Universidad Zaragoza
1542

Trabajo Fin de Grado

Título del trabajo : Aplicación de algoritmos “Machine Learning” a procesos de localización de objetos.

Autor/es

José Paulo Gonzales Parvina

Director/es

Ana María López Torres

Escuela Universitaria Politécnica de Teruel

2020

Aplicación de algoritmos “Machine Learning” a procesos de localización de objetos

Resumen

En este proyecto vamos a realizar la detección de distintas clases de monedas en una secuencia de vídeo mediante una cámara web. Para ello, vamos a adentrarnos en el tema del aprendizaje automático mediante visión por computador, usando algoritmos basados en las redes neuronales convolucionales (CNN), lo que se conoce como Deep Learning.

Estos son los algoritmos predominantes en la clasificación de imágenes. Para resolver dicho problema, se va a implementar un tipo de aprendizaje supervisado para generar el modelo correcto de predicción del valor de las monedas.

Se ha creado una red neuronal convolucional multicapa conformada por distintas capas y unas capas que contienen técnicas para evitar el sobreajuste,. Por ello, se ha realizado un entrenamiento necesario para optimizar sus parámetros para que se comporte correctamente en el problema de clasificación de monedas.

INDICE

Indice de Ilustraciones

1.Objetivos

El objetivo de este proyecto es aprender y generar el mejor modelo para la predicción de monedas a partir de una buena arquitectura mediante las redes neuronales convolucionales con el lenguaje de programación. Este lenguaje permite a los programadores reducir los tiempos de trabajo y también contiene una gran cantidad de herramientas para la visualización y preprocesamiento de datos. Por ello, la popularidad de las redes neuronales con python va en aumento.

Se hará uso de la API funcional de Keras en Tensorflow, es una librería de código abierto para el aprendizaje automático desarrollada para satisfacer necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones , correlaciones que son similares al aprendizaje y el rozamiento humano. Por lo que, para una clasificación de imágenes es la mas adecuada. También para la detección de objetos y la secuencia de video se hará uso de Opencv ya que nos proporciona herramientas como el análisis y procesamiento de imágenes.

Se hara uso librerías como Matplotlib para la generación de graficas, Numpy para el desarrollo de cálculos matriciales, Pathlib para obtener rutas del sistema de archivos como objetos, PIL que permite la edición de imágenes directamente desde python con una gran variedad de formatos soportados como gif,jpeg,png,jpg,..etc

El sistema estará formado de dos partes, la primera corresponderá al entrenamiento de la red neuronal convolucional donde se creara el modelo y la otra sera una aplicación cuyo funcionamiento sera capturar cada moneda y predecir el valor de dicha moneda mediante el modelo entrenado en una secuencia de video en tiempo real.

2.Introducción

2.1 Machine learning

El machine learning es un subcampo de la inteligencia artificial que crea sistemas que aprenden automáticamente. Aprender en este contexto quiere decir identificar patrones complejos en millones de datos. La maquina que realmente aprende es un algoritmo que revisa los datos y es capaz de predecir comportamientos futuros. También se puede entender como "el campo de estudio que brinda a la computadoras la capacidad de aprender sin estar programado explícitamente" (Arthur Samuel,1959).

Los sistemas de aprendizaje automático se pueden clasificar según la cantidad y el tipo de supervisión que reciben durante el entrenamiento.

Hay tres categorías o tipos principales los cuales son el aprendizaje supervisado, aprendizaje no supervisado y aprendizaje reforzado. Para este proyecto se utilizara el aprendizaje supervisado.

2.2 Aprendizaje supervisado

Este tipo de aprendizaje se puede aplicar a dos tipos de problemas que son regresión y clasificación. Debido a que nuestro proyecto es un problema de clasificación nos centraremos en ello.

Este tipo de aprendizaje consiste en un conjunto de técnicas que permite realizar predicciones futuras basadas en comportamientos o características analizadas a través de ejemplos etiquetados (la etiqueta es la clasificación real a la que corresponde un elemento por ejemplo una moneda de un euro corresponderá a la etiqueta de un euro y así cada una a su categoría correspondiente). Para ello necesitaremos un set de datos o conjunto de datos para el entrenamiento que serán las muestras con las que enseñaremos a nuestro modelo y otro conjunto de validación para validar el entrenamiento. Con estos datos, que son distintos a los del entrenamiento, podemos observar si nuestro esta aprendiendo realmente la estructura de nuestro problema o

tan solo esta memorizando los datos aprendidos en el entrenamiento. Por lo tanto debemos tener cuidado con este aspecto.

En este tipo de aprendizaje la predicción obtenida es representada por medio de una funcion donde las entradas representan características analizadas y la salida la variable que se quiere predecir.

2.2.1 Funcionamiento

Para comprender el funcionamiento del aprendizaje supervisado se tiene un ejemplo como se puede observar en la Ilustración 1. En los algoritmos de aprendizaje supervisado se genera un modelo predictivo, basado en datos de entrada y salida. La palabra supervisado viene de la idea de tener un conjunto de datos previamente etiquetado y clasificado. Es decir, tener un conjunto de muestra, el cual ya se sabe a que grupo, valor o categoría pertenecen los ejemplos mediante las etiquetas. Con estos grupos de datos que son el conjunto de entrenamiento y conjunto de validación o test, se realiza el ajuste al modelo inicial planteado. Es de esta forma como el algoritmo va aprendiendo a clasificar las muestras de entrada comparando el resultado del modelo, y la etiqueta real de la muestra, realizando las compensaciones respectivas al modelo de acuerdo a cada error en la estimación del resultado.

