PROYECTO HORUS

Taller de Robótica Educativa con el Robot Butiá

Federico Gil (5.198.750-6), federico.gil@fing.edu.uy, Ingeniería en Computación.
Facundo Fleitas (5.072.134-9), facundo.fleitas@fing.edu.uy, Ingeniería en Computación.

Tutores: Guillermo Trinidad, Gonzalo Tejera.

Abstract:
Motivados por la falta de herramientas que permitan la accesibilidad a personas con capacidades motoras reducidas, decidimos desarrollar una herramienta multipropósito que permita diversas aplicaciones, desde una silla de ruedas hasta controlar un mouse, todo esto mediante la detección de la dirección en el que se mira. Para esto diseñamos y entrenamos una red neuronal convolucional que clasifica las direcciones: derecha, izquierda, centro y los ojos cerrados. A partir de esto se le pueden dar infinidad de usos a esta herramienta, la cual alcanzó resultados muy satisfactorios, con una tasa de acierto de más del 98%. Para fines demostrativos haremos una analogía entre controlar una silla de ruedas y controlar el robot butiá. En el siguiente documento incluimos una breve introducción a los conceptos de redes neuronales, convolucionales y distintas métricas, también incluiremos los resultados obtenidos, así como una guía para su replicación e implementación.

Adicionalmente, se creó un material audiovisual en el que explicamos brevemente las bases teóricas y en detalle la utilización de esta herramienta para el control del robot Butiá.

Palabras clave: Inteligencias artificiales, Redes Convolucionales, Robótica, Software, Modelo Neuronal, Detección de ojos.

Facultad de Ingeniería. Universidad de la República.
Montevideo, Uruguay.

Índice

Motivación

Motivados por la falta de accesibilidad que presentan las personas con reducciones motoras importantes y ante la falta de herramientas que lo permitan, surgió la idea de desarrollar un sistema que permita controlar al robot Butiá, utilizando movimientos oculares.

Para este proyecto, saber a qué dirección está mirando una persona es una tarea complicada, hay muchas maneras de hacerlo, y en este caso, decidimos embarcarnos en una aventura en el mundo de la inteligencia artificial con el objetivo de poder desarrollar una herramienta multiuso de código abierto que resuelva el problema.

Esta herramienta podrá dar lugar a muchos futuros proyectos, ya que podríamos pensar que este problema es análogo a controlar una silla de ruedas mediante los ojos, para personas con reducciones motores severas, o controlar cualquier otro sistema programable.

Para fines demostrativos utilizaremos esta herramienta para mover al **robot Butiá**.

Introducción a la Inteligencia Artificial

Seguro que muchas veces nos preguntamos *¿Las máquinas piensan?*, esta idea de máquinas artificiales capaces de "pensar" surge de hace muchos años y da lugar al término Inteligencia Artificial o IA, *y* hace referencia a cualquier software que puede replicar capacidades humanas, para tareas específicas o más amplias, este comportamiento nos hace creer que las máquinas pueden pensar aunque no tengan neuronas físicas como los humanos.

Este concepto, aunque muchas veces es difícil de detectar, aparece muchas veces en las aplicaciones que utilizamos día a día, como por ejemplo, cuando queremos ver una película, seguro que la aplicación nos recomienda películas similares a las que nos gusta, es decir, aprendió nuestros gustos, o cuando queremos hacernos una foto y aparece un recuadro en nuestra cara, ¿cómo el sistema conoció que ahí hay una cara?.

Cada día, estos sistemas "inteligentes" tienen más impacto en nuestra vida y las máquinas superan al hombre al reconocer patrones en imágenes u comportamientos repetitivos.

Imagen 2: Rostro de una persona. Fuente: DotCSV

Seguro que si vemos esta foto, fácilmente podríamos identificar que hay un rostro de una persona, esto es porque podemos detectar la presencia de algunos elementos que conforman un rostro: ojos, boca, nariz, etc.

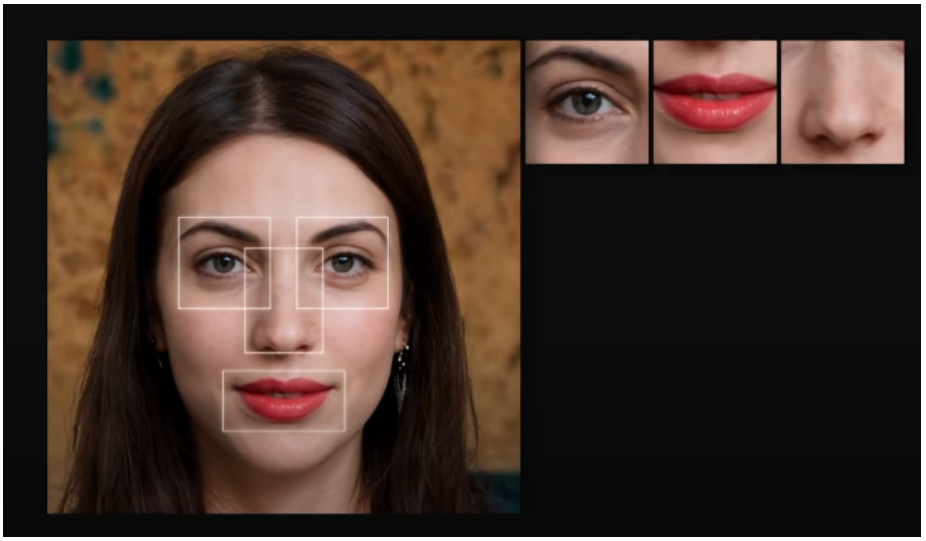


Imagen 3:Detectando elementos del rostro. Fuente: DotCSV

Y repetimos este procedimiento para cada uno de los elementos ¿cómo sabemos qué es un ojo y qué no?. Seguramente nuestro cerebro detectó pupilas, pestañas, superficies blancas, porque somos capaces de detectar estos patrones y texturas.

