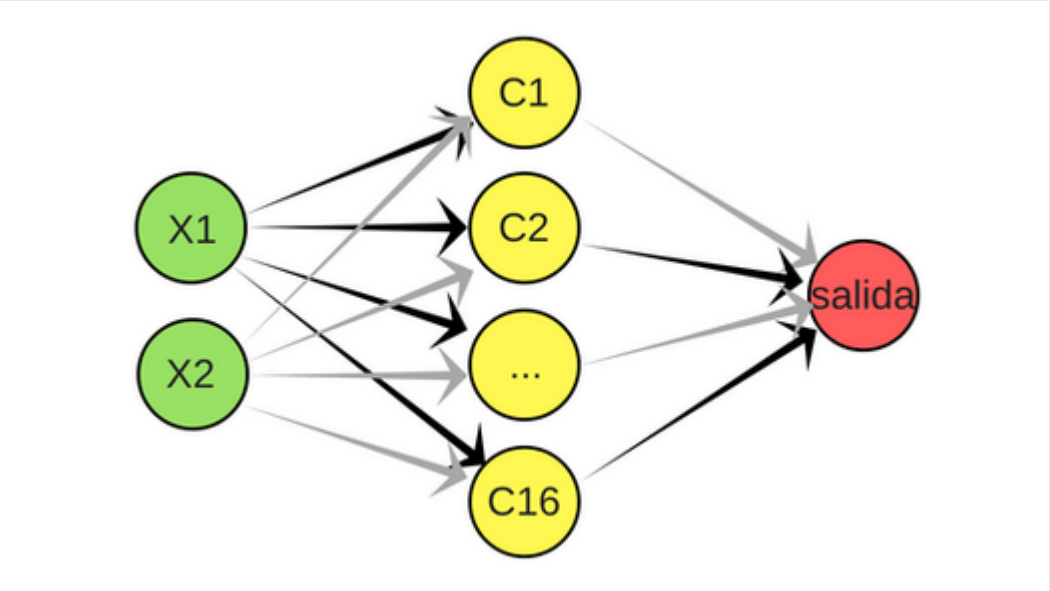
ML_CLASIFICACION_RN_05	Red Neuronal utilizando la librería Keras	ML

En esta práctica se utiliza la librería 'Keras' para entrenar y testar la red neuronal. En el ejemplo tenemos dos entradas ('inputs' X1 y X2) binarias (1 ó 0) y la función de evaluación (compuertas XOR) dan como salida 1 sólo si una entrada es verdadera (1) y la otra falsa (0). Existen cuatro combinaciones posibles (XOR(0,0) = 0, XOR(0,1) = 1, XOR(1,0) = 1, XOR(1,1) = 0).

Utilizamos *Keras* que es una librería de alto nivel, para que nos sea más fácil describir las capas de la red que creamos y en background, es decir el motor que ejecutará la red neuronal y la entrenará, estará la implementación de Google llamada *Tensorflow*, que es la mejor que existe hoy en día.



Esquema de la red neuronal. (2 inputs, 16 nodos ocultos, 1 ouput)

SOLUCIÓN

Importar las librerías necesarias para realizar la práctica.

```
# Librerías
import numpy as np
print('numpy: %s' % np.__version__)
# de la librería Keras importar el tipo de modelo Sequential y el tipo de capa
Dense (la más normal)
from keras.models import Sequential
from keras.layers.core import Dense
```

Crear los arrays de entrada y salida

```
# Crear los arrays de entrada y salida
# compuertas XOR. Cuatro entradas [0,0], [0,1], [1,0],[1,1] y sus salidas:
0,1,1,0.
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# y las salidas, en el mismo orden
target_data = np.array([[0],[1],[1],[0]], "float32")
```

Crear la arquitectura de la red neuronal. Compilar y ajustar

```
# Crear la arquitectura de la red neuronal
# Utilizar un modelo de tipo 'Sequential' para crear capas secuenciales, "una
delante de otra"
# Agregamos las capas Dense: entrada con 2 neuronas (XOR), capa oculta (16
neuronas)
# Función de activación utilizar "relu" que da buenos resultados
model = Sequential()
model.add(Dense(16, input_dim=2, activation='relu'))
# Agregamos la capa de salida con función de activación 'sigmoid'
model.add(Dense(1, activation='sigmoid'))

# Definir el tipo de pérdida (loss) a utilizar, el "optimizador" de los pesos
de las conexiones
# de las neuronas y las métricas a obtener
model.compile(loss='mean_squared_error',optimizer='adam',metrics=['binary_accu
racy'])

# Entrenar la red neuronal
# Con 1000 iteraciones de aprendizaje (epochs) de entrenamiento
model.fit(training_data, target_data, epochs=1000)
```

```
Epoch 997/1000
4/4 [==============================] - 0s 4ms/step - loss: 0.0080 -
binary_accuracy: 1.0000
Epoch 998/1000
4/4 [==============================] - 0s 4ms/step - loss: 0.0080 -
binary_accuracy: 1.0000
Epoch 999/1000
4/4 [==============================] - 0s 4ms/step - loss: 0.0080 -
binary_accuracy: 1.0000
Epoch 1000/1000
4/4 [==============================] - 0s 4ms/step - loss: 0.0080 -
binary_accuracy: 1.0000
4/4 [==============================] - 0s 0us/step
```

Evaluar la red neuronal

```
# Evaluamos el modelo
scores = model.evaluate(training_data, target_data)
```

Predecir

```
# Métricas
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
print (model.predict(training_data).round())
```

```
binary_accuracy: 100.00%
[[0.]
 [1.]
 [1.]
 [0.]]
```