UNIVERSIDAD DE GRANADA

MASTER EN INGENIERÍA INFORMÁTICA

Redes neuronales artificiales (NMIST)

Inteligencia Computacional

Autor

Juan Carlos Serrano Pérez
jcsp0003@correo.ugr.es

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Diciembre de 2018

Redes neuronales artificiales (NMIST)

Juan Carlos Serrano Pérez

Palabras clave: inteligencia artificial, redes neuronales, nmist

Resumen

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales.

Deberá evaluar el uso de varios tipos de redes neuronales para resolver un problema de OCR: el reconocimiento de dígitos manuscritos de la base de datos MNIST (http://yann.lecun.com/exdb/mnist/).

Introducción

Las Redes Neuronales son un campo muy importante dentro de la Inteligencia Artificial, las cuales basándose en el comportamiento conocido del cerebro humano (principalmente el referido a las neuronas y sus conexiones), trata de crear modelos artificiales que solucionen problemas difíciles de resolver mediante técnicas algorítmicas convencionales.

El objetivo del proyecto es construir una red neuronal artificial para el análisis del conjunto de datos mixto del Instituto Nacional de estándares y tecnología (MNIST). Se trata de una colección de 70.000 pequeñas imágenes de dígitos escritos a mano. Los datos fue creados para actuar como un referente para los algoritmos de reconocimiento de imagen.

Las imágenes del conjunto MNIST son de tan solo 28 x 28 píxeles (784 píxeles en total) y sólo hay 10 dígitos posibles (del cero a nueve) a reconocer y hay 60.000 imágenes de entrenamiento para la creación de un modelo de reconocimiento de imagen de un conjunto de prueba (formado por 10.000) para comprobar la exactitud del modelo desarrollado.

El reconocimiento de simplemente dígitos del conjunto MNIST que a priori puede parecer algo trivial, podemos encontrar casos que las redes neuronales serán capaces de identificar ciertas imágenes que una persona puede llegar a tener dudas. En la siguiente imagen se muestra un conjunto de 9 figuras del conjunto.

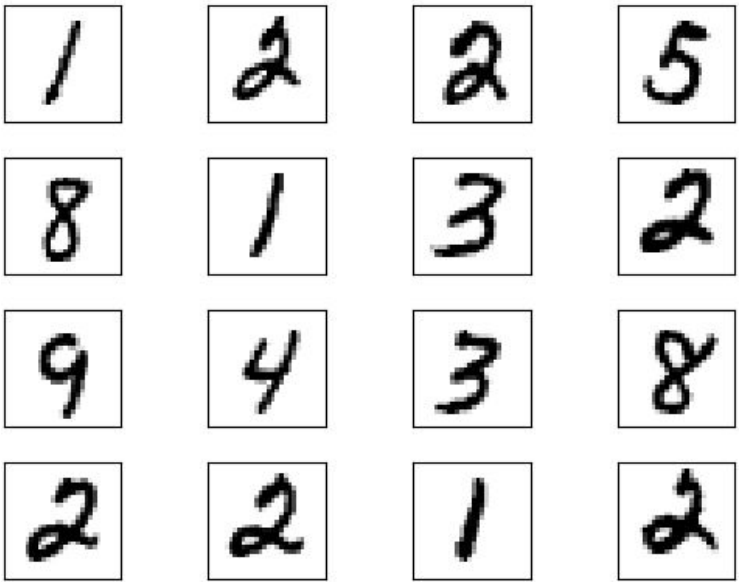


Figura 1: ejemplo de imágenes

Evolución del proyecto

El objetivo de la práctica era de desarrollo de diversas redes neuronales cada vez más complejas tratando minimizar el error cometido.

Así pues, el proyecto comenzó con el desarrollo de una red neuronal perceptrón en Java formada por 10 neuronas cuya tasa de error media era del 15%.

Posteriormente se trató de implementar nuevamente en Java el método propagación hacia atrás o backpropagation formado por una capa de entrada, una capa oculta y una capa de salida pero la tasa de error se encontraba entre el 20% y 30%.

La implementación final del algoritmo se ha realizado finalmente de Python haciendo uso de la librería de código abierto Keras para aprendizaje automático. En esta implementación final se ha logrado reducir el error a un 0.82%.

Implementación

TensorFlow

TensorFlow es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos.

TensorFlow está desarrollado en Python y C++, y está disponible en lenguajes como Python, Java, C++, Go y Rust.