Pero reproducir el procedimiento que nuestro cerebro realiza sobre las imágenes en una máquina es mucho más difícil, primero tenemos que definir lo que queremos que la IA resuelva, es decir, tener un problema, que puede ser tan complejo como queramos y luego de esto, tener una muestra muy grande de entradas del problema y su respuesta con lo que la máquina podrá aprender.

Para distintos problemas, existen distintos tipos de redes neuronales que lo resuelven, en nuestro caso, cuando el problema es reconocer patrones en imágenes, las redes neuronales convolucionales es la mejor opción.

Red Neuronal Convolucional

Una red neuronal convolucional o CNN está diseñada para replicar todo el proceso que mencionamos anteriormente, es decir, hay un procesamiento por pasos, en las primeras capas se identifican los elementos básicos y generales, y en posteriores capas esto se combina (convolucionando) para detectar patrones más complejos.

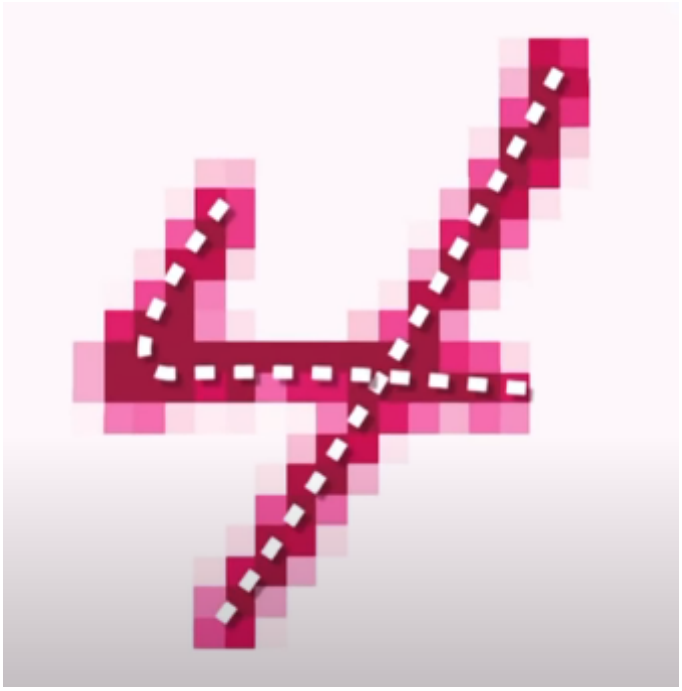


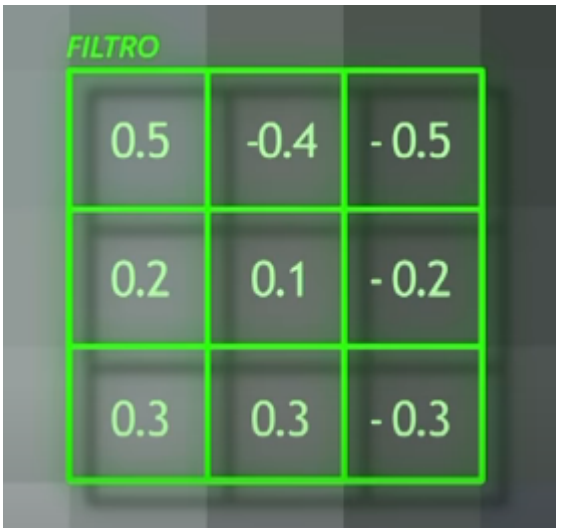
Imagen 4:Detectando patrones en los píxeles de una imágen. Fuente: DotCSV

En la imágen anterior, así como lo haría una CNN, podemos detectar patrones en los píxeles de una imágen, y saber que cada píxel tiene dependencia con sus píxeles vecinos.

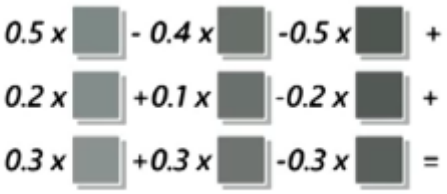
Esta red se caracteriza por aplicar un tipo de capa donde se realizan operaciones matemáticas conocidas como **convoluciones**, una convolución aplicada sobre los píxeles de una imágen, se colocarán filtros sobre la imágen original para obtener un píxel nuevo en base a sus vecinos, ya que, como mencionamos, necesitamos establecer qué dependencia tienen los píxeles con los que tiene a su lado y detectar un patrón.

Para entenderlo un poco mejor, esto es similar a los filtros que aplicamos en nuestras fotos en aplicaciones de edición que utilizamos: desenfoque, colores más vivos, pasar a blanco y negro, etc. Le damos una imágen de entrada, y nos devuelve una imagen con filtros.

Sigue el siguiente procedimiento:



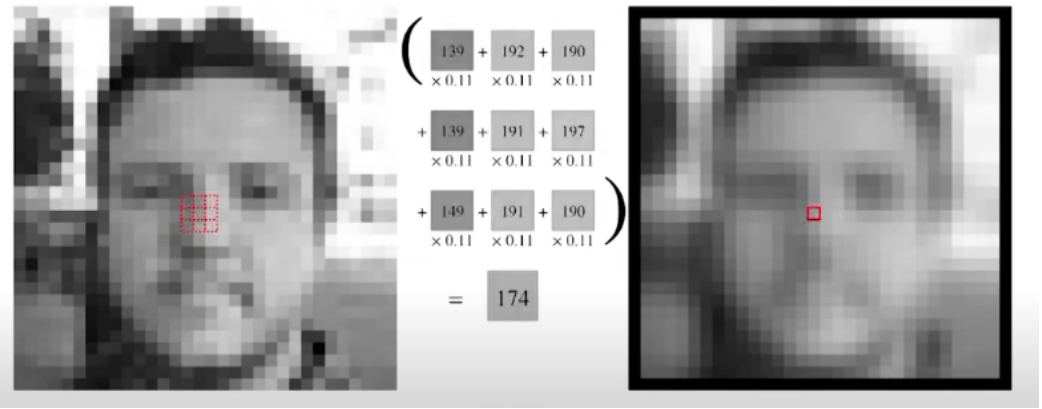
Imágen 5: Aplicando filtro a los píxeles de una imágen. Fuente: DotCSV



