# Título del Proyecto:

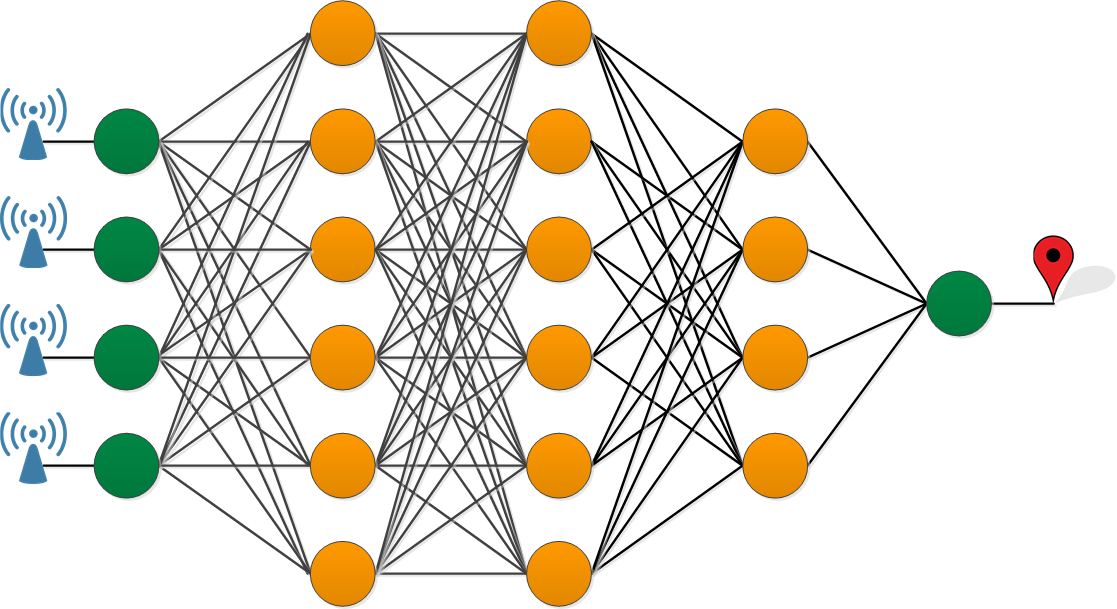
**WiFlow**

# Autor:

Fernández Ippólito, Juan Carlos

# Director:

Dr. Tomás Gironés, Jesús



WiFlow

**TESINA PARA LA**

**OBTENCIÓN DEL TÍTULO DE:**

# Máster en Desarrollo de Aplicaciones Android

**Septiembre de 2018**

Contenido

[Introducción 3](#_Toc421525768)

[Descripción del problema 3](#_Toc421525769)

[Objetivos 3](#_Toc421525770)

[Motivación 3](#_Toc421525771)

[Situación de… / Tecnologías utilizadas 3](#_Toc421525772)

[Arquitectura de la aplicación 3](#_Toc421525773)

[Esquema del diseño 3](#_Toc421525774)

[Modelo de datos 3](#_Toc421525775)

[Vistas 3](#_Toc421525776)

[Conclusiones 3](#_Toc421525777)

[Anexo fuentes 4](#_Toc421525778)

[Listado de fuentes entregadas o enlace a GitHub 4](#_Toc421525779)

[Manual de usuario 4](#_Toc421525780)

# Introducción

## Descripción del problema

Antecedentes y descripción de la problemática a resolver.

La localización derivada de señales satelitales puede volverse muy dificultosa o incluso imposible dentro de una edificación, ya sea un museo, una universidad, etcétera. Esto es debido a la atenuación que supone a esas señales, de por sí ya muy pequeñas, los materiales utilizados en la construcción. Esto es especialmente cierto si se desea discernir pocos metros y la exactitud de la posición empeora debido al efecto de esas atenuaciones.

## Objetivos

El objetivo es encontrar una solución tecnológica que pueda dar respuesta a la pregunta ¿dónde estamos?, en el contexto descripto anteriormente: localización en interiores.

## Motivación

Como proyecto final de la asignatura Android Fundamentos, dictada por la UPV a fin del año 2015, realicé un trabajo denominado WiFi Locus[[1]](#endnote-1). Este proyecto consistía en una aplicación para localización en interiores. Mediante las potencias emitidas por los puntos de acceso WiFi que llegaban al dispositivo móvil, registrados en una en una etapa inicial de entrenamiento haciendo uso de la aplicación en distintos lugares del edificio, intentaba luego calcular la posición en que se encontraba el dispositivo móvil, utilizando fundamentalmente un método de mínima diferencia de cuadrados más algunos otros algoritmos de ajuste, descriptos en la memoria de ese proyecto[[2]](#endnote-2).

¿Sería posible obtener una funcionalidad similar utilizando técnicas de inteligencia artificial, como lo son las redes neuronales con sistemas de aprendizaje automático?

Un desafío interesante, especialmente estando disponible TensorFlow, con la posibilidad entonces de utilizarlo en Android.

# Tecnologías utilizadas

Incluir la información necesaria que facilite la comprensión del proyecto. Solo se incluirá este punto para describir alguna tecnología, paradigma de diseño, plataforma, … no descrita durante el Diploma o Master. Su extensión será siempre reducida.

**Redes neuronales de aprendizaje automático**

Los sistemas de AA (**A**prendizaje **A**utomático) supervisado, aprenden cómo combinar entradas para producir predicciones útiles sobre datos nunca antes vistos, mediante una serie de ejemplos que se les suministra durante la etapa de entrenamiento.

**Tensorflow**

TensorFlow es una biblioteca de código abierto para aprendizaje automático desarrollada por Google, mediante la cual es posible construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones. Fue liberado como software de código abierto en noviembre del 2015.

El nombre TensorFlow deriva de las operaciones qué tales redes neuronales realizan sobre arrays multidimensionales de datos. Estos arrays multidimensionales son referidos como "tensores"

**T**ensor**F**low proporciona una API de Python, así como APIs de C++ y otros lenguajes. La enorme cantidad de ejemplos y documentación de TensorFlow disponible en Python puede hacer que tal vez el lector comience utilizando este lenguaje para familiarizarse con **TF**, aunque Python parezca inicialmente poco amigable. Al menos ese fue mi caso.

**Keras**

La API de alto nivel de Keras, proporciona bloques de construcción para crear, entrenar y utilizar en predicciones a modelos de AA.

Tiene una interfaz simple optimizada para casos de uso común. Proporciona comentarios bastante claros los errores en la escritura del código y en la ejecución.

Los modelos de las redes en Keras se hacen conectando bloques de construcción configurables, con pocas restricciones.

TensorFlow tiene embebida una implementación de la especificación de la API de Keras, de manera que instalando **TF**, Keras ya está disponible.

Trabajar con Keras, en lugar de hacerlo solo con TensorFlow, hace la curva de aprendizaje del trabajo con redes neuronales menos traumática.

Como pasa en muchos otros paradigmas, al aumentar el nivel y ganar en simplicidad, el costo que hay que pagar es alguna pérdida de versatilidad. En particular en este trabajo, la dificultad más grande a superar fue que el formato modelo de red entrenado usando Keras, no puede ser utilizado directamente en Android y el proceso de obtener uno compatible con ese sistema operativo, fue casi artesanal.

**Python**

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Otros paradigmas están soportados mediante el uso de extensiones. Es un lenguaje interpretado, usa tipado dinámico (no se comprueban los tipos a medida que se van haciendo los cálculos) y es multiplataforma.

Es muy usado en áreas de ciencia e ingeniería. Tanto en forma nativa como por medio de bibliotecas importadas posee gran potencia para el manejo de datos y cálculos numéricos.

# Arquitectura de la aplicación

NOTA: Los siguientes puntos son orientativos

La idea general de proyecto es colectar cuidadosamente los datos de potencia e identificación (BSSID, es decir el MAC address al que se conecta) de los puntos de acceso que son detectados en un dispositivo móvil, a lo largo y ancho de distintos espacios dentro de una edificación, tomando nota de la ubicación en que se obtuvo cada medición.

A medida que nos desplacemos en cada recinto, el dispositivo móvil registrará distintas potencias e incluso pueden cambiar la constelación de puntos de accesos recibidos: es posible que algunos se pierdan y también que aparezcan otros nuevos.

Estos datos serán utilizados para entrenar y validar el funcionamiento del modelo de la red neuronal.

Las potencias y las BSSIDs constituirán las características utilizadas como entrada en el entrenamiento de la red, mientras que las posiciones correspondientes (una para cada conjunto de potencias y BSSIDs obtenidas en un punto determinado de cada recinto, oficina o espacio) serán las etiquetas que se utilizarán para ajustar los pesos que conectan las neuronas, distribuidas en distintas capas.

## Componentes y herramientas utilizadas

Wifi Locus modificado: https://github.com/jon1721/WiFiLocusModificado.git

JADE Javascript based Database Editor: es un plugin para Chrome, disponible en Chrome Web Store, que permite editar y observar base de datos SQLite: https://chrome.google.com/webstore/detail/javascript-based-database/bponbdjkefbmgkfiiphhabghkkfocook

RebaseData: conversor en línea de SQLite a CSV: https://www.rebasedata.com/convert-sqlite-to-csv-online

Microsoft Excel

PyCharm: IDE para Python desarrollado por JetBrains. Tiene enormes ventajas para quienes han usado Android Studio, pues la GUI y su uso son muy similares. La búsqueda, descarga e instalación de paquetes está muy simplificada. La versión libre Community es totalmente funcional: https://www.jetbrains.com/pycharm/

TensorFlow: https://www.tensorflow.org/

Keras: https://keras.io/

Tensorboard: https://www.tensorflow.org/guide/summaries\_and\_tensorboard

Paso a paso

Obtención de los datos

El área de medición fue la que se muestra en el siguiente plano:

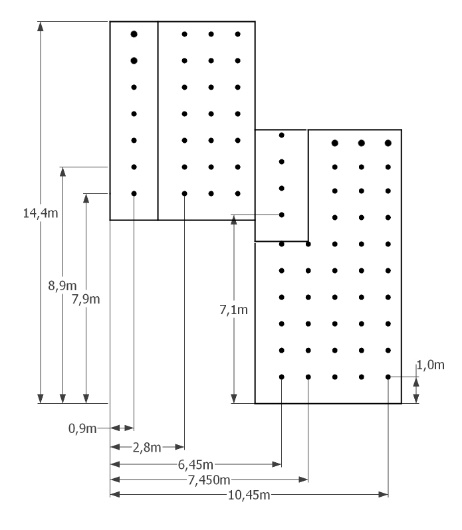


Figura 1

En el plano se observan las dimensiones en metros, el perímetro, la ubicación de las paredes y los puntos en los que realizaron las mediciones, representados por pequeños círculos negros.

La que sigue es una captura de pantalla de la aplicación WiFi Locus modificada[[3]](#endnote-3) que se utilizó para la captura de los datos:

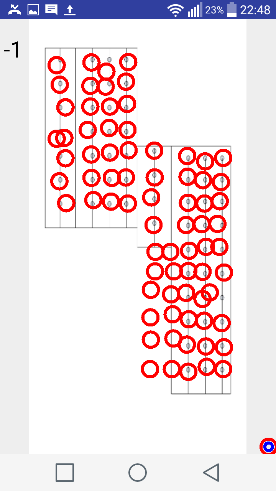


Figura 2

Para realizar cada medición, hay que situarse físicamente en el punto a medir. En esa condición, pulsar la pantalla. Eso dispara unas mediciones sucesivas, cuya cantidad debe haber sido configurada previamente en los ajustes de la aplicación, almacenándose los datos en una base de datos SQLite en la memoria no volátil del dispositivo. El punto medido en cuestión queda indicado en la pantalla mediante un círculo rojo, sumando esa indicación cada vez que se realiza una nueva medición.

