

SEÑALES DE TRAFICO

Aliendo Marcos

ESTRUCTURA DE PRESENTACIÓN

01

Problema vs.
solución

03

Resultados

02

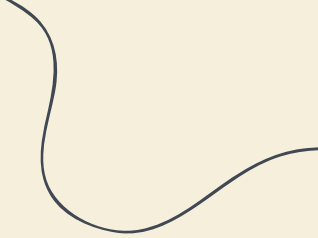

Código

04

Conclusión




INTRODUCCIÓN



**Problema vs.
solución**

01



PROBLEMA

Mientras la investigación avanza en el desarrollo de coches autónomos, uno de los retos clave es la visión por computadora, que permite a estos coches que desarrollen una comprensión de su entorno a partir de imágenes digitales.



SOLUCIÓN

Utilizaremos TensorFlow para construir una red neuronal que sea capaz de clasificar señales de tráfico basadas en imágenes de dichas señales.

DATASET

GTSRB

German Traffic Sign Recognition Benchmark.

Este dataset alemán contiene

+ 40 clases de señales de tráfico.

+ 50.000 imágenes en total.

PROGRAMAR UNA RED NEURONAL

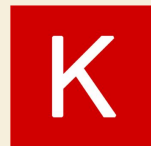
Modelos completos

Pasamos a tratar a las redes neuronales de un punto de vista de capas a un punto de vista de modelo.



Composición de capas

El código programado con Keras en mucho más compacto y orientado al prototipado rápido de arquitecturas de deep learning.



Librerías de diferenciación automática

Se encargan de calcular automáticamente las derivadas parciales para implementar cualquier arquitectura que se diseñe.



VISION ARTIFICIAL

OpenCV

Es una biblioteca de software de código abierto ampliamente utilizada para aplicaciones de visión por computadora y machine learning.

- Clasificación de imágenes.
- Localización de objetos.
- Detección de objetos.
- Segmentación de imágenes.

La visión por computadora es un campo científico interdisciplinario que se ocupa de cómo las computadoras pueden obtener una comprensión de alto nivel a partir de imágenes o videos digitales.





Código

02

CODIGO

01

MAIN

Aceptamos como argumentos el dataset y nombre del modelo entrenado.

02

LOAD DATA

Se dividen los datos de entrenamiento y test

03

GET MODEL

Se obtiene la red neuronal compilada que luego se ajusta en función de los datos de prueba.

LOAD DATA

Esta función recibe como parámetro el directorio donde se encuentran las imágenes y devuelve dos listas, una con las imágenes y otra con las etiquetas de salida.

Las etiquetas de salida son los nombres de las carpetas que contienen las imágenes.

Por ejemplo, si tenemos una imagen de un semáforo, la etiqueta de salida será 0, ya que la imagen se encuentra en la carpeta 0. Si tenemos una imagen de un límite de velocidad de 30 km/h, la etiqueta de salida será 1, ya que la imagen se encuentra en la carpeta 1. Y así sucesivamente.

SEPARAMOS LOS DATOS DE ENTRENAMIENTO DE LOS DE PRUEBA

La función de scikit learn **train_test_split()** nos permite separar los datos de entrenamiento de los de prueba. Esta función recibe como parámetros los datos de entrada y las etiquetas de salida y nos devuelve los datos de entrada de entrenamiento, los datos de entrada de prueba, las etiquetas de salida de entrenamiento y las etiquetas de salida de prueba.

El parámetro **test_size** nos permite indicar el porcentaje de datos que queremos que se utilicen para el conjunto de prueba. En este caso hemos indicado que el 40% de los datos se utilicen para el conjunto de prueba, por lo que el 60% de los datos se utilizarán para el conjunto de entrenamiento.

GET MODEL

```
# ALTERNATIVA 1
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(NUM_CATEGORIES, activation='softmax') # Cambia el número de unidades en
    la capa de salida a 3
])
```

keras.Sequential, es una forma de decirle a keras que queremos crear un modelo conformado por una secuencia de capas.

