Modelos conexionistas © EDICIONES ROBLE, S.L.

Indice

Modelos conexionistas

I. Introducción

Uno de los términos que más de moda se ha puesto en los últimos años dentro del aprendizaje automático es *Deep Learning* (aprendizaje profundo). En muchas presentaciones de nuevos servicios basados en el uso de Inteligencia Artificial es habitual escuchar referencias a esta expresión. Gracias a estas tecnologías los ordenadores pueden realizar tareas que hasta hace pocos años se consideraban exclusivas de los humanos. Actualmente es posible encontrar sistemas de traducción automática, reconocimiento de voz, asistentes personales e identificación de personas, objetos o, incluso, lugares solamente a partir de imágenes.

El término *Deep Learning* hace referencia al conjunto de algoritmos que se han desarrollado para el entrenamiento de redes neuronales profundas, también conocidos como sistemas conexionistas. Las redes neuronales profundas son aquellas que tienen múltiples capas ocultas, entendiéndose por capas ocultas aquellas que no forman parte de la entrada y salida del sistema, es decir, aquellas capas que no reconocen directamente los datos de entrada ni de salida. En los últimos años, el estudio de las redes neuronales se ha desarrollado gracias a la mejora en los algoritmos para el entrenamiento de redes profundas, al aumento de capacidad de cálculo de los ordenadores y a la ampliación de los conjuntos de datos necesarios para el entrenamiento de estos sistemas.

Las redes neuronales, como su propio nombre indica, pretenden reproducir el comportamiento de las neuronas del cerebro. Se utilizan para ello sistemas sencillos conectados entre sí. Aunque los primeros estudios de redes neuronales datan de 1940, este estudio no ha gozado de la relevancia que ahora tiene hasta hace unos pocos años. Esto se debe al coste computacional que conlleva el entrenamiento de redes profundas y a la necesidad de datos masivos para evitar la aparición de sobreajuste.

En esta unidad se presentará una introducción a los sistemas conexionistas o redes neuronales artificiales. Se comenzará por el concepto de perceptrón, la red neuronal artificial más básica, hasta llegar a la creación de un sistema que permita el reconocimiento de números escritos a mano en un conjunto de imágenes.

Para trabajar con redes neuronales, se utilizará una librería de Python que permite la escritura de código simbólico Theano. Esta librería requiere un aprendizaje previo, ya que dispone de una sintaxis propia. Una vez entendida, la implementación de redes neuronales en Python se convierte en una tarea más sencilla que, además, requiere de menos líneas de código.

Esta unidad se complementa con un notebook Python en el que se incluye todo el código utilizado y algunos ejemplos adicionales a los que se hace referencia en el texto. Es aconsejable seguir el texto con este notebook U6_Códigos

II. Objetivos

Los objetivos que los alumnos alcanzarán tras el estudio de esta unidad son:

- Comprender el concepto de perceptrón.
- Conocer la sintaxis básica de Theano para la construcción de una red neuronal.
- Saber entrenar una red neuronal utilizando propagación hacia atrás (*backpropagation*).
- Conocer los principales tipos de neuronas disponibles en Theano.
- Comprender el algoritmo de dropout para la regularización de las redes neuronales.

III. Perceptrones

El término perceptrón, dentro del campo del aprendizaje automático, hace referencia a dos conceptos: uno de los primeros modelos de redes neuronales desarrollado por Frank Rosenblatt en 1957 y al modelo matemático básico de una única neurona. En esta unidad, se utilizará esta segunda acepción.

La idea de perceptrón se muestra en el esquema que aparece en la figura 5.1. En esta figura se aprecia un objeto (a) que recibe dos entradas (x_1 y x_2), además de un término constante que se denomina bias. Estas entradas se multiplican por unos pesos para las entradas (w_1 y w_2) y por otro para el bias (b) para obtener un valor de entrada en el perceptrón. El perceptrón recoge estos valores de entrada y los multiplica por los pesos para obtener un nuevo valor al que se le aplica una función de transferencia. El valor de salida de esta función es la respuesta de la neurona.

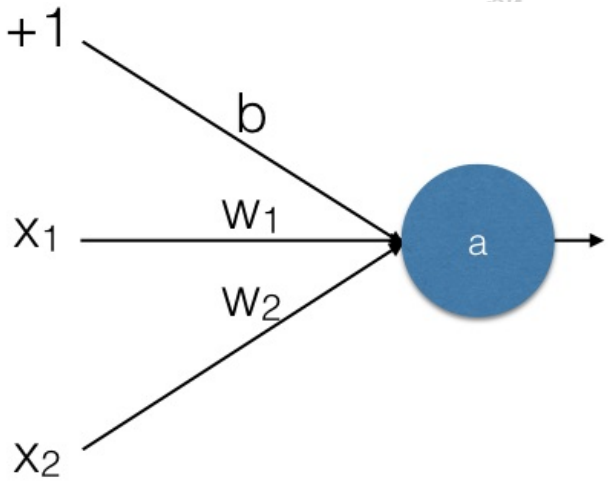


Figura 5.1. Perceptrón. *Fuente:* elaboración propia.

1

Matemáticamente, los parámetros de entrada se pueden escribir como un vector de la siguiente forma:

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

2

De igual manera los pesos:

$$\boldsymbol{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix}$$

3

De modo que la entrada al perceptrón se puede calcular mediante un producto vectorial de estos dos:

$$z = w^t x = b + \sum_{i=1}^{m} w_i x_i$$

4

En el perceptrón existe una función de activación, a(z), que opera sobre el valor de entrada calculado previamente. Una de las funciones de activación más sencillas es la función escalón, la cual se define como 0 cuando el valor de activación es negativo y 1 cuando es cero o positivo, es decir:

$$a(z) = \begin{cases} 1 & si\ z \geq 0 \\ 0 & si\ z < 0 \end{cases}$$

5

Otra función de activación muy utilizada es la logística, estudiada en la unidad 2, dentro de los modelos supervisados. La función de activación queda definida de la siguiente forma:

$$a(z) = \frac{1}{1 + e^{-z}}$$

6

El perceptrón así definido se puede utilizar para la resolución de problemas de clasificación simples, por ejemplo, para la creación de un operador lógico tipo AND. Posteriormente, se verá cómo entrenar un modelo, pero, de momento, se puede conseguir esta puerta simplemente utilizando como pesos los valores b = -1,5, w_1 = 1 y w_2 =1. La comprobación de que se ha implementado el operador lógico AND mediante esta configuración se puede ver en la tabla 5.1., donde se muestran los resultados.

x_1	x_2	AND	z	a(z)
0	0	0	-1,5	≈0
0	1	0	-0,5	≈0
1	0	0	-0,5	≈0
1	1	1	0,5	≈1

Tabla 5.1. Implementación del operador lógico AND con un perceptrón.

En el planteamiento del perceptrón se ha visto que la forma óptima de presentar las redes neuronales es mediante la utilización del álgebra matricial. Este tipo de operaciones no son fáciles de implementar en un lenguaje de propósito general como es Python, por lo que es aconsejable la utilización de otros lenguajes específicos. En esta unidad se va a utilizar Theano para tal propósito.

3.1. Introducción a Theano

Para la implementación de redes neuronales en Python, se va a utilizar la librería Theano. Este paquete no se encuentra instalado por defecto en Anaconda, por lo que es necesario hacerlo por separado. Para esto se ha de ejecutar el siguiente comando en la terminal del sistema:

```
pip install Theano
```

Para verificar que Theano ha sido instalado se ha de referenciar la librería en un notebook, en caso de que Theano ya se encuentre instalado, el siguiente comando debería funcionar:

```
import theano
```

En caso contrario, aparecería un error de librería no encontrada.

En Theano, los programas son compilados a C para aumentar su rendimiento, por lo que es aconsejable contar con un compilador en el sistema. En caso de que no exista o no pueda ser encontrado, las funciones no serán compiladas y su rendimiento será bastante pobre, especialmente en los ejemplos finales de esta unidad.

Descripción Theano

Theano es un paquete de software que permite escribir código simbólico en Python y compilarlo para mejorar el rendimiento. Fue desarrollado por investigadores de aprendizaje automático de la Universidad de Montreal. Aunque su uso no se limita únicamente al desarrollo de redes neuronales fue creado pensando en este tipo de problemas, por lo que facilita la escritura de código. En este apartado se verá la sintaxis básica de Theano, necesaria para comprender los códigos de las secciones siguientes.

Un componente importante de Theano son los tensores, por convenio se suelen importar dentro de Python con el alias T. Por lo que las primeras líneas en un programa escrito con Theano suelen ser:

```
import theano

import theano.tensor as T
```

A la hora de escribir código simbólico, las variables son el bloque básico de la construcción de algoritmos. En Theano, las variables son siempre tensores. Un tensor es la forma generalizada de una matriz de dimensión T. En base al número de dimensiones del tensor, se dice que pueden ser escalares para T = 0, vectores para T = 1 y matrices cuando T = 2.

Cuando se define una matriz en Theano se ha de indicar el tipo, para esto existen las siguientes funciones dentro de T:

- scalar
- vector
- matrix
- row: matriz tipo fila.
- col: matriz tipo columna.
- tensor3
- tensor4

Los constructores pueden recibir dos parámetros opcionales:

- name: es una cadena de texto para identificar el objeto.
- dtype: indica el tipo de valor del tensor (doble, entero, etc.).

También existen constructores con el mismo nombre que los anteriores que comienzan con un modificador que identifica el tipo de dato que contendrá el tensor. Por ejemplo, los de tipo doble comienzan por d (dvector, dmatrix) y los enteros por i (ivector, imatrix).

Operaciones con Theano

Cuando se ha creado una variable en Theano puede ser utilizada para la realización de operaciones matemáticas empleando la sintaxis estándar de Python. Esto es, se puede sumar, multiplicar, dividir, etc. Por ejemplo, para elevar al cuadrado un número se puede utilizar la siguiente fórmula:

```
x = T.scalar('x')

y = x ** 2
```

Se ha creado un escalar (x) y posteriormente se ha elevado este escalar al cuadrado.

