FACULTAD DE INGENIERIA
Universidad de Buenos Aires

75.23 - 95.25 Inteligencia Artificial

Trabajo Práctico Final

Alumnos:

Agustín Contini - 89180
Martina Ágata Panetta - 103713
Matías Ruiz Huidobro - 102251

Docente:

José Luis Cabrera

Primer Cuatrimestre del 2020

Introducción

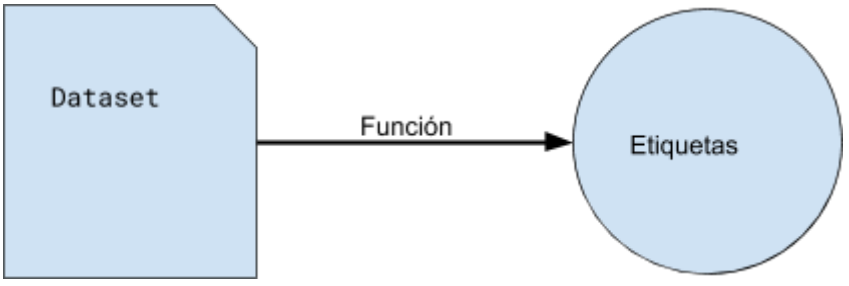
En este documento se analizarán tres ejemplos de redes neuronales artificiales, entendiéndose éstas como modelos computacionales para la resolución de problemas (o simplemente para su modelado) basados en la emulación de las redes neuronales biológicas, es decir, utilizandose redes interconectadas en paralelo que buscan interactuar del mismo modo que lo hace un sistema nervioso biológico.

En los ejemplos se utilizará *Keras*; una biblioteca de redes neuronales de código abierto escrita en Python, y como framework de base se utilizará *TensorFlow*, una librería de código abierto de Google para aprendizaje automático.

¿Qué es una red neuronal?

Es un algoritmo de machine learning que permite, a partir de un conjunto de datos inicial, generar nueva información. De esta forma una red neuronal puede partir de un grupo de imágenes clasificadas y tras ser entrenado clasificar nuevas imágenes. Otros ejemplos recientes de usos de este tipo de “Machine Learning” incluyen, estudios del cuerpo humano, clasificación de tumores, ciberseguridad, reconocimiento facial, reconocimiento de texto, reconocimiento de voz, reconocimiento de emociones y vehículos auto-controlados.

La red neuronal obtiene estos resultados partiendo de un conjunto de datos ya “etiquetados” y buscando una función que relaciona los datos con sus etiquetas.



Esta función se obtiene durante la etapa de aprendizaje, el cual puede ser o no supervisado.

Para encontrar esta función utilizará prueba y error, pero irá ajustando los parámetros de la función inteligentemente de acuerdo al grado de error, para acercarse al objetivo. Estos parámetros se ajustan durante iteraciones, con el objetivo de que cada iteración sea más precisa que la anterior. Como las combinaciones de valores para los parámetros es infinita esta red neuronal siempre tendrá un grado de error y dependerá del problema en cuestión cuando la red tenga un error tolerable. Estos ajustes son realizados en las neuronas de cada capa de la red neuronal, donde cada neurona recibe una o más entradas, le aplica los pesos a las entradas y entrega un valor, resultado de una función de activación. Estas capas pueden estar configuradas de diversas maneras, y para encontrar la mejor configuración para el problema que se enfrenta se prueban distintas combinaciones en busca de la que entregue los mejores resultados.
Cuando se busca aplicar una red neuronal para resolver un problema hay varias características que se deberán tener en cuenta. Algunas de estas son:

- Cantidad de capas de neuronas, siempre hay una capa de entrada y una de salida, pero puede haber varias capas intermedias.
- Cantidad de neuronas por capa.
- Conectividad entre las capas.
- Funciones de activación para las neuronas.
- Algoritmo o función para ajustar los pesos. (Suele utilizarse variaciones de Descenso por el Gradiente).
- Función de error. Esta función alimentará al algoritmo de ajuste para que este decida cómo ajustar los pesos.