En total se tomaron mediciones válidas en 74 puntos, 10 mediciones en cada punto, cada una indicada mediante su ID de registro en la base de datos, el cual con cada medición se mostraba en el ángulo superior izquierdo de la pantalla, con el objetivo de llevar un registro externo del proceso.

La siguiente imagen muestra el registro manual del ID de registro inicial de cada punto:

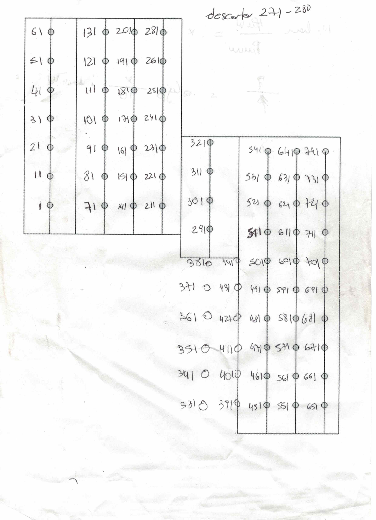


Figura 3

De esta manera, la aplicación generó el archivo Puntoswifi.db[[4]](#endnote-4), en formato SQLite.

En la imagen siguiente se observa una vista de Puntoswifi.db obtenida con la aplicación Jade:

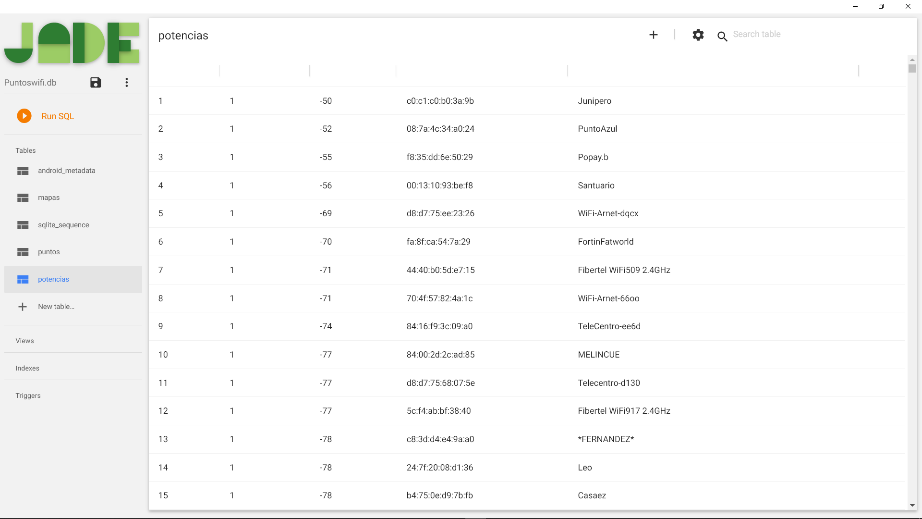


Figura 4

Aunque Python puede manejar archivos en formato SQlite, preferí exportarlo a CSV, para poder ver y analizar los resultados en Excel. Utilicé entonces RebaseData [[5]](#endnote-5)para exportar Puntoswifi.db a CSV.

De los archivos obtenidos en este proceso los importante son potencias.csv y puntos.csv, pues contienen la identificación del punto (su posición dentro del edificio) y la identificación de los puntos de acceso WiFi con el nivel de potencia medido para cada uno de ellos en ese lugar.

A partir de ahora, el proceso de analizar, filtrar y acomodar los datos para su procesamiento posterior en la red neuronal, fue llevado a cabo en el archivo wiflow.xlsx4

En ese archivo Excel, la hoja Puntos y la hoja Mediciones son el resultado de importar los archivos CSV mencionados.

Como ejemplo de lo dicho, en la hoja Puntos vemos los resultados de 3 mediciones, en relación a las coordenadas de los puntos de ellas en el mapa en pantalla. Las coordenadas de pantalla x e y coinciden, pues son 3 mediciones tomadas en el mismo lugar. De hecho, cada vez que se pulsaba la pantalla la aplicación fue configurada para tomar 10 mediciones sucesivas.



Figura 5

En la hoja de cálculo se agregaron además las coordenadas físicas en metros del punto en el edificio, para tener una mejor idea si el resultado de la predicción posterior coincidía con la realidad y si su error era significativo.



Figura 6

Simultáneamente y en el mismo proceso del registro anterior, se almacenaron los valores de potencia e identificación de los puntos de acceso WiFI que eran obtenidos por el móvil:



Figura 7

En la tabla justo arriba de este texto solo se ven 8 entradas para el id de medición mediciónID con valor 1, pero si el lector observa la hoja Mediciones del archivoWiFlow.xlsx, podrá ver que en esa tabla para el medicionId igual a 1 se registraron 48 entradas, es decir que, para la primera medición en el punto de coordenadas x=0.9; y=7.9 el móvil midió y registró la potencia de 48 puntos de acceso distintos. Para el caso del medicioID igual a 2, la cantidad ya no es 48, sino 50, a pesar de que el lugar es el mismo y entre una y otra adquisición solo transcurren pocos segundos. Dos puntos de acceso “desaparecieron del radar”.

Un fenómeno casi siempre observado en lugares rodeados de puntos de acceso es que muchas de las señales de los puntos de acceso WiFi recibidas son marginales e inestables. Esto se ve claramente en el siguiente gráfico, obtenido mediante una captura de pantalla de la aplicación WiFi Analyzer[[6]](#endnote-6):

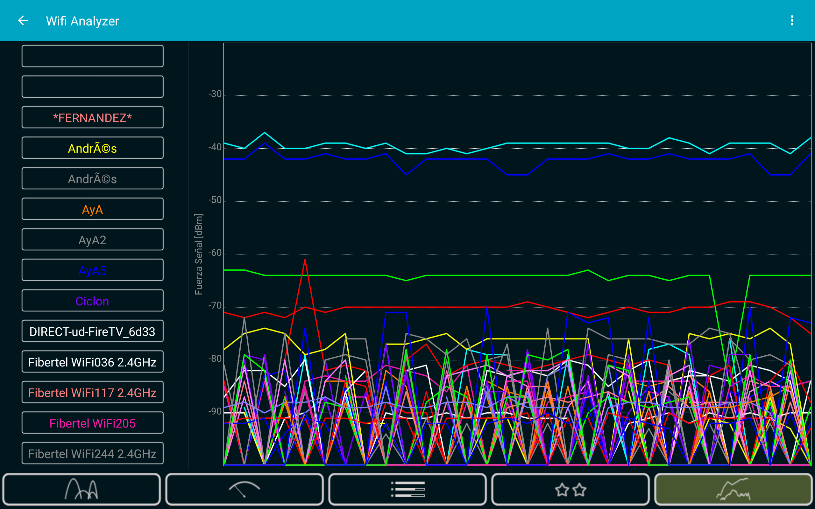


Figura 8

En el gráfico se aprecia que las señales que tiene una presencia relativamente constante están por encima de un umbral de aproximadamente -70. Aunque consideré ese umbral en este proyecto, el mismo es arbitrario y otra persona podría preferir uno un poco distinto.

Es posible que cerca de los -90 otro factor de inestabilidad sea el umbral de recepción de las primeras etapas del receptor WiFi del móvil y que además la señal se confunda con el nivel de ruido térmico o ruido de Johnson-Nyquist.

En cualquier caso y puesto que las features o características que se ingresan en la capa de entrada de la red neuronal, serán las potencias de cada punto de acceso, si consideráramos todas las señales que alguna vez aparecieron, aunque esta aparición sea esporádica, de bajo nivel e inestable, sería necesario disponer de muchas más neuronas en la capa de entrada de la red y alimentarlas con valores nulos cuando la señal no está presente. A su vez, debido a que su aparición (o no) tiene cierta característica aleatoria, sería necesario multiplicar las características de entrada en el entrenamiento de la red tantas veces como combinatorias de esta condición quieran considerarse, a los efectos de atenuar el error introducido por la ausencia absoluta de una característica que alguna vez existió.

Hecho esto con datos sintéticos, los resultados no fueron buenos y por ello preferí descartar por filtrado aquellos puntos de acceso WiFi cuyo nivel recibido fue menor que -70.

Como resultado de ese filtrado obtuve la hoja “> -71 dBm” del archivo WiFlow: en ella, observando la columna potenciaId, se aprecia que muchas mediciones fueron descartadas, pues los valores no son consecutivos (comparar la Figura 7 con la ). La tabla que sigue es una pequeña muestra del total de los resultados al aplicar el filtro.



Figura 9

Ya en esta instancia, fue posible obtener las MACs (BSSID) que aparecían, en un listado sin repetición de las mismas y asignarles un identificador de punto de acceso WiFi más cómodo (un entero consecutivo) para su procesamiento posterior en Python.



Figura 10

nota: Excel permite obtener un listado de celdas (MACs) no duplicadas en forma simple mediante Datos-Quitar duplicados

Las features o características que se ingresan en la capa de entrada de la red neuronal, serán las potencias de cada punto de acceso. Esa información será procesada también por las neuronas de la capa oculta y finalmente la red neuronal, luego del proceso de entrenamiento, terminará dando una predicción en las 2 neuronas de salida: una para la coordenada X y otra para la coordenada Y.

Es decir que habrá tantas neuronas de entrada como puntos de acceso se hayan registrado y dos neuronas de salida.

Es decir que habrá tantas neuronas de entrada como puntos de acceso y para cada medición en cada una de las estas neuronas se ingresará la potencia registrada. Una vez que la red esté entrenada, en las neuronas de salida se obtendrá la predicción que el modelo hace de la posición.

Para lograr esto, la forma en la que están ordenados ahora los datos no es la conveniente. Tal vez sería mejor tener columnas que representen a cada uno de los puntos de acceso y debajo de ellas los registros que correspondan a cada medición, con la potencia que aportó cada punto de acceso bajo la columna del mismo, en el caso de que exista una potencia, para esa medición y para el punto de acceso WiFi que indica la columna. Es dedir, algo asi:



Figura 11

En la Figura 11, para cada medición que se llevó a cabo se ven las potencias registradas para cada uno de los 20 puntos de accesos que resultaron del filtrado de superiores a -70 dB. Aquellos que no tenían indicación de potencia igual requieren un valor a ser aplicado en la neurona correspondiente de la capa de entrada de la red, por lo cual asigné arbitrariamente -100 dB, un valor suficientemente bajo y que no debe estar muy lejos de la verdad.

En el margen derecho vemos las coordenadas x e y en las que cada medición se efectuó.

EL procedimiento para llegar a este formato fue el siguiente:

Partiendo de la hoja “> -71 dBm” del archivo WiFlow.xlsx y utilizando las funciones INDICE y COINCIDIR de Excel, se creó la hoja “-71 dBm”, donde se asocia a los datos de anterior, un índice de 0..19 para cada punto de acceso y la coordenada x e y a cada registro. Una vista del área de interés de esa hoja es esta:



Figura 12

Ahora, desde el Excel, guardamos como CSV esa selección (el área de interés). De esa manera se generó el archivo martriz.csv, disponible en Github4.

Para generar a partir de matriz.scv un formato como el de la Figura 11, se procesó dicho archivo con un script Python: mtr.py, que da como resultado valores.csv, disponibles ambos en Github4. Este archivo fue importado a la hoja “tabla (python)” de WiFLow.xlsx.