En las líneas anteriores ni la variable x ni la variable y tienen un valor asignado, como ya se ha comentado. Theano es un lenguaje simbólico, por lo que solamente se han representado las operaciones, es decir, se ha definido una función matemática. Para obtener un valor se tienen que evaluar indicando los valores en un par clave-valor encerrado en llaves, donde la clave es el nombre de la variable y el valor es el número a evaluar. Para evaluar el ejemplo anterior se puede escribir:

```
y.eval({x:3})
```

Devuelve 9, el resultado de 3 elevado al cuadrado. También se pueden realizar operaciones más complejas en Theano como las que se muestran en la figura 5.2., donde el resultado obtenido es 32.

```
x = T.scalar('x')
y = T.scalar('y')
z = 2 * x + 3 * y

z.eval({x: 1, y : 10})
```

Figura 5.2. Operaciones con Theano.

Utilización de funciones Python en Theano

El método eval de las variables no es la forma habitual de ejecutar operaciones en Theano, para lo que se suelen crear funciones que pueden ser más complejas. Las funciones se crean utilizando el método theano.function, al que hay que indicarle las variables de entrada (inputs) y las variables de salida (outputs) de las funciones. Por ejemplo, la operación de la figura 5.2. se puede definir mediante la función:

```
f = theano.function(inputs=[x, y], outputs= z)
```

Se ha indicado que hay dos variables de entrada ([x, y]) y una de salida (z). La relación entre las variables es la que se ha definido previamente, cuando se ha creado z. A partir de aquí se puede llamar a la función f() con dos parámetros para obtener el resultado de la operación, es decir, con f(1, 2) se obtendrá 8. Esta forma de evaluar las operaciones en Theano es más cómoda que utilizar directamente el método eval.

De la misma manera que las funciones de Theano pueden tener varias variables de entrada, también se pueden indicar varias variables de salida y obtener así varios resultados al ejecutar una única función. Por ejemplo, obtener el valor de z y k se puede escribir del siguiente modo:

```
f = theano.function(inputs=[x, y], outputs= [z, k])
```

De esta forma, se obtiene un vector con los resultados para estas dos variables.

Es importante destacar que en los inputs se han de indicar todas las variables necesarias para el cálculo de las variables de salida. En caso de que esto no sea así se obtendría un error de compilación.

Para facilitar la escritura de código se pueden utilizar funciones de Python para la creación de variables Theano. Esto se puede ver en el código de la figura 5.3., donde se han creado dos variables (x, y) y se suman en una función sum_vars que se asigna a una nueva variable Theano. Posteriormente se crea una función y se evalúa, de lo que se obtiene el valor esperado: 3.

```
x = T.scalar('x')
y = T.scalar('y')

def sum_vars(x, y):
    return x + y

z = sum_vars(x, y)

f = theano.function(inputs = [x, y], outputs = z)
f(1, 2)
```

Figura 5.3. Utilización de funciones Python en Theano.

La utilización de funciones no se limita únicamente a las que cree el usuario, sino que pueden ser utilizadas la mayoría de las funciones matemáticas existentes en la librería de Python. Por ejemplo, en la figura 5.4. se muestra el uso de la función trigonométrica cos para la definición de la variable que será utilizada en una función.

```
x = T.scalar('x')
y = cos(x)

f = theano.function(inputs = [x], outputs = y)

f(0)
```

Figura 5.4. Utilización de funciones de librería de Python en Theano.

Control de flujo en Theano

En las funciones que se creen para ser utilizadas con Theano no pueden usarse operadores lógicos con las variables, por lo que no se puede crear controles de flujo utilizando los estándares de Python. Para esto se han de utilizar los que suministra Theano, como la función T.switch y las funciones de comparación. Por ejemplo, en la figura 5.5., se muestra cómo construir una función valor absoluto.

```
x = T.scalar('x')

y = T.switch(T.gt(x, 0), x, -x)

f = theano.function(inputs = [x], outputs = y)

print "abs(3) = ", f(3)
print "abs(-3) = ", f(-3)
```

Figura 5.5. Control de flujo en Theano.

En este código se usa una variable simbólica, x, que se utiliza para la definición de la función. En esta se utiliza T.switch, donde el primer parámetro es la condición lógica, el segundo es el valor que se obtendrá si la condición es cierta y el tercero, el que se obtendrá en caso contrario. Para la escritura de las operaciones lógicas también se usan las funciones suministradas en Theano, nunca los operadores estándares de Phyton. En esta ocasión, se ha utilizado el operador mayor que devuelve cierto, cuando el primer valor que se introduce es mayor que el segundo, y falso en el caso contrario. Algunos de los operadores lógicos más utilizados en Theano son:

- T.gt: mayor que.

- T.ge: mayor o igual que.
- T.lt: menor que.
- T.le: menor o igual que.
- T.and_: y lógico.
- T.or_: o lógico.

Comparación entre el T.switch e ifelse

El control de flujo se puede hacer también utilizando la función ifelse que se encuentra en Theano. La forma de utilizar ambas es exactamente igual, pero en la función en T.switch se ejecutan los dos resultados independientemente del resultado de la comparación, mientras que en ifelse solamente se ejecuta el cierto. Esto se traduce generalmente en un mayor tiempo de ejecución de T.switch. Esta diferencia entre las dos opciones se puede comprobar ejecutando el código de la figura 5.6.

```
import time
from theano.ifelse import ifelse

a, b = T.scalars('a', 'b')
x, y = T.matrices('x', 'y')

z_switch = T.switch(T.lt(a, b), T.mean(x), T.mean(y))
z_ifelse = ifelse(T.lt(a, b), T.mean(x), T.mean(y))

f_switch = theano.function([a, b, x, y], z_switch)
f_ifelse = theano.function([a, b, x, y], z_ifelse)

val1 = 0.
val2 = 1.
big_mat = numpy.ones((15000, 15000))

tic = time.clock()
f_switch(val1, val2, big_mat, big_mat)
print('El tiempo utilizando switch es %f' % (time.clock() - tic))

tic = time.clock()
f_ifelse(val1, val2, big_mat, big_mat)
print('El tiempo utilizando switch es %f' % (time.clock() - tic))
```

Figura 5.6. Comparación entre el T.switch e ifelse.

En estas líneas se crean dos funciones que hacen exactamente lo mismo, en base a un par de escalares (a y b) deciden si calcular la media de la primera matriz x o de la matriz y. La única diferencia entre las dos funciones es que en la primera (f_switch) se emplea T.switch, mientras que en la segunda (f_ifelse) se emplea ifelse. Para comprobar el tiempo que tarda cada una se ejecutan las dos funciones con una matriz de 15.000 por 15.000 registros: la primera tarda aproximadamente el doble de tiempo que la segunda, lo esperado, ya que en la primera implementación hace el doble de trabajo al calcular la media y la mediana para todos los valores.

Valores por defecto en funciones de Theano

En caso de necesitar valores por defecto en la función de Theano, pueden ser indicados mediante la función theano.In, como se muestra en el ejemplo de la figura 5.7., donde se crea una multiplicación y el segundo parámetro se ha configurado por defecto a tres. Los resultados obtenidos son, obviamente, 30 y 20 respectivamente.

```
x = T.scalar('x')
y = T.scalar('y')

z  = x * y

f = theano.function(inputs = [x, theano.In(y, value = 3)], outputs= z)

print f(10)
print f(10, 2)
```

Figura 5.7. Valores por defecto en funciones de Theano.

A ninguna de las variables definidas en Theano se le ha asignado un valor explícito, solamente se han utilizado para indicar las operaciones a realizar. Los únicos valores que se han usado son los de las entradas de las funciones. Poder indicar valores explícitos para algunas variables es necesario en muchas ocasiones, de hecho, los parámetros de un modelo se han de indicar de esta manera. En Theano, para que una variable pueda tener un valor explícito esta ha de ser de tipo shared (compartido).

Variables compartidas

Para crear una variable compartida se ha de utilizar la función theano.shared(), a la que se le ha de pasar un objeto con los valores. Theano es más exigente que Python con los tipos de datos, por lo que es aconsejable indicarlos. Por ejemplo, para definir un escalar se puede utilizar la siguiente línea:

x = theano.shared(np.array(1, dtype = theano.config.floatX))

En ella se indica como tipo de dato el valor por defecto de la configuración.

Las variables compartidas tienen la posibilidad de ser actualizadas durante la ejecución de una función de Theano. Para esto se ha de señalar la fórmula de actualización en la opción updates durante la definición de la función. Una forma de hacerlo es utilizar llaves para la variable compartida separada con dos puntos de la expresión empleada para la actualización. En la figura 5.8. se muestra un ejemplo:

```
x = theano.shared(np.array(1, dtype = theano.config.floatX))
A = T.scalar()
f = theano.function(inputs = [A], outputs = x, updates = {x: x - A})

print f(np.array(1))
print x.get_value()
```

Figura 5.8. Variables compartidas.

En la figura 5.8. se puede ver cómo se crea una variable compartida (x) de tipo escalar que tiene como valor 1, además, se ha creado un escalar (A) para ser utilizado en una función. Posteriormente se crea una función que tiene como entrada el escalar A, como salida la variable compartida y una función para actualizar la variable compartida con el escalar A.

Al ejecutar el código se observa que la salida de la función es 1 y, posteriormente, la variable compartida tiene un valor de 0. Esto da una pista del momento en el que se ejecuta la actualización de la variable compartida. Inicialmente se ejecuta la función, en este caso se devuelve el valor de x, y posteriormente se actualiza el valor. Este orden se ha de tener en cuenta para definir las funciones.

Por otro lado, en esta ocasión no ha sido necesario indicar que la variable de salida es x para que se pueda actualizar. La variable también se actualizará si se define la función de la forma:

f = theano.function(inputs = [A], updates = {x: x - A})

Pero en esta ocasión la salida de la función será un vector vacío.

Las variables compartidas pueden ser también matrices o tensores. En la figura 5.9. se ha creado una matriz de 2 por 2 y se emplea una función como la definida anteriormente.

```
x = theano.shared(np.array([[1, 2], [3, 4]], dtype = theano.config.floatX))
A = T.matrix()
f = theano.function(inputs = [A], outputs = x, updates = {x: x - A})

print f(np.array([[1, 1], [1, 1]]))
print x.get_value()
```

Figura 5.9. Matrices en variables compartidas.

