Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Reconocimiento de CAPTCHAs con redes neuronales

CAPTCHA recognition using neural networks

Alejandro León Fernández

La Laguna, 10 de junio de 2019

Dª. **Rosa María Aguilar Chinea**, con N.I.F. 43.778.956-C Catedrática de Universidad adscrita al Departamento de Ingeniería de Sistemas y Automática de la Universidad de La Laguna, como tutora.

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A (N)

Que la presente memoria titulada:

”*Reconocimiento de CAPTCHAs con redes neuronales*”

ha sido realizada bajo su dirección por D. **Alejandro León Fernández**, con N.I.F. 43.484.348-L.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de junio de 2019

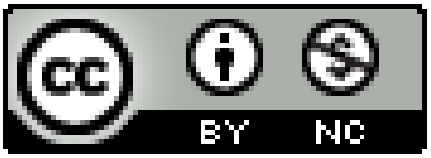
Agradecimientos

Deseo agradecer a mi tutora, Rosa María Aguilar Chinea, por la ayuda y los consejos recibidos, y por su predisposición a resolver cualquier duda que me surgiese.

Quiero agradecer también a los profesores que he tenido en mi paso por la universidad, que me han ayudado a comprender los contenidos cursados y en ocasiones a despertar un gran interés en algunas asignaturas.

Finalmente, me gustaría agradecer a mi familia y a mis compañeros de la facultad por haber estado ahí todo este tiempo y haber hecho el camino mucho más agradable.

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial 4.0 Internacional.

Resumen

El objetivo de este proyecto es desarrollar un programa capaz de reconocer ciertos tipos de CAPTCHAs mediante el uso de redes neuronales, además de estudiar su funcionamiento, comprender los factores que favorecen un mejor rendimiento y aquellos que lo dificultan.

Se ha implementado un programa en el que se entrena una red neuronal para reconocer letras individuales. Para ello se usan datos de entrenamiento generados dinámicamente, CAPCTCHAs creados por el propio programa. Junto con otros métodos para separar las letras del CAPTCHA y para corregir errores tras la clasificación de la red neuronal, se investiga la efectividad de este procedimiento, sus inconvenientes y sus posibles mejoras.

Palabras clave: CAPTCHAs, redes neuronales, clasificación, Python

Abstract

The objective of this project is to develop a program that is able to recognize certain types of CAPTCHAs using neural networks, besides studying how it works, understanding the factors that improve its performance and those which make it worse.

The program that has been implemented trains a neural network to be able to recognize individual letters. For this purpose, training data is generated dinamically, CAPTCHAs to train the network are created by the program itself. Alongside other methods to split CAPTCHA's letters and to amend mistakes made after the neural net classification, the effectiveness of this process, its drawbacks and possible improvements are investigated.

Keywords: CAPTCHAs, neural networks, classification, Python

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Antecedentes

1.1. CAPTCHAs

Los CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart), como su nombre indica, están diseñados con el objetivo de diferenciar a los humanos de los ordenadores. Son un sistema comúnmente utilizado por las páginas web en el proceso de registro para comprobar que es un humano quien lo está realizando. Con esto se pretende evitar que programas automatizados creen gran cantidad de cuentas falsas con las que llenar la sección de comentarios con spam, o que puedan adulterar encuestas.

Para lograrlo, los CAPTCHAs habitualmente contienen imágenes con caracteres u otro tipo de información que se espera que solo un humano pueda identificar. Para este proyecto, nos centraremos en aquellos en aquellos basados en caracteres. Más concretamente, limitaremos los CAPTCHAs a aquellos que contengan palabras inglesas de 4 letras.

1.2. Redes neuronales

Las redes neuronales artificiales son un modelo computacional inspirado en la biología, concretamente en los cerebros de los animales. Dentro de ellos, las neuronas transmiten impulsos a otras neuronas a través del axón si reciben suficiente estimulación desde las dendritas. Si bien el caso biológico es más complejo y tiene más características, de aquí se extrae la versión simplificada que se utiliza para crear las redes neuronales artificiales.

