Sistemas Cognitivos Artificiales

Frameworks de aprendizaje profundo

Índice

Esquema 3

Ideas clave 4

3.1. ¿Cómo estudiar este tema? 4

3.2. Frameworks de aprendizaje profundo 5

3.3. TensorFlow. Grafos de computación 6

3.4. Otros frameworks 14

3.5. Keras 17

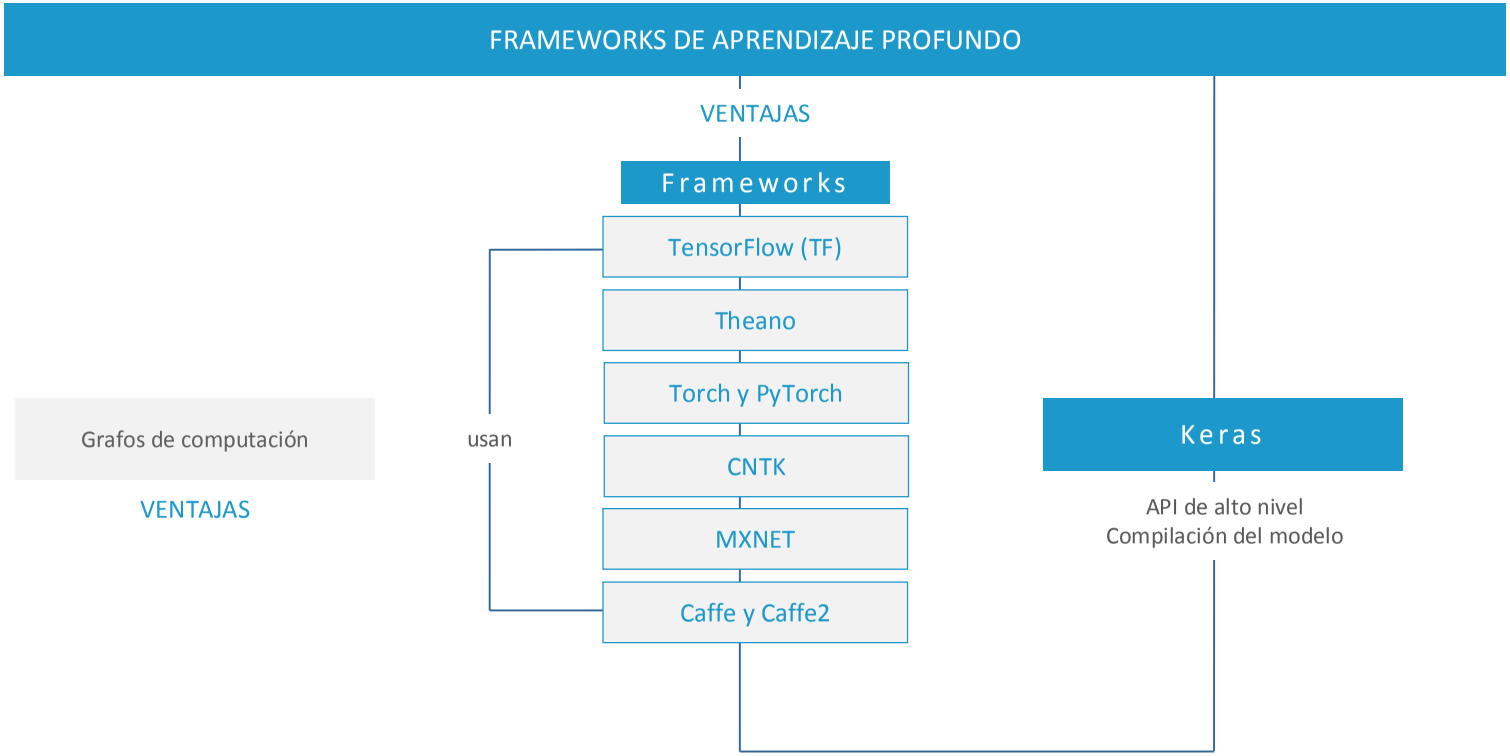
3.6. Referencias bibliográficas 20

Lo + recomendado 21

+ Información 23

Test 25

Esquema



Ideas clave

3.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

En este tema estudiaremos los frameworks de *deep learning* y qué ventajas aportan al entrenamiento de redes neuronales. Profundizaremos un poco más en TensorFlow (el framework más conocido) y en el concepto de tensor y grafo de computación. Asimismo, veremos otros framework utilizados en la actualidad y sus puntos fuertes. Finalmente, echaremos un primer vistazo a Keras, una librería de alto nivel que facilita aún más el entrenamiento de redes neuronales con Python. En este tema:

- Es necesario comprender el concepto de qué es un framework de aprendizaje profundo y por qué son importantes, relacionando lo visto en este tema con los conceptos básicos de entrenamiento de redes neuronales estudiados durante el curso.
- También es esencial comprender el concepto de grafo de computación y cómo TensorFlow lo utiliza como base del entrenamiento de modelos. Si bien no es necesario conocer con detalle su funcionamiento interno, de gran complejidad, sí que hay que tener una idea global de cómo se utiliza para entrenar redes neuronales y ser capaces de desarrollar modelos sencillos.
- El tema sirve igualmente como introducción a Keras, el cual usaremos durante la asignatura. Tenemos que entender su funcionamiento y moverse de manera ágil por su documentación.