Al igual que en el caso anterior, la función devuelve el valor original de la variable compartida, posteriormente esta variable compartida ya tiene el valor actualizado según la función.

Operaciones matriciales

Theano es un lenguaje con una gran potencia en cálculo matricial, de hecho, esta es una de las ventajas por las que se utiliza en programación de redes neuronales. El producto matricial de una matriz por un vector se escribe de la siguiente manera:

x = W × v + b

x, **v** y **b** son vectores, W es una matriz y x es el operador multiplicación. Resulta ser una operación tan sencilla de implementar como la suma o la multiplicación de dos números en Python.

Para implementar esta operación se han de definir dos vectores, una matriz y utilizar el operador producto (T.dot), esto es lo que se implementa en la figura 5.10.

```
W = T.matrix('W')
v = T.vector('v')
b = T.vector('biases')

x = T.dot(v, W) + b

f = theano.function(inputs = [v, W, b], outputs = x)

f([1,1], [[2,4],[3,5]], [2, 3])
```

Figura 5.10. Operaciones matriciales.

El resultado que se obtiene con el código de la figura 5.10 es el vector [7, 12], que es el resultado esperado de la operación.

Gradientes

Finalmente, otra de las grandes ventajas de Theano es la facilidad con la que se pueden calcular los gradientes de las funciones. El gradiente, como se ha visto en la segunda unidad, es necesario para poder obtener los parámetros de los modelos utilizando el método del gradiente descendente.

Para calcular el gradiente de una variable respecto a otra, se ha de utilizar la función T.grad indicando como primer parámetro la variable sobre la que se desea estimar el gradiente y como segundo la variable respecto a la que se realiza. Esta función devuelve una nueva variable simbólica que se puede ser utilizada en las funciones o evaluada.

En la figura 5.11. se muestra un ejemplo. En ella, se define una variable simbólica (x), otra como la potencia al cuadrado de la primera (y = x**2) y, finalmente, se calcula la derivada de y respecto a x. En esta ocasión, el valor de la derivada es fácil de calcular manualmente (2x) y se puede comprobar que el cálculo es correcto.

```
x = T.scalar()
y = x**2

# y_grad = dy/dx
y_grad = T.grad(y, x)

# dy/dx = 2 * x
y_grad.eval({x: 10})
```

Figura 5.11. Gradientes.

La potencia de la función gradiente de Theano es tal que permite derivar cualquier tipo de función en polinomios, incluyendo funciones trigonométricas, logaritmos y otras funciones más complejas.

Esta potencia se puede utilizar para implementar en pocas líneas el algoritmo de gradiente descendente. Por ejemplo, en la figura 5.12. se muestra una forma de implementar la regresión lineal que se ha visto en la unidad 2.

```
trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.50 + 10

X = T.scalar()
Y = T.scalar()

def model(X, w, c):
    return X * w + c

w = theano.shared(np.asarray(0., dtype = theano.config.floatX))
c = theano.shared(np.asarray(0., dtype = theano.config.floatX))
y = model(X, w, c)

cost     = T.mean(T.sqr(y - Y))
gradient_w = T.grad(cost = cost, wrt = w)
gradient_c = T.grad(cost = cost, wrt = c)
updates  = [[w, w - gradient_w * 0.01], [c, c - gradient_c * 0.01]]

train = theano.function(inputs = [X, Y], outputs = cost, updates = updates)

for i in range(15):
    for x, y in zip(trX, trY):
        cost_i = train(x, y)
    print ('En el paso', i, 'el valor de w es', w.get_value(),
           'y c es', c.get_value(), 'con un coste', cost_i)
```

Figura 5.12. Implementación de una regresión lineal en Theano.

Lo primero que se hace en este código es un conjunto de datos de muestra. Para ello, se toman 101 puntos entre -1 y 1 para calcular el valor que tendría un modelo lineal para la expresión: $y = 2x + 10$ a la que se le ha añadido un ruido blanco de amplitud y media igual a 0,25. Posteriormente, se crean dos variables para el modelo en el que se utilizan dos variables compartidas, una para cada uno de los parámetros del modelo.

Una vez se ha definido el modelo, se ha de calcular la función de coste (cost) empleando el error cuadrático medio, para lo cual se ha de calcular el gradiente con respecto a los dos parámetros (gradient_w y gradient_c) y utilizarlos para definir una función de actualización. Esta actualización, como se ha visto en la unidad 2, es el valor del parámetro menos el gradiente por el factor de aprendizaje, en el ejemplo de la figura 5.12. es de 0,01. Con todo esto, solo falta crear una función Theano en la que se indiquen los valores de entrada, los de salida y la función de actualización.

En el ejemplo se realiza el entrenamiento que se repite 15 veces y en la salida del código se puede ver que los valores se aproximan a los esperados.

3.2. Implementación de un perceptrón con Theano

Una vez estudiada la sintaxis básica de Theano, ya es posible utilizar este lenguaje para la implementación de una red neuronal con un único perceptrón, como la definida en la figura 5.1. Esta implementación se puede ver en la figura 5.13.

```
# Definición de las variables simbólicas
x = T.vector('x')

# Definición de las variables compartidas
w = theano.shared(np.array([1, 1], dtype = theano.config.floatX))
b = theano.shared(-1.5)

# Definición de la neurona
z = T.dot(x, w) + b
a = T.switch(T.lt(z, 0), 0, 1)

# Conjunto de datos
inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]

# Creación de la función
neuron = theano.function([x], a)

# Iteramos sobre todas las entradas
for i in range(len(inputs)):
    t = inputs[i]
    out = neuron(t)
    print 'El resultado para [%d, %d] es %d' % (t[0], t[1], out)
```

Figura 5.13. Implementación de una red neuronal con una neurona en Theano.

El código comienza con la creación de una variable simbólica (x) para los vectores de entrada de la red. Posteriormente, se crean dos variables compartidas. En la primera (w), se guardan los pesos utilizados para los vectores de entrada y en la segunda (b) el valor del bias del perceptrón. A partir de aquí, se crea la neurona como la multiplicación escalar del vector de entrada (x) por el vector de pesos (w) y se le añade el bias (b). El resultado se evalúa en una neurona (a). Para facilitar la implementación se ha utilizado el control de flujo switch, de tal manera que se obtiene 0 cuando z es positivo o 1 en cualquier otro caso, para posteriormente definir la matriz de inputs y crear la función Theano. En el bucle final se evalúan los inputs para comprobar que los resultados son los esperados, la función AND.

Al ejecutar el código, los resultados son los esperados, concretamente, se observa la siguiente salida:

- con [0, 0] se obtiene 0.
- con [0, 1] se obtiene 0.
- con [1, 0] se obtiene 0.
- con [1, 1] se obtiene 1.

3.3. Entrenamiento de las redes mediante propagación hacia atrás

En la sección anterior se ha visto cómo ejecutar una red neuronal en Theano para la que se conocen los valores de sus parámetros. En los problemas reales esto no sucede, por lo que es necesario entrenar las redes como al resto de modelos que se ha estudiado en las unidades anteriores. Las redes neuronales se suelen entrenar mediante el algoritmo de propagación hacia atrás de errores o retropropagación (*backpropagation*).

El algoritmo de propagación hacia atrás se implementa en dos etapas. En la primera se aplica a la red neuronal un estímulo de entrada para el que la respuesta es conocida, que se propaga desde la entrada hasta la salida produciendo una respuesta. En ese momento se compara el valor obtenido con el deseado para calcular el error. Este error se propaga hacia atrás por la red, desde la salida hasta la entrada, en base a la contribución relativa que cada una de las neuronas ha tenido en la respuesta final.

En la figura 5.14. se muestra un ejemplo de código para entrenar una neurona como la que se había definido en la figura 5.1., partiendo de la idea utilizada en la figura 5.13. En esta nueva implementación se han introducido unos cambios respecto a la anterior. En primer lugar, se ha definido a x como una matriz, en lugar de un vector, para poder usar un enfoque vectorial en donde se determinan todos los resultados de una forma conjunta. Por otro lado, se ha cambiado la forma de la neurona de una función escalón a una función logística.

Figura 5.14. Implementación de la propagación hacia atrás.

```
# Definición de las variables simbólicas
x = T.matrix('x')
w = theano.shared(np.array([1, 1], dtype = theano.config.floatX))
b = theano.shared(1.0)
learning_rate = 0.01

# Definición de la neurona
z = T.dot(x, w) + b
a = 1 / (1 + T.exp(-z))

# Definición de la función de coste
a_hat = T.vector('a_hat')
cost = -(a_hat * T.log(a) + (1 - a_hat) * T.log(1 - a)).sum()

# Gradiente de la función de coste
dw, db = T.grad(cost, [w, b])

train = theano.function(
    inputs = [x, a_hat],
    outputs = [a, cost],
    updates = [[w, w - learning_rate * dw], [b, b - learning_rate * db]])
```

Como función de coste se ha empleado la entropía cruzada (*cross entropy*), que se define como:

$$J(z) = -\sum_{i=1}^{n}(\bar{a}_i * log(a(z_i)) + (1 - \bar{a}_i) * log(1 - a(z_i)))$$

$\bar{a}_i$ es la respuesta conocida y$a(z_i)$ es la respuesta del modelo a la entrada dada. Se ha de recordar que el valor de:

$$a(z) = \frac{1}{1 + e^{-z}}$$

z es el producto de la entrada (x) por los pesos (w) a lo que se añade el bias (b).

Una de las grandes ventajas de Theano es que una vez definida la función de esfuerzo, por complicada que esta sea, el gradiente se puede obtener directamente con la función T.grad. Tal y como se puede ver en el código, es posible calcular más de una derivada en la misma línea, simplemente se han de indicar las variables sobre las que se ha de derivar.

Finalmente, se define la función de entrenamiento. Como inputs se determinan las entradas de las redes y las salidas deseadas, como salida se indica el resultado de la red (a) y la función de coste. Además, se ha de señalar cómo actualizar las variables compartidas en las que se almacenan los parámetros, tanto los pesos (w) como el bias (b) se han de actualizar restando la derivada parcial de la función de esfuerzo por la ratio de aprendizaje.