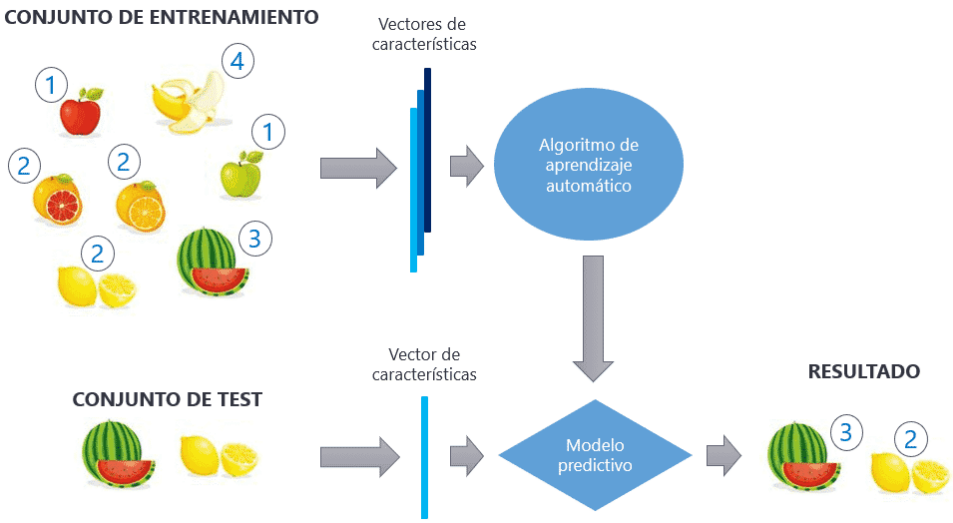


Ilustración 1: Funcionamiento del Aprendizaje Supervisado

2.3 Redes Neuronales

Las redes neuronales es un gran rama de machine learning, conocida como Deep Learning, la cual ha tenido un grandioso impacto en los últimos años en contextos como los chatbot, que cuando platicamos con ellos nos hace pensar que son humanos o los coches que se manejan de forma autónoma. Para que después podamos desarrollar aplicaciones usando redes neuronales. Por lo que, es muy importante conocer su funcionamiento y principales parámetros.

2.3.1 Red Neuronal Biológica

Las redes neuronales están inspiradas en el funcionamiento del cerebro humano. Este tiene millones de neuronas que están conectadas entre ellas en funcion de los estímulos eléctricos que recibe una neurona, decide si se activa o no. Si se activa quiere decir que esta neurona va mandar un impulso eléctrico a las neuronas que están conectadas con ella. En caso contrario no mandara ninguna información. Por lo que, desde este punto de vista funcional, las neuronas constituyen procesadores de información sencillos. Como todo sistema de este tipo, posee un canal de entrada de información, las dendritas; un órgano de computo, el soma y un canal de salida, el axón.(ver Ilustración 2)

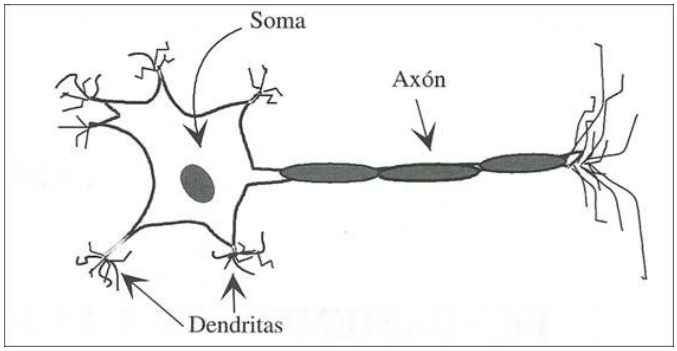


Ilustración 2: Componentes de una neurona biológica

2.3.2 Red Neuronal Artificial

Como hemos comentado antes, una neurona es un procesador elemental que a partir de un vector de entrada procedente del exterior o de otras neuronas, proporciona una única respuesta o salida.

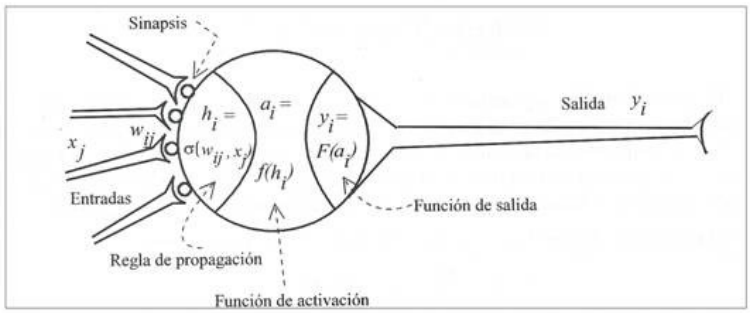


Ilustración 3: Neurona artificial

Los elementos que constituyen la neurona son:

- Entradas Xi(t): Las variables de entrada y salida pueden ser binarias(digitales) o continuas (analógicas) dependiendo del modelo de aplicación.
- Pesos sinápticos Wij : Representan la intensidad de interacción entre cada neurona presinaptica j y la neurona postsináptica i.
- Regla de propagación σ(w, x(t)) : Proporciona el valor del potencial postsináptica, hi(t), de la neurona i en función de sus peso y entradas. Es decir

$$h_i(t)=\sigma(w,x(t)) \quad [1]$$

La función mas habitual de este tipo es lineal, y se basa en una suma ponderada de las entradas con los pesos sinápticos.

$$h_i(t)=\sum_j w_{ij}x_j=w_i{}^T x \quad [2]$$

2.3.3 Funcion De Activación

La función de activación o transferencia proporciona el estado de activación final de la neurona en funcion de su estado anterior, y de su potencial postsináptico actual, aunque en muchos modelos de ANS (Artificial Neural Systems) se considera que el estado actual de una neurona no depende de su estado anterior, si no unicamente

del estado actual. Se buscan funciones que las derivadas sean simples, para minimizar con ello el coste computacional.

La funcion de activación f() se suele considerar determinista, y en la mayor parte de los modelos es monótoma creciente y continua. La forma y=f(x) de las funciones de activación mas empleadas se muestran en la Ilustración 4, donde x representa el potencial postsináptico, y el estado de activación.