Imágen 6:Calculando nuevos pixeles. Fuente: DotCSV

Imágen 7:Resultado de aplicar el filtro. Fuente: DotCSV

Cambiando los valores del filtro, podemos realizar distintos efectos, como el desenfoque:



Imágen 8: Ejemplo de filtro con otros valores. Fuente: DotCSV

Escaneando con distintos filtros, podemos encontrar un filtro que, al hacer los cálculos, el píxel resultante se active (píxel blanco) cuando encuentre el patrón que queremos buscar. Puede detectar cosas diferentes según cuáles sean los valores del filtro que definamos, pero esto no es nuestra tarea, la red automáticamente irá aprendiendo poco a poco para ir haciendo mejor su tarea. Probar distintos filtros y aprenderlos es el trabajo principal de estas redes neuronales convolucionales.

Esto es lo que hace una **capa de convolución** de una CNN, hay una imagen de entrada, aplica filtros, y sale otra imagen. Pero este proceso se puede repetir muchas veces, y que luego de esta capa, la salida de la anterior sea la entrada de la siguiente.

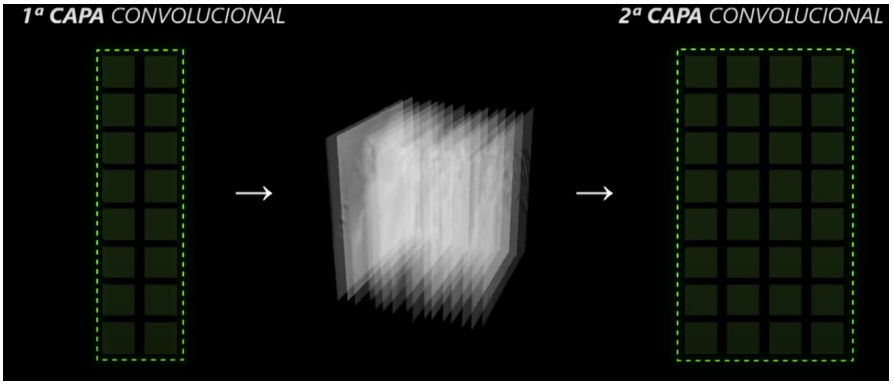


Imagen 9: Capas convolucionales secuenciales. Fuente: DotCSV

Secuencialmente podremos aplicarlo las veces que queramos, pero esto va a depender del problema que nos enfrentemos, en general, mientras más complejo sea, más capas necesitaremos.

Las salidas de cada capa convolucional son lo que llamamos **mapas de características** o de activación (es lo que observamos en la Imágen 9 entre capa y capa), son

todas las imágenes que resultaron de aplicar filtros a la imagen de entrada, la cantidad de filtros puede ser muy grande en cada capa, resultando de muchas imágenes por cada entrada.

Max Pooling

Cada vez la operación de convolución se va haciendo más potente, detectando patrones cada vez más avanzados según la información de la foto. Esto, además, es algo que además podemos incentivar reduciendo cada cierto tiempo la resolución de nuestro mapa de características, estas reducciones son conocidas como **max pooling**.

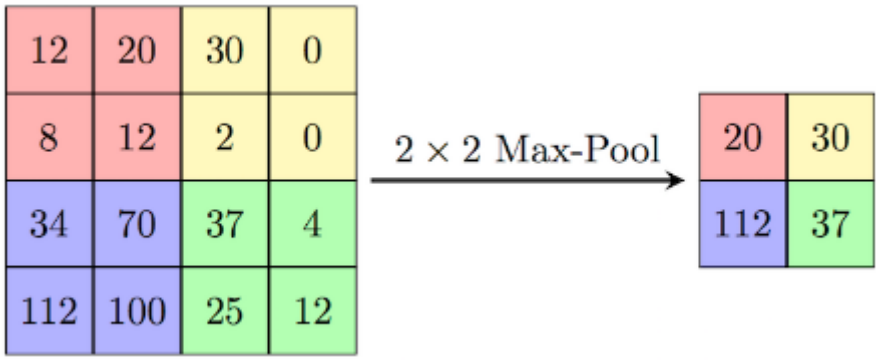


Imagen 10: Aplicando max pooling.

En el ejemplo, aplicando un max pooling de 2x2, observamos que se queda con el valor más alto de ese cuadrado, dando como resultado una imagen más chica.

Entonces, en resumen, las capas convolucionales van haciendo detecciones o activaciones sobre las detecciones anteriores, componiendo cada vez patrones más complejos, teniendo en cuenta esto, podemos observar que si la red es muy compleja necesitará hacer muchos cálculos para entrenar por la cantidad de cuentas que tiene, y en el mayor de los casos se debe tener un buen poder de cómputo para que el entrenamiento sea rápido.

Cuando la imagen pase por todas la capas de convoluciones, si el entrenamiento fue eficaz, va a reconocer los patrones necesarios y ahora sí es posible pasar a la siguiente etapa, meter los resultados como entrada en una red neuronal de capas densas que acabará por tomar la decisión de que es lo que hay en esa imágen.

