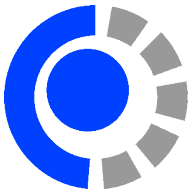


Universidad Nacional Del Comahue

Facultad de Informática

Sistemas Inteligentes

Trabajo práctico obligatorio
Redes neuronales

Alumnos:

LUSSO, Adriano Mauricio FAI-2908
adriano.lusso@est.fi.uncoma.edu.ar
TORRES, Bianca Antonella FAI-2991
bianca.torres@est.fi.uncoma.edu.ar

Profesores del curso:

Sandra Roger
Christian Gimenez

Abril 2024

Índice

1. Introducción

Las redes neuronales se han abierto paso en la ciencia de datos y la inteligencia artificial, siendo de gran utilidad en entornos académicos e industriales. Algunas de sus aplicaciones son el reconocimiento de patrones y la predicción de ciertos eventos.

A pesar de la gran perspectiva actual y a futuro que presentan, estos modelos cuentan con desafíos significativos. Entre estos, la necesidad de grandes datos para entrenamiento, riesgos tales como el *overfitting* y la compleja parametrización que requieren para obtener resultados útiles. En este contexto, este trabajo busca comparar dos implementaciones diferentes de una red neuronal, con el fin de comparar sus resultados. Una de ella, hecha a través de código Python. La otra, siendo un modelo predefinido de red neuronal que proporciona el software WEKA (Bouckaert y cols., 2010). Con esto, se busca lograr un entendimiento profundo de lo que involucra la creación y utilización de redes neuronales, así como el análisis de las diversas métricas que permiten caracterizar su rendimiento.

una implementación propia realizada en código Python

¿APA?

Recordar la estructura de la introducción: falta metodología, resultados, conclusiones y estructura del artículo.

2. Marco teórico

2.1. Redes neuronales

¡Bien! Explicación del modelo RNA.

Las redes neuronales surgen como parte de los modelos conexionistas, uno de los paradigmas alternativos a la inteligencia artificial simbólica. Estos modelos se inspiran en el funcionamiento del cerebro, a través del modelado de neuronas individuales y sus conexiones. Formalmente, se las define como una estructura en red de elementos de procesamiento simples, es decir, sus neuronas. Estas poseen un alto grado de interconectividad parametrizada, de forma que la información fluye en forma de cascada. Cada conexión entre neuronas cuenta con un peso asociado, que es un valor que indica la importancia de la información comunicada por esa conexión. El aprendizaje de información tiene lugar en la actualización de los pesos.

2.2. Elementos del modelo

El primer elemento, ya introducido, es la neurona. En la figura 1, puede verse su estructura básica. Es la unidas básica de procesamiento, encargada de recibir, procesar y transmitir la información. Las neuronas tienen embebidas en su funcionamiento una función de activación. La misma recibe de entrada para el procesamiento una entrada neta y un sesgo o *bias*. La entrada neta se obtiene de promediar los datos provenientes de las conexiones de entrada de la neurona y sus respectivos pesos. El bias es un valor constante añadido a la entrada neta, que permite que la red neuronal tenga mayor flexibilidad. Haciendo uso de estas dos entrada, se utiliza la función de activación, que permite introducir no linealidad en el procesamiento de datos del modelo. Es decir, permite expresar relaciones entre variables con funciones de mayor complejidad que la dada por funciones

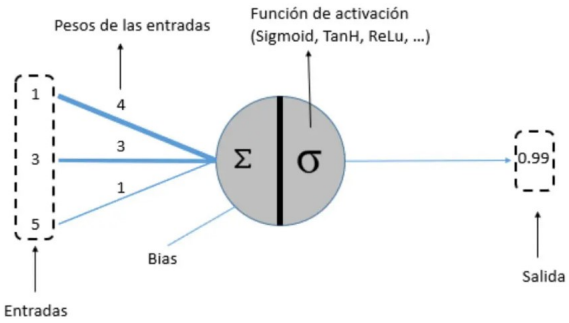


Figura 1: Modelo básico de una neurona artificial. ~~Se identifican en esta las entradas, sus pesos, el bias y su función de activación.~~

lineales. En la figura 3, pueden verse alguna de las funciones de activación mas utilizadas.