Las demás hojas de WiFLow.xlsx muestran la separación de los datos en dos conjuntos, uno de entrenamiento y otro de test.

En cada lugar del plano en el que se tomaron datos, se realizaron 10 mediciones, 9 de ellas se asignaron a entrenamiento y 1 a test. Esto hace que los datos con los que se entrena la red no sean iguales a los de verificación. Un aspecto importante.

Finalmente, dado que el valor mínimo de las potencias será -100, por lo comentado anteriormente, por comodidad apliqué a transformación lineal sin efecto en el entrenamiento de sumar 100 por comodidad, para que todos los atributos sean positivos. Puede ni aplicarse esto con resultados idénticos.

A partir de las hojas train+100 y test+100 se generaron los archivos train.csv y test.csv (Github4), que se utilizarán para entrenar y testear la red neuronal.

Estos archivos tienen un contenido como este:

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,x,y

50,47,45,44,31,30,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0.9,7.9

51,48,43,43,0,31,0,36,0,0,0,0,0,0,0,0,0,0,0,0,0.9,7.9

51,48,42,46,0,0,0,33,0,0,0,0,0,0,0,0,0,0,0,0,0.9,7.9

…

La primera línea es de encabezados, con el índice de los puntos de accesos WiFi, de 0 a 19 y luego el encabezado de las coordenadas: x e y.

**La red neuronal**

La definición, entrenamiento y guardado del modelo entrenado en disco, se realiza en mediante el código escrito en el archivo wiflow.py, disponible en Github4.

En este archivo se definirá una red con una capa de entrada de 20 neuronas y una de salida de 2.

La cantidad de neuronas de entrada está dada por la cantidad de características o features que disponemos. En este caso del análisis, filtrado y procesamiento de los datos, nos quedaron 20 routers (puntos de acceso WiFi). Cada uno de ellos irá ingresando en las neuronas de la capa entrada y durante la etapa de entrenamiento los valores medidos, de a 20 por vez y uno por cada router.

La cantidad de neuronas de la capa salida se define por la cantidad de coordenadas esperadas para indicar la posición, dos en este caso: una para la coordenada x, y la segunda para la coordenada y.

Tanto la cantidad de capas ocultas como la cantidad de neuronas de cada una de esas capas, es un tema de discusión [[7]](#endnote-7).

Un criterio bastante aceptado es que, para la mayoría de los problemas, una sola capa oculta es suficiente y que el número de neuronas en esa capa es la media de las neuronas entre las capas de entrada y salida.

Aplicando ese criterio, tendría que haber utilizado solo una capa intermedia con 11 neuronas ().

Durante la experimentación con datos sintéticos, probando distintas configuraciones, encontré que una convergencia rápida y con pequeño error absoluto medio se daba con 4 capas ocultas de 32, 64, 32 y 16 neuronas respectivamente.

Es probable no sean necesarias tantas. Debido a que durante la elaboración del trabajo había aun muchos problemas sin resolver y el tiempo era limitado, experimentar con otras configuraciones es una de las líneas abiertas que le quedan a este proyecto.

Un aspecto muy importante para logran buenos valores de convergencia y error absoluto medio, es elegir la función de activación que tendrán las neuronas de cada capa.

Experimentalmente definí para las capas de entrada y salida la función de activación rectificador, o Unidades lineales rectificadas (Rectified Linear Units) “ReLU”, que es una función cuyo valor es cero para entradas negativas e igual a la entrada, para valores positivos.

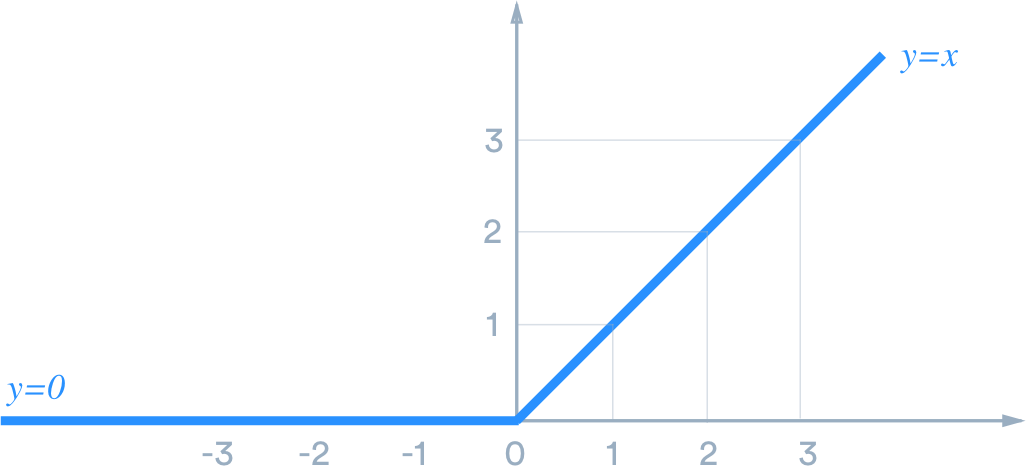


Figura 13

En tanto que, para las capas ocultas, utilicé la función de activación sigmoidea, cuya gráfica y expresión pueden verse en la Figura 14:

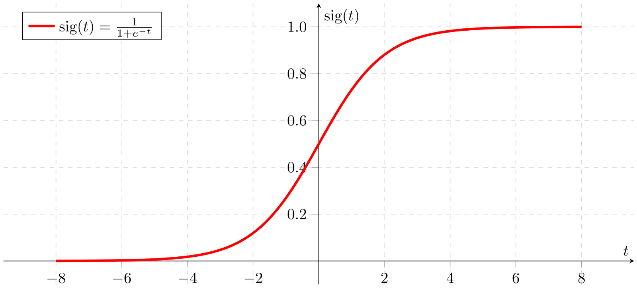


Figura 14

Detengámonos ahora un momento en el script Python wiflow.py para describir el código que lee los datos, crea la red con la arquitectura y configuración descripta, la entrena y salva el modelo entrenado.

Veamos las primeras líneas:

**import** tensorflow **as** tf  
**from** tensorflow **import** keras **as** K  
**import** numpy **as** np  
**import** pandas **as** pd  
**import** matplotlib.pyplot **as** plt

Por medio de ellas se importan los paquetes que se utilizarán y se le asigna un alias. Por ejemplo, se importa el paquete tensforflow y se le asigna el alias tf. Luego, desde tensorflow, que incluye la API Keras, en una versión compatible con la del paquete de tensorflow importado, se importa Keras y se le asigna el alias K.

Para hacer uso de los métodos disponibles en estos paquetes, podemos usar esta sintaxis:

print(tf.\_\_version\_\_)

que imprime en consola la versión de tensorflow utilizada.

Las siguientes líneas, que incluyen algunas de debug comentadas anteponiendo el carácter ‘#’, utilizando los paquetes pandas[[8]](#endnote-8) y numpy[[9]](#endnote-9) leen de disco el archivo train.csv4.

Como resultado se obtienen dos matrices que serán utilizadas por la red en su entrenamiento: atributos\_data y etiquetas\_data.

archivo = **"train.csv"**

df = pd.read\_csv(archivo)  
*# print (df)*pr = pd.DataFrame(df, columns=[**'0'**, **'1'**,**'2'**, **'3'**, **'4'**, **'5'**, **'6'**, **'7'**, **'8'**, **'9'**, **'10'**, **'11'**, **'12'**, **'13'**, **'14'**, **'15'**, **'16'**, **'17'**, **'18'**, **'19'**])  
*# print (pr)*xy = pd.DataFrame(df, columns=[**'x'**, **'y'**])  
*# print (xy)*atributos = pr[pr.columns[:20]].as\_matrix()  
*# print (atributos)*etiquetas = xy[xy.columns[:2]].as\_matrix()  
*# print(etiquetas)*atributos\_data = np.array(atributos, **"float32"**)  
etiquetas\_data = np.array(etiquetas, **"float32"**)

atributos\_data es un array que contiene los arrays de los datos de entrada, es decir, atributos\_data[0] es un array con 20 valores de potencia, atributos\_data[1] los siguientes 20 y así.

En correspondencia, a cada elemento de atributos\_data le corresponde un elemento con el mismo índice en etiquetas\_data, que es la salida ideal de la red neuronal, para esa entrada. El entrenamiento consistirá en encontrar las relaciones internas de la red neuronal, es decir los valores de pesos y bias, que den un error absoluto medio aceptable para el conjunto de datos de entrenamiento y esta consigna se continúe verificando con los datos de prueba, obtenidos estos últimos resultados con la red ya entrenada.

Para mayor detalle, copio debajo de esta línea la salida en consola para la instrucción print(atributos\_data) y para print(etiquetas\_data).

atributos\_data = [[50. 47. 45. ... 0. 0. 0.]

[51. 48. 43. ... 0. 0. 0.]

[51. 48. 42. ... 0. 0. 0.]

...

[78. 41. 40. ... 0. 0. 0.]

[78. 41. 40. ... 0. 0. 0.]

[78. 43. 40. ... 0. 0. 0.]]

etiquetas\_data = [[ 0.9 7.9]

[ 0.9 7.9]

[ 0.9 7.9]

...

[10.4 10. ]

[10.4 10. ]

[10.4 10. ]]

La API Keras provee dos caminos para crear la red: uno se denomina Model Class API y el otro, que fue el que escogí, denominado Sequential model.

Este modelo es menos versátil, pero más simple: solo permite que las capas se agreguen en secuencia, una a continuación de la otra. Model Class API permite la construcción de gráficos computacionales más complejos.

Como resumen, el modelo secuencial es un apilamiento lineal de capas de neuronas.

La siguiente instrucción:

model = K.Sequential()

crea un modelo secuencial “vacío”, sin información sobre capas de neuronas, tanto en cantidad como en la forma en que se inicializaran los valores para pesos y bias, ni tampoco que funciones de activación se utilizaran.

Veamos ahora la siguiente instrucción en la usamos el método add del objeto model:

model.add(K.layers.Dense(32, input\_dim=20, activation=**'relu'**, kernel\_initializer=inicializador, bias\_initializer=inicializador))

el primer parámetro de add es K.layers.Dense(…), que es una simplemente una capa de neuronas totalmente conectada, donde input\_dim=20, indica la dimensión del array de routers.

El número 32 indica la dimensión del espacio de salida. Recordemos que anteriormente mencionamos que nuestra red tendría 4 capas ocultas de 32, 64, 32 y 16 neuronas respectivamente. De manera que la capa de entrada, de 20 neuronas (input\_dim=20) se conectara con una primera capa oculta de 32 neuronas. La cantidad de neuronas de la segunda capa que definido por ese parámetro de la capa de entrada.

Tenemos hasta ahora una capa de entrada de 20 neuronas y definida la cantidad de neuronas de la siguiente (la primera oculta): 32.

La función de activación de la capa de entrada se especifica con activation=**'relu'**. La función ReLU fue descripta en el texto que refiere a la Figura 13.

Finalmente restan dos parámetros: bias\_initializer (derivado de la ordenada al origen en regresión lineal) y kernel\_initializer, que definirá como se inicializan los pesos que vinculan la salida de esta capa de la red neuronal con la entrada de la siguiente. En este contexto, el peso es una medida de cuanto influirá la salida de una neurona en la entrada de otra, a la que está conectada. El entrenamiento de la red consiste en el ajuste de los valores asignados a pesos y bias. De manera que los valores iniciales de ambos definen el lugar desde el cual comienza ese proceso. Debido a que en el entrenamiento se realiza un proceso iterativo de ajuste, puede suponerse que cualquier valor inicial es bueno, pues los algoritmos de ajuste los irán modificando hasta encontrar valores adecuados, sin embargo, este aspecto no es tan trivial, pues está vinculado al tema de [mínimos locales](#Mínimoslocales).