Las redes neuronales se modelan conectando neuronas artificiales entre sí, habitualmente organizadas en capas. De esta forma, las neuronas de una capa solo envían señales a las de la siguiente. La capa de entrada recibe la información y la red produce unos valores de salida. Este resultado depende de la entrada y de los pesos de los enlaces entre las neuronas.

Así, para que una red neuronal realice una tarea, es necesario entrenarla con conjuntos de datos. Estos deben tener la entrada y la salida esperada, de forma que la red pueda ir modificando los pesos de sus enlaces (inicialmente aleatorios) en función de los errores que vaya teniendo.

Capítulo 2

Objetivos

2.1. Objetivo general del proyecto

Se busca entender el funcionamiento de las redes neuronales en el reconocimiento de texto, más específicamente de CAPTCHAs formados por caracteres en este caso, e implementar un programa en el lenguaje de programación Python que entrene una red neuronal para tal fin. Además de lo anterior, se pretende estudiar y comprender los factores que complican o ayudan a mejorar la efectividad de la clasificación de CAPCTCHAs, llevar diferentes mejoras a la práctica y analizar sus resultados.

2.2. Objetivos específicos

A través del desarrollo del presente proyecto, se esperan lograr los objetivos expuesto a continuación:

- Estudiar las redes neuronales artificiales
- Crear un conjunto de datos de entrenamiento formado por CAPTCHAs
- Entrenar las red neuronal para clasificar letras individuales
- Mejorar la precisión del clasificador usando un diccionario
- Entender los factores más relevantes para mejorar el proceso de clasificación de CAPTCHAs

Capítulo 3

Estado del arte

3.1. Reconocimiento de CAPTCHAs

Los sistemas para solucionar CAPTCHAs de forma automática han ido mejorando con el paso del tiempo, pudiendo resolver en función de la complejidad de los CAPTCHAs desde caracteres hasta imágenes complejas. En este contexto, ha surgido un nuevo sistema llamado reCaptcha que evalúa las acciones del usuario para saber si se trata de un bot o una persona, y se sale del concepto clásico de CAPTCHA al que estamos acostumbrados.

En el caso concreto de los CAPTCHAs basados en caracteres, habitualmente se realizan los procesos de segmentación y clasificación de las letras en único paso.

3.2. Redes neuronales

Hoy en día las redes neuronales están en auge y son uno de los campos que más se investigan en todo el mundo. Gracias a su gran flexibilidad, tienen aplicaciones en un amplio rango de disciplinas, que podemos agrupar a grandes rasgos dentro de las siguientes categorías: clasificación, procesamiento de datos, robótica, ingeniería de control y aproximación de funciones.

3.3. Lista de herramientas

3.3.1. Entornos de trabajo

Visual Studio Code

Desarrollado por Microsoft, se trata de un editor de código fuente que se encuentra disponible para Windows, macOs y Linux. Entre sus funciones, incluye control integrado con Git, autocompletado de código, resaltado de sintaxis y refactorización de código. Cuenta además con soporte para la depuración y la posibilidad de añadirle una gran variedad de extensiones.

Para este proyecto, he usado una extensión de Python que incluye resaltado de código, formateo automático, refactorización, depuración, test unitarios y más funciones de utilidad.

IPython

Es una shell interactiva que incluye funcionalidades adicionales respecto al modo interactivo incluido con Python. Entre ellas podemos encontrar resaltado de líneas y errores, una sintaxis extendida para la shell, autocompletado de variables y ficheros con el tabulador, etc.

En este proyecto, lo he utilizado para poder mostrar los gráficos creados por la librería *matplotlib*, corriendo IPython en la terminal y a continuación ejecutando el fichero deseado con: *run <nombre_fichero.py>*

3.3.2. Librerías utilizadas

Las librerías usadas más importantes son las siguientes:

- **NumPy:** Permite trabajar con arrays de manera mucho más cómoda, y sus arrays son el tipo de dato usado por otras de las librerías.
- **Pillow:** Provee funciones para crear imágenes y escribir en ellas, imprescindible para crear los CAPTCHAs.
- **scikit-image:** Cuanta con diferentes funciones de manipulación de imágenes. Es la librería usada para deformar las imágenes creadas con *Pillow* para obtener CAPTCHAs y para el proceso de segmentación de los mismos.
- **scikit-learn:** Librería para el aprendizaje automático en Python que cuanta con funciones necesarias para la creación de los conjuntos de datos para el entrenamiento de la red neuronal.
- **PyBrain:** Usada para la creación y el entrenamiento de la red neuronal.