Además es imprescindible partir de un conjunto de datos abarcativo para el problema en cuestión, ya que estos datos son la base sobre la que la red neuronal construirá las predicciones.

Herramientas

Sobre Keras y TensorFlow

Keras es una deep learning API de alto nivel que corre sobre la plataforma de TensorFlow, con su estructura core de datos de capas y modelos. Permite construir también grafos de capas arbitrarias, incluso modelos completamente de 0 a través del subclassing.

- Inicialmente fue desarrollada como parte del proyecto ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System) con la idea de dar una herramienta que permita experimentar de forma rápida, pasando a obtener resultados tempranos a las ideas surgidas, como base para sustentar las investigaciones.

- Se destaca por ser accesible, una interfaz muy buena para resolver problemas de machine learning, enfocado principalmente en el deep learning.

- La modalidad en bloques y las abstracciones que provee permite desarrollar rápidamente soluciones de machine learning.

- Se destaca a su vez como un framework muy flexible que permite resolver desde cosas muy sencillas hasta de mayor complejidad, apoyándose en un aprendizaje escalonado de rápido inicio, para ir avanzando en etapas a los escenarios más complejos cuando se vuelve necesario.

Ventajas en la utilización de Keras en conjunto con TensorFlow

Keras se enriquece de esta plataforma open-source end-to-end de machine learning que permite ejecutar de forma eficiente cálculos tensoriales tanto sobre CPU, GPU o TPU; de fácil escalabilidad y que pueden ser fácilmente trasladados a otros entornos como el browser, dispositivos móviles, etc. Se puede correr Keras en TPU o en clusters de GPUs así como exportar los modelos al browser o incluso a dispositivos móviles.

Descarga e instalación

Debido a que Keras es una biblioteca escrita en Python, es requisito previo poseerlo instalado. Su última versión puede obtenerse en: https://www.python.org/downloads/

Instalación mediante terminal de Linux para Python 3:

Instalación de Pip, Tensorflow y Keras:

```
sudo apt install python3-pip
pip install --upgrade pip
sudo pip3 install -U virtualenv
pip3 install --upgrade tensorflow
pip install keras (instalación de keras)
```

Instalación de librerías:

```
pip install numpy scipy
pip install scikit-learn
pip install pillow
pip install h5py
```

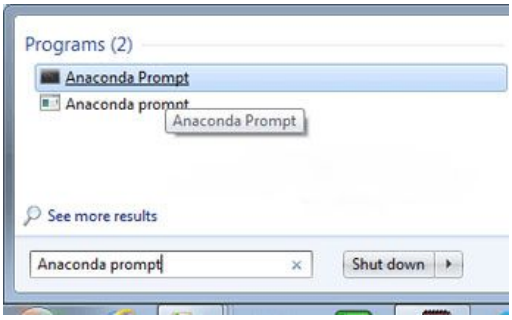
Revisar en ~/.keras/keras.json el archivo json (fragmento a continuación) que “image_data_format” (orden de dimensión de imagen) sea "channels_last" y backend esté seteado en Tensorflow.

```
{
    “image_data_format”: “channels_last”,
    “epsilon”: 1e-07,
    “floatx”: “float32”,
    “backend”: “tensorflow”
}
```

Instalación en Windows:

1. Instalar Conda: https://docs.conda.io/projects/conda/en/latest/user-guide/install/windows.html

2. Una vez Conda esté instalado, abrir el Prompt de Conda como administrador:



3. Ejecutar el comando: `conda install -c conda-forge keras`

4. Para comprobar si la instalación de Keras ha sido correcta abrimos Anaconda y ejecutamos el siguiente código:

```
from keras.models import Sequential
model = Sequential()
```

   Si todo ha sido correcto se debería leer en la terminal:

```
Using TensorFlow backend.
```

5. Agregar las dependencias adicionales requeridas por el modelo que se desea probar:
   a. Para el Ejemplo 1 y el Ejemplo del Anexo:
      i. `conda install -c conda-forge matplotlib`
      ii. `conda install -c anaconda numpy`

   b. Para el Ejemplo 2:
      i. `conda install -c anaconda numpy`

Método

En esta sección definiremos brevemente las herramientas y procedimientos que se utilizarán en los tres ejemplos de implementación de redes neuronales realizados.