El segundo elemento, son las capas de de neuronas. Estas pueden ser capas de entrada, ocultas y de salida. La capa de entrada es la primera de una red neuronal, encargada de recibir las señales de entrada al algoritmo. Las capas ocultas son capas intermedias entre las de entrada y de salida. Extraen patrones relevantes de los datos, haciendo uso de las conexiones entre neuronas. La capa de salida es la ultima capa de la red, que produce la salida final. En la figura 2, se observa la estructura general de una red neuronal.

El tercer elemento, también ya introducido previamente, son las conexiones entre neuronas. Cada conexión tiene un peso asociado, que permite ajustar la influencia entre neuronas. Es decir, caracteriza la importancia del dato transportado por esa conexión. Los pesos son un elemento fundamental ya que son los parámetros que se ajustan durante el entrenamiento del modelo. Es decir, este elemento es el que caracteriza la simulación de aprendizaje.

El cuarto y quinto elemento son, respectivamente, la función de perdida y el algoritmo de optimización. A la hora de entrenar una red neuronal, se realizan n iteraciones, donde se ajustan los pesos de la misma. A estas iteraciones se las denomina **épocas**. En cada época, se realiza una ejecución de la red neuronal con una entrada, y se calcula la función de perdida con la salida obtenida. Esta función muestra la diferencia entre el resultado y el verdadero valor que debería haber inferido la red neuronal. La función de perdida es utilizada como la función objetivo del algoritmo de optimización, que busca minimizar la diferencia entre el valor inferido de la red neuronal y el valor real a partir de la modificación de los pesos.

Además del ajuste de pesos, el algoritmo de optimización tiene implementada una técnica denominada **retro-propagación**. Esta técnica calcula el gradiente de la función de perdida con respecto a cada peso en la red, propagándolo desde la ultima capa hacia la primera el error calculado, con el fin de que el ajuste de los pesos de las conexiones minimice el error.

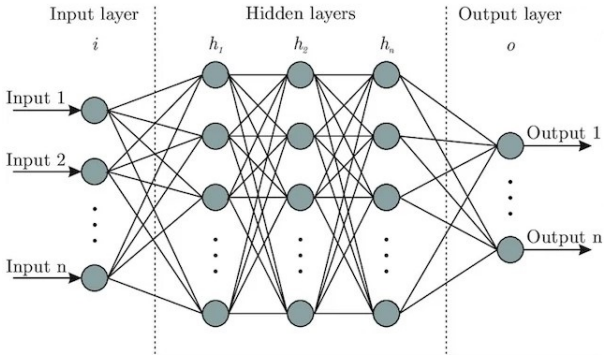


Figura 2: Modelo básico de una red neuronal, ~~donde pueden verse las capas de entrada,salida y ocultas.~~ Explicaciones en el texto.

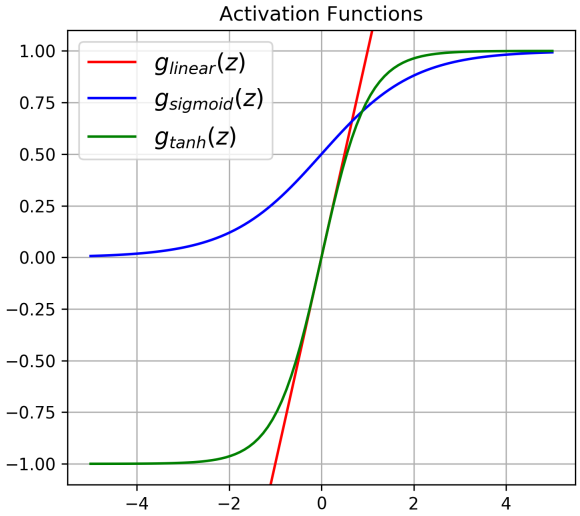


Figura 3: Gráfico de algunas de las funciones de activación más conocidas.