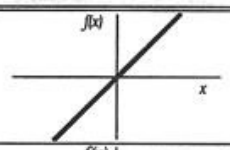
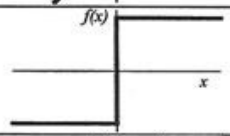
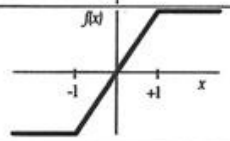
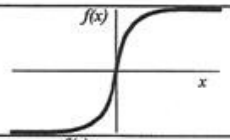
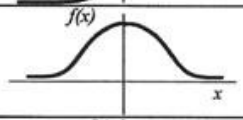
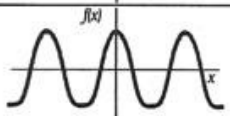
	Función	Rango	Gráfica
Identidad	$y = x$	$[-\infty, +\infty]$	
Escalón	$y = sign(x)$ $y = H(x)$	$\{-1, +1\}$ $\{0, +1\}$	
Lineal a tramos	$y = \begin{cases} -1, & si\ x < -l \\ x, & si\ +l \le x \le -l \\ +1, & si\ x > +l \end{cases}$	$[-1, +1]$	
Sigmoidea	$y = \frac{1}{1+e^{-x}}$ $y = tgh(x)$	$[0, +1]$ $[-1, +1]$	
Gaussiana	$y = Ae^{-Bx^2}$	$[0,+1]$	
Sinusoidal	$y = A\,\mathrm{sen}(\omega x + \varphi)$	$[-1,+1]$	

Ilustración 4: Funciones de activaciones básicas

En la redes neuronales convolucionales se hará uso de funciones de activación como ReLu y softmax.

ReLu - Rectified Lineal Unit

La funcion Relu también es conocida como función rampa transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran. (ver Ilustración 5)

La función viene dada como:

$$f(x) = \max(0, x) = \begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$$

y se define como R(z)=max(0,z)

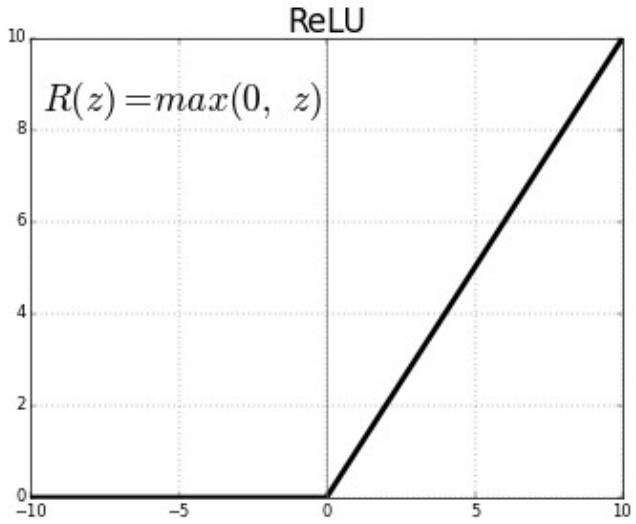


Ilustración 5: Representación de la funcion de activación Relu

Softmax

La funcion softmax transforma las salidas a una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas es 1. Donde Z es el vector de entrada, Zi son los elementos del vector de entrada a la funcion ,K el numero de clases en el clasificador de clases múltiples , y por ultimo el termino inferior de la formula corresponde al termino de nacionalización. Asegura que todos los valores de la salida de la funcion sumen 1.

$$f(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad [3]$$

2.3.4 Estructura De Una Red Neuronal Artificial

Los tres conceptos a emular de los sistemas nerviosos son: paralelismo de calculo, memoria distribuida y adaptabilidad al entorno.

El procesamiento paralelo resulta esencial en este tipo de tareas para poder realizar gran cantidad de calculo en un intervalo de tiempo lo mas reducido posible.
Otro concepto importante que aparece en el cerebro es el de la memoria distribuida. Mientras que en un computador la información ocupa posiciones de memoria bien identificados, en los sistemas neuronales se encuentran distribuida por la sinapsis de la red, de modo que si una sinapsis resulta dañada, no perderemos mas que una parte muy pequeña de la información. El ultimo concepto fundamental es la adaptabilidad. Las redes neuronales artificiales se adaptan fácilmente al entorno modificando su sinapsis, y aprenden de la experiencia, pudiendo generalizar conceptos a partir de casos particulares.

A partir de estos 3 conceptos concluimos en la realización de una red neuronal artificial puede establecerse una estructura jerárquica similar. Por lo que, el elemento esencial de partida sera la neurona artificial, que se organizaran en capas, y varias capas constituirán la red neuronal, y por ultimo una red neuronal que sera el conjunto de todas ellas, junto con las interfaces de entrada y salida. (ver Ilustración 6)

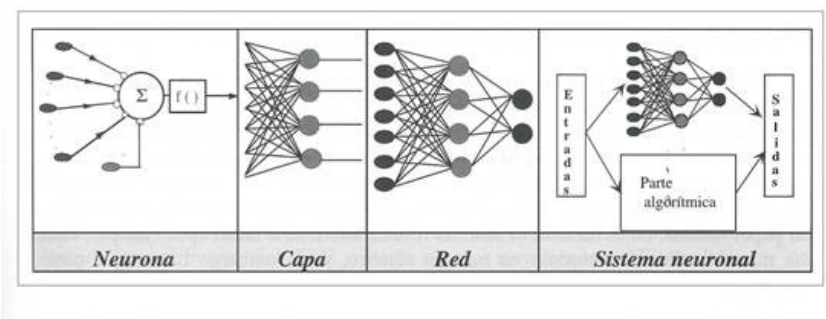


Ilustración 6: Estructura de una red neuronal artificial

La capa de entrada se encargara de recibir todos lo datos de los objetos a clasificar como se puede observar en la ilustración 1. Los objetos son manzanas, plátanos, limones y sandias, para pasar a las siguientes capas ocultas. La funcion de esta

capa es quedarse con las características mas importantes de cada imagen, o elemento que se recibe en la capa de entrada. Por ultimo, estará la capa de salida que se encarga de recibir información de las capa ocultas para poder tomar una decisión y dar un resultado que en este caso seria etiquetar correctamente cada una de la imágenes que estamos pasando por la red.

2.3.5 Tipos De Redes Neuronales

Red Neuronal Monocapa

Se corresponde con la red neuronal mas sencilla ya que se tiene una capa de neuronas que proyectan las entradas a una capa de neuronas de salida donde se realizan los diferentes cálculos.