Contemplaremos como pasos estructurales que cumple toda implementación de redes neuronales aquellos enlistados ordenadamente en la siguiente figura:

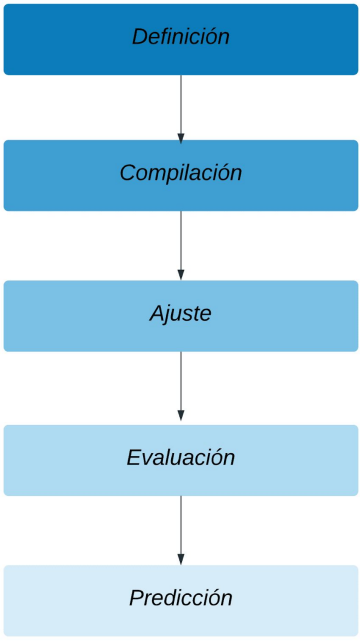


Figura 1. Pasos en la utilización de redes neuronales.

A continuación, una interpretación de estos pasos específicamente en el marco de Keras:

1. **Definición:** En Keras las redes neuronales se definen como una secuencia de capas que se contienen en la clase “Sequential”, debido a esto, en primer lugar debe crearse una instancia de la clase y añadir las capas en el orden en que deben conectarse.
   Las capas pueden caracterizarse de diversas formas; por ejemplo, uno de los tipos más importantes es el “Dense” que denota que cada neurona recibe información de todas las neuronas de la capa anterior, o sea, el output de todas las neuronas de una capa es el input de las neuronas de la capa siguiente, por lo tanto, está densamente conectada o es "altamente conexa".
   Otro tipo importante es “Flatten”. En Keras, utilizar “Flatten” en un tensor implica eliminar todas las dimensiones excepto una, obteniéndose así una matriz unidireccional de elementos.

Otra característica importante en esta etapa es la elección de la función de activación ya que definirá la forma en que se realizarán las predicciones. Las funciones de activación deciden si una neurona debe activarse o no calculando la suma ponderada y agregando más sesgo con ella.
La más usada en deep learning (y que usaremos en los ejemplos en general) es la Unidad Lineal Rectificada (RELU). Se trata de una función sencilla que devuelve 0 si recibe alguna entrada negativa pero para cualquier valor positivo x devuelve ese mismo valor. Por lo tanto, se puede escribir como f(x) = máx(0, x).

2. **Compilación:** Es un paso necesario tras definir el modelo, ya que transforma la secuencia simple de capas que definimos en una serie muy eficiente de transformaciones matriciales en un formato destinado a ejecutarse en la GPU o CPU según sea el caso. Incluye tanto definir un esquema de optimización como cargar un conjunto de pesos pre-entrenados desde un archivo guardado. Su importancia radica en que prepara una representación eficiente de la red que se requerirá para hacer predicciones en el hardware. Requiere de la especificación de varios parámetros diseñados específicamente para entrenar la red, como el algoritmo de optimización que se utilizará para entrenar la red (cuyo fin es definir los atributos de una red neuronal como pesos, tasa de aprendizaje, etc. para reducir pérdidas proporcionando resultados lo más precisos posibles) y la función de pérdida utilizada para evaluar la red que el algoritmo de optimización minimiza.

   En estos ejemplos utilizaremos como algoritmo de optimización “Adam”, para el cual no entraremos en detalles pero podemos decir que a grandes rasgos se trata de una combinación de *Root Mean Square Propagation* (RMSprop) y *Descenso de Gradiente Estocástico* (SGD) con impulso. Adam es un método de tasa de aprendizaje adaptativo, o sea, calcula tasas de aprendizaje individuales para diferentes parámetros; usa estimaciones del primer y segundo momento de gradiente para adaptar la tasa de aprendizaje para cada peso de la red neuronal.

