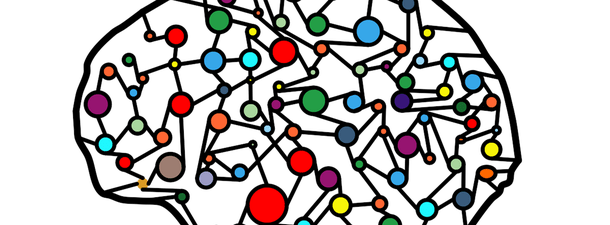
Trabajo Práctico Integrador

Desarrollo de un modelo de aprendizaje automático

2019



|  |  |
| --- | --- |
| DOCENTES   * García, Mario Alejandro (Adjunto) * Forte, Guillermo Omar (JTP) | INTEGRANTES   * Andrada, Emiliano – 59809 * Castillo, Luis Michel – 70259 * Crespo, Mickaela - 71291 * Perez Pinelli, Juan – 69602 |

# Índice

[**Fundamentos teóricos del modelo de clasificación elegido**](#_heading=h.ltfceriaf1rj) **2**

[**Descripción del problema resuelto**](#_heading=h.tc23x5l0o6hf) **4**

[**Explicación de la solución**](#_heading=h.46bz0sha9lzj) **5**

[Diseño de la Solución](#_heading=h.iimu03b98pp7) 5

[Entrenamiento](#_heading=h.wgabd3renshf) 5

[**Código fuente**](#_heading=h.6nso2rnf437n) **7**

[**Análisis del comportamiento de la solución**](#_heading=h.hwqrwn3t7l1x) **8**

[**Problemas encontrados y soluciones aplicadas**](#_heading=h.z21v81exk6xy) **9**

[Tasa de entrenamiento](#_heading=h.1t3h5sf) 9

[Consumo de recursos](#_heading=h.137x33se51g6) 10

[**Conclusiones**](#_heading=h.qow9ffp7rpyz) **11**

# Fundamentos teóricos del modelo de clasificación elegido

Para comenzar a fundamentar por qué utilizamos una red neuronal, empezaremos por definirla. Una Red Neural o Red Neuronal es una estructura compuesta de un número determinado de unidades interconectadas, las cuales reciben el nombre de neuronas. Cada una de estas unidades posee una característica de entrada/salida y lleva asociada una ‘computación’, cálculo o función, en donde la salida está fuertemente determinada al igual que su interconexión con otras unidades, etc.

Esta suerte de topología’, como una forma de arreglo masivo de procesamiento, el alto grado de interconectividad de sus respectivos elementos en los que la información puede fluir fueron los que nos hicieron decantarnos a utilizar una para resolver esta actividad práctica.

Ya con las entradas de datos en nuestro poder, decidimos graficarlas para poder tener un pantallazo general de qué tipo de red neuronal nos iba a ser necesario. Teníamos en cuenta de que, si las clases son linealmente separables, una red neuronal con tan sólo una neurona nos iba a alcanzar (simplemente, un clasificador lineal). Lamentablemente este no era el caso. Por lo cual, nos vimos en la necesidad de tener que incluir capas intermedias u ocultas para poder separar las clases ya no con hiperplanos, sino con hipersuperficies. Así, nuestra red constará de una capa de entrada, una o varias capas intermedias y una capa de salida.

En el desarrollo de este trabajo se detallan cuántas neuronas utilizamos y con ello, cómo planteamos nuestra red para que, luego de mucho entrenamiento, llegáramos a conseguir los resultados necesarios para cumplir con el objetivo.

# Descripción del problema resuelto

Se realizó un análisis sobre los datos de entrada mediante diversos gráficos en 2 dimensiones en busca de pistas que puedan colaborar a la solución del mismo. Se observó que el conjunto de entradas es claramente no linealmente separable, al menos de a 2 dimensiones y se hicieron pruebas aplicando funciones adicionales sobre las entradas.

Algunos ejemplos del análisis:

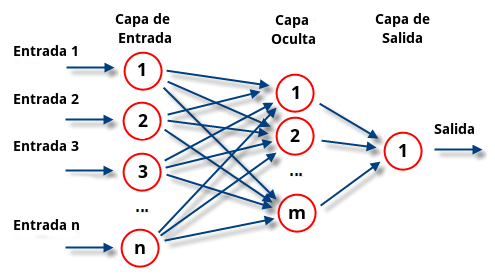
|  |  |
| --- | --- |
| X0 vs X2 | X0 vs X4 |
| X2 vs X4 | X42 vs X0 \* X1 \* X2 \* X3 |

En términos generales, los resultados más interesantes surgieron de analizar la última entrada (X4), pero no se pudo encontrar información adicional útil a partir de este análisis.

# Explicación de la solución

## Diseño de la Solución

Se procedió entonces a desarrollar un algoritmo de entrenamiento para una red neuronal de topología variable, es decir, la cantidad de capas ocultas y cantidad de neuronas por capa son configurables, y todas las neuronas de cada capa están interconectadas con todas las neuronas de la capa siguiente.