Keras

Se trata de una biblioteca de código abierto escrita en Python que puede ser ejecutada sobre TensorFlow, Microsoft Cognitive Toolkit o Theano. Diseñado para permitir una rápida experimentación con redes neuronales profundas, se enfoca en ser fácil de usar, modular y extensible. Fue desarrollado como parte del esfuerzo de investigación del proyecto ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), y su autor principal y mantenedor es François Chollet, un ingeniero de Google.

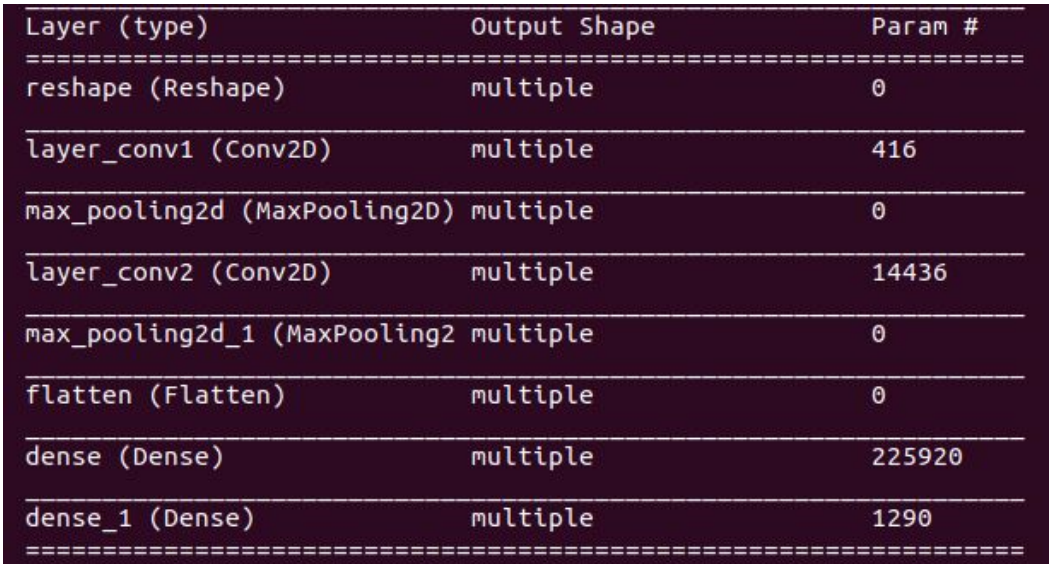
En primer lugar se comenzó a realizar el proyecto haciendo uso únicamente de la biblioteca TensorFlow, aunque posteriormente se modificó para añadir la API Layers que permitía la adición de capas al modelo de forma más intuitiva y sencilla, pero finalmente se optó por Keras que dispone de una mejor documentación y mayor variedad de funciones.

Capas

La librería Keras permite permite construir redes neuronales de forma funcional y secuencial. Para el desarrollo del proyecto se ha optado por la forma secuencial que permite la adición de capas en forma de secuencia.

Las capas que conforman la red neuronal son:

- Capa 1: la primera capa Reshape transforma la matriz aplanada de 784 elementos (28*28) y para las capas convolucionales es necesario que las imágenes tengan forma (28, 28, 1).
- Capa 2: la segunda capa es una capa convolucional llamada Conv2D con ReLU-activation.
- Capa 3: la tercera capa está destinada al uso de la función max-pooling.
- Capa 4 y 5: se tratan de dos nuevas capas para con ReLU-activation y max-pooling como las anteriores.
- Capa 6: esta capa mediante la función Flatten pretende aplanar la salida de 4 niveles de las capas convolucionales anteriores a 2-rank.
- Capa 7: hace uso de la función Dense para formar una capa totalmente conectada con función de activación de rectificación y con 128 neuronas.
- Capa 8: la última capa, vuelve a tratarse de una capa totalmente conectada con 10 neuronas y una función de activación softmax para clasificación.



Layer (type)	Output Shape	Param #
reshape (Reshape)	multiple	0
layer_conv1 (Conv2D)	multiple	416
max_pooling2d (MaxPooling2D)	multiple	0
layer_conv2 (Conv2D)	multiple	14436
max_pooling2d_1 (MaxPooling2	multiple	0
flatten (Flatten)	multiple	0
dense (Dense)	multiple	225920
dense_1 (Dense)	multiple	1290

Figura 2: Sumario de las capas

Calibrado de parámetros

- img_size tiene valor 28 ya que corresponde al ancho y alto de las imágenes.
- img_size_flat tiene valor 784 (28*28) correspondiendose con el número de píxeles de las imágenes.
- num_classes tiene valor 10 para indicar el número de dígitos del 0 al 9.
- epoch se ha establecido con valor 20 ya que tal y como se muestra en el gráfico siguiente de ejemplo, se han realizado diversas ejecuciones de hasta 20 épocas para el entrenamiento y generalmente, antes de la época 15 suele obtener valores de precisión cercanos a 1 sin requerir de un tiempo de entrenamiento excesivo.