3.2. Frameworks de aprendizaje profundo

Una de las consecuencias de la reciente popularización del *deep learning* ha sido el surgimiento de una gran cantidad de paquetes de software que facilitan enormemente el entrenamiento de redes neuronales. Como hemos visto, entrenar una red supone la realización de complejas operaciones sobre una gran cantidad de parámetros. Asimismo, implica el cálculo de gradientes que hemos de programar para que el entrenamiento sea correcto.

Si bien esto no es especialmente difícil para redes pequeñas, la cosa se complica con las arquitecturas complejas que iremos viendo durante el curso. El hecho de que haya gran cantidad de decisiones que tomar a la hora de entrenar una red (función de coste, unidad de activación, algoritmo de optimización…), así como operaciones críticas cuyo fallo puede hacer que una red no aprenda, ha llevado a la comunidad de *machine learning* a desarrollar soluciones que permiten facilitar la implementación de modelos de redes neuronales.

En nuestro caso, entenderemos un framework como una librería o conjunto de librerías de programación que facilita el desarrollo e implementación eficiente de redes neuronales o de algoritmos de *machine learning* en general.

Presentan grandes **ventajas** en tanto en cuanto:

Simplifica el desarrollo de grafos de computación de gran tamaño. La mayoría de frameworks entiende las redes neuronales como un grafo de computación, donde cada nodo en la red es una serie de operaciones a partir de los datos de los nodos anteriores. En vez de tener que codificar las neuronas y sus operaciones nosotros mismos, los frameworks nos dan una serie de abstracciones y operaciones ya disponibles que podemos conectar en forma de un grafo de computación. Esto abstrae al programador de gran parte de la complejidad en el desarrollo de grandes redes neuronales.

Calcula de manera automática los gradientes. Uno de los puntos más complicados en el entrenamiento de redes neuronales, especialmente cuando se utilizan funciones de creciente complejidad, es el de tener que calcular analíticamente y plasmar en código los gradientes necesarios para *backpropagation*. Como este es un proceso en el que es fácil equivocarse, tradicionalmente se hacían comprobaciones numéricas con gradientes aproximados para asegurarse que los cálculos eran correctos y la red neuronal se estaba entrenando correctamente. Los frameworks modernos realizan el *backward pass* de *backpropagation* de manera automática, infiriendo los gradientes correctos a partir del grafo de computación. De esta manera, el programador solo necesita definir la arquitectura de la red para obtener todos los gradientes de manera gratuita.

Facilita la ejecución y entrenamiento en GPU y entornos distribuidos. Como veremos más adelante durante el curso, las tarjetas gráficas (GPU) se utilizan con frecuencia para agilizar el entrenamiento de redes neuronales. El uso de frameworks nos permite utilizarlas de manera más sencilla, sin tener que programar código específico para correr en GPU. Similarmente, algunos frameworks como TensorFlow permiten simplificar la ejecución en sistemas distribuidos, donde es posible entrenar utilizando clusters de muchas máquinas.

Optimiza el entrenamiento y ejecución de los algoritmos. Del mismo modo, los frameworks suelen utilizar código altamente optimizado, en muchas ocasiones utilizando librerías numéricas que aprovechan al máximo las instrucciones especiales del procesador y sus capacidades multihilo.

3.3. TensorFlow. Grafos de computación

De todos los frameworks de *deep learning*, probablemente el más utilizado sea TensorFlow (TF): es *open-source* para computación numérica que utiliza grafos de computación donde los datos, en forma de tensores

(*Tensor*), «fluyen» (*flow*) entre distintas operaciones. En principio, TF puede utilizarse para una gran cantidad de aplicaciones que precisen de cálculos numéricos, si bien sus aplicaciones comunes giran en torno al *machine learning* y, más en particular, al aprendizaje profundo.

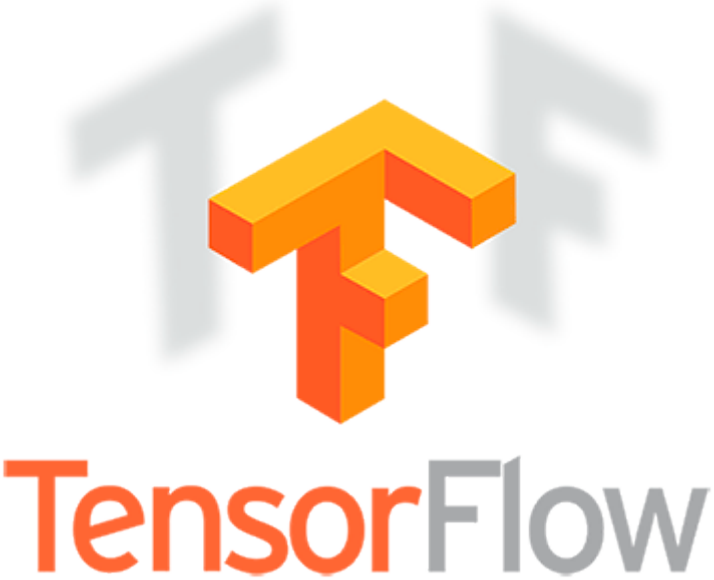


Figura 1. Logo de TensorFlow.

Fue desarrollado por Google, donde es utilizado tanto en investigación como para sistemas en producción. Su código fue liberado como *open-source* en 2015, y desde entonces se ha convertido en un ecosistema software muy completo, con una gran comunidad de usuarios.

Se utiliza principalmente con una API en Python, tal y como haremos en este curso, aunque tiene también API en otros lenguajes como C++ o Java. Sin embargo, la mayor parte de las operaciones y del núcleo de TF están programados en C++ para una mayor velocidad de ejecución.

Tensores y grafos de computación

Un tensor, la abstracción básica en TF, es básicamente un array n-dimensional. Por ejemplo:

- Si tenemos un array de una dimensión, hablamos de un **vector**.
- Si es de dos dimensiones, hablamos de una **matriz**.
- Igualmente, un tensor 0-dimensional no es más que un número, un **escalar** en términos algebraicos.

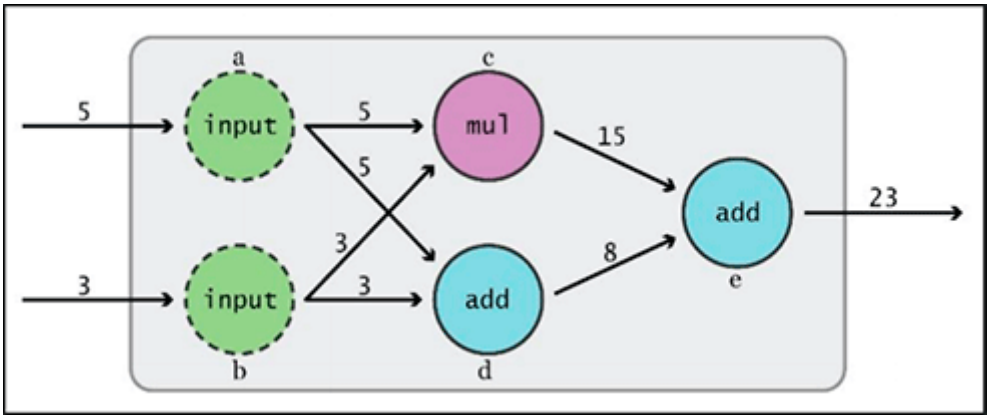


Figura 2. Ejemplo de grafo de computación.
Fuente: Abrahams, Hafner, Erwitt, y Scarpinelli, 2016.

Un **grafo de computación** es una representación de una serie de operaciones matemáticas. En particular, la representación consiste en un grafo dirigido donde los nodos se corresponden con operaciones y las aristas con datos (en nuestro caso, tensores). En la figura 3 se muestra un ejemplo de un grafo de computación para la función $f(x, y) = xy + (x + y)$.

Podemos ver cómo la entrada de la función se representa con dos nodos *input*, los cuales se conectan con dos operaciones: por un lado la multiplicación de x e y, y por otro lado su suma. Las salidas de estas dos operaciones son usadas para realizar una suma final.

Básicamente, TensorFlow funciona mediante la construcción de un grafo de computación y el uso de una sesión que ejecuta las operaciones en el grafo. Con la definición y construcción del grafo, el framework obtiene la información necesaria sobre qué operaciones han de ejecutarse, en qué orden y, además, se asegura de su correcta definición y compatibilidad. Una vez el grafo se ha construido, basta con ejecutar las operaciones con los datos necesarios.

Para aclarar esto, veamos un ejemplo de cómo construir y ejecutar un grafo de computación sencillo con TensorFlow:

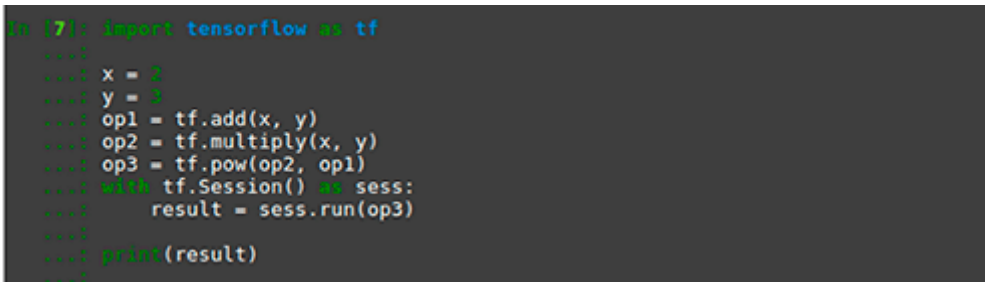


Figura 3. Grafo de computación básico en TF.

En este código puede verse cómo definimos dos variables y construimos un grafo mediante la utilización de primitivas de TF. Por ejemplo:

- `tf.add()` define la suma de dos elementos.
- `tf.multiply` define la multiplicación.

Podemos percibir también que, mediante la utilización de funciones de TF en vez de los operadores matemáticos de Python, estamos definiendo implícitamente la construcción de un grafo de computación; es posible ver el generado con herramientas de visualización como TensorBoard:

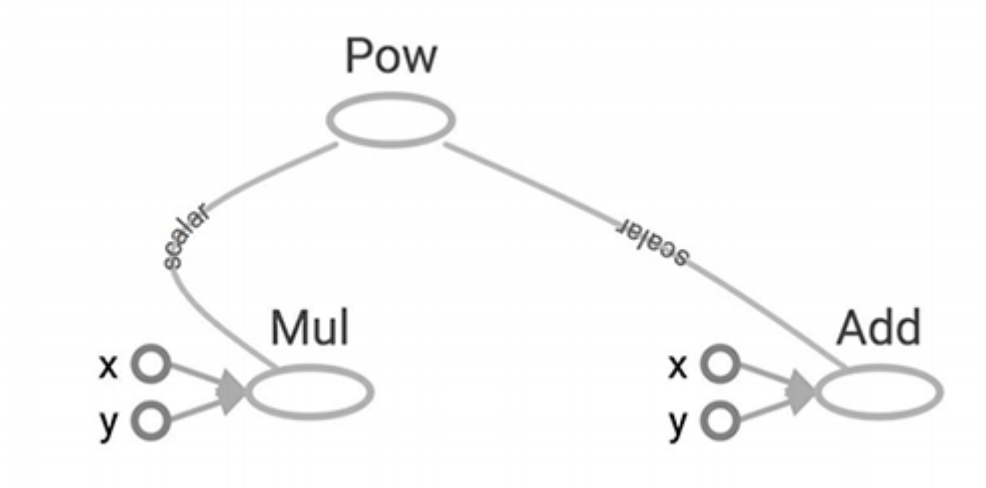


Figura 4. Visualización del grafo de computación del ejemplo anterior mediante TensorBoard.

Finalmente, la sesión instanciada mediante `tf.Session()` es el entorno donde se ejecutan las operaciones definidas en el grafo computacional, lo cual ocurre mediante una llamada a `run()`. La sesión reserva la memoria necesaria para todas las variables empleadas y se encarga de evaluar los tensores.

Es importante comprender que, en el código anterior, las operaciones no se están evaluando línea por línea; la variable `op1` no contiene el resultado de la suma de 3 y 2. Lo que el código está haciendo es construir el grafo de computación que, una vez evaluado con los valores numéricos particulares usados en este ejemplo (2 y 3), proveerá un resultado. De hecho, si ejecutamos el código podemos ver que la variable `op1` es de tipo `tensorflow.python.framework.ops.Tensor` y su contenido es `<tf.Tensor 'Add:0' shape=() dtype=int32>`. Esto no es más que la información que TF va generando sobre el grafo construido. En particular, nos indica varias cosas:

- El resultado de `tf.add()` aplicado sobre dos escalares de tipo entero es un tensor (`tf.Tensor`).
- Este tensor es el resultado de la operación definida como `Add:0`. TF va guardando las operaciones creadas con nombres únicos, así como las relaciones entre ellas. Por ejemplo, el hecho de que op3 dependa de `op1` y `op2` hace que TF guarde estas relaciones, de manera que las operaciones estén relacionadas en un único grafo.

- El tensor resultante es un escalar, un número, ya que la *shape* es una tupla vacía. Si el resultado fuera una matriz, por ejemplo, veríamos una *shape* del tipo (3,3), indicando que nos encontramos ante una matriz 3x3.
- El tensor resultante es de tipo `int32`. Esto implica que el tensor contiene enteros. Otros ejemplos de tipos válidos (definidos en `tf.DType`) son `float32`, `float64`, `bool` o `string`.

La evaluación o el momento en el que hacemos «fluir» los datos reales por el grafo y obtenemos el resultado ocurre, como ya hemos dicho, cuando ejecutamos `run()` sobre el objeto sesión. Al hacer `sess.run(op3)`, estamos evaluando el grafo de computación que acaba en `op3`. En ese momento, TF obtiene del grafo todas las operaciones que es necesario ejecutar (en este caso: `op1`, `op2` y `op3`) y rellena los valores de los tensores de salida mediante una evaluación nodo a nodo.

En el caso de que hubiéramos definido otras operaciones en el grafo que no son necesarias para la evaluación de `op3`, estas operaciones serían ignoradas.

Ventajas de los grafos de computación

¿Por qué TensorFlow y otros frameworks utilizan grafos de computación? Algunas de las razones por las que es conveniente utilizar grafos de computación son las siguientes:

Optimización de los cálculos a realizar: las operaciones que no son realmente necesarias no son ejecutadas. Igualmente, ya que todas las operaciones a realizar son conocidas de antemano, se pueden realizar optimizaciones sobre ellas que repercutan en una mayor velocidad de ejecución.

La ejecución se divide en trozos, lo que facilita la autodiferenciación. Cuando necesitamos aplicar *backpropagation*, TF recurre al grafo para ver qué operaciones hay que añadir con las derivadas automáticamente calculadas.

Facilita enormemente la ejecución distribuida en varias máquinas o GPU. Podemos dividir el grafo en varios subgrafos y hacer que distintos componentes de nuestro sistema distribuido se encarguen de diferentes partes. Por ejemplo, podemos asignar las operaciones de cómputo más complejas a la GPU mientras la CPU se encarga de leer y preprocesar datos.

Ejemplo completo en TensorFlow

Como muestra de un ejemplo más completo, aquí tenemos el siguiente código donde, utilizando regresión lineal, se intenta predecir la esperanza de vida a partir del número de hijos por familia.

Si bien en este curso no nos detendremos a explicar muchos de los detalles de TensorFlow que se ven en el ejemplo, es importante tener una idea de cómo se entrena un algoritmo de *machine learning* completo con este framework.

```
import tensorflow as tf

import utils

DATA_FILE = "data/birth_life_2010.txt"

# Step 1: read in data from the .txt file
# data is a numpy array of shape (190, 2), each row is a datapoint
data, n_samples = utils.read_birth_life_data(DATA_FILE)

# Step 2: create placeholders for X (birth rate) and Y (life expectancy)
X = tf.placeholder(tf.float32, name='X')
Y = tf.placeholder(tf.float32, name='Y')

# Step 3: create weight and bias, initialized to 0
w = tf.get_variable('weights', initializer=tf.constant(0.0))
b = tf.get_variable('bias', initializer=tf.constant(0.0))

# Step 4: construct model to predict Y (life expectancy from birth rate)
Y_predicted = w * X + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name='loss')

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)

with tf.Session() as sess:
                    # Step 7: initialize the necessary variables, in this case, w and b
                    sess.run(tf.global_variables_initializer())

                    # Step 8: train the model
                    for i in range(100): # run 100 epochs
                                        for x, y in data:
                                                            # Session runs train_op to
minimize loss
                                                            sess.run(optimizer,
feed_dict={X: x, Y:y})

                    # Step 9: output the values of w and b
                    w_out, b_out = sess.run([w, b])
```

Figura 5. Ejemplo completo de entrenamiento de un modelo de regresión lineal.
Fuente: https://docs.google.com/document/d/1kMGs68rIHWHifBiqlU3j_2ZkrNj9RquGTe8tJ7eR1sE/edit

Accede a más código en Github a través del aula virtual o desde la siguiente dirección:

https://github.com/chiphuyen/stanford-tensorflow-tutorials/blob/master/examples/03_linreg_placeholder.py

3.4. Otros frameworks

La popularidad del *machine learning* y del *deep learning,* en particular, ha hecho que la oferta en frameworks se haya multiplicado en los últimos años. La competencia es fuerte, con nuevos *benchmarks* saliendo cada poco tiempo, teniendo en cuenta lo que cuesta entrenar con cada librería algunas de las arquitecturas más populares en aprendizaje profundo. Aquí comentaremos varios de los más importantes.

Theano

Figura 6. Logo de Theano.

Theano es una librería en Python para computación numérica. Fue creado en 2007 por Yoshua Bengio y es uno de los primeros frameworks que empezaron a utilizarse en el mundo del aprendizaje profundo para optimizar los entrenamientos. Incluye soporte para utilizar GPU y es similar a TensorFlow en cuanto a que las operaciones se definen en forma de un grafo que se ejecuta posteriormente. De hecho, tanto TensorFlow como muchos de los framework modernos deben mucho a Theano, ya que utilizan y perfeccionan varias de las ideas presentes en este. Su desarrollo fue oficialmente interrumpido a finales de 2017.

Torch y PyTorch

Figura 7. Logos de Torch y PyTorch.

PyTorch es uno de los frameworks más utilizados en la actualidad. Proviene de Torch, otro de los frameworks «clásicos» de *deep learning* como Theano.

- Está escrito en Lua mientras que PyTorch (como su nombre indica) en Python, lo cual lo hace **más accesible** a la mayoría de desarrolladores e investigadores.
- Otra de las diferencias que sin duda hacen a PyTorch una versión más moderna de Torch es la **autodiferenciación**. PyTorch está desarrollado por Facebook.
- PyTorch utiliza grafos de **computación dinámica** en vez de grafos de computación estática, como hace TensorFlow. Un grafo de computación dinámico es calculado a la vez que el código se ejecuta, frente a TF que requiere de dos pasos: definir el grafo y luego ejecutarlo.

Los **grafos de computación dinámica** presentan dos claras ventajas:

- El código es más sencillo y más parecido a la programación procedural «de toda la vida», lo que además facilita en gran medida la depuración (*debuggear*) del código.
- El grafo puede cambiar de manera dinámica durante el entrenamiento. Ciertas arquitecturas tienen tamaños que van cambiando según el dato particular que se

está evaluando, lo cual hace que el grafo a ejecutar sea distinto en cada iteración. Los grafos de computación dinámica permiten directamente este caso de uso.

TensorFlow también dispone de una variante de ejecución con grafos de computación dinámica llamada Eager Execution, aunque fue añadida más tarde ante el avance de PyTorch y otros.

Caffe y Caffe2

Figura 8. Logos de Caffe y Caffe2.

Caffe también pertenece a los frameworks «clásicos» de *deep learning*. Está escrito en C++ y su énfasis está en los sistemas en producción con redes neuronales. A diferencia de gran parte de los otros frameworks, no es necesario escribir código para definir modelos: se definen en ficheros de configuración (*.PROTOTXT).

El uso de Caffe en la actualidad ha descendido mucho dadas sus dificultades para definir modelos más complejos como RNN y CNN, que requieren de larguísimos ficheros de configuración. Sin embargo, aún existen una gran cantidad de sistemas en funcionamiento utilizando modelos entrenados con este framework.

Caffe2, desarrollado de nuevo por Facebook, es la evolución de Caffe y funciona, de manera similar a TensorFlow, mediante grafos estáticos de computación y una interfaz en Python sobre un core escrito en C++.

Es interesante ver cómo Facebook trabaja en dos frameworks, PyTorch y Caffe2, donde el primero está más enfocado a la investigación y el segundo a la puesta de sistemas en producción. Esta estrategia es distinta a la seguida por Google con TF, que intenta ser un framework válido para ambos aspectos.

CNTK y MXNET

Figura 9. Logos de CNTK y MXNET.

Para acabar de dejar claro que el mundo de los framework de *deep learning* es un campo de batalla entre los gigantes de la tecnología, los dos últimos frameworks que mencionamos aquí son Microsoft Cognitive Toolkit (CNTK) y Apache MXNet (principalmente apoyado por Amazon). Ambos están pensados como dos frameworks muy escalables y de gran rendimiento. En particular, Amazon facilita la utilización de MXNet en AWS, su ecosistema en la nube.

3.5. Keras

TensorFlow y las otras librerías vistas en este tema son frameworks muy potentes y versátiles, capaces de implementar a bajo nivel toda suerte de redes neuronales innovadoras; por «bajo nivel» entendemos el nivel de operaciones matemáticas y de arquitectura de red. Sin embargo, en muchas ocasiones queremos utilizar una arquitectura estándar que nos gustaría definir rápidamente, evitando en la medida de lo posible tener que reescribir el código de los mismos elementos una y otra vez. Es por esto que casi todos los frameworks aquí mencionados disponen de API de alto nivel que permiten la definición de redes neuronales de una manera más sencilla y directa.

Figura 10. Logo de Keras.

Keras es una de estas librerías de alto nivel, probablemente la primera y más utilizada de ellas. Keras especifica una interfaz modular y *user-friendly* en Python para el desarrollo de redes neuronales, facilitando la tarea a gran parte de los desarrolladores que no necesitan definir arquitecturas de bajo nivel. Internamente, Keras funciona sobre TensorFlow, aunque también es posible usar como *backend* otros frameworks como Theano y CNTK.

La definición de modelos con Keras es realmente sencilla, como podemos ver en el siguiente ejemplo extraído de su página web:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Generate dummy data
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Figura 11. Red neuronal con Keras.
Fuente: https://keras.io/getting-started/sequential-model-guide/

En el ejemplo podemos ver varios de los elementos estudiados durante el curso. Primero, se generan datos aleatorios. Después, se define el modelo como un modelo secuencial mediante `model = Sequential()`.

Keras dispone de dos estilos de API: **secuencial**, más sencillo y directo, y **funcional**, más versátil, permitiendo arquitecturas más complicadas.

Acto seguido, se van añadiendo al modelo las distintas capas. En este caso, se añaden tres capas `Dense` (*fully-connected layers*), las dos primeras de 64 unidades y con `'relu'` como unidad de activación.

La última capa tiene 10 unidades con una activación `softmax`, lo que indica que es la capa final y que nos encontramos ante un problema de clasificación con 10 clases. Como medida de regularización, entre medias de las distintas capas se añaden unidades de `Dropout`.

Una vez que la arquitectura de la red está definida, hay que compilar el modelo. Para ello, es necesario especificar una *loss function* y un optimizador. En este caso, debido a que se trata de un problema de clasificación con `softmax`, la *loss function* es `categorical_crossentropy`, mientras que el algoritmo de optimización es nuestro buen amigo *stochastic gradient descent*, en este caso aderezado con `learning rate decay` y `Nesterov momentum`.

La **compilación del modelo** permite a Keras comprobar que las dimensiones especificadas tienen sentido y que los distintos elementos del modelo son compatibles entre sí. Finalmente, para entrenar, hacemos una llamada a `fit()`.

> **Página oficial de Keras:** La página oficial de Keras contiene toda la documentación necesaria para utilizar esta librería así como una gran lista de ejemplos. Imprescindible leer el *Getting started*.
>
> Puedes verlo en el siguiente enlace:
> https://keras.io/

3.6. Referencias bibliográficas

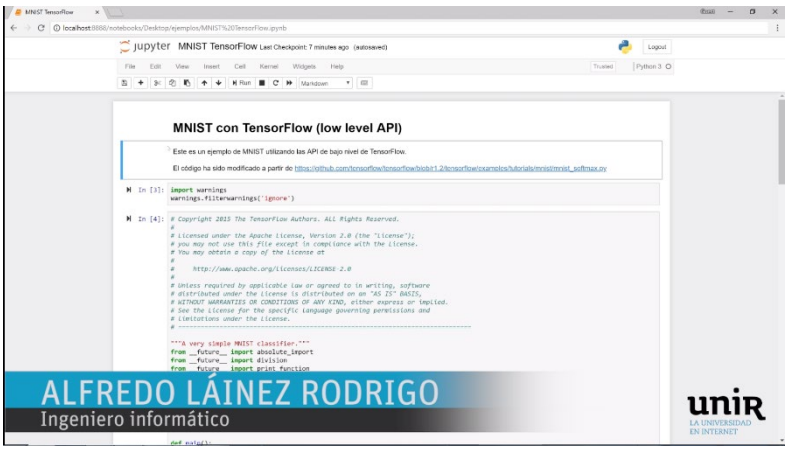
Abrahams, S., Hafner, D., Erwitt, E. y Scarpinelli, A. (2016). *TensorFlow for Machine Intelligence: A hands-on introduction to learning algorithms*. [Lugar desconocido]: Bleeding Edge Press.

Lo + recomendado

Lecciones magistrales

Entrenamiento de una red neuronal con TensorFlow

Abordaremos de nuevo el entrenamiento de una red neuronal en MNIST y utilizaremos las API de bajo nivel de TensorFlow. Asimismo, veremos cómo se construye y se ejecuta un grafo.



Accede a la lección magistral a través del aula virtual

No dejes de leer

TensorFlow Eager Execution

Shankar, A. y Dobson, W. (31 de octubre de 2017). Eager Execution: An imperative, define-by-run interface to TensorFlow [Blog post].

Este artículo explica la llegada a TensorFlow del modo Eager Execution. Interesante para comprender el cambio de paradigma.

Accede al artículo a través del aula virtual o desde la siguiente dirección web: https://ai.googleblog.com/2017/10/eager-execution-imperative-define-by.html

Lecture note 3: Linear and Logistic Regression

Huyen, C. (Sin fecha). Lecture note 3: Linear and Logistic Regression in TensorFlow [Lecture notes].

Documento con las *lecture notes* de las dos primeras clases, un gran compendio de información básica para programar con TensorFlow y, por lo tanto, un recurso muy recomendado.

Accede al documento a través del aula virtual o desde la siguiente dirección web: https://docs.google.com/document/d/1kMGs68rIHWHifBiqlU3j_2ZkrNj9RquGTe8tJ7eR1sE/edit

+ Información

A fondo

Comparativa de deep learning software

Comparison of deep learning software. [Actualizado el 20 de agosto de 2018]. En *Wikipedia*.

Esta tabla, continuamente actualizada, ofrece una comparativa de frameworks y librerías utilizadas en el mundo del *deep learning*.

Accede a la tabla a través del aula virtual o desde la siguiente dirección web:
https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Webgrafía

Página oficial de TensorFlow

La página oficial de TensorFlow está llena de información sobre el framework, incluyendo documentación extensiva sobre la API y tutoriales para principiantes.

