

Autor: Adrian Maiza Coupin

Fecha: 08 de Enero de 2021

INTRODUCCIÓN:

Este trabajo de Visión Artificial trata de abordar el siguiente problema: reconocer señales de tráfico por medio de visión artificial e inteligencia artificial. Está separado en tres fases: clasificación de señales, que trata de clasificar cada una de las imágenes en su respectiva clase; detección de señales, que trata de detectar todas las señales en una imagen; y localización y clasificación, la cual combina las dos anteriores fases.

Toda la información expuesta en este informe viene directamente de los resultados obtenidos a partir del código proporcionado en el *Notebook de Jupyter TrabajoFinal.ipynb*.

FASE 1 : Clasificación de señales



Dado que los datos que tenemos en este problema son imágenes, tenemos que extraer información que nos pueda servir a la hora de clasificar dichas imágenes en sus respectivas clases. Existen muchos métodos para hacer esto, desde el algoritmo de Viola-Johns [1] (características de Haar) hasta extracción de características mediante el Histograma de Gradientes Orientados (*Histogram of Oriented Gradients* o *HOG*) [2]. En este trabajo utilizamos esta última técnica. A partir de estas características podemos pasar a la generación de modelos que clasifiquen estas imágenes.

Recogida de datos

Para empezar, a partir del *dataset* proporcionado, se han recogido los datos de cada una de las 43 clases y más de 1200 imágenes, transformando todas ellas a un tamaño de 83x83 píxeles. El tamaño es algo peculiar, pero haciendo pruebas con distintos tamaños (desde 64x64 hasta 256x256), se puede observar bastante mejoría en el porcentaje de acierto de todos los modelos y sobre todo a la hora de extraer las características *HOG* [2]. Además, se han cargado todas las imágenes a escala de grises, pues después de hacer pruebas parece funcionar mejor que con los tres canales de colores.

Procesado de imágenes

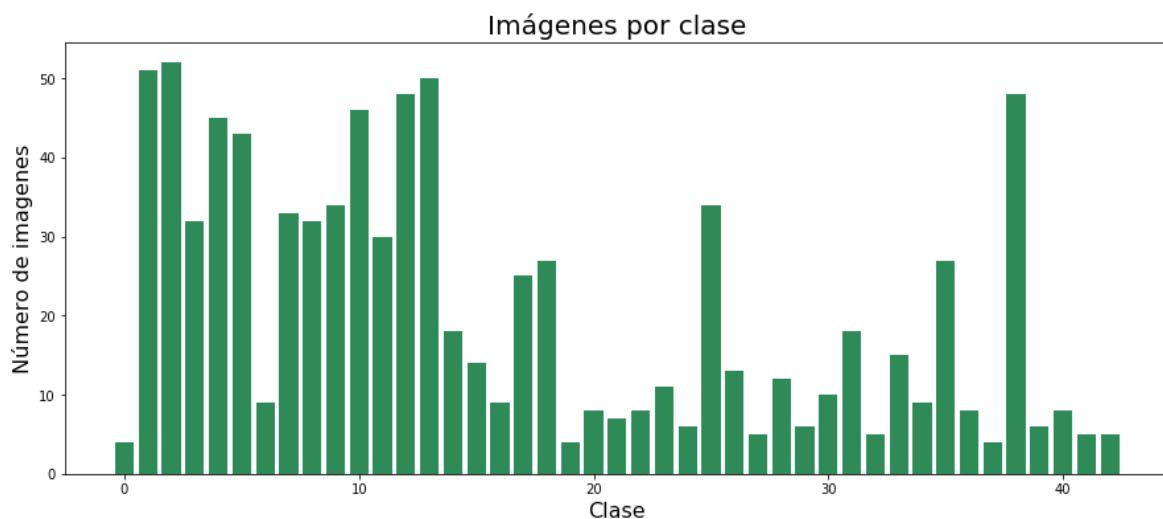
Respecto al tratamiento de las imágenes, no ha hecho falta utilizar ninguna de las técnicas aprendidas en clase, tales como estiramiento del rango dinámico, ecualización del histograma o filtrado por Sobel, por ejemplo, dado que la implementación del algoritmo *HOG* utilizado (OpenCV) seguramente ya procesa las imágenes antes de extraer las características.