Capítulo 4

Fases y desarrollo del proyecto

El cógido fuente para realizar las tareas que se exponen en este capítulo, así como para algunas de las pruebas y los resultados, en encuentra subido en GitHub en https://github.com/AlejandroLF/TFGcaptchas.

4.1. Funcionamiento del programa

Descrito de forma simplificada, el proceso que se seguirá para resolver CAPTCHAs consiste en:

1. Segmentar el CAPTCHA en imágenes que contengan letras individuales
2. Clasificar cada uno de las letras por separado usando la red neuronal
3. Volver a juntar las letras para formar una palabra
4. Comparar las palabras con un diccionario

4.1.1. Creación de CAPTCHAs para el conjunto de entrenamiento

Para poder entrenar la red neuronal, es necesario crear imágenes de entrenamiento. Para ello, usamos *Image*, *ImageDraw* e *ImageFont* la librería *Pillow* para dibujar letras en una imagen, y *transform* de la librería *scikit-image* para deformar dichas letras. Asimismo, utilizamos los arrays de la librería *NumPy*, ya que *scikit-image* trabaja con este tipo de dato. Es necesario haber descargado una fuente para poder usarla como tipo de letra.

Se crea una función que toma una palabra y cuánta deformación realizar como parámetros, además del tamaño de la imagen resultante de forma opcional. Es importante añadir que solo permitiremos píxeles en blanco y negro en la imagen creada. Al acabar normalizamos dividiendo entre el valor máximo de la imagen, para asegurarnos de que solo hay valores entre 0 y 1.

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage import transform as tf

def create_captcha(text, shear = 0, size = (100, 24)):
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
```

```
    font = ImageFont.truetype(r"bretan\Coval-Regular.ttf", 22)
    draw.text((3, -2), text, fill = 1, font = font)

    image = np.array(im)
    affine_tf = tf.AffineTransform(shear = shear)
    image = tf.warp(image, affine_tf)

    return image / image.max()
```

Para poder mostrar un ejemplo, usamos *pyplot* de la librería *matplotlib* desde IPython.

```
from IPython import get_ipython
get_ipython().run_line_magic('matplotlib', 'tk')
from matplotlib import pyplot as plt

image = create_captcha("VIDA", shear = 0.2)
plt.imshow(image, cmap = 'Greys')
```

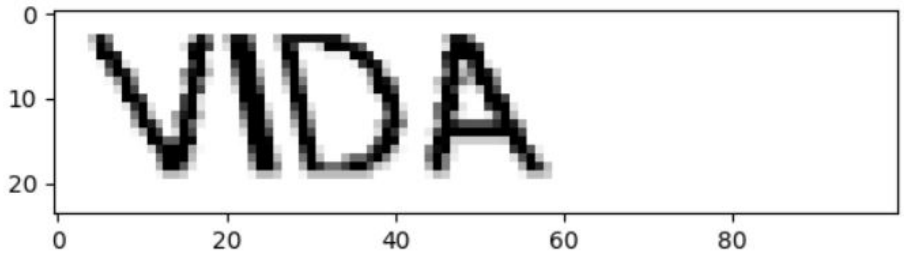


Figura 4.1: Ejemplo de CAPTCHA

4.1.2. Segmentación de los CAPTCHAs

Dado que la red neuronal va a clasificar imágenes de letras individuales, es necesario obtener imágenes más pequeñas a partir de los CAPTCHAs, de forma que cada una de ellas contenga una letra. Para ello, averiguamos las diferentes componentes conexas del CAPTCHA, que asumimos que se corresponden con las letras. Para cada una de ellas, extraemos la región rectangular que la envuelve de la imagen original. De esta forma obtenemos las imágenes de cada una de las letras.