Accede a la página web a través del aula virtual o desde la siguiente dirección:
https://www.tensorflow.org/

TensorFlow for Deep Learning Research

Página web de un completo curso sobre TensorFlow de la universidad de Stanford; es una guía muy completa con abundante código de buena calidad. En particular, como parte de este tema, hemos visto el ejemplo completo de *Linear Regression*.

Accede a la página web a través del aula virtual o desde la siguiente dirección:

http://web.stanford.edu/class/cs20si/

Test

1. Los grafos de computación (marca todas las respuestas correctas)*:*

A. Facilitan el cálculo de derivadas automáticas.

B. Facilitan la ejecución de subtareas en entornos distribuidos.

C. Implican normalmente menor velocidad de ejecución ya que tienen que ser definidos de antemano.

2. En el siguiente código

```
x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf. multiply(x, y)
mul_op_2 = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
    z = sess.run(pow_op)
```

add_op es:

A. 5.

B. Un tensor definido por la operación `tf.add()` sin valor definido.

C. Un tensor definido por la operación `tf.add()` con valor 5.

3. En el siguiente código

```
x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf. multiply(x, y)
mul_op_2 = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
    z = sess.run(pow_op)
```

`mul_op_2` (marca todas las respuestas correctas):

A. No es utilizado en el resto del grafo y, por tanto, es incorrecto.

B. No es evaluado como parte de `sess.run(pow_op)`.

C. Sería evaluado junto con mul_op y add_oip si hiciéramos `sess.run(mul_op_2)`.

D. Sería evaluado junto con `add_op` si hiciéramos `sess.run(mul_op_2)`.

4. En el siguiente código

```
x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf. multiply(x, y)
mul_op_2 = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
    z = sess.run(pow_op)
```

`z` (marca todas las respuestas correctas):

A. Es un entero y tiene valor 15625.

B. Es un objeto de tipo `tf.Session`.

C. Es un objeto de tipo `tf.Session.run()`.

D. Es un objeto de tipo `tf.Tensor`.

5. Selecciona todas las sentencias verdaderas sobre grafos de computación estáticos y dinámicos:

A. Un grafo de computación estático necesita ser definido antes de evaluarlo con datos.

B. Tanto los grafos de computación estáticos como dinámicos pueden ser evaluados mientras se definen.

C. Los grafos de computación dinámicos permiten la utilización de sentencias de control como `IF/ELSE` de manera sencilla, ya que las instrucciones son evaluadas al momento.

D. Los grafos de computación estáticos permiten la utilización de sentencias de control como `IF/ELSE` de manera sencilla, ya que las instrucciones son evaluadas al momento.

6. TensorFlow (marca todas las respuestas correctas):

A. Es un framework de alto nivel y por tanto es complicado definir nuevos conceptos matemáticos y numéricos

B. Tiene una interfaz en Python, pero los cálculos numéricos se hacen en C++.

C. Utiliza C++ para los cálculos numéricos en vez de Python por su mayor velocidad de cómputo.

D. Utiliza Python para los cálculos numéricos porque es un lenguaje muy utilizado por la comunidad.

E. Se utiliza solo para *deep learning*.

7. Marcar todos los framework que utilizan grafos de computación:

A. TensorFlow

B. TensorFlow en su variante Eager Execution

C. PyTorch

D. Theano.

8. Marca todas la respuesta correcta acerca de Keras:

A. Es utilizado normalmente en investigación para definir novedosas arquitecturas de bajo nivel.

B. Es una librería derivada de TensorFlow.

C. Es una librería de alto nivel que define una interfaz limpia y sencilla para el entrenamiento de redes neuronales.

9. Cuando un modelo con TensorFlow es definido, hemos visto que se añade un optimizador que minimiza una *loss function*. Como sabemos, esto implica que es necesario obtener los gradientes que tienen que ser utilizados durante el entrenamiento. ¿Cómo afecta esto al grafo de computación que se calcula para el entrenamiento del modelo? (marca la respuesta correcta):

A. No afecta al grafo, ya que el cálculo de gradientes no es una operación y no es necesario actualizar ningún tensor.

B. Al definir el optimizador como parte del modelo, TensorFlow calcula las operaciones de gradientes necesarias a partir del grafo actual y las añade a este, de modo que es posible evaluar los gradientes y aplicar *gradient descent* durante la ejecución.

C. No afecta al grafo. Los gradientes son computados de manera automática mediante mecanismos ajenos al grafo de computación.

10. En un modelo secuencial con Keras (marca la respuesta correcta):

A. Las distintas capas se van añadiendo una detrás de otra y el modelo se compila con un optimizador.

B. Las distintas capas se añaden al estilo de un grafo de computación y el modelo se compila con un optimizador.

C. No existe el concepto de modelo secuencial en Keras.