**INVESTIGACIÓN POR INTERNET DE EJEMPLO DE UNA RED NEURONAL**

Referencia: Esta página que encontré navegando por la web en busaca de algo parecido, me pareció muy completa, dentro de ignorancia en el tema, supuse que estaba muy completo, ya que se toman la molestia de explicar con detalle cada proceso.

<https://www.aprendemachinelearning.com/una-sencilla-red-neuronal-en-python-con-keras-y-tensorflow/>

La página tiene como título “Aprende Machine Learning”, Este nos da una bienvenida a un trabajo hecho con Python, keras y tensorflow.

Python es un lenguaje de programación orientado a objetos, de fácil uso y aprendizaje rápido. Aunque también es multiparadigma y multiplataforma.

Keras es una librería para Python la cual es de alto nivel, que ayudara a realizar las capas de red de una forma más sencilla.

Tensorflow es una biblioteca de código abierto de aprendizaje automatico, creado por google. La cual es la mejor herramienta para hacer Deep learning (aprendizaje profundo).

Realiza una breve explicación de la compuerta Xor, la cual va a ser el ejemplo de cómo se pueden emular compuertas básicas usando redes neuronales.

Python:

import numpy as np

from keras.models import Sequential

from keras.layers.core import Dense

# cargamos las 4 combinaciones de las compuertas XOR

training\_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")

# y estos son los resultados que se obtienen, en el mismo orden

target\_data = np.array([[0],[1],[1],[0]], "float32")

model = Sequential()

model.add(Dense(16, input\_dim=2, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

model.compile(loss='mean\_squared\_error',

              optimizer='adam',

              metrics=['binary\_accuracy'])

model.fit(training\_data, target\_data, epochs=1000)

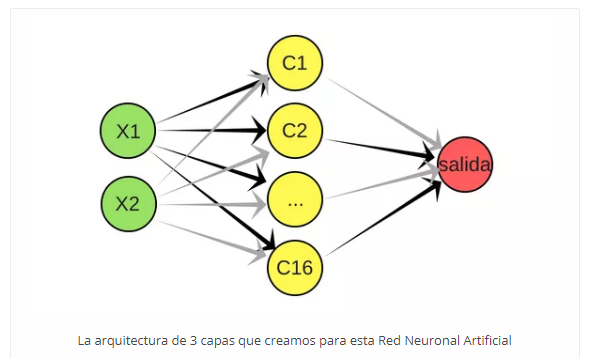
# evaluamos el modelo

scores = model.evaluate(training\_data, target\_data)

print("\n%s: %.2f%%" % (model.metrics\_names[1], scores[1]\*100))

print (model.predict(training\_data).round())

Description:

1. Primero que todo se usaran numpy para el manejo de arrays. De Keras importamos el tipo de modelo Sequential y el tipo de capa Dense que es la “normal”. Y se crean los arrays de entrada y salida, cada de una son 4 elementos de tipo flotante.
2. Después se crea un modelo vacío de tipo Sequential. Este modelo se refiere a que crearemos una serie de capas de neuronas secuenciales, una detrás de la otra.
3. Agregamos dos capas Dense con «model.add()». Realmente serán 3 capas, pues al poner input\_dim=2 estamos definiendo la capa de entrada con 2 neuronas (para nuestras entradas de la función XOR) y la primer capa oculta (hidden) de 16 neuronas. Como función de activación utilizaremos «relu» que sabemos que da buenos resultados. Podría ser otra función, esto es un mero ejemplo, y según la implementación de la red que haremos, deberemos variar la cantidad de neuronas, capas y sus funciones de activación.
4. 
5. Ahora toca entrenar la red. Para esto se indican el tipo de perdida (loss) que se va a utilizar, se optimizan los pesos de las conexione de la neurona y las métricas que queremos obtener. La línea de model.fit…, se indica que las entradas, salidas y la cantidad de iteraciones de aprendizaje de entrenamiento.
6. Después se revisa el resultado del entrenamiento:
   1. Epoch 1/1000
   2. 4/4 [==============================] - 0s 43ms/step - loss: 0.2634 - binary\_accuracy: 0.5000
   3. Epoch 2/1000
   4. 4/4 [==============================] - 0s 457us/step - loss: 0.2630 - binary\_accuracy: 0.2500

Con esto vemos que la primera iteración “tuvo algo de suerte” y acierto la mita de las salidas pero a partir de la segunda, solo acierto de 1 de cada 4. Luego de la epoch 24 recupera el 0.5 de aciertos, en este caso ya no es por suerte, sino por acomodar bien los pesos

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Epoch 24/1000  4/4 [==============================] - 0s 482us/step - loss: 0.2549 - binary\_accuracy: 0.5000    Epoch 107/1000  4/4 [==============================] - 0s 621us/step - loss: 0.2319 - binary\_accuracy: 0.7500    Epoch 169/1000  4/4 [==============================] - 0s 1ms/step - loss: 0.2142 - binary\_accuracy: 1.0000 |
|  |  |

1. Ahora se configura la evaluación y se prueban las predicciones, ya que se siente preparado para trabajar. Al evaluar los modelos, se aprecia que se tuvo una precisión del 100%.
2. Se cambian los parámetros de la red. Se debe de tener en cuenta que este es un ejemplo muy sencillo y que cuenta con solo 4 entradas posibles. Pero si en la realidad tuviéramos una red compleja, deberemos poder ajustar muchos más parámetros.

La cantidad de rede en nuestro caso es 3. La cantidad de neuronas en cada red (tenemos 2 de entrada, 16 en capa oculta y 1 de salida). Las funciones de activación de cada capa. Utilizamos relu y sigmoid. Al compilar el modelo definir las funciones de pérdida, optimizer y métrica.

1. Se guarda la red creada y la ponemos a trabajar: De ser esto un caso real, en el cual entrenamos una red, la ajustamos y obtenemos buenos resultados, ahora deberíamos guardar esa es ya que esa red optima, tiene los pesos que estamos buscando. Sería tonto y lento entrenar cada vez la red antes de “publicar en producción”.

Con el siguiente código se hará el trabajo de “ponerla a funcionar” y usar normalmente loadded\_model.predict():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | # 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!")    # mas tarde...    # cargar json y crear el modelo  json\_file = open('model.json', 'r')  loaded\_model\_json = json\_file.read()  json\_file.close()  loaded\_model = model\_from\_json(loaded\_model\_json)  # cargar pesos al nuevo modelo  loaded\_model.load\_weights("model.h5")  print("Cargado modelo desde disco.")    # Compilar modelo cargado y listo para usar.  loaded\_model.compile(loss='mean\_squared\_error', optimizer='adam', metrics=['binary\_accuracy']) |

Conclusiones: ¿Vale la pena usar una red neuronal en comparación con un típico if-then-else?

En este ejemplo puede que sea mucho más sencillo aplicar un código que se base en condicionales. Pero en el caso donde sean mayor cantidad de elementos, que en inteligencia artificial se traduce en mayores redes neuronales, sería demasiado caótico como para hacerlo “a pedal”. Esta herramienta de predicciones nos facilita estas tareas.