El código para entrenar la red se muestra en la figura 5.15. En este se ha definido una matriz con los cuatro posibles valores de entrada de la red (inputs) y las cuatro posibles salidas (outputs). Posteriormente se crea un vector vacío para los costes y se itera 10.000 veces sobre el conjunto de entrenamiento, añadiendo en cada paso el resultado de la función de esfuerzo al vector de coste. Finalmente, se imprimen los resultados obtenidos por la pantalla.

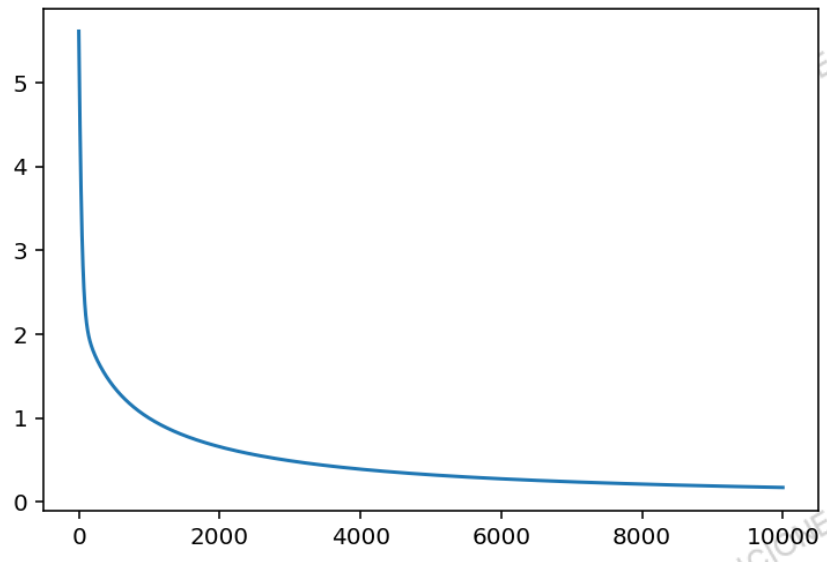
Figura 5.15. Obtención de los parámetros.

```
# Conjunto de datos de entrenamiento
inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]
outputs = [0,0,0,1]

# Iteramos sobre el conjunto de entrenamiento
cost = []
for iteration in range(10000):
    pred, cost_iter = train(inputs, outputs)
    cost.append(cost_iter)

# Se imprimen los resultados por pantalla
print 'Los resultados de la red son:'
for i in range(len(inputs)):
    print 'El resultado para [%d, %d] es %.2f' % (inputs[i][0], inputs[i][1], pred[i])

# Resultados
print
print 'El vector w es [%.2f, %.2f]' % (w.get_value()[0], w.get_value()[0])
print 'El valor del bias es %.2f' % b.get_value()

# Función de esfuerzo en función del número de iteraciones
plt.plot(cost)
```

Los resultados obtenidos son los esperados: una red que devuelve aproximadamente la unidad cuando las dos entradas son unos y valores cercanos a cero en el resto de los casos. Los pesos obtenidos son 5,58 y 5,58, mientras que el bias es -8,55.

En la figura 5.16. se muestra el valor de la función de esfuerzo según el número de iteraciones. En ella se puede apreciar que en menos de 2000 pasos se ha llegado a una solución cercana al final. A partir de este punto, el valor de la entropía cruzada se reduce de una forma más suave.

Figura 5.16. Valores de la función de esfuerzo con respecto al número de iteraciones.



IV. Redes neuronales

En la sección anterior se ha visto que, mediante la utilización de un perceptrón o neurona, es posible resolver problemas sencillos, como una operación lógica de tipo AND. En caso de que sea necesario resolver problemas más complejos, como los de clasificación múltiple, donde el número de posibles respuestas es mayor, es necesaria la utilización de más neuronas. En estos casos se puede utilizar una neurona para predecir cada posible respuesta.

Por ejemplo, para la identificación de números escritos a mano se pueden utilizar diez neuronas, la primera para reconocer el 0, la segunda para el 1 y así hasta el 9. Esto se llama una capa de neuronas. Esta idea se muestra en la figura 5.17., donde se puede ver que cada neurona recibe todos los valores de entrada, cada una tendrá unos pesos diferentes y ofrece una respuesta diferente. El número de valores de entrada no tiene por qué coincidir con el número de neuronas, por ejemplo, en el caso anterior, al implementar el perceptrón, había dos valores para una única neurona.

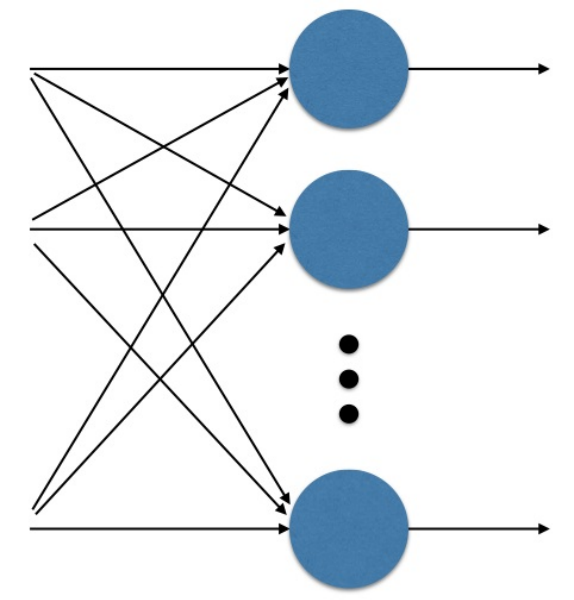


Figura 5.17. Capa de neuronas.

Mediante la utilización de una capa, se pueden resolver problemas más complejos de los que sería posible con una única neurona. Aun así, estos sistemas no son suficientes para resolver cualquier problema. Ante esto se plantea la necesidad de implementar redes de neuronas con múltiples capas, de tal forma que la respuesta de una capa pueda ser la entrada de la siguiente. Estos sistemas multicapa son lo que se conocen como redes neuronales.

La adición de capas a las redes neuronales puede hacer que existan capas que no reciban directamente los datos de entrada ni respondan directamente al problema. Estas capas se denominan capas intermedias o capas ocultas. La capa que recibe directamente los datos del problema es la que se conoce como capa de entrada, mientras que la que da la respuesta al final es la que se conoce como capa de salida. Hasta ahora las redes utilizadas tenían una única capa que era a la vez la capa de entrada y la de salida.

El esquema de una red multicapa es la que se muestra en la figura 5.18. En esta figura se puede observar la capa de entrada en verde, la capa de salida en rojo, las capas ocultas en azul y las conexiones mediante líneas negras. En esta figura se puede ver que la capa de entrada tiene tres neuronas, la primera capa oculta tiene cuatro, la segunda dos, la tercera vuelve a tener cuatro y la de salida tiene tres. Se observa, pues, que la cantidad de neuronas en cada una de las capas puede ser diferente al resto.

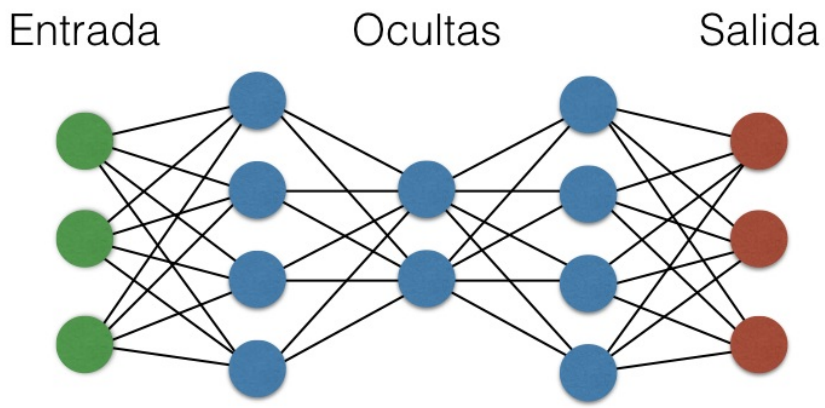


Figura 5.18. Esquema de red neuronal multicapa.

Esquema red neuronal para resolver la red XNOR

En la sección anterior se implementó una red neuronal para resolver un problema tipo AND lógico, en esta sección se va a crear una red con dos capas para resolver un problema de tipo XNOR. Esta operación es un OR excluyente, solamente es cierto si una de las entradas es falsa y la otra es verdadera, a la que se le aplica un operador negación. La tabla de un problema XNOR se muestra en la tabla 5.2.

x_1	x_2	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Tabla 5.2. Operador XNOR.

El esquema de la red neuronal que se va a utilizar para resolver la red XNOR se muestra en la figura 5.19. En este esquema se puede apreciar que la capa de entrada consta de dos neuronas y la capa de salida solamente de una. Al igual que en el problema anterior, la cantidad de variables de entrada es dos. Nótese que ahora hay nueve pesos, tres por cada neurona, por lo que el entrenamiento va a ser más complejo y por lo tanto requerirá de más iteraciones para llegar a una solución satisfactoria.

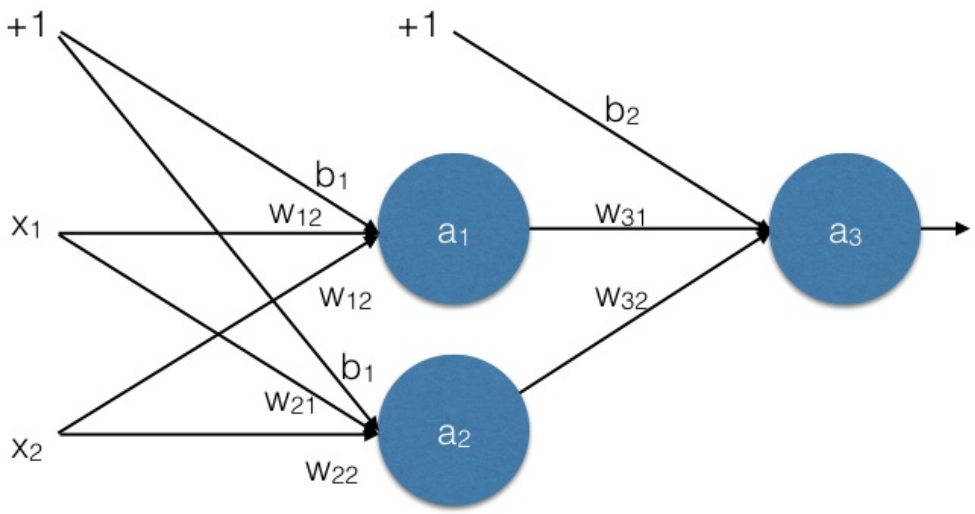


Figura 5.19. Red neuronal de dos capas utilizada para la implementación del operador XNOR.