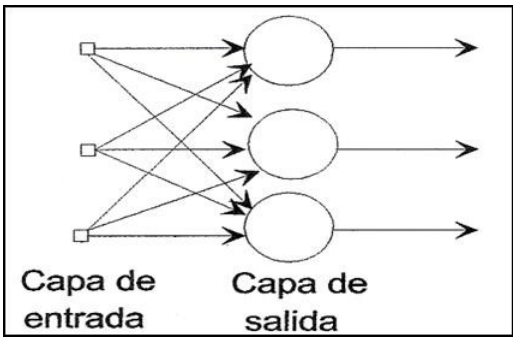


Ilustración 7: Red neuronal monocapa

Red Neuronal Multicapa

Es una generalización de la anterior, existiendo un conjunto de capas intermedias llamadas capas ocultas entre la capa de entrada y la de salida. Este tipo de red puede estar total o parcialmente conectada.

Estas tipo de red cuando todas las neuronas de una capa están conectadas con todas las de la siguiente capa se corresponde con una capa de una red neuronal convolucional como la que usaremos para este trabajo.

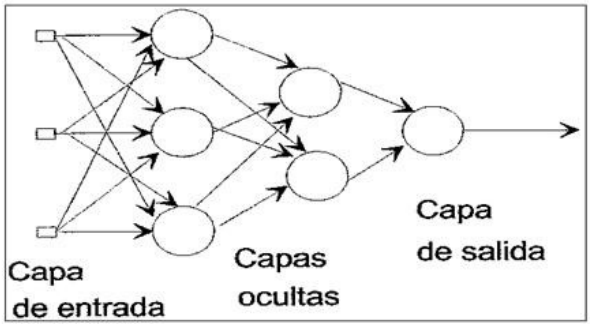


Ilustración 8: Red neuronal multicapa

Red Neuronal Recurrente

Este tipo de red se diferencia de las anteriores en la existencia de lazos de realimentación en la red. Estos lazos pueden ser entre neuronas de diferentes capas, neuronas de la misma capa o, entre una misma neurona Esta estructura la hace especialmente adecuada para estudiar la dinámica de los sistemas no lineales.

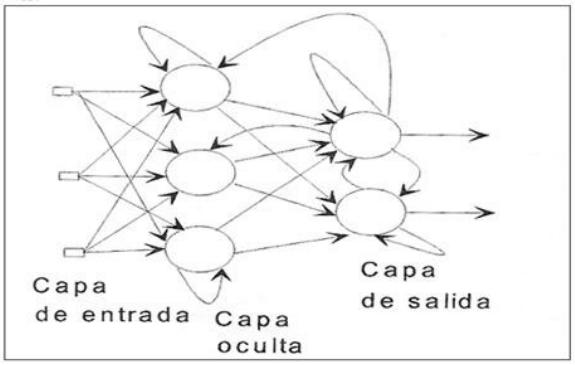


Ilustración 9: Red neuronal recurrente

2.4 Redes Neuronales Convolucionales

Una red neuronal convolucional, también conocida por sus siglas en ingles CNN(Convolution Neural Network) se denomina así por que se basan en el concepto matemático de la convolucion, que es una trasformación lineal sobre dos funciones para producir una tercera. La primera funcion x es comúnmente conocida como la entrada y la funcion w, una funcion de ponderación, como kernel.

La operación de convolucion puede expresarse como:

$$y(t) = g(t) * u(t) = \int_0^t g(t-\tau)u(\tau)d\tau \quad [4]$$

donde $\tau \in R$, $t \in R$, ω: $R \rightarrow R$, x: $R \rightarrow R$. La funcion y: $R \rightarrow R$, también conocida como mapa de característica, se define como la integral de ambas funciones después de que una de ellas haya sido invertida o desplazada un intervalo (τ)

2.5 Convolucion en imágenes

La convolucion en imágenes es el tratamiento de una matriz (matriz de la imagen) por otra que se llama kernel. La convolución se hace en dos dimensiones correspondientes a la altura y la anchura de la imagen. Por lo que, la imagen sera una matriz que dependerá de parámetros como la altura, anchura y el canal. El canal es un termino que se utiliza para referirse a un determinado numero de componentes de una imagen, la imagen a color tiene 3 canales correspondiente a los colores primarios rojo, verde y azul (RGB). Una imagen en escala de grises tiene solo un canal.

El proceso consiste en aplicar un kernel sobre la imagen, que sera una matriz de números donde el patrón y el tamaño definirá como quedara la imagen finalmente. La salida de este proceso sera la imagen filtrada y es lo que en DeepLearning se conoce como mapa de característica (feature map). Se puede observar en en las ilustraciones 10, 11 y 12 la salida de la imagen por diferentes kernels o filtros adaptados al reconocimiento de diferentes mapas de caracteristicas.