Como se observó que el conjunto es no linealmente separable, entonces la red debe incorporar no linealidad para poder resolverlo, para lo cual se introduce no linealidad mediante el uso de neuronas tipo Adaline, las cuales tienen una función de activación no lineal, en nuestro caso la función sigmoide. Esto además impide que la red tenga una simplificación a una única neurona equivalente.

La salida siempre será una única neurona ya que sólo debemos clasificar entre dos categorías, “0” y “1”. Como la salida de la red es probabilística, se la binariza cortando en 0,5 el valor de corte para la función sigmoide utilizada como función de activación.

## Entrenamiento

Se utiliza el 75% (1500 registros) de los datos provistos en el archivo “X\_train.csv” para entrenamiento del algoritmo y el 25% (500 registros) restante se utiliza para validación del algoritmo. Se divide el set de datos con salidas conocidas en dos para poder detectar casos de sobreentrenamiento de la red, en cuyo caso se observará una disminución del error para el conjunto de entrenamiento pero un aumento para el conjunto de prueba.

La red se entrena mediante la técnica de “*Backpropagation*” (propagación hacia atrás), mediante la cual se calcula la incidencia de cada neurona en la última capa sobre el error en la salida y se ajusta en consecuencia, luego se sigue propagando de igual manera hacia las capas anteriores. De esta manera cada capa se ajusta en función a su contribución relativa y una fracción del error total. Al aplicar esta técnica se observa que cada capa busca reducir el error basado en el vector gradiente del mismo.

La configuración de los parámetros se fue haciendo de manera experimental, teniendo en cuenta las variables modificables:

* Entradas: Aplicar funciones no lineales sobre las entradas.
* Cantidad de capas de la red.
* Cantidad de neuronas de cada capa.
* Tasa de aprendizaje.
* Cantidad de ciclos de entrenamiento.

Al probar con diversas configuraciones, se alcanzó una primera solución con 3 capas ocultas y un error entre 11% (para el set de entrenamiento), y 15% (para el set de prueba). Luego de más pruebas se halló una red más compleja, de 5 capas ocultas que requirió más tiempo de entrenamiento y logró reducir el error al 11% para ambos sets de datos.

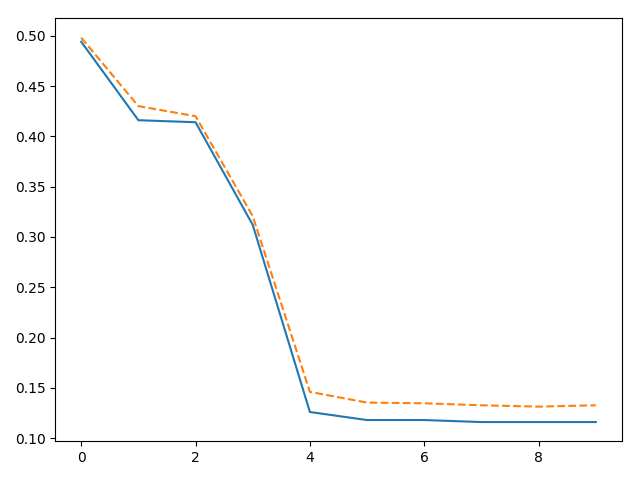


Gráfico de proporción de error durante el entrenamiento. (1 : 2000 iteraciones)

En línea continua se observa la proporción de error para el set de entrenamiento, y en línea entrecortada la proporción de error para el set de prueba.

# Código fuente

Se anexa el código fuente en python.

# Análisis del comportamiento de la solución

Se decidió utilizar una red neuronal para el desarrollo de este script. En primer lugar, se importan las librerías ***csv***, que nos va a permitir la lectura de los archivos, ***Numpy*** para el procesamiento numérico, y ***matplotlib.pyplot*** que nos va a servir para los gráficos.

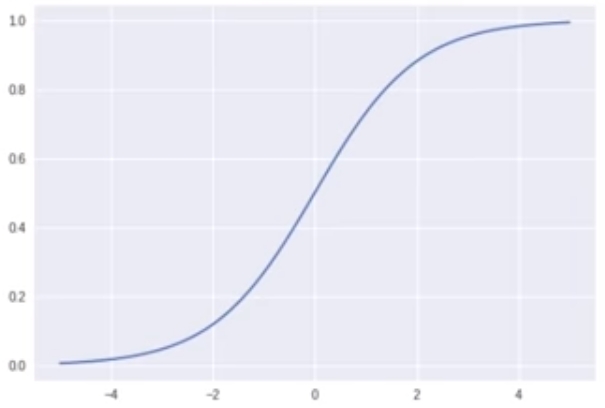
Como bien sabemos, la unidad de procesamiento básica de una red neuronal es la neurona. Juntando varias de estas neuronas en diferentes capas, nos va a permitir el procesamiento de información más compleja que lo que podríamos hacerlo habitualmente. A su vez, una característica muy importante de la codificación de las redes neuronales es que se nos permite vectorizar cada una de las fórmulas que utilizamos en su construcción. Lo cual nos añade una cuota de eficiencia y performance en el rendimiento a la hora de procesar la información disponible. Esto quiere decir que dentro de una misma capa, vamos a estar realizando las mismas operaciones para todas las neuronas, por que tienen la misma función de activación y por qué no también por que aplicamos la misma fórmula de *backpropagation*. Esto quiere decir que podríamos considerar a cada capa de nuestra red neuronal como módulos bien definidos.

Es por lo charlado anteriormente, que es crítico para el caso la creación de nuestra primer capa (*class Capa:*). En ella, solamente se van a incluir los parámetros propios de una capa y nada de lógica, entre los cuales encontramos:

* Número de conexiones que entran a esta capa, de la capa anterior.
* Números de neuronas de la capa.
* Función de activación.

Definidas estas variables, ya tenemos una clase que nos permita modelar nuestra nuestra red neuronal. Si bien inicialmente sólo iba a ser una estructura de datos, luego se agregó la función “*procesar\_lote():*” para hacer el procesamiento de un conjunto de entradas dado.

Seguidamente se ha de crear nuestras funciones de activación. En primera instancia, recordemos que estas funciones son aquellas por las cuales se pasa la suma ponderada de valores de entrada que se realiza en la neurona y que nos permite introducir no linealidades. Si bien existen varios tipos, se decidió utilizar la función sigmoide.



Función sigmoide

Tenemos que tener en cuenta de que las redes neuronales utilizan el algoritmo de *backpropagation*. Por ello, debemos considerar que una de las derivadas que utilizamos, es la derivada de la función de activación (por lo cual lo definimos justo después de la definición de dicha función).

Tenemos todo lo necesario para comenzar a armar nuestras capas. Por lo que, vector mediante, construiremos la topología de nuestra red, definiendo cuántas capas vamos a tener y la cantidad de neuronas que cada una de ellas va a contener. Siempre teniendo en cuenta que, en la última capa (capa de salida), vamos a necesitar tener solo 1 neurona dado que buscamos que el resultado sea binario.

Es necesario que cuando creemos la red, consideremos también la función de activación de cada una de las capas (en este caso, vamos a considerar que todas las que poseemos, posean la misma función de activación. De todas maneras, esto no es necesario pero se adoptó por simplicidad), por lo que ni bien definimos la arquitectura, definimos también dicha función.

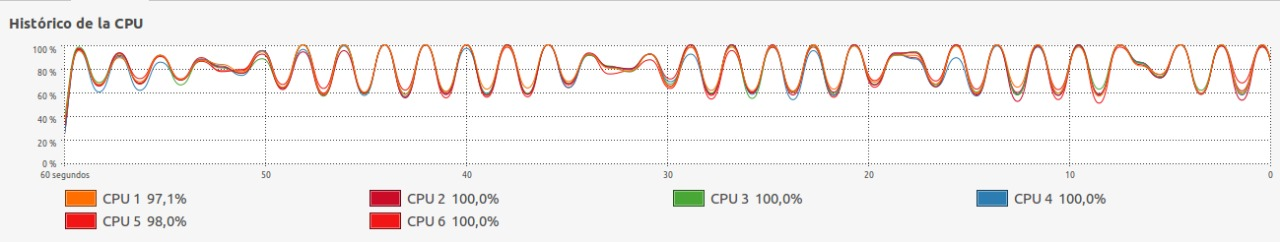
# Problemas encontrados y soluciones aplicadas

## Tasa de entrenamiento

Al hacer pruebas con topologías más complejas se observó un aumento considerable en el tiempo de entrenamiento. Se solucionó aplicando una tasa de aprendizaje variable, de manera que pueda aproximarse más rápidamente.

## Consumo de recursos

Otro problema hallado fue el consumo de recursos, particularmente CPU, para el cálculo de redes de mediana complejidad, llegando a causar un apagado por sobrecalentamiento. El problema se solucionó agregando una pausa cada una dada cantidad de iteraciones según la cantidad de datos, liberando parcialmente al procesador. Esto tiene como efecto secundario una demora adicional en el entrenamiento.



El consumo de CPU durante los ciclos de entrenamiento con pausas de 0.5s cada 300 iteraciones.

# Conclusiones

Llegado a este punto, hemos de resaltar que desarrollar una actividad donde se nos plantea la necesidad de llevar a la práctica los conocimientos adquiridos durante el cursado, fue muy interesante. Y no solo desde el punto de vista del desafío que presentó la misma, sino también que nos permitió afianzar aquellos conceptos adquiridos durante nuestro paso por la cátedra.

El mundo de la inteligencia artificial es demasiado amplio como para sólo quedarse en ejercicios con lápiz y papel, por ello la importancia que le damos a este trabajo. Fue increíble cómo lo que parecía muy simple en la hoja, llevado al código representó todo un mundo por descubrir. No podemos dejar de pensar cuán compleja puede volverse una red, y las posibilidades que puede llegar a otorgar siendo que sólo ‘*modelamos’* una red de baja complejidad.