Separación de datos

Para el posterior análisis de los distintos modelos que se usarán para la clasificación, hace falta separar los datos en tres conjuntos de datos diferentes: *train*, con el que se entrenará a todos los modelos; *val*, con el que se compararán los modelos; y *test*, que será el que da el porcentaje de acierto real del modelo con esa configuración. Para ello se usa la función `train_test_split` de `sklearn.model_selection` con los tamaños siguientes: 70% para *train*, 15% para *val* y 15% para *test*. También se hace uso del argumento `stratify` para equilibrar bien los conjuntos de imágenes.

Balanceado de clases

Es importante comentar que se ha barajado la posibilidad de usar *oversampling*, dado que el *database* proporcionado está bastante desequilibrado en cuanto a imágenes por clase. Aquí se puede ver una gráfica que lo refleja:

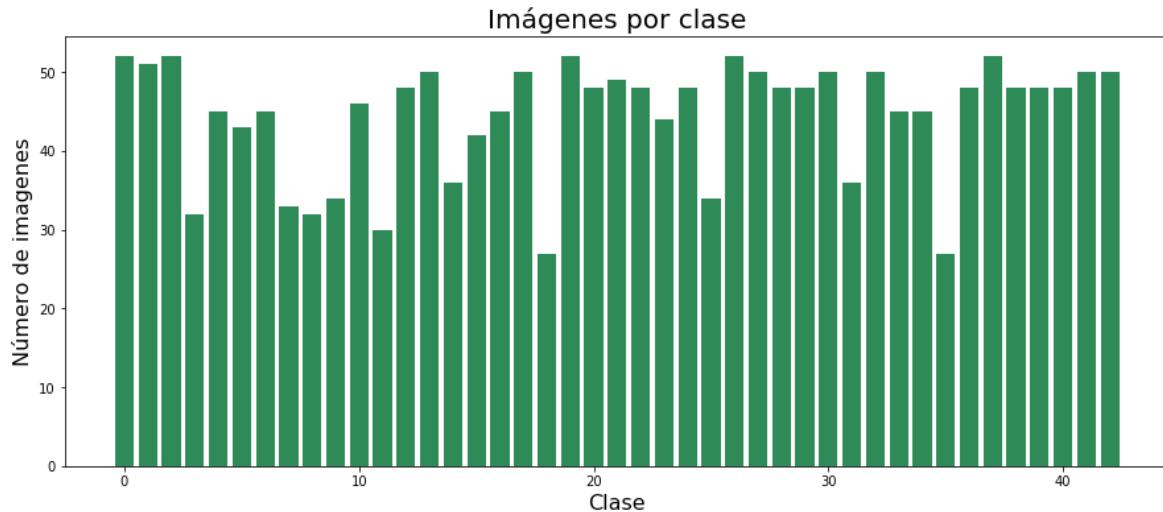


Se usaría el método de *oversampling* de la siguiente manera:

Primero se calcula para cada clase el número de veces que se repetirán todas las imágenes de la clase:

Lo único que se hace aquí es restar el número máximo de imágenes de entre todas las clases con el número de elementos de la clase *i* y dividir esto por el número de elementos de la clase *i*.

Después se usa un método de distorsión de imágenes. El resultado es el siguiente:



Y aquí un ejemplo de la distorsión de las imágenes de una de las clases: (se muestran a color para mejor visualización)



Aún así, el uso de este método es innecesario pues los resultados obtenidos en general no han sido mejores que sin haberlo usado.

Estandarización de datos

Según las diferentes pruebas realizadas, no se considera necesaria la estandarización de los datos.