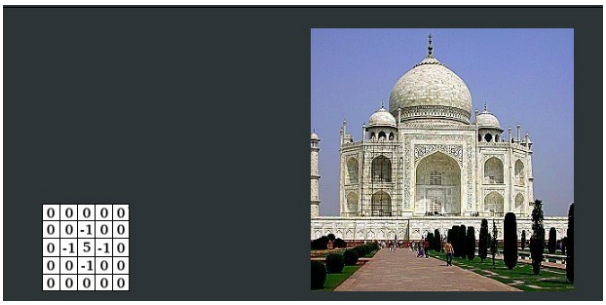


Ilustración 10: Enfoque

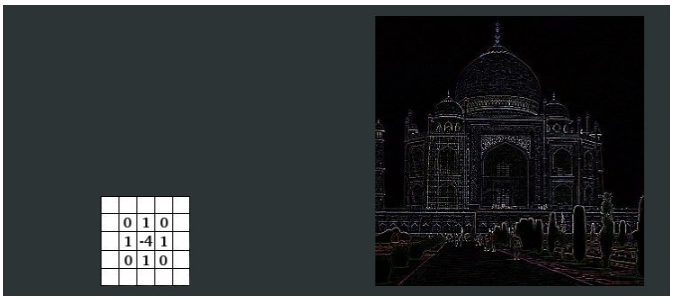


Ilustración 11: Detectar bordes

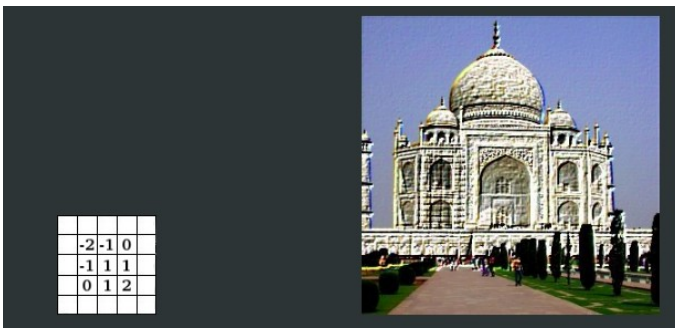


Ilustración 12: Repujado

2.6 Arquitectura de una Red Neuronal Convolucional

Una red neuronal convolucional es una red multicapa que consta de capas convolucionales y de reducción alternadas, y finalmente de una capa clasificadora como se observa en la ilustración 13.

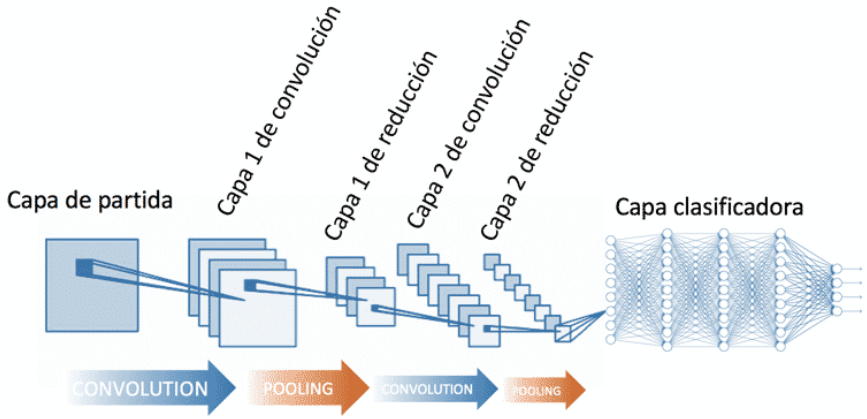


Ilustración 13: Arquitectura CNN

Esta ultima capa esta compuesta por una serie de capas fully connected que proporciona el razonamiento de alto nivel y una capa loss (perdida). Esta capa especifica la desviación entre valores generados por la red y los reales.

2.6.1 Capa De Convolucion

Se realiza la convolucion, con operaciones de productos y sumas entre la capa de partida que es la imagen y varios filtros o kernel que genera diferentes mapas de características (capa convolucionada). La ventaja que tiene una capa de convolucion es que tiene en cuenta la estructura espacial de la entrada, esta característica es lo que hace que en red neuronal convolucional sea mas eficaz que un arquitectura fully connected.

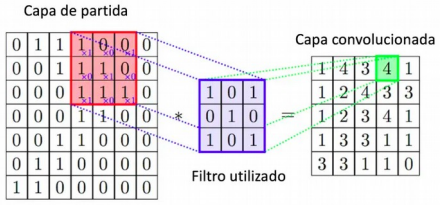


Ilustración 14: Aplicación del kernel

Como se puede ver en la ilustración 14, cuando se realiza la aplicación del kernel la capa de partida se reduce. Para lograr que la capa convolucionada mantenga el mismo tamaño que las dimensiones de la capa de partida se tendrá el proceso de rellenado (padding) y el parámetro stride (saltos de píxel)

Padding

Este proceso consiste en agregar píxeles con valores iguales a 0 a los bordes de la matriz de la imagen (capa de partida). Agregar relleno a una imagen procesada por un CNN (Convolutional Neural Networks) permite un análisis mas preciso de las imágenes. (Ver Ilustración 15)

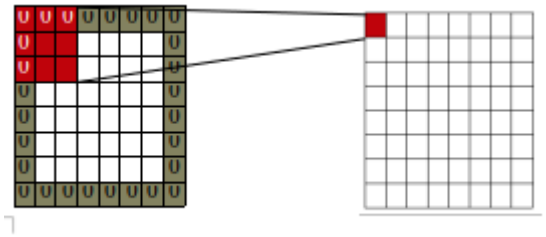


Ilustración 15: Proceso Padding

Stride

Este parámetro indica el paso o salto que hará el kernel en la capa de partida en el proceso de la aplicación del kernel. Cuando este parámetro tiene como valor 1 significa que se recorrerá el kernel por toda la capa de partida píxel por píxel, como se puede observar en la Ilustración 16. Cunado este parámetro sea mayor que 1 significa que el recorrerá el kernel por toda la capa de partida haciendo saltos de pixel según que valor se haya definido como se puede observar en la Ilustración 17 donde se aprecia que el valor es 2. Esto provoca un mayor reducción del la capa convolucionada (mapa de características)

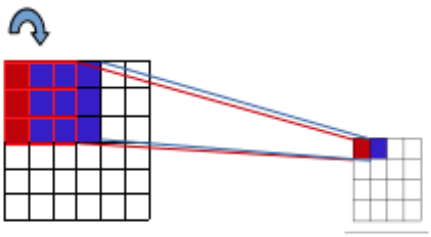


Ilustración 16: Salto píxel a píxel

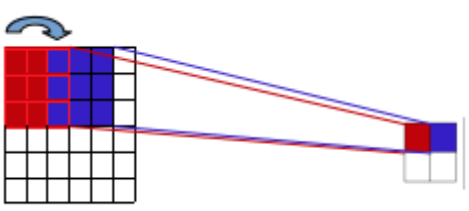


Ilustración 17: Salto de dos píxeles

Después de aplicar la aplicar la convolucion se le aplica a los mapa de características una funcion de activación como se observa en la Ilustración 14.

La funcion de activación mas comunes son ReLu, Sigmoide y Softmax.

2.6.2 Capa Pooling

En la capa Pooling se realiza la reducción de la dimensión espacial de su entrada, donde se disminuye la cantidad de parámetros al quedarse con las características mas comunes.

La forma de reducir los parámetros se realiza mediante la extracción de estadísticas como el promedio o el máximo de una región fija del mapa de características, al reducir estas características el método pierde precisión aunque mejora su compatibilidad. (ver Ilustración 18 y 19)

Ilustración 19: Max7pooling

Ilustración 18: Averagepooling

2.6.3 Capa Clasificadora

Después de un bloque o bloques de convolucion y pooling, y para que una red pueda proporcionar el resultado, es necesario añadir una ultima capa en la que cada píxel se considera como una neurona separada al igual que en una red neuronal artificial formada por múltiples capas. La ultima capa de esta red es una capa clasificadora que tendrá tantas neuronas como el numero de clases a predecir. Dicha capa esta compuesta por una capa Flatten, una capa fully connected y una capa loss.

Flatten

Antes de que la capa fully connected pueda procesar la entrada, es necesario convertirla en un vector de características de una dimensión. Esta operación se llama Flattening. Se coge la salida de la capa de convolucion y se transforma en un vector único.

Fully Connected

Después de las capas de convolucion y de haber transformado su salida mediante flatten, el razonamiento de alto nivel de la red es manejado por una serie de capas fully connected. Sus capas son idénticas a las que podemos encontrar en un perceptrón multicapa.

Loss

Durante el entrenamiento de una red, necesitamos una forma de penalizar la desviación entre el resultado esperado y el resultado predicho por la red. La forma mas común es usar una funcion softmax que tiene la capacidad de convertir sus entradas en una distribución probabilista.

Perceptrón multicapa

El perceptrón multicapa esta compuesto por una capa de entrada, una capa de salida y un numero de capas ocultas entremedias. Se caracteriza por tener salidas disjuntas pero relacionadas entre si, de tal manera que la salida de una neurona es la entrada de la siguiente.

En el perceptrón multicapa se puede diferenciar dos fases:

1. Propagación en la que se calcula el resultado de la salida de la red desde los valores de entrada hacia adelante.
2. Aprendizajes en la que lo errores obtenidos a la salida del perceptrón se va propagando hacia atrás (backpropagation) con el objetivo de modificar los pesos

de las conexiones para que el valor estimado de la red se asemeje cada vez mas al real, esta aproximación se realiza mediante un optimizador de la funcion de coste.

2.7 Backpropagation

Es un método de calculo del gradiente utilizado en algoritmos de aprendizaje supervisado utilizados para entrenar redes neuronales artificiales. Una vez que se ha aplicado un patrón a la entrada de la red como estimulo, este se propaga desde la primera capa a través de las capas siguientes de la red, hasta generar una salida. Esta salida se compara con la salida deseada y se calcula una señal de error para cada una de las salidas.

Las salidas de error se propagan hacia atrás, partiendo de la capa de salida, hacia todas la neuronas de la capa oculta que contribuyen directamente a la salida. Si embargo la neuronas de la capa oculta solo reciben una fracción de la totalidad de la señal de error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original. Este proceso se repite capa por capa hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa correspondiente al error total.

Para el entrenamiento de una red se debe tener en cuenta que la salida de cada neurona no va depender unicamente de las entradas del problema, si no que también depende de las salidas que ofrezcan el resto de neuronas. Por este mismo motivo también podemos afirmar que el error cometido por una neurona no solo dependerá de que sus pesos sean los correctos, si no también dependerá del error acumulado del resto de neuronas que le precedan en la red.

La funcion que controla el error cometido es la siguiente:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in salidas} (t_{kd} - o_{kd})^2 \quad [5]$$

Donde cada parámetro significa lo siguiente:

- Es el vector de pesos
- D es el conjunto de ejemplos de entrenamiento.
- d es un ejemplo de entrenamiento concreto.
- Salidas es el conjunto de neuronas de salida.
- k es una neurona de salida.
- tkd es la salida correcta que debería dar la neurona de salida k al aplicarle a la red el ejemplo de entrenamiento d.
- okd es la salida que calcula la neurona de salida k al aplicarle a la red el ejemplo de entrenamiento d.

2.7 Funcion de coste

La funcion de coste trata de determinar el error entre el valor estimado y el valor real, con el fin de optimizar los parámetros de la red neuronal.

En el apartado de anexos se explicara los diferentes tipos de funciones de coste. En este trabajo utilizaremos la entropía cruzada categórica.

2.7.1 Entropía Cruzada Categórica – Categorical Cross- Entropy

La entropía cruzada categórica, es una medida de precisión para variables categóricas.

Características de entropía cruzada categórica:

- Mas difícil diferenciación y convergencia.
- Escala univariante.
- Simétrica.
- Es mas fácil de interpretar.

$$\text{cross-entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} t_{i,j}\log(p_{i,j}) \qquad [6]$$

Donde N es el numero de muestras, k es el numero de clases, log es el logaritmo natural, $t_{i,j}$ es 1 si la muestra i esta en la clase j y es 0 en caso contrario, $P_{i,j}$ es la probabilidad predicha de que la muestra i esta en la clase j.

2.8 Optimizador de la funcion de coste.

Existen distintos tipos de optimizadores para la funcion de coste. En el anexo se explicara otros tipos de optimizadores. En este trabajo se hará uso del optimizador Adam.

2.8.1 Adam

La estimación del momento adaptativo (Adam) es otro método que calcula las tasas de aprendizaje para cada parámetro. Ademas de almacenar un promedio en declive exponencial de gradientes cuadrados pasados vt como Adadelta y Rmsprop, Adam también mantiene un promedio exponencial decreciente de gradiente pasados mt , similar al impulso. Mientras que el impulso puede verse como una bola que baja por un pendiente, Adam se comporta como una bola pesada con fricción, que por lo tanto prefiere mínimos planos en la superficie de error.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad [7]$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad [8]$$

Donde mt y st son estimaciones del primer y segundo momento de los gradientes respectivamente. Como ambos están sesgados hacia el cero, es necesario corregirlos :

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad [9]$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad [10]$$

Los pesos se actualizan tal como hemos visto en optimizador Adadelta.

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad [11]$$

2.9 Métricas

La importancia de las métricas es comparar el funcionamiento de los diferente modelos, gracias a ello podemos evaluar la calidad del modelo generado.

2.9.1 Matriz de confusión

Una tabla describe el desempeño de un modelo de clasificación en un conjunto de datos de prueba cuyos valores verdaderos son conocidos. Una matriz de confusión es altamente interpretativa y puede ser usada para estimar un numero de otras métricas.

Matriz de confusión de una clasificación binaria:

		predicción	
		0	1
realidad	0	70	10
	1	15	5

		predicción	
		0	1
realidad	0	TN	FP
	1	FN	TP

Ilustración 20: Matriz de confusión

En la Ilustración 20, en la matriz de confusión de la izquierda podéis ver los valores para este ejemplo. En la matriz de confusión de la derecha, los nombres genéricos cuando usamos la nomenclatura inglesa: True Negative(TN), True Positive(TP), False Negative(FN), False Positive(FP).

- Positivo o negativo se refiere a la predicción. Si el modelo predice 1 entonces sera positivo, en caso contrario predice 0 y sera negativo.
- Verdadero o falso se refiere si la predicción es correcta o no.

2.9.2 Exactitud

La exactitud mide le porcentaje de casos que el modelo ha acertado. Esta es una de las métricas mas usadas.

La exactitud se calcula de la siguiente forma:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad [12]$$

2.9.3 Precisión

Con esta métrica se puede medir la calidad del modelo de machine learning en tareas de clasificación.

Se define por la siguiente formula:

$$precision = \frac{TP}{TP + FP} \quad [13]$$

2.9.4 Exhaustividad

Esta métrica va a informar sobre la cantidad que el modelo de machine learning es capaz de identificar.

Se define por la siguiente formula:

$$recall = \frac{TP}{TP + FN} \quad [14]$$

2.9.5 Puntaje de F1

Se utiliza para combinar las medidas de precisión y recall en un solo valor. Esto es practico por que hace mas fácil el poder comparar el rendimiento combinado de la precisión y la exhaustividad entre varias soluciones.
F1 se calcula haciendo la medida armónica entre la precisión y la exhaustividad.

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad [15]$$

2.10 Overfitting y Underfitting

En Machine Learning el objetivo de un buen modelo de aprendizaje automático es generalizar bien los datos de entrenamiento a cualquier dato del dominio del problema. Esto nos permite hacer predicciones en el futuro sobre los datos que el modelo nunca ha visto.

Las principales causas cuando se obtienen malos resultados al entrenar diferentes modelos de Machine Learning son el overfitting y underfitting de los datos que servirán como indicadores de que si nuestro modelo esta aprendiendo o simplemente esta memorizando los datos de entrenamiento.

Es muy común que, al comenzar a entrenar a un modelo, se caiga en el problema de underfitting. Lo que ocurrirá es que no hay suficiente información de entrada. Por lo que se obtiene un modelo demasiado sencillo. Cuando se cae en Overfitting, significa que se esta sobre-entrenando el modelo. Por lo que, sera excesivamente complejo para reproducir exactamente los mismos datos de entrada. (ver Ilustración 21)

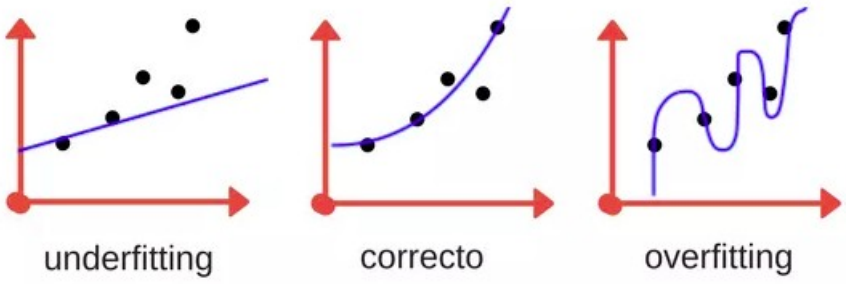


Ilustración 21: Overfitting y Underfitting

Para conseguir un modelo que generalice bien es importante prestar atención a la arquitectura empleada. La cantidad de capas, la elección de capas, el ajuste de los hiperparámetros y el uso de técnicas de prevención de overfitting es esencial, Este proceso se llama regularización existen múltiples técnicas para llevarlo a cabo. Algunas técnicas mas comunes son: L1 y L2, dropout y Earlystopping.

L1, L2 y Elastic Net Regularization

La regularización L1 y L2 son dos métodos muy populares de evitar el efecto de pesos elevados. La L1 consiste en añadir $-\lambda \|w\|$ a la función de perdida. Este método ayuda a dispersar los vectores de pesos. La L2 añade $-\frac{1}{2}\lambda \|w\|^2$ a la funcion de perdida. Es decir, penaliza la magnitud cuadrada de todos los parámetros directamente en la funcion de perdida. Esto ayuda a mitigar los posibles picos en los vectores de los pesos. Estas dos técnicas de regularización se pueden combinar para obtener - $(\lambda \|w\| + \frac{1}{2}\lambda \|w\|^2)$. Esta técnica se conoce como Elastic Net Regularization.

Dropout

Esta técnica difiere de las vistas hasta el momento. El procedimiento es sencillo: por cada nueva entrada a la red en fase de entrenamiento, se desactivara aleatoriamente un porcentaje de las neuronas en cada capa oculta, acorde a una probabilidad de descarte previamente definida. Dicha probabilidad puede ser igual para toda la red, o distinta en cada capa.

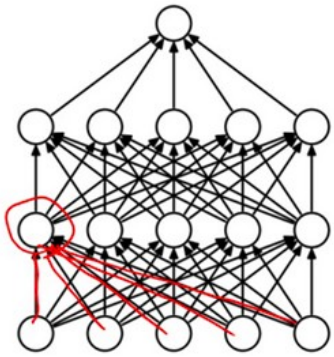


Ilustración 22: Red neuronal sin dropout

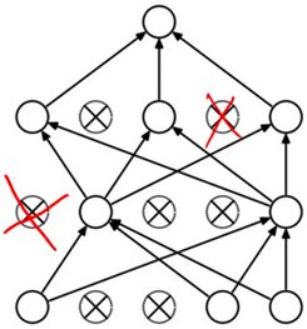


Ilustración 23: Red neuronal con dropout

Earlystopping

Para hacer una detección temprana de los fenómenos de overfitting, ya que como sabemos durante el entrenamiento debemos estar en un punto medio, ya que abusar

del numero de veces que se entrena el modelo puede generar overfitting como también el no establecer un numero correcto de veces que se entrena el modelo, básicamente lo que se hace con esta técnica es una llamada para que se detenga todo el entrenamiento cuando una métrica monitoreada haya dejado de mejorar.

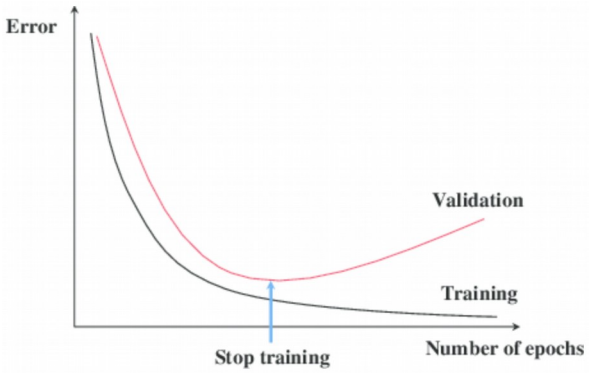


Ilustración 24: Detección temprana de overfitting

ReducirLROnPlateau

Los modelos a menudo se benefician de reducir la tasa de aprendizaje en un factor de 2 a 10 una vez que el aprendizaje se estanca. Esta técnica monitorea una cantidad y si no se observa ninguna mejora en un numero determinado de veces que se entrena el modelo, la tasa de aprendizaje se reduce.

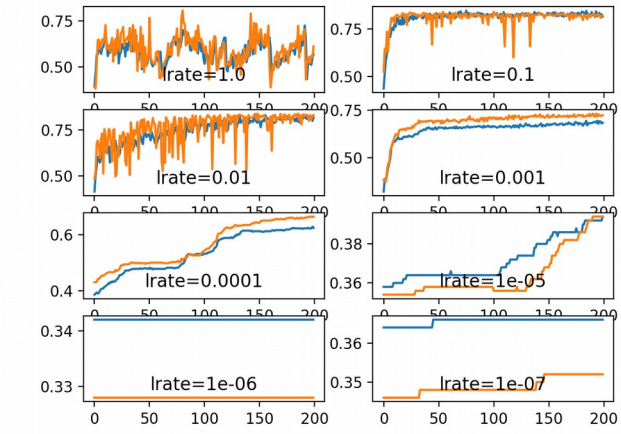


Ilustración 25: Reducción de la tasa de aprendizaje

3.Puesta en marcha

3.1 Entorno hardware y software:

Este punto es importante ya que ha sido de gran dificultad comprender los distintos requerimiento tanto software como hardware para poder comenzar utilizar esta librería para el desarrollo de redes neuronales, por lo que tenemos que tener algunos aspectos en cuenta. Una de las principales dificultades son las elevadas necesidades computacionales durante el proceso de entrenamiento. Por lo que, se necesita de una infraestructura hardware adecuada.

3.1.1 Importancia de usar GPUs

Básicamente, en los ordenadores normales de hoy en día se pueden utilizar dos unidades de procesamiento para realizar cálculos. Por un lado, se encuentran las unidades de procesamiento central (CPUs) y, por otro, las unidades de procesamiento grafico (GPUs).

La diferencia mas importante entre los dos componentes a nivel de arquitectura es el numero de núcleos que suelen tener. Las CPUs tienen muchos menos núcleos que las GPUs, lo que hace que para el procesamiento paralelo sean peores. Sin embargo, los núcleos de las CPUs son mas potentes, por lo que para tareas secuenciales son mejores.

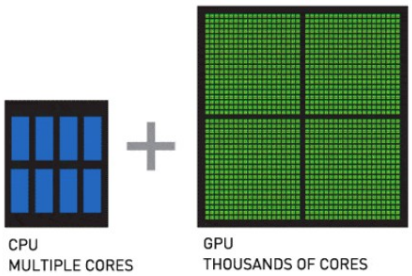


Ilustración 26: Núcleos de GPU y CPU

La potencia de la GPUs se ha ido aprovechando para problemas de calculo matricial. Afortunadamente, este hecho cambio con la llegada de las arquitecturas CUDA, que permitían usar las GPUs para calculo paralelo de propósito general. (ver Ilustración 27)

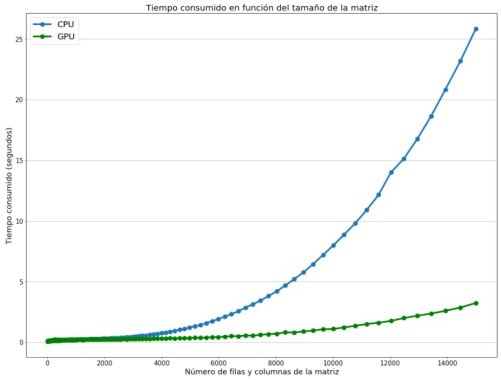


Ilustración 27: Tiempo de consumo en calculo matricial CPU VS GPU

3.1.1 Entorno Virtual

Lo primero que debemos hacer para hacer uso de Tensorflow es utilizar un entorno virtual donde instalaremos los paquetes correspondientes a la instalación de tensorflow debido a que si no aislamos el sistema de instalación de paquetes podríamos tener conflictos de librerías con otras librerías interna del sistema operativo, lo que haría que tensorflow no nos funcione correctamente.

```
$ python3 -m venv --system-site-packages ./venv
```

Ilustración 28: Creación del entorno virtual

3.1.2 Instalación Tensorflow

Este proyecto se ha realizado con la ultima versión de tensorflow que tiene soporte para GPU, en nuestro caso hemos instalado la versión 2.3.0. Para instalar esta versión de tensorflow se necesita el script pip en su versión mas reciente por lo que se recomienda tener pip 19.0 o posteriores por lo que nosotros tenemos instalado el paquete pip con la versión 20.2.4. