenamiento, el cual se puede poner a `False` si queremos evitar que aplique gradientes a dicha variable. En la mayoría de las funciones de TensorFlow encontramos el argumento `name`, que es un argumento opcional que permite elegir un identificador único a cada una de las operaciones generadas por dicha función (nodos del grafo) que servirá para darles nombres en el grafo creado por TensorFlow. Estos nombres identificativos son muy importantes ya que permitirán corregir el código cuando hay errores en la creación y diseño del grafo, aunque TensorFlow, en el caso de no especificarse ninguno, establecerá uno de forma automática.

Además de este método para crear variables existe otro equivalente, que suele ser el preferido a la hora de programar, y que es el que sigue:

```
weights = tf.get_variable(shape = [300,200], initializer = tf.random_normal_
    initializer(stddev = 0.5), trainable = True, name = "weights")
```

Si nos fijamos, en esta ocasión en vez de usar un constructor que cree una variable, se ha usado una función que devuelve una variable ya creada, y si no lo está, la crea según los argumentos impuestos. Para ello, en sus argumentos hay que indicarle todo lo que necesitamos para definir la variable, como su inicialización, que se establece mediante el argumento `initializer`, en el cual se ha usado el constructor de la clase `random_normal_initializer`, que devuelve un inicializador con las características impuestas mediante sus argumentos con funciones para devolver los valores iniciales necesarios para crear la variable. Si nos fijamos, este inicilizador lo que hace es crear una variable aleatoria con una distribución Gaussiana, que es el método usual para la inicialización de variables en las redes neuronales, al cual se le puede indicar la desviación estándar para ajustarse mejor a las características de nuestra red, como ya veremos. Debido a que se le indica un inicializador a usar en vez de un valor concreto, hay que definir su tamaño mediante el argumento `shape`.

Cuando se crea una variable en TensorFlow, en el grafo creado por defecto por TensorFlow se añadirán al inicio tres operaciones: en primer lugar, se crea el inicializador y el tensor que será el valor inicial de nuestra variable; luego, se le asigna dicho valor a la variable mediante el método `tf.assign` y por tanto ya tendremos nuestra variable inicializada, y por último, se ejecutará la operación que haga que se mantenga en memoria el estado de la variable, pudiendo asignarle en cualquier otro momento un valor cualquiera mientras que el grafo esté ejecutándose. Sin embargo, para que surjan estas tres operaciones, es necesario ejecutar la función `tf.global_variables_initializer()` como veremos más adelante, que desencadenará la ejecución dichas operaciones sobre todas las variables declaradas en el código.

Con esto, ya sabemos varios métodos para crear las variables que queramos y poder empezar ya a diseñar la red neuronal.

Una vez tenemos conocimientos sobre las variables y las operaciones de TensorFlow, sabremos casi todo lo que necesitamos para diseñar una red neuronal. Sólo falta saber cómo alimentamos a nuestra red neuronal con unos datos de entrada, y durante el entrenamiento, cómo indicamos cuáles son las salidas deseadas. Para ello no podremos usar una variable, ya que ésta se inicializa una única vez y mantiene su valor en memoria, por lo que queremos un componente que pueda ser reinicializado con otros valores tras cada ejecución del grafo. TensorFlow resuelve este problema mediante los llamados *placeholders*. Los *placeholders* son tensores a los que no se les impone ningún valor inicial, ya que simplemente lo que se hace tras declarar uno, es reservar un espacio de memoria que se mantendrá tras cada ejecución del grafo, lo que permitirá asignarle al tensor un valor inicial distinto guardado en memoria para ejecutar de nuevo el grafo; y por ello, son los componentes usados para introducir los datos de entrada a cualquier grafo programado con TensorFlow. Estos componentes se pueden usar como cualquier otra variable y tensor, pudiéndole aplicar también las operaciones que se mencionaron anteriormente. Un *placeholder* se puede definir de la siguiente forma:

```
x = tf.placeholder(tf.float32, shape=[None, 784], name = "x")
```

Como podemos observar, no se ha inicializado el *placeholder*, simplemente se le indica qué tipo de variable va a ser y su tamaño. Hay que destacar que en el tamaño del *placeholder* se suele imponer en la primera dimensión un valor de `None` que quiere decir que su magnitud no está especificada y podrá ser cualquiera. Esto se suele hacer porque el valor de la primera dimensión de los datos de entrada a una red neuronal corresponde con el número de ejemplos que forman el *minibatch*, y, al no estar especificado, podremos alimentar a la red con el número de ejemplos que queramos.

Con esto, ya tenemos todos los componentes necesarios para diseñar nuestra red neuronal por medio de un grafo de flujo de datos, sin embargo, una vez diseñada y programada, sólo habremos declarado el formato de nuestro grafo, por lo que nos faltaría ejecutarlo para obtener valores numéricos de él.

Para ejecutar un grafo computacional en TensorFlow se usan sesiones (`tf.Session`). Una sesión permite inicializar todas las variables y ejecutar el grafo o subgrafo que le indiquemos. Por ejemplo, podemos considerar el siguiente código:

```
import tensorflow as tf
```

```
#Creamos el grafo Computacional.
#Primero la creacion de variables:
x = tf.placeholder(tf.float32, shape=[None, 2], name = "input")
w = tf.Variable(tf.random_normal([2, 1]), name = "weights")
b = tf.Variable(tf.zeros(shape = [1,1]), name = "biases")

#Segundo la realizacion de las operaciones necesarias:
b = tf.add(b,[[10]])
output = tf.add(tf.matmul(x, w) , b)

#Por ultimo, se ejecuta el grafo Computacional
sess = tf.Session()
sess.run(tf.global_variables_initializer())
numerical_out = sess.run(output, feed_dict = {x:[[1,1],[10,15],[30,50]]} )

print('Output = ', numerical_out)
```

Como se puede ver, en el código lo que estamos haciendo es declarar un subgrafo mediante la llamada de distintas funciones, y tras ello, se crea una sesión con el uso del comando `tf.Session()`. Tras la creación de la sesión, es necesario la inicialización de las variables, y para ello debemos ejecutar a través del método `run` de la sesión `sess` la función `tf.global_variables_initializer()`. Esto nos permitirá obtener valores numéricos en los tensores de `w` y `b`. Una vez hecho esto, se puede proceder a ejecutar las operaciones que calculen el valor de cualquier tensor o tensores indicándolo en los argumentos de `run`. En este caso, se quiere calcular el tensor `output`, que depende de `x`, y al ser un *placeholder*, se le debe indicar a la función `run` mediante el argumento `feed_dict`, el valor numérico de los *placeholders* utilizados y así poder ejecutar el subgrafo que genera el valor del tensor `output`. Sin embargo, en este caso se ha introducido como valor del *placeholder* una matriz, cuando el *placeholder* tiene dimensiones de vector. Esto provoca que TensorFlow coja de la matriz los datos que correspondan con el tamaño indicado en la definición del *placeholder*, y ejecutará el grafo con cada uno de los valores tantas veces como sea necesario hasta finalizar la matriz. Es decir, en el ejemplo, primero cogería de la matriz el vector `[1,1]` y calculará el valor de salida del grafo, luego, hará lo mismo con el vector `[10,15]`, así hasta finalizar la matriz. De esta forma, al ejecutar el código, TensorFlow nos devolverá una matriz con tres resultados correspondientes a los tres valores de `output` para cada valor del *placeholder*. Por ello mismo, la primera dimensión del *placeholder* se estableció a `None`, ya que nos permitirá alimentar al grafo con el número de datos de entrada que queramos, en este caso tres.

Además, hay que destacar que antes de definir el tensor `output`, se ha cambiado el valor del tensor `b`, por tanto, en nuestro subgrafo, antes del cálculo de `output`, se han añadido las operaciones necesarias para actualizar `b` tal y como se ve en la Figura 2.2, ya que `output` depende del valor de `b`. De hecho, cada vez que se ejecuta `sess.run()`, la función lo que hace es recorrer el subgrafo indicado desde el final (resultado que queremos obtener) hasta el inicio para comprobar todas las dependencias existentes (incluyendo la creación de variables añadiendo las tres operaciones que se mencionaron anteriormente), luego comprobará que todos los valores aportados por `feed-dict` corresponden a lo que se ha definido al declarar los distintos *placeholders*, y por último recorrerá de nuevo el subgrafo computacional desde el inicio hasta el final para calcular el resultado.

Mediante el comando `sess.run()` no sólo se puede calcular los valores de cualquier tensor, sino que puede ser usado hasta para entrenar redes neuronales como ya veremos. Esto se debe al gran potencial que tiene el hecho de definir con código grafos computacionales.

Ahora que sabemos cómo se crean los grafos computacionales en TensorFlow, es necesario aclarar ciertos conceptos a la hora de la creación de variables. Por ejemplo, veamos el siguiente caso:

```
import tensorflow as tf

#Definimos un primer modelo:
```

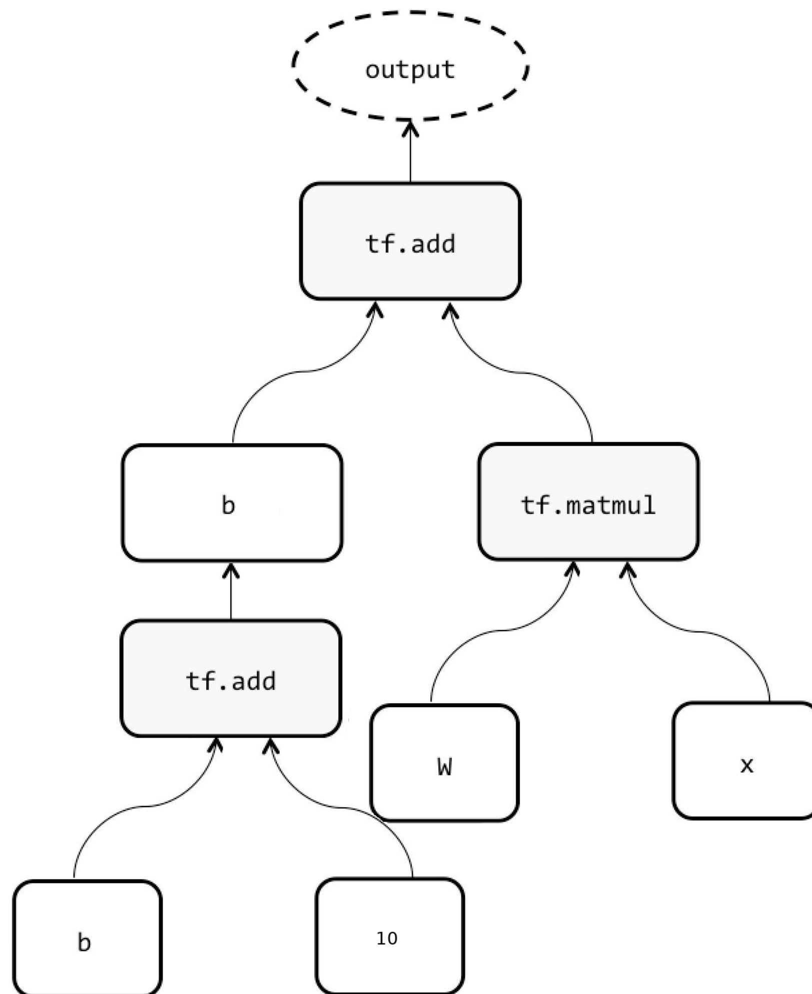


Figura 2.2 Representación gráfica del subgrafo computacional programado anteriormente.

```
def modelo1():
  x = tf.placeholder(tf.float32, shape=[None, 2], name="input1")
  w = tf.Variable(tf.random_normal([2, 1]), name="weights1")
  b = tf.Variable(tf.zeros(shape = [1,1]), name="biases1")

  output = tf.add(tf.matmul(x, w) , b)
  return output

#Definimos un segundo modelo:
def modelo2():
  x = tf.placeholder(tf.float32, shape=[None, 2], name="input2")
  w = tf.get_variable(shape = [2,1], initializer = tf.random_normal_initializer
      (), name = "weights2")
  b = tf.get_variable(shape = [1,1], initializer = tf.zeros_initializer(), name
       = "biases2")

  output = tf.add(tf.matmul(x, w) , b)
  return output

#Creamos dos subgrafos computacionales de model1:
out1_modelo1 = model1()
out2_modelo1 = model1()
print('Creados ambos subgrafos del primer modelo')
```

```
#Creamos dos subgrafos computacionales de model2:
out1_modelo2 = model2()
out2_modelo2 = model2()
print('Creados ambos subgrafos del segundo modelo')
```

Vemos que en dicho código se han declarado dos pares de subgrafos exactamente iguales, sólo que en un grupo de ellos se ha utilizado la función `tf.Variable()` para crear las variables, mientras que en el otro se ha utilizado `tf.get_variable()`. Cuando lo ejecutamos, al pertenecer todos los subgrafos al mismo grafo computacional por defecto de TensorFlow, veremos que los dos subgrafos del primer modelo se crean correctamente, sin embargo, a a la hora de la declaración del segundo subgrafo del segundo modelo, `out2_model2`, salta un error. Esto se debe a que a diferencia del primer modelo, en este caso, al estar ya creado un subgrafo que posee las variables `input2`, `weights2` y `biases2`, siendo éste `out1_model2`, la función `tf.get_variable()` no creará variables nuevas, a diferencia de lo que haría `tf.Variable()`, sino que intentará reutilizar esas variables ya creadas. Sin embargo, TensorFlow no permite compartir de forma predeterminada las variables de distintos subgrafos si se encuentran en el mismo grafo (en este caso, en el predeterminado de TensorFlow), con lo que lanzará un error. En un principio, no se va a crear más de un grafo en el mismo código ya que no se van a crear modelos tan complejos, por lo que hay que tener cuidado a la hora de crear subgrafos que tengan variables u operaciones con el mismo nombre. Sin embargo, en el primer caso, al usar `tf.Variable`, se crearán dos subgrafos computacionales con variables totalmente distintas, de hecho, TensorFlow a la hora de crear el segundo subgrafo `out2_model1`, al saber que existen ya las variables `input1`, `weights1` y `biases1`, creará unas nuevas variables pero con nombres distintos: `input1_1`, `weights1_1` y `biases1_1`.

Debido a esto, es preferible siempre usar la función `tf.get_variable()`, ya que evitará que surjan errores cuando se programe los grafos computacionales.

FeedForward Network en TensorFlow

Con esto ya tenemos todos los conocimientos necesarios para poder manejar TensorFlow y empezar a crear redes neuronales mediante grafos computacionales. En primer lugar, crearemos una red neuronal sencilla, como hicimos en los apartados teóricos, por ello, se creará una *FeedForward Network* con tres *hidden layers*.

Para hacer el modelo de red neuronal, se procede a crear e inicializar las variables que formarán el vector de parámetros, y tras ello, se define el conjunto de operaciones a realizar sobre dichos tensores:

```
def neural_network_model(data):

    #En esta funcion estamos definiendo el subgrafo de la red
    #neuronal

    hidden_layer_1={'weights': tf.get_variable('weights_l1', shape = [size_
        images, n_nodes_hl1], initializer = tf.random_normal_initializer(),
        trainable = True), 'biases': tf.get_variable('biases_l1', shape = [n_
        nodes_hl1], initializer = tf.random_normal_initializer(), trainable =
        True)}

    hidden_layer_2={'weights': tf.get_variable('weights_l2', shape = [n_nodes_
        hl1, n_nodes_hl2], initializer = tf.random_normal_initializer(),
        trainable = True), 'biases': tf.get_variable('biases_l2', shape = [n_
        nodes_hl2], initializer = tf.random_normal_initializer(), trainable =
        True)}

    hidden_layer_3={'weights': tf.get_variable('weights_l3', shape = [n_nodes_
        hl2, n_nodes_hl3], initializer = tf.random_normal_initializer(),
```

```
        trainable = True), 'biases': tf.get_variable('biases_l3', shape = [n_
        nodes_hl3], initializer = tf.random_normal_initializer(), trainable =
        True)}

    output_layer={'weights': tf.get_variable('weights_out', shape = [n_nodes_hl
        5, n_classes], initializer = tf.random_normal_initializer(), trainable
        = True),'biases': tf.get_variable('biases_out', shape = [n_classes],
        initializer = tf.random_normal_initializer(), trainable = True)}


    layer1 = tf.add(tf.matmul(data , hidden_layer_1['weights']), hidden_layer
        _1['biases'])
    layer1 = tf.nn.relu(layer1)

    layer2 = tf.add(tf.matmul(layer1 , hidden_layer_2['weights']), hidden_layer
        _2['biases'])
    layer2 = tf.nn.relu(layer2)

    layer3 = tf.add(tf.matmul(layer2 , hidden_layer_3['weights']), hidden_layer
        _3['biases'])
    layer3 = tf.nn.relu(layer3)

    output = tf.add(tf.matmul(layer3 , output_layer['weights']), output_layer['
        biases'])

    return output
```

Sin embargo, esta forma de definir el subgrafo que formará el modelo de la red neuronal resulta un poco complicado y difícil de leer. Por ello, TensorFlow ayuda mediante una función llamada `tf.variable_-scope()`, que nos permitirá controlar de forma segura los nombres de las variables para evitar que tengan la misma etiqueta y surjan los errores como se vió en el apartado anterior tras usar `tf.get_variable()`. Veamos como quedaría el código usando esta función:

```
def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de las hidden layers

    weights = tf.get_variable(shape = weight_shape, initializer = tf.random_
        normal_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.random_normal
        _initializer(), trainable = True, name = 'b')

    output = tf.nn.relu( tf.add( tf.matmul( input , weights ) , biases ))
    return output

def out_layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de la capa de salida

    weights = tf.get_variable(shape = weight_shape, initializer = tf.random_
        normal_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.random_normal
        _initializer(), trainable = True, name = 'b')

    output = tf.add( tf.matmul( input , weights ) , biases )
    return output

def neural_network_model(data):
```

```
    #En esta funcion estamos definiendo el grafo de la red
    #neuronal

    with tf.variable_scope('layer_1'):
        layer1_out = layer(data,
          [size_images, n_nodes_hl1] , [1,n_nodes_hl1] )

    with tf.variable_scope('layer_2'):
        layer2_out = layer(layer1_out,
          [n_nodes_hl1, n_nodes_hl2] , [1,n_nodes_hl2] )

    with tf.variable_scope('layer_3'):
        layer3_out = layer(layer2_out,
          [n_nodes_hl2, n_nodes_hl3] , [1,n_nodes_hl3] )

    with tf.variable_scope('output'):
        output = out_layer(layer3_out,
          [n_nodes_hl3, n_classes] , [1,n_classes] )

    return output
```

De esta forma, podemos usar un conjunto de operaciones definidas en una misma función que generan variables, creándose distintos subgrafos (dependiendo del *scope*) que se encuentran en el grafo por defecto, y por lo tanto, creándose distintas variables, evitando el error que surge cuando se usa el mismo nombre cuando nos referimos a diferentes variables. De hecho, lo que ocurre es que cuando declaramos un *scope* con la función `tf.variable_scope()`, podremos llamar a las operaciones que generan variables, en este caso `tf.get_variable()`, y aunque estas operaciones indiquen el mismo nombre en el argumento `name`, el *scope* creado hace que al nombre de dicha variable se le añada el string indicado. Por ejemplo, en el caso del *scope* de `layer_1`, se crearán las variables `layer_1/w` y `layer_1/b`, evitando que se mezclen con las variables que se crearán para el resto de capas. Esto nos permitirá tener un código mucho más limpio y reducido, ya que podremos aprovechar aquellas operaciones que generen variables de una misma forma.

Con este código tendremos las llamadas a las funciones necesarias para crear un subgrafo que modele nuestra *FeedForward Neural Network* y que genere una salida dada una entrada a partir de un *placeholder*.

Ahora que tenemos el modelo de la red neuronal, vamos a proceder a crear la función que genere un subgrafo para entrenarla:

```
def train_model(cost):

    #Learning rate por defecto = 0.001
    optimizer = tf.train.AdamOptimizer()

    #La funcion minimize combina las funciones:
    #compute_gradients y apply_gradients
    train_operation = optimizer.minimize(cost)

    return train_operation
```

Como podemos ver, el entrenamiento de una red neuronal se puede hacer de forma muy sencilla con TensorFlow. Debemos crear las operaciones que generen el optimizador que se quiere usar, en nuestro caso, se va a usar el algoritmo *Adam*, que se le dejará el *learning rate* por defecto. Luego, se realiza el proceso de entrenamiento en sí, y para ello se usa el método del optimizador `minimize`, el cual lo que hace es minimizar el valor indicado en su argumento (en este caso el coste total de la red neuronal) por medio del cambio de valor de todas las variables declaradas que se puedan entrenar (es decir, aquellas con

el argumento `trainable = True`). Este método no es más que la unión de dos métodos del optimizador: `compute_gradients` y `apply_gradients`, cuyos funcionamientos son bastantes descriptivos.

El siguiente paso, es crear la función que genere el subgrafo para el cálculo del coste de la red neuronal:

```
def loss(prediction, y):

  #Calcula el error de 'prediction' teniendo los resultados
  #deseados 'y'
    entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits = prediction,
        labels = y)

    #A partir del error en las salidas, calculamos un valor
    #unico haciendo la media del error
    cost = tf.reduce_mean(entropy)

    return cost
```

Esta función es muy importante ya que la primera llamada que realiza, no sólo calcula el error, sino que también añade sobre la salida de la red neuronal la función *softmax*. Si nos fijamos, a la salida del modelo de la red neuronal no se le aplicó dicha función, por lo que sus salidas no representan las probabilidades que tiene cada clase de ser la correcta, es decir, no posee las salidas escaladas. Esto no es un problema en un principio, ya que aquella salida que posea un valor más alto sin aplicarle la función *softmax*, será la correcta ya que será la que más probabilidades tenga. Sin embargo, en este caso, al tratarse de un problema de clasificación en el que las clases son mutuamente excluyentes y se usa el formato *one-hot encoding*, se va a utilizar la *Cross Entropy Loss Function*, que calcula el error de una clasificación cuyo resultado se da con valores, en un rango entre 0 y 1, que representan la probabilidad de cada clase de ser la correcta. Entonces, esta función incluye la función *softmax* para poder escalar los valores de la clasificación, y además de esta forma, la función podrá calcular el error de una forma mucho más eficiente que calculando *softmax* aparte. Finalmente, por tener un valor normalizado del error, se realiza la media de los errores obtenidos en las clasificaciones de cada uno de los ejemplos del *minibatch*, obteniendo un único valor que representará el coste de la red neuronal ante un conjunto de ejemplos, valor que tendrá que minimizar el optimizador *Adam*.

Por último, creamos la función que genere el subgrafo que calcule la precisión en las predicciones de la red neuronal según el *test dataset*:

```
def eval_test(prediction, y):

    #Se comprueba que la salida corresponde con la deseada
    correct = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))

    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    return accuracy
```

Este código ya resulta muy sencillo comparado con el resto, puesto que simplemente calculamos respecto a un conjunto de ejemplos si se ha acertado en la predicción o no, y respecto a todos esos resultados, hacemos la media para tener un valor que representará el porcentaje de aciertos de la red neuronal sobre los ejemplos de testeo.