Usando distintos modelos

Se han utilizado los siguientes métodos de clasificación: *SVM*, *NN*, *CNN*, *Random Forests*, *OVA* (con *SVM*), *OVO* (con *SVM*), *Decision Trees*, Naïve-Bayes y *LDA*. Los resultados de cada uno de ellos se muestra en la siguiente tabla:

Precisión	SVM	NN	CNN	RF	OVA	OVO	DT	NB	LDA
Validación	99.474%	95.789%	84.211%	96.842%	100%	99.474%	75.263%	94.211%	98.947%
Train	100%	100%	99.661%	100%	100%	100%	100%	98.643%	100%
Test	96.316%	94.211%	86.316%	93.684%	96.316%	95.789%	75.789%	86.842%	95.263%

Además, es importante mostrar el tiempo de clasificación de cada conjunto de datos de cada uno de estos modelos:

Tiempo	SVM	NN	CNN	RF	OVA	OVO	DT	NB	LDA
Validación	3.14 s	24 ms	2.03 s	120 ms	8.61 s	1 min 46 s	21 ms	2.55 s	17 ms
Train	680 ms	8 ms	478 ms	32 ms	1.81 s	22.7 s	5 ms	539 ms	5 ms
Test	726 ms	7 ms	489 ms	32 ms	1.83 s	22.4 s	5 ms	555 ms	3 ms

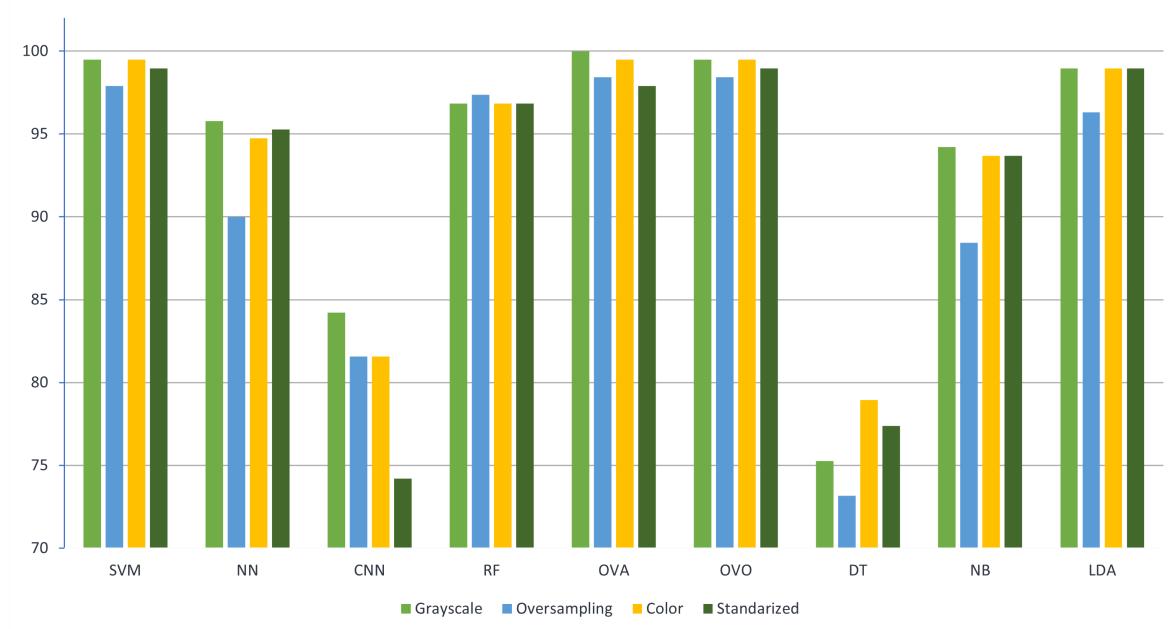
Obviamente, esta tabla está para mostrar las diferencias entre los tiempos de ejecución y no para mostrar como de rápido irá en realidad, puesto que el tiempo de ejecución depende íntegramente de en que máquina se ejecuta.

El tiempo que le cuesta entrenar a cada modelo no es demasiado importante, pues el entrenamiento se puede hacer de manera local en un ordenador potente y exportar luego el modelo para poder cargarlo en otros equipos. Como puede observarse en el código proporcionado, esto se hace con las funciones `dump` y `load` de la librería `joblib`.