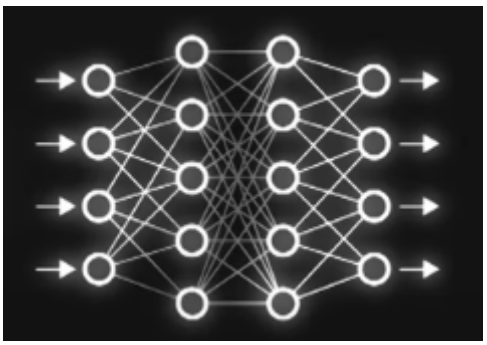


Imagen 11: Capas densas de una red neuronal. Fuente: DotCSV

La entrada de esta etapa será una imágen aplanada, y la capa de salida es lo que detectó.

Si por ejemplo queremos probar un modelo previamente entrenado para que detecte perros y gatos, al mandarle de entrada la imágen de un perro, probablemente la salida de esa red sea “perro”, ya que fue la categoría en la que metió a la imagen.

Durante el entrenamiento esta red de neuronas guarda valores internos, y tiene que ir probando muchas combinaciones para ver cuál de estas combinaciones detecta mejor. Para que la red conozca si está aprendiendo o no también debemos alimentarla con un banco de imágenes que llamamos **imágenes de validación o test** (que generalmente representa un 20% de las imágenes totales, el otro 80% son imágenes de entrenamiento), las cuales ya están categorizadas, la red toma cada una, predice lo que hay y luego compara con la categoría real, repite este procedimiento hasta tener el mayor porcentaje de aciertos que pueda o tantas **epochs** como se le pida, este parámetro define el número de veces que la red va a ver las imágenes de entrenamiento en su totalidad. Pero en un modelo ya entrenado, estos valores ya están calculados y nuestro único trabajo será darle una imágen de entrada, y su respuesta probablemente sea una buena detección si la entrada fue algo que la red conoce, pero puede fallar, y esto simplemente depende del buen o mal entrenamiento que tuvo la red. Si la red solo conoce caras, al colocarle una imágen de un perro, no sabrá que hacer y probablemente reconozca una cara donde no la hay.

Matriz de Confusión

Durante la fase de test también puede pasar que al darle una imágen con una cara, el modelo no la detecta, es decir, el modelo se confunde, para esto podemos construir una **matriz de confusión**, una herramienta que permite la visualización del desempeño de un modelo.

Para entender la semántica de esta matriz, observemos la matriz de ejemplo que aparece a continuación: de 8 gatos reales, el sistema predijo que 5 eran gatos y 3 eran perros; y de 6 perros, el sistema predijo que uno era un conejo y dos eran gatos. A partir de la matriz se puede ver que el sistema tiene problemas distinguiendo entre perros y gatos, pero que puede distinguir razonablemente bien entre conejos y otros animales.

		Valor Predicho		
		Gato	**Perro**	**Conejo**
Valor Real	**Gato**	5	3	0
	Perro	2	3	1
	Conejo	0	2	11

Imagen 12: Ejemplo de matriz de confusión. Fuente: Wikipedia

Arquitectura del sistema

Como mencionamos, queremos detectar la orientación de los ojos y con eso tomar una decisión sobre el Robot, para esto, decidimos dividir el problema en partes:

1 - Cómo detectar la cara.
2 - Cómo detectar los ojos.
3 - Detectar la orientación de los ojos.
4 - Comandar el robot según la orientación.

Para cumplir con la parte 1, 2 y 3, necesitaremos obtener una red que detecta rostros, luego otra para que detecte ojos y luego otra para detectar la posición a la que está mirando, cada una tiene una tarea distinta. Pero esto implica una tarea muy difícil y puede llevar mucho tiempo, por eso utilizaremos modelos ya entrenados. A continuación un diagrama que explica esto con mayor claridad:

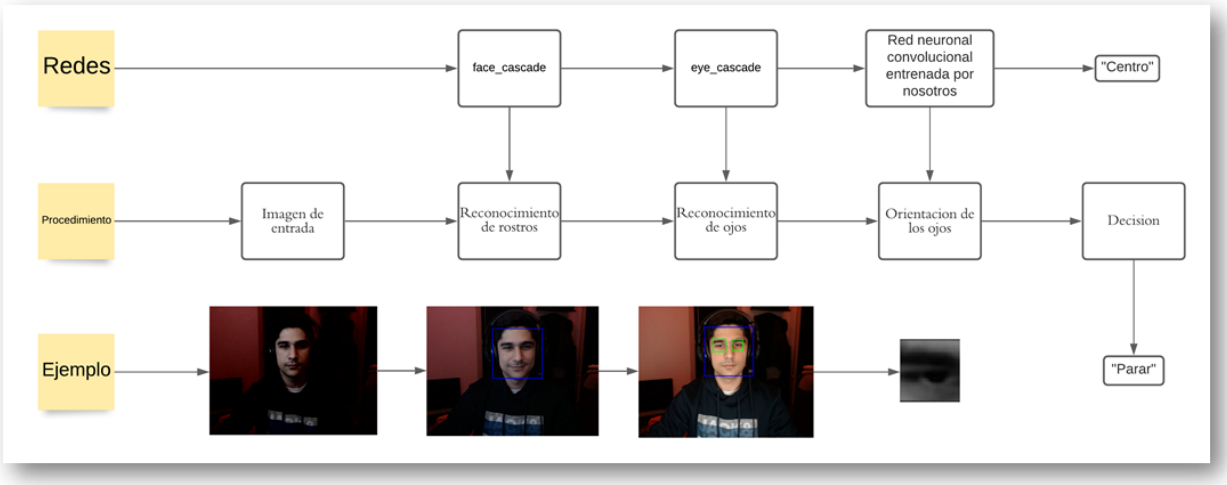


Imagen 13: Arquitectura del Sistema

Dónde *face cascad*e es la primera red, la cual recibe una imagen y extrae el rostro de ella, luego se la envía a la segunda red (*eye cascade*), especialista en detectar ojos.[1] A continuación, esta red le pasa los ojos con un tamaño estandarizado de 128x128 a nuestra red (ya que sólo fue entrenada con datos de este tamaño) con el fin de detectar en qué dirección se está mirando. Finalmente se toma la decisión de en qué dirección mover el robot Butiá.
Para utilizar la tercera red, que fue entrenada para este proyecto, no es necesario volver a pasar por el proceso de entrenamiento, como mencionamos en la parte teórica, un modelo ya entrenado guarda los valores, es por esto que dejamos nuestro modelo entrenado a disposición para todos, lo único que necesitaremos es cargarlos en nuestro programa y darle imágenes de entradas acordes a lo que fue entrenada.