3. **Ajuste:** Tras la compilación, la red puede ajustarse para adaptar los pesos a un conjunto de datos de entrenamiento; una matriz de patrones de entrada y una de patrones de salida.
   La red se entrena por medio del algoritmo de retropropagación (automáticamente) y se optimiza de acuerdo con el algoritmo de optimización y la función de pérdida seleccionados anteriormente. El algoritmo de retropropagación requiere que la red esté entrenada durante un número específico de épocas o exposiciones al conjunto de datos de entrenamiento. Cada época puede dividirse en grupos de pares de patrones de entrada-salida (lotes), lo que define el número de patrones a los que la red está expuesta antes de que los pesos se actualicen dentro de una época.

4. **Evaluación:** Se realiza tras entrenar la red. Puede evaluarse el rendimiento de la red en un conjunto de datos separado, invisible durante las pruebas, lo cual puede proporcionar una estimación del rendimiento de la red para hacer predicciones en el futuro de datos no vistos.
   El modelo evalúa tanto la pérdida (loss) en todos los patrones de prueba como cualquier otra métrica especificada cuando se compiló el modelo y se obtiene una lista de métricas de evaluación.
   La evaluación permite identificar si estamos satisfechos con el rendimiento del modelo de ajuste y, en caso contrario, tomar medidas para mejorarlo.

5. **Predicción:** Después de una evaluación con resultados satisfactorios ya se puede utilizar el modelo para realizar predicciones sobre nuevos datos, las cuales se obtendrán en el formato proporcionado por la capa de salida de la red.
   Para un problema de clasificación binaria, las predicciones pueden ser una matriz de probabilidades para la primera clase que se puede convertir en 1 o 0 al redondear, mientras que para un problema de clasificación multiclase, los resultados pueden tener la forma de una matriz de probabilidades que puede necesitar convertirse en una predicción de salida de clase única usando la función de numpy “argmax”.

Ejemplo I - Análisis de sentimiento

Como primer acercamiento a Keras, realizamos un ejemplo de red neuronal con el “IMDB movie review sentiment classification dataset”. Este dataset es utilizado para la clasificación binaria de reseñas (positivas o negativas) de películas. Consiste en 25.000 reseñas tomadas de IMDB, etiquetadas según sentimiento (positivo o negativo). Estas reseñas ya han sido pre procesadas, y cada una codificada como una secuencia de índices (enteros). Estos índices representan palabras que aparecen en la reseña. Estas palabras fueron indexadas según su frecuencia en el dataset, es decir el índice 5 representa a la quinta palabra más frecuente en el dataset. Esto permite un rápido filtro durante operaciones tales como considerar solo las 5000 palabras más frecuentes del vocabulario y otros.

Ejemplo de dato (review) del dataset:

Review: # please give this one a miss and the rest of the cast rendered terrible performances the show is flat flat flat i don't know how michael madison could have allowed this one on his plate he almost seemed to know this wasn't going to work out and his performance was quite # so all you madison fans give this a miss

Label: 0 -> Sentimiento Negativo

En el siguiente código Python perteneciente al archivo nombrado “movies.py” se presentan los cinco pasos para la implementación y utilización de redes neuronales para el caso del identificador de dígitos propuesto. Se detalla en los comentarios el significado de cada operación realizada.