```
from skimage.measure import label, regionprops

def segment_image(image):
    labeled_image = label(image > 0)
    subimages = []
    for region in regionprops(labeled_image):
```

```
        start_x, start_y, end_x, end_y = region.bbox
        subimages.append(image[start_x:end_x, start_y:end_y])

    if len(subimages) == 0:
        return [image,]
    return subimages
```

Podemos mostrar un ejemplo de la siguiente forma:

```
from IPython import get_ipython
get_ipython().run_line_magic('matplotlib', 'tk')
from matplotlib import pyplot as plt

image = create_captcha("VIDA", shear = 0.2)
subimages = segment_image(image)
f, axes = plt.subplots(1, len(subimages), figsize = (10, 3))
for i in range(len(subimages)):
    axes[i].imshow(subimages[i], cmap = 'gray')
```

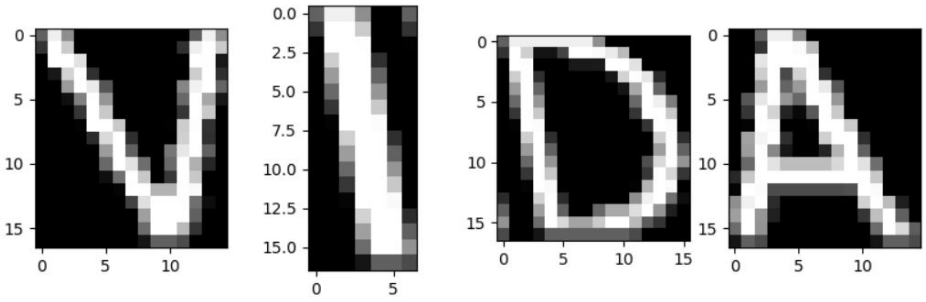


Figura 4.2: Ejemplo de CAPTCHA segmentado

Esta forma de separar las letras puede usarse en este caso dado que las transformaciones realizadas a las palabras no son muy complejas. No obstante, como veremos más adelante, aun en este caso este es uno de los principales problemas para obtener mejores resultados.

4.1.3. Creación del conjunto de entrenamiento

Inicialmente, creamos una lista con las letras del abecedario inglés para asociar un índice a cada letra y poder codificarlas como categorías para identificar la salida de la red neuronal. También creamos un array de la librería *Numpy* con los posibles valores para la deformación de las letras. Asimismo, usamos *check_random_state* de *scikit-learn* para las elecciones aleatorias.

```
from sklearn.utils import check_random_state
random_state = check_random_state(30)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

```
shear_values = np.arange(0, 0.5, 0.05)
```

Versión inicial

En la versión inicial, construimos el conjunto de entrenamiento a partir de creación de CAPTCHAs que contengan una sola letra. Usamos una función que genera una letra aleatoria y devuelve el CAPTCHA asociado junto con el índice de la letra que representa.

```
def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)

    return create_captcha(letter, shear=shear, size=(20,20)),
        letters.index(letter)

dataset, targets = zip(*(generate_sample(random_state) for _ in range(10000)))
```

Letras previamente segmentadas

En una mejora posterior, se construye el conjunto de entrenamiento de forma que sea lo más parecido posible a lo que se va a encontrar la red neuronal cuando vaya a clasificar letras extraídas de palabras. Para ello, se crean en primer lugar CAPTCHAs con 4 letras aleatorias. A continuación, se extraen las imágenes de las letras individuales mediante el método usado para segmentar CAPTCHAs (visto en la sección anterior). Finalmente, se devuelve una lista de pares de las imágenes asociadas con el índice de la letra que representa cada una de ellas.

```
def generate_samples(random_state=None):
    random_state = check_random_state(random_state)
    word = ""
    for _ in range(4):
        word += random_state.choice(letters)
    shear = random_state.choice(shear_values)
    captcha = create_captcha(word, shear=shear, size=(100,24))

    letter_images = segment_image(captcha)
    for image in letter_images:
        image = resize(image, (20, 20))
    indexes = list(letters.index(letter) for letter in word)
    return list(zip(letter_images, indexes))

dataset, targets = zip(*(sample for _ in range(3000) for sample in
    generate_samples(random_state)))
```

Esto es relevante dado que en ocasiones, a partir de cierto nivel de deformación, pueden aparecer partes de otras letras en las imágenes de las letras individuales tras segmentar el CAPTCHA. Así pues, entrenando la red neuronal con imágenes creadas a partir de CAPTCHAs de una sola letra no la estamos preparando adecuadamente para lo que se va a encontrar al clasificar CAPTCHAs completos.

