Perceptron Multicapa - Keras XOR

May 15, 2019

1 Importamos las librerias necesarias

```
In [1]: from keras.models import Sequential
        from keras.layers import Dense
        import os
        import tensorflow as tf
        import numpy
```

```
Using TensorFlow backend.
```

2 Definimos la clase Keras

```
In [2]: class Keras:
            """
            Perceptron Multicapa
            Codificación de un perceptron multicapa capaz de aprender
            la compuerta XOR mediante la libreria Keras y TensorFlow.
            """
            def __init__(self, seed = 7):
                """
                Metodo constructor de la red,
                inicialza los valores por defecto.

                Parametros:
                    seed: int
                        Fija las semillas aleatorias para la reproducibilidad,
                        con el objetivo de tener la misma salida siempre.
                """
                numpy.random.seed(seed)

                # Inicializa el modelo para la red.
                self.model = Sequential()

                # Datos de entrada y salida
                self.entrada = []
```

```
def cargarData(self, FICHERO):
    """
    Metodo destinado a cargar los datos en la red del set de datos.
    """
    # Carga el set de datos de la compuerta XOR de FICHERO.
    dataset = numpy.loadtxt(FICHERO, delimiter=',')

    # Separa las entrada y salidas.
    self.entrada = dataset[:, 0:2]
    self.salidas = dataset[:, 2]

def cargarModelo(self):
    """
    Metodo destinado a cargar el modelo que se desea en la red:
    Utilizando la clase Dense, se puede anidar varias capas para el
    perceptron, el primer parametro determina las neuronas que se
    utilizaran, el segundo la cantidad de datos de entrada, y el
    tercero la funcion de activacion. Finalmente, un perceptron de tres
    capas con numero de neuronales variables por capa y distintas funciones
    de activacion.
    """
    # Capa 1.
    # 12 neuronas, 2 datos de entrada, funcion de activacion: rectificador.
    self.model.add(Dense(12, input_dim=2, activation='relu'))

    # Capa 2.
    # 2 neuronas, funcion de activacion: rectificador
    self.model.add(Dense(2, activation='relu'))

    # Capa 3.
    # 1 neurona, funcion de activacion: Sigmoid
    self.model.add(Dense(1, activation='sigmoid'))

def compilarModelo(self):
    """Metodo encargado de compilar el modelo previamente diseñado"""

    # Debemos especificar la función de pérdida para un conjunto de pesos,
    # el optimizador para buscar a través de diferentes pesos para la red
    # y cualquier métrica opcional que nos gustaría recopilar y reportar
    # durante el entrenamiento.

    # En este caso, utilizaremos la pérdida logarítmica,
    # que para un problema de clasificación binaria se define en Keras
    # como binary_crossentropy. Algoritmo de descenso de gradiente
    # adam por su alta eficiencia en estos problemas, y "accuracy"
    # como la métrica, totalmente a criterio del diseñador.
    self.model.compile(loss='binary_crossentropy',
```

```
                              optimizer='adam',
                              metrics=['accuracy'])

    def entrenamiento(self):
        """Metodo encargado de entrenar la red."""

        # El siguiente metodo utiliza los datos de entrada y salida previamnete
        # cargados para entrenar la red, "epochs" representa el numero de
        # iteraciones que tendra para entrenar, "batch_size" representa el
        # numero de iteraciones antes de actualizar los pesos de las capas, y
        # "verbose" representa las salidad por pantalla de cada iteracion del
        # entramiento (Nota: en caso de querer estudiar las salidad por
        # iteracion eliminar el ultimo parametro).
        self.model.fit(self.entrada,
                       self.salidas,
                       epochs=1000,
                       batch_size=10,
                       verbose=0)

    def evaluador(self):
        """Metodo encargado de evaluar los parametro de la red."""

        # Evalua las metricas de red con los datos de entrada y salida.
        scores = self.model.evaluate(self.entrada, self.salidas)

        # Imprime el porcentaje de acierto que tiene la red.
        print("%s: %.2f%%" % (self.model.metrics_names[1], scores[1] * 100))

    def respuesta(self):
        """
        Salida de la red, aplica metodo de prediccion luego del entrenamiento.
        """
        # Calcula el resultado con datos de entrada luego del entrenamiento.
        predicciones = self.model.predict(self.entrada)

        # Redondeamos las predicciones
        redondeo = [round(x[0]) for x in predicciones]
        print("Salida de la red:", redondeo)

        # Descomentar la siguiente linea en caso de querer observar los
        # datos reales de prediccion, sin el redondeo.
        # print(predicciones)
```

Se hara un enfasis en el metodo del entrenamiento:

Notese los parametros "epochs" y "batch_size", los mismos son muy importantes en esta ocasion, ya que sus valores depende plenamente del diseñador de la red, mediante ensayo y error, estos valores son los que ayudan finalmente a que la red termine aprendiendo correcta o incorrectamente con los datos de entradas.

3 Main principal para probar la red

```
In [3]: if __name__ == "__main__":
            # Eliminacion de mensajes de advertencia de TensorFlow y Keras.
            os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
            tf.logging.set_verbosity(tf.logging.ERROR)

            # Ruta del fichero con la data de entrada.
            FICHERO = 'data/data-xor.csv'

            # Se inicializa la red.
            k = Keras()

            # Se cargan los datos de entrada.
            k.cargarData(FICHERO)

            # Se carga el modelo del perceptron y se compila.
            k.cargarModelo()
            k.compilarModelo()

            # Se obtiene la respuesta de la red sin entrenar
            k.respuesta()
            print("")

            # Se entrena la red para la compuerta XOR.
            k.entrenamiento()

            # Se evalua el porcetanje de acierto de la red.
            k.evaluador()

            # Se obtiene la respuesta de la red luego de entrenarla.
            k.respuesta()
```

```
Salida de la red: [0.0, 0.0, 0.0, 0.0]

4/4 [==============================] - 0s 15ms/step
acc: 100.00%
Salida de la red: [0.0, 1.0, 1.0, 0.0]
```

La primera salida de la red, es de cuando aún, la misma, no se ha entrenado, por lo que resulta como solo ceros, mientras que después de entrenarla, podemos observar en la respuesta de evaluarla como el "acc" o la métrica "accuracy" que difinimos anteriormente en el modelo, nos muestra un "100.00%", esto representa el porcentaje de acierto que tiene la red.

Finalmente, la ultima salida de la red representa la prediccion de la misma, una vez entrenada con los datos de entrada, resultando la compuerta XOR:

```
[x1, x2] -----> [s]
  [0, 0] -----> [0]
```

```
[0, 1] -----> [1]
[1, 0] -----> [1]
[1, 1] -----> [0]
```