Conv2D, capa convolucional de dos dimensiones que prende 32 filtros.

MaxPooling2D, aplica convolucion, tamaño de pooling 2x2.

Funcion **flatten**, convierte la matriz en un arreglo.

Capa de salida, **capa densa**, softmax da prediccion porcentual.

GET MODEL

```
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
```

Lo primero que tenemos que hacer es compilar nuestro modelo, para así comunicar al backend de TensorFlow que queremos entrenar nuestro modelo con el algoritmo de optimización **Adam**.

Con el método compile se entrena el modelo.

Loss es función de pérdida. **binary_crossentropy** es la mejor para caracterizar perdida en clasificación binaria.

Y la métrica es la precisión (**accuracy**).

ENTRENAMIENTO DEL MODELO

```
model.fit(x_train, y_train, epochs=EPOCHS)
```

Una vez que tenemos el modelo compilado, tenemos que entrenarlo y esto se consigue llamando al método **fit()** de nuestro modelo.

Este método recibe como parametros los datos de entrada de entrenamiento, las etiquetas de salida de entrenamiento y el número de épocas que queremos entrenar nuestro modelo.

Una época es una pasada completa por todos los datos de entrenamiento

EVALUAMOS EL MODELO

```
model.evaluate(x_test, y_test, verbose=2)
```

- Valor de pérdida.
<
- Precisión del modelo.
>



Resultados

03

ALTERNATIVA 1

```
# ALTERNATIVA 1
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(NUM_CATEGORIES, activation='softmax') # Cambia el número de unidades en
    la capa de salida a 3
])
```

RESULTADOS

```
Epoch 1/10
500/500 [=====] - 10s 20ms/step - loss: 0.2218 - accuracy: 0.5699
Epoch 2/10
500/500 [=====] - 10s 20ms/step - loss: 0.0268 - accuracy: 0.8764
Epoch 3/10
500/500 [=====] - 10s 20ms/step - loss: 0.0176 - accuracy: 0.9284
Epoch 4/10
500/500 [=====] - 10s 20ms/step - loss: 0.0122 - accuracy: 0.9546
Epoch 5/10
500/500 [=====] - 10s 20ms/step - loss: 0.0116 - accuracy: 0.9619
Epoch 6/10
500/500 [=====] - 10s 20ms/step - loss: 0.0098 - accuracy: 0.9705
Epoch 7/10
500/500 [=====] - 10s 20ms/step - loss: 0.0107 - accuracy: 0.9700
Epoch 8/10
500/500 [=====] - 11s 22ms/step - loss: 0.0072 - accuracy: 0.9814
Epoch 9/10
500/500 [=====] - 10s 20ms/step - loss: 0.0069 - accuracy: 0.9823
Epoch 10/10
500/500 [=====] - 10s 20ms/step - loss: 0.0072 - accuracy: 0.9825
333/333 - 1s - loss: 0.0186 - accuracy: 0.9521 - 1s/epoch - 3ms/step
```

ALTERNATIVA 2

```
# ALTERNATIVA 2
model = tf.keras.models.Sequential([
    # Capas de convolución y pooling
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    # Capas totalmente conectadas
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(NUM_CATEGORIES, activation='softmax')
])
```

RESULTADOS

```
Epoch 1/10
500/500 [=====] - 7s 12ms/step - loss: 0.1089 - accuracy: 0.4593
Epoch 2/10
500/500 [=====] - 6s 12ms/step - loss: 0.0284 - accuracy: 0.8485
Epoch 3/10
500/500 [=====] - 6s 12ms/step - loss: 0.0148 - accuracy: 0.9349
Epoch 4/10
500/500 [=====] - 7s 14ms/step - loss: 0.0104 - accuracy: 0.9600
Epoch 5/10
500/500 [=====] - 7s 14ms/step - loss: 0.0071 - accuracy: 0.9758
Epoch 6/10
500/500 [=====] - 7s 13ms/step - loss: 0.0061 - accuracy: 0.9800
Epoch 7/10
500/500 [=====] - 7s 14ms/step - loss: 0.0056 - accuracy: 0.9814
Epoch 8/10
500/500 [=====] - 7s 13ms/step - loss: 0.0049 - accuracy: 0.9849
Epoch 9/10
500/500 [=====] - 7s 13ms/step - loss: 0.0038 - accuracy: 0.9886
Epoch 10/10
500/500 [=====] - 7s 13ms/step - loss: 0.0044 - accuracy: 0.9874
333/333 - 1s - loss: 0.0089 - accuracy: 0.9719 - 1s/epoch - 4ms/step
```