Este dataset, como varios otros de Keras que utilizaremos en otros ejemplos, tiene la opción de ser “desempaquetado” como dos tuplas por medio de la utilización de la función load_data(), obteniéndose dos tuplas correspondientes a imágenes y etiquetas de entrenamiento, y otras otras dos que corresponden también a imágenes y etiquetas pero de prueba.

```
##movies.py
# Importar Keras, TensorFlow y otras librerías útiles
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

# Importar el dataset
imdb_dataset = keras.datasets.imdb

# Al cargar el dataset se obtienen 4 arreglos NumPy
maximum_index = 20000

(train_data, train_labels), (test_data, test_labels) =
imdb_dataset.load_data(num_words=maximum_index)


print("TRAIN DATA:", train_data[0], "....\n")
print("TRAIN LABELS:", train_labels)

# Si quisiéramos cambiar la proporción de training data vs test data podemos concatenar los
datasets y volver
# a dividirlos

all_data = np.concatenate((train_data, test_data), axis=0)
all_labels = np.concatenate((train_labels, test_labels), axis=0)

# Ejemplo de como se ve una de las reviews
indice_ejemplo = 0

index = imdb_dataset.get_word_index()
reverse_index = dict([(value, key) for (key, value) in index.items()])
decoded = " ".join( [reverse_index.get(i - 3, "#") for i in test_data[indice_ejemplo]] )
print("Review: ", decoded, "\n")
print("Label: ", test_labels[indice_ejemplo])

# Normalización del training y test set.
def normalizar(secuencias, dimension = maximum_index):
    # Relleno con 0 cuando la dimensión es menor
    normalizado = np.zeros((len(secuencias), dimension))
    for i, secuencia in enumerate(secuencias):
        normalizado[i, secuencia] = 1
    return normalizado

normalized_data = normalizar(all_data)
```

```
# Convierto los labels a floats
float_labels = np.array(all_labels).astype("float32")
# Separo el dataset en los casos utilizados para entrenamiento y para pruebas.
factor_entrenamiento = 80 # Porcentaje para entrenamiento, el resto para testing
cantidad_casos = len(normalized_data)
cantidad_entrenamiento = (int)((cantidad_casos * factor_entrenamiento) / 100)
test_data = normalized_data[cantidad_entrenamiento:]
test_labels = float_labels[cantidad_entrenamiento:]
train_data = normalized_data[:cantidad_entrenamiento]
train_labels = float_labels[:cantidad_entrenamiento]
model = keras.models.Sequential()
# Input - Layer
model.add(keras.layers.Dense(50, activation = "relu", input_shape=(maximum_index, )))
# Hidden - Layers
model.add(keras.layers.Dropout(0.3, noise_shape=None, seed=None))
# Dropout selecciona neuronas al azar para ignorar durante el entrenamiento, previene
overfitting
model.add(keras.layers.Dense(50, activation = "relu"))
model.add(keras.layers.Dropout(0.2, noise_shape=None, seed=None))
model.add(keras.layers.Dense(50, activation = "relu"))
# Output- Layer
model.add(keras.layers.Dense(1, activation = "sigmoid"))
model.summary()

# Compila el modelo, utiliza Adam como optimizador
model.compile(
 optimizer = "adam",
 loss = "binary_crossentropy",
 metrics = ["accuracy"]
)
results = model.fit(
 train_data, train_labels,
 epochs = 4,
 batch_size = 500,
 validation_data = (test_data, test_labels)
)
print("Precisión del modelo:" , np.mean(results.history["val_accuracy"]))

# Ejemplo tomo una review
review_index = 0
decoded = " ".join( [reverse_index.get(i - 3, "#") for i in all_data[review_index]] )
print("Reseña: ", decoded, "\n")
print("Esperado : ", all_labels[review_index])
### Ejemplo de predicción con modelo entrenado ###
prediccion = model.predict(test_data)

print(prediccion)
print("Predicción:" , prediccion[review_index])
```

Ejemplo II - Identificador de imágenes

En este ejemplo vamos a implementar un identificador de imágenes de 10 categorías utilizando como dataset al CIFAR-10 compuesto por 60k imágenes de 32x32 píxeles RGB, 6k imágenes pertenecientes a cada una de las siguientes categorías:

- 0: avión
- 1: auto
- 2: pájaro
- 3: gato
- 4: ciervo
- 5: perro
- 6: rana
- 7: caballo
- 8: barco
- 9: camión

CIFAR-10 es un dataset muy estudiado y muy utilizado en benchmarking de algoritmos de machine learning. Las imágenes son muy pequeñas y de muy baja resolución, lo que puede explicar el por qué incluso con alguno de los mejores algoritmos de hoy en día, la performance obtenida sobre el dataset es limitada.