Todas las configuraciones de los modelos (por ejemplo los parámetros *C* y *gamma* de la *SVM*) aparecen en el código proporcionado.

Comparaciones

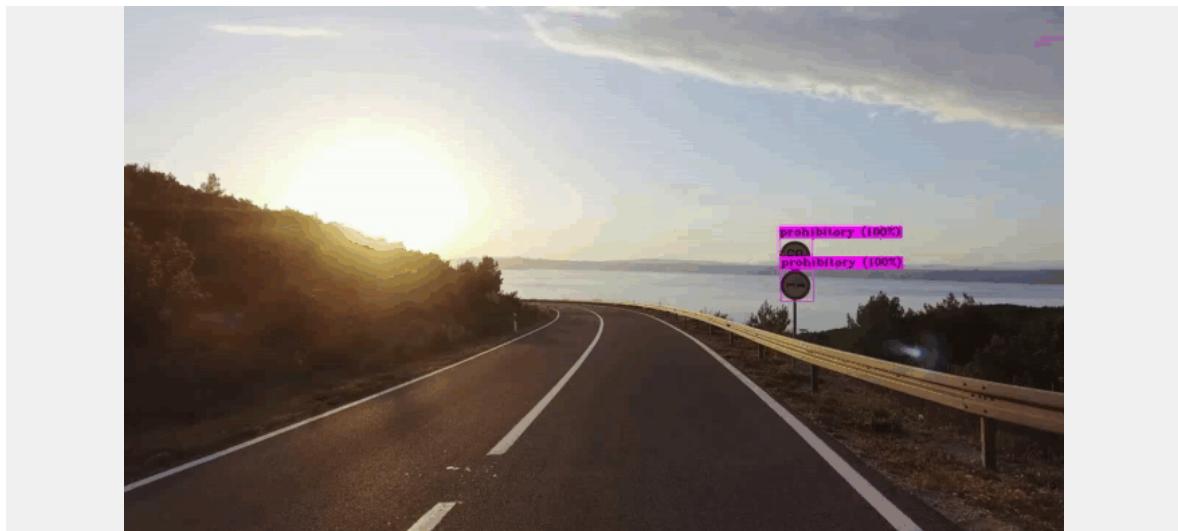
Aquí se muestra una gráfica de barras que muestra las diferencias entre las precisiones de cada modelo siguiendo distintos métodos explicados anteriormente. Es decir, usar *oversampling*, las imágenes a color o estandarizar los datos (con *MinMaxScaler*). Únicamente aparecen los porcentajes de precisión en validación.



Conclusión

Así pues, viendo los datos anteriores, el mejor modelo obtenido es el **ensemble OVA** con **SVM** cuyos parámetros son $C = 100$ y $\gamma = 0.1$ entrenado con el conjunto de entrenamiento de las imágenes a **escala de grises sin procesar ni estandarizar**. Este modelo nos da un porcentaje de acierto en **validación** del **100%** y el resultado final del conjunto de **test** es de **96.316%**. Por lo tanto vamos a guardar este modelo y lo entrenamos con el dataset al completo.

FASE 2 : Detección de señales



Esta parte trata de identificar y marcar señales de tráfico en una imagen. Para ello necesitamos generar un dataset de "parches" que nos den información de objetos que se hallan con frecuencia en la carretera, tales como árboles, coches, edificios, nubes, postes eléctricos, etc. Estos parches serán nuestra clase NO señal. Para la clase SI señal utilizaremos las señales proporcionadas en el dataset de la parte 1.

Generación de parches

Para generar estos parches, se han cogido varias imágenes al azar y se han extraído bloques en diferentes tamaños mediante el uso de funciones como `pyramid_gaussian` de `skimage.transform` para generar imágenes con menor resolución pero manteniendo el ratio original, y utilizando un algoritmo de ventana deslizante bloque por bloque (se puede ver el código en el *Notebook* de *Jupyter*). Después de extraer todos estos fragmentos de imágenes de varias imágenes del dataset, se han limpiado a mano para mantener una homogeneidad en el dataset, eliminando todos aquellos parches de señales que se hubiesen generado.

Recogida de datos

En este apartado se han reutilizado las imágenes de las señales de 83x83 cargadas anteriormente. Como estas imágenes ya están separadas de manera equilibrada en los tres conjuntos de `train`, `val` y `test` de la primera parte, lo único que se ha hecho es cambiar las clases a una sola (SI señal) y mezclar después con las imágenes de NO señal, reordenando de manera aleatoria. El tamaño final del dataset, entre las imágenes de las señales y los parches de NO señal, es de unas 5700 imágenes.

Procesado de parches

Al igual que en la fase anterior, se ha decidido no procesar las imágenes antes de extraer las características de HOG.

Calculo de features de HOG

De la misma manera que en la fase anterior, se calculan los parametros HOG de los tres datasets. Esta vez el número de Bins va a ser 31 en vez de 9, lo que hará que el número de características para los modelos sea bastante más grande, teniendo en cuenta que el dataset de imágenes es ahora unas 4.5 veces el tamaño del dataset de la fase 1. Esto hara que el proceso, tanto de entrenamiento como de predicción, sea más lento.

Usando distintos modelos

Se han utilizado los siguientes métodos de clasificación: *SVM*, *NN*, *CNN* (con imágenes a color), *Random Forests*, *OVA* (con *SVM*), *Decision Trees* y *Naïve-Bayes*. Los resultados de cada uno de ellos se muestra en la siguiente tabla:

Precisión	SVM	NN	CNN	RF	OVA	DT	NB
Validación	99.533%	99.533%	98.716%	99.183%	99.533%	96.033%	93.699%
Train	100%	100%	98.973%	100%	100%	100%	93.741%
Test	99.533%	99.533%	98.133%	99.299%	99.533%	96.266	93.816%

Además, es importante mostrar el tiempo de clasificación de cada conjunto de datos de cada uno de estos modelos:

Tiempo	SVM	NN	CNN	RF	OVA	DT	NB
Validación	24.5 s	303 ms	7.62 s	537 ms	26.6 s	260 ms	2.19 s
Train	5.26 s	42 ms	1.67 s	136 ms	5.79 s	62 ms	457 ms
Test	5.26 s	44 ms	1.66 s	122 ms	5.38 s	49 ms	358 ms

Conclusión de selección de modelos

Ya que los porcentajes de acierto en validación de los modelos *SVM*, *OVA* y *NN* son los mismos, escogemos la *NN* por ser la más rápida clasificando. Al igual que antes, se vuelve a entrenar el modelo *NN* con todos los datos.

PROBANDO DISTINTOS MÉTODOS DE DETECCIÓN

Aunque en el enunciado del problema a tratar se propone un algoritmo de ventana deslizante para la detección de señales en la imagen, aquí se propone, además de ésta, otro método diferente que será explicado más adelante. En todos los métodos probados se hace uso de la función `cv2.dnn.NMSBoxes` para la supresión de no máximos para la eliminación de múltiples positivos en la misma zona.

1. Ventana deslizante

Aquí se intenta solucionar el problema mediante un algoritmo de ventana deslizante. Se utilizan 4 tamaños diferentes de ventana con un desplazamiento de diferentes tamaños, para evitar ralentizar demasiado el proceso. Aquí se muestran algunos de los resultados obtenidos con este método:



Como se puede observar, los resultados son bastante mediocres. Además, el tiempo de ejecución es bastante lento pues se recorre la imagen varias veces con distintos tamaños de ventana que son cada vez más pequeños. Al no conseguir corregir tantos errores, se propone lo siguiente.

2. Máscaras de color y Canny [4]

La idea principal de este método [3] es la de utilizar los colores que más se repiten en las señales de tráfico (azul, rojo, amarillo, ...) junto con la detección de bordes de Canny [4] (para las señales monocromáticas) para luego hacer uso de un MSER [5] para extraer puntos de interés. Esto permite deshacerse de regiones sin importancia de la imagen, aumentando considerablemente la velocidad de procesamiento y detección.