El primer paso para implementar esta red es definir las variables necesarias, esto es lo que se hace en el código de la figura 5.20. En este código se puede observar que se ha creado una matriz (x) para los datos de entrada, tres variables compartidas (w_1, w_2 y w_3) para almacenar los pesos de cada neurona, dos variables compartidas (b_1 y b_2) para los pesos de los bias y el ratio de aprendizaje (learning_rate). Como se puede comprobar en la figura 5.19., se usa el mismo bias para las dos neuronas de la capa de entrada, por lo que solamente se han definido dos variables. Por otro lado, los valores iniciales de los pesos podrían ser aleatorios, pero fijándolos de forma manual se garantiza que los resultados sean los mismos en todas las ejecuciones.

```
x = T.matrix('x')
w1 = theano.shared(np.array([.1, .2], dtype = theano.config.floatX))
w2 = theano.shared(np.array([.3, .4], dtype = theano.config.floatX))
w3 = theano.shared(np.array([.5, .6], dtype = theano.config.floatX))
b1 = theano.shared(1.)
b2 = theano.shared(1.)
learning_rate = 0.01
```

Figura 5.20. Definición de las variables para el problema de la figura 5.19.

Posteriormente, se han de crear las neuronas de la red, esto es lo que se muestra en el código de la figura 5.21. En este se crean dos neuronas (a_1 y a_2) que reciben los valores de entrada multiplicados por el peso que se asocia a cada una (w_1 y w_2). Una vez que se han definido las dos neuronas, se juntan sus respuestas mediante la función T.stack para ser utilizadas como la entrada de la tercera neurona.

```
a1 = 1 / (1 + T.exp(-T.dot(x, w1) - b1))
a2 = 1 / (1 + T.exp(-T.dot(x, w2) - b1))
x2 = T.stack([a1, a2], axis = 1)
a3 = 1 / (1 + T.exp(-T.dot(x2, w3) - b2))
```

Figura 5.21. Definición de las neuronas para el problema de la figura 5.19.

El siguiente paso es construir la función para el entrenamiento de la red definida en el paso anterior. Esto es lo que se muestra en el código de la figura 5.22. Al igual que en el ejercicio de la sección anterior, inicialmente se crea una variable para los resultados de entrenamiento (a_hat) y se define la función de coste. En este punto es importante destacar que para la función de coste solamente se utiliza la neurona de la capa de salida, en ningún momento se emplea ninguna referencia a las neuronas a_1 o a_2. Esto es así porque el error se comete en la última capa. En este punto se ha de calcular el gradiente de la función de esfuerzo respecto a todos los parámetros del modelo (w_1, w_2, w_3, b_1 y b_2). La función para el entrenamiento es similar a la que se ha empleado en el caso anterior, pero las variables que se actualizan en esta ocasión son seis.

```
a_hat = T.vector('a_hat')
cost = -(a_hat * T.log(a3) + (1 - a_hat) * T.log(1 - a3)).sum()
dw1, dw2, dw3, db1, db2 = T.grad(cost, [w1, w2, w3, b1, b2])

train = theano.function(
    inputs = [x,a_hat],
    outputs = [a3,cost],
    updates = [
        [w1, w1 - learning_rate * dw1],
        [w2, w2 - learning_rate * dw2],
        [w3, w3 - learning_rate * dw3],
        [b1, b1 - learning_rate * db1],
        [b2, b2 - learning_rate * db2]
    ]
)
```

Figura 5.22. Creación de la función de entrenamiento.

Finalmente, se puede entrenar la red neuronal de una forma análoga a como se ha realizado en la sección anterior. El procedimiento se puede ver en el código de la figura 5.23. En esta ocasión, debido a que el problema es más complejo, se ha aumentado la cantidad de iteraciones de 20.000 hasta 50.000.

```
inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]
outputs = [1, 0, 0, 1]

# Iteramos sobre el conjunto de entrenamiento
cost = []
for iteration in range(50000):
    pred, cost_iter = train(inputs, outputs)
    cost.append(cost_iter)
```

Figura 5.23. Entrenamiento de la red neuronal.

Tras el proceso de entrenamiento se puede observar que la red reproduce los resultados deseados. Concretamente, los resultados son 0,99 para [0,0] y [1,1] y 0,01 para [0, 1] y [1, 0], que es lo que se buscaba. Los vectores que se han obtenido después del entrenamiento son:

- w_1 = [4,17 4,17]
- w_2 = [9,55 9,55]
- w_3 = [12,83 12,83]
- b_1 = -6,25
- b_2 = 4,64

Finalmente, en la figura 5.24. se puede ver la evolución de la función de esfuerzo frente al número de iteraciones. En ella se comprueba una fuerte caída inicial del valor para mantenerse estable durante aproximadamente 15.000 iteraciones, momento en el que el error vuelve a caer. A partir de aquí, se puede deducir que durante los entrenamientos de las redes neuronales es posible que estas se estanquen en un mínimo local antes de alcanzar otros como el mínimo global u otros mínimos locales.

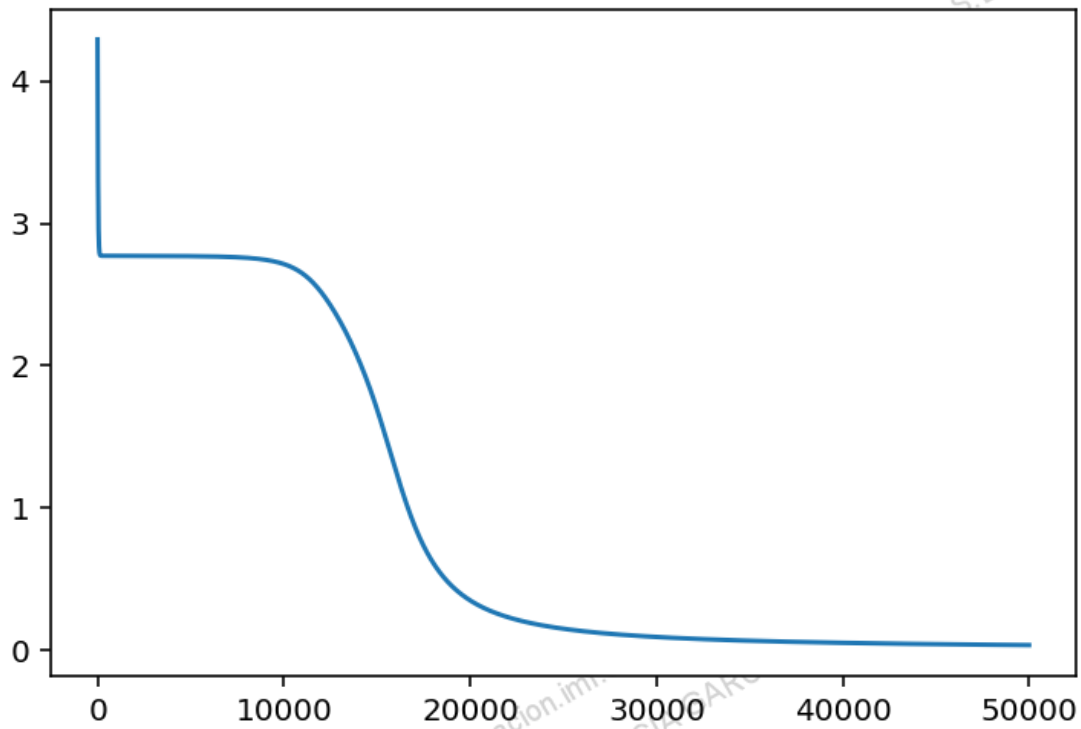


Figura 5.2. Evolución de la función de error con el número de iteraciones.

En las implementaciones que se han visto hasta el momento, las neuronas se han definido una a una. En una aplicación real, donde se pueden implementar cientos de neuronas en cada capa, este no es un enfoque práctico. A causa de esto, en Theano ya se encuentran implementados múltiples tipos de neuronas en theano.tensor.nnet. Entre las que se pueden destacar:

sigmoid

$$a(z) = \frac{1}{1 + e^{-z}}$$

softplus

$$a(z) = log_e(1 + e^z)$$

donde log_e indica el logaritmo en base e.

softmax

$$a_{ij}(z) = \frac{e^{z_{ij}}}{\sum_k e^{z_{ik}}}$$

softsign

$$a(z) = \frac{1}{1 + |z|}$$

Además, hasta el momento, los valores de los pesos iniciales se han fijado manualmente, lo que en producción tampoco es adecuado, ya que puede ser necesario configurar miles. Una forma de abarcar esto es con la generación aleatoria de los pesos, para esto se puede usar la función de la figura 5.25., en donde solo hay que indicar el tamaño de la matriz para crear los pesos. En caso de que sea necesario garantizar la respetabilidad de los resultados simplemente se ha de fijar la semilla del generador de números aleatorios.

```
def floatX(X):
    return np.asarray(X, dtype = theano.config.floatX)

def init_weights(shape):
    return theano.shared(floatX(np.random.randn(*shape) * 0.01))
```

Figura 5.25. Funciones para la creación de pesos aleatorios.