La asignación a kernel\_initializer y a kernel\_initializer se realiza mediante la variable inicializador, asignada a su vez de esta manera:

inicializador = **'random\_normal'**

El string **'random\_normal'** indica que se utilizará un valor aleatorio con distribución gaussiana normal. El código fuente del script wiflow.py4 hay varios otros inicializadores comentados, si el lector quiere probarlos.

El inicializador de tipo **'random\_normal'** bueno para muchos casos y recomendado en las lecturas realicé. Luego de haber probado varios, buscando siempre un rápido ajuste y un bajo valor de error absoluto medio, me decidí por ese.

La definición de las siguientes capas es similar, aunque cambié la función de activación por la función sigmoidea, debido a que obtuve experimentalmente valores de ajuste más estables (no se quedaba oscilando tan frecuentemente en [mínimos locales](#Mínimoslocales))

model.add(K.layers.Dense(64, activation=**'sigmoid'**, kernel\_initializer=inicializador, bias\_initializer=inicializador))  
model.add(K.layers.Dense(32, activation=**'sigmoid'**, kernel\_initializer=inicializador, bias\_initializer=inicializador))  
model.add(K.layers.Dense(16, activation=**'sigmoid'**, kernel\_initializer=inicializador, bias\_initializer=inicializador))  
model.add(K.layers.Dense(2, activation=**'relu'**, kernel\_initializer=inicializador, bias\_initializer=inicializador))

**Entrenamiento de la red:**

En aprendizaje automático (Machine Learning) supervisado, el entrenamiento es un proceso iterativo y automático (afortunadamente) en el que la API utiliza algoritmos para cambiar los valores de pesos y bias tratando de minimizar el valor suministrado por la función de pérdida, aplicando en la entrada de la red los atributos y comparando las etiquetas con los valores obtenidos por la predicción con el modelo.

La pérdida es el valor absoluto de la diferencia entre el ejemplo que se le dio a la red (la etiqueta) y la predicción. Habitualmente la función de pérdida es la media de la pérdida al cuadrado (elevar cada una de las pérdidas al cuadrado y luego obtener la media) considerando todas las predicciones y todas las etiquetas, penalizando así aquellas predicciones que más se alejan del valor ideal. Este proceso se denomina minimización del riesgo empírico.

Reducción de la pérdida

Para reducir la media del cuadrado de la pérdida, se pueden utilizar distintas aproximaciones. Recomiendo visitar la página [An Interactive Tutorial on Numerical Optimization](http://www.benfrederickson.com/numerical-optimization/)[[10]](#endnote-10), que cuenta con animaciones sobre optimización con posibilidad que el usuario ajuste los parámetros y observe como varía la velocidad y exactitud con que las distintas funciones encuentran mínimos, ya sea en dos o en más dimensiones.

Básicamente el proceso consiste en cambiar los hiperparámetros de la red (pesos y bias) y observar si la función de pérdida escogida aumenta o disminuye, es decir, si nos estamos dirigiendo hacia un mínimo o no. En la siguiente iteración se ajusta la dirección y eventualmente la tasa con la que se está llevando a cabo la búsqueda (su nombre en inglés es learning rate).

Un learning rate grande llegará más rápidamente al entorno al mínimo, pero es posible que se pase y eventualmente oscile cerca de el sin alcanzar un valor suficientemente próximo.

Por otra parte, un learning rate demasiado pequeño, requerirá un número muy grande de iteraciones, demandando más tiempo o agotando el número configurado antes de alcanzarlo.

El learning rate es por lo tanto un de los valores de ajuste experimental durante el desarrollo, aunque los valores por defecto suelen funcionar bien en muchos casos.

El mecanismo o estrategia que se utilizará para reducir la función de pérdida está implementado en la clase [optimizers](https://keras.io/optimizers/)[[11]](#endnote-11) de Keras.

Uno de los más comunes es el descenso estocástico de gradiente SGD. Utilizando derivadas o derivadas parciales se “avanza” en la dirección en la que el gradiente desciende. Es un método estable, aunque lento.

En este proyecto utilicé Adam. Puede leerse un artículo interesante sobre como escoger un optimizador en Keras en este link[[12]](#endnote-12) <https://www.dlology.com/blog/quick-notes-on-how-to-choose-optimizer-in-keras/>

En el script wiflow.py4 podrá encontrar varios métodos comentados y elegir otro si prefiere probarlo.

El optimizador se asigna a una variable en esta línea de código:

optimizador = K.optimizers.Adam(lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=**None**, decay=0.0, amsgrad=**False**)

Los parámetros de Adam utilizados son los indicados en el paper original.

Compilar el modelo

El siguiente paso es obtener una versión compilada del modelo para entrenamiento. En ese punto utilizaremos el optimizador elegido e indicaremos la función de pérdida y las métricas que queremos que se nos muestre en el proceso de entrenamiento.

La línea de código que realiza lo dicho es la que sigue:

model.compile(loss=**'mse'**, optimizer=optimizador, metrics=[**'mae'**])

Tanto la función de pérdida utilizada **'mse'** (mean squared error), como la métrica **mae**, han sido definidas mediante los strings que Keras tiene previsto a tal efecto, pero podría haberse utilizado un procedimiento similar al que aplicamos para el optimizador. Puede consultarse la documentación de [métricas](https://keras.io/metrics/)[[13]](#endnote-13) de Keras.

Las métricas utilizan los mismos métodos que la pérdida, pero no se usan para ajustar el modelo.

Encontré una buena referencia sobre como usar las métricas la encontré en una guía de la web de [Machine Learning Mastery](https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/)[[14]](#endnote-14)

**Entrenar el modelo**

Llegó el momento de entrenar el modelo.

Como vamos a usar [Tensorboard](https://www.tensorflow.org/guide/summaries_and_tensorboard)[[15]](#endnote-15) , uno de los parámetros del método de entrenamiento será una callback que generará un log del modelo durante ese proceso. Almacenamos una referencia a ella en una variable.

tbCallBack = K.callbacks.TensorBoard(log\_dir=**'./logs'**, histogram\_freq=0, write\_graph=**True**, write\_images=**False**)

y ahora entrenamos:

epochs = 5000

history = model.fit(atributos\_data, etiquetas\_data, epochs=epochs, verbose=1, callbacks=[tbCallBack])

Como era de esperarse, vemos que se le pasan los atributos y la etiquetas. Puesto que es aprendizaje supervisado, las etiquetas son imprescindibles.

El parámetro epochs indica la cantidad de iteraciones que se llevarán a cabo. Es posible detener el proceso antes de alcanzar el valor asignado a epochs utilizando la callback de [EarlyStopping](https://keras.io/callbacks/#earlystopping)[[16]](#endnote-16), una línea abierta que le queda a este proyecto.

Con verbose=1 veremos en consola barras de progreso del entrenamiento.

El método model.fit devuelve un objeto History cuyo atributo History.history es un registro de valores de pérdida de entrenamiento y valores de métricas en épocas sucesivas.

**Evaluación del modelo**

La evaluación final para saber si el entrenamiento resultó satisfactorio debe realizarse con unos datos a distintos a los utilizados para entrenar el modelo, pero al menos en forma preliminar los datos que predice el modelo entrenado, contrastados con las etiquetas (datos de resultados ideales), deben estar dentro del rango del error admisible para el caso en desarrollo.

El código que sigue muestra gráficos con la evolución a lo largo de las iteraciones de la función de pérdida y de la media del error absoluto.

loss = history.history[**'loss'**]  
mae = history.history[**'mean\_absolute\_error'**]  
  
plt.xlabel(**'Epoch'**)  
plt.ylabel(**'mse/mae'**)  
plt.plot(history.epoch, loss, label=**'mse'**)  
plt.plot(history.epoch, mae, label=**'mae'**)  
plt.legend()  
plt.axis([0, epochs\*1.2, 0, 10])  
plt.show()

El gráfico resultante podría ser uno como este:

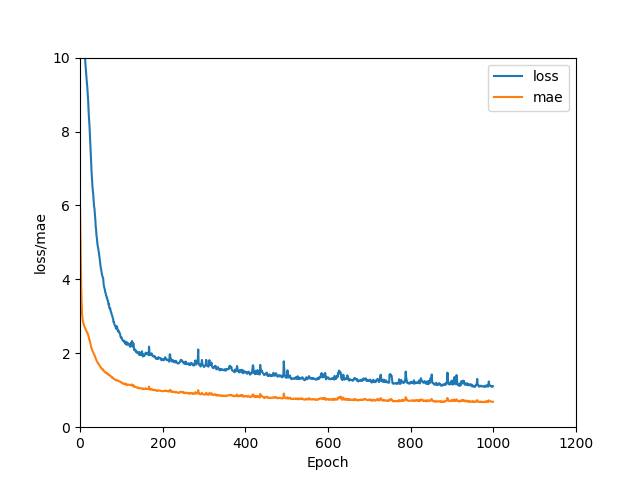


Figura 15

Keras también proporciona un método para evaluar el modelo entrenado. Lo usaremos más adelante con datos de prueba que es donde realmente merece la pena, pero en el script wiflow.py4 podemos también usarlo con los datos de entrenamiento y ver unos resultados preliminares:

scores = model.evaluate(atributos\_data, etiquetas\_data, verbose=1)  
print(**"model.metrics\_names"**)  
print(model.metrics\_names)  
print(scores)

Antes de seguir, hablemos sobre un tema mencionado anteriormente: el problema de los mínimos locales.

**Mínimos locales**

Como los valores de pesos y bias se asignan con un valor aleatorio de distribución gaussiana normal, el entrenamiento comienza en un punto al azar.

Los tensores de entrada constituyen un espacio n dimensional. Si n=3, podríamos esforzarnos en imaginar la superficie en el espacio con sus ondulaciones, máximos y mínimos. Buscar un mínimo consistiría en cortar con la imaginación esa superficie con un plano e ir bajándolo hasta hallar el primer mínimo. Si la superficie tiene más de un mínimo, al descender un poco más ese plano, encontraríamos ese segundo caso. Cada uno de ellos constituye un mínimo local. Y puede haber más.

