Perceptron Multicapa - Keras

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 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 = []
                self.salidas = []
```

```
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[:, 0:8]
    self.salidas = dataset[:, 8]

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, 8 datos de entrada, funcion de activacion: rectificador.
    self.model.add(Dense(12, input_dim=8, activation='relu'))

    # Capa 2.
    # 8 neuronas, funcion de activacion: rectificador
    self.model.add(Dense(8, 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=150,
                       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/pima-indians-diabetes.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, descomentar
            # en caso de querer ver toda la salida.
            #k.respuesta()
            #print("")

            # Se entrena la red para los datos de personas diabeticas.
            k.entrenamiento()

            # Se evalua el porcetanje de acierto de la red.
            print("Porcentaje de acierto despues de entrenar:")
            k.evaluador()

            # Se obtiene la respuesta de la red luego de entrenarla, descomentar
            # en caso de querer ver toda la salida desdpues de entrenarla.
            #k.respuesta()
```

```
Porcentaje de acierto despues de entrenar:
768/768 [==============================] - 0s 129us/step
acc: 78.52%
```

Podemos observar en la respuesta de evaluarla como el "acc" o la métrica "accuracy" que difinimos anteriormente en el modelo, nos muestra un "78.52%", esto representa el porcentaje de acierto que tiene la red.

Para ver los datos de salida antes y depues del entrenar, descomentar las lineas 21, 22 y 33.