Aquí se tienen algunos ejemplos del resultado de aplicar máscaras de color y el detector de bordes de Canny [4]:



Para extraer las zonas de los colores deseados, se han ecualizado las imágenes y se han pasado al espacio de color HSV para mejorar la robustez en los cambios de iluminación

Después de eso se pasa la imagen generada por el detector de MSER y se utilizan esas regiones como las imágenes a pasar por el modelo seleccionado en el apartado anterior.

Aquí se muestran algunos de los resultados obtenidos con este método:



Como se puede observar, los resultados son bastante mejores que en el algoritmo de ventana deslizante. Es por eso mismo que se utiliza este método para la fase siguiente.

FASE 3: Detección y clasificación de señales

En esta última parte se juntan las dos partes anteriores. Mediante el modelo elegido en la fase 1 y el método de detección de señales de la fase 2, se marcarán y clasificarán las señales en las imágenes de entrada.

Aquí se muestran algunos de los resultados obtenidos:



Implementación adicional de red YOLO v3

En este apartado se usa una red Darknet[6] YOLOv3[7] entrenada en los servidores de Google Colaboratory, mediante el uso de sus frameworks de GPU con el dataset de entrenamiento TrainIJCNN2013[8] obtenido en <https://sid.berkeley.edu/public/archives/f17dc924eba88d5d01a807357d6614c/published-archive.html>", haciendo uso del fichero gt.txt que tiene las coordenadas de las señales y la clase a la que pertenecen. Hizo falta transformar estos datos para adaptarlos a YOLO. Está disponible en la carpeta YOLO_v3 el notebook de Jupyter con el código de entrenamiento del modelo, además del código que transforma los datos mencionados anteriormente.

Así pues, aquí se muestran algunos de los resultados obtenidos mediante el uso de este modelo:



Resultados

Todos los resultados obtenidos se almacenan en [repositorio del proyecto](#) [9] así como en el zip de [Google Drive](#) [10] y en [OneDrive](#) [11], por si se quieren visualizar las imágenes.

Video

Puedes ver el video de la presentación del proyecto en [este enlace](#). Siento que sea algo largo, pero sino no podía explicar todo lo que había probado. Te recomiendo ponerlo a velocidad x2.

Referencias

- [1] Viola, Paul & Jones, Michael. (2004). Robust Real-Time Face Detection. International Journal of Computer Vision. 57. 137-154.
- [2] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in Proc. IEEE CVPR, 2005, vol. 1, pp. 886–893.
- [3] Y. Yang, H. Luo, H. Xu and F. Wu, "Towards Real-Time Traffic Sign Detection and Classification," in IEEE Transactions on Intelligent Transportation Systems, vol. 17, no. 7, pp. 2022-2031, July 2016
- [4] Canny, J., A Computational Approach To Edge Detection, IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.
- [5] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," Image Vis. Comput., vol. 22, no. 10, pp. 761–767, 2004.
- [6] Joseph Redmon, "Darknet: Open Source Neural Networks in C", <http://pjreddie.com/darknet/>, 2013-2016
- [7] Redmon, Joseph and Farhadi, Ali, YOLOv3: An Incremental Improvement, arXiv, 2018
- [8] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The German traffic sign recognition benchmark: a multi-class classification competition," in Proc. IEEE IJCNN, 2011, pp. 1453–1460.
- [9] A. Maiza, "Proyecto final de Visión Artificial" <https://github.com/maizabros/TrafficSignals> , 2021
- [10] A. Maiza, "Proyecto final de Visión Artificial" https://drive.google.com/file/d/1L0gjXy0L2DgD4bMsмоYs_52Higs52u2u/view?usp=sharing , 2021
- [11] A. Maiza "Proyecto final de Visión Artificial", 2021 https://unavarra-my.sharepoint.com/:u/g/personal/maiza_115667_e_unavarra_es/EZq2RTzAJlMudV60RwKktoBC6QUE9Y0uRm3J8Xq5ENogw?e=0T45FY