[1] Extraídas de la librería de código abierto OpenCV

Arquitectura de nuestra red

A continuación un diagrama que explicita la arquitectura que utilizamos en nuestra red, entrenada para detectar posicionamientos de ojos, la cual consta de las siguientes capas:

- ➔ Convolución de 32 filtros de un tamaño de 16x16.
- ➔ Max Pooling de tamaño 4x4.
- ➔ Convolución de 16 filtros de un tamaño de 8x8.
- ➔ Max Pooling de tamaño 4x4.
- ➔ Aplanado de imágen a un tamaño de 400.
- ➔ Dropout de 256 neuronas.
- ➔ Capa densa de 256 neuronas.
- ➔ Capa final o capa de salida con las 4 categorías que queremos detectar.

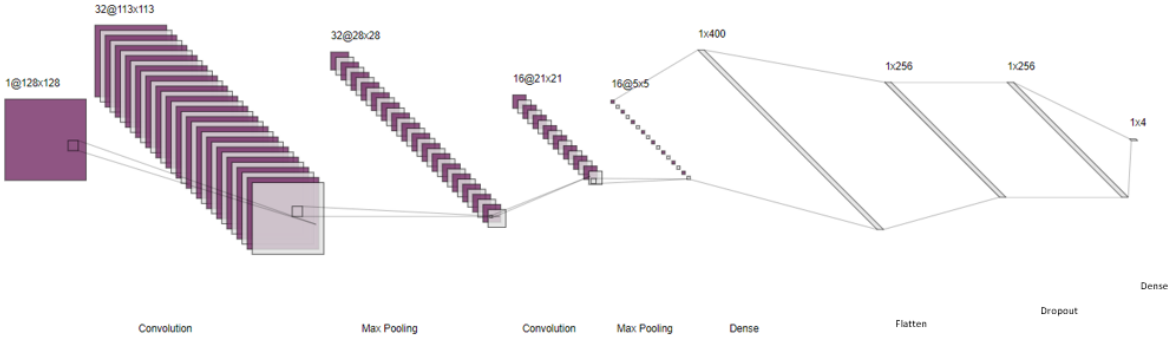


Imagen 13: Arquitectura de la red

Recolección de datos

Para entrenar CNN´s hacen falta grandes volúmenes de datos, por lo cual decidimos pedir la ayuda de 20 voluntarios, los cuales se grabaron durante 10 segundos mirando en cada una de las siguientes direcciones:

- ➔ Centro
- ➔ Derecha
- ➔ Izquierda
- ➔ Arriba
- ➔ Abajo
- ➔ Cerrados

Cada uno de estos videos fue procesado en las dos primeras capas de nuestra arquitectura, es decir la capa de detección facial y la de detección de los ojos, estos ojos fueron guardados en carpetas, luego se procedió a la eliminación de imagenes mal formadas o que no fueran ojos[2], luego se los clasificó según la dirección en la que se miraba.

[2] Pues los modelos de detección facial y detección de ojos no son perfectos.

Fueron extraídas, procesadas y clasificadas: **14.610** imágenes en total, de las cuales **11.775** imágenes fueron para el set de entrenamiento (80% del total), y **2.835** imágenes para el set de validación. (20% del total). Cada una de estas imágenes tiene una dimensión de 128x128 pixeles y está en escala de grises.

Entrenamiento

Para la etapa de entrenamiento es posible modificar diversos valores que harán que el modelo final se comporte distinto los cuales son la cantidad de épocas o la cantidad de imágenes, entre otros. Dentro de los parámetros a considerar también están los parámetros de cada capa en sí: tamaño de los filtros, cantidad, entre otros que fueron nombrados anteriormente.

Tras muchos modelos entrenados con muchos parámetros distintos y diferentes cantidades de imágenes y tamaños de las mismas, los parámetros que arrojaron el mejor resultado fueron los descritos en la sección de arquitectura de nuestra red, con 10 épocas.

Para el entrenamiento de nuestro sistema, pues nuestras computadoras no son tan potentes, decidimos utilizar *Google Colab*[3] pues acelera el proceso de entrenamiento exponencialmente, tomándole menos de un minuto por epoch.

Se deja a disposición todo el código utilizado para el entrenamiento en el repositorio del proyecto[4].

Los primeros modelos entrenados no parecían tener buenos resultados, por lo cual, empezamos un análisis para obtener los mejores resultados. Uno de los cambios que resultaron más importantes en este proceso fue la **eliminación** de las categorías **arriba y abajo**, ya que estas podían presentar confusiones para el modelo y bajar mucho la precisión del modelo resultante.

Como podemos observar en el ejemplo de abajo, a veces incluso para un humano es difícil distinguir estos casos.

Imagen 14: Imagen extraída de nuestro banco de imágenes,
la primera foto es una mirando hacia abajo y la otra de un ojo cerrado.

[3] Google Colab es una plataforma de Google que permite correr Júpiter Notebooks en computadoras en la nube, las cuales cuentan con hardware especializado para el entrenamiento de inteligencias artificiales. En este caso utilizamos una sesión con GPU, la que nos fue asignada fue una Nvidia Tesla T4

[4] En la sección recursos se deja un link al repositorio donde se encuentra todo el código y modelos.