A partir de estas consideraciones se puede revisar la implementación de la red neuronal de la figura 5.19. Se muestra una forma más eficaz en la figura 5.26. Las principales diferencias que se observan son la utilización del generador de números aleatorios para crear los pesos iniciales, la función T.nnet.sigmoid para la creación de las dos capas de neuronas (a_1 y a_2) y que la función de esfuerzo se calcula utilizando el método que imprenta la entropía cruzada binaria de Thenano. La salida de la segunda capa de neuronas (a_2) se ha de procesar utilizando la función flatten para que esta tenga la misma forma que el vector de entrenamiento.

```
# Semilla
rng = np.random.RandomState(1)

# Bias
b1 = theano.shared(1.)
b2 = theano.shared(1.)

# Pesos iniciales aleatorios
w1 = init_weights((2, 3))
w2 = init_weights((3, 1))

# Definición de la red
a1 = T.nnet.sigmoid(T.dot(x, w1) + b1)
a2 = T.nnet.sigmoid(T.dot(a1, w2) + b2)
a3 = T.flatten(a2)

# Función de esfuerzo
cost = T.nnet.binary_crossentropy(a3, a_hat).mean()

# Función de entrenamiento
train = theano.function(inputs = [x, a_hat],
                        outputs = [a3, cost],
                        updates = [
                            (w1, w1 - learning_rate * T.grad(cost, w1)),
                            (w2, w2 - learning_rate * T.grad(cost, w2)),
                            (b1, b1 - learning_rate * T.grad(cost, b1)),
                            (b2, b2 - learning_rate * T.grad(cost, b2))
                        ])
```

Figura 5.26. Definición de la red neuronal con funciones nativas de Theano.

Los resultados que se obtienen ejecutando esta última versión son similares a los obtenidos previamente, concretamente: 1,00 cuando la red se activa con [0, 0] y [1, 1] y 0,00 en el resto de los casos, [1, 0] y [0, 1].

La gráfica de la función de esfuerzo respecto al número de iteraciones también muestra resultados similares, como se puede ver en la figura 5.27. Al igual que en el caso anterior, el valor de la función de esfuerzo cae inicialmente para estabilizarse durante aproximadamente 18.000 iteraciones, para volver a caer después.

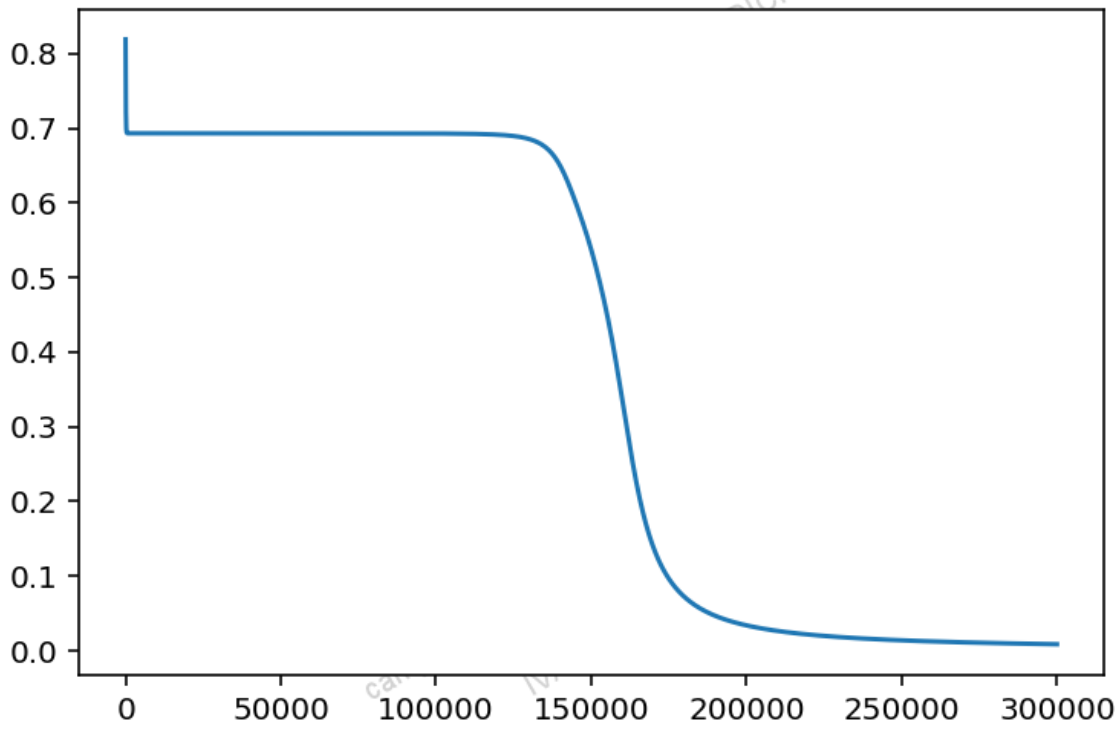


Figura 5.27. Evolución de la función de error con el número de iteraciones.

Utilizar las funciones que suministra Theano permite escribir redes más complejas de una forma más eficiente, como se verá en la siguiente sección.

V. Clasificación de dígitos escritos a mano

La utilización de redes neuronales es cada vez más habitual. Una de las aplicaciones más populares, dentro de los sistemas de identificación de patrones en imágenes, es el reconociendo óptico de caracteres (OCR, *Optical Character Recognition*). En este apartado se va a implementar un pequeño OCR utilizando el popular conjunto de datos MNIST (*Modified National Institute of Standards and Technology*), en el que se encuentran 60.000 imágenes de números entre 0 y 9 etiquetados para su entrenamiento. Este conjunto de datos se encuentra disponible entre los ejemplos que contiene scikit-learn, pero en esta sección se utilizará la versión de la página web The MNIST database of handwritten digits, debido a que la resolución es mayor en esta última. La forma en la que se encuentran almacenados los datos está diseñada para optimizar el espacio, por lo que hay que utilizar una función escrita especialmente para su importación. Esta función se puede consultar en el notebook que se adjunta a esta unidad. Esta función devuelve dos conjuntos de datos aleatorios, uno para el entrenamiento (trX y trY) y otro para la validación (teX, teY). En ambos conjuntos, las salidas etiquetadas con una X contienen las imágenes y las etiquetadas con Y contienen vectores lógicos de longitud 10, en los que todos los valores son falsos salvo aquel que pertenece a la posición del número correspondiente.

Yann LeCun (Courant Institute), Corinna Cortes (Google Labs), Christopher J. C. Burges (Microsoft Research), The MNIST database of handwritten digits. [En línea] URL disponible en: http://yann.lecun.com/exdb/mnist/

En la figura 5.28 se muestra un ejemplo de las imágenes que se pueden encontrar en este conjunto de datos. En esta figura se aprecia que la imagen es en blanco y negro con una resolución de 28 por 28 pixeles, lo que indica que cada una tiene 784 píxeles. El vector correspondiente a esta imagen es [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], donde solamente el primer valor es distinto de cero.

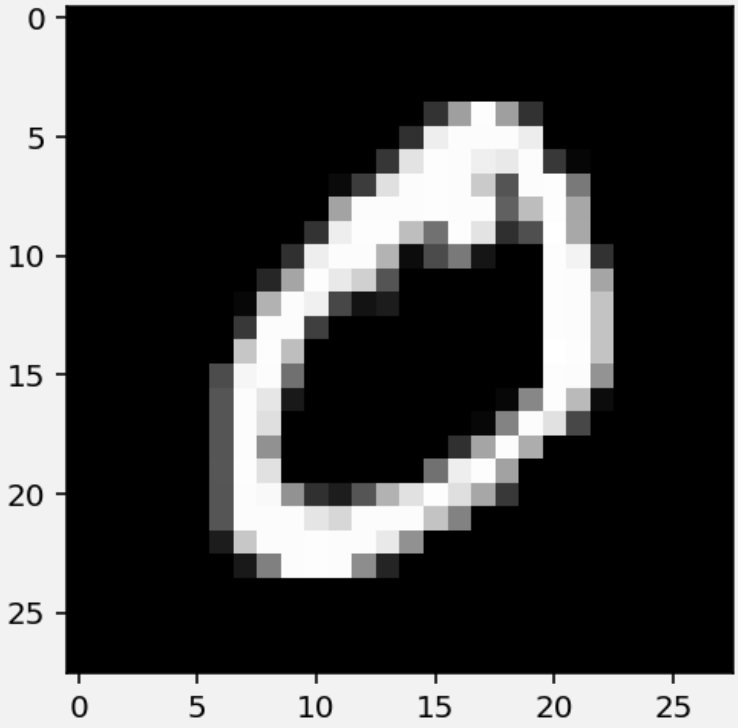


Figura 5.28. Ejemplo de uno de los números disponibles en el conjunto de datos MNIST.

1

Antes de implementar una red neuronal que permita clasificar las imágenes del conjunto de datos, es necesario analizar la forma que han de tener las capas de entrada y salida de esta. Para ello se ha de comprobar la forma de los datos disponibles. La capa de entrada es definida por el tamaño de las imágenes, en este caso, cada una tiene 784 píxeles, por lo que el tamaño será de 784. Por otro lado, la capa de salida es definida por los posibles estados que hay que etiquetar, en este caso son 10, uno para cada uno de los posibles caracteres a identificar. Así, la red ha de tener una capa de entrada para 784 pesos y una de salida con 10.

Se puede configurar una red sencilla para clasificar las imágenes solamente con 10 neuronas, en la que cada una de las ellas necesitará 784 pesos, uno por cada pixel. La forma de la neurona puede ser la que se ha empleado en los ejemplos anteriores, la sigmoid. La respuesta de la red así configurada será un vector con 10 valores numéricos. Como lo que se necesita es una categoría, la predicción se obtiene seleccionando la neurona en la que el valor de activación sea el mayor del conjunto. Para la función de esfuerzo también se puede emplear la función utilizada en los ejemplos anteriores, la entropía cruzada, para actualizar los parámetros con su gradiente. Finalmente, se ha de crear una función de entrenamiento en la que se inyecten los datos, se obtenga la función de coste y se actualicen los valores. Para obtener la predicción para una imagen, lo más adecuado es definir una función de predicción en la que solamente se pase la imagen y se obtenga el resultado. Esta implementación se muestra en el código de la figura 5.29.

```
trX, teX, trY, teY = mnist()

X = T.fmatrix()
Y = T.fmatrix()

w = init_weights((784, 10))

py_x = T.nnet.sigmoid(T.dot(X, w))

y_pred = T.argmax(py_x, axis=1)

cost = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
gradient = T.grad(cost, w)
update = [[w, w - gradient * 0.1]]

train = theano.function(inputs = [X, Y],
                        outputs = cost,
                        updates = update,
                        allow_input_downcast=True)
predict = theano.function(inputs = [X],
                          outputs = y_pred,
                          allow_input_downcast = True)

for i in range(num_iter):
    for start, end in zip(range(0, len(trX), 128), range(128, len(trX), 128)):
        cost = train(trX[start:end], trY[start:end])
    print i, np.mean(np.argmax(teY, axis=1) == predict(teX)), cost
```

Figura 5.29. Red neuronal de una capa con neuronas sigmoid.

2

En este código se ha entrenado la red neuronal las veces que marca la variable num_iter (en el notebook se ha utilizado 25), imprimiendo por pantalla la precisión conseguida por el modelo en cada iteración. Tras 25 iteraciones, este modelo tan sencillo puede conseguir una precisión del 64%, aunque no es un resultado bueno, es mucho mejor que el 10% que se conseguirá prediciendo el valor al azar.

Al definir las funciones de Theano se puede apreciar que se ha utilizado la opción allow_input_downcast tanto en la función para entrenamiento como para predicción. Esta opción permite que se realice una conversión de los tipos de datos utilizados en las variables de entrada para que se adapten los necesarios para el modelo. Por defecto, esta opción no se encuentra activada y en esta ocasión es necesario realizar esta conversión.

Una de las posibilidades para mejorar la capacidad de predicción de la red neuronal es modificar la forma de las neuronas utilizadas. Otra de las neuronas que se encuentran implementadas en Theano es softmax, matemáticamente se puede definir mediante la expresión:

$$softmax(x_{ij}) = \frac{e^{x_{ij}}}{\sum_k e^{x_{ik}}}$$

x_{ij} representa una matriz de valores. A diferencia de la neurona presentada previamente, en este caso la suma de los pesos de todas las neuronas de la capa es igual a la unidad, ya que se promedia en esta. En esta función se obtiene para cada elemento la exponencial del valor de entrada ponderado por la suma de las exponenciales del conjunto de datos, de este modo, los posibles resultados se encuentran acotados entre 0 y 1. En la figura 5.30. se muestra una implementación con esta nueva función.

```
w = init_weights((784, 10))

py_x = T.nnet.softmax(T.dot(X, w))

y_pred = T.argmax(py_x, axis=1)

cost = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
gradient = T.grad(cost, w)
update = [[w, w - gradient * 0.1]]

train = theano.function(inputs = [X, Y],
                        outputs = cost,
                        updates = update,
                        allow_input_downcast = True)
predict = theano.function(inputs = [X],
                          outputs = y_pred,
                          allow_input_downcast = True)

for i in range(num_iter):
    for start, end in zip(range(0, len(trX), 128), range(128, len(trX), 128)):
        cost = train(trX[start:end], trY[start:end])
    print i, np.mean(np.argmax(teY, axis=1) == predict(teX)), cost
```

Figura 5.30. Red neuronal de una capa con neuronas softmax.

3

En este último código solamente se ha cambiado, con respecto al anterior, la función de la neurona. El tamaño de la matriz de pesos es exactamente el mismo, la predicción se sigue realizando escogiendo la neurona que muestra mayor nivel de activación y la función de coste sigue siendo la entropía cruzada.

Los resultados obtenidos después de 25 iteraciones son bastante mejores que en el ejemplo anterior, concretamente, se llega a una precisión cercana a un 92%. Estos resultados se han conseguido cambiando únicamente el tipo de neurona utilizada, el número de neuronas y la función de esfuerzo son los mismos en ambas ocasiones. El rendimiento ya es satisfactorio, teniendo en cuenta el problema planteado.

Para mejorar la capacidad predictiva de la red neuronal se puede aumentar el número de capas. Esto es lo que se realiza en el código que se muestra en la figura 5.31. En este listado se crean dos conjuntos de pesos, el primero para 625 neuronas que van a recibir los 784 píxeles como entrada, la segunda para las 10 neuronas que recibirán la salida de la capa anterior. En la primera capa se van a utilizar neuronas de tipos sigmoid, mientras que en la segunda se emplearán las de tipo softmax. Se puede ver la forma en la que se combinan en el código; inicialmente se calcula la respuesta de la primera capa guardándola en la variable h, posteriormente, el resultado se utiliza como entrada de la segunda capa, guardándolo en la variable pv_x. De forma análoga a las implementaciones anteriores, se ha de seleccionar como respuesta la neurona que muestre un nivel de activación mayor.

```
w_h = init_weights((784, 625))
w_o = init_weights((625, 10))

h = T.nnet.sigmoid(T.dot(X, w_h))
py_x = T.nnet.softmax(T.dot(h, w_o))
y_x = T.argmax(py_x, axis = 1)

cost = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
updates = [[w_h, w_h - T.grad(cost, w_h) * 0.1],
           [w_o, w_o - T.grad(cost, w_o) * 0.1] ]

train = theano.function(inputs = [X, Y],
                        outputs = cost,
                        updates = updates,
                        allow_input_downcast = True)
predict = theano.function(inputs = [X],
                          outputs = y_x,
                          allow_input_downcast = True)

for i in range(num_iter):
    for start, end in zip(range(0, len(trX), 128), range(128, len(trX), 128)):
        cost = train(trX[start:end], trY[start:end])
    print i, np.mean(np.argmax(teY, axis=1) == predict(teX)), cost
```

Figura 5.31. Red neuronal de una capa con neuronas sigmoid y softmax.

4

La función de coste que se ha utilizado vuelve a ser la entropía cruzada, pero en esta ocasión se han de actualizar dos conjuntos de parámetros, por lo que se ha de calcular la derivada respecto a ambos. No ha sido necesario modificar el resto del código.

En esta implementación con dos capas se observa, después de 25 iteraciones, una precisión de 93%, muy similar al caso anterior. Esto, en cierta medida, es esperable, ya que las mejoras marginales que se pueden obtener son cada vez menores, debido a que el grado de acierto ya es bastante bueno en este momento.

A medida que se incrementa la cantidad de neuronas y capas en una red neuronal es fácil que los modelos se vuelvan más inestables, al mismo tiempo que aparece sobreajuste. Para solucionar este problema se emplean técnicas de regularización como dropout. En esta técnica se apaga aleatoriamente un porcentaje de neuronas durante el proceso de entrenamiento, de modo que su contribución al resultado final se ignora y sus pesos no se corrigen mediante la propagación hacia atrás.

Para comprender cómo funciona este proceso, hay que tener en cuenta que a medida que una red neuronal aprende, las neuronas se van especializado en ciertas características, al mismo tiempo que ignoran otras, confiando en que estas sean resueltas por sus vecinas. Si esto se lleva demasiado lejos se producen modelos excesivamente especializados que son inestables. En el caso de que algunas neuronas sean apagadas aleatoriamente, sus vecinas tendrán que responder en su lugar, provocando que aparezcan múltiples representaciones internas para los mismos datos y generando así sistemas más robustos.

En la figura 5.32. se muestra una implementación en el código de la función de dropout. En este código, primero se importa un generador de números para ser utilizado en Theano (MRG_RamdomStream) que se emplea en la función dropout. En esta función se ha de pasar un vector, en el que se encuentran los pesos de una capa, y se elimina aleatoriamente un porcentaje p. Para ello se utiliza la función del método binomial, que devuelve la unidad con probabilidad p y cero con probabilidad 1 – p. En esta función finalmente se normalizan los pesos por 1 – p.

```
from theano.sandbox.rng_mrg import MRG_RandomStreams

srng = MRG_RandomStreams()

def dropout(X, p):
    if p > 0:
        X *= srng.binomial(X.shape, p = 1 - p, dtype = theano.config.floatX)
        X /= 1 - p
    return X

def model(X, w_h, w_o, p_drop):
    X = dropout(X, p_drop)
    h = T.nnet.sigmoid(T.dot(X, w_h))

    h = dropout(h, p_drop)
    py_x = T.nnet.softmax(T.dot(h, w_o))

    return h, py_x
```

Figura 5.32. Implementación de dropout.

5

Posteriormente, se ha de crear una función modelo para poder cambiar la probabilidad de que las neuronas se apaguen. Esto es necesario porque en predicción no es necesario ni aconsejable apagar un porcentaje de las neuronas.

A partir de estas dos funciones se puede repetir el código de la figura 5.31., pero utilizando dropout durante el entrenamiento, esto es lo que se muestra en el código de la figura 5.33. En este código, la principal diferencia se encuentra en la utilización de la función model para crear un modelo para entrenar y otro diferente para la predicción cambiando únicamente el porcentaje de neuronas que se apagan en cada caso, 5% en entrenamiento y 0% en predicción.

```
w_h = init_weights((784, 625))
w_o = init_weights((625, 10))

# Modelo de entrenamiento
h, py_x = model(X, w_h, w_o, 0.05)
y_x = T.argmax(py_x, axis = 1)

cost = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
updates = [[w_h, w_h - T.grad(cost, w_h) * 0.1],
           [w_o, w_o - T.grad(cost, w_o) * 0.1] ]

train = theano.function(inputs = [X, Y],
                        outputs = cost,
                        updates = updates,
                        allow_input_downcast = True)

# Modelo para evaluación, con p == 0
h_predict, py_predict = model(X, w_h, w_o, 0.0)
y_predict = T.argmax(py_predict, axis = 1)

predict = theano.function(inputs = [X],
                          outputs = y_predict,
                          allow_input_downcast = True)

# Evaluación del modelo
for i in range(num_iter):
    for start, end in zip(range(0, len(trX), 128), range(128, len(trX), 128)):
        cost = train(trX[start:end], trY[start:end])
    print i, np.mean(np.argmax(teY, axis=1) == predict(teX)), cost
```

Figura 5.33. Evaluación de dropout.

Al ejecutar este código se observa que los resultados son ligeramente mejores a los obtenidos en la fase anterior. Esta mejora es debida a que se ha forzado a la red a disponer de varias neuronas especializadas en cada tipo de característica.

VI. Resumen

En esta unidad se ha realizado una presentación de los sistemas de aprendizaje automático más importantes de los últimos años: los sistemas conexionistas o redes neuronales.

La unidad ha comenzado realizando un repaso del concepto de perceptrón, o red neuronal más básica, que consiste en una única neurona. En esta parte se ha visto su funcionamiento y cómo se utiliza para realizar predicciones básicas en las que las cantidades de entradas se encuentran limitadas.