Con esto, ya tenemos definidos todos los subgrafos necesarios, sólo falta definir la red neuronal y ejecutar todos los subgrafos:

```
#Cargamos los dataset de entrenamiento y de testeo:
```

```
train_x , labels , train_y = load_images_and_labels_arrays('Datasets/Train_
    Dataset')

size_images = np.shape(train_x)[1]
n_train_examples = np.shape(train_x)[0]
print('Loaded ' + str(n_train_examples) + ' Training Images')

test_x, labels, test_y = load_images_and_labels_arrays('Datasets/Test_Dataset')
print('Loaded ' + str(np.shape(test_x)[0]) + ' Test Images')

#Modelo de la red neuronal
n_nodes_hl1 = 2000
n_nodes_hl2 = 1000
n_nodes_hl3 = 500

batch_size = 100
n_batches = n_train_examples/batch_size
n_epochs = 100
#One-hot encoding
n_classes = 3

#Las imagenes no estan dadas como una matriz (height x width),
#sino como un vector. Debido a ello, tendremos que el shape de
#x (entrada de la red), no tendra altura determinada (ejemplos
#del batch), y el ancho sera de size_images, que en nuestro
#caso es 112x112x3 (RGB)
x = tf.placeholder(tf.float32, shape=[None, size_images])
y = tf.placeholder(tf.float32, shape=[None, n_classes])

#Creamos el subgrafo de la red neuronal
prediction = neural_network_model(x)

#Creamos el subgrafo que calcule el error de la red neuronal
cost = loss(prediction, y)

#Creamos el subgrafo que entrene la red neuronal
train_op = train_model(cost)

#Creamos el subgrafo que calcule el porcentaje de aciertos sobre
#las imagenes de testeo
accuracy = eval_test(prediction,y)

#Iniciamos sesion para poder ejecutar todo nuestro modelo y
#poder entrenarlo
with tf.Session() as sess:

    #Iniciamos todas las variables como weights y biases
    sess.run(tf.global_variables_initializer())

    prev_acc = 0
    #Bucle para completar todos los epochs
    for epoch in range(n_epochs):
        total_epoch_cost = 0
        avg_epoch_cost = 0

        i=0
```

```
        #Bucle para los minibatches
        while(i < n_train_examples):
            start = i
            end = i + batch_size

            batch_x = train_x[start:end][:]
            batch_y = train_y[start:end][:]

            sess.run(train_op, feed_dict = {x: batch_x ,
              y: batch_y})
            minibatch_cost = sess.run(cost, feed_dict =
              {x: batch_x, y: batch_y})

            total_epoch_cost += minibatch_cost
            i += batch_size

        avg_epoch_cost = total_epoch_cost/n_batches

        print('Epoch', epoch+1, 'completed out of', n_epochs,
          'cost:', avg_epoch_cost)

        acc = sess.run(accuracy, feed_dict = {x:test_x,
          y:test_y})*100

        if acc < prev_acc:
            prev_acc = acc
            print('Test accuracy: ' + str(acc) + ' %')
            print('Possible overfitting!!')
            ans = raw_input('Continue?(y/n): ')
            if ans == 'n':
                break
        elif acc > 80:
            print('Test accuracy: ' + str(acc) + '%')
            ans = raw_input('High accuracy... Continue?(y/n): ')
            if ans == 'n':
                break
        else:
           print('Test accuracy: ' + str(acc) + '%')
           prev_acc = acc

    acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y})*100
    print('Accuracy: ' + str(acc) + ' %')
```

Como vemos, se ha definido la red neuronal imponiendo el número de neuronas de cada capa, se han inicializado los *placeholders*, se han llamado a todas las funciones definidas anteriormente para crear los subgrafos, y finalmente se ha creado una sesión para ejecutarlos. Si nos fijamos, por cada *minibatch*, cogemos un número de ejemplos igual al tamaño del *minibatch* impuesto, y se ejecuta primero el subgrafo que entrena la red neuronal, calculando en un inicio la media del error de la red neuronal para ese conjunto de ejemplos con la función `loss`, y cambiando luego el valor de todas las variables que forman la red neuronal; y en segundo lugar, se ejecuta sólo el subgrafo que calcula el coste para saber la media del error de la red neuronal con las variables entrenadas en ese *minibatch*. Cada vez que se ejecuta un *minibatch*, se suma su error para más tarde hacer la media y saber el error total de la red neuronal en dicho *epoch*. Además, por cada *epoch*, se calcula el porcentaje de aciertos de la red neuronal ante los ejemplos de testeo, para así tener cierto control para evitar entrenamientos irregulares o con *overfitting*. Finalmente, una vez acabado el entrenamiento de la red, se vuelve a calcular el porcentaje de aciertos de la red neuronal respecto los ejemplos de testeo impuestos y asegurarse que la red neuronal está correctamente entrenada.

Conclusión

Con respecto a todo lo anterior, hemos conseguido un código con una buena *Deep Neural Network* que puede ser entrenada para cualquier tipo de datos. Simplemente, es necesario obtener unos buenos *datasets* para entrenarla, y a partir de ahí, ajustarla para conseguir los mejores resultados posibles: cambiando el número de ejemplos del *minibatch*, de *epochs*, de *hidden layers*, de neuronas en las distintas capas...

Para comprobar el funcionamiento de la red neuronal, se probó en dos *datasets*. En los Anexos 2 y 3, podremos encontrar el código de las dos redes neuronales programadas específicamente para ser usadas en cada uno de ellos.

El primer dataset es MNIST, un *dataset* que se considera como el *"Hello, world!"* de las redes neuronales. Se compone de 60000 imágenes de entrenamiento y 10000 de testeo con un tamaño de 28x28 en las que aparecen imágenes de números escritos a mano entre el 0 y el 9, y por tanto, con 10 clases distintas que son mutuamente excluyentes. Cuando es entrenada la red neuronal (en Anexo 2), se llega a conseguir hasta un 97.8 % de aciertos en los ejemplos de testeo. Estos resultados son muy buenos, pero teniendo en cuenta que dicho *dataset* es muy sencillo para las redes neuronales, es un resultado bastante mejorable.

A parte de dicho *dataset*, se creó otro pequeño formado por 5000 imágenes de entrenamiento y 100 de testeo de frutas y verduras. Para elaborar dicho *dataset*, se utilizaron dos *scripts* de Python que se pueden encontrar en los Anexos 4 y 5. El primero de ellos sirve para tomar imágenes de forma continua (como si de un vídeo se tratara) y guadarlas en las carpetas según el tipo de fruta o verdura. El otro *script*, lo que hace es procesar dichas imágenes, es decir, como nuestra red neuronal acepta como entrada un array de datos, lo que se hizo fue en primer lugar, cambiar el tamaño de todas las imágenes para que fueran tres matrices (imágenes RGB) con un tamaño de 112x112, y luego, se transformaron a un único array con tamaño 1x112·112·3, de esta forma, obtuvimos un *dataset* formado por una matriz en la que cada fila correspondía a una única imagen (por ejemplo, el dataset de entrenamiento tendría un tamaño de 5000x37632). Además, el *script* lo que hace es crear otras dos matrices de etiquetas (*labels*) para cada una de las imágenes con el objeto de estar ya clasificadas, una de ellas en formato *one-hot encoding*, por lo que dicha matriz es perfecta para comprobar las predicciones de la red neuronal y entrenarla. Con esto, tenemos por tanto, una matriz con ejemplos de entrenamiento y otra con las salidas deseadas para cada uno de los ejemplos.

Una vez entrenada una red neuronal diseñada específicamente para este *dataset* (en Anexo 3), se consiguió en los ejemplos de testeo una precisión del 80 %, que no está mal, aunque sigue siendo una precisión bastante baja ya que no siempre se conseguía. Estos resultados empeoraban con imágenes tomadas en otras situaciones, debido a que la relación señal-ruido es baja, porque por ejemplo, la iluminación, el fondo de la imagen y otros factores podían interferir en gran medida en el resultado de la red neuronal.

Con esto, debe quedar claro que una red neuronal *FeedForward* no está preparada para procesar y clasificar imágenes, aunque si podría estarlo para otro tipo de tareas.

Por ello, en los siguientes apartados estudiaremos la *Convolutional Neural Network*, un tipo de red neuronal especialmente diseñada para procesar y clasificar imágenes.

Convolutional Neural Networks

Introducción

Como ya se ha comprobado, las *FeedForward Networks* no tienen el potencial suficiente para procesar imágenes, por lo que, se deben encontrar otras alternativas que nos permitan obtener buenos resultados.

Uno de los problemas de visión por computador que inicialmente era difícil, fue encontrar un algoritmo que fuera capaz de detectar si en una imagen hay una cara humana o no, tarea muy sencilla para un ser humano pero compleja para un computador. Este problema lo abordaron Paul Viola y Michael Jones en 2001, proponiendo una solución que permitía a un algoritmo aprender con gran eficacia a distinguir caras humanas en imágenes. Para ello, se propuso un algoritmo de *machine learning* al cual se le introducen como entradas imágenes que poseyeran o no caras humanas, y, de esta forma, el algoritmo terminaría construyendo un clasificador capaz de realizar dicha tarea. Sin embargo, esto no era posible, sobre todo porque la relación señal-ruido en una imagen suele ser demasiado baja, por lo que el algoritmo no sería capaz de encontrar un clasificador eficaz.

La solución que propusieron Paul Viola y Michael Jones, consistía en que, para aumentar la relación de señal-ruido, se le indicaría al algoritmo aquellas características que se creen que son importantes para identificar una cara humana. Por ejemplo, estos investigadores propusieron características basadas en determinados patrones de iluminación que se suelen dar en caras: la primera consistía en que normalmente la zona del puente de la nariz se encuentra más iluminada que a los lados de la misma, y la segunda, que las mejillas también suelen estar más iluminadas que la zona de los ojos.

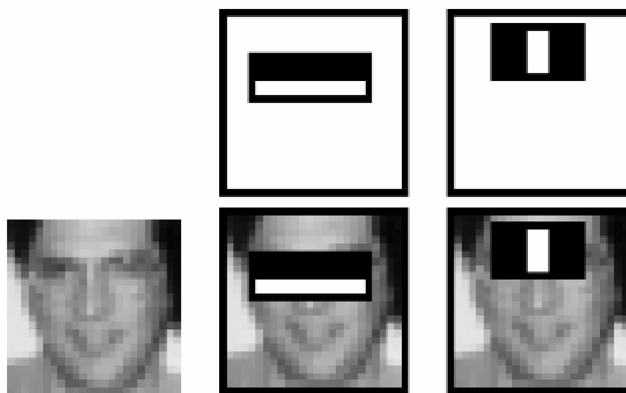


Figura 3.1 Características de caras humanas escogidas por Paul Viola y Michael Jones para introducirlo en un algoritmo de *machine learning*.

Cuando ambas características se usan en un algoritmo de *machine learning*, la efectividad del algoritmo aumenta considerablemente, pudiendo encontrar un clasificador capaz de detectar caras humanas con una precisión superior al 90 %. Sin embargo, este algoritmo sólo funcionará con imágenes que posean unas condiciones determinadas, como caras humanas vistas de frente y sin estar cubiertas, y con una iluminación concreta. Esto implica, que el algoritmo realmente sigue sin ”saber” qué es una cara, por tanto, se necesitaría un gran número de características para introducirlo en el algoritmo y que éste fuera capaz de generar un clasificador para identificar cualquier cara en cualquier situación. Esto implicaría hacer una gran búsqueda de todas esas características que nos permitiera tener un algoritmo de *machine learning* realmente eficaz, algo que con problemas más complejos, sería prácticamente inviable.

El problema de la búsqueda de características se resuelve gracias al *Deep Learning*, debido a que las redes neuronales son perfectas para este tipo de trabajo, ya que como ya sabemos, cada una de las capas que forma la red neuronal es capaz de modelar y extraer características cada vez más complejas que representan los datos de entrada. En concreto para el procesamiento de imágenes, existe una *Deep Neural Network* especial que permite el tratamiento de imágenes de una forma mucho más eficaz que cualquier otra red neuronal, siendo éstas las *Convolutional Neural Networks*.

Estas redes neuronales se basan en el funcionamiento del cortex visual del cerebro, que es la parte que se encarga del procesamineto de las imágenes. Para ello, el cerebro captura cierta información a partir de la luz que reciben los ojos, la cual, es más tarde procesada y transportada al cortex visual del cerebro, para finalmente analizarla. Todo este procedimiento es realizado únicamente por unas neuronas especializadas en el procesamiento de imágenes, y por tanto, es lógico transferir este proceso biológico a un modelo matemático para poder construir un modelo de red neuronal que sea eficaz ante imágenes. La primera red neuronal de este tipo fue LeNet-5, creada en 1998 por LeCun, que aunque estuviera basada en el funcionamiento del cortex visual, no pudo demostrarse su eficacia de forma experimental hasta mucho más tarde, tras la implementación de las GPU para el procesamiento de redes neuronales.

Para demostrar el gran potencial de este tipo de redes neuronales, se puede poner de ejemplo el caso del reto ImageNet, uno de los retos más complicados en clasifiación de imágenes. Cada año muchos investigadores se proponen realizar un algoritmo que permita clasificar las 450.000 imágenes en una de las posibles 200 clases existentes. En el caso de un humano, el porcentaje de aciertos en el reto ronda el 96%; en 2011 el algoritmo ganador obtuvo un error del 25.7%. Sin embargo, fue en 2012 cuando Alex Krizhevsky, fue pionero en este reto introduciendo una *Deep Convolutional Neural Network* pudiendo mejorar el error a un 16%, algo que nunca se había conseguido hasta la fecha. Esta red neuronal se denominó AlexNet, y provocó gran expectación sobre las redes neuronales, pudiéndose demostrar el gran potencial que tienen las *Deep Neural Networks* sobre cualquier otro algoritmo de *machine learning*, y sobre todo, se demostró finalmente el potencial que tienen las *Convolutional Neural Networks* en visión por computador, revolucionando dicho campo. A partir de ese momento, este tipo de redes neuronales se empezaron a investigar y a usar para aprovechar toda su capacidad, permitiendo tener resultados muy parecidos a los que obtendría un ser humano.

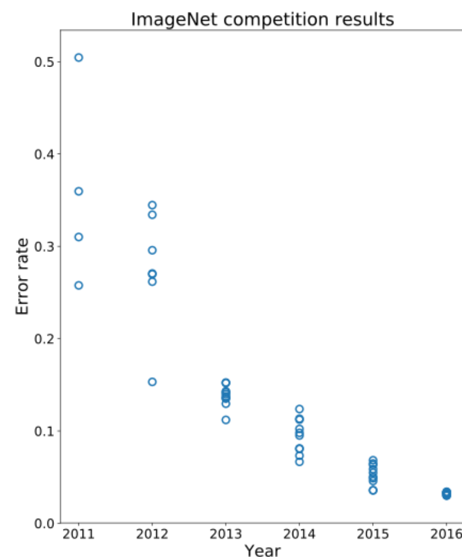


Figura 3.2 Errores a lo largo de los años en el reto ImageNet. Se puede observar la gran diferencia que hubo en 2012 gracias al potencial de AlexNet.

AlexNet fue, pues, la primera red neuronal que revolucionó realmente estos campos, demostrando la gran capacidad que tienen dichas redes neuronales para capturar las distintas características de las imágenes permitiendo su clasificación con errores mínimos.

Por ello, en este proyecto se utilizará una *Convolutional Neural Network*, ya que nos permitirá obtener unos resultados que superarán con seguridad los obtenidos por las *FeedForward Networks*.

Conceptos de Convolutional Neural Networks

Introducción

Una de las grandes desventajas de las *FeedForward Networks* es que su estructura se basa en que todas las neuronas de una capa se encuentran conectadas con todas las neuronas de la capa siguiente, es decir, se tratan de *fully-connected networks*. En tareas de clasificación de imágenes, esta situación puede ser un tanto derrochadora, ya que, sobre todo cuando tratamos imágenes a color, implicaría tener cientos de miles de *weights* en las primeras *hidden layers*. El hecho de tener tantas variables implica normalmente un *overfitting* en el entrenamiento de la red neuronal, ya que la relación de número de ejemplos de entrenamiento entre número de variables, se hace muy pequeña.

Sin embargo, las *Convolutional Neural Networks* evitarán estos problemas, ya que el número de variables se verá drásticamente reducido, gracias al hecho de que sabemos que estamos analizando imágenes. Para ello, la estructura de este tipo de *Deep Neural Network* es un tanto diferente a la que nos encontramos en las *FeedForward Networks*, ya que las capas que forman la red neuronal, serán capas organizadas de forma tridimensional, pudiendo decirse que cada una de ellas posee un ancho, un alto y una profundidad, tal y como se muestra en la Figura 3.3.

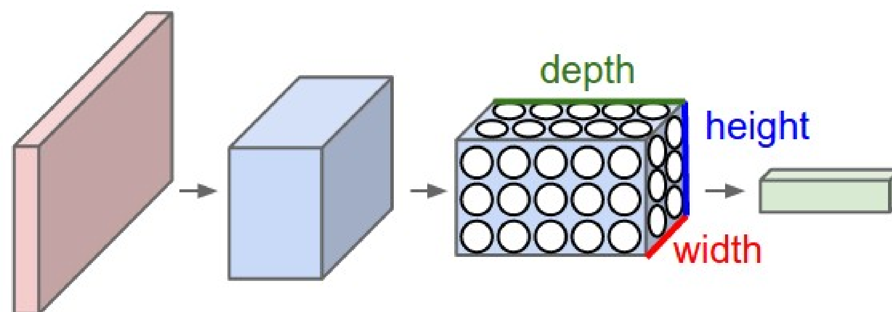


Figura 3.3 Estructura de una *Convolutional Neural Network*, en la que cada capa se considera como un espacio tridimensional.

Como veremos más adelante, cada una de las neuronas que forman cada capa, están solamente conectadas a una pequeña parte de las neuronas que forman la capa anterior, por lo que de esta forma, nos evitamos la ineficiencia del uso de gran cantidad de variables como en las *fully-connected networks*. Sin embargo, suele ser común que en *Convolutional Neural Networks* nos encontremos que las últimas capas sean de tipo *fully-connected* para así poder relacionar toda la información que se ha podido extraer de las imágenes y realizar una tarea de clasificación.

Filtros y Convoluciones

Uno de los estudios más importantes que ayudaron al desarrollo de este tipo de redes neuronales, fue el realizado por David Hubel y Torsten Wiesel en 1959. En este estudio, utilizaron un gato al que le insertaron electrodos en el cerebro y le proyectaron imágenes en blanco y negro en una pantalla. Gracias a este estudio, se dieron cuenta, de que el cortex visual del cerebro estaba formado por capas, y que cada una de ellas, trabajaba sobre las características detectadas por la capa anterior. Por ejemplo, por medio de la detección de líneas de una capa, la siguiente podría detectar los contornos, luego la forma, y por último un objeto en concreto. Además, hicieron otro descubrimiento muy importante, cuando el gato veía imágenes con líneas verticales, sólo un conjunto de neuronas se disparaban, mientras que otras sólo se disparaban con imágenes que poseían líneas horizontales, y por tanto, según el ángulo que poseía cada línea que observaba, se activaban unas neuronas u otras.

Este concepto, no es más que el de filtro, es decir, un detector de características de una imagen. Por ello, Paul Viola y Michael Jones no estaban lejos cuando mencionaban que era necesario imponer ciertos filtros a

los algoritmos para detectar ciertas características en una imagen.

Un filtro, también llamado *kernel*, matriz de convolución o máscara en procesamiento de imagen, no es más que una pequeña matriz cuadrada que permiten la modificación de una imagen para destacar ciertas características. Por ejemplo, es común el uso de *kernels* para difuminar imágenes, enfocarlas e incluso para detección de contornos. Este proceso se consigue mediante la convolución de dicha matriz y la imagen original.

El proceso de convolución no es más que el hecho de sumar el valor de cada píxel con el de sus vecinos en una porción de la imagen, ponderados respecto los del filtro. Esto se basa en la convolución matemática, sólo que en este caso se está produciendo en un sistema de dos dimensiones, ya que trabajamos con matrices.

La convolución matemática se define como:

$$\begin{aligned}(f*g)(t) &\overset{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \\ &= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau \end{aligned} \tag{3.1}$$

donde $f(t)$ y $g(t)$ son las dos funciones continuas a las que le queremos aplicar la convolución, y donde t no tiene por qué ser el dominio temporal, aunque suele ser lo común. De esta forma, podemos decir que la convolución es un proceso que se realiza sobre dos funciones para producir una tercera, llamada convolución, que expresa cómo la forma de una de las funciones es modificada por la otra función. De hecho, la convolución es una operación matemática muy importante a la hora de trabajar con sistemas LTI, ya que nos permitirá hallar la respuesta del sistema LTI ante una señal de entrada, relacionando de forma directa el dominio temporal y el dominio frecuencial mediante la transformada de Laplace, siendo equivalente la convolución en el dominio temporal con el producto en el dominio frecuencial.

Sin embargo, en nuestro caso no se van a trabajar con funciones continuas sino con funciones discretas, ya que trabajamos con píxeles. Si la definición de convolución la transformamos a un sistema discreto obtendremos lo siguiente:

$$(f*g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{3.2}$$

A pesar de ello, es usual trabajar con funciones con un número de valores finitos, por lo que la convolución discreta pasará a tener forma:

$$(f*g)[n] = \sum_{m=-M}^{M} f[m]g[n-m] \tag{3.3}$$

Sin embargo, debido al procesamiento de imágenes y vídeos, la convolución en sistemas multidimensionales pasó a tener una gran importancia, sobre todo, como ya hemos mencionado, para la aplicación de filtros sobre imágenes. Puesto que vamos a trabajar con imágenes, el tipo de convolución a tratar es la convolución 2D, por tanto, sabiendo cómo es la convolución discreta, la convolución discreta en sistema bidimensional es:

$$(f*g)[n_1,n_2] = \sum_{m_1=-M_1}^{M_1} \sum_{m_2=-M_2}^{M_2} f[m_1,m_2]g[n_1-m_1,n_2-m_2] \tag{3.4}$$

Con esto, podremos aplicar un filtro bidimensional sobre una imagen, y de esta forma calcular la convolución.

El proceso de convolución entre filtros e imágenes, consiste en sumar los valores de todos los píxeles de una porción de la imagen equivalente al tamaño del filtro, ponderados por los valores del filtro, estando éste centrado sobre el píxel de la imagen del cual queremos calcular la convolución. Por ejemplo, si tenemos

dos matrices, siendo la primera el filtro, $f(n_1,n_2)$, y la segunda la imagen, $g(n_1,n_2)$, y queremos calcular la convolución sobre el píxel central de la matriz de imagen, en la posición [0,0], tendremos que:

$$f(n_1,n_2) = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$
$$g(n_1,n_2) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

y por tanto, la convolución sobre el píxel central $g(0,0) = 5$, será:

$$\begin{aligned}(f * g)[0,0] &= \sum_{m_1=-1}^{1} \sum_{m_2=-1}^{1} f[m_1,m_2] g[-m_1,-m_2] \\ &= (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9)\end{aligned}$$

Cabe destacar, que la referencia de origen [0,0] del filtro siempre debe ser el píxel central, $f[0,0] = e$, sin embargo, la referencia de la imagen puede estar situada en cualquier píxel de ésta, aunque en este caso se ha tomado la misma que en el filtro.

Si nos fijamos, la convolución consistiría en voltear las filas y columnas del filtro, realizar el producto matricial de Hadamard entre el filtro y la porción de la imagen, y por último sumar todos los términos de la matriz resultante.

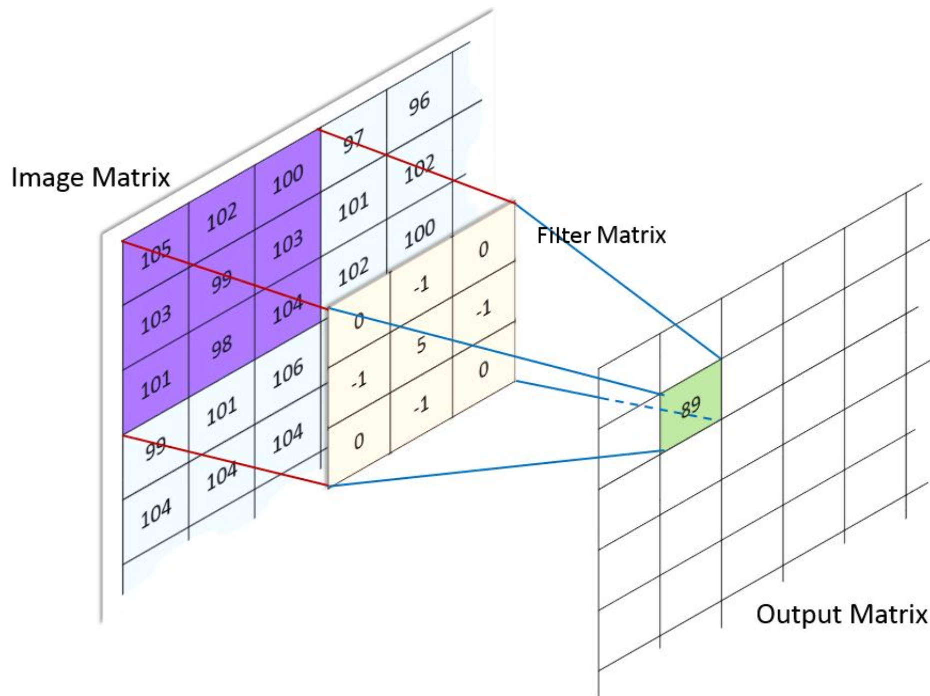


Figura 3.4 Se puede observar un filtro ya volteado vertical y horizontalmente, siendo aplicado a un conjunto de píxeles de la imagen de entrada.

Para realizar el proceso de convolución a una imagen al completo, lo que haríamos sería ir desplazando el filtro un número de píxeles determinado a lo largo de la imagen hasta cubrir el total del su area y aplicar la convolución sobre todos los píxeles necesarios. Lo usual, es mover el filtro píxel por píxel para realizar el proceso de convolución sobre todos los píxeles que forman la imagen, de forma que finalmente se obtiene

como resultado una imagen del mismo tamaño que la original. No obstante, esto no es posible si a la imagen no se le añaden píxeles extra alrededor, ya que en tal caso, si se aplica la convolución en un píxel del borde de la imagen, el filtro sobresaldría de la propia imagen, y por tanto, no habría píxeles a los cuales aplicar el filtro para realizar la convolución al completo. El proceso que se suele hacer es el denominado *zero padding*, que consiste en añadir alrededor de la imagen el número de píxeles necesarios con valor nulo para evitar que el filtro sobresalga.