Volvamos a las animaciones de la página <http://www.benfrederickson.com/numerical-optimization/> 10 y analicemos un problema más sencillo en solo 2 dimensiones:

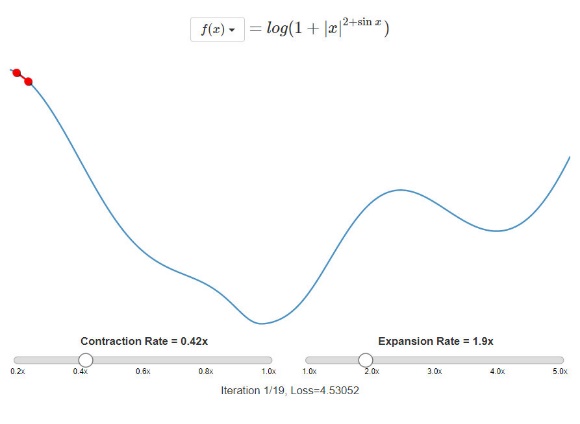


Figura 16

En la Figura 15 vemos el primer salto del proceso de optimización. El círculo rojo más arriba y a la izquierda muestra el punto inicial, la longitud del segmento que lo vincula con el círculo rojo más abajo y a la derecha depende del learning rate. Vemos que la primera iteración fue satisfactoria y el proceso continúa hasta que terminan las iteraciones o hasta que el siguiente valor de pérdida da un valor superior al previo, indicando que la pendiente es positiva, por lo cual será necesario cambiar la dirección en la siguiente iteración. Un resumen del proceso total se muestra en las siguientes 3 figuras:

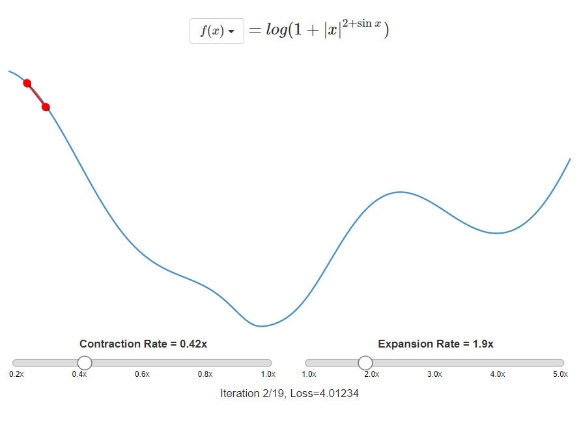


Figura 17

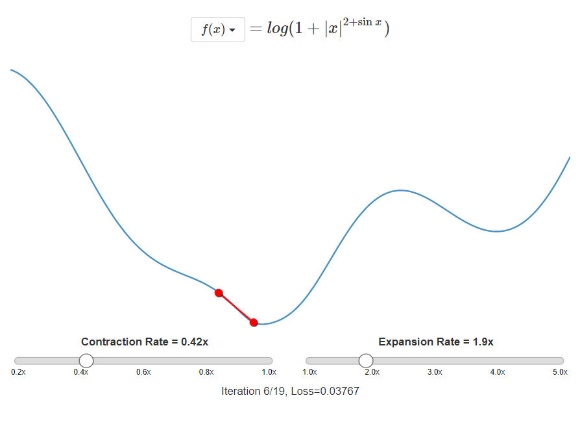


Figura 18

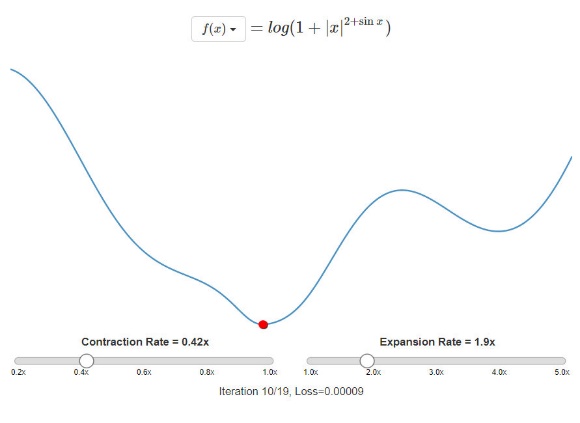


Figura 19

Ahora imaginemos que el proceso comienza en otro punto:

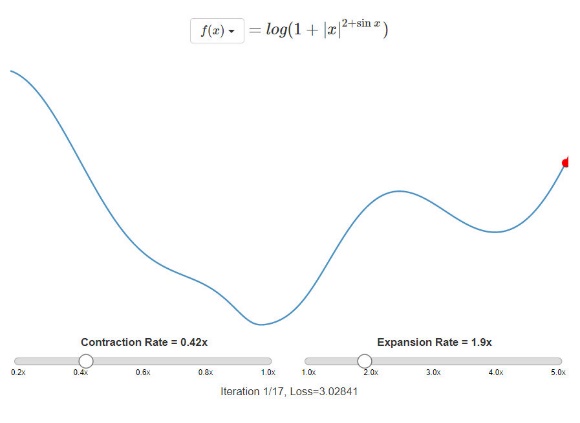


Figura 20

Y si comenzamos el proceso de tratar de encontrar el mínimo, según el procedimiento anteriormente descripto, podría suceder algo así:

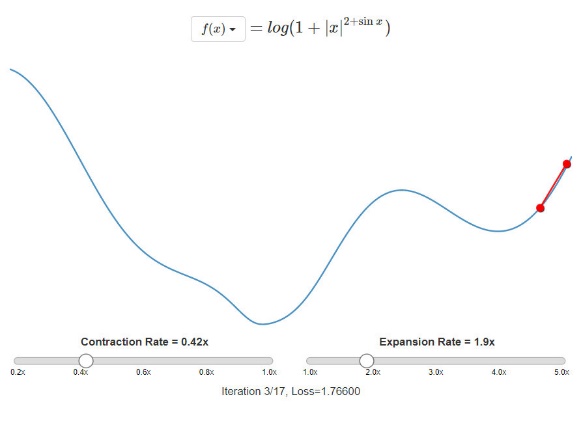


Figura 21

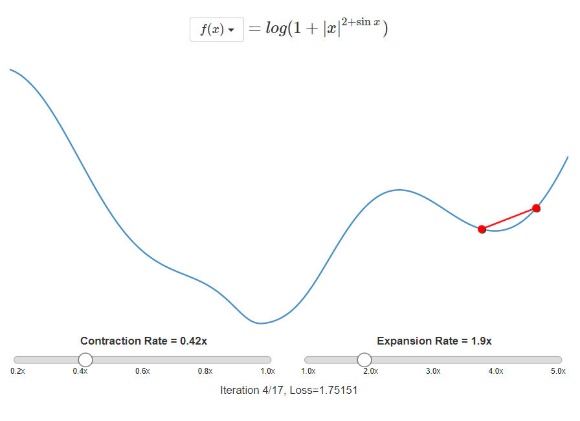


Figura 22

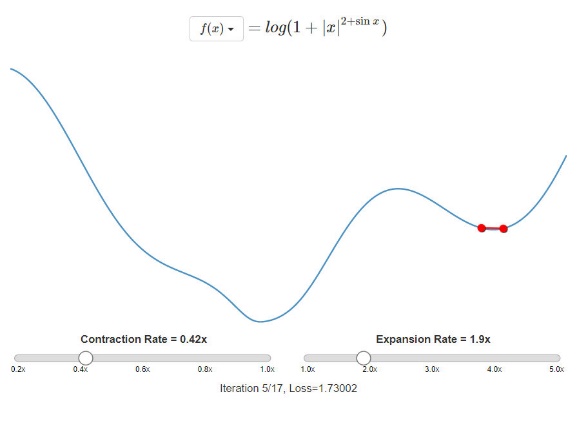


Figura 23

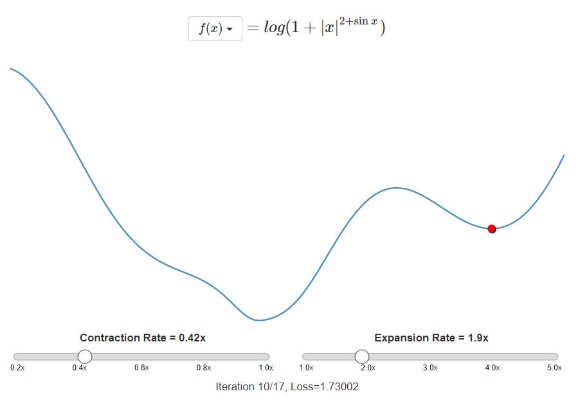


Figura 24

La Figura 23 muestra que, a pesar de haber usado el mismo modelo con los mismos datos, hemos encontrado un mínimo de distinto y muy probablemente una constelación de valores de pesos y bias también distintos a los anteriores.

En la experiencia realizada en este proyecto, en ocasiones no solo los mínimos diferían algo, sino que podía ser de un valor demasiado elevado o incluso no lograr la convergencia, oscilando entre valores muy dispares.

La ventaja de usar una distribución aleatoria es que es fácil reiniciar el entrenamiento y probar si unos nuevos valores de inicialización resuelven el problema.

Que no converja en unos pocos entrenamientos, digamos hasta 3 o 4, no implica que necesariamente el modelo no sirva. En la experiencia de este proyecto, es caso no se dio en más de 2 veces consecutivas.

**Guardar el modelo**

Una vez que consideramos el modelo satisfactoriamente entrenado, podemos guardarlo para usarlo posteriormente en un ambiente de producción, sin tener que repetir el proceso de entrenamiento.

Puede guardarse toda la definición del modelo (definición de la estructura de la red y valores de pesos y bias) en un solo archivo o, la configuración de la red por un lado y los pesos por otro.

El siguiente código realiza ambas cosas:

*# serializar el modelo a JSON*model\_json = model.to\_json()  
**with** open(**"model.json"**, **"w"**) **as** json\_file:  
 json\_file.write(model\_json)  
*# serializar los pesos a HDF5*model.save\_weights(**"model.h5"**)  
print(**"Modelo Guardado!"**)  
model.save(**'model.hdf5'**)  
print(**"Modelo HDF5 Guardado!"**)

**Recuperar el modelo guardado**

El código que recupera y usa el modelo guardado en las etapas anteriores, está en el archivo wiflowGuardado.py4. Aún trabajando en Python y Keras, no en Android y Tensorflow.

Las primeras líneas tienen partes es muy similares a la utilizadas en el entrenamiento del modelo.

Los atributos que se cargan en este caso, no son los mismos que los del entrenamiento del modelo, sino que son los de test.

Las etiquetas son los valores ideales que se esperaría que la red devuelva al procesar los atributos. Serán usados para compararlos con la predicción y así conocer la pérdida del modelo.

Esta es la fracción de código que hace eso:

archivo = **"test.csv"**df = pd.read\_csv(archivo)  
*#print (df)*pr = pd.DataFrame(df, columns=[**'0'**, **'1'**,**'2'**, **'3'**, **'4'**, **'5'**, **'6'**, **'7'**, **'8'**, **'9'**, **'10'**, **'11'**, **'12'**, **'13'**, **'14'**, **'15'**, **'16'**, **'17'**, **'18'**, **'19'**])  
*#print (pr)*xy = pd.DataFrame(df, columns = [**'x'**, **'y'**])  
*#print (xy)*atributos = pr[pr.columns[:20]].as\_matrix()  
*#print (atributos)*etiquetas = xy[xy.columns[:2]].as\_matrix()  
*#print(etiquetas)*atributos\_data = np.array(atributos, **"float32"**)  
etiquetas\_data = np.array(etiquetas, **"float32"**)

Las líneas que efectivamente cargan en modelo son estas:

modeloGuardado = **'model\_final'**modelo = modeloGuardado + **'.json'**pesos = modeloGuardado + **'.h5'***# cargar json y crear el modelo*json\_file = open(modelo, **'r'**)  
loaded\_model\_json = json\_file.read()  
json\_file.close()  
loaded\_model = K.models.model\_from\_json(loaded\_model\_json)  
*# cargar pesos al nuevo modelo*loaded\_model.load\_weights(pesos)  
print(**"Cargado modelo desde disco."**)

Una vez cargada la definición del modelo, esta debe ser complidada:

*# optimizador = tf.train.RMSPropOptimizer(0.001)*optimizador = K.optimizers.Adam(lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=**None**, decay=0.0, amsgrad=**False**)  
*# Compilar modelo cargado y listo para usar.*loaded\_model.compile(loss=**'mean\_squared\_error'**, optimizer=optimizador)

Finalmente, realizamos la predicción con el modelo sobre los atributos de test que leímos del archivo **test.csv** al principio:

print(**"Predicción con el modelo cargado"**)  
loaded\_resultado = loaded\_model.predict(predecir)

la predicción queda almacenada en la variable loaded\_resultado.