ALTERNATIVA 3

```
# ALTERNATIVA 3
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (5, 5), activation='relu', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.3), # va descartando la mitad de las entidades previene el overfitting
    tf.keras.layers.Dense(NUM_CATEGORIES, activation='sigmoid')
])
```

El valor del dropout afecta la regularización
y puede prevenir el sobreajuste

RESULTADOS

```
Epoch 1/10  
500/500 [=====] - 30s 59ms/step - loss: 0.2510 - accuracy: 0.4175  
Epoch 2/10  
500/500 [=====] - 31s 61ms/step - loss: 0.0452 - accuracy: 0.7419  
Epoch 3/10  
500/500 [=====] - 27s 54ms/step - loss: 0.0323 - accuracy: 0.8253  
Epoch 4/10  
500/500 [=====] - 26s 52ms/step - loss: 0.0259 - accuracy: 0.8680  
Epoch 5/10  
500/500 [=====] - 26s 52ms/step - loss: 0.0215 - accuracy: 0.8964  
Epoch 6/10  
500/500 [=====] - 27s 53ms/step - loss: 0.0193 - accuracy: 0.9047  
Epoch 7/10  
500/500 [=====] - 26s 52ms/step - loss: 0.0171 - accuracy: 0.9232  
Epoch 8/10  
500/500 [=====] - 26s 52ms/step - loss: 0.0159 - accuracy: 0.9281  
Epoch 9/10  
500/500 [=====] - 26s 52ms/step - loss: 0.0151 - accuracy: 0.9321  
Epoch 10/10  
500/500 [=====] - 26s 52ms/step - loss: 0.0146 - accuracy: 0.9352  
333/333 - 2s - loss: 0.0150 - accuracy: 0.9437 - 2s/epoch - 5ms/step
```


ALTERNATIVA 4

```
# ALTERNATIVA 4
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'), # Nueva capa densa oculta
    tf.keras.layers.Dense(32, activation='relu'), # Nueva capa densa oculta
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(NUM_CATEGORIES, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

RESULTADOS