Seguidamente, guardamos el conjunto de imágenes de letras de entrenamiento y los índices de las letras que representan por separado. Para las imágenes, las ajustamos todas a un tamaño de 20x20 píxeles y las 'aplanamos' en una sola dimensión. Así pues, la red neuronal tendrá 400 neuronas en la capa de entrada, una por cada píxel.

Para los índices, transformamos las 26 posibilidades de índices para las letras en una lista de 0s en la que la posición correspondiente al índice es un 1, es decir, hacemos una codificación one-hot de las categorías. Esta transformación se realiza fácilmente usando la función *OneHotEncoder* de la librería *scikit-learn*. La red neuronal tendrá entonces 26 neuronas en la capa de salida, que representarán cada una de las letras posibles.

Finalmente, separamos un 90 % de los datos (imágenes y categorías) para entrenamiento y dejamos un 10 % para tests, mediante *train_test_split*, también de la librería *scikit-learn*. Para cada uno de los conjuntos de datos (entrenamiento y tests), creamos un *SupervisedDataSet* (de la librería *PyBrain*) en el que añadimos las parejas imagen-categoría una a una.

```
dataset = np.array(dataset)
targets = np.array(targets)

from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder(categories='auto')
y = onehot.fit_transform(targets.reshape(targets.shape[0], 1))
y = y.todense()

dataset = np.array([resize(sample, (20, 20)) for sample in dataset])
X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.shape[2]))

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.9)

from pybrain.datasets import SupervisedDataSet
training = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_train.shape[0]):
    training.addSample(X_train[i], y_train[i])
testing = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_test.shape[0]):
    testing.addSample(X_test[i], y_test[i])
```

4.1.4. Creación y entrenamiento de la red neuronal

Para crear la red neuronal, usamos la función *buildNetwork* de la librería *PyBrain*, que recibe un tupla con el tamaño de las capas de la red. Como ya hemos visto en la sección

anterior, la capa de entrada tendrá 400 entradas correspondientes a los píxeles de la imagen, y la de salida 26 correspondientes a las categorías de las letras del abecedario inglés.

Para las capas intermedias se realizan diferentes pruebas. Con dos capas, aun aumentando el número de datos de entrenamiento y las repeticiones (epochs) en el entrenamiento de la red neuronal significativamente, la elección de la red neuronal no mejora mucho una elección aleatoria. Para una sola capa intermedia, los mejores resultados se dan alrededor de 100 neuronas, que resulta ser un punto intermedio entre las capas de entrada y salida. Así pues, se usan 100 neuronas para la capa intermedia.

Para poder entrenar la red neuronal, creamos un *BackPropTrainer* de la librería *PyBrain*, es decir, mediante propagación hacia atrás de los errores. Usamos el método *trainEpochs* para entrenar la red neuronal, indicándole que realice 20 repeticiones (epochs). En cualquiera de las diferentes versiones usadas, no se observan mejoras a partir de esta cifra.

```
from pybrain.tools.shortcuts import buildNetwork
net = buildNetwork(X.shape[1], 100, y.shape[1], bias=True)

from pybrain.supervised.trainers import BackpropTrainer
trainer = BackpropTrainer(net, training, learningrate=0.01, weightdecay=0.01)
trainer.trainEpochs(epochs=20)
```

4.1.5. Clasificación de palabras completas

Para clasificar palabras completas, creamos una función que recibe un CAPTCHA y la red neuronal como parámetros y devuelve la palabra que cree que contiene el CAPTCHA.