Posteriormente, se ha estudiado el lenguaje Theano, que permite la escritura de operaciones matriciales de una forma más sencilla que Phyton. En esta unidad se ha visto cómo utilizar Theano para implementar un perceptrón y entrenarlo. También se ha mostrado la construcción de redes más complejas para resolver problemas con una mayor cantidad de variaciones

En la última sección se ha utilizado todo lo aprendido hasta el momento para la creación de una red neuronal que permita implementar un OCR para la identificación de número escritos a mano.

Ejercicios

Caso práctico

Se ha de implementar una red neuronal que permita reproducir la función XOR de tres entradas, es decir, se reproducirá la tabla 5.3.

x_1	x_2	x_2	XOR
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabla 5.3. Operador XNOR.

Solución

En esta ocasión hay tres entradas, por lo que se va a utilizar una capa con tres neuronas para intentar predecir el resultado. En el diseño de la red neuronal se ha de tener en cuenta que los vectores de peso tienen que tener tres elementos. En la figura 5.34. se puede ver el esquema.

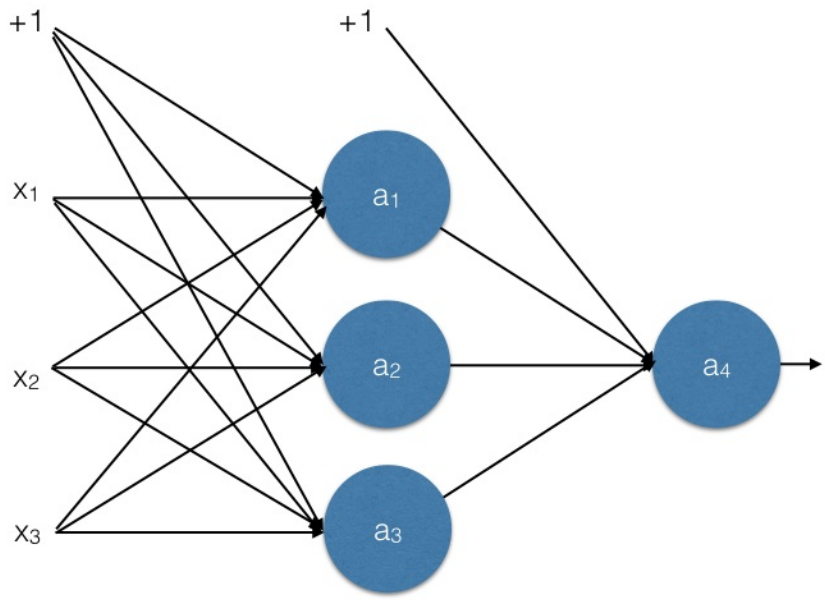


Figura 5.34. Esquema de red neuronal para la función XOR de tres entradas.

La implementación del modelo es similar a la realizada durante la unidad. En primer lugar, se han de crear los vectores de peso como variables compartidas, que en esta ocasión serán cuatro vectores de pesos y dos escalares para el bias. Los valores iniciales se fijan para garantizar la reproducibilidad de los resultados. Posteriormente se multiplica la matriz de entrada por los pesos para crear las tres neuronas de la primera capa e insertar estas en la neurona de la cuarta capa. Esto es lo que se puede ver en el código de la figura 5.35.

```
x = T.matrix('x')
w1 = theano.shared(np.array([-1, 0, 1], dtype = theano.config.floatX))
w2 = theano.shared(np.array([0, 1, -1], dtype = theano.config.floatX))
w3 = theano.shared(np.array([1, -1, 1], dtype = theano.config.floatX))
w4 = theano.shared(np.array([1, 2, 3], dtype = theano.config.floatX))
b1 = theano.shared(1.)
b2 = theano.shared(1.)
learning_rate = 0.01

a1 = 1 / (1 + T.exp(-T.dot(x, w1) - b1))
a2 = 1 / (1 + T.exp(-T.dot(x, w2) - b1))
a3 = 1 / (1 + T.exp(-T.dot(x, w3) - b1))
x2 = T.stack([a1, a2, a3], axis = 1)

a4 = 1 / (1 + T.exp(-T.dot(x2, w4) - b2))
```

Figura 5.35. Definición de la red neuronal.

Una vez definida la red, se ha de calcular la función de coste utilizando la entropía, las derivadas de estas respecto a las variables compartidas y definir una función que permita actualizar los valores. Esto es lo que se muestra en la figura 5.36.

```
a_hat = T.vector('a_hat')
cost = -(a_hat * T.log(a4) + (1 - a_hat) * T.log(1 - a4)).sum()
dw1, dw2, dw3, dw4, db1, db2 = T.grad(cost, [w1, w2, w3, w4, b1, b2])

train = theano.function(
    inputs = [x, a_hat],
    outputs = [a4, cost],
    updates = [
        [w1, w1 - learning_rate * dw1],
        [w2, w2 - learning_rate * dw2],
        [w3, w3 - learning_rate * dw3],
        [w4, w4 - learning_rate * dw4],
        [b1, b1 - learning_rate * db1],
        [b2, b2 - learning_rate * db2]
    ]
)
```

Figura 5.36. Definición de la función de actualización.

Finalmente, se ha de entrenar la red neuronal ejecutando la función de entrenamiento una cantidad de veces. Para esto, se utilizan los siguientes vectores de entrada y salida:

inputs = [[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 0], [0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]]

outputs = [0, 1, 1, 0, 1, 0, 0, 1]

Los resultados obtenidos se pueden ver en la figura 5.37., donde se aprecia que la red necesita 25,000 iteraciones para llegar a unos resultados estables. Al ser la red más compleja que en los ejemplos anteriores se puede observar que esta se queda parada en dos mínimos locales antes de alcanzar el valor que se ha tomado como final. El primer salto, después de la gran caída inicial, se produce en torno a las 5.000 iteraciones y el segundo en torno a las 25.000.

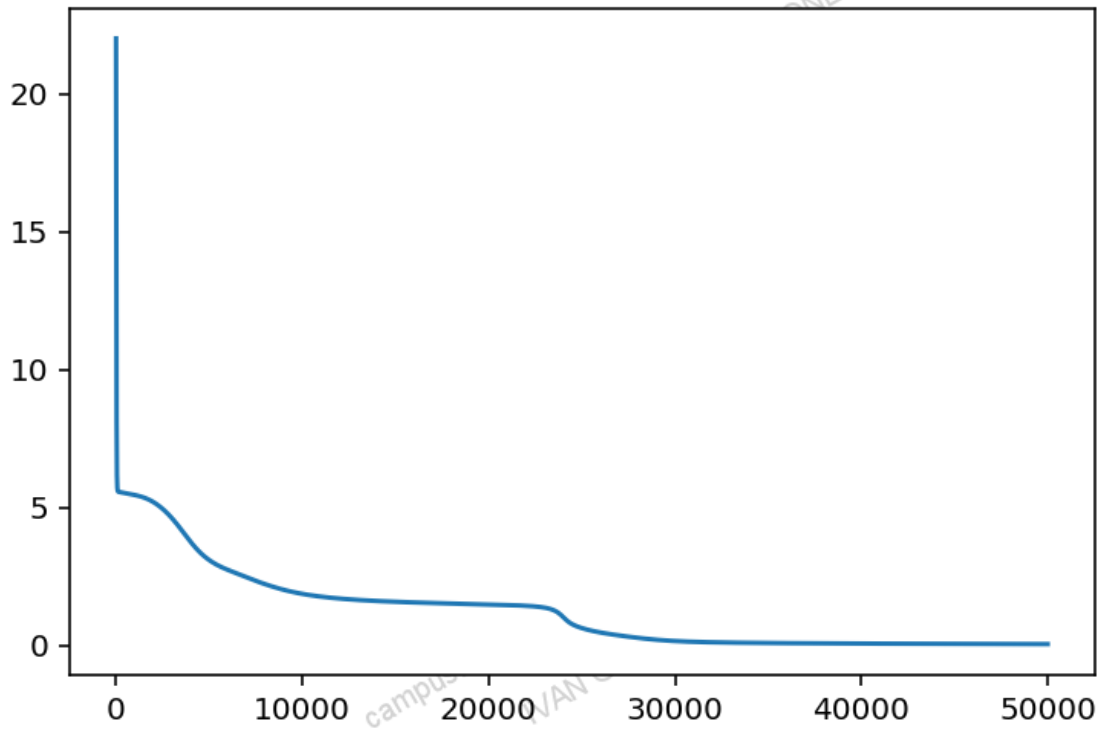


Figura 5.37. Evolución de la función de esfuerzo frente al número de evaluaciones.

El código completo del caso se puede encontrar en elnotebook U6_caso. Se adjunta además el siguiente png del caso

Recursos

Bibliografía

- ***Deep Learning Tutorial*, University of Montreal.** : LISA lab. [En línea] URL disponible en: http://www.cs.virginia.edu/yanjun/teach/2014f/lecture/L20-handout-deeplearningTutorial.pdf
- ***Designing Machine Learning Systems with Python*.** : Julian, D. Birmingham: Packt Publishing; 2016.
- ***Mastering Machine Learning With scikit-learn*.** : Hackeling, G. Birmingham: Packt Publishing; 2014.
- **Deep Learning.** : Goodfellow, I.; Bengio, Y. y Courville, A. Mit Press; 2016.

Glosario.

- **Dropout**: Técnica de regularización que consiste en apagar aleatoriamente un porcentaje de las neuronas durante el proceso de entrenamiento.

- **Función de activación**: En redes neuronales, es la función matemática que se utiliza en cada neurona para convertir la entrada en un valor de salida.

- **Perceptrón**: Puede hacer referencia a los primeros modelos de redes neuronales desarrollados o al modelo matemático básico de una única neuronal.

- **Red neuronal**: Son modelos inspirados en el comportamiento de las neuronas de los cerebros, que basan su inteligencia en las conexiones entre sistemas más simples llamados neuronas.

- **Sistemas conexionistas**: Son los modelos de inteligencia artificial en donde la inteligencia aparece debido a las conexiones de sistemas más simples.