```
Epoch 1/10  
500/500 [=====] - 19s 38ms/step - loss: 3.9324 - accuracy: 0.2308  
Epoch 2/10  
500/500 [=====] - 18s 36ms/step - loss: 1.8342 - accuracy: 0.4754  
Epoch 3/10  
500/500 [=====] - 17s 35ms/step - loss: 1.3722 - accuracy: 0.5915  
Epoch 4/10  
500/500 [=====] - 17s 35ms/step - loss: 1.1149 - accuracy: 0.6715  
Epoch 5/10  
500/500 [=====] - 17s 35ms/step - loss: 0.9003 - accuracy: 0.7261  
Epoch 6/10  
500/500 [=====] - 18s 36ms/step - loss: 0.8079 - accuracy: 0.7603  
Epoch 7/10  
500/500 [=====] - 17s 34ms/step - loss: 0.7362 - accuracy: 0.7858  
Epoch 8/10  
500/500 [=====] - 17s 35ms/step - loss: 0.6379 - accuracy: 0.8093  
Epoch 9/10  
500/500 [=====] - 17s 35ms/step - loss: 0.5443 - accuracy: 0.8398  
Epoch 10/10  
500/500 [=====] - 17s 35ms/step - loss: 0.5253 - accuracy: 0.8487  
333/333 - 1s - loss: 0.4024 - accuracy: 0.9055 - 1s/epoch - 4ms/step
```

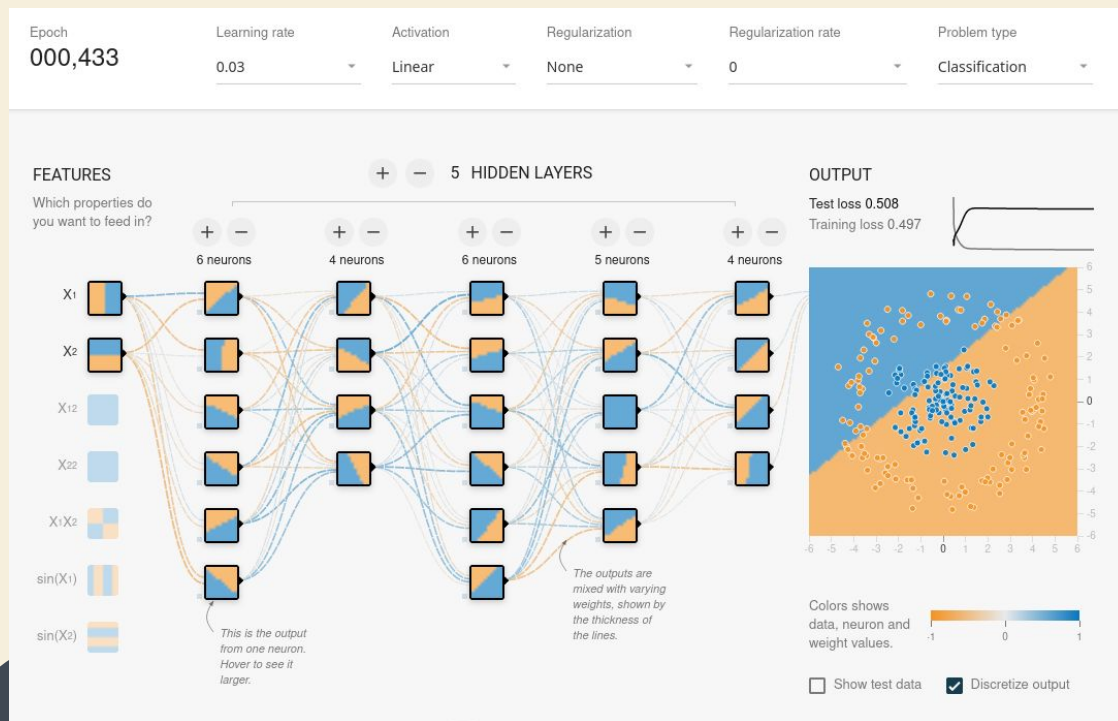
ALTERNATIVA 5

```
# ALTERNATIVA 5
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(IMG_WIDTH, IMG_HEIGHT, 3)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='linear'), # Nueva capa densa oculta
    tf.keras.layers.Dense(64, activation='linear'), # Nueva capa densa oculta
    tf.keras.layers.Dense(64, activation='linear'), # Nueva capa densa oculta
    tf.keras.layers.Dense(64, activation='linear'), # Nueva capa densa oculta
    tf.keras.layers.Dense(64, activation='linear'), # Nueva capa densa oculta
    tf.keras.layers.Dense(64, activation='linear'), # Nueva capa densa oculta
    tf.keras.layers.Dense(32, activation='tanh'), # Nueva capa densa oculta
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(NUM_CATEGORIES, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