Versión básica

En la versión inicial, segmentamos el CAPTCHA usando la función *segment_image* creada previamente y, para cada una de las imágenes resultado, activamos la red neuronal para obtener la letra que hay en la imagen. Para terminar, juntamos todas las letras obtenidas que forman la palabra que creemos que contiene el CAPTCHA.

```
def predict_captcha(captcha_image, neural_network):
subimages = segment_image(captcha_image)
predicted_word = ""
for subimage in subimages:
   subimage = resize(subimage, (20, 20))
   outputs = net.activate(subimage.flatten())
   prediction = np.argmax(outputs)
   predicted_word += letters[prediction]
return predicted_word
```

Uso de un diccionario

Dado que sabemos CAPTCHAs que queremos clasificar contienen palabras inglesas de 4 letras, podemos usar esa información para mejorar el resultado. Así pues, la función tendrá un nuevo parámetro en el que reciba la lista de palabras válidas. Para ello, obtenemos el corpus *words* de la librería *Natural Language Toolkit*. Para poder usarlo, hay que descargarlo previamente (basta con hacerlo una sola vez) de la siguiente forma:

```
import nltk
nltk.download('words')
```

En este caso, nos interesan las palabras de 4 letras, que podemos obtener de forma sencilla de la siguiente manera:

```
from nltk.corpus import words
valid_words = [word.upper() for word in words.words() if len(word) == 4]
```

En caso de que la palabra resultado de la versión básica no esté entre las palabras válidas, escogemos la que más se le parezca. Para ello, contamos cada error de clasificación en una letra como 1 unidad de distancia y tomamos la palabra válida más cercana.

```
def compute_distance(prediction, word):
    return len(prediction) - sum(prediction[i] == word[i] for i in
        range(len(prediction)))

from operator import itemgetter
def improved_prediction(captcha, net, dictionary):
    prediction = predict_captcha(captcha, net)
    if prediction not in dictionary:
        distances = sorted([(word, compute_distance(prediction, word)) for word
            in dictionary], key=itemgetter(1))
        best_word = distances[0]
        prediction = best_word[0]
    return prediction
```

Uso de la matriz de confusión

Es posible obtener la matriz de confusión usando *consusion_matrix* de *scikit-learn* y el conjunto de datos de testeo. Con ella, podemos mejorar la medida de distancia entre dos letras para que sea proporcional a la probabilidad de que una se confunda con la otra. El código quedaría ahora de la siguiente manera:

```
predictions = trainer.testOnClassData(dataset=testing)
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(np.argmax(y_test, axis=1), predictions)
```

```
number_of_letter_predictions = []
for i in range(len(cm)):
    predictions = 0
    for j in range(len(cm)):
        predictions += cm[j][i]
    number_of_letter_predictions.append(predictions)

def confusion_probability(original_letter, new_letter):
    original_letter_index = letters.index(original_letter)
    new_letter_index = letters.index(new_letter)
    original_letter_predictions =
        number_of_letter_predictions[original_letter_index]
    # No info at all in this case
    if original_letter_predictions == 0:
        return 0.5
    return cm[new_letter_index][original_letter_index] /
        original_letter_predictions

def compute_distance(prediction, word):
    return len(prediction) - sum(confusion_probability(prediction[i], word[i])
        for i in range(len(prediction)))
```

4.2. Problemas encontrados

Una de las librerías, *PyBrain*, aparentemente se instalaba bien usando Pip. No obstante, al ejecutar el código faltaban dependencias de la librería. Tras instalarlas manualmente, al ejecutar el código se descubrían nuevas dependencias no detectadas previamente.

Finalmente, para solucionar el problema, fue necesario descargar la librería desde GitHub y añadir su ubicación en el sistema al PATH de Python.

4.3. Partes a mejorar

Tal como veremos con más detalle en los resultados, el principal punto a mejorar, que es la fuente de casi todos los errores de clasificación, es la fase de segmentación. A partir de cierto nivel de deformación de las palabras, algunas de las letras llegan a "hacer contacto", es decir, tienen píxeles comunes o que se tocan directamente, que provocan que al realizar las componentes conexas dos o más letras queden juntas.