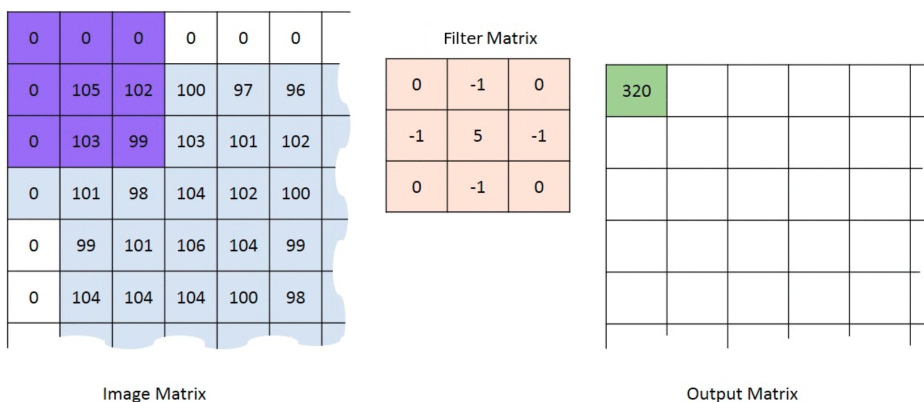


Figura 3.5 Proceso de *zero padding* de un único píxel para ajustarse al tamaño del filtro.

La imagen resultado obtenida tras realizar el proceso de convolución al total de la imagen original, se suele denominar *feature map*, es decir, mapa de características, ya que el resultado obtenido es un conjunto de píxeles con las caracteríticas de la imagen destacadas que se quieran hallar con el filtro utilizado, como por ejemplo, los contornos de los objetos.

Si tratamos de utilizar este mismo esquema mediante neuronas, observaremos que el proceso de convolución es muy sencillo de aplicar utilizando el mismo método que en las *FeedForward Networks*, ya que el filtro se puede considerar como un conjunto de *weights* que se aplican sólo a una parte de la imagen, cuyo resultado será el valor de una única neurona de la capa siguiente, y por tanto, el valor de uno de los píxeles del *feature map* resultado. Por ejemplo, en la Figura 3.6, se puede observar un filtro que es aplicado a una porción de la imagen (considerando la imagen como un array de datos) formada únicamente por tres píxeles, y que además, dicho filtro se desplaza un único píxel a lo largo de la imagen para poder aplicar la convolución a todos los píxeles que la forman, excepto los de los extremos. Como podemos observar, el filtro posee tres valores únicos, es decir, tiene tres *weights*, representados cada uno con un tono de gris, que son aplicados a cada uno de los píxeles de la porción de la imagen con la que estamos trabajando para hacer el proceso de convolución.

Por tanto, en dicho caso, los *weights* que conectan una capa con la siguiente, deberán formar los filtros para realizar la convolución a cada uno de los píxeles de la imagen, y por ello, los conjuntos de *weights* que forman estos filtros, deberán tener los mismos valores, ya que se debe aplicar el mismo filtro a toda la imagen.

Una vez realizado el proceso de convolución a la imagen al completo, para entrenar la red neuronal, sí se modificarán los valores de dichos *weights* para formar un filtro diferente que sea capaz de capturar mejor las características que se requieran de la imagen. Para realizar esto mediante el proceso de *backpropagation* como si se tratara de una *FeedForward Network*, se podría conseguir inicializando todos los conjuntos de *weights* que forman los filtros con los mismos valores, en nuestro caso, tres valores aleatorios ya están formados por tres *weights*. Luego, para cada *weight* que forma el filtro, se calcula el valor a actualizar para disminuir el error en cada uno de los píxeles del *feature map* resultado, y mediante el conjunto de valores obtenidos por todos los filtros, se realiza la media y se actualizará cada *weight*. Si nos fijamos en la Figura 3.6, al tener cuatro píxeles en el *feature map*, implica que hemos aplicado el mismo filtro un total de cuatro veces, y por tanto, realizaremos la media de los valores a actualizar del mismo *weight* de esos cuatro filtros. De esta forma, cada uno de los *weights* de cada filtro, se actualizarán de igual manera, manteniéndose el mismo filtro en todas las convoluciones realizadas en la imagen.

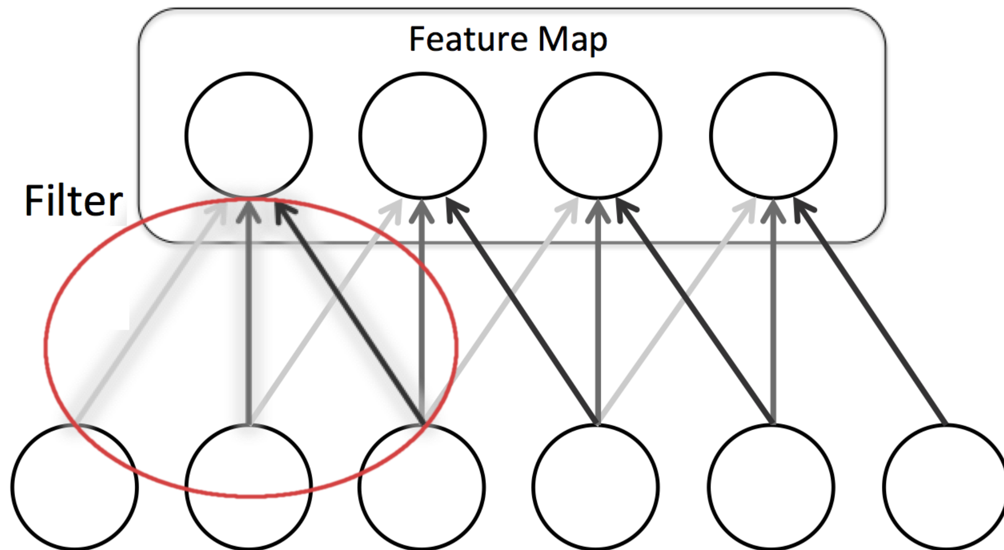


Figura 3.6 Proceso de convolución visto mediante una *FeedForward Network* con un filtro formado por tres *weights* (indicados en rojo).

Como hemos podido observar, se puede encontrar una equivalencia entre las *FeedForward Networks* y las *Convolutional Neural Networks*, sin embargo, lo explicado anteriormente no representa en su totalidad el modo de funcionamiento de una *Convolutional Neural Network*. Esto se debe, a que, como explicamos antes, este tipo de redes neuronales actúan sobre volúmenes y no áreas, y por tanto no operarán sobre un único *feature map*. Principalmente se debe a que será necesario el uso de muchas características para obtener la suficiente información como para clasificar con exactitud imágenes. Es muy frecuente el uso de imágenes RGB, que implica que poseemos tres matrices de entrada que cada una representan un canal de color de la imagen original, y por tanto, tendremos que trabajar sobre un volumen.

Como podemos ver en la Figura 3.7, en un principio tenemos tres matrices de entrada que pertenecen a una imagen RGB, a la que se le ha realizado un proceso de *zero padding* para poder aplicar el filtro a cada uno de los píxeles de la imagen. Esto implica que nuestro volumen de entrada, tal y como queda representado en la Figura 3.3, tiene el ancho y alto de las matrices que forman la imagen, y como profundidad el número de matrices, que en este caso es tres debido a que tenemos tres canales de color. Debido a que la profundidad del volumen de entrada es de tres, cada filtro estará formado por tres matrices, es decir, tendrán la misma profundidad que los datos de entrada, y de esta forma, se realizará la suma de los resultados obtenidos en las tres convoluciones realizadas para cada una de las matrices, resultando en un único valor de un píxel de un *feature map* de salida. Si nos fijamos, los filtros tienen un tamaño de 3x3, que junto con el tamaño 5x5, son los filtros que más se suelen usar, aunque en ocasiones se usan otros con un tamaño superior como de 7x7. Además, se puede observar que también se ha usado un *bias* que se añadirá al final tras calcular el resultado de los filtros. Finalmente, debido a que se han aplicado dos filtros, se han obtenido dos *feature maps* en la salida.

Por tanto, una *convolutional layer* estará formada por un conjunto de filtros, y convertirá un volumen de información en otro volumen. Para ello, los filtros deberán poseer la misma profundidad que los datos de entrada, y el número de filtros determinará la profundidad de los *feature maps* de salida.

Sabiendo estos conceptos, se puede calcular los tamaños de los volúmenes de salida de una *convolutional layer* teniendo en cuenta los parámetros de los datos de entrada y de los filtros aplicados. Para calcular el ancho y alto del volumen de salida, podremos usar las siguientes ecuaciones:

$$w_{out} = \frac{w_{in} - e + 2p}{s} + 1 \tag{3.5}$$

$$h_{out} = \frac{h_{in} - e + 2p}{s} + 1 \tag{3.6}$$

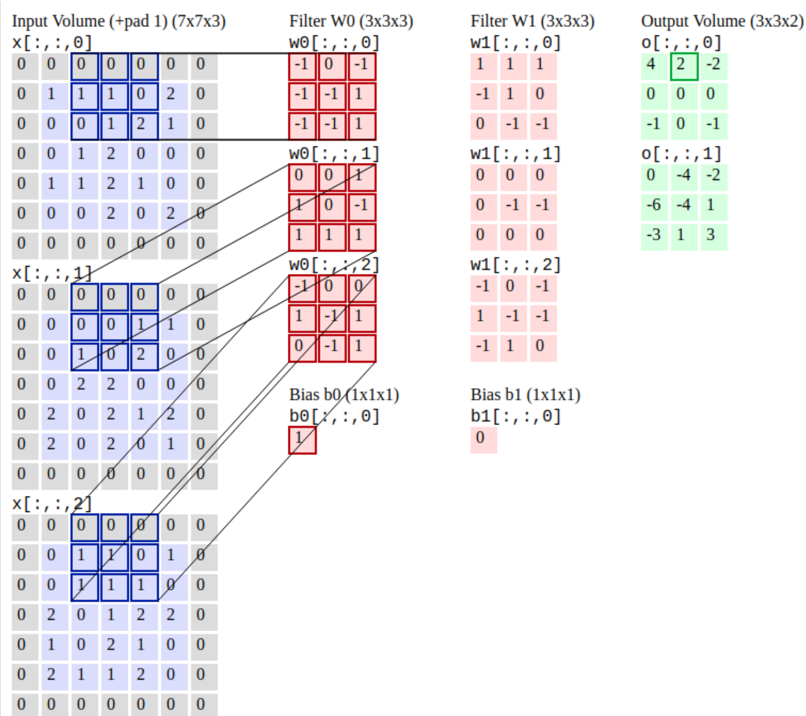


Figura 3.7 Se puede observar una imagen de entrada RGB (las tres matrices de entrada en color azul), dos filtros aplicados a cada canal de color de la imagen (en color rojo) y dos *feature maps* de salida (color verde).

donde w_{in} y w_{out} son el ancho del volumen de entrada y de salida de una capa (el ancho de las matrices que forman dichos volúmenes), h_{in} y h_{out} son el alto de los volúmenes (y por tanto de las matrices), e es el denominado *spatial extent* (es el ancho y alto de los filtros, ya que deben ser matrices cuadradas), s es el *stride* (número de píxeles que desplazamos el filtro a lo largo de la imagen o *feature map*), y p es el *zero padding* (número de píxeles con valor nulo añadidos alrededor de la imagen o *feature map*). Con esto ya sabremos el tamaño de los volúmenes de salida, ya que como ya sabemos, la profundidad del volumen es igual al número de filtros aplicados. Por ejemplo, en la Figura 3.7, nos encontramos con que $e = 3, s = 2$ y $p = 1$, por lo que finalmente los *feature maps* serán de tamaño 3x3.

Con esto ya tenemos los conocimientos para entender el proceso de convolución, que es el procedimiento más importante que permite a las *Convolutional Neural Networks* trabajar con imágenes de una forma eficaz.

Max Pooling

Las *Convolutional Neural Networks* no suelen estar formadas sólo por *convolutional layers*, sino que se les suele añadir *max pooling layers*. Estas capas permiten reducir y concentrar la cantidad de información que vamos extrayendo debido a que disminuye de tamaño los *feature maps* de salida provenientes de las convoluciones.

La idea principal de esta capa es dividir el *feature map* en secciones más grandes que los píxeles que lo forman, produciendo un *feature map* más reducido y condensado. En concreto, debido a que estamos realizando el proceso de *max pooling*, cada sección tendrá el valor del píxel con mayor peso que se encuentra en dicha sección.

Sabiendo el método de aplicación de este proceso, dependiendo de los parámetros podremos calcular al igual que antes el ancho y alto de los *feature maps* resultantes:

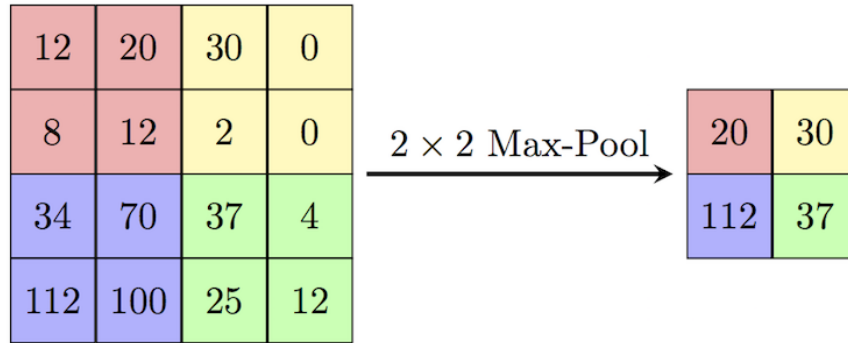


Figura 3.8 Ejemplo del proceso de *max pooling*.

$$w_{out} = \frac{w_{in} - e}{s} + 1 \quad (3.7)$$

$$h_{out} = \frac{h_{in} - e}{s} + 1 \quad (3.8)$$

sabiendo que e es el *spatial extent* y s es el *stride*. Por ejemplo, en la Figura 3.8, $e = 2$ y $s = 2$, resultando al final que el *feature map* sea de tamaño 2x2.

Una de las grandes ventajas de *max pooling* es que es localmente invariante, es decir, que aunque la entrada cambie algo, la salida del *max pooling* se mantendrá constante. Esta propiedad es muy importante en el tratamiento de imágenes, sobre todo cuando es más importante saber si realmente la imagen tiene ciertas características antes que saber dónde se encuentran éstas en la imagen. Sin embargo, hay que tener cuidado a la hora de utilizar *max pooling* porque puede destruir la habilidad de la red neuronal de capturar dichas características e información importante de la imagen. Para evitar estos problemas se suele dejar un valor de *spatial extent* pequeño.

Hay que destacar que en las últimas investigaciones se ha estudiado un método de *max pooling* distinto: *fractional max pooling*. Este método, utiliza números aleatorios no enteros para dividir la imagen en secciones de dicho tamaño, y principalmente permite evitar el *overfitting* en las *Convolutional Neural Networks*.

Con esto, ya tenemos todos los conceptos necesarios para realizar y diseñar una *Convolutional Neural Network* compuesta por *convolutional layers*, *max pooling layers* y *fully-connected layers*. Las *fully-connected layers* nos permitirán relacionar la información recogida y concentrada por los procesos de convolución y de *max pooling*, para finalmente poder clasificar imágenes con una *softmax layer*. Podemos observar algunos ejemplos de este tipo de redes neuronales en la Figura 3.9, que si nos fijamos, es común realizar varios procesos de convolución antes de añadir un *max pooling*, ya que si se colocan muchas *max pooling layers*, se puede llegar a destruir información importante tal y como se mencionó anteriormente. Sin embargo, si se colocan varias *convolutional layers* podemos llegar a recolectar una mayor información de las imágenes, aunque por supuesto, todo depende de la aplicación de la *Convolutional Neural Network*.

Convolutional Neural Networks en TensorFlow

La gran ventaja de TensorFlow es que el diseño y la programación de los grafos de redes neuronales se hace de forma muy sencilla sabiendo las funciones a utilizar, y con las *Convolutional Neural Networks* no es menos.

Al igual que hicimos con las *FeedForward Networks*, para programar los grafos computacionales del modelo de la red neuronal con mayor facilidad, se van a crear un conjunto de funciones que nos genere unos grafos específicos. En primer lugar, se va a crear la función que genere el grafo de la *convolutional layer*, que gracias a otras funciones de TensorFlow, se hace realmente sencillo:

```
def conv2d(input, weight_shape, bias_shape):
```

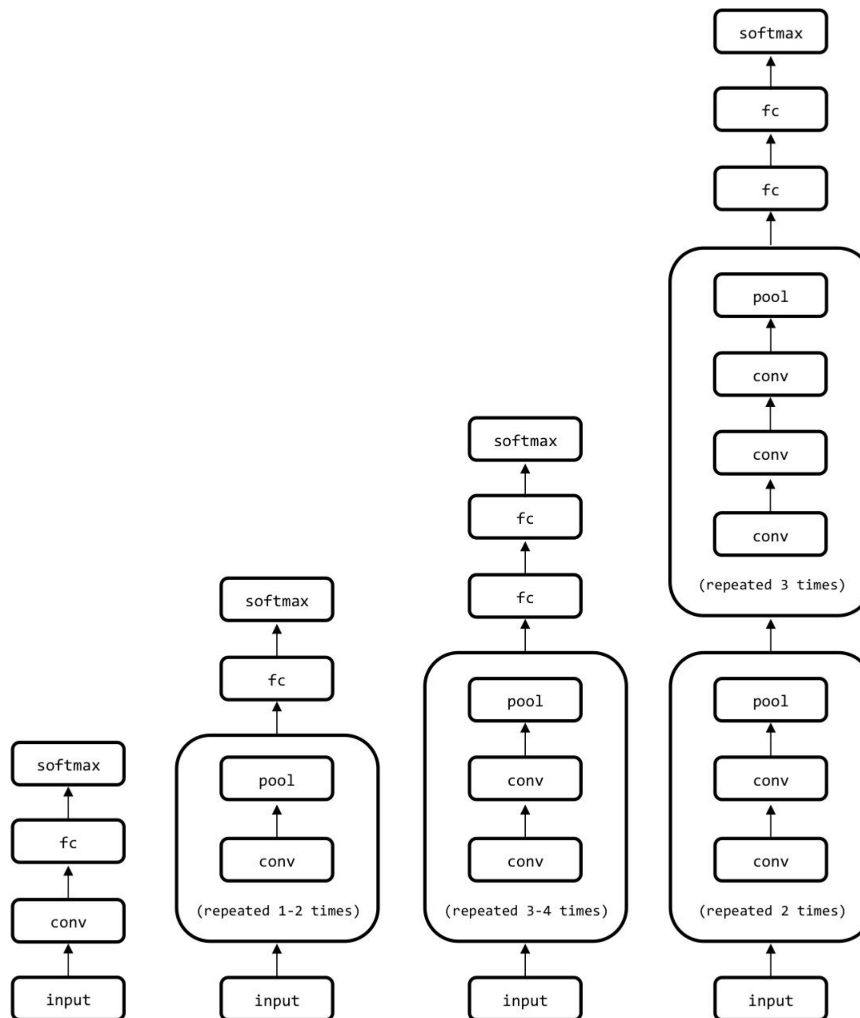


Figura 3.9 Ejemplos de *Convolutional Neural Networks*.

```
    #En esta funcion se define el grafo de una convolutional
    #layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0.1), trainable = True, name = 'b')

    conv_out = tf.nn.conv2d(input , weights , strides = [1,1,1,1], padding = '
        SAME')

    output = tf.nn.leaky_relu( tf.nn.bias_add( conv_out , biases ))
    return output
```

Podemos observar en el código, que al igual que se hizo en las redes neuronales de apartados anteriores, en primer lugar se crean e inicializan las variables con la función `tf.get_variable()` para evitar errores; además, si nos fijamos se ha utilizado un inicializador distinto al que se ha venido usando, `tf.truncated_normal_initializer()`. Este inicializador sigue creando números aleatorios según una distribución Gaussiana, sin embargo, cuando el número creado es superior a dos veces la desviación estándar indicada, el valor es rechazado y se vuelve a calcular. Esto evita que los valores de las variables sean muy elevados y haya descompensación respecto otras con valores más pequeños. También cabe destacar que los *biases* de cada filtro se han inicializado todos a un valor constante y pequeño con la función `tf.constant_initializer()`, para evitar desde un principio posibles muertes de neuronas. Después, el proceso de

convolución sobre una imagen (es decir, una convolución en dos dimensiones), se puede realizar con la función `tf.nn.conv2d()`, que tiene argumentos muy importantes. Los primeros argumentos, serán la entrada del proceso de convolución (imágenes o *feature maps*), y los filtros con sus valores ya inicializados y con los tamaños correctos, que, como ya se comentó, podemos encontrar una equivalencia entre los filtros y los *weights*, de ahí que el proceso de creación de dichas variables sea el mismo. Luego, nos encontramos con el argumento `strides`, que no es más que el desplazamiento del filtro por los datos de entrada, y será un array con cuatro valores en los que se le indica el desplazamiento por los ejemplos que forman el *minibatch*, por los píxeles de ancho y alto de las imágenes o *feature maps*, y por último, por los canales o número de *feature maps*. Es decir, en este caso, se moverá ejemplo a ejemplo, píxel a píxel a lo largo y ancho de la entrada de la capa, y recorriendo cada uno de los canales que la forman. Por último, nos encontramos el argumento `padding` que indica el *zero padding* a añadir a la entrada. En este caso se le ha impuesto el valor `SAME`, que quiere decir que se le va a añadir a la entrada el número de píxeles necesarios con valor cero como para que después del proceso de convolución, el ancho y alto de la entrada no se vea modificado cuando se usa un *stride* unitario. Esto se debe a que la fórmula que utiliza TensorFlow para calcular el tamaño de salida es:

$$output_shape = \frac{input_shape}{stride_shape} \tag{3.9}$$

donde *shape* se refiere al ancho y alto. Finalmente, al igual que todas las capas de *FeedForward Networks* se añade una función de activación, que si nos fijamos, en este caso se han añadido neuronas de tipo *Leaky ReLUs* para evitar que éstas mueran durante el proceso de entrenamiento.

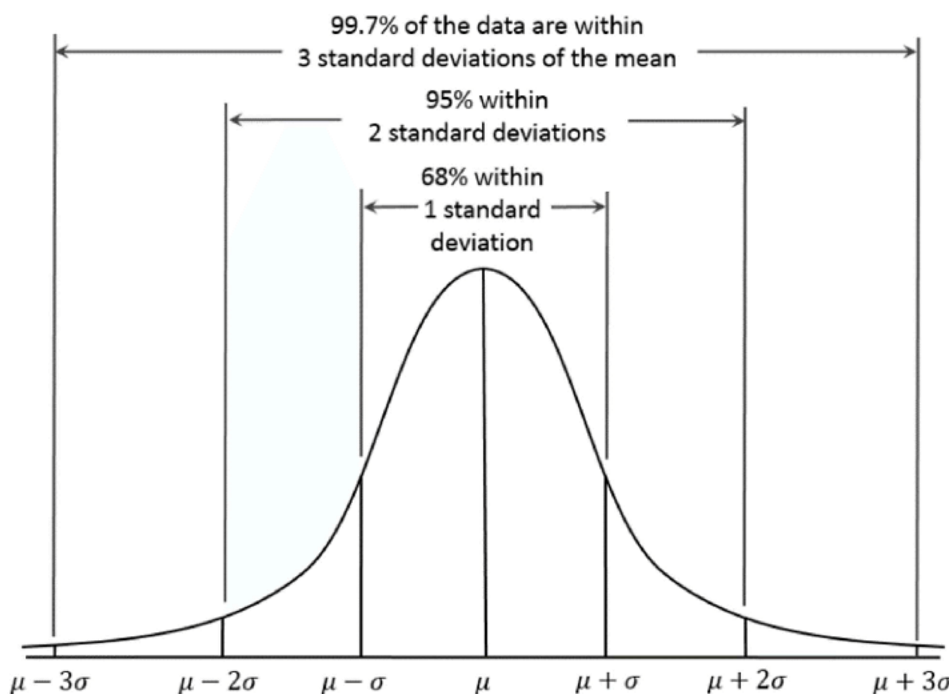


Figura 3.10 Distribución Gaussiana con una representación de las desviaciones estándar.

Como ya sabemos, otro grafo a diseñar sería el del *max pooling layer*, que otra vez gracias a las funciones de TensorFlow, su programación se puede realizar con gran facilidad si se conoce qué funciones utilizar:

```
def max_pool(input):
    #En esta funcion se define el grafo de una max pooling layer

    output = tf.nn.max_pool(input , ksize = [1,2,2,1] , strides = [1,2,2,1] ,
       padding = 'SAME')
    return output
```

En el código superior vemos que la creación de una *max pooling layer*, se hace simplemente con una función de *max pooling*, `tf.nn.max_pool()`. Para ello, se indica el tamaño del *kernel* que se va a utilizar en el proceso de *max pooling* en un único *feature map*. Con esto, el tamaño impuesto hará que el proceso se realice en un único ejemplo del *batch*, en una porción de 2x2 píxeles y en un único *feature map*; y por ello mismo, el *stride* deberá ser de forma que el *kernel* se desplace ejemplo a ejemplo, de dos en dos píxeles (para evitar un *overlapping* en el proceso) y de *feature map* en *feature map*. Puesto que se ha utilizado un `padding='SAME'` y que el *stride* no es unitario, sino que nos desplazamos de dos en dos píxeles, cada vez que se hace el proceso de *max pooling*, los *feature maps* quedan reducidos a la mitad de ancho y alto.