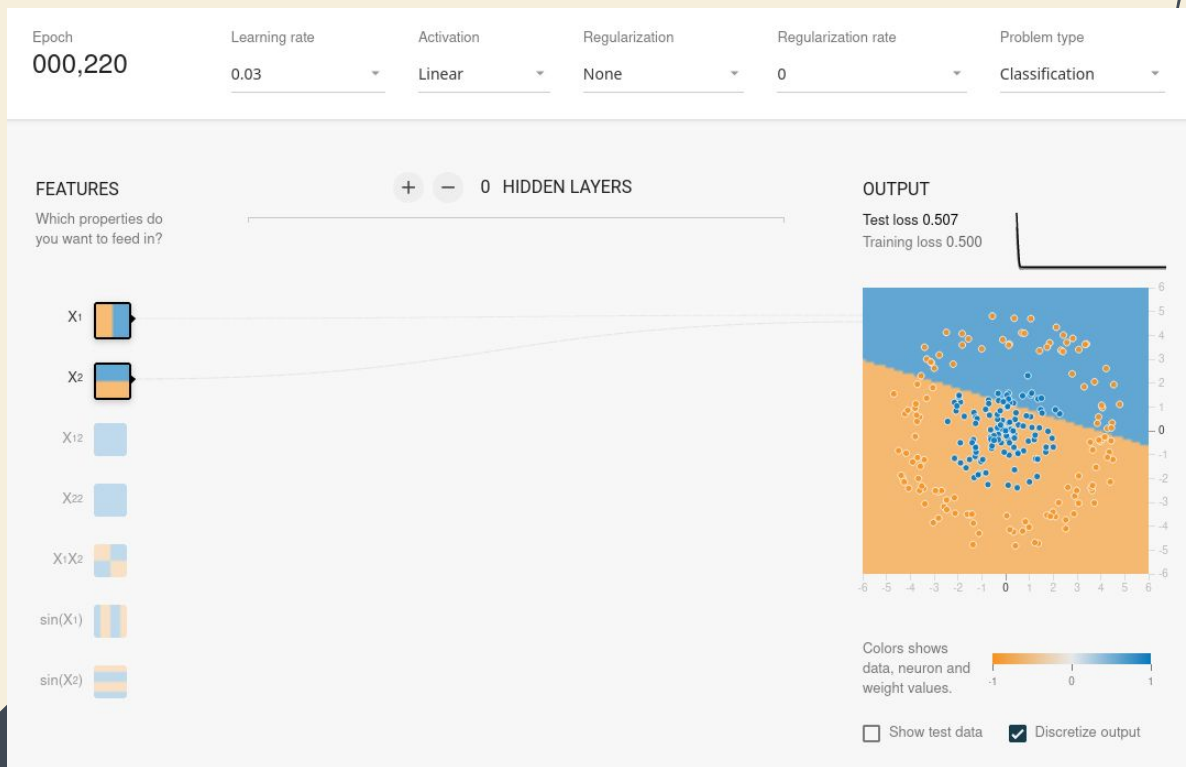
RESULTADOS

```
Epoch 1/10
500/500 [=====] - 18s 35ms/step - loss: 3.7259 - accuracy: 0.0439
Epoch 2/10
500/500 [=====] - 18s 35ms/step - loss: 3.6059 - accuracy: 0.0502
Epoch 3/10
500/500 [=====] - 17s 35ms/step - loss: 3.5811 - accuracy: 0.0521
Epoch 4/10
500/500 [=====] - 17s 35ms/step - loss: 3.5613 - accuracy: 0.0534
Epoch 5/10
500/500 [=====] - 17s 34ms/step - loss: 3.5477 - accuracy: 0.0509
Epoch 6/10
500/500 [=====] - 17s 35ms/step - loss: 3.5323 - accuracy: 0.0564
Epoch 7/10
500/500 [=====] - 17s 35ms/step - loss: 3.5296 - accuracy: 0.0541
Epoch 8/10
500/500 [=====] - 17s 34ms/step - loss: 3.5262 - accuracy: 0.0535
Epoch 9/10
500/500 [=====] - 17s 35ms/step - loss: 3.5218 - accuracy: 0.0538
Epoch 10/10
500/500 [=====] - 17s 35ms/step - loss: 3.5162 - accuracy: 0.0541
333/333 - 2s - loss: 3.5037 - accuracy: 0.0515 - 2s/epoch - 5ms/step
```

RESULTADOS




RESULTADOS





Conclusión

04





GRACIAS!