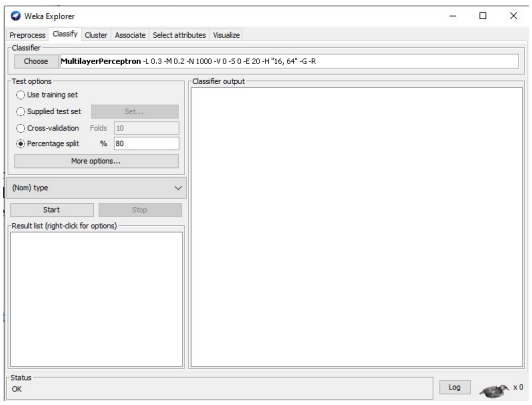
3. Explicación del data-set

Data-set utilizado

Para las pruebas desarrolladas, se utilizó un data-set de de animales, donde se especifica su nombre, ciertas características físicas y su tipo (Forsyth, 1990). Cuenta con 101 instancias caracterizadas por los siguientes atributos:

1. animal name: el nombre del animal, único por cada instancia
2. hair: si tiene pelo
3. feathers: si tiene plumas
4. eggs: si pone huevos
5. milk: si da leche
6. airborne: si puede volar
7. aquatic: si es acuático
8. predator: si es un depredador
9. toothed: si tiene dientes
10. backbone: si es vertebrado
11. breathes: si respira
12. venomous: si es venenoso
13. fins: si tiene aletas
14. legs: cuantas piernas tiene
15. tail: si tiene cola
16. domestic: si es domestico
17. catsize: si tiene el tamaño de un gato
18. type: el tipo del animal

De los atributos mencionados, el nombre del animal será descartado, ya que no aporta información relevante para la detección de patrones. De los atributos 2 al 13 y del 15 al 17, son de tipo booleano. El atributo 14 puede tomar valores del conjunto $\{0, 2, 4, 6, 8\}$. El atributo 18 puede tomar valores del rango $[1, 7]$. Para la red neuronal, se utilizarán los atributos del 2 al 17 como datos de entrada (denominados como conjunto X), y el atributo 18 será el que debe inferir la red neuronal (denominado como conjunto y).



¿Se explica cómo se separa el conjunto de entrenamiento del conjunto de test?

Figura 4: Configuración en WEKA de los conjuntos de entrenamiento y prueba

4. Detalles de la implementación

Toda la implementación se encuentra en el mismo directorio en el que este informe fue entregado.

4.1. Implementación en WEKA

Esta descripción no va acá, no es una implementación de ustedes.
Bien podría ir al marco teórico como algo que utilizaron.

El primer paso consistió en crear un archivo *attribute-relation file format* (ARFF) (Tribus, 2010) con el data-set. Este es un formato estandarizado para representar datos en función de su nombre y de su tipo, y es utilizado por WEKA para incorporar data-sets en sus diferentes algoritmos. El archivo ARFF en cuestión es `zoo_modified.arff`. Notar que existe otro archivo llamado `zoo.arff`. Este ultimo es el data-set original sin las modificaciones establecidas en la sección 3. Por lo tanto, `zoo_modified.arff` fue el utilizado para las pruebas en WEKA.

Se decidió utilizar `MultilayerPerceptron` que, como su nombre indica, es el módulo de WEKA que implementa un perceptron multicapa. Sus neuronas tienen implementadas una funcion de activacion sigmoide (Abdel-Sattar, Aboukarima, y Alnahdi, 2021). Se le configuró un ratio de aprendizaje de 0,01, y 1000 épocas de entrenamiento, además de permitir la normalización de los datos. Se dividieron los datos en un 80 % de datos de entrenamiento y un 20 % de prueba usando el el módulo de división automática que incluye WEKA. El mismo realiza la división entre conjunto de prueba y entrenamiento de manera aleatoria con el uso de una *seed*. En las figuras 4 y 5 se puede apreciar las configuraciones previamente explicadas.

semilla (/seed/)

4.2. Implementación en código

Prototipo Implementado en Python

Para la implementación en código se utilizó el lenguaje de programación Python, además de las librerías Numpy (Oliphant, 2006–) y Pandas (McKinney

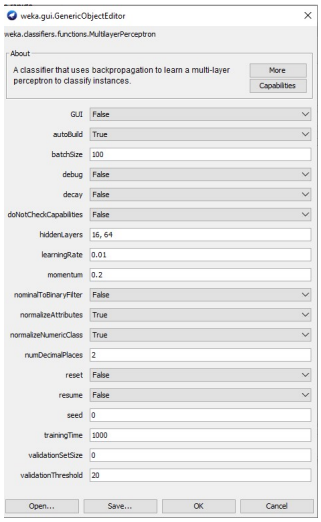


Figura 5: Configuración en WEKA de ratio de aprendizaje y otros parámetros

y cols., 2010) para un sencillo manejo de grandes estructuras de datos.