En la siguiente figura -Figura 25- se muestra la comparación entre el valor real de la posición y la predicción, mediante la expresión del error absoluto:



Figura 25

El resultado de calcular la media del error absoluto de la predicción de los atributos de test es 1.4. Si calculamos la distancia de la posición de la predicción a la posición real como la raíz de la suma del cuadrado del error en x más el cuadrado del error en y, su valor lo vemos en la columna de la derecha, de nombre *distancia*. Su valor medio en este caso es 2.2 metros.

* Valor medio del error absoluto del error de posición en ***X*** y en ***Y***: 1.4 metros
* Valor medio del error de distancia de la predicción: 2.2 metros

En el histograma que sigue (Figura 26) vemos la cantidad de casos por cada rango de distancia de error. Por ejemplo, errores entre 0 y 1 metro: 21 casos de 74 y el porcentaje de casos acumulado para cada uno de los rangos, esto es: de 0 a 1 metros, entre 1 y 2 metros, etc.

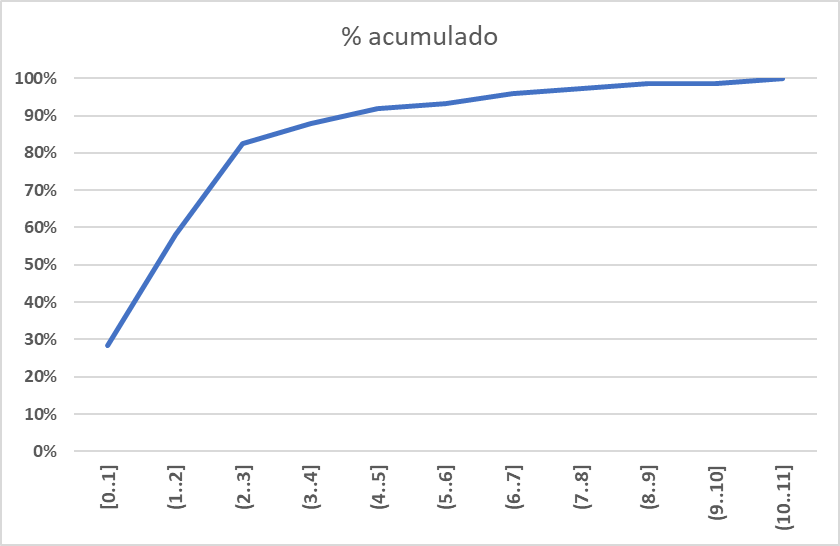
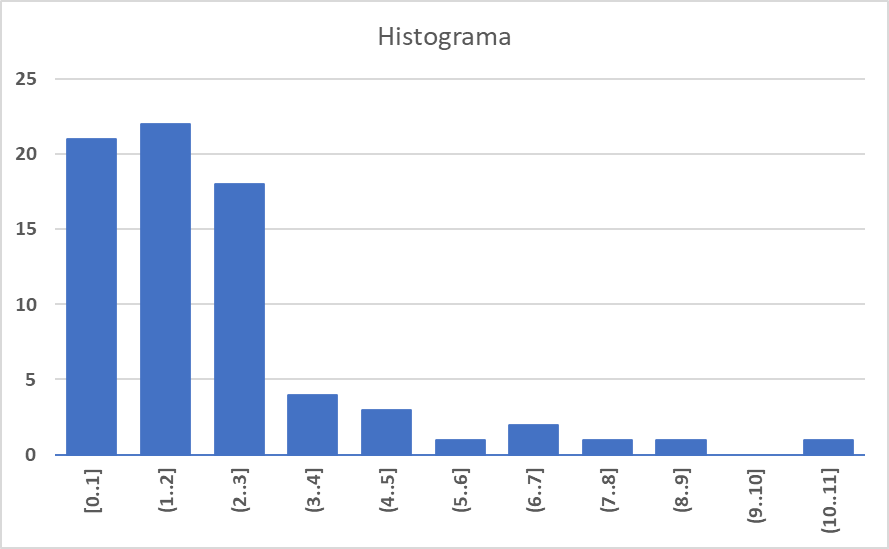


Figura 26

**Tensorboard**

Tensorboard es un conjunto de herramientas de visualización para visualizar el modelo, trazar métricas cuantitativas sobre la ejecución del modelo y mostrar datos adicionales.

Tensorboard se instala junto con TensorFlow, no requiere instalación adicional.

Tensorboard lee el (o los) archivo/s de los logs que se escribieron en la carpeta que definimos al instanciar la callback en ele apartado *Entrenar el modelo*. En nuestro caso: log\_dir=**'./logs'**

Para utilizarlo, en una consola de comando, escribimos:

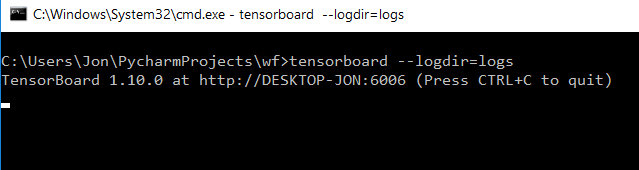
tensorboard –logdir=path\_archivos\_de\_log

si estuviésemos situados en la carpeta en la que se entrenó el modelo, el comando sería:

tensorboard –logdir=logs

**nota**: observe que son dos guiones medios antes de la palabra logdir

La respuesta indicará una url a ejecutar en un navegador para utilizar Tensorboard



En el caso que se muestra: http://DESKTOP-JON:6006

Veamos ahora algunas de las pantallas de Tensorboard:

En la Figura 27podemos observar algunas métricas, como la evolución de la media del error absoluto a lo largo de las iteraciones.

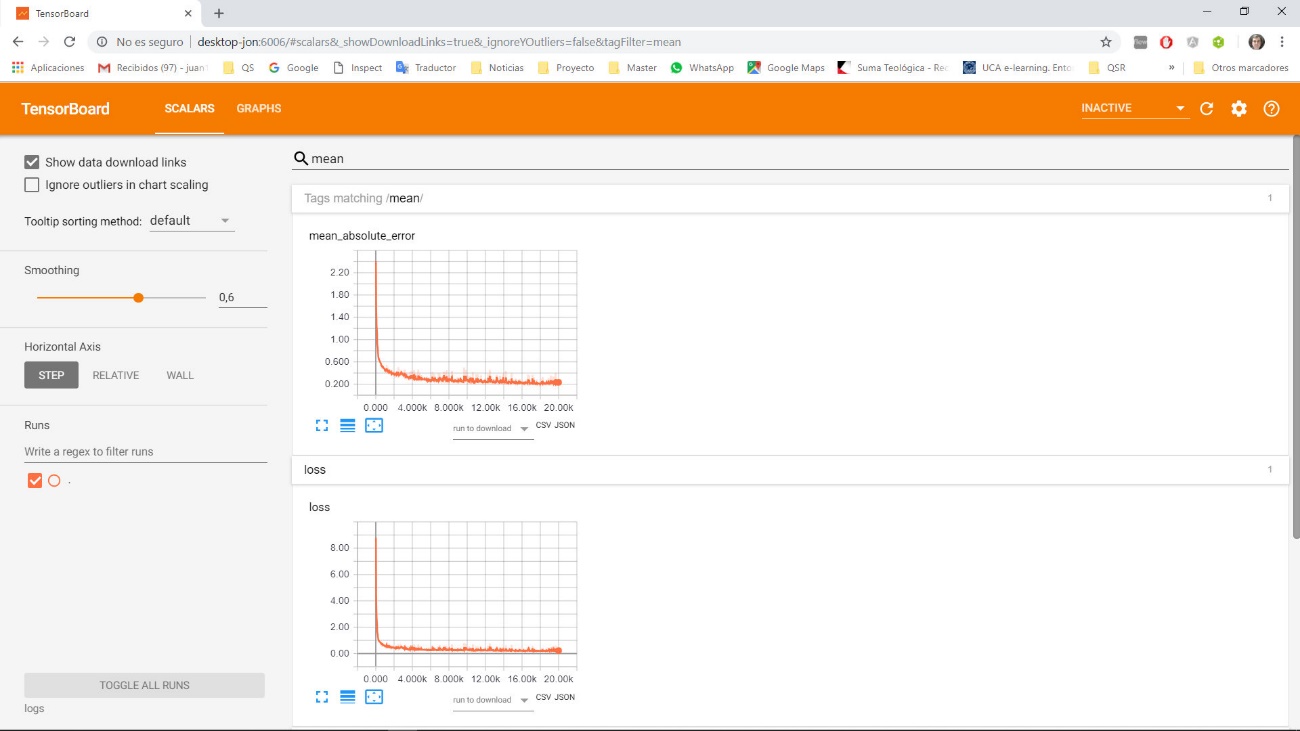


Figura 27

La Figura 28 muestra un mapa de los grafos de la red neuronal, con sus vinculaciones y operaciones. Vemos que los nombres de las distintas capas. La capa de entrada y de salida se muestras en color violeta, mientras que las ocultas en verde.

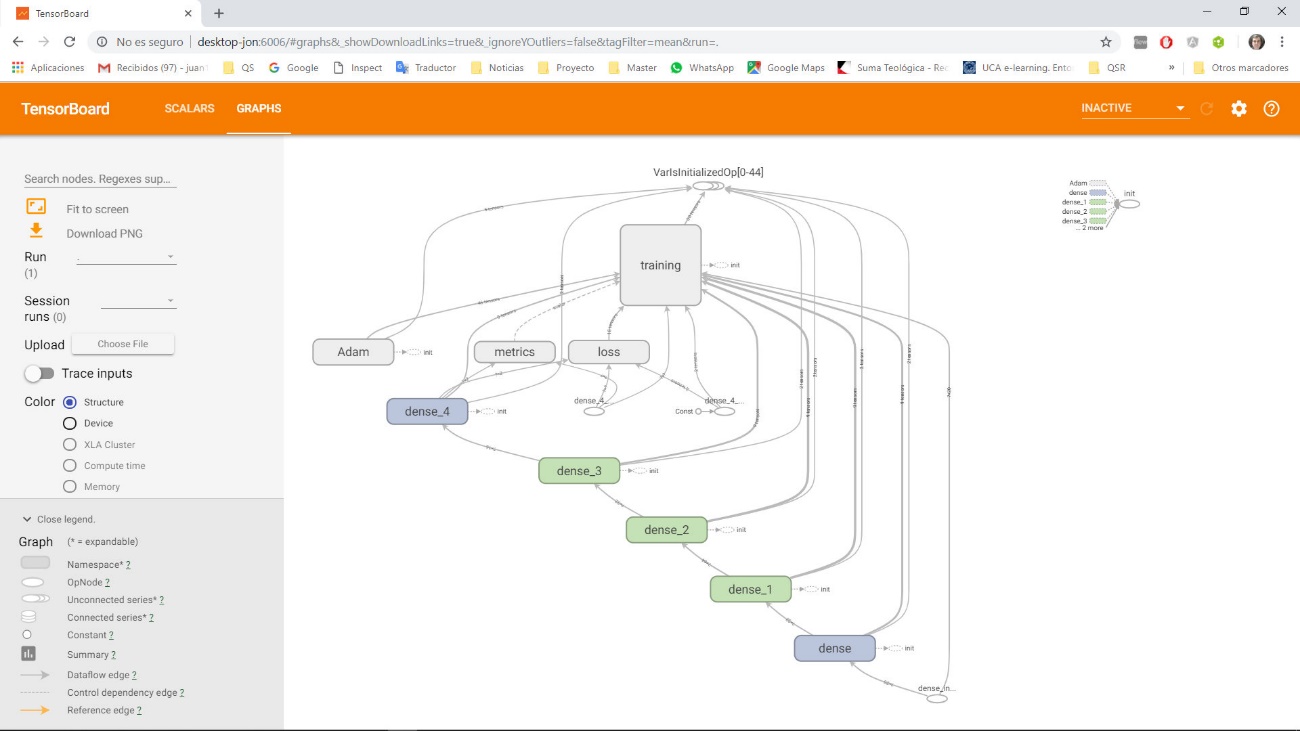


Figura 28

En la Figura 29, vemos que al hacer doble click sobre la capa dense, podemos ver una expansión de ella y su operaciones internas. Tomemos nota del Operation-Placeholder de nombre dense\_input, que “alimenta” a la capa de entrada, pues lo necesitaremos para exportar el modelo desde Keras a Android.