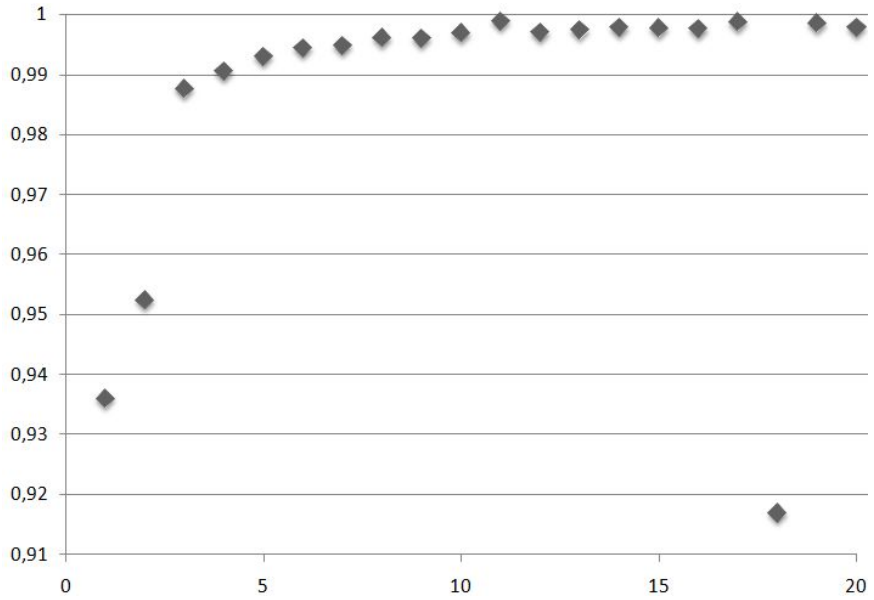


Gráfico 1: precisión de entrenamiento

- batch_size que define el tamaño de los lotes, se ha establecido como 128. A continuación se muestra cómo disminuye el error en cada bloque.

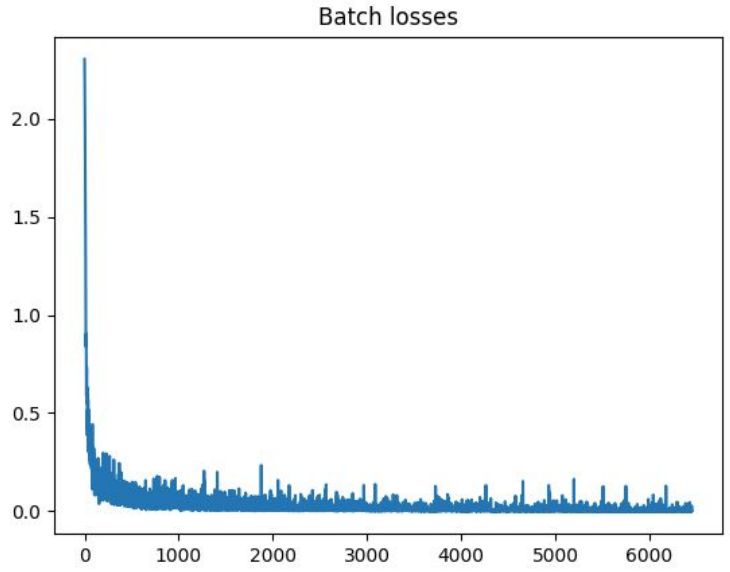


Figura 3: pérdida por bloque

Resultados

Tras calibrar los parámetros anteriormente comentados y los distintos parámetros de las capas para minimizar la posibilidad de sobreaprendizaje se ha tratado de minimizar el error cometido en los conjuntos de entrenamiento y de test.

El tiempo total de entrenamiento de la red neuronal es de 745 segundos.
La precisión sobre el conjunto de entrenamiento es de un 99.9%
El tiempo total de prueba de la red neuronal es de 10 segundos.
La precisión sobre el conjunto de prueba es de un **99.18%**.

Conclusión

La realización de este proyecto ha sido realmente interesante tanto por el haber utilizado un lenguaje como Java o Python que llevaba bastante tiempo sin usar, como por el primer contacto con redes neuronales, de las cuales llevaba escuchando hablar desde el comienzo de la carrera y parecía algo muy abstracto y complejo.