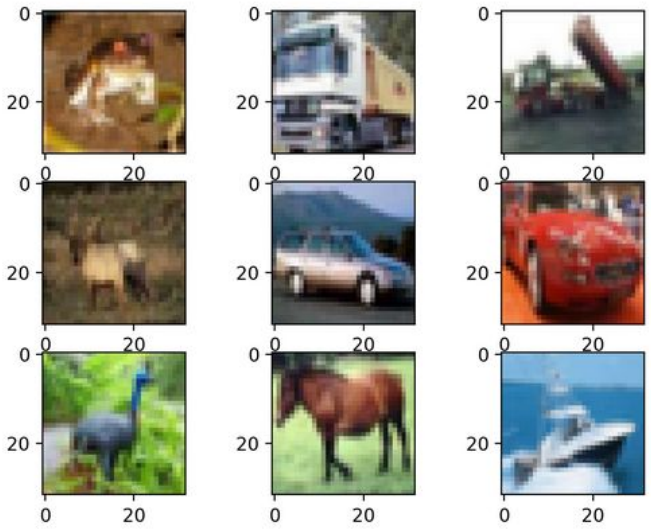


Figura 1. Subset del dataset del CIFAR-10.

A continuación, se presenta el código que carga el set de datos, construye y compila el modelo, lo evalúa y finalmente se guarda para su posterior utilización:

```
##imageCNN.py
import tensorflow as tf
import numpy
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, BatchNormalization,
Activation
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.constraints import max_norm
from tensorflow.keras import utils
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

# Ponemos un seed random para poder reproducir
seed = 21
chosenActivation = 'relu'
chosenOptimizer = 'Adam'
# Cargamos el dataset para entrenar y testear
def load_dataset():
    # Cargar dataset
    (X_train, y_train), (X_test, y_test) = cifar10.load_data()
    # one-hot encode: representación de la clase como vector binario
    y_train = to_categorical(y_train)
    y_test = to_categorical(y_test)
    return X_train, y_train, X_test, y_test

# Normalizamos las imágenes entre [0,1]
def scale_pixels(train, test):
    # Convertimos de integers a floats
    train_norm = train.astype('float32')
    test_norm = test.astype('float32')
    # Normalizamos a rango 0-1
    train_norm = train_norm / 255.0
    test_norm = test_norm / 255.0
    # Devolvemos imágenes normalizadas
    return train_norm, test_norm
```

```
# Defino modelo CNN
def define_model(X_train, class_num):
    model = Sequential()
    # Relu es la activación más común, y padding='same' solo significa que no estamos cambiando el tamaño de la imagen
    model.add(Conv2D(32, (3, 3), input_shape=X_train.shape[1:], activation=chosenActivation, padding='same'))

    # Capa de exclusión para evitar el sobreajuste, que funciona al eliminar al azar algunas de las conexiones entre las capas (0.2 significa que elimina el 20% de las conexiones existentes)
    model.add(Dropout(0.2))
    # Normalización de lotes: normaliza las entradas que se dirigen a la siguiente capa, asegurando que la red siempre cree activaciones con la misma distribución que deseamos
    model.add(BatchNormalization())
    # Otra capa convolucional, pero el tamaño del filtro aumenta para que la red pueda aprender representaciones más complejas
    model.add(Conv2D(64, (3, 3), activation=chosenActivation, padding='same'))

    # Capa de agrupación: ayuda a hacer que el clasificador de imágenes sea más robusto para que pueda aprender patrones relevantes.
    # También está la deserción y la normalización de lotes.
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())

    # Repetimos capas para darle a la red más representaciones para usar
    model.add(Conv2D(128, (3, 3), padding='same'))
    model.add(Activation(chosenActivation))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())

    # Flatten los datos + capa de abandono
    model.add(Flatten())
    model.add(Dropout(0.2))

    # Creamos la primera capa densamente conectada
    # Necesitamos especificar el número de neuronas en la capa densa.
    # El número de neuronas en las capas siguientes disminuye, acercándose al mismo número de neuronas que hay de clases en el conjunto de datos (en este caso 10). La restricción del kernel puede regularizar los datos a medida que se aprende, otra cosa que ayuda a evitar el sobreajuste.
```

```
    # En la capa final, cada neurona representa una clase, y la salida de este capa será un vector de 10 neuronas, con cada neurona almacenando una probabilidad de que la imagen en cuestión pertenezca a la clase que representa
    model.add(Dense(256, kernel_constraint=max_norm(3)))
    model.add(Activation(chosenActivation))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())
    model.add(Dense(128, kernel_constraint=max_norm(3)))
    model.add(Activation(chosenActivation))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())
    # softmaxfunción de activación selecciona la neurona con la mayor probabilidad de salida, votando que la imagen pertenece a esa clase
    model.add(Dense(class_num))
    model.add(Activation('softmax'))
    # Optimizador
    optimizer = chosenOptimizer
    # Compilador + metrica
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    print(model.summary())
    return model
# Se corre, entrena, evalúa y guarda el modelo.
def run():
    # Se carga el dataset
    X_train, y_train, X_test, y_test = load_dataset()
    # Se prepara la data
    X_train, X_test = scale_pixels(X_train, X_test)
    # Se define el modelo
    class_num = y_test.shape[1]
    model = define_model(X_train, class_num)
    # Se entrena el modelo con épocas
    epochs = 25
    numpy.random.seed(seed)
    model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=epochs, batch_size=64, verbose=0)
    # Se guarda el modelo
    model.save('model.h5')
 # Punto de entrada
run()
```