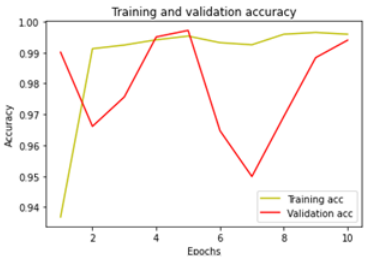
Resultados de la mejor red

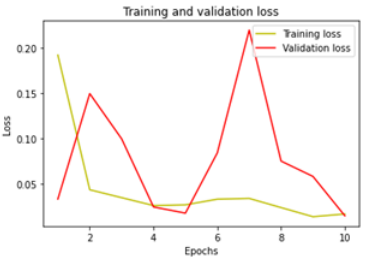
Después de muchos entrenamientos, probando distintos parámetros y analizando resultados, obtuvimos una red con la que estamos satisfechos y que tiene muy buenas predicciones al pasarle una imágen de un ojo en alguna de las direcciones entrenadas.
Para observar qué tan bien se comporta nuestro modelo definitivo, decidimos visualizar su precisión y tasa de aciertos:

Accuracy y loss[5]

Como podemos apreciar en la imagen de la primera imagen, al paso de los epochs, la precisión aumenta a la vez que la pérdida (loss) disminuye, a pesar de los picos en el segundo y séptimo epoch.

Si bien obtuvimos gráficos donde existe una mejor correlación entre la función de pérdida y la precisión, como la segunda imagen, los resultados finales de precisión y pérdida son mejores en la primera, así como los recall[6] por categoría que estudiaremos en la siguiente sección.





En síntesis, obtuvimos una precisión de 99,64% sobre el set de entrenamiento y una precisión de 99,33% sobre el set de validación.

Respecto a la velocidad de muestreo que podemos obtener, en sobre 10.000 muestras, el modelo predice en un tiempo de aproximadamente 32 ms por muestra. Pero si tomamos en cuenta las capas previas del sistema (las capas de las redes provistas por OpenCV), este tiempo asciende a 45 ms en promedio. Lo cual se traduce en una tasa de muestreo de unas 22 muestras por segundo,

[5] El objetivo de una función de pérdida o loss es oficiar de función a minimizar durante el entrenamiento. Para esto existen muchas variantes, en nuestro modelo utilizamos la llamada "categórica cross entropy".

[6] El recall es una métrica que nos permite ver que tan bien se comporta una red. Esta se calcula como la división de verdaderos positivos (aciertos) entre la suma de verdaderos positivos y falsos negativos.

Matriz de confusión

La matriz de confusión para nuestro modelo es la mostrada a continuación.

Imagen 15: Matriz de confusión de nuestro modelo.

Como mencionamos anteriormente, la matriz de confusión nos ayuda a detectar errores en el modelo pues que podemos apreciar con que frecuencia se confunde dos categorías dadas.

Y como podemos apreciar, en nuestro modelo la tasa de acierto no solo es alta sino que la confusión es muy baja, clasifica correctamente más del 98% de las veces.

Mapas de características

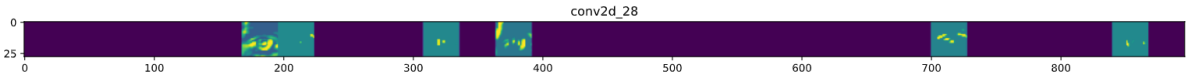
Un aspecto interesante sobre las CNN´s es que podemos ver cuales son los filtros que la red está aplicando, dichos filtros aplicados una imagen de entrada se llaman feature maps, o mapas de características en español, en los que podemos visualizar que es lo que ve nuestra red neuronal, como un intento de interpretar cómo funciona nuestra red.
Consideremos la siguiente imagen de entrada:

Tras ser ajustada a las dimensiones especificadas y ser transformada a blanco y negro:

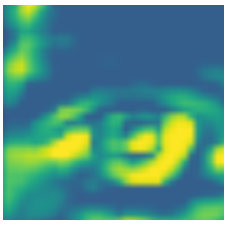
Imagen 16: Imagen extraída de nuestro banco de imágenes.

De la cual para cada capa podemos apreciar los feature maps.

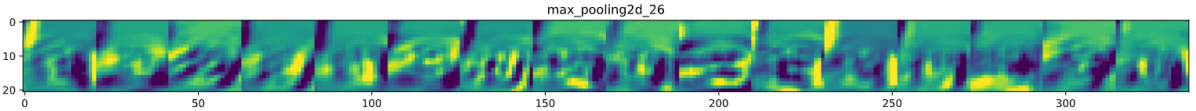
Primera convolución



En la cual podemos observar lo que para nosotros es claro que es un ojo en uno de los filtros aplicados.

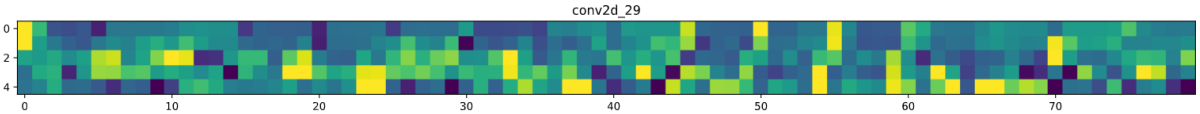


Primer Max Pooling



De la cual el modelo toma las características más importantes para pasar a la siguiente capa.

Segunda convolución



De la que ya no quedan rastros de lo que una vez fue un ojo.

Implementar Sistema Predictor

Para implementar todo el sistema de detección, utilizamos Python 3 sobre una máquina con Windows, la máquina que tendrá la webcam y con ella enviar señales al robot Butiá.