Los resultados obtenidos tanto en conocimientos nuevos, como los generados por el programa son satisfactorios. Aunque inicialmente se comenzó realizando el algoritmo sin el uso de librerías y finalmente se tuvo que recurrir a ellas porque los resultados generados no eran los indicados, demuestra que el desarrollo de las redes neuronales no es algo para nada trivial por mucho que usando éstas el proyecto quede en unas 100 míseras líneas de código.

Bibliografía

[1] "Red neuronal artificial", *Es.wikipedia.org*, 2018. [Online]. Available: https://es.wikipedia.org/wiki/Red_neuronal_artificial. [Accessed: 16- Nov- 2018].

[2]"TensorFlow", TensorFlow, 2018. [Online]. Available: https://www.tensorflow.org/?hl=es. [Accessed: 16- Nov- 2018].

[3]"petewarden/tensorflow_makefile", GitHub, 2018. [Online]. Available: https://github.com/petewarden/tensorflow_makefile. [Accessed: 19- Nov- 2018].

[4]"Vikramank/Deep-Learning-", GitHub, 2018. [Online]. Available: https://github.com/Vikramank/Deep-Learning-. [Accessed: 19- Nov- 2018].

[5]"TENSOR FLOW PARA PRINCIPIANTES (IV): Uso de la API Layers", Apsl.net, 2018. [Online]. Available: https://www.apsl.net/blog/2018/01/19/tensor-flow-para-principiantes-iv-uso-de-la-api-layers/. [Accessed: 23- Nov- 2018].

[6]"TENSOR FLOW PARA PRINCIPIANTES (VI): Uso de la API Keras", Apsl.net, 2018. [Online]. Available: https://www.apsl.net/blog/2018/02/02/tensor-flow-para-principiantes-vi-uso-de-la-api-keras/. [Accessed: 23- Nov- 2018].

Anexo 1: Entorno de ejecución

Para poder entender y comparar correctamente el tiempo de ejecución del algoritmo con los de otros, es necesario conocer la máquina y el entorno en el que se ha ejecutado.

El hardware del equipo utilizado ha sido:

-Procesador: Intel® Core™ i5-7300HQ
-Memoria RAM: 8 GB DDR4-2400

Características de la máquina virtual en la que se ha ejecutado:

-Software de virtualización: VirtualBox 5.2.22
-Sistema Operativo: Ubuntu 18.04.1 LTS
-Memoria RAM: 4 GB
-Python 3.7.1

Anexo 2: Documentos generados

Los documentos adjuntados en la práctica son:

1. mnist.py: fichero ejecutable Python del algoritmo.
2. requirement.txt: documento de texto con las librerías y versiones utilizadas para el desarrollo del proyecto.
3. salida.txt: ejemplo de una salida impresa por la red neuronal.
4. img1.png: imagen generada durante la ejecución del algoritmo con la pérdida por lote.