¿Cómo es la implementación? ¿Algún retazo de código que muestre lo siguiente?

- Función sigmoide
- Clase Perceptron o forma multicapa
- Diseño de las clases (un UML si se desea)
- ¿Cómo se ejecuta?

Explicar más su diseño y códigos.

Para empezar, dado los conjuntos X e y definidos en 3, se deben subdivir en X_{train}, X_{test}, y_{train} e y_{test}. Como los nombres de los subconjuntos indican, estos son los respectivos subconjuntos de entrenamiento y prueba. Esta subdivisión se hizo de manera de que los conjuntos sean lo suficientemente representativos en sus valores, tanto en X como en y. Pueden ser encontrados en los archivos `X_train.csv`, `X_test.csv`, `y_train.csv` e `y_test.csv` respectivamente. Finalmente, los datos fueron normalizados utilizando un método fuertemente inspirado en la normalización min-max (Patro y Sahu, 2015), simplificando su rango de valores a $[0, 1]$.

Luego, se definió una implementación de un perceptron multicapa en una clase con el mismo nombre. En esta, se cuenta con una capa de entrada, de salida y una única capa oculta. Tanto sus pesos como sus sesgos pueden ser inicializados tanto aleatoriamente como de manera personalizada por el usuario. Implementa una función sigmoide (Narayan, 1997) para todas sus neuronas, y como optimizador, el algoritmo de descenso de gradiente (Amari, 1993). Para el optimizador, se definió también la función sigmoide derivada, que permite el cálculo del gradiente.

Los experimentos se pueden poner en una sección aparte, porque involucra a los dos.

Primero habría que comentar un poco cómo son los pasos típicos:
1. preparación del data-set
2. entremaniento y creación del modelo
3. test del modelo
4. uso del modelo (predicción)

4.2.1. Primer experimento: inicialización aleatoria de pesos y sesgos

En principio, se decidió hacer múltiples entrenamientos de la red neuronal, sin especificar los pesos y sesgos de inicialización. De esta manera, estos serian inicializados aleatoriamente. Para esto, se configuró 1000 épocas de entrenamiento y un ratio de aprendizaje de 0,01. Tras un muestreo de 100 entrenamientos, y su consecuente verificación de accuracy, se encontró una inicialización cuyo

```
X_train,X_test,y_train,y_test = cargar_train_test()
X_train,X_test,y_train,y_test = preparar_datos(X_train,X_test,y_train,y_test)
weights_input_hidden,weights_hidden_output,bias_hidden,bias_output = importar_pesos_bias()

input_size = 16#X_train.shape[1]
hidden_size = 64
output_size = 7

perceptron = PerceptronMulticapa(input_size, hidden_size, output_size,weights_input_hidden,
                                 weights_hidden_output,bias_hidden,bias_output)
# esta inicializacion de abajo fue usada previamente. Aca, no se pasan por parametros los pesos y bias.
#Por lo tanto, se inicializan aleatoriamente.
# perceptron = PerceptronMulticapa(input_size, hidden_size, output_size)

perceptron.entrenar(X_train, y_train, epocas=1000, ratio_aprendizaje=0.01,debug=True)

predicciones = perceptron.predecir(X_test)
```

Figura 6: Celda de ejecución del segundo experimento en la implementación en código Python

entrenamiento resultaba en un rendimiento aceptable para los esperados en el marco de este trabajo. Los mismos fueron guardados en los siguientes archivos:

- `initialized_weights_input_hidden.csv`
- `initialized_weights_hidden_output.csv`
- `initialized_weights_bias_hiden.csv`
- `initialized_weights_bias_output.csv`

4.2.2. Segundo experimento: inicialización de pesos y sesgos óptimos

En este segundo experimento, se decidió usar la misma configuración que la usada para el de la sección 4.2.1, con la diferencia de que no inicializarían los pesos y sesgos aleatoriamente. Esta vez, se utilizarían aquellos encontrados en el experimento anterior. Este flujo de trabajo es el que, finalmente, quedó implementado para su ejecución. El código del experimento se encuentra en `nn_python.ipynb`, y se puede ejecutar abriendo el notebook y activando todas sus celdas desde un entorno local o desde Google Colab. En la figura 6, se ve la celda que ejecuta el experimento en sí. Esta celda hace uso de las celdas previas, donde se definen las funciones relevantes.