Como mencionamos anteriormente utilizaremos de ayuda algunas librerías para cargar modelos entrenados y procesado de imágenes.

Algunas librerías requieren ser instaladas previamente, estas son:

- ➔ *numpy*
- ➔ *tensorflow*
- ➔ *openCV* (que es cv2)

Para realizar la instalación, debemos abrir una terminal en una máquina con Python instalado e introducir el comando:

pip install <nombre de la librería>

Los demás no necesitan ser instalados ya que son nativos en Python.

Todos los recursos necesarios como los modelos pre-entrenados en formato *.h5* y *.xml* pueden encontrarse en el archivo comprimible adjunto a este documento y deben estar en la misma carpeta del .py que se va a implementar a continuación.

Recomendamos leer los comentarios marcados con el formato (*#<comentario>*) para poder entender paso a paso el procedimiento a seguir, además, en el material audiovisual adjunto construimos paso a paso el predictor.

```
#Lo primero que tenemos que hacer es incluir las librerías que utilizaremos.
import numpy as np
import cv2
from keras.models import load_model
from keras.preprocessing.image import load_img, img_to_array
import statistics as stat

#Cargamos el modulo detector de Rostros.
predictor_caras = cv2.CascadeClassifier('predictor_caras.xml')

#Cargamos el modulo detector de Ojos.
predictor_ojos = cv2.CascadeClassifier('predictor_ojos.xml')

#Cargamos nuestro modelo.
model = load_model('modelo/modelo_horus.h5')

#Declaramos las dimensiones que tendrán las imágenes con las que trataremos
dim = (128,128)
```

```
#Declaramos la función que predice utilizando nuestro modelo
def predict(ojos):
    x = load_img(ojos, target_size=dim)
    x = img_to_array(x)
    x = np.expand_dims(x, axis=0)
    array = model.predict(x)
    result = array[0]
    answer = np.argmax(result)
    return answer
pass

#Creamos un arreglo de tres elementos para almacenar predicciones de 3 fotogramas continuos, esto servirá para minimizar la cantidad de errores producidos por las primeras dos redes, pues son las que introducen un mayor error (por poca luz, etc). Ya que el modelo tiene probabilidad de equivocarse, al tomar la moda en tres muestras ganaremos certeza y "smoothness".
predicciones = [0,0,0]
i = 0

#Cargamos la captura de video, "ret" es una variable que indica que se está recibiendo información de la cámara e img es el fotograma actual
cap = cv2.VideoCapture(0)
ret, img = cap.read()

#Mientras se reciba información
while ret:
   #pasamos a grises la imágen recibida por la cámara
    imagen_en_gris = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

#introducimos el detector de rostros (modelo pre entrenado) y guardamos las coordenadas en la variable faces
    faces = predictor_caras.detectMultiScale(imagen_en_gris, 1.3, 5)

   #para cada una de las caras detectadas (x,y son coordenadas, w ancho y h altura:
    for (x,y,w,h) in faces:

#recortamos la cara de la imágen en gris
        cara = imagen_en_gris[y:y+h, x:x+w]

         #introducimos el detector de ojos(modelo pre entrenado) y guardamos las coordenadas en la variable eyes
        eyes = predictor_ojos.detectMultiScale(cara)

        if len(eyes) == 0:
            print("No se detectan Ojos")
        else:
            #me quedo con un solo ojo
            ex,ey,ew,eh = eyes[0]

             #recortamos los ojos de la cara
```

```
        ojos = cara[ey:ey+ew,ex:ex+eh]

         #reescalamos , mostramos y guardamos la imágen del ojo
        ojos = cv2.resize(ojos, dim, interpolation = cv2.INTER_AREA)
        cv2.imshow('img',ojos)
        cv2.imwrite('ojo.jpg',ojos)
}
         #guardamos en el banco de 3 ultimas predicciones
        predicciones[i] = predict('ojo.jpg')
        i+=1
        if i == 3:
           i=0
         #stat.mean toma la moda (valor que más se repite) en las predicciones de 3 fotogramas continuos.
          print('Prediccion: ',stat.mean(predicciones))

#almaceno siguiente fotograma
 ret, img = cap.read()

#Liberamos la captura de video y destruimos las ventanas de visualización
cap.release()
cv2.destroyAllWindows()
```

Todo el código anterior es posible encontrarlo en los archivos adjuntos. (**predictorWebcam.ipynb**)

Comunicación con el Butiá

Para la comunicación con el robot, la librería de Pybot incluida en el plugin de Butiá, nos permite levantar un servidor desde el robot utilizando Python, servidor que nos conectaremos vía Wi-Fi a través de la máquina cliente, que es la que tiene la webcam y envía señales de dirección en función de lo que el módulo de predicción haya retornado.

Para levantar el servidor desde una consola, debemos estar en la carpeta pybot (en la máquina robot) y ejecutar el comando

```
python pybot_server.py DEBUG
```

La bandera DEBUG es para ver la salida de lado del servidor (o Butiá).

Cuando se levanta el servidor, podemos conectarnos vía Telnet, una herramienta que nos permite conectarnos a otra máquina y controlarla. Para esto, Python incluye nativamente comandos que nos ayudarán a conectarnos al servidor desde la máquina con la webcam. Esto funciona tanto en Windows como Linux, utilizando Python 3.
En el código hay que poner lo siguiente para poder utilizarlo:

```
from telnetLib import Telnet
```

Luego de esto, es posible conectarse mediante la siguiente línea

```
tn =Telnet("<ip>",puerto)
```

La IP se obtiene mediante el comando **ifconfig** en la terminal de Linux de la máquina Butiá donde levantamos el servidor y buscamos IPv4, el **puerto siempre es 2009**.