Una vez obtenido el modelo, lo tenemos disponible para usar. Con el siguiente archivo podemos levantar el modelo y, con el mismo, evaluar una imagen para ser clasificada en alguna de las 10 categorías antes mencionadas.

```python
## testExample.py
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.models import load_model

# Se carga y prepara la imagen
def load_image(filename):
    # Cargo la imagen
    img = load_img(filename, target_size=(32, 32))
    # Convertir a array
    img = img_to_array(img)
    # Reformar en una muestra de 3 canales
    img = img.reshape(1, 32, 32, 3)
    # Se prepara la info en pixel
    img = img.astype('float32')
    img = img / 255.0
    return img

# Se carga la imagen y se predice su categoría usando el model.h5 previo
def run_example():
    # Cargar la imagen
    img = load_image('sample_image.png')
    # Cargar el modelo
    model = load_model('model.h5')
    # Predecir la clase
    result = model.predict_classes(img)
    print(result[0])

# Entrada para correr el ejemplo
run_example()
```

Utilizando la imagen de la Figura 4 obtenida del dataset, obtenemos el resultado número 4 que es la categoría correspondiente al ciervo.

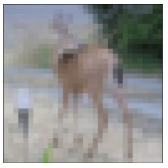


Figura 2. Imagen de un ciervo obtenida del dataset.

Conclusión

A partir del abordaje de varios ejemplos utilizando Keras-TensorFlow, encontramos en estas herramientas una gran simplicidad de aprendizaje y de uso. La ventaja de utilización de Python y de sus múltiples librerías permite que la construcción de modelos sea una tarea muy sencilla y clara y que, además, se cuente con mucha documentación.

Podemos reducir las tareas realizadas en los ejemplos en los siguientes pasos:

- Desarrollo de un entorno de pruebas robusto para la evaluación de un modelo y para establecer una base de performance para tareas de clasificación.
- Extensión del modelo base para mejorar su aprendizaje y capacidad.
- Desarrollo de un modelo completo, evaluación de su performance y utilización para realizar predicciones.