Anexo 3: Código

```
import matplotlib.pyplot as plt
import numpy as np
import math
import keras

# Importar la API keras
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import InputLayer, Input
from tensorflow.python.keras.layers import Reshape, MaxPooling2D
from tensorflow.python.keras.layers import Conv2D, Dense, Flatten

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.python.keras.optimizers import Adam
import keras.callbacks as cb
```

```
class LossHistory(cb.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        batch_loss = logs.get('loss')
      self.losses.append(batch_loss)

#Cargo los datos
data = input_data.read_data_sets('data/MNIST/', one_hot=True)
data.test.cls = np.argmax(data.test.labels, axis=1)

#Se definen las variables
img_size = 28
img_size_flat = img_size * img_size
# Tupla con la altura y el ancho de las imagenes utilizadas para
remodelar matrices.
# Esto se utiliza para pintar las imagenes.
img_shape = (img_size, img_size)
# Tupla con altura, anchura y profundidad utilizada para remodelar
matrices.
# Esto se usa para remodelar en Keras.
img_shape_full = (img_size, img_size, 1)
# Numero de canales de color para las imagenes: 1 canal para escala
de grises.
num_channels = 1
# Numero de clases, una clase para cada uno de 10 digitos.
num_classes = 10


#Construccion de la red neuronal de forma secuencial
model = Sequential()
# La entrada es una matriz aplanada con 784 elementos (img_size *
img_size),
# pero las capas convolucionales esperan imagenes con forma (28, 28,
1), por tanto hacemos un reshape
model.add(Reshape(img_shape_full))
# Primera capa convolucional con ReLU-activation y max-pooling.
model.add(Conv2D(kernel_size=5, strides=1, filters=16,
padding='same', activation='relu', name='layer_conv1'))
model.add(MaxPooling2D(pool_size=2, strides=2))
# Segunda capa convolucional con ReLU-activation y max-pooling.
model.add(Conv2D(kernel_size=5, strides=1, filters=36,
padding='same', activation='relu', name='layer_conv2'))
model.add(MaxPooling2D(pool_size=2, strides=2))
# Aplanar la salida de 4 niveles de las capas convolucionales
# a 2-rank que se puede ingresar a una capa totalmente conectada
```

```
model.add(Flatten())
# Primera capa completamente conectada  con ReLU-activation.
model.add(Dense(128, activation='relu'))
# Ultima capa totalmente conectada con activacion de softmax para
usar en la clasificacion.
model.add(Dense(num_classes, activation='softmax'))

#Anadir funcion de coste, un optimizador y las metricas de
rendimiento
optimizer = Adam(lr=1e-3)
model.compile(optimizer=optimizer, loss='categorical_crossentropy',
metrics=['accuracy'])

#Fase de entrenamiento
print("Comienza el entrenamiento")
history = LossHistory()
model.fit(x=data.train.images, y=data.train.labels,
callbacks=[history], epochs=15, batch_size=128)
result = model.evaluate(x=data.train.images, y=data.train.labels)
for name, value in zip(model.metrics_names, result):
    print(name, value)

#Pruebo con el conjunto de test
print("Pruebo el conjunto de test")
result = model.evaluate(x=data.test.images, y=data.test.labels)
for name, value in zip(model.metrics_names, result):
    print(name, value)

model.summary()

plt.switch_backend('agg')
plt.ioff()
fig = plt.figure()

ax = fig.add_subplot(111)
ax.plot(history.losses)
ax.set_title('Batch losses')

plt.show()
fig.savefig('img1.png')
```

Anexo 4: Salida

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz

Comienza el entrenamiento
Epoch 1/15
55000/55000 [==============================] - 48s 875us/step - loss: 0.2207 -
acc: 0.9340
Epoch 2/15
55000/55000 [==============================] - 47s 863us/step - loss: 0.0579 -
acc: 0.9821
Epoch 3/15
55000/55000 [==============================] - 47s 846us/step - loss: 0.0401 -
acc: 0.9872
Epoch 4/15
55000/55000 [==============================] - 47s 852us/step - loss: 0.0303 -
acc: 0.9906
Epoch 5/15
55000/55000 [==============================] - 47s 848us/step - loss: 0.0242 -
acc: 0.9923
Epoch 6/15
55000/55000 [==============================] - 48s 869us/step - loss: 0.0193 -
acc: 0.9941
Epoch 7/15
55000/55000 [==============================] - 48s 869us/step - loss: 0.0158 -
acc: 0.9948
Epoch 8/15
55000/55000 [==============================] - 51s 934us/step - loss: 0.0128 -
acc: 0.9959
Epoch 9/15
55000/55000 [==============================] - 47s 857us/step - loss: 0.0103 -
acc: 0.9966
Epoch 10/15
55000/55000 [==============================] - 57s 1ms/step - loss: 0.0099 - acc:
0.9968
Epoch 11/15
55000/55000 [==============================] - 60s 1ms/step - loss: 0.0070 - acc:
0.9979
Epoch 12/15
55000/55000 [==============================] - 48s 868us/step - loss: 0.0074 -
acc: 0.9974
Epoch 13/15
55000/55000 [==============================] - 48s 875us/step - loss: 0.0060 -
acc: 0.9981
Epoch 14/15
55000/55000 [==============================] - 52s 950us/step - loss: 0.0043 -
```

acc: 0.9988
Epoch 15/15
55000/55000 [==============================] - 50s 909us/step - loss: 0.0058 - acc: 0.9981
55000/55000 [==============================] - 20s 357us/step
('loss', 0.003285756561911727)
('acc', 0.9990363636363636)

Pruebo el conjunto de test
10000/10000 [==============================] - 4s 373us/step
('loss', 0.032011635006228424)
('acc', 0.9918)

____
Layer (type)   Output Shape   Param #
===
reshape (Reshape)   multiple   0
____
layer_conv1 (Conv2D)   multiple   416
____
max_pooling2d (MaxPooling2D) multiple   0
____
layer_conv2 (Conv2D)   multiple   14436
____
max_pooling2d_1 (MaxPooling2 multiple   0
____
flatten (Flatten)   multiple   0
____
dense (Dense)   multiple   225920
____
dense_1 (Dense)   multiple   1290
===
Total params: 242,062
Trainable params: 242,062
Non-trainable params: 0