Si bien un pequeño porcentaje de estos CAPTCHAs se acierta gracias a la comparación posterior con un diccionario, contar con menos de 4 imágenes de letras individuales hace fallar la clasificación en la gran mayoría de los casos. Se analizan posibles soluciones al problema en las líneas futuras.

Capítulo 5

Pruebas y resultados

Dado que los CAPTCHAs a resolver son palabras inglesas de 4 letras, tomamos el corpus ya mencionado de la librería Natural Language Toolkit con las palabras de 4 letras filtradas. Para cada una de las palabras, creamos un CAPTCHA con un nivel de deformación aleatorio, se lo pasamos a la función encargada de resolverlo (*predict_captcha* o *improved_prediction*) y comprobamos el resultado.

Además, como hay muchos valores aleatorios que pueden condicionar los resultados, los siguientes datos son la media de diferentes ejecuciones.

5.1. Precisión en la clasificación

Versión inicial del conjunto de entrenamiento:

- Versión inicial sin diccionario: 37,72 %
- Usando el diccionario: 38,44 %
- Diccionario + matriz de confusión: 40,71 %

Conjunto de entrenamiento a partir de CAPTCHAs completos:

- Versión inicial sin diccionario: 61,70 %
- Usando el diccionario: 62,59 %
- Diccionario + matriz de confusión: 63,58 %

5.2. Errores en la segmentación

En la versión final (conjunto de entrenamiento a partir de CAPTCHAs complejos, diccionario y matriz de confusión):

- Porcentaje de acierto: 63,58 %
- Porcentaje de segmentación incorrecta: 37.01 %
- Porcentaje de acierto con segmentación correcta: 98,93 %

Capítulo 6

Presupuesto

Dado que todas las herramientas software utilizadas con gratuitas, y el resto de materiales necesarios (ordenador personal, internet del hogar) ya se encontraban disponibles independientemente de la realización del TFG, los gastos a tener en cuenta son:

Descripción	**Cantidad**	**Precio**
Horas de trabajo	300	2700€
Gasolina (traslados)	-	35€
Total	-	2735€

Tabla 6.1: Presupuesto

Capítulo 7

Conclusiones y líneas futuras

7.1. Conclusiones

En conclusión, este proyecto me ha resultado muy interesante ya que gracias a él he explorado el mundo de la inteligencia artificial con más profundidad de lo que lo había hecho antes. Concretamente, he podido aprender bastante sobre redes neuronales, las dificultades que se pueden presentar al trabajar con ellas y su enorme potencial.

Uno de las principales ideas extraídas de este proyecto es la importancia de la calidad de los datos a la hora de obtener los mejores resultados posibles, ya que estos son claves en el entrenamiento de una red neuronal. Siempre que se pueda, estos deben ser lo más parecidos posibles a lo que nos vamos a encontrar en el uso real de la red. Asimismo, también es muy importante la cantidad de datos disponibles según el modelo se vuelve más complejo, de forma que se hacen necesarios más tiempo y capacidad de cómputo para obtener resultados relevantes.

Por otro lado, aunque las redes neuronales son capaces de realizar tareas de otra forma muy complejas o imposibles, en muchos casos un post-procesamiento adecuado puede corregir errores cometidos por la red neuronal y así mejorar los resultados que se obtienen.

7.2. Líneas futuras

7.2.1. Realizar correcciones simples tras la segmentación

En los casos en los que la segmentación dé únicamente 3 imágenes como resultado, se plantea identificar la imagen que contiene más de una letra escogiendo la de mayor tamaño. Dado que el proceso que usamos para crear los CAPTCHAs no incluye cambios en el tamaño de las letras, por lo general la elección debería ser la correcta. A continuación, se separa dicha imagen en 2 sub-imágenes, bien por la mitad o con otro método algo más complejo. Con este cambio, se podrían obtener mejoras interesantes por usa serie de razones:

- Tal como se crea el conjunto de entrenamiento actualmente, si las letras no segmentadas adecuadamente no son las últimas, a todas las que vengan después se les asigna una categoría que no es la correcta, y por consiguiente se entrena a la red neuronal con información errónea. De esta manera, siempre se les asigna la categoría correcta a las letras segmentadas adecuadamente.