Por último, tendremos que crear la función que genere las *fully-connected layers*, que como ya sabemos, son las mismas que las utilizadas en las *FeedForward Networks*, por lo que se puede reutilizar el código:

```
def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una fully-connected
    #layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0.1), trainable = True, name = 'b')

    output = tf.nn.leaky_relu( tf.nn.bias_add( tf.matmul( input , weights ) ,
        biases ))
    return output
```

Como vemos, se ha utilizado la misma función que en las *FeedForward Networks*, con la diferencia de que se han seguido utilizando las *Leaky ReLUs* para evitar la muerte de las neuronas durante el entrenamiento.

Con esto ya tenemos todas las funciones necesarias para crear la *Convolutional Neural Network*. Como ejemplo, se va a utilizar la estructura de red neuronal creada específicamente para el entrenamiento de MNIST, que como ya sabemos, al ser el *"Hello, world!"* de las redes neuronales, no es necesario optar por una estructura muy compleja:

```
def neural_network_model(data , keep_prob):

    #Los datos de entrada (imagenes) las volvemos a transformar
    #en imagenes con [width,height,channels], ya que en un
    #inicio todas las imagenes estan dadas como arrays
    data = tf.reshape(data, shape=[-1,28,28,1])

    #En esta funcion estamos definiendo el grafo del modelo
    #de la red neuronal
    with tf.variable_scope('layer_1'):
        #La convolucion poseera 64 filtros de 5x5 que se
        #aplican a un canal
        conv_1 = conv2d(data , [5,5,1,64] , [64])
        pool_1 = max_pool(conv_1)

    with tf.variable_scope('layer_2'):
        #La convolucion poseera 64 filtros de 5x5 que se
        #aplican a los 64 feature maps formados por la
        #convolucion anterior
        conv_2 = conv2d(pool_1 , [5,5,64,64] , [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope('layer_fc'):
        #Los feature maps anteriores se aplanan para formar
        #un array de datos formados por los resultados de
```

```
        #los features maps y asi formar una red FeedForward
        #fully-connected con dichos datos
        pool_flattened = tf.reshape(pool_2 , [-1,7*7*64])

        fc_out = layer(pool_flattened , [7*7*64 , 1024] , [1024])

        #Para reducir Overfitting se aplica el proceso de
        #dropout. Para ello se usa un placeholder y asi
        #podremos activar dropout en entrenamiento y
        #desactivar en testeo
        fc_drop = tf.nn.dropout(fc_out , keep_prob)

    with tf.variable_scope('output'):
        output = layer(fc_drop , [1024 , 10] , [10])

    return output
```

Como podemos ver, en un principio se debe remodelar el formato del *minibatch* con el que se alimenta la red neuronal, de forma que el tensor sea de cuatro dimensiones. La primera dimensión corresponderá con los ejemplos que forman el *minibatch*, y las otras tres dimensiones corresponderán con la imagen: ancho, alto y número de canales, que al estar trabajando con MNIST, las imágenes son en blanco y negro y tendrán un único canal. Luego, creamos una capa formada por una *convolutional layer* y por una *max pooling layer*. Como vemos, cuando se crea la *convolutional layer*, debemos indicarle los datos de entrada, que en este caso son las imágenes que forman el *minibatch*, y los tamaños de los filtros (*weights*) y *biases*. Cuando indicamos el tamaño de los filtros, las dos primeras dimensiones corresponden con el ancho y alto de éste (al ser cuadrados, deben tener el mismo valor, en este caso de 5 píxeles), la tercera dimensión indica a cuántos canales o *feature maps* hay que aplicar el filtro (profundidad de los datos de entrada), y la cuarta dimensión, corresponderá al número de filtros a aplicar, es decir, equivale a cuántos *feature maps* se crearán a la salida de la convolución (profundidad de los datos de salida). Luego, a la salida de la convolución, simplemente se le aplica un proceso de *max pooling*, haciendo que los datos de entrada pasen de tener un tamaño de 28x28 a uno de 14x14. Este proceso se realiza una segunda vez, y después se aplica una *fully-connected layer*. Para ello, hay que volver a remodelar los datos de salida del último *max pooling* para que pasen a tener un formato de array, tal y como se trataron las imágenes con las *FeedForward Networks*. Sin embargo, la diferencia en este caso es que a dicha capa se le aplica un proceso de *dropout* con `keep_prob=0.5` durante el entrenamiento, tal y como se explicó en el apartado 5.6 del primer capítulo. Este proceso evitará el *overfitting* en la red neuronal, ya que con una probabilidad del 50 %, algunas neuronas se desactivarán, obligando a la red neuronal a tener que utilizar la mayoría de sus conexiones en la misma proporción. Por último, a la red neuronal se le ha añadido una capa de salida, que al igual que hicimos con las *FeedForward Networks*, no se le añade una *softmax layer* debido a que será computado directamente en el cálculo del error.

Con esto, tendríamos diseñada una *Convolutional Neural Network* y con la que podemos obtener unos resultados óptimos en el tratamiento de imágenes.

Conclusión

En los apartados anteriores se explicó el funcionamiento de las *Convolutional Neural Networks* y cómo se pueden programar mediante TensorFlow. Sabiendo todo eso, se pueden programar *Convolutional Neural Networks* facilmente y con diseños muy diversos según las especificaciones y tareas requeridas.

En los códigos presentados anteriormente, se usó de ejemplo una red neuronal específicamente diseñada para el dataset MNIST. Cuando se ejecutó dicha red neuronal, tras su entrenamiento, se llegó a conseguir un porcentaje de aciertos en las imágenes de testeo del 99.2 %, muy cerca del *state of art* actual que se encuentra en torno al 99.4 %. Esto indica que dicha red neuronal está bien diseñada y entrenada, y que además, ha podido superar con diferencia el resultado obtenidos por las *FeedForward Networks*. Ya con este *dataset*, se puede observar el potencial que poseen las *Convolutional Neural Networks* frente a las *FeedForward* respecto el tratamiento de imágenes. El código completo de dicha red neuronal se podrá encontrar en el Anexo 6.

El siguiente reto, es probar una *Convolutional Neural Network* para que sea entrenada respecto el dataset que se creó con imágenes de frutas y verduras. Con las redes neuronales *FeedForward* no se obtuvieron resultados muy buenos, por lo que es de esperar conseguir superar dichos resultados. Para ello, se creó una *Convolutional Neural Network* específicamente para dicho dataset y así obtener los mejores resultados posibles. Debido a ello, dicha red neuronal es más compleja que la creada para MNIST, ya que en este caso, no sólo hay que diferenciar formas, sino colores también. Una vez entrenada dicha red neuronal, se llegó a conseguir el fascinante porcentaje de 95.5 % de aciertos, muy por encima del conseguido con las *FeedForward*. Además, cuando se le presentaba a la red neuronal imágenes con fondos cambiados, distinta iluminación... seguía pudiendo identificar con éxito las frutas y verduras enseñadas. Esto quiere decir, que la red neuronal "aprendió" qué era cada fruta o verdura que se le presentaba, sin importar las condiciones en la que se encontraba, lo que implica que la relación señal-ruido aumentó significativamente respecto a la que nos encontramos con las *FeedForward Neural Networks*. El código de dicha red neuronal también se podrá encontrar al completo en el Anexo 7.

La conclusión de todas estas pruebas es que, con *Convolutional Neural Networks*, podremos conseguir unos resultados impresionantes ante el tratamiento de imágenes, comparado con las sencillas *FeedForward Neural Networks*. Por ello, en este proyecto se utilizará una *Convolutional Neural Network*, para que, finalmente, el robot creado sea capaz de identificar cada situación en la que se encuentre con total precisión.

Material y Método

Introducción

Una vez llegados a este punto, deberemos tener asimilados todos los conceptos básicos sobre redes neuronales, y de esta forma, tendremos la capacidad de diseñar y programar una red que se ajuste a nuestros objetivos.

Como ya se mencionó, la finalidad de este proyecto consiste en diseñar una red neuronal que se aplicará a un robot móvil. Dicha red, deberá ser capaz de clasificar imágenes de pasillos de cualquier edificio tomadas desde el punto de vista del propio robot en tres clases distintas: según si existe un obstáculo a la izquierda, si lo hay a la derecha, o no hay ningún obstáculo y no hay peligro de colisionar.

Principalmente, los obstáculos serán las paredes que limitan los propios pasillos, aunque la red neuronal también será entrenada para que sea capaz de esquivar otro tipo de obstáculos, como por ejemplo columnas, puertas o incluso muebles, ya que una de las intenciones fue crear un *dataset* lo más variado posible.

Dependiendo de la salida que obtengamos de la red neuronal, se le enviará al robot ciertas instrucciones de giro para evitar colisiones, y de esta forma, será capaz de recorrer todo un pasillo evitando todos los obstáculos que se le pongan por delante.

Así, habremos sido capaces de generar un algoritmo de percepción que por sí sólo, sin ser programado explícitamente, será capaz de identificar los obstáculos y dónde están situados.

Sin embargo, antes de ello, deberemos proceder a la contrucción del robot a utilizar.

El Robot

Para el proyecto, se ha querido usar el robot móvil más sencillo que se pueda construir, en este caso, se ha elegido un robot con cuatro ruedas de tipo diferencial. Este diseño es muy sencillo de controlar, es barato, y con cuatro motores, tendrá la suficiente potencia para transportar todo el equipo necesario para hacerlo funcionar. Este equipo estará formado por los siguientes componentes:

– Chasis del robot diferencial con los cuatro motores con reductoras y las cuatro ruedas
– Electrónica para el control de motores (*driver* L293D)
– Batería de 5V con capacidad de 15000mAh
– Raspberry Pi Model 3 con sistema operativo Raspbian Stretch
– Tres cámaras con conexión USB

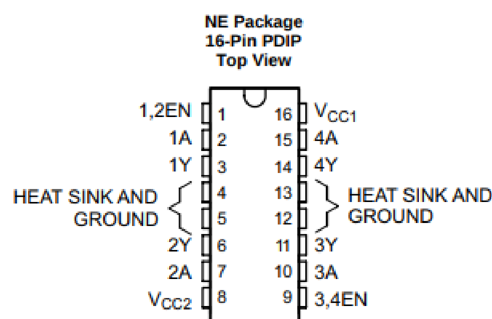
Una vez montado el chasis, es necesario diseñar la electrónica que permitirá el control de los cuatro motores mediante señales PWM, y de esta forma, controlaremos la velocidad a la que gira cada una de las ruedas para que el robot se mueva en la dirección que queramos.

Esta electrónica es muy sencilla de diseñar, ya que se ha usado un único *driver* para los cuatro motores, siendo éste el famoso puente en H modelo L293D. Este *driver*, tiene las características necesarias para soportar los cuatro motores funcionando a los 5V de salida que ofrece la batería, tal y como se puede observar en el *datasheet* del L293D de Texas Instruments de la Figura 4.2.

Las señales PWM se generarán mediante los pines GPIO de la Raspberry Pi. Por tanto, mediante el *driver*, conectaremos la Raspberry Pi con los motores, y de esta forma, con un pequeño *script* de Python, podremos controlar las señales PWM y la velocidad de giro de los motores. Para ello, como veremos más tarde, se programó dos pines GPIO con dos PWM, de forma que controlaríamos de igual forma ambas ruedas de cada lado del robot.

Además, se montó un pequeño circuito electrónico que conectara todos los componentes mencionados, y de esta forma, tendríamos el control básico de nuestro robot diferencial. El circuito montado se puede ver en la Figura 4.4, que como vemos, se han usado varios pines de la Raspberry Pi. En primer lugar, se usó

Figura 4.1 Chasis del robot diferencial usado en el proyecto.



Pin Functions

PIN		TYPE	DESCRIPTION
NAME	NO.		
1,2EN	1	I	Enable driver channels 1 and 2 (active high input)
<1:4>A	2, 7, 10, 15	I	Driver inputs, noninverting
<1:4>Y	3, 6, 11, 14	O	Driver outputs
3,4EN	9	I	Enable driver channels 3 and 4 (active high input)
GROUND	4, 5, 12, 13	—	Device ground and heat sink pin. Connect to printed-circuit-board ground plane with multiple solid vias
V_{CC1}	16	—	5-V supply for internal logic translation
V_{CC2}	8	—	Power VCC for drivers 4.5 V to 36 V

Figura 4.2 Características del *driver* L293D.

el Pin 2 con la salida a 5V para alimentar la lógica del L293D, el Pin número 6 para la referencia a tierra de dicho *driver*, y por último, los pines 11 y 13 para las dos salidas de las dos señales PWM usadas para el control de los dos pares de motores. Como ya se mencionó, la batería usada es de 5V y será la que transmita la potencia al L293D para los motores, ya que su salida máxima es de 3.1A, suficiente para transimitir la potencia requerida para mover el robot sin problemas gracias a las reductoras que poseen los motores.

Finalmente, para terminar el robot, hizo falta colocar las tres cámaras que necesitaríamos para crear los *datasets*. Estas cámaras fueron conectadas directamente a los puertos USB de la Raspberry Pi, por lo que, se tuvo que elegir una batería con gran capacidad, ya que ésta no sólo alimentaría a los motores, sino que también a la Raspberry Pi con las tres cámaras. La capacidad de dicha batería es de 15000mAh, una

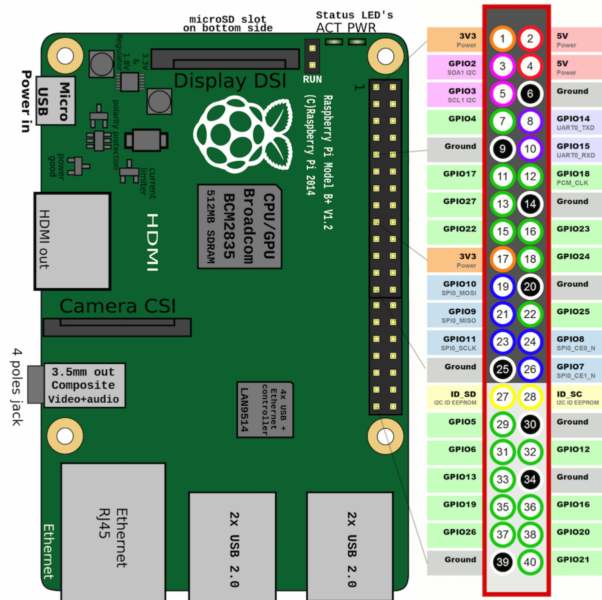


Figura 4.3 Pines de salida GPIO de una Raspberry Pi 3 Model B.

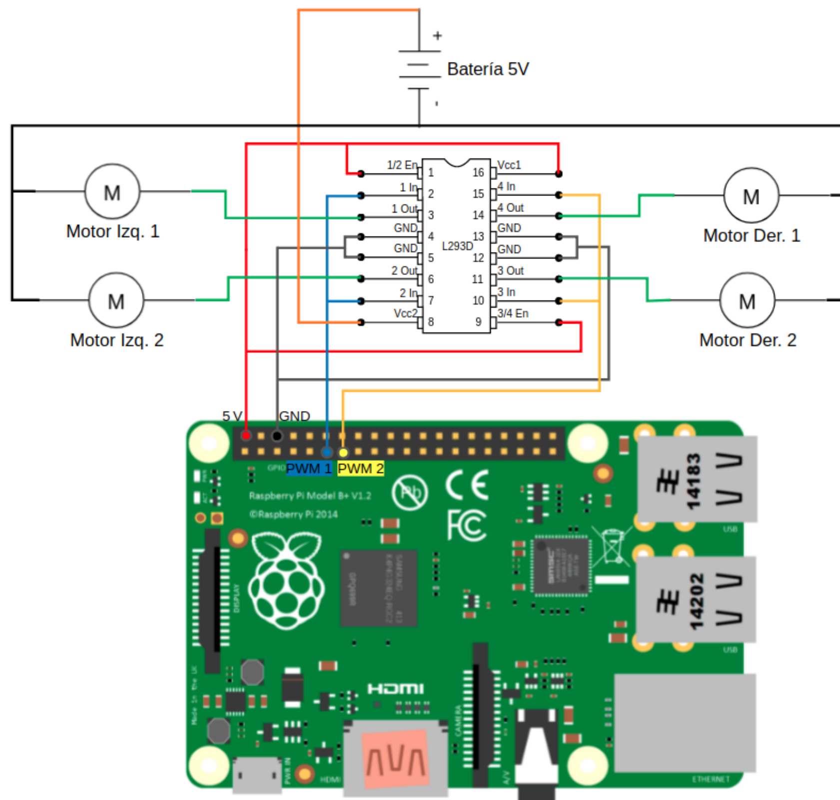


Figura 4.4 Esquemático de la electrónica usada en el robot.

capacidad idónea para tener una autonomía de varias horas con el robot en continuo funcionamiento. El resultado final se puede observar en la Figura 4.5.

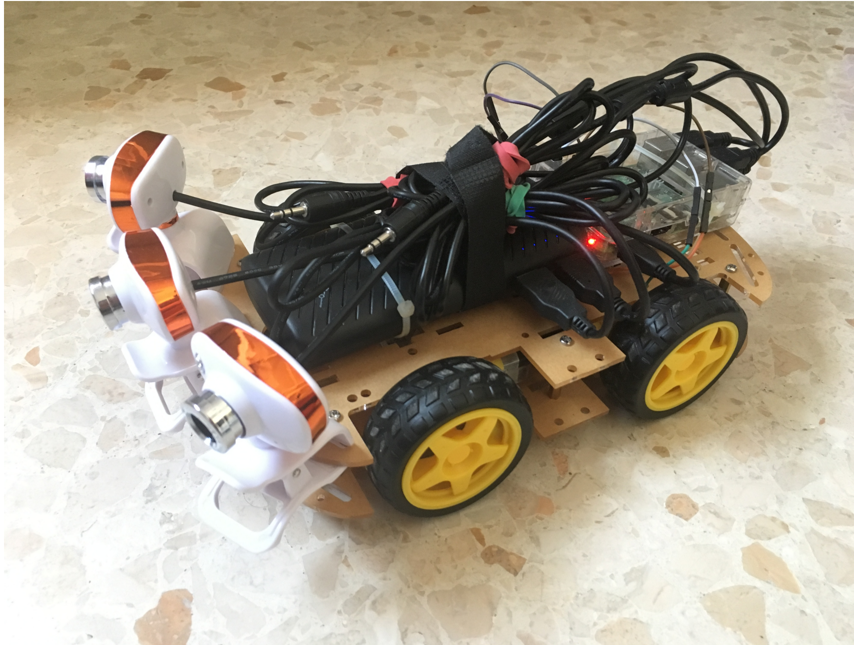


Figura 4.5 Diseño del robot final una vez montados todos los componentes.

Captura de imágenes

Con el robot montado, el siguiente paso sería tomar una gran cantidad de imágenes con las cámaras para poder construir los datsets de entrenamiento y testeo.

Este proceso fue relativamente sencillo comparado con la dificultad que se suele tener para generar los *datasets* para el entrenamiento de una red neuronal. Esta facilidad se debió principalmente al uso de las tres cámaras que están montadas encima del robot, tal y como se puede ver en la Figura 4.5.

Si nos fijamos, la cámara frontal, simplemente graba imágenes de lo que se encuentra frente al robot, mientras que las cámaras laterales se encuentran giradas respecto a la cámara frontal 30º, lo que permite una visión de los alrededores del robot.

El proceso de toma de imágenes consistiría en ir moviendo el robot a lo largo de un pasillo mientras las tres cámaras toman imágenes, de esta forma, conseguiríamos tres clases de imágenes: una del suelo del pasillo sin obstáculos (cámara frontal), otra de obstáculos al lado izquierdo (cámara de la izquierda) y otra de obstáculos del lado derecho (cámara derecha). Con ello, tendríamos ya las imágenes clasificadas, que es uno de los grandes problemas a la hora de crear *datasets*.

Para ello se programaron dos *scripts* en Python que permitirían el control del robot desde un PC de forma inalámbrica, los cuales se pueden ver en el Anexo 8. Para esto, se creó mediante una tarjeta WiFi externa, un *Hotspot* (punto de acceso WiFi) al que se podía conectar la Raspberry Pi. Con la creación de dicha red LAN (red local), la comunicación entre el PC y la Raspberry se hace muy sencillo con la creación de *sockets* en los *scripts* de Python usando una comunicación TCP.

De esta forma, se crea en el PC el lado del servidor y en la Raspberry Pi el lado del cliente. Cuando la Raspberry se conecta al servidor, el PC detectará las pulsaciones del teclado para, dependiendo de la tecla pulsada, enviar una orden determinada por medio de los *sockets* a la Raspberry. La Raspberry actuará dependiendo de la orden, de forma que el robot se hace completamente controlable, pudiéndolo mover en todas las direcciones y a la velocidad indicada, además de poder controlar la toma de imágenes de las cámaras.

Uno de los grandes problemas de esta parte del proyecto, era hacer funcionar tres cámaras al mismo tiempo en la Raspberry Pi. Esto no es posible debido al limitado ancho de banda del bus USB, por lo que

una de las soluciones por las que se optó, fue hacer un pequeño protocolo que desactivara y activara las cámaras de forma que sólo estuvieran funcionando dos al mismo tiempo, impidiendo el desbordamiento de datos del bus. El único inconveniente de esta solución, es que la velocidad de toma de imágenes se ve drásticamente reducida, tomando como mucho 2 imágenes por segundo. Debido a que se está creando un dataset de entrenamiento, esto es inviable, porque es necesario tener miles de imágenes por cada clase para un correcto entrenamiento de la red neuronal y evitar un *overfitting* sobre los ejemplos de entrenamiento.

La solución final fue tomar las imágenes por partes, es decir, primero se tomaron las imágenes con las dos cámaras laterales, ya que son las imágenes más difíciles de tomar y con las que más cuidado hay que tener, y luego, se tomaron las imágenes con la cámara frontal, que son imágenes muy sencillas de capturar. Esto se hizo fácilmente mediante los *scripts* creados, ya que se tiene la oportunidad de indicar en los argumentos de dichas funciones qué cámaras deben funcionar, por lo que, finalmente, se puede obtener una gran cantidad de imágenes.

Figura 4.6 Ejemplos de imágenes tomadas por las tres cámaras en un mismo instante de tiempo.

Otro de los grandes retos a la hora de tomar buenas imágenes de entrenamiento y testeo, fue controlar la inclinación de las cámaras respecto el suelo. Esto se debe, a que las cámaras tienen un ángulo de visión muy pequeño, por lo que, dependiendo de la situación, se debe cambiar la inclinación de dichas cámaras para evitar tomar imágenes que puedan causar problemas durante el entrenamiento de la red neuronal. Por ejemplo, un error típico fue que las cámaras laterales estuvieran muy inclinadas hacia el suelo, por lo que no se tomaban buenas imágenes de los obstáculos que había alrededor. Otro error, era levantar demasiado la cámara frontal, por lo que al final la cámara terminaba por captar obstáculos que estuvieran en frente del robot.

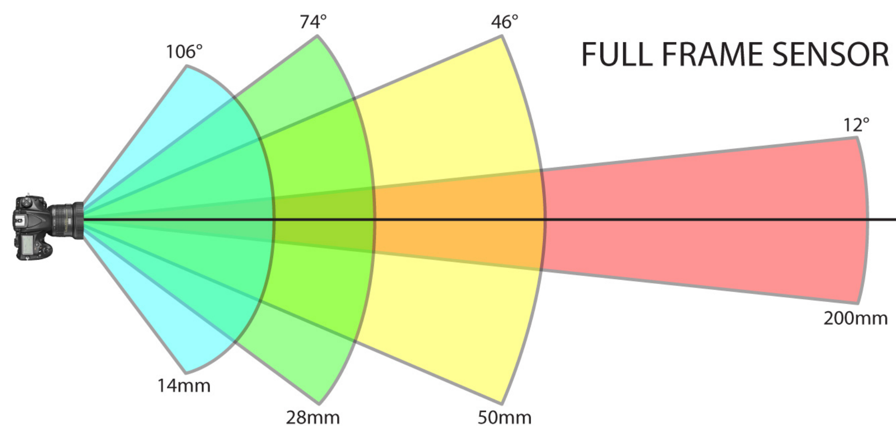


Figura 4.7 Ejemplo de diferentes ángulos de visión en una cámara.

Se tomaron casi 17000 imágenes en total de varios pasillos o espacios limitados con obstáculos, de forma, que las imágenes fueran lo más variadas y distintas posibles, ya que cuanto mayor sea la diversidad de las imágenes, menos posibilidades hay de causar *overfitting*, lo que permitirá a la red neuronal aprender a generalizar entre las distintas situaciones.

Datasets de Entrenamiento y Testeo

Todas esas imágenes tomadas, fueron divididas en dos *datasets* distintos, uno el de testeo, compuesto por 225 imágenes, y el de entrenamiento, compuesto por unas 16700 imágenes. Hay que destacar que, como ya se ha mencionado en varias ocasiones, el dataset de testeo posee imágenes que no se encuentran en el dataset de entrenemiento, por lo que serán situaciones que la red neuronal nunca ha visto, y por tanto, los resultados que se obtienen por medio de dicho *dataset* son unas medidas fiables del buen funcionamiento de la red neuronal.

Una de las condiciones esenciales a la hora de crear los *datasets*, es que todas las imágenes que los componen, deben ser perfectamente diferenciables por un ser humano, en caso contrario, tales imágenes no deben estar incluidas en el *dataset*, ya que la red neuronal tampoco será capaz de diferenciarlas.

Otra condición bastante importante para las imágenes de entrenamiento, es que debe haber aproximadamente el mismo número de imágenes para todas las clases en las que se pueden clasificar, porque si no es así, se puede producir *overfitting* sobre una de las clases, evitando tener una buena respuesta de la red neuronal ante determinadas situaciones. Por último, otra condición, es que las imágenes que componen el *dataset* de entrenamiento deben estar mezcladas de forma aleatoria, y de esta manera, evitar que surjan relaciones lineales entre los distintos ejemplos de entrenamiento que pueden provocar que la red neuronal tenga malos resultados ante el dataset de testeo.

Una vez que se capturaron las imágenes, fueron tratadas y reconvertidas tal y como se hizo en el pequeño dataset de prueba de frutas y verduras. Por tanto, en este proceso, lo que se hizo fue disminuir el tamaño de las imágenes a una dimensión de 112x112 píxeles; de esta forma la computación del entrenamiento de la red neuronal será más eficiente y rápida. Además, para simplificar aún mas la red neuronal, se procedió a pasar las imágenes de formato RGB a blanco y negro, ya que no es necesario que la red neuronal detecte colores, simplemente debe aprender a detectar bordes y esquinas para saber si existe un obstáculo, como por ejemplo una pared que limite el pasillo. Por último, se transformó la matriz que forma la imagen a un único vector de valores para simplificar la escritura y posterior lectura de la matriz de ejemplos de entrenamiento, ya que cada fila de la matriz corresponderá a una única imagen. De esta forma, cada imagen tendría un formato de 1x112·112, por lo que el *dataset* de entrenamiento sería una matriz de 16700x12544. El *script*de Python utilizado para el procesamiento de las imágenes es el mismo que se usó con el *dataset* de frutas y verduras, que lo podemos ver en el Anexo 5.

Debido a la gran cantidad de imágenes, el dataset de entrenamiento se tuvo que dividir en tres partes para que se procesaran cada uno por separado, ya que el proceso se realentizaba cuantas más imágenes se procesaban. Finalmente, tras unas 20 horas de procesamiento, se obtuvieron todas las imágenes en una única matriz, además de obtener otra matriz con las etiquetas en formato *one-hot encoding* de cada una de las imágenes.

Con esto, ya teníamos todas las imágenes procesadas y listas para introducirlas a una red neuronal y entrenarla.

La Red Neuronal

Una vez tenemos nuestro conjunto de imágenes de entrenamiento y testeo, es necesario diseñar la red neuronal que se va a utilizar. Como ya se comentó, la red será de tipo *Convolutional Neural Network*, debido principalmente al gran potencial que poseen en el procesamiento de imágenes.

La red neuronal debía ser capaz de detectar todo aquello que pudiera hacer colisionar al robot: desde las paredes que limitan los pasillos hasta los obstáculos que hubiera en éstos. Sin embargo, las características que debía ser capaz de extraer nuestra red neuronal no son muy complejas, ya que simplemente debía ser capaz de detectar si había obstáculos, y en tal caso, dónde se encontraban situados, si a la izquierda o a la derecha.

Por esto mismo, el modelo de la red neuronal diseñada no es muy complejo, poseyendo cuatro *convolutional layers* seguidas cada una de ellas por una *max pooling layer*, y finalmente, tiene una *fully-connected layer* con *dropout* para evitar *overfitting* seguida de una capa de salida de tipo *softmax* para el entrenamiento. De esta forma, el modelo de la red queda de la siguiente forma programada con TensorFlow:

```
def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una fully-connected
    #layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.contrib.
        layers.xavier_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0), trainable = True, name = 'b')

    output = tf.nn.leaky_relu( tf.nn.bias_add( tf.matmul( input , weights ) ,
        biases ))
    return output

def conv2d(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una convolutional
    #layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.contrib.
        layers.xavier_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0), trainable = True, name = 'b')

    conv_out = tf.nn.conv2d(input , weights , strides = [1,1,1,1], padding = '
        SAME')

    output = tf.nn.leaky_relu( tf.nn.bias_add( conv_out , biases ))
    return output

def max_pool(input):
    #En esta funcion se define el grafo de una max pooling layer

    output = tf.nn.max_pool(input , ksize = [1,2,2,1] , strides = [1,2,2,1] ,
        padding = 'SAME')
    return output

def neural_network_model(data , keep_prob):

    #Los datos de entrada (imagenes) las volvemos a transformar
    #en imagenes con [width,height,channels], ya que en un
    #inicio todas las imagenes estan dadas como arrays
    data = tf.reshape(data, shape=[-1,112,112,1])

    #En esta funcion estamos definiendo el grafo de la red
    #neuronal

    with tf.variable_scope('layer_1'):
        #La convolucion poseera 64 filtros de 5x5 que se
        #aplican a un unico canal de color
        conv_1 = conv2d(data , [5,5,1,64] , [64])
        pool_1 = max_pool(conv_1)

    with tf.variable_scope('layer_2'):
        #La convolucion poseera 64 filtros de 5x5 que se
```

```
        #aplican a los 64 feature maps formados por la
        #convolucion anterior
        conv_2 = conv2d(pool_1 , [5,5,64,64] , [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope('layer_3'):
        conv_3 = conv2d(pool_2 , [5,5,64,64] , [64])
        pool_3 = max_pool(conv_3)

    with tf.variable_scope('layer_4'):
        conv_4 = conv2d(pool_3 , [3,3,64,64] , [64])
        pool_4 = max_pool(conv_4)

    with tf.variable_scope('layer_fc'):
        #Los feature maps anteriores se aplanan para formar
        #1 array de datos formados por los resultados de
        #los features maps y asi formar una red FeedForward
        #fully connected con dichos datos
        pool_flattened = tf.reshape(pool_4 , [-1,7*7*64])

        fc_out = layer(pool_flattened , [7*7*64 , 512] , [512])

        #Para reducir Overfitting se aplica el proceso de
        #dropout. Para ello se usa un placeholder y asi
        #podremos activar dropout en entrenamiento y
        #desactivar en testeo
        fc_drop = tf.nn.dropout(fc_out , keep_prob)

    with tf.variable_scope('output'):
        output = layer(fc_drop , [512 , 3] , [3])

    return output
```

El código, como se puede observar, es muy parecido a los ya presentados en el tema de las *Convolutional Neural Networks*. Sin embargo, hay que destacar un detalle no presentado anteriormente. Se trata de la forma de inicialización de las variables que componen las *convolutional layers* y las *fully-connected layers*. Si nos fijamos, para los *weights* se ha usado el inicializador `tf.contrib.layers.xavier_initializer()`, es decir, se está utilizando la inicialización de Xavier. Este tipo de inicialización fue creado por Xavier Glorot y Yoshua Bengio en 2010 tras explicar la enorme dificultad y lo importante que es la inicialización de las variables de una red neuronal.

La motivación de la inicialización de Xavier en las redes neuronales, permite que las variables se inicialicen con unos valores adecuados, es decir, que las variables no comiencen ni con valores muy grandes ni con valores muy pequeños, permitiendo finalmente una mejor y más rápida convergencia de las redes neuronales y obteniendo entrenamientos mucho más satisfactorios. Cuando realizaron el estudio, la inicialización de los *bias* la impusieron con valor constante nulo, debido a ello, en nuestro caso también se inicializaron con valor cero mediante la función `tf.constant_initializer()`.

Sin embargo, en 2015 apareció otro estudio por parte de Kaiming He proponiendo otro tipo de inicialización, denominada inicialización de He. En este estudio se propuso otra inicialización basada en los conceptos que propuso Xavier en 2010. Sin embargo, en este caso, debido a que los estudios de Xavier se hicieron mediante neuronas lineales, el estudio de He se realizó mediante neuronas de tipo ReLU, declarando que la inicialización de Xavier no es viable para este tipo de neuronas ya que puede evitar la convergencia en *Deep Neural Networks* cuando hay muchas capas.

A pesar de ello, tras varias pruebas resultó que, en mi caso, la inicialización de Xavier resultó mucho más satisfactoria que la de He. Con esto se puede concluir que en el ámbito de *Deep Learning* nada está escrito y

todo es suceptible a cambios, ya que dependiendo de la situación, cada red neuronal trabajará mejor en unas condiciones u otras. Esto se debe a que las redes neuronales son un problema multidimensional en el que se deben tener en cuenta una grandísima cantidad de variables.

Así, el modelo de la red neuronal diseñada deberá ser lo suficientemente complejo para poder obtener todas las características necesarias de las imágenes y poder clasificarlas con precisión, pero no tanto como para que haya demasiadas variables con respecto al número de ejemplos de entrenamiento y que se produzca *overfitting*.

El *overfitting* y la generalización de la red neuronal será uno de los objetivos a la hora de entrenar la red. El hecho de aplicar una capa con *dropout* y la gran cantidad de imágenes de entrenamiento tomadas ayuda bastante a dicha tarea, aunque hay que seguir teniendo cuidado ya que existen muchas variables que pueden evitar un correcto entrenamiento.

Sin embargo, tras varias pruebas de entrenamiento y varios modelos de redes neuronales creados, el diseño de la red expuesto en el código superior llegó a dar unos muy buenos resultados, siendo capaz de generalizar las imágenes con bastante eficacia.

Funcionamiento de la Red Neuronal a tiempo real

El siguiente paso tras entrenar la red neuronal, sería aplicar dicha red con imágenes tomadas a tiempo real del robot.

Esto puede llegar a ser un reto, debido principalmente a la baja velocidad de procesamiento de la Raspberry Pi comparado con un PC del altas prestaciones. Por ello mismo, se descartó la posibilidad de usar la librería TensorFlow en la propia Raspberry Pi.

Lo que se hizo en este caso, fue algo parecido a lo realizado para la toma de imágenes para los *datasets*. En primer lugar, se creó un servidor en el PC para que se pudiera conectar la Raspberry Pi y pudieran comunicarse entre ellos.

La única función de la Raspberry Pi, en este caso, sería de tomar imágenes de la cámara, ya que sólo es necesario el uso de una, por medio de la librería OpenCV, que fue instalada en el sistema operativo Raspbian. Esto no fue sencillo debido a que, principalmente, las tareas de I/O son muy lentas, y lo es mucho más cuando se trata de captura de imágenes por medio de una cámara conectada por USB. Además, esta lentitud se intensifica con el procesamiento de imágenes, ya que son tareas muy pesadas para la CPU, lo que provoca finalmente un importante cuello de botella en nuestro programa.

Por tanto, la función que usa OpenCV en la lectura de imágenes bloquea al código, haciendo que se relentice todo el programa en general. Esto provocaba que las imágenes que se recibían habían sido tomadas por el robot aproximadamente 10s antes, lo que causaba que el control del robot mediante la red neuronal se hiciera imposible.

La mejor solución fue mover la tarea de captura de imágenes a un hilo separado del hilo principal del programa, de forma que mientras que el código principal continuaba ejecutándose, el otro hilo va tomando las imágenes de la cámara. Esta solución provocaba un retraso máximo, entre la posición del robot y la recepción de la imagen, de aproximadamente de 1s, diez veces menos que sin la utilización de otro hilo.

Para evitar sobrecargar la Raspberry Pi y relentizar aún más la toma de imágenes, se evitó en lo máximo posible que ésta procesara dichas imágenes, de forma que cuando se capturaba una, simplemente se guardaba como un archivo de imagen .jpeg, sobrecargando lo mínimo la CPU.

Una vez la Raspberry había tomado y guardado la imagen, el PC utilizaría el protocolo SCP para copiar dicha imagen en su memoria, de forma que a partir de ahí pudiera leerla y procesarla, para finalmente introducirla en la red neuronal. Cuando se entrenó la red neuronal, se guardó dicho modelo con los valores de todas las variables ya entrenadas, de forma que el PC sólo tenía que cargar dicho modelo y ya podría

utilizar la red introduciéndole las imágenes ya procesadas. Una vez realizado todo el procesamiento de la red y obtenida su predicción, el PC enviaría a la Raspberry por medio del *socket* la orden de capturar otra imagen, y tras ello, se le enviaría la orden de qué movimiento hacer con el robot dependiendo de la predicción obtenida por la red neuronal. Este procedimiento se realiza en bucle hasta que se le indique al PC su finalización, enviando éste una señal a la Raspberry Pi para que acabe con el hilo de captura de imágenes y el proceso del programa en general.

Todo estos procedimientos se pueden observar en los códigos aportados en el Anexo 9.

Resultados y Conclusión

Una vez realizados todos los procedimientos anteriormente explicados, se hicieron múltiples pruebas para obtener unos resultados óptimos.

En primer lugar, la red neuronal fue entrenada con las 16700 imágenes de entrenamiento, hasta que finalmente, se llegó a obtener un porcentaje de aciertos del 99.55 % en el *dataset* de testeo. Esto implica que la red neuronal pudo conseguir con gran precisión las características que se requerían de una imagen para detectar si había o no obstáculos, y si lo había, si estaba a la izquierda o a la derecha. Esto quiere decir, que debido a que la mayoría de situaciones son muy parecidas en este ámbito, la red neuronal pudo generalizar con bastantes buenos resultados en situaciones que nunca había visto.

De hecho, finalmente la red neuronal fue testada ejecutándose a tiempo real con imágenes obtenidas del robot. Esta situación es mucho más difícil de afrontar satisfactoriamente, ya que se tienen las limitaciones de captura y transmisión de imágenes, que como ya se mencionó, podría haber un retraso de aproximadamente 1s desde que se toma la imagen hasta que se recibe y procesa. Por tanto, esto implica que si la red neuronal tiene un fallo en una de sus predicciones, es muy probable que el robot llegue a colisionar.

A pesar de ello, tras hacer múltiples pruebas, la red neuronal no solía fallar apenas, lo que implicaba que el robot llegaba a esquivar los obstáculos que encontraba con relativa facilidad.

Por tanto, para llevar a cabo dichas tareas de forma satisfactoria, hay que tener en cuenta muchísimos factores para que el funcionamiento de la red neuronal sea el óptimo; algunos de ellos son:

– En primer lugar, en el caso del procesamiento de imágenes, lo mejor es utilizar una *Convolutional Deep Neural Network*, ya que el procesamiento de imágenes lo hace con gran efectividad comparado con el resto de redes neuronales, permitiéndonos obtener unos resultados inmejorables. Sin embargo, hay que tener cuidado en el diseño de la red neuronal, cuanto más compleja no es mejor, de hecho, suele ser una premisa bastante errónea. En el caso de que se realice más compleja, el número de variables de los que depende la red aumenta considerablemente, empeorando el rendimiento de procesamiento durante su entrenamiento, y probablemente empeorando los resultados sobre el *dataset* de testeo debido al *overfitting*. Aunque, por supuesto, también hay que evitar una red neuronal muy simple, ya que no se conseguiría captar todas las características necesarias para poder clasificar las imágenes; es decir, hay que encontrar un equilibrio. Otro factor en el diseño de la red a tener muy en cuenta, es la inicialización de las variables, ya que no todas las inicializaciones permiten una convergencia a la red. Es muy recomendable el uso y estudio de las inicializaciones de Xavier y de He, ya que permiten obtener buenos resultados.

– Otros factores a tener muy en cuenta durante el entrenamiento, son el tamaño del *minibatch* y el número de *epochs*. Cambiando el tamaño del *minibatch* podremos cambiar en gran medida los resultados del entrenamiento de la red, permitiendo una convergencia más rápida, evitando el estancamiento en puntos de silla o incluso consiguiendo resultados mejores en los ejemplos de testeo. Con el número de *epochs* también hay que tener cuidado, y es mejor no fijarlo a un valor determinado, ya que dependiendo de la situación se necesitarán más o menos *epochs* para obtener los resultados requeridos. Principalmente hay que tener cuidado y evitar entrenar la red neuronal durante muchos *epochs*, ya que aunque el coste sobre los ejemplos de entrenamiento siga disminuyendo, eso podría indicar un

overfitting sobre los ejemplos, queriendo decir que la red neuronal está ”aprendiendo de memoria” los ejemplos en vez de ”aprender” las características que distinguen cada clase del *dataset*. Por ello es recomendable calcular el error de testeo tras cada *epoch* para dejar de entrenar cuando sea conveniente.

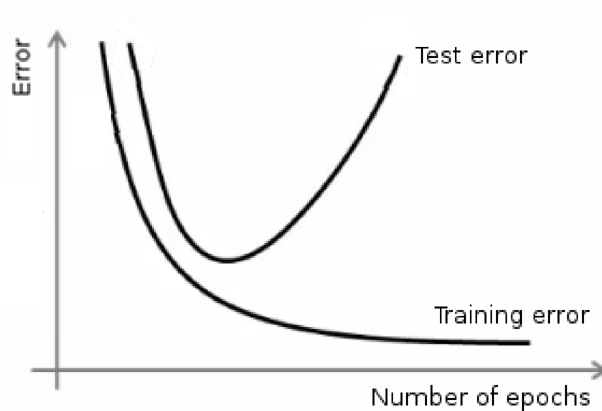


Figura 4.8 Se puede observar que aunque el error en los ejemplos de entrenamiento siga disminuyendo, esto puede provocar un *overfitting* y por tanto una mala generalización de la red ante los ejemplos de testeo, aumentando dicho error.

– Como vemos, en general uno de los objetivos es evitar el *overfitting*, algo que se puede hacer con el diseño de la red imponiendo una capa con *dropout* o con *regularizations* en las variables, aunque no todo es el diseño y el ajuste de variables de la red. Es muy importante la creación los *datatsets* de entrenamiento y testeo. Cuanto más variados y mayor número de ejemplos formen el *dataset* de entrenamiento, mejor se entrenará la red neuronal y podremos evitar el *overfitting* facilmente. Sin embargo, para que el entrenamiento sea óptimo, habrá que estudiar los ejemplos de entrenamiento y observar si realmente son identificables, y si una persona no puede hacerlo, la red neuronal tampoco, impidiendo un buen entrenamiento. Por último, como ya sabemos, los ejemplos que formen el *dataset* de testeo deben ser ejemplos nunca vistos por la red neuronal, y de esta forma, podremos saber con certeza si ésta se ha entrenado de forma adecuada o no.

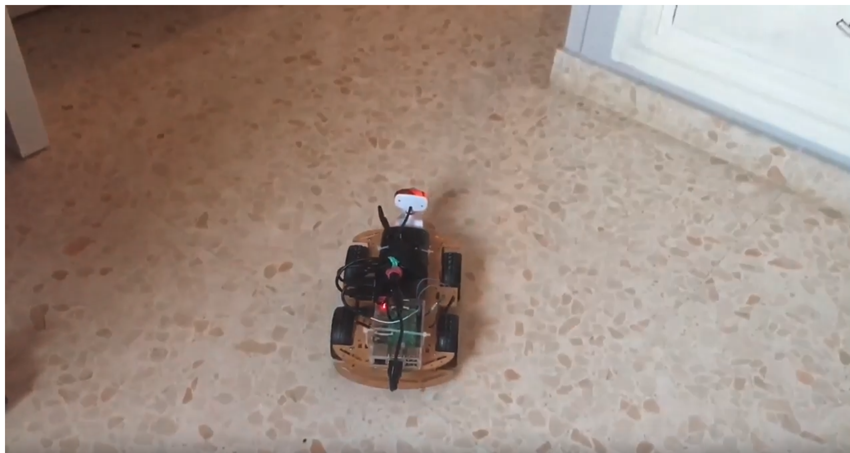


Figura 4.9 Vídeo del robot funcionando a tiempo real (enlace a YouTube).

Sin embargo, a pesar de todas estas aclaraciones y normas para poder diseñar y programar una buena red neuronal, en el mundo del *Deep Learning*, actualmente no existen reglas, ya que el desarrollo en este campo es muy moderno. Por ello, una de las princiales reglas que hay que tener en cuenta, es que nada funcionará a la primera, por lo que el proceso de diseño consistirá más en un proceso de prueba-error, hasta

que finalmente, se obtengan los resultados que se deseen.

Finalmente, teniendo todos estos factores en cuenta, el estudio y diseño de redes neuronales en este proyecto ha resultado ser un éxito. El funcionamiento de la red neuronal con imágenes tomadas por el robot a tiempo real ha dado resultados más que satisfactorios teniendo en cuenta las limitaciones de potencia de hardware y los materiales de baja calidad utilizados. Este éxito se debe principalmente al gran *dataset* creado y el cuidado que se tuvo con éste para obtener imágenes que fuesen claramente identificables, impidiendo cualquier tipo de error durante el entrenamiento de la red.

En este proyecto, se ha profundizado sobre los conceptos de *Deep Learning* y *Deep Neural Networks* usando el conocimiento disponible hasta la fecha. La aplicación de esta base teórica al problema propuesto ha dado como resultado una precisión que podemos calificar de excelente.

No obstante, queda aún mucho recorrido en el campo del *Deep Learning* lo que hace que esta disciplina sea apasionante en un mundo tecnológico que empezamos a vislumbrar.

Bibliografía

[1] N. Buduma, *Fundamentals of Deep Learning*. O'Reilly Media, 2017.

[2] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, p. 85–117, 2015.

[3] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, p. 1798–1828, 2013.

[4] Y. Bengio, Y. LeCun, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[6] A. K. Maan, D. A. Jayadevi, and A. P. James, "A survey of memristive threshold logic circuits," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–13, 2016.

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[8] A. Zell, *Simulation Neuronaler Netze [Simulation of Neural Networks]*. Addison-Wesley, 1994.

[9] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1961.

[10] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

[11] A. L. Maas, A. Y. Hannun, and A. Y. Ng, *Rectifier Nonlinearities Improve Neural Network Acoustic Models (PDF)*. 2014.

[12] Dustinstansbury, "Derivation: Error backpropagation & gradient descent for neural networks." https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/. Última vez visto en Julio, 2018.

[13] M. Nielsen, "Neural networks and deep learning." http://neuralnetworksanddeeplearning.com. Última vez visto en Julio, 2018.

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[15] P. Viola and M. Jones, *Rapid Object Detection using a Boosted Cascade of Simple Features(PDF)*. 2001.

[16] K. He, X. Zhang, S. Ren, and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (PDF)*. 2015.

[17] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks (PDF)*. 2010.

[18] P. Wu, Y. Cao, Y. He, and L. Decai, *Vision-Based Robot Path Planning with Deep Learning (PDF)*. 2017.

Anexos

Anexo 1

```
import numpy as np

'''
En este codigo se va a proceder a hacer una deep neural network con solo
la libreria numpy.
'''

'''             ACTIVATION FUNTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''

#Funcion de activacion ReLU
def relu(input):

    n_elements = np.size(input)

    output = np.empty([1 , n_elements])
    i=0
    for _ in range(n_elements):
        #Leaky ReLUs (evita la muerte de neuronas)
        output[0,i] = max(0.01*input[0,i], input[0,i])
        i+=1

    return output

#Derivada de la funcion de activacion ReLU
def relu_slope(input):

    n_elements = np.size(input)

    output = np.empty([1 , n_elements])
    i=0
    for _ in range(n_elements):
        if input[0,i] > 0:
            output[0 , i] = 1
        else:
            #Pendiente Leaky ReLUs
            output[0 , i] = 0.01
```

```
        i+=1

    return output


'''           LAYERS FUNCTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''
#Definicion de una capa sin funcion de activacion (suele ser capa de salida)
def forward_layer(input , layer):

    #print('Shape input: ' + str(input.shape))
    #print('Shape weights: ' + str(layer['weights'].shape))
    #print('Shape biases: ' + str(layer['biases'].shape))

    nodes_output = np.add(np.dot(input , layer['weights']) , layer['biases'])

    return nodes_output

#Definicion de una capa con funcion de activacion (suelen ser hidden layers)
def forward_layer_with_relu(input , layer):

    #print('Shape input: ' + str(input.shape))
    #print('Shape weights: ' + str(layer['weights'].shape))
    #print('Shape biases: ' + str(layer['biases'].shape))

    #Array de las entradas de todos los nodos de la hidden layer
    nodes_input = np.add(np.dot(input, layer['weights']) , layer['biases'])

    #Array de las salidas (aplicacion funcion de activacion sobre la entrada)
    #de todos los nodos de la hidden layer
    nodes_output = relu(nodes_input)

    #print('Shape output: ' + str(nodes_output.shape))

    return nodes_output , nodes_input


'''          GRADIENT DESCENT Y BACKPROPAGATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''
#Calculo del gradiente de una hidden layer
def gradient_hidden_layer(next_layer , actual_layer , previous_layer , relu):
    '''
    El formato de la matriz gradiente debe ser el mismo que el de la
    matriz en la que se definen los weights de cada layer: las columnas
    corresponderan con cada uno de los nodos que forman la hidden layer de
    la cual queremos cambiar los weights que la alimentan, mientras que
    las filas corresponderan al numero de nodos de la layer anterior, ya
    que cada uno de estos nodos conecta con un weight a todos los nodos de
    la layer actual
    '''

    #Si no tiene ReLU no se aplica su derivada
    if relu==1:
        #Si actual hidden layer tiene funcion de activacion (ReLU):
        relu_sl = relu_slope(actual_layer['input'])
```

```
        #print('Slope in Relu: ' + str(relu_sl) + ' ,shape: ' + str(relu_sl.
            shape))
        actual_layer['error'] = np.multiply(np.dot(next_layer['error'] , (next_
            layer['weights']).T) , relu_sl)
    else:
        actual_layer['error'] = np.dot(next_layer['error'] , (next_layer['
            weights']).T)

    #print('Layer error: ' + str(layer['error']))

    actual_layer['weights_gradient'] = (np.dot((previous_layer['out']).T ,
        actual_layer['error']))
    actual_layer['bias_gradient'] = actual_layer['error']

#Calculo del gradiente de la capa de salida
def gradient_output_layer(final_layer, previous_layer , output_targets , relu):

    #Array con los errores de todos los nodos de output layer
    errors = final_layer['out'] - output_targets

    #print('Error values: ' + str(errors) + ' ,shape: ' + str(errors.shape))

    #Si no tiene ReLU no se aplica su derivada
    if relu==1:
        #Si output layer tiene funcion de activacion (ReLU):
        relu_sl = relu_slope(final_layer['input'])
        #print('Slope in Relu: ' + str(relu_sl) + ' ,shape: ' + str(relu_sl.
            shape))
        final_layer['error'] = np.multiply(errors , relu_sl)
    else:
        final_layer['error'] = errors

    #print('Layer error: ' + str(output_layer['error']))

    final_layer['weights_gradient'] = (np.dot((previous_layer['out']).T , final
        _layer['error']))
    final_layer['bias_gradient'] = final_layer['error']

    #print('Final layer weights gradient: \n' + str(final_layer['weights_
        gradient']))
    #print('Final layer bias gradient: \n' + str(final_layer['bias_gradient']))


'''
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     EN EL CASO DE ACTUALIZAR EL MODELO DE LA RED NEURONAL, MODIFICAR

    EL CODIGO A PARTIR DE ESTE COMENTARIO, EL RESTO DE FUNCIONES PUEDEN

                    SER RECICLADAS SIN MODIFICARLAS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''

#Algoritmo de Backpropagation para calcular el error de todas las capas
def backpropagation(desired_output):
```

```
    gradient_output_layer(out_layer,layer3,desired_output,0)
    #print('Weights gradient 1 = \n' + str(out_layer['weights_gradient']))
    #print('Bias gradient 1 = \n' + str(out_layer['bias_gradient']))

    gradient_hidden_layer(out_layer,layer3,layer2,1)
    #print('Weights gradient 2 = \n' + str(layer3['weights_gradient']))
    #print('Bias gradient 2 = \n' + str(layer3['bias_gradient']))

    gradient_hidden_layer(layer3,layer2,layer1,1)
    #print('Weights gradient 3 = \n' + str(layer2['weights_gradient']))
    #print('Bias gradient 3 = \n' + str(layer2['bias_gradient']))

    gradient_hidden_layer(layer2,layer1,input_layer,1)
    #print('Weights gradient 4 = \n' + str(layer1['weights_gradient']))
    #print('Bias gradient 4 = \n' + str(layer1['bias_gradient']))


def update_variables(learning_rate):

    out_layer['weights'] = out_layer['weights'] - out_layer['weights_gradient']
         * learning_rate
    layer3['weights'] = layer3['weights'] - layer3['weights_gradient'] *
        learning_rate
    layer2['weights'] = layer2['weights'] - layer2['weights_gradient'] *
        learning_rate
    layer1['weights'] = layer1['weights'] - layer1['weights_gradient'] *
        learning_rate

    #print('Weights layer 1: \n' + str(layer1['weights']))
    #print('Weights layer 2: \n' + str(layer2['weights']))
    #print('Weights layer 3: \n' + str(layer3['weights']))
    #print('Weights out layer: \n' + str(out_layer['weights']))

    out_layer['biases'] = out_layer['biases'] - out_layer['bias_gradient'] *
        learning_rate
    layer3['biases'] = layer3['biases'] - layer3['bias_gradient'] * learning_
        rate
    layer2['biases'] = layer2['biases'] - layer2['bias_gradient'] * learning_
        rate
    layer1['biases'] = layer1['biases'] - layer1['bias_gradient'] * learning_
        rate

    #print('Bias layer 1: \n' + str(layer1['biases']))
    #print('Bias layer 2: \n' + str(layer2['biases']))
    #print('Bias layer 3: \n' + str(layer3['biases']))
    #print('Bias out layer: \n' + str(out_layer['biases']))

'''          TRAINING FUNCTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''

def train_model(train_data , target_data , minibatch_size , n_epochs):

    for epoch in range(n_epochs):
        epoch_loss = 0

        i=0
```

```
    while(i < np.shape(train_data)[0]):

        #Inicializacion de los datos del minibatch
        start = i
        end = i + minibatch_size

        #Usamos el error cuadratico medio (mean squared error)
        minibatch_loss = 0

        minibatch_x = np.asmatrix(train_data[start:end , :])
        minibatch_y = np.asmatrix(target_data[start:end , :])

        #En cada minibatch se calculan los gradientes y se hace la
        #media con ellos, despues del minibatch se actualizan los
        #weights
        total_weights_gradient_layer1 = np.zeros((input_data_size , layer1_
            nodes))
        total_weights_gradient_layer2 = np.zeros((layer1_nodes , layer2_
            nodes))
        total_weights_gradient_layer3 = np.zeros((layer2_nodes , layer3_
            nodes))
        total_weights_gradient_output_layer = np.zeros((layer3_nodes , n_
            classes))

        total_bias_gradient_layer1 = np.zeros((1 , layer1_nodes))
        total_bias_gradient_layer2 = np.zeros((1 , layer2_nodes))
        total_bias_gradient_layer3 = np.zeros((1 , layer3_nodes))
        total_bias_gradient_output_layer = np.zeros((1 , n_classes))

        #Se calculan los gradientes totales del minibatch al completo
        for n in range(minibatch_size):
            prediction = neural_network(minibatch_x[n , :])

            if np.isnan(prediction) == 0:
                input_layer['out'] = minibatch_x[n , :]
                backpropagation(minibatch_y[n , :])
                minibatch_loss = minibatch_loss + (prediction - minibatch_y[n
                     , :])**2

                #Se acumulan los gradientes obtenidos para hacer la media
                #luego:
                total_weights_gradient_layer1 += layer1['weights_gradient']
                total_weights_gradient_layer2 += layer2['weights_gradient']
                total_weights_gradient_layer3 += layer3['weights_gradient']
                total_weights_gradient_output_layer += out_layer['weights_
                    gradient']

                total_bias_gradient_layer1 += layer1['bias_gradient']
                total_bias_gradient_layer2 += layer2['bias_gradient']
                total_bias_gradient_layer3 += layer3['bias_gradient']
                total_bias_gradient_output_layer += out_layer['bias_gradient
                    ']

        #Se calculan las medias de los Gradientes para este minibatch
        layer1['weights_gradient'] = total_weights_gradient_layer1 /
            minibatch_size
```

```
            layer2['weights_gradient'] = total_weights_gradient_layer2 /
                minibatch_size
            layer3['weights_gradient'] = total_weights_gradient_layer3 /
                minibatch_size
            out_layer['weights_gradient'] = total_weights_gradient_output_layer
                / minibatch_size

            layer1['bias_gradient'] = total_bias_gradient_layer1 / minibatch_
                size
            layer2['bias_gradient'] = total_bias_gradient_layer2 / minibatch_
                size
            layer3['bias_gradient'] = total_bias_gradient_layer3 / minibatch_
                size
            out_layer['bias_gradient'] = total_bias_gradient_output_layer /
                minibatch_size


            #print('Weights layer 1: \n' + str(layer1['weights']))
            #print('Weights layer 2: \n' + str(layer2['weights']))
            #print('Weights layer 3: \n' + str(layer3['weights']))
            #print('Weights out layer: \n' + str(out_layer['weights']))
            #print('Bias layer 1: \n' + str(layer1['biases']))
            #print('Bias layer 2: \n' + str(layer2['biases']))
            #print('Bias layer 3: \n' + str(layer3['biases']))
            #print('Bias out layer: \n' + str(out_layer['biases']) + '\n\n')

            #Calculamos mean squared error
            minibatch_loss = minibatch_loss / minibatch_size
            #Usamos Gradient Descent para disminuir loss (error, mean squared
            #error)
            update_variables(0.001)

            epoch_loss += minibatch_loss
            i += minibatch_size

        print('Epoch ' + str(epoch+1) + ' completed out of ' + str(n_epochs) +
            ', loss: ' + str(epoch_loss))


'''          NEURAL NETWORK MODEL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''
#Definicion del modelo de red neuronal FeedForward
def neural_network(input_data):

    layer1['out'] , layer1['input'] = forward_layer_with_relu(input_data ,
        layer1)
    #layer1['out'] = forward_layer(input_data , layer1)
    #print('Layer 1 out: ' + str(layer1['out']))

    layer2['out'] , layer2['input'] = forward_layer_with_relu(layer1['out'] ,
        layer2)
    #layer2['out'] = forward_layer(layer1['out'] , layer2)
    #print('Layer 2 out: ' + str(layer2['out']))

    layer3['out'] , layer3['input'] = forward_layer_with_relu(layer2['out'] ,
        layer3)
```

```
    #layer3['out'] = forward_layer(layer2['out'] , layer3)
    #print('Layer 3 out: ' + str(layer3['out']))

    # Output layer sin relu:
    #out_layer['out'] , out_layer['input'] = forward_layer_with_relu(layer3['
        out'] , out_layer)
    out_layer['out'] = forward_layer(layer3['out'] , out_layer)
    #print('Out layer out: ' + str(out_layer['out']))

    return out_layer['out']


'''          MAIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''

'''
Los datos de entrada deben ser matrices, siendo cada fila un solo dato de
entrada.
Eso implica que las entradas pueden ser arrays de muchos elementos (indica las
dimensiones de los datos de entradas en la variable inputa_data_size)
'''
#Los datos de entrenamiento siempre se ponen de forma aleatoria para impedir
#errores.
#En este caso se estan generando 1000 sumas aleatorias, con numeros del 0 al 9
#para hacer aprender a nuestra red neuronal a sumar dos digitos:
m=1000 #Numero de ejemplos de entrenamiento

train_data_x = np.empty([m, 2])
train_data_y = []
x = []
y = []
for i in range(int(m/10)):
    n1 = np.arange(10)
    np.random.shuffle(n1)
    x.extend(n1)

    n2 = np.arange(10)
    np.random.shuffle(n2)
    y.extend(n2)

    train_data_y.extend(n1 + n2)

#train_data_x corresponde a las entradas de la red neuronal de los ejemplos de
#suma
train_data_x[:,0] = x
train_data_x[:,1] = y
#train_data_y son los resultados que queremos obtener con cada entrada de
#train_data_x
train_data_y = (np.asmatrix(train_data_y)).T

#Definimos el numero de entradas que tendra la red neuronal (cada dato de
#entrada es una fila de nuestra matriz de datos)
input_data_size = len(train_data_x[0 , :])

#Definimos el modelo de la red neuronal (numero de neuronas por capa)
layer1_nodes = 100
```

```
layer2_nodes = 50
layer3_nodes = 10

n_classes = 1


#En la inicializacion de los weights y biases se les diminuye su valor en un
#principio para evitar que si hay muchos nodos, las predicciones iniciales sean
#muy altas y por tanto el error inicial tambien. Para ello dividimos los
#valores aleatorios de los parametros entre el numero de neuronas de cada capa
#(en estos conceptos se basa la famosa inicializacion de Xavier):
weights_layer1 = np.random.rand(input_data_size , layer1_nodes) / layer1_nodes
biases_layer1 = np.random.rand(1,layer1_nodes) / layer1_nodes
layer1 = {'weights': weights_layer1 , 'biases': biases_layer1, 'out': [] , '
    input': [] , 'weights_gradient': [] , 'bias_gradient': [] , 'error': [] , '
    size': layer1_nodes}

weights_layer2 = np.random.rand(layer1_nodes , layer2_nodes) / layer2_nodes
biases_layer2 = np.random.rand(1,layer2_nodes) / layer2_nodes
layer2 = {'weights': weights_layer2 , 'biases': biases_layer2, 'out': [] , '
    input': [] , 'weights_gradient': [] , 'bias_gradient': [] , 'error': [] , '
    size': layer2_nodes}

weights_layer3 = np.random.rand(layer2_nodes , layer3_nodes) / layer3_nodes
biases_layer3 = np.random.rand(1,layer3_nodes) / layer3_nodes
layer3 = {'weights': weights_layer3 , 'biases': biases_layer3, 'out': [] , '
    input': [] , 'weights_gradient': [] , 'bias_gradient': [] , 'error': [] , '
    size': layer3_nodes}

weights_output = np.random.rand(layer3_nodes , n_classes)
biases_output = np.random.rand(1, n_classes)
out_layer = {'weights': weights_output , 'biases': biases_output, 'out': [] , '
    input': [] , 'weights_gradient': [] , 'bias_gradient': [] , 'error': [] , '
    size': n_classes}

input_layer = {'out': []}

#Entrenamos modelo
minibatch = 10
epochs = 15
train_model(train_data_x , train_data_y , minibatch , epochs)


#Comprobamos su funcionamiento creando ejemplos de testeo con sumas que
#contengan digitos del 0 al 49:
test_data_x = np.empty([5, 2])

x = np.arange(50)
np.random.shuffle(x)

y = np.arange(50)
np.random.shuffle(y)

test_data_x[:,0] = x[0:5]
test_data_x[:,1] = y[0:5]

#Comprobamos con los datos de testeo que acabamos de crear:
```

```
print('\n   Begin test:')

i=0
for _ in range(test_data_x[:,0].size):
    prediction = neural_network(test_data_x[i , :])
    print('Input: ' + str(test_data_x[i , 0]) + ' + ' + str(test_data_x[i , 1])
         + ' , Prediction rounded: ' + str(np.round(prediction)) + ' , real: '
        + str(prediction))
    i += 1
```

Anexo 2

```
import tensorflow as tf
import numpy as np

#Aqui se define una FeedForward Network para el dataset MNIST

def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de las hidden layers

    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.random_normal
        _initializer(), trainable = True, name = 'b')

    output = tf.nn.relu( tf.add( tf.matmul( input , weights ) , biases ))
    return output

def out_layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de la capa de salida

    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.random_normal
        _initializer(), trainable = True, name = 'b')

    output = tf.add( tf.matmul( input , weights ) , biases )
    return output

def neural_network_model(data):

    #Debido a que con estas redes neuronales se procesan arrays de informacion,
    #se transforman las imagenes a arrays de 28*28
    data = tf.reshape(data, shape=[-1,28*28*1])

    #En esta funcion estamos definiendo el grafo de la red neuronal
    with tf.variable_scope('layer_1'):
        layer1_out = layer(data, [28*28*1, n_nodes_hl1] , [1,n_nodes_hl1] )

    with tf.variable_scope('layer_2'):
        layer2_out = layer(layer1_out, [n_nodes_hl1, n_nodes_hl2] , [1,n_nodes_
            hl2] )

    with tf.variable_scope('layer_3'):
```

```
        layer3_out = layer(layer2_out, [n_nodes_hl2, n_nodes_hl3] , [1,n_nodes_
            hl3] )

    with tf.variable_scope('output'):
        output = out_layer(layer3_out, [n_nodes_hl3, n_classes] , [1,n_classes]
             )

    return output

def train_model(cost):

    #Learning rate por defecto = 0.001
    optimizer = tf.train.AdamOptimizer()

    #La funcion minimize combina las funciones: compute_gradients y
    #apply_gradients
    train_operation = optimizer.minimize(cost)

    return train_operation

def loss(prediction, y):

    #Esta funcion ya realiza el softmax sobre los logits, por lo que espera
    #logits que no esten escalados. Si usas la salida con softmax, esto no dara
    #buenos resultados. Esta funcion calcula el error con la Cross Entropy loss
    #function para cada salida de la red neuronal
    entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits = prediction,
        labels = y)

    #A partir del error en las salidas, calculamos un valor unico haciendo la
    #media del error
    cost = tf.reduce_mean(entropy)

    return cost

def eval_test(prediction, y):

    #Se comprueba que la salida corresponde con la deseada
    correct = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))

    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    return accuracy



#Cargamos los dataset de entrenamiento y de testeo:
mnist = tf.keras.datasets.mnist
(train_x, train_y),(test_x, test_y) = mnist.load_data()

with tf.Session() as sess:
    train_y = tf.one_hot(train_y, 10).eval()
    test_y = tf.one_hot(test_y, 10).eval()

n_train_examples = np.shape(train_x)[0]
print('Loaded ' + str(n_train_examples) + ' Training Images')
print('Loaded ' + str(np.shape(test_x)[0]) + ' Test Images')
```

```
batch_size = 100
n_batches = n_train_examples/batch_size
n_epochs = 100
#One-hot encoding
n_classes = 10

#Modelo de la red neuronal
n_nodes_hl1 = 1000
n_nodes_hl2 = 500
n_nodes_hl3 = 100

#En MNIST las imagenes estan dadas en formato [batch, height, width]
x = tf.placeholder(tf.float32, shape=[None, 28 , 28])
y = tf.placeholder(tf.float32, shape=[None, n_classes])

#Creamos el subgrafo de la red neuronal
prediction = neural_network_model(x)

#Creamos el subgrafo que calcule el error de la red neuronal
cost = loss(prediction, y)

#Creamos el subgrafo que entrene la red neuronal
train_op = train_model(cost)

#Creamos el subgrafo que calcule el porcentaje de aciertos sobre las imagenes
#de testeo
accuracy = eval_test(prediction,y)

#Iniciamos sesion para poder ejecutar todo nuestro modelo y poder entrenarlo
with tf.Session() as sess:

    #Iniciamos todas las variables como weights y biases
    sess.run(tf.global_variables_initializer())

    prev_acc = 0
    #Bucle para completar todos los epochs
    for epoch in range(n_epochs):
        total_epoch_cost = 0
        avg_epoch_cost = 0

        i=0

        #Bucle para los minibatches
        while(i < n_train_examples):
            start = i
            end = i + batch_size

            batch_x = train_x[start:end][:]
            batch_y = train_y[start:end][:]

            sess.run(train_op, feed_dict = {x: batch_x , y: batch_y})
            minibatch_cost = sess.run(cost, feed_dict = {x: batch_x, y: batch_y
                })

            total_epoch_cost += minibatch_cost
            i += batch_size
```

```
        avg_epoch_cost = total_epoch_cost/n_batches

        print('Epoch', epoch+1, 'completed out of', n_epochs, 'cost:', avg_
            epoch_cost)

        acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y})*100

        if acc < prev_acc:
            prev_acc = acc
            print('Test accuracy: ' + str(acc) + ' %')
            print('Possible overfitting!!')
            #ans = raw_input('Continue?(y/n): ')
            #if ans == 'n':
            #    break
        elif acc > 98:
            print('Test accuracy: ' + str(acc) + '%')
            ans = raw_input('High accuracy... Continue?(y/n): ')
            if ans == 'n':
                break
        else:
           print('Test accuracy: ' + str(acc) + '%')
           prev_acc = acc

    acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y})*100
    print('Accuracy: ' + str(acc) + ' %')
```

Anexo 3

```
import tensorflow as tf
from images_and_labels import load_images_and_labels_arrays
from images_and_labels import create_images_and_labels
import cv2 as cv
import numpy as np
import time

#En este codigo nos encontramos con una FeedForward Network para entrenarla con
#nuestro propio dataset

def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de las hidden layers

    weights = tf.get_variable(shape = weight_shape, initializer = tf.random_
        normal_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.random_normal
        _initializer(), trainable = True, name = 'b')

    output = tf.nn.relu( tf.add( tf.matmul( input , weights ) , biases ))
    return output

def out_layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de la capa de salida
```

```
    weights = tf.get_variable(shape = weight_shape, initializer = tf.random_
        normal_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.random_normal
        _initializer(), trainable = True, name = 'b')

    output = tf.add( tf.matmul( input , weights ) , biases )
    return output

def neural_network_model(data):

    #En esta funcion estamos definiendo el grafo de la red neuronal

    with tf.variable_scope('layer_1'):
        layer1_out = layer(data, [size_images, n_nodes_hl1] , [1,n_nodes_hl1] )

    with tf.variable_scope('layer_2'):
        layer2_out = layer(layer1_out, [n_nodes_hl1, n_nodes_hl2] , [1,n_nodes_
            hl2] )

    with tf.variable_scope('layer_3'):
        layer3_out = layer(layer2_out, [n_nodes_hl2, n_nodes_hl3] , [1,n_nodes_
            hl3] )

    with tf.variable_scope('output'):
        output = out_layer(layer3_out, [n_nodes_hl3, n_classes] , [1,n_classes]
             )

    return output

def train_model(cost):

    #Learning rate por defecto = 0.001
    optimizer = tf.train.AdamOptimizer()

    #La funcion minimize combina las funciones: compute_gradients y
    #apply_gradients
    train_operation = optimizer.minimize(cost)

    return train_operation

def loss(prediction, y):

    #Esta funcion ya realiza el softmax sobre los logits, por lo que espera
    #logits que no esten escalados. Si usas la salida con softmax, esto no dara
    #buenos resultados. Esta funcion calcula el error con la Cross Entropy loss
    #function para cada salida de la red neuronal
    entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits = prediction,
        labels = y)

    #A partir del error en las salidas, calculamos un valor unico haciendo la
    #media del error
    cost = tf.reduce_mean(entropy)

    return cost

def eval_test(prediction, y):
```

```
    #Se comprueba que la salida corresponde con la deseada
    correct = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))

    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    return accuracy



#Cargamos los dataset de entrenamiento y de testeo:
train_x , labels , train_y = load_images_and_labels_arrays('Datasets/Train_
    Dataset')

size_images = np.shape(train_x)[1]
n_train_examples = np.shape(train_x)[0]
print('Loaded ' + str(n_train_examples) + ' Training Images')

test_x, labels, test_y = load_images_and_labels_arrays('Datasets/Test_Dataset')
print('Loaded ' + str(np.shape(test_x)[0]) + ' Test Images')

#Modelo de la red neuronal
n_nodes_hl1 = 2000
n_nodes_hl2 = 1000
n_nodes_hl3 = 500

batch_size = 100
n_batches = n_train_examples/batch_size
n_epochs = 100
#One-hot encoding
n_classes = 3

#Las imagenes no estan dadas como una matriz (height x width), sino como un
#vector. Debido a ello, tendremos que el shape de x (entrada de la red), no
#tendra altura determinada (ejemplos del batch), y el ancho sera de
#size_images, que en nuestro caso es 112x112x3 (RGB)
x = tf.placeholder(tf.float32, shape=[None, size_images])
y = tf.placeholder(tf.float32, shape=[None, n_classes])

#Creamos el subgrafo de la red neuronal
prediction = neural_network_model(x)

#Creamos el subgrafo que calcule el error de la red neuronal
cost = loss(prediction, y)

#Creamos el subgrafo que entrene la red neuronal
train_op = train_model(cost)

#Creamos el subgrafo que calcule el porcentaje de aciertos sobre las imagenes
#de testeo
accuracy = eval_test(prediction,y)

#Iniciamos sesion para poder ejecutar todo nuestro modelo y poder entrenarlo
with tf.Session() as sess:

    #Iniciamos todas las variables como weights y biases
    sess.run(tf.global_variables_initializer())
```

```
prev_acc = 0
#Bucle para completar todos los epochs
for epoch in range(n_epochs):
    total_epoch_cost = 0
    avg_epoch_cost = 0

    i=0

    #Bucle para los minibatches
    while(i < n_train_examples):
        start = i
        end = i + batch_size

        batch_x = train_x[start:end][:]
        batch_y = train_y[start:end][:]

        sess.run(train_op, feed_dict = {x: batch_x , y: batch_y})
        minibatch_cost = sess.run(cost, feed_dict = {x: batch_x, y: batch_y
            })

        total_epoch_cost += minibatch_cost
        i += batch_size

    avg_epoch_cost = total_epoch_cost/n_batches

    print('Epoch', epoch+1, 'completed out of', n_epochs, 'cost:', avg_
        epoch_cost)

    acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y})*100

    if acc < prev_acc:
        prev_acc = acc
        print('Test accuracy: ' + str(acc) + ' %')
        print('Possible overfitting!!')
        #ans = raw_input('Continue?(y/n): ')
        #if ans == 'n':
        #    break
    elif acc > 80:
        print('Test accuracy: ' + str(acc) + '%')
        ans = raw_input('High accuracy... Continue?(y/n): ')
        if ans == 'n':
            break
    else:
       print('Test accuracy: ' + str(acc) + '%')
       prev_acc = acc

acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y})*100
print('Accuracy: ' + str(acc) + ' %')


'''
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
         Evaluamos cualquier imagen de la camara
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''
cap = cv.VideoCapture(0)
```

```
    while(True):

      #Capturamos la imagenes frame por frame a cada iteracion del bucle
      ret, frame = cap.read()
      cv.imshow('window', frame)

      #Convierto las imagenes a RGB ya que openCV lee en BGR:
      frame = cv.cvtColor(frame, cv.COLOR_BGR2RGB)
      key = cv.waitKey(1)

      if key & 0xFF == ord('q'): break

      img = tf.reshape(tf.cast(tf.image.resize_images(frame, [112,112]), tf.
          uint8), shape=[-1,112*112*3]).eval()

      output = sess.run(prediction, feed_dict={x: img , keep_prob: 1.0})

      pred = tf.argmax(output,1).eval()

      if pred == 0:
          out = 'pepino'
      elif pred == 1:
          out='limon'
      else:
          out = 'tomate'

      print(out)

      time.sleep(0.5)

    cap.release()
    cv.destroyAllWindows()
```

Anexo 4

```
import numpy as np
import cv2 as cv
import os


'''
Esta primera funcion nos permite tomar imagenes de forma continua como si de
un video se tratara. Todas las imagenes tomadas de las camaras se guardan en
la carpeta indicada.
'''
def start_recording(folder):
    cap = cv.VideoCapture(0)
    dumb_count=0


    print('Start recording? start/stop (s) ')
    print('To quit press "q"')
    save=0
```

```
    while(True):

        #Capturamos la imagenes frame por frame a cada iteracion del bucle
        ret, frame = cap.read()

        #Se muestra la imagen tomada
        cv.imshow('frame',frame)

        if save==1:
            cv.imwrite(os.getcwd() + '/' + folder + '/Image' + str(dumb_count)
                +'.jpeg', frame)
            dumb_count+=1
            print('Image ' + str(dumb_count+1) + ' saved')

        key=cv.waitKey(1)
        if  key & 0xFF == ord('q'):
            break
        if key & 0xFF == ord('s'):
            if save==0:
                save=1
            else:
                save=0

    #Cuando se termina de tomar imagenes, se desconecta la camara
    cap.release()
    cv.destroyAllWindows()

'''
Esta segunda funcion sirve para tomar una unica imagen cada vez que se indique
guardandose en la carpeta indicada en el argumento
'''
def take_one_shot(folder):
    cap = cv.VideoCapture(0)
    dumb_count=0


    print('Take photo? (y) ')
    print('To quit press "q"')
    save=0

    while(True):

        #Capturamos la imagenes frame por frame a cada iteracion del bucle
        ret, frame = cap.read()

        #Se muestra la imagen tomada
        cv.imshow('frame',frame)

        if save==1:
            cv.imwrite(os.getcwd() + '/' + folder + '/Image' + str(dumb_count)
                +'.jpeg', frame)
            dumb_count+=1
            save=0

        key=cv.waitKey(1)
        if  key & 0xFF == ord('q'):
            break
```

```
        if key & 0xFF == ord('y'):
            save=1

    #Cuando se termina de tomar imagenes, se desconecta la camara
    cap.release()
    cv.destroyAllWindows()
```

Anexo 5

```
import tensorflow as tf
import numpy as np
import os
import cv2 as cv
import cPickle

'''
Estas funciones se han creado para que el argumento 'dir' contenga el path de
una carpeta de un dataset, la cual contendra subcarpetas (training dataset,
test dataset...), que cada una de ellas contendra subcarpetas con las imagenes
clasificadas segun las respectivas clases (cada subcarpeta tendra el nombre de
cada clase). Las imagenes son transformadas a una unica linea de pixeles,
haciendo facil la lectura de cada imagen del dataset y facil la reconstruccion
(reshape), ya que se devuelve una matriz en la que cada fila corresponde a una
imagen. Luego devuelve otras dos matrices con las etiquetas de cada imagen, una
en formato string y otro en formato one-hot (cada fila corresponde a la
etiqueta o clase de la imagen de la misma fila de la matriz de imagenes).
'''
def create_images_and_labels(dir,n_classes):
  '''El argumento dir indica el nombre de la carpeta donde se encuentran el
  resto de subcarpetas con las imagenes clasificadas
                     -> Subcarpeta 'label 1' (imagenes de label 1, gatos)
  dir (carpeta raiz) -> Subcarpeta 'label 2' (imagenes de label 2, perros)
                     -> Subcarpeta 'label 3' (imagenes de label 3, vacas)
  '''
  dir_list=[]

  gen = tf.gfile.Walk(os.getcwd() + '/' + dir) #Devuelve un generator (yield)

  for (top, subdir, files) in gen:
    dir_list.append(top)

  #Array con los path completos a todas las subcarpetas de la carpeta raiz
  image_dirs=np.array(dir_list)
  #print(image_dirs)
  #dim=image_dirs.shape #Te devuelve un tuple con las dimensiones

  file_list=[]
  labels=[]
  labels_one_hot=[]

  n=0
  pos=0
  initial_label=[0,] * n_classes
```

```
for image_dir in image_dirs[1:,]:
  '''Las imagenes estan clasificadas en subcarpetas, cada subcarpeta
  indica la clase de cada imagen, y por tanto su label'''
  dir_name = os.path.basename(image_dir) #Label, subdirectorio

  file_glob = os.path.join(image_dir,'*.jpeg')
  #print(file_glob)

  #Paths de las distintas imagenes encontradas de tipo jpeg (son tuples
  #con los paths de las imagenes en cada directorio):
  image_file_list = tf.gfile.Glob(file_glob)
  #print(image_file_list)

  #Si hago append estoy uniendo listas en listas, yo solo quiero una lista
  #(array) no varias listas (matriz), por ello se hace un extend:
  file_list.extend(image_file_list) #Lista con todos los path de todas
  #las imagenes

  for image_class in image_file_list:

    labels.append(dir_name)

    label = initial_label[:]
    label[n] = 1
    labels_one_hot.append(label)

  n+=1
  #print(labels_one_hot)

#Se reconvierten las imagenes de las subcarpetas de 3 matrices a un solo
#array con el numero de pixeles disminuido:
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())

  images=[]

  n=0

  #Leemos la imagen transformandola en un Tensor para que TensorFlow pueda
  #trabajar con ella
  for img in file_list:

    #Abre la imagen dado el path
    img_opened=tf.read_file(img)

    #Detecta la imagen y la decodifica para que pase la imagen a un Tensor de
    #valores uint8. De esta forma TF puede trabajar con la imagen:
    decoded_img=tf.image.decode_jpeg(img_opened, channels=3)

    #Ten cuidado, por defecto, openCV utiliza el formato BGR en vez de RGB:
    #cv.imwrite(os.getcwd() + '/' + dir + '_decoded/' + str(labels[n]) + '/
        Image' + str(n) +'.jpeg', cv.cvtColor(decoded_img.eval() , cv.COLOR_
        RGB2BGR))

    #Cambia el tamanio de la imagen
    image_resized=tf.cast(tf.image.resize_images(decoded_img, [112,112]), tf.
        uint8)
```

```
      #cv.imwrite(os.getcwd() + '/' + dir + '_resized/' + str(labels[n]) + '/
          Image' + str(n) +'.jpeg', cv.cvtColor(image_resized.eval() , cv.COLOR
          _RGB2BGR))

      #Convertimos la imagen en un array de pixeles
      image_reshaped=tf.reshape(image_resized,[1,112*112*3])

      #Reconvertimos en una matriz de numpy ya que las sesiones de TF no
      #trabajan con tensors, sino con numpy arrays, escalares..etc
      new_image=image_reshaped.eval()

      #Creamos un numpy array con la imagen (no matriz) para poder hacer
      #append en una lista correctamente
      new_image = np.squeeze(new_image)
      #cv.imwrite(os.getcwd() + '/' + dir + '_reshaped/' + str(labels[n]) + '/
          Image' + str(n) +'.jpeg', cv.cvtColor(new_image , cv.COLOR_RGB2BGR))

      images.append(new_image)

      if n % 250 == 0:
        print('Procesadas ' + str(n) + ' imagenes')


      n+=1

  #Las listas de listas, las pasamos a matrices de numpy:
  images = np.asmatrix(images)
  labels_one_hot = np.asmatrix(labels_one_hot)
  labels = np.asmatrix(labels).T

  #Mezclamos tanto las imagenes como sus labels
  random_index = np.random.permutation(np.shape(images)[0])
  images = images[random_index][:]
  labels_one_hot = labels_one_hot[random_index][:]
  labels = labels[random_index][:]

  #Guardamos los numpys arrays
  dir,subdir = dir.split('/')
  name1 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_images.p'
  name2 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_labels_one_hot.p'
  name3 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_labels.p'
  cPickle.dump( images, open( name1, 'wb' ) )
  cPickle.dump( labels_one_hot, open( name2, 'wb' ) )
  cPickle.dump( labels, open( name3, 'wb' ) )

  return images, labels, labels_one_hot


'''
Esta funcion tiene la misma funcion que la anterior pero transforma las
imagenes a blanco y negro en vez de RGB
'''
def create_images_and_labels_grayscale(dir,n_classes):
  '''El argumento dir indica el nombre de la carpeta donde se encuentran el
  resto de subcarpetas con las imagenes clasificadas
                    -> Subcarpeta 'label 1' (imagenes de label 1, gatos)
  dir (carpeta raiz) -> Subcarpeta 'label 2' (imagenes de label 2, perros)
```

```
                   -> Subcarpeta 'label 3' (imagenes de label 3, vacas)
'''
dir_list=[]

gen = tf.gfile.Walk(os.getcwd() + '/' + dir) #Devuelve un generator (yield)

for (top, subdir, files) in gen:
  dir_list.append(top)

#Array con los path completos a todas las subcarpetas de la carpeta raiz
image_dirs=np.array(dir_list)
#print(image_dirs)
#dim=image_dirs.shape #Te devuelve un tuple con las dimensiones

file_list=[]
labels=[]
labels_one_hot=[]

n=0
pos=0
initial_label=[0,] * n_classes

for image_dir in image_dirs[1:,]:
  '''Las imagenes estan clasificadas en subcarpetas, cada subcarpeta
  indica la clase de cada imagen, y por tanto su label'''
  dir_name = os.path.basename(image_dir) #Label, subdirectorio

  file_glob = os.path.join(image_dir,'*.jpeg')
  #print(file_glob)

  #Paths de las distintas imagenes encontradas de tipo jpeg (son tuples
  #con los paths de las imagenes en cada directorio):
  image_file_list = tf.gfile.Glob(file_glob)
  #print(image_file_list)

  #Si hago append estoy uniendo listas en listas, yo solo quiero una lista
  #(array) no varias listas (matriz), por ello se hace un extend:
  file_list.extend(image_file_list) #Lista con todos los path de todas
  #las imagenes

  for image_class in image_file_list:

    labels.append(dir_name)

    label = initial_label[:]
    label[n] = 1
    labels_one_hot.append(label)

  n+=1
  #print(labels_one_hot)

#Se reconvierten las imagenes de las subcarpetas de 3 matrices a un solo
#array con el numero de pixeles disminuido:
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())

  images=[]
```

```
  n=0

  #Leemos la imagen transformandola en un Tensor para que TensorFlow pueda
  #trabajar con ella
  for img in file_list:

    #Abre la imagen dado el path
    img_opened=tf.read_file(img)

    #Detecta la imagen y la decodifica para que pase la imagen a un Tensor de
    #valores uint8. De esta forma TF puede trabajar con la imagen:
    decoded_img=tf.image.decode_jpeg(img_opened, channels=3)

    #Ten cuidado, por defecto, openCV utiliza el formato BGR en vez de RGB:
    #cv.imwrite(os.getcwd() + '/' + dir + '_decoded/' + str(labels[n]) + '/
        Image' + str(n) +'.jpeg', cv.cvtColor(decoded_img.eval() , cv.COLOR_
        RGB2BGR))

    #Pasamos la imagen a blanco y negro
    img_grayscale = tf.image.rgb_to_grayscale(decoded_img)

    #Cambia el tamanio de la imagen
    image_resized=tf.cast(tf.image.resize_images(img_grayscale, [112,112]),
        tf.uint8)
    #cv.imwrite(os.getcwd() + '/' + dir + '_resized/' + str(labels[n]) + '/
        Image' + str(n) +'.jpeg', cv.cvtColor(image_resized.eval() , cv.COLOR
        _RGB2BGR))

    #Convertimos la imagen en un array de pixeles
    image_reshaped=tf.reshape(image_resized,[1,112*112*1])

    #Reconvertimos en una matriz de numpy ya que las sesiones de TF no
    #trabajan con tensors, sino con numpy arrays, escalares..etc
    new_image=image_reshaped.eval()

    #Creamos un numpy array con la imagen (no matriz) para poder hacer
    #append en una lista correctamente
    new_image = np.squeeze(new_image)
    #cv.imwrite(os.getcwd() + '/' + dir + '_reshaped/' + str(labels[n]) + '/
        Image' + str(n) +'.jpeg', cv.cvtColor(new_image , cv.COLOR_RGB2BGR))

    images.append(new_image)

    if n % 250 == 0:
      print('Procesadas ' + str(n) + ' imagenes')

    n+=1

#Las listas de listas, las pasamos a matrices de numpy:
images = np.asmatrix(images)
labels_one_hot = np.asmatrix(labels_one_hot)
labels = np.asmatrix(labels).T

#Mezclamos tanto las imagenes como sus labels
random_index = np.random.permutation(np.shape(images)[0])
images = images[random_index][:]
```

```
  labels_one_hot = labels_one_hot[random_index][:]
  labels = labels[random_index][:]

  #Guardamos los numpys arrays (las carpetas y archivos deben estar creados!)
  dir,subdir = dir.split('/')
  name1 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_images.p'
  name2 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_labels_one_hot.p'
  name3 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_labels.p'
  cPickle.dump( images, open( name1, 'wb' ) )
  cPickle.dump( labels_one_hot, open( name2, 'wb' ) )
  cPickle.dump( labels, open( name3, 'wb' ) )

  return images, labels, labels_one_hot


'''
Esta funcion agrega al dataset indicado con dir (leyendo el pickle con las
imagenes ya creadas), un conjunto de imagenes en blanco y negro ya clasificadas
que se encuentran en el directorio dir_add
'''
def add_images_and_labels_grayscale(dir,dir_add,n_classes):

  dir_list=[]

  gen = tf.gfile.Walk(os.getcwd() + '/' + dir_add)

  for (top, subdir, files) in gen:
    dir_list.append(top)

  #Array con los path completos a todas las subcarpetas de la carpeta raiz
  image_dirs=np.array(dir_list)
  #print(image_dirs)
  #dim=image_dirs.shape #Te devuelve un tuple con las dimensiones

  file_list=[]
  labels_add=[]
  labels_one_hot_add=[]

  n=0
  pos=0
  initial_label=[0,] * n_classes

  for image_dir in image_dirs[1:,]:
    '''Las imagenes estan clasificadas en subcarpetas, cada subcarpeta
    indica la clase de cada imagen, y por tanto su label'''
    dir_name = os.path.basename(image_dir) #Label, subdirectorio

    file_glob = os.path.join(image_dir,'*.jpeg')
    #print(file_glob)

    #Paths de las distintas imagenes encontradas de tipo jpeg (son tuples
    #con los paths de las imagenes en cada directorio):
    image_file_list = tf.gfile.Glob(file_glob)
    #print(image_file_list)

    #Si hago append estoy uniendo listas en listas, yo solo quiero una lista
    #(array) no varias listas (matriz), por ello se hace un extend:
```

```
    file_list.extend(image_file_list) #Lista con todos los path de todas
    #las imagenes

    for image_class in image_file_list:

      labels_add.append(dir_name)

      label = initial_label[:]
      label[n] = 1
      labels_one_hot_add.append(label)

    n+=1
    #print(labels_one_hot)

  #Se reconvierten las imagenes de las subcarpetas de 3 matrices a un solo
  #array con el numero de pixeles disminuido:
  with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    images_add=[]

    n=0

    #Leemos la imagen transformandola en un Tensor para que TensorFlow pueda
    #trabajar con ella
    for img in file_list:

      #Abre la imagen dado el path
      img_opened=tf.read_file(img)

      #Detecta la imagen y la decodifica para que pase la imagen a un Tensor de
      #valores uint8. De esta forma TF puede trabajar con la imagen:
      decoded_img=tf.image.decode_jpeg(img_opened, channels=3)

      #Ten cuidado, por defecto, openCV utiliza el formato BGR en vez de RGB:
      #cv.imwrite(os.getcwd() + '/' + dir + '_decoded/' + str(labels[n]) + '/
          Image' + str(n) +'.jpeg', cv.cvtColor(decoded_img.eval() , cv.COLOR_
          RGB2BGR))

      #Pasamos la imagen a blanco y negro
      img_grayscale = tf.image.rgb_to_grayscale(decoded_img)

      #Cambia el tamanio de la imagen
      image_resized=tf.cast(tf.image.resize_images(img_grayscale, [112,112]),
          tf.uint8)
      #cv.imwrite(os.getcwd() + '/' + dir + '_resized/' + str(labels[n]) + '/
          Image' + str(n) +'.jpeg', cv.cvtColor(image_resized.eval() , cv.COLOR
          _RGB2BGR))

      #Convertimos la imagen en un array de pixeles
      image_reshaped=tf.reshape(image_resized,[1,112*112*1])

      #Reconvertimos en una matriz de numpy ya que las sesiones de TF no
      #trabajan con tensors, sino con numpy arrays, escalares..etc
      new_image=image_reshaped.eval()

      #Creamos un numpy array con la imagen (no matriz) para poder hacer
```

```
      #append en una lista correctamente
      new_image = np.squeeze(new_image)
      #cv.imwrite(os.getcwd() + '/' + dir + '_reshaped/' + str(labels[n]) + '/
          Image' + str(n) +'.jpeg', cv.cvtColor(new_image , cv.COLOR_RGB2BGR))

      images_add.append(new_image)

      if n % 250 == 0:
        print('Procesadas ' + str(n) + ' imagenes')

      n+=1

  #Las listas de listas, las pasamos a matrices de numpy:
  images_add = np.asmatrix(images_add)
  labels_one_hot_add = np.asmatrix(labels_one_hot_add)
  labels_add = np.asmatrix(labels_add).T

  #Cargamos el dataset al que queramos agregar dichas imagenes:
  images, labels, labels_one_hot = load_images_and_labels_arrays(dir)

  #Agregamos las nuevas imagenes:
  images = np.append(images, images_add, axis=0)
  labels = np.append(labels, labels_add, axis=0)
  labels_one_hot = np.append(labels_one_hot, labels_one_hot_add, axis=0)

  #Mezclamos tanto las imagenes como sus labels
  random_index = np.random.permutation(np.shape(images)[0])
  images = images[random_index][:]
  labels_one_hot = labels_one_hot[random_index][:]
  labels = labels[random_index][:]

  #Guardamos los numpys arrays (las carpetas y archivos deben estar creados!)
  dir,subdir = dir.split('/')
  name1 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_images.p'
  name2 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_labels_one_hot.p'
  name3 = os.getcwd() + '/' + dir + '/Data/' + subdir + '_labels.p'
  cPickle.dump( images, open( name1, 'wb' ) )
  cPickle.dump( labels_one_hot, open( name2, 'wb' ) )
  cPickle.dump( labels, open( name3, 'wb' ) )

  return images, labels, labels_one_hot

'''
Esta funcion lee los .p (archivos pickle) creados por las funciones anteriores
para devolver las mtrices de imagenes y labels creados
'''
def load_images_and_labels_arrays(dir):
  dir,subdir = dir.split('/')
  images = cPickle.load( open(os.getcwd() + '/' + dir + '/Data/' + subdir + '_
      images.p' , "rb" ) )
  labels_one_hot = cPickle.load( open(os.getcwd() + '/' + dir + '/Data/' +
      subdir + '_labels_one_hot.p' , "rb" ) )
  labels = cPickle.load( open(os.getcwd() + '/' + dir + '/Data/' + subdir + '_
      labels.p' , "rb" ) )

  return images, labels, labels_one_hot
```

Anexo 6

```
import tensorflow as tf
from images_and_labels import load_images_and_labels_arrays
from images_and_labels import create_images_and_labels
import cv2 as cv
import numpy as np
import os

#Convolutional Neural Network para el dataset MNIST

def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una fully-connected layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0.1), trainable = True, name = 'b')

    output = tf.nn.leaky_relu( tf.nn.bias_add( tf.matmul( input , weights ) ,
        biases ))
    return output

def conv2d(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una convolutional layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0.1), trainable = True, name = 'b')

    conv_out = tf.nn.conv2d(input , weights , strides = [1,1,1,1], padding = '
        SAME')

    output = tf.nn.leaky_relu( tf.nn.bias_add( conv_out , biases ))
    return output

def max_pool(input):
    #En esta funcion se define el grafo de una max pooling layer

    output = tf.nn.max_pool(input , ksize = [1,2,2,1] , strides = [1,2,2,1] ,
        padding = 'SAME')
    return output

def neural_network_model(data , keep_prob):

    #Las imagenes de MNIST no tienen el formato de [height, width, channels],
    #sino que estan dadas en formato [height, width], y las Convnet procesan
    #volumenes de informacion, con lo que se debe remodelar las dimensiones de
    #los datos de entrada
    data = tf.reshape(data, shape=[-1,28,28,1])

    #En esta funcion estamos definiendo el grafo de la red neuronal
    with tf.variable_scope('layer_1'):
        #La convolucion poseera 64 filtros de 5x5 que se aplican a un canal
        conv_1 = conv2d(data , [5,5,1,64] , [64])
        pool_1 = max_pool(conv_1)
```

```
    with tf.variable_scope('layer_2'):
        #La convolucion poseera 64 filtros de 5x5 que se aplican a los 64
        #feature maps formados por la convolucion anterior
        conv_2 = conv2d(pool_1 , [5,5,64,64] , [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope('layer_fc'):
        #Los feature maps anteriores se aplanan para formar 1 array de datos
        #formados por los resultados de los features maps y asi formar una red
        #FeedForward fully-connected con dichos datos
        pool_flattened = tf.reshape(pool_2 , [-1,7*7*64])

        fc_out = layer(pool_flattened , [7*7*64 , 1024] , [1024])

        #Para reducir Overfitting se aplica el proceso de dropout. Para ello se
        #usa un placeholder y asi podremos activar dropout en entrenamiento y
        #desactivar en testeo
        fc_drop = tf.nn.dropout(fc_out , keep_prob)

    with tf.variable_scope('output'):
        output = layer(fc_drop , [1024 , 10] , [10])

    return output

def train_model(cost):

    #Learning rate por defecto = 0.001
    optimizer = tf.train.AdamOptimizer()

    #La funcion minimize combina las funciones: compute_gradients y
    #apply_gradients
    train_operation = optimizer.minimize(cost)

    return train_operation

def loss(prediction, y):

    #Esta funcion ya realiza el softmax sobre los logits, por lo que espera
    #logits que no esten escalados. Si usas la salida con softmax, esto no dara
    #buenos resultados. Esta funcion calcula el error con la Cross Entropy loss
    #function para cada salida de la red neuronal
    entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits = prediction,
        labels = y)

    #A partir del error en las salidas, calculamos un valor unico haciendo la
    #media del error
    cost = tf.reduce_mean(entropy)

    return cost

def eval_test(prediction, y):

    #Se comprueba que la salida corresponde con la deseada
    correct = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))

    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

```
    return accuracy



#Cargamos los dataset de entrenamiento y de testeo:
mnist = tf.keras.datasets.mnist
(train_x, train_y),(test_x, test_y) = mnist.load_data()

with tf.Session() as sess:
    train_y = tf.one_hot(train_y, 10).eval()
    test_y = tf.one_hot(test_y, 10).eval()

n_train_examples = np.shape(train_x)[0]
print('Loaded ' + str(n_train_examples) + ' Training Images')
print('Loaded ' + str(np.shape(test_x)[0]) + ' Test Images')

batch_size = 50
n_batches = n_train_examples/batch_size
n_epochs = 100
#One-hot encoding
n_classes = 10

#En MNIST las imagenes estan dadas en formato [batch, height, width]
x = tf.placeholder(tf.float32, shape=[None, 28 , 28])
y = tf.placeholder(tf.float32, shape=[None, n_classes])
keep_prob = tf.placeholder(tf.float32, shape = None)

#Creamos el subgrafo de la red neuronal
prediction = neural_network_model(x,keep_prob)

#Creamos el subgrafo que calcule el error de la red neuronal
cost = loss(prediction, y)

#Creamos el subgrafo que entrene la red neuronal
train_op = train_model(cost)

#Creamos el subgrafo que calcule el porcentaje de aciertos sobre las imagenes
#de testeo
accuracy = eval_test(prediction,y)

#Iniciamos sesion para poder ejecutar todo nuestro modelo y poder entrenarlo
with tf.Session() as sess:

    #Iniciamos todas las variables como weights y biases
    sess.run(tf.global_variables_initializer())

    prev_acc = 0
    #Bucle para completar todos los epochs
    for epoch in range(n_epochs):
        total_epoch_cost = 0
        avg_epoch_cost = 0

        i=0

        #Bucle para los minibatches
        while(i < n_train_examples):
```

```
            start = i
            end = i + batch_size

            batch_x = train_x[start:end][:]
            batch_y = train_y[start:end][:]

            sess.run(train_op, feed_dict = {x: batch_x , y: batch_y , keep_prob:
                 0.5})
            minibatch_cost = sess.run(cost, feed_dict = {x: batch_x, y: batch_y
                , keep_prob: 1.0})

            total_epoch_cost += minibatch_cost
            i += batch_size

        avg_epoch_cost = total_epoch_cost/n_batches

        print(’Epoch’, epoch+1, ’completed out of’, n_epochs, ’cost:’, avg_
            epoch_cost)

        acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y , keep_prob:
            1.0})*100

        if acc < prev_acc:
            prev_acc = acc
            print(’Test accuracy: ’ + str(acc) + ’ %’)
            print(’Possible overfitting!!’)
            #ans = raw_input(’Continue?(y/n): ’)
            #if ans == ’n’:
            #    break
        elif acc > 99.2:
            print(’Test accuracy: ’ + str(acc) + ’%’)
            ans = raw_input(’High accuracy... Continue?(y/n): ’)
            if ans == ’n’:
                break
        else:
           print(’Test accuracy: ’ + str(acc) + ’%’)
           prev_acc = acc

    acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y , keep_prob: 1.0})
        *100
    print(’Accuracy: ’ + str(acc) + ’ %’)
```

Anexo 7

```
import tensorflow as tf
from images_and_labels import load_images_and_labels_arrays
from images_and_labels import create_images_and_labels
import cv2 as cv
import numpy as np
import os
import time

#En este codigo nos encontramos con una Convolutional Neural Network para
#entrenarla con nuestro propio dataset
```

```
def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una fully-connected layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0.1), trainable = True, name = 'b')

    output = tf.nn.leaky_relu( tf.nn.bias_add( tf.matmul( input , weights ) ,
        biases ))
    return output

def conv2d(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una convolutional layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.truncated_
        normal_initializer(stddev=0.1), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0.1), trainable = True, name = 'b')

    conv_out = tf.nn.conv2d(input , weights , strides = [1,1,1,1], padding = '
        SAME')

    output = tf.nn.leaky_relu( tf.nn.bias_add( conv_out , biases ))
    return output

def max_pool(input):
    #En esta funcion se define el grafo de una max pooling layer

    output = tf.nn.max_pool(input , ksize = [1,2,2,1] , strides = [1,2,2,1] ,
        padding = 'SAME')
    return output

def neural_network_model(data , keep_prob):

    #Los datos de entrada (imagenes) las volvemos a transformar en imagenes con
    #[width,height,channels], ya que en un inicio todas las imagenes estan
    #dadas como arrays
    data = tf.reshape(data, shape=[-1,112,112,3])

    #En esta funcion estamos definiendo el grafo de la red neuronal

    with tf.variable_scope('layer_11'):
        #La convolucion poseera 32 filtros de 5x5 que se aplican a los 3
        #canales (RGB)
        conv_11 = conv2d(data , [5,5,3,32] , [32])
    with tf.variable_scope('layer_12'):
        conv_12 = conv2d(conv_11, [5,5,32,16],[16])
        pool_1 = max_pool(conv_12)

    with tf.variable_scope('layer_21'):
        #La convolucion poseera 32 filtros de 5x5 que se aplican a los 16
        #feature maps formados por la convolucion anterior
        conv_21 = conv2d(pool_1 , [5,5,16,32] , [32])
    with tf.variable_scope('layer_22'):
        conv_22 = conv2d(conv_21, [5,5,32,16],[16])
        pool_2 = max_pool(conv_22)
```

```
    with tf.variable_scope('layer_31'):
        conv_31 = conv2d(pool_2 , [5,5,16,32] , [32])
    with tf.variable_scope('layer_32'):
        conv_32 = conv2d(conv_31, [5,5,32,16],[16])
        pool_3 = max_pool(conv_32)

    with tf.variable_scope('layer_41'):
        conv_41 = conv2d(pool_3 , [3,3,16,32] , [32])
    with tf.variable_scope('layer_42'):
        conv_42 = conv2d(conv_41, [5,5,32,16],[16])
        pool_4 = max_pool(conv_42)

    with tf.variable_scope('layer_fc1'):
        #Los feature maps anteriores se aplanan para formar 1 array de datos
        #formados por los resultados de los features maps y asi formar una red
        #FeedForward fully connected con dichos datos
        pool_flattened = tf.reshape(pool_4 , [-1,7*7*16])

        fc1_out = layer(pool_flattened , [7*7*16 , 512] , [512])

    with tf.variable_scope('layer_fc2'):
        fc2_out = layer(fc1_out , [512 , 256] , [256])

        #Para reducir Overfitting se aplica el proceso de dropout. Para ello se
        #usa un placeholder y asi podremos activar dropout en entrenamiento y
        #desactivar en testeo
        fc_drop = tf.nn.dropout(fc2_out , keep_prob)

    with tf.variable_scope('output'):
        output = layer(fc_drop , [256 , n_classes] , [n_classes])

    return output

def train_model(cost):

    #Learning rate por defecto = 0.001
    optimizer = tf.train.AdamOptimizer()

    #La funcion minimize combina las funciones: compute_gradients y
    #apply_gradients
    train_operation = optimizer.minimize(cost)

    return train_operation

def loss(prediction, y):

    #Esta funcion ya realiza el softmax sobre los logits, por lo que espera
    #logits que no esten escalados. Si usas la salida con softmax, esto no dara
    #buenos resultados. Esta funcion calcula el error con la Cross Entropy loss
    #function para cada salida de la red neuronal
    entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits = prediction,
        labels = y)

    #A partir del error en las salidas, calculamos un valor unico haciendo la
    #media del error
    cost = tf.reduce_mean(entropy)
```

```
    return cost

def eval_test(prediction, y):

    #Se comprueba que la salida corresponde con la deseada
    correct = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))

    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    return accuracy



#Cargamos los dataset de entrenamiento y de testeo:
train_x , labels , train_y = load_images_and_labels_arrays('Datasets/Train_
    Dataset')

size_images = np.shape(train_x)[1]
n_train_examples = np.shape(train_x)[0]
print('Loaded ' + str(n_train_examples) + ' Training Images')

test_x, labels, test_y = load_images_and_labels_arrays('Datasets/Test_Dataset')
print('Loaded ' + str(np.shape(test_x)[0]) + ' Test Images')

batch_size = 100
n_batches = n_train_examples/batch_size
n_epochs = 20
#One-hot encoding
n_classes = 3

#Las imagenes no estan dadas como una matriz (height x width), sino como un
#vector. Debido a ello, tendremos que el shape de x (entrada de la red), no
#tendra altura determinada (ejemplos del batch), y el ancho sera de
#size_images, que en nuestro caso es 112x112x3 (RGB)
x = tf.placeholder(tf.float32, shape=[None, size_images])
y = tf.placeholder(tf.float32, shape=[None, n_classes])
keep_prob = tf.placeholder(tf.float32, shape = None)

#Creamos el subgrafo de la red neuronal
prediction = neural_network_model(x,keep_prob)

#Creamos el subgrafo que calcule el error de la red neuronal
cost = loss(prediction, y)

#Creamos el subgrafo que entrene la red neuronal
train_op = train_model(cost)

#Creamos el subgrafo que calcule el porcentaje de aciertos sobre las imagenes
#de testeo
accuracy = eval_test(prediction,y)

#Iniciamos sesion para poder ejecutar todo nuestro modelo y poder entrenarlo
with tf.Session() as sess:

    #Iniciamos todas las variables como weights y biases
    sess.run(tf.global_variables_initializer())
```

```
prev_acc = 0
#Bucle para completar todos los epochs
for epoch in range(n_epochs):
    total_epoch_cost = 0
    avg_epoch_cost = 0

    i=0

    #Bucle para los minibatches
    while(i < n_train_examples):
        start = i
        end = i + batch_size

        batch_x = train_x[start:end][:]
        batch_y = train_y[start:end][:]

        sess.run(train_op, feed_dict = {x: batch_x , y: batch_y , keep_prob:
             0.5})
        minibatch_cost = sess.run(cost, feed_dict = {x: batch_x, y: batch_y
            , keep_prob: 1.0})

        total_epoch_cost += minibatch_cost
        i += batch_size

    avg_epoch_cost = total_epoch_cost/n_batches

    print('Epoch', epoch+1, 'completed out of', n_epochs, 'cost:', avg_
        epoch_cost)

    acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y , keep_prob:
        1.0})*100

    if acc < prev_acc:
        prev_acc = acc
        print('Test accuracy: ' + str(acc) + ' %')
        print('Possible overfitting!!')
        #ans = raw_input('Continue?(y/n): ')
        #if ans == 'n':
        #    break
    elif acc > 95:
        print('Test accuracy: ' + str(acc) + '%')
        ans = raw_input('High accuracy... Continue?(y/n): ')
        if ans == 'n':
            break
    else:
       print('Test accuracy: ' + str(acc) + '%')
       prev_acc = acc

acc = sess.run(accuracy, feed_dict = {x:test_x, y:test_y , keep_prob: 1.0})
    *100
print('Accuracy: ' + str(acc) + ' %')

'''
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
         Evaluamos cualquier imagen de la camara
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''
```

```
    cap = cv.VideoCapture(0)

    while(True):

      #Capturamos la imagenes frame por frame a cada iteracion del bucle
      ret, frame = cap.read()
      cv.imshow('window', frame)

      #Convierto las imagenes a RGB ya que openCV lee en BGR:
      frame = cv.cvtColor(frame, cv.COLOR_BGR2RGB)
      key = cv.waitKey(1)

      if key & 0xFF == ord('q'): break

      img = tf.reshape(tf.cast(tf.image.resize_images(frame, [112,112]), tf.
          uint8), shape=[-1,112*112*3]).eval()

      output = sess.run(prediction, feed_dict={x: img , keep_prob: 1.0})

      pred = tf.argmax(output,1).eval()

      if pred == 0:
          out = 'pepino'
      elif pred == 1:
          out='limon'
      else:
          out = 'tomate'

      print(out)

      time.sleep(0.5)

    cap.release()
    cv.destroyAllWindows()
```

Anexo 8

```
import cv2 as cv
import numpy as np
import time
import RPi.GPIO as gpio
import socket
import os
import sys

'''
Este script permite controlar el robot con la RPI desde el PC (script para RPI)
'''

#Argumentos script -> maximo 3 argumentos con numeros del 0 al 2 (3 camaras).
#Con los argumentos se indica que camaras van a grabar imagenes (no se
#recomienda grabar con 3 camaras al mismo tiempo)
n_cameras = len(sys.argv)-1
```

```
#Lado del cliente
TCP_IP = '192.168.1.38'
TCP_PORT = 6666
BUFFER_SIZE = 20 # Normalmente 1024, pero queremos una respuesta rapida

#Creamos socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP, TCP_PORT))
s.setblocking(0)
print('Conectado al servidor!')

#Asignamos layout de pines por defecto de rpi
gpio.setmode(gpio.BOARD)

#Ponemos los pines como salida
gpio.setup(11, gpio.OUT) #Ruedas izquierda
gpio.setup(13, gpio.OUT) #Ruedas derecha
#Activamos dichos pines como pwm con frecuencia de 50Hz
pwm_izq = gpio.PWM(11,50)
pwm_der = gpio.PWM(13,50)
#Iniciamos los PWM con un duty cycle de 0%
pwm_izq.start(0)
pwm_der.start(0)

n = 0
save = 0
ch = 0
v = 50
off = 20

c = 0 #Camaras activas

if n_cameras == 3:
  print('Cuidado!! Se van a usar 3 camaras! No recomendable!')
else:
  print('Se van a utilizar ' + str(n_cameras) + ' camara(s)!')
  cap = []
  for i in range(n_cameras):
    cap.append(cv.VideoCapture(int(sys.argv[i+1])))

print('Inicializado Completo!')

while True:

    if (save == 1 and n_cameras == 2):
    ret1, img1 = cap[0].read()
    ret2, img2 = cap[1].read()

    cv.imwrite(os.getcwd() + '/Images/Camera1/Image' + str(n) + '.jpeg', img1)
    cv.imwrite(os.getcwd() + '/Images/Camera2/Image' + str(n) + '.jpeg', img2)

    time.sleep(0.2)
    n+=1

    elif (save == 1 and n_cameras == 1):
    ret, img = cap[0].read()
```

```
  cv.imwrite(os.getcwd() + '/Images/Camera3/Image' + str(n) + '.jpeg', img)

  time.sleep(0.2)
  n+=1

  elif (save == 1 and n_cameras == 3):
  # Capture frame-by-frame
  if c == 1:
    cap3.release()
    time.sleep(0.1)

  cap1 = cv.VideoCapture(0)
  ret1, img1 = cap1.read()

  if c == 1:
    cap1.release()
    time.sleep(0.1)

  cap2 = cv.VideoCapture(1)
  ret2, img2 = cap2.read()

  if c == 0:
    c = 1

  cap2.release()
  time.sleep(0.1)
  cap3 = cv.VideoCapture(2)
  ret3, img3 = cap3.read()

  #print('Imagenes tomadas: ', n)
  cv.imwrite(os.getcwd() + '/Images/Camera1/Image' + str(n) + '.jpeg', img1)
  cv.imwrite(os.getcwd() + '/Images/Camera2/Image' + str(n) + '.jpeg', img2)
  cv.imwrite(os.getcwd() + '/Images/Camera3/Image' + str(n) + '.jpeg', img3)

  n+=1

#Recibimos la informacion del servidor
  try:
  ch = s.recv(BUFFER_SIZE)
except:
  ch=0

  if ch == 'q':
  break
  elif ch == 'w':
  pwm_izq.ChangeDutyCycle(v)
  pwm_der.ChangeDutyCycle(v)
  #print('Avanza')
  elif ch == 'z':
  v-=5
  if v<0: v = 0
  pwm_izq.ChangeDutyCycle(v)
  pwm_der.ChangeDutyCycle(v)
  #print('Avanza mas lento',v)
  elif ch == 'm':
  pwm_izq.ChangeDutyCycle(32)
  pwm_der.ChangeDutyCycle(32+off)
```

```
    #print('Minimo de velocidad')
    elif ch == 'x':
    v+=5
    if v>100: v = 100
    pwm_izq.ChangeDutyCycle(v)
    pwm_der.ChangeDutyCycle(v)
    #print('Avanza mas rapido',v)
    elif ch == 's':
    pwm_izq.ChangeDutyCycle(0)
    pwm_der.ChangeDutyCycle(0)
    #print('Para')
    elif ch == 'd':
    pwm_izq.ChangeDutyCycle(100)
    pwm_der.ChangeDutyCycle(10)
    #print('Derecha')
    elif ch == 'a':
    pwm_izq.ChangeDutyCycle(10)
    pwm_der.ChangeDutyCycle(100)
    #print('Izquierda')
    elif ch == 'i':
    if save == 1:
      save = 0
      print('Parando de grabar...')
    else:
      save = 1
      print('Grabando Imagenes...')

if 'cap1' in locals(): cap1.release()
if 'cap2' in locals(): cap2.release()
if 'cap3' in locals(): cap3.release()
if 'cap[0]' in locals(): cap[0].release()
if 'cap[1]' in locals(): cap[1].release()
s.close()
pwm_izq.stop()
pwm_der.stop()
gpio.cleanup()
print('Done!')
```

```
import socket
import os
import numpy
import sys, termios, tty, os, time

'''
Este script permite conectarse a la RPI para controlar el robot(script para PC)
'''

#Funcion que detecta pulsaciones de teclado
def getch():
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    try:
        tty.setraw(sys.stdin.fileno())
        ch = sys.stdin.read(1)

    finally:
```

```
        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    return ch

#Este script crea el servidor para que el coche se conecte y puedan comunicarse
TCP_PORT = 6666
BUFFER_SIZE = 20 #Normalmente 1024, pero queremos respuesta rapida

#Creamos el socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#Asociamos la ip de esta maquina con el puerto que queremos abrir
s.bind(('192.168.1.38', TCP_PORT))

#Escuchamos hasta que el cliente se conecte
s.listen(1)
conn, addr = s.accept()
print('Raspberry Pi conectada con IP', addr)

save = 0

while True:
    char = getch()

    if char == 'w':
        conn.send('w')
    elif char == 'z':
        conn.send('z')
    elif char == 'x':
        conn.send('x')
    elif char == 'm':
        conn.send('m')
    elif char == 'd':
        conn.send('d')
    elif char == 's':
        conn.send('s')
    elif char == 'a':
        conn.send('a')
    elif char == 'i':
        conn.send('i')
        if save == 0:
            print('Grabando Imagenes...')
            save = 1
        else:
            print('Parando de grabar...')
            save = 0
    elif char == 'q':
        conn.send('q')
        break

conn.close()
s.close()
print('Done!')
```

Anexo 9

```
import cv2 as cv
import numpy as np
import os
import socket
from Queue import Queue
from threading import Thread
import RPi.GPIO as gpio

#Este script toma imagenes de la camara conectada a la RPI y mueve el robot
#segun las salidas de la red neuronal mandadas por el PC (script para RPI)

#Creacion de una clase para generar un hilo que tome imagenes de la camara
class WebcamVideoStream:
    def __init__(self, src=0):
        #Se inicializa la webcam y se lee la primera imagen
        self.stream = cv.VideoCapture(src)
        (self.grabbed, self.frame) = self.stream.read()

        # Se inicializa una variable para indicar si se debe parar el hilo
        self.stopped = False

    def start(self):
        #Comienza el hilo para leer y decodificar las imagenes de la camara
        Thread(target=self.update, args=()).start()
        return self

    def update(self):
        #Hasta que no se pare el hilo, se toman imagenes
        while True:
            #Dejamos de iterar cuando la variable de parar se active
            if self.stopped:
                return

            #Sino, seguimos leyendo de la camara
            (self.grabbed, self.frame) = self.stream.read()

    def read(self):
        # Devuelve la imagen leida
        return self.frame

    def stop(self):
        #Activa la variable de parar el hilo
        self.stopped = True



#Lado del cliente
TCP_IP = '10.42.0.1'
TCP_PORT = 6666
BUFFER_SIZE = 20 #Normalmente 1024, pero queremos una respuesta rapida

#Creamos socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
s.connect((TCP_IP, TCP_PORT))
print('Conectado al servidor!')

#Asignamos layout de pines por defecto de rpi
gpio.setmode(gpio.BOARD)
#Ponemos los pines como salida
gpio.setup(11, gpio.OUT) #Ruedas izquierda
gpio.setup(13, gpio.OUT) #Ruedas derecha
#Activamos dichos pines como pwm con frecuencia de 50Hz
pwm_izq = gpio.PWM(11,50)
pwm_der = gpio.PWM(13,50)
#Iniciamos los PWM con un duty cycle de 0%
pwm_izq.start(0)
pwm_der.start(0)

#Capturamos las imagenes de la camara
vs = WebcamVideoStream(src=0).start()
frame = vs.read()
#Guardamos la imagen para recuperarla desde el PC
cv.imwrite(os.getcwd() + '/Live_image/img.jpeg',frame)

while(True):

  #Recibimos la informacion del servidor
  pred = s.recv(BUFFER_SIZE)

  if pred == '0':
      #Capturamos imagen cuando el PC lo indique
      frame = vs.read()
      cv.imwrite(os.getcwd() + '/Live_image/img.jpeg',frame)
  elif pred == '1':
      pwm_izq.ChangeDutyCycle(45)
      pwm_der.ChangeDutyCycle(45)
      #print('Frente')
  elif pred == '2':
      pwm_izq.ChangeDutyCycle(75)
      pwm_der.ChangeDutyCycle(20)
      #print('Obs. a izquierda')
  elif pred == '3':
      pwm_izq.ChangeDutyCycle(20)
      pwm_der.ChangeDutyCycle(75)
      #print('Obs. a derecha')
  elif pred == '5':
      break

s.close()
vs.stop()
pwm_izq.stop()
pwm_der.stop()
gpio.cleanup()
print('Done!')
```

```
import tensorflow as tf
from images_and_labels import load_images_and_labels_arrays
from images_and_labels import create_images_and_labels_grayscale
import cv2 as cv
```

```
import numpy as np
import os
import socket
import cPickle
import signal

#Este script coge las imagenes tomadas por la RPI y las procesa a la red
#neuronal para devolver una orden de movimiento a la RPI y controlar el robot
#(script para el PC)

def layer(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una fully-connected layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.contrib.
        layers.xavier_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0), trainable = True, name = 'b')

    output = tf.nn.leaky_relu( tf.nn.bias_add( tf.matmul( input , weights ) ,
        biases ))
    return output

def conv2d(input, weight_shape, bias_shape):
    #En esta funcion se define el grafo de una convolutional layer
    weights = tf.get_variable(shape = weight_shape, initializer = tf.contrib.
        layers.xavier_initializer(), trainable = True, name = 'w')
    biases = tf.get_variable(shape = bias_shape, initializer = tf.constant_
        initializer(value=0), trainable = True, name = 'b')

    conv_out = tf.nn.conv2d(input , weights , strides = [1,1,1,1], padding = '
        SAME')

    output = tf.nn.leaky_relu( tf.nn.bias_add( conv_out , biases ))
    return output

def max_pool(input):
    #En esta funcion se define el grafo de una max pooling layer

    output = tf.nn.max_pool(input , ksize = [1,2,2,1] , strides = [1,2,2,1] ,
        padding = 'SAME')
    return output

def neural_network_model(data , keep_prob):

    #Los datos de entrada (imagenes) las volvemos a transformar en imagenes con
    #[width,height,channels], ya que en un inicio todas las imagenes estan
    #dadas como arrays
    data = tf.reshape(data, shape=[-1,112,112,1])

    #En esta funcion estamos definiendo el grafo de la red neuronal

    with tf.variable_scope('layer_1'):
        #La convolucion poseera 64 filtros de 5x5 que se aplican a un unico
        #canal
        conv_1 = conv2d(data , [5,5,1,64] , [64])
        pool_1 = max_pool(conv_1)

    with tf.variable_scope('layer_2'):
```

```
        #La convolucion poseera 64 filtros de 5x5 que se aplican a los 64
        #feature maps formados por la convolucion anterior
        conv_2 = conv2d(pool_1 , [5,5,64,64] , [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope('layer_3'):
        conv_3 = conv2d(pool_2 , [5,5,64,64] , [64])
        pool_3 = max_pool(conv_3)

    with tf.variable_scope('layer_4'):
        conv_4 = conv2d(pool_3 , [3,3,64,64] , [64])
        pool_4 = max_pool(conv_4)

    with tf.variable_scope('layer_fc'):
        #Los feature maps anteriores se aplanan para formar 1 array de datos
        #formados por los resultados de los features maps y asi formar una red
        #FeedForward fully connected con dichos datos
        pool_flattened = tf.reshape(pool_4 , [-1,7*7*64])

        fc_out = layer(pool_flattened , [7*7*64 , 512] , [512])

        #Para reducir Overfitting se aplica el proceso de dropout. Para ello se
        #usa un placeholder y asi podremos activar dropout en entrenamiento y
        #desactivar en testeo
        fc_drop = tf.nn.dropout(fc_out , keep_prob)

    with tf.variable_scope('output'):
        output = layer(fc_drop , [512 , 3] , [3])

    return output

TCP_PORT = 6666
BUFFER_SIZE = 20

#Creamos el socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#Asociamos la ip de esta maquina con el puerto que queremos abrir
s.bind(('10.42.0.1', TCP_PORT))

#Escuchamos hasta que el cliente se conecte
print('Esperando conexion del cliente...')
s.listen(1)
conn, addr = s.accept()
print('Raspberry Pi conectada con IP', addr)


#Las imagenes no estan dadas como una matriz (height x width), sino como un
#vector. Debido a ello, tendremos que el shape de x (entrada de la red), no
#tendra altura determinada (ejemplos del batch), y el ancho sera de
#size_images, que en nuestro caso es 112x112x1
x = tf.placeholder(tf.float32, shape=[None, 112*112*1])
keep_prob = tf.placeholder(tf.float32, shape = None)

#Creamos el subgrafo de la red neuronal
output = neural_network_model(x,keep_prob)
```

```
#Creamos la operacion para calcular que movimiento hacer
prediction = tf.argmax(output,1)

#Creamos las operaciones necesarias para guardar el modelo entrenado
saver = tf.train.Saver()

with tf.Session() as sess:

  #Cargamos el modelo entrenado
  saver.restore(sess, os.getcwd() + '/Trained_models/Convnet_model')
  print("Model restored")

  '''
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            Evaluamos cualquier imagen de la camara
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  '''
  try:
      while True:
              #ssh-keygen
              #ssh-copy-id pi@raspberrypi.local
              #Utilizamos scp para transmitir la imagen de la rpi al PC
              os.system('scp pi@raspberrypi.local:~/Desktop/Live_image/img.jpeg
                  ~/')

              #Transformamos las imagenes para que pueda computarlas la red
              #neuronal
              img = tf.reshape(tf.cast(tf.image.resize_images(tf.image.rgb_to_
                  grayscale(tf.image.decode_jpeg(tf.read_file('/home/victor/img.
                  jpeg'), channels=3)), [112,112]), tf.uint8), shape
                  =[-1,112*112]).eval()

              pred = sess.run(prediction, feed_dict={x: img , keep_prob: 1.0})

              conn.send('0') #Recibir nueva imagen de la rpi

              if pred == 0:
                  conn.send('1')
                  print('Frente')
              elif pred == 1:
                  conn.send('2')
                  print('Obs. a izquierda')
              else:
                  conn.send('3')
                  print('Obs. a derecha')
  except KeyboardInterrupt:
      conn.send('5')
      conn.close()
      s.close()
      print('Done!')
```