Por otro lado, Keras proporciona algunos modelos de deep-learning, que pueden ser utilizados directamente (como lo hicimos en los ejemplos) para realizar predicciones o extraer características, lo cual es una gran herramienta para adentrarse en el modelado de redes neuronales, ya que se puede directamente utilizar uno de estos datasets, muy bien documentados y con funciones muy útiles que simplifican muchas operaciones.

Anexo

Ejemplo - Identificador de dígitos

En este ejemplo, se implementa un identificador de dígitos utilizando como dataset MNIST; un muy básico set de dígitos escritos a mano con un training set de 60.000 ejemplos y un test set de 10.000 ejemplos, cuyos dígitos fueron normalizados y centrados en imágenes de tamaño fijo.

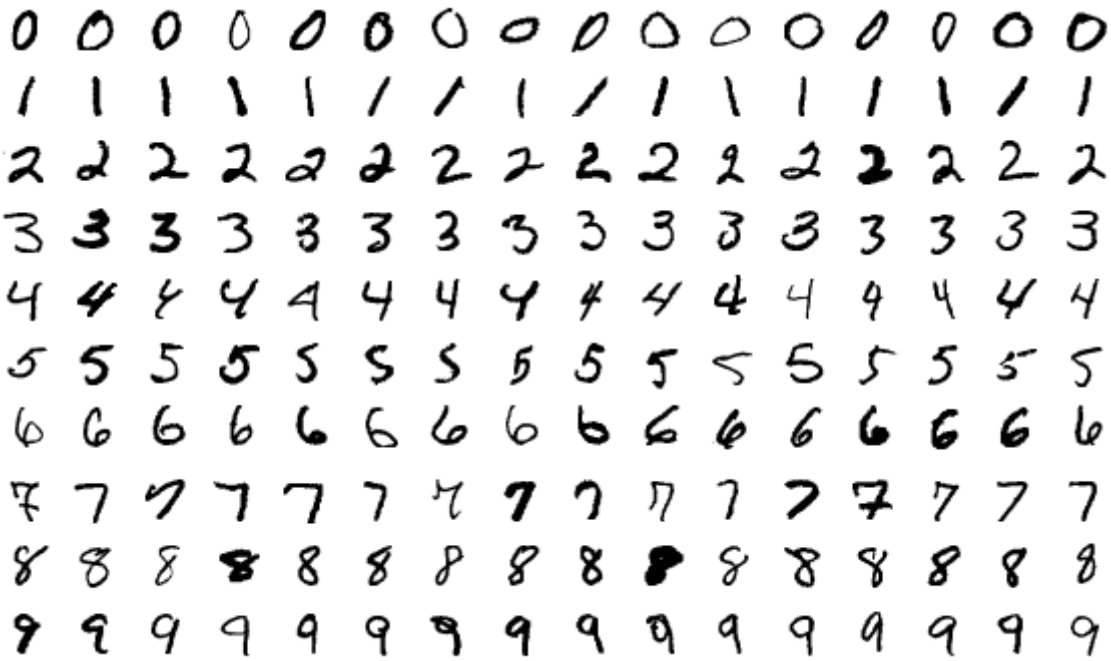


Figura 3. Imágenes de dígitos pertenecientes a la database MNIST.

```
##digits.py
# Importar Keras, TensorFlow y otras librerías útiles
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
```

```
# Importar el dataset
mnist = keras.datasets.mnist

# Carga del dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Construcción del modelo
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10)
])

# Compilación del modelo
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Entrenamiento y evaluación del modelo
results = model.fit(train_images, train_labels, epochs=10)
loss, accuracy = model.evaluate(train_images, train_labels)


### Ejemplo de predicción con modelo entrenado ###

# Tomar una imagen del dataset
img = test_images[0]

# Añadir la imagen a un conjunto en el que es el único elemento
img = (np.expand_dims(img,0))

# Predecir la label correcta para la imagen
predictions_single = probability_model.predict(img)

# Obtener predicción con mayores probabilidades
prediction = np.argmax(predictions_single[0])

print("Número predecido:", prediction)
```