- Durante la clasificación, obtendríamos siempre una palabra de 4 letras en la que las letras que fueron segmentadas adecuadamente están en la posición que les corresponde. Esto le da más opciones a la fase de post-procesamiento a encontrar la palabra correcta al comparar la distancia con las palabras del diccionario.
- A pesar de que la separación manual de las imágenes puede contener errores, le da oportunidad a la red neuronal a identificar las letras que representan aunque solo tenga partes de ellas, o información 'innecesaria' de otras letras dependiendo del caso.
- Los patrones que haya con ciertas letras que juntas provoquen una segmentación errónea pueden ser útiles para la clasificación de la red neuronal. Asimismo, en los casos en los que la red sea incapaz de clasificar correctamente, puede que también aparezca algún patrón que se refleje en la matriz de confusión, y por lo tanto sirva para la comparación con el diccionario durante el post-procesamiento.

Capítulo 8

Extended summary and conclusions

8.1. Extended summary

In this project, a program that can recognize certain types of CAPTCHAs is implemented. To do so, we train a neural network to be able to recognize images of individual letters that are part of a CAPTCHA.

To create the training dataset, we first implement a function that takes a word and a shear value and returns the corresponding CAPTCHA. We then use this function to create CAPTCHAs with random shear values, divide them into the images of individual letters and associate each image with its category (which letter it represents). All images of individual letters are resized to 20x20 pixels.

After that, the neural net is created with 400 neurons in the input layer (one for each pixel of the images), 100 neurons in the input layer (decided after several tests) and 26 neurons in the output layer (one for each possible letter). It is trained with the dataset we created and ready to predict letters.

To be able to get a word from a CAPTCHA, we divide the CAPTCHA as we did when creating the training dataset, that is, by checking the pixels that are touching each other and extracting a rectangular region for each set of pixels so that it contains all of them. Finally, we activate the net for each image, join the letters obtained and compare the result with a dictionary.

8.2. Conclusions

In conclusion, I have found this project very interesting because it has made me explore the world of artificial intelligence more deeply than before. In particular, I have been able to learn a lot about neural networks, the difficulties that can appear when working with them and its huge potential.

One of the main ideas drawn of this project is the importance of the quality of the data to get the best possible results, because these are key when training a neural network. Whenever possible, data should be as likely as possible to what we will face in the use of the network in the real world. Likewise, the quantity of data is also very important as the model gets more complex, which makes more time and computational power necessary to obtain relevant results.

Lastly, even though neural networks can perform tasks that could seem very complex or impossible otherwise, in many cases an appropiate post-processing can fix mistakes made by the neural network and therefore improve obtained results.

Bibliografía

Wikipedia. (s.f.) Neural network. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Neural_network

Wikipedia. (s.f.) Artificial neural network. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Artificial_neural_network

Wikipedia. (s.f.) Red neuronal artificial. Obtenido de Wikipedia.org: https://es.wikipedia.org/wiki/Red_neuronal_artificial

Wikipedia. (s.f.) Neurona. Obtenido de Wikipedia.org: https://es.wikipedia.org/wiki/Neurona

PyBrain. (s.f.) Trainers. Obtenido de PyBrain.org: http://pybrain.org/docs/api/supervised/trainers.html

PyBrain. (s.f.) Documentación buildNetwork. Obtenido de PyBrain.org: http://pybrain.org/docs/api/tools.html

Scikit Learn. (s.f.) Documentación confusion_matrix. Obtenido de scikit-learn.org: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

PyBrain. (s.f.) Installing PyBrain and its dependencies. Obtenido de PyBrain.org: http://pybrain.org/docs/quickstart/installation.html

Python docs. (s.f.) The Python Language Reference. Obtenido de docs.python.org: https://docs.python.org/3/reference/index.html

Scikit-Learn. (s.f.) Documentación cross_validate. Obtenido de scikit-learn.org: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html

Wikipedia. (s.f.) One-hot. Obtenido de Wikipedia.org: https://es.wikipedia.org/wiki/One-hot