Luego de esto, puedo mandar comandos de escritura con:

```
tn = tn.write(b'<comando>')
```

Los comandos posibles los podemos encontrar más detalladamente en la documentación oficial de PyBot[7]

Siguiendo lo mencionado anteriormente realizamos algunas modificaciones leves en el código *predictorWebcam.ipynb* implementado anteriormente para que en vez de imprimir señales, envíe controles de dirección al robot Butiá.

[7] En la bibliografía se encuentra un link a la documentación oficial de PyBot.

Puedes enviar los controles que quieras al Butiá, es decir, podemos hacer que al cerrar los ojos me devuelva lo que está leyendo el sensor de distancia o podemos hacer que al cerrar los ojos, el robot frene, esto tiene un abanico grande de posibilidades, todo depende del objetivo del proyecto para lo que quieras utilizar el módulo.

A modo de ejemplo, utilizaremos los 4 comandos que las predicciones nos brindan (centro, cerrado, izquierda y derecha) para mover muy básicamente al Butiá.
Para esto definimos una función de movimiento que envía señales al servidor en base a la dirección pasada por parámetro (0 = cerrado, 1 = centro, 2 = izquierda, 3 = derecha).

```
def mover(tn,direccion):
    if direccion == 2:
        #Izquierda
        tn.write(b'CALL motors setvel2mtr 1 100 0 100')

    elif direccion == 3:
        #Derecha
        tn.write(b'CALL motors setvel2mtr 0 100 1 100')

    elif direccion == 0:
        #Frenar
        tn.write(b'CALL motors setvel2mtr 0 0 0 0')

    elif direccion == 1:
        #Acelerar
        tn.write(b'CALL motors setvel2mtr 0 100 0 100')
pass
```

Y en la sección donde se guarda la predicción, llamamos a esa función:

```
direccion = stat.mean(predicciones)
mover(tn,direccion)
```

*tn es la conexión via Telnet, inicializada anteriormente como tn = Telnet("<ip>",2009)

Estas funcion (**mover**) se encuentra junto a una función **connect** en un módulo llamado **interfazButiaTelnet.py**, y está en los archivos del proyecto.
Para utilizar nuestro módulo simplemente hay que importarlo:

```
import interfazButiaTelnet as Butia
```

Nuestro módulo **interfazButiaTelnet** incluye dos funciones

Butia.connect(HOST,PORT) : Conecta al servidor del Butiá en el host HOST y puerto PORT. (HOST es la ip)

Butia.mover(TN,DIRECCION) : Envía la señal de mover al robot Butiá según la dirección pasada por parámetro, siempre se mueve a una velocidad de 100.

Los controles pueden ser tan avanzados como se quiera, en este caso es solo un ejemplo, es posible definir la función 'mover' de distintas maneras según se ajuste a los objetivos del proyecto que se quiera implementar esta herramienta.

Para la demostración de la implementación de la conexión inalámbrica con Telnet en el predictor también dejamos **material audiovisual**, y además, el código que utilizamos (**predictorWebcamButia.py**).

Agradecimientos

Agradecimientos especiales a las siguientes personas cuya ayuda fue invaluable para la realización de este proyecto.

Sara Demov
Sofia Suarez
Luciano Melonio
Natalia Rossi
Mariella Teran
Paula Soria
Matias Landin
Micaela Rodriguez
Camila Gil
Micaela Gelos
Judith Cruz
Juan Manuel Tassino
Agustin Rivero
Jonathan Rodriguez
Gabriel Gil
Franco Montero
Facundo Diaz
Jose Luis Rodriguez

Recursos y bibliografía

Documentación útil

- https://pythonprogramming.net/haar-cascade-face-eye-detection-python-opencv-tutorial/
- https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_righteye_2splits.xml
- https://www.tensorflow.org/tutorials/images/cnn
- https://pillow.readthedocs.io/en/stable/reference/Image.html
- https://www.tutorialkart.com/opencv/python/opencv-python-resize-image/
- https://www.tensorflow.org/tutorials/keras/classification?hl=es-419
- https://www.tensorflow.org/tutorials/keras/save_and_load

Recursos audiovisuales útiles

- https://www.youtube.com/watch?v=YEZMk1P0-yw&ab_channel=AntoineLam%C3%A9
- https://www.youtube.com/watch?v=kbdbZFT9NQI&ab_channel=Pysource
- https://www.youtube.com/watch?v=sYIYQW03BZ8&ab_channel=KrishNaik
- https://www.youtube.com/watch?v=88HdqNDQsEk&ab_channel=sentdex
- https://www.youtube.com/watch?v=pDXdlXlaCco&ab_channel=NicholasRenotte
- https://www.youtube.com/watch?v=yqkISICHH-U&ab_channel=NicholasRenotte
- https://www.youtube.com/watch?v=7HPwo4wnJeA&ab_channel=codebasics
- https://www.youtube.com/watch?v=j-3vuBynnOE&ab_channel=sentdex
- https://www.youtube.com/watch?v=p3CcfIjycBA&list=PLdyfTrNDH-jZo5d7lqaRf3HNrxQGr1v-o&ab_channel=DigitalSreeni

Notebooks y repositorios auxiliares

- https://colab.research.google.com/drive/1VIhuNYUhFEAFIQl3fKYBSfeZ78dd0neW?authuser=1#scrollTo=5K7yqCjX582S
- https://github.com/codebasics/deep-learning-keras-tf-tutorial/blob/master/16_cnn_cifar10_small_image_classification/cnn_cifar10_dataset.ipynb

Sobre Pooling

https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

Sobre el diagramado de CNN

https://github.com/gwding/draw_convnet/blob/master/draw_convnet.py

Para obtener los feature maps de la red

https://towardsdatascience.com/convolutional-neural-network-feature-map-and-filter-visualization-f75012a5a49c

Documentación oficial de PyBot

https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/PyBot

Repositorio oficial del proyecto

https://github.com/Federico-Gil/Horus