5. Resultados

Sobre el experimento en WEKA, se logró de forma consistente un un accuracy de 60 %, cuya matriz de confusión asociada es

$$\begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \end{pmatrix} \tag{1}$$

donde un elemento en (i, j) indica que fue clasificado como de categoría i siendo realmente de categoría j, tal que $i, j \in [1, 7]$. En los archivos `weka_nn_result` y `weka_nn.model` se encuentran el log de resultado y los parametros optimizados respectivamente.

Sobre el experimento en código, los resultados del entrenamiento con inicialización aleatoria eran mucho menos consistentes. Los mismos podían resultar en un accuracy desde el 4 % hasta el 47 %. Una vez que se inicializaba el modelo con los parámetros que lograban optimizar hacia un 47 %, el accuracy se mantenía. Su matriz de densidad es:

$$\begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{2}$$

6. Discusiones

En principio, se ve una mejora importante en términos de accuracy del modelo WEKA sobre el modelo en código. Esto es algo esperable, ya que es probable que WEKA realice otro tipo de procesamiento y técnicas más complejas que no fueron implementadas en el código. Es interesante marcar la gran diferencia en estabilidad de optimización en los modelos. Como se menciono en la sección 5, el modelo en código tenía alta variabilidad en los accuracy conseguidos en diferentes optimizaciones a comparación del modelo WEKA, incluso aunque ambos tuvieran el mismo tipo de inicialización aleatoria de conjuntos de prueba y entrenamiento. No se cree que sea un error en la representabilidad de los datos en la creación del modelo en código. Para esto, se uso la función `train_test_split(X, y, test_size=0.2, stratify=y)`. Lo importante esta en el atributo `stratify=y`, que toma todos los valores posibles en los atributos de y y garantiza una buena proporción de representación de los mismos en la prueba y el entrenamiento.

7. Conclusiones

Se realizó una comparación exitosa de el perceptron multicapa implementando en WEKA contra una implementación más simple del mismo modelo en código Python. Se compararón los resultados y de detectaron posibles causas a ciertos problemas de performance en la implementación en Python. Como futura mejora, se plantea mejorar la implementación en Python para permitir multiples capas ocultas.

Referencias

Abdel-Sattar, M., Aboukarima, A. M., y Alnahdi, B. M. (2021). Application of artificial neural network and support vector regression in predicting mass of ber fruits (ziziphus mauritiana lamk.) based on fruit axial dimensions. *Plos one*, *16*(1), e0245228.

Amari, S.-i. (1993). Backpropagation and stochastic gradient descent method. *Neurocomputing*, *5*(4-5), 185–196.

Bouckaert, R. R., Frank, E., Hall, M., Kirkby, R., Reutemann, P., Seewald, A., ... others (2010). Weka manual for version 3-6-2. *The University of Waikato*.

Forsyth, R. (1990). *Zoo*. UCI Machine Learning Repository. (DOI: https://doi.org/10.24432/C5R59V)

McKinney, W., y cols. (2010). Data structures for statistical computing in python. En *Proceedings of the 9th python in science conference* (Vol. 445, pp. 51–56).

Narayan, S. (1997). The generalized sigmoid activation function: Competitive supervised learning. *Information sciences*, *99*(1-2), 69–82.

Oliphant, T. (2006–). *NumPy: A guide to NumPy*. USA: Trelgol Publishing. Descargado de `http://www.numpy.org/` ([Online; accessed ¡today¿])

Patro, S., y Sahu, K. K. (2015). Normalization: A preprocessing stage. *arXiv preprint arXiv:1503.06462*.

Tribus, H. (2010). *Static code features for a machine learning based inspection: An approach for c.*