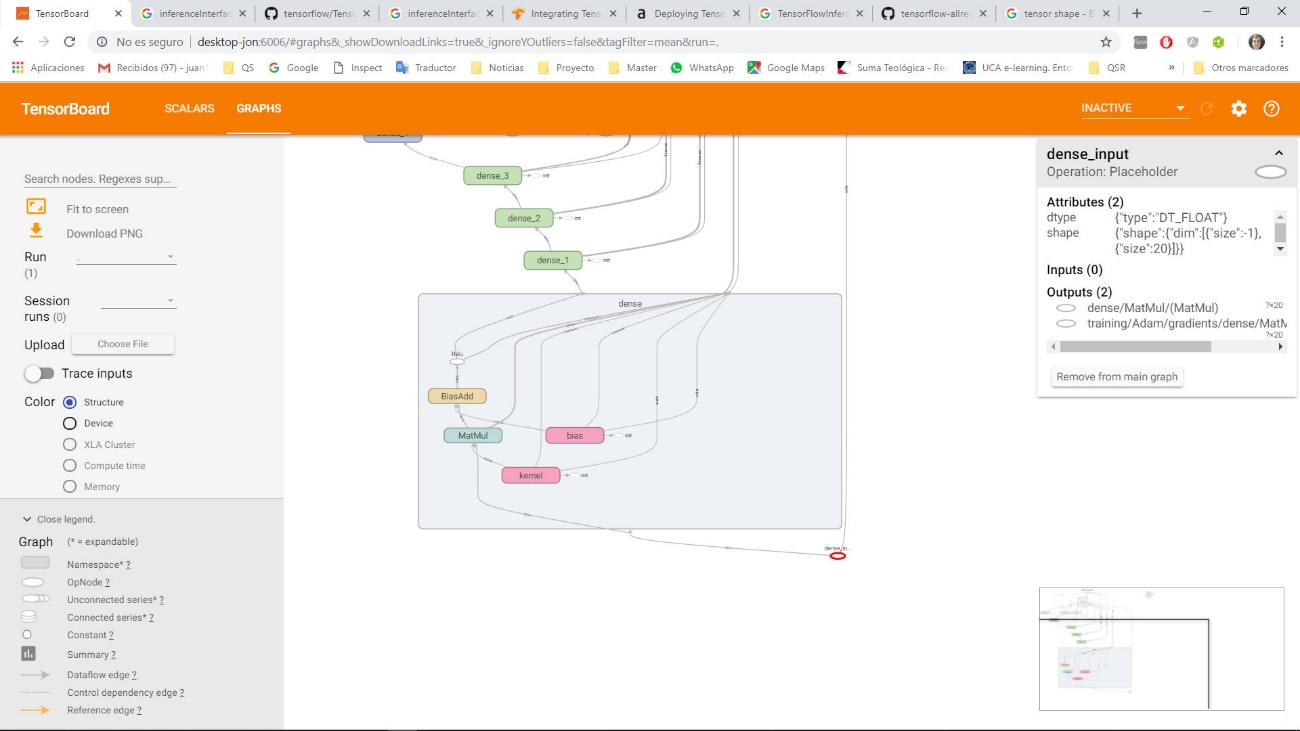


Figura 29

Finalmente, en la Figura 30 hemos ampliado la capa de salida dense\_4. Tomemos aquí nota del nombre de la operación de salida dense\_4/Relu: menciona el nombre de la capa y su función de activación. También lo necesitaremos al exportar el modelo desde Keras a Android.

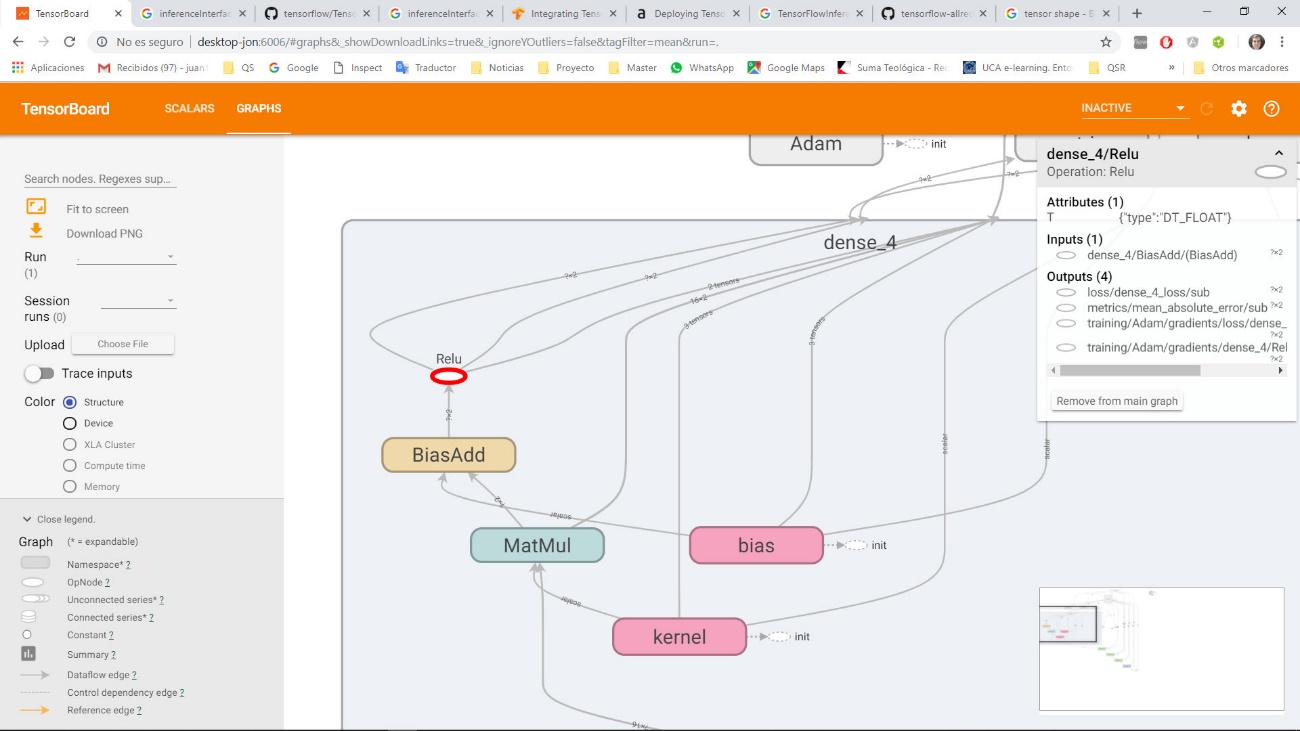


Figura 30

**Usar Keras en Android**

Usar Keras en Android es muy sencillo, encontrar como hacerlo fue una de las cosas más complicadas de este proyecto.

Buscando en la web “*export keras model to Android*” y otras frases similares, se encuentran muchas entradas con ejemplo de código que en algún punto piden el uso del método freeze\_graph:

…

from tensorflow.python.tools import freeze\_graph

…

freeze\_graph.freeze\_graph('out/' + model\_name + \_graph.pbtxt', None, \

False, 'out/' + model\_name + '.chkp', output\_node\_name, \

"save/restore\_all", "save/Const:0", \

'out/frozen\_' + model\_name + '.pb', True, "")

Invariablemente la ejecución del código termina en el error “list index out of range”, comentado en muchos lugares en la web, tanto en [stackoverflow](https://stackoverflow.com/questions/51854908/tensorflow-freeze-graph-indexerror-list-index-out-of-range)[[17]](#endnote-17) como en [github](https://github.com/tensorflow/tensorflow/issues/22029)[[18]](#endnote-18), pero sin encontrar solución en el momento de escribir esta memoria.

Finalmente di con un repositorio en [github](https://github.com/amir-abdi/keras_to_tensorflow)[[19]](#endnote-19) de [Amir Abdi](https://www.linkedin.com/in/amir-abdi-26135939/)[[20]](#endnote-20), a quien menciono por el crédito del script que permitió llevar a cabo la tarea y del cual obtuve el código Python que convierte un archivo con el modelo en formato Keras en otro en formato Tensorflow.

En este proyecto el nombre del script es export.py4 y tiene pequeñas modificaciones para que los parámetros default coincidan con los nombres utilizados en las etapas anteriores.

El script export.py espera un archivo de nombre model\_final.hdf5, generado mediante el método K.model.save(), de Keras. Ese archivo se genera en wiflow.py con la línea

model.save(**'model\_final.hdf5'**)

como salida, se obtiene el archivo model\_final.hdf5.pb, que deberá residir en la carpeta assets del proyecto [WiFlow](https://github.com/jon1721/wiflow)4 en Android Studio.

Preparar Android Studio

En el archivo build.gradle de la aplicación, agregar en dependencias la siguiente línea para importar la librería tensorflow:

dependencies {

. . .

. . .

. . .

implementation **'org.tensorflow:tensorflow-android:1.8.0'**

}

Las siguientes líneas están escritas en referencia al archivo MainActivity del proyecto WiFlow4

Importamos el paquete de interface java a TensorFlow:

**import** org.tensorflow.contrib.android.TensorFlowInferenceInterface;

Declaramos la variable que contendrá un objeto de la clase TensorFlowInferenceInterface:

**private** TensorFlowInferenceInterface **inferenceInterface**;

Instanciamos el objeto y le pasamos al constructor el archivo con el modelo entrenado en formato TensorFlow:

**inferenceInterface** = **new** TensorFlowInferenceInterface(getAssets(), **"model\_final.hdf5.pb"**);

Ahora ya estamos listos para hacer predicciones:

Las siguientes matrices contienen los atributos que queremos usar para predecir, es decir, los atributos de test, y los valores reales de las posiciones que nos van a servir para calcular el error con el que el modelo predice.

**float**[][] atributos = {  
 {50,48,45,44,31,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  
 {49,45,35,44,35,0,34,34,30,0,0,0,0,0,0,0,0,0,0,0},  
 . . . . . . . . . . . . . . . . . . . . . . .  
 {72,45,39,44,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}  
};  
  
**double**[][] etiquetas = {  
 {0.9,7.9},  
 {0.9,8.9},  
 . . . .   
 {10.4,10.0}  
};

El siguiente bucle itera los atributos llamando al método, definido más abajo, e imprimiendo en pantalla y en el logcat los valores que predice el modelo y los valores reales de la posición:

**for**(**int** i=0 ; i<atributos.**length** ; i++){  
 **output** = predict(atributos[i]);  
 Log.*d*(**TAG**, **"Predicción: "** + String.*format*(**"%.2f"**, **output**[0]) + **", "** + String.*format*(**"%.2f"**, **output**[1])  
 + **" Posición real: "** + String.*format*(**"%.2f"**, etiquetas[i][0]) + **", "** + String.*format*(**"%.2f"**, etiquetas[i][1]));  
 **texto** += **"Predicción: "** + String.*format*(**"%.2f"**, **output**[0]) + **", "** + String.*format*(**"%.2f"**, **output**[1])  
 + **" Posición real: "** + String.*format*(**"%.2f"**, etiquetas[i][0]) + **", "** + String.*format*(**"%.2f"**, etiquetas[i][1]) + **"\n"**;  
  
 **salida**.setText(**texto**);  
}

Veamos finalmente el método que realiza la predicción:

**private float**[] predict(**float**[] input){  
 *// nuestro modelo tiene dos neuronas de salida* **float** output[] = **new float**[2];  
  
 **inferenceInterface**.feed(**"dense\_input"**, input, 1, input.**length**);  
 **inferenceInterface**.run(**new** String[]{**"dense\_4/Relu"**});  
 **inferenceInterface**.fetch(**"dense\_4/Relu"**, output);  
  
 **return** output;  
}

Se utilizan 3 métodos de la clase TensorFlowInferenceInterface:

* feed
* run
* fetch

**feed** tiene 3 parámetros: el primero es un String con el nombre de la primera capa, seguido por **\_input** (podemos obtener el nombre usando Tensorboard), el segundo es el array con los atributos, el tercero es el número de dimensiones del array y el cuarto es su tamaño.

**run** realiza la predicción en función de parámetros que recibió feed y la deja disponible para se consultada en los nodos de salida. Recibe una lista de strings con los nombres de los nodos de salida y su función de activación (también podemos obtener el nombre usando Tensorboard).

**fetch** realiza la consulta de la predicción realizada por run, en la salida indicada en el primer parámetro, dejando el resultado en el segundo.

## Esquema del diseño

Diagrama con los diferentes componentes del diseño y sus interrelaciones.

Justificación de las principales decisiones tomadas en el diseño.

## Modelo de datos

Esquema de la base de datos, incluyendo relaciones entre las tablas.

Listado de servicios webs.

## Vistas

Esquema que muestre las principales pantallas de la aplicación y el diagrama de navegación.

# Capítulos adicionales

Si consideras puedes poner más capítulos adicionales

# Conclusiones

Grado de cumplimiento de los objetivos planteados.

Líneas abiertas.

Consideraciones personales.

# Anexos

## Listado de fuentes entregadas / Código fuente en GitHub

## Manual de usuario

Links

Estoy preparando cierto marco teórico en la memoria del proyecto y me gustaría saber si estás de acuerdo con los siguiente:

**¿Qué es Deep Learning? (Ilustración 1)**

Dentro de la IA hay muchas cosas: razonamiento, planificación, toma de decisiones, procesos de decisión de Márkov, etc.

Una parte (no pequeña) de la IA es el aprendizaje automático. Dentro de Aprendizaje Automático estarías las Redes Bayesianas, las Máquinas Vectores de Soporte, K-medias, Aprendizaje por refuerzo,

Las Redes Neuronales constituyen un área dentro del Aprendizaje Automático

Deep Learning es un área dentro de las RedesNeuronales, que consiste en tomar las Redes Neuronales, aplicarles nuevas técnicas y organizarlas de cierta forma.

Nota: al Aprendizaje Automático, también se lo llama Machine Learning

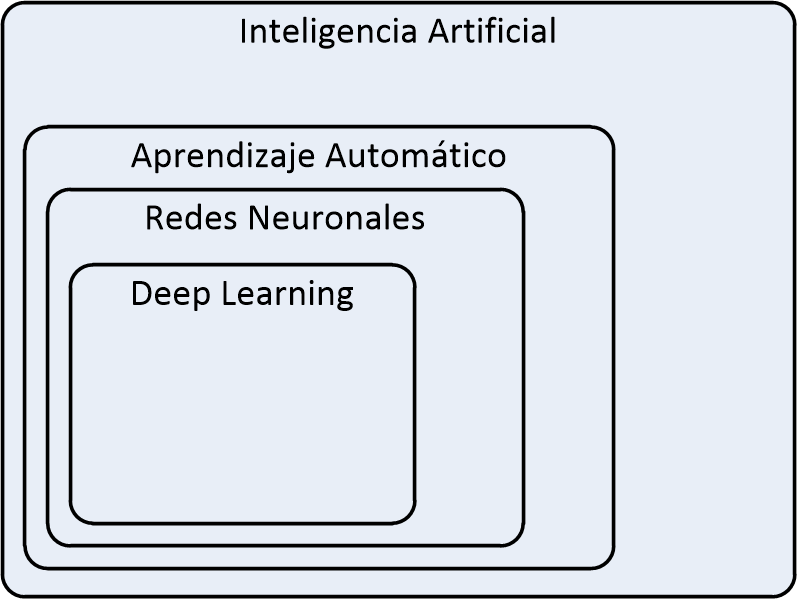


Ilustración 1

Deep Learning (Ilustración 2)

Lógica y SBC:

En Sistemas Basados en el Conocimiento y Lógica (primera columna), escribimos un programa en forma manual: estructura, arquitectura… todo. Son sistemas sin fase de entrenamiento .  
Ingresan datos y dan una salida. Si es incorrecta, hay que rehacer le programa.

Aprendizaje tradicional:

Sistemas probabilísticos, aprendizaje por refuerzo, las primeras redes neuronales. Se divide el problema, pues se diseña manualmente no todo el programa, sino ciertas propiedades. El desarrollador decide qué características son significativas. Por ejemplo, en redes bayesianas, decide cuales son los nodos y cuales son las relaciones entre esos nodos. Lo que se aprende mediante entrenamiento en este caso son las tablas de probabilidad condicional.

En la Ilistración 2, lo que lleva a cabo un humano está dibujado en fondo blanco, mientras que lo que hace la máquina tiene un fondo casi anaranjado (ni verde, ni blanco).

Aprendizaje de Representación:

Se busca que el agente inteligente aprenda las propiedades.

En Redes Neuronales, la importancia de las características la asigna la misma red. Es decir en la fase de entrenamiento usa las propiedades para ver cuanto peso le tiene que asignar a cada una.

Deep Learning:

Poniendo ciertos tipos de redes neuronales en cierto orden se observo que las primeras aprendían algunas características y la siguiente capa, basada en el aprendizaje anterior aprendía otras características más complejas.

Esta “profundidad” de capas de aprendizaje le da el nombre a Deep Learning

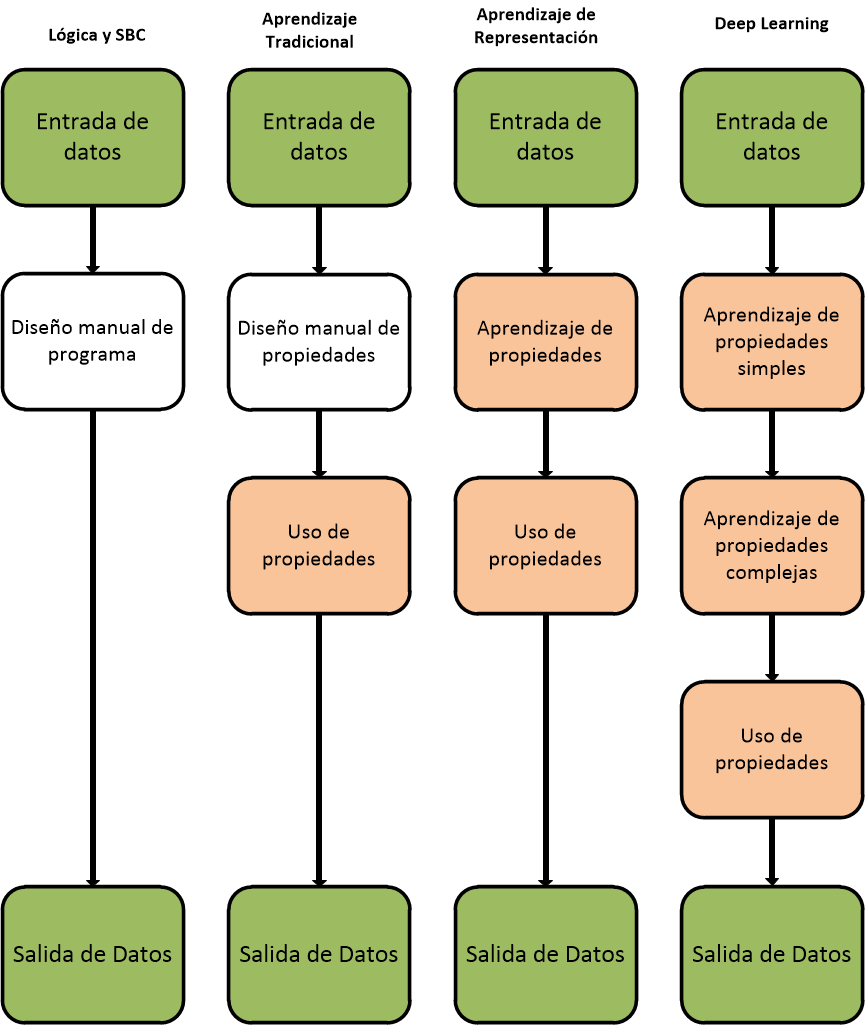


Ilustración 2

1. http://androidcurso.com/index.php/recursos/proyectos/67-proyectos-android-fundamentos/proyectos-android-fundamentos-6-ed?start=5 [↑](#endnote-ref-1)
2. http://www.dcomg.upv.es/~jtomas/android/ProyectosFundamentos2015/WifiLocus.pdf [↑](#endnote-ref-2)
3. https://github.com/jon1721/WiFiLocusModificado.git [↑](#endnote-ref-3)
4. Archivo disponible en https://github.com/jon1721/wiflow.git [↑](#endnote-ref-4)
5. https://www.rebasedata.com/convert-sqlite-to-csv-online [↑](#endnote-ref-5)
6. https://play.google.com/store/apps/details?id=com.farproc.wifi.analyzer.classic [↑](#endnote-ref-6)
7. https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw [↑](#endnote-ref-7)
8. https://pandas.pydata.org/ [↑](#endnote-ref-8)
9. http://www.numpy.org/ [↑](#endnote-ref-9)
10. http://www.benfrederickson.com/numerical-optimization/ [↑](#endnote-ref-10)
11. https://keras.io/optimizers/ [↑](#endnote-ref-11)
12. https://www.dlology.com/blog/quick-notes-on-how-to-choose-optimizer-in-keras/ [↑](#endnote-ref-12)
13. https://keras.io/metrics/ [↑](#endnote-ref-13)
14. https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/ [↑](#endnote-ref-14)
15. https://www.tensorflow.org/guide/summaries\_and\_tensorboard [↑](#endnote-ref-15)
16. https://keras.io/callbacks/#earlystopping [↑](#endnote-ref-16)
17. https://stackoverflow.com/questions/51854908/tensorflow-freeze-graph-indexerror-list-index-out-of-range [↑](#endnote-ref-17)
18. https://github.com/tensorflow/tensorflow/issues/22029 [↑](#endnote-ref-18)
19. https://github.com/amir-abdi/keras\_to\_tensorflow [↑](#endnote-ref-19)
20. https://www.linkedin.com/in/amir-abdi-26135939/ [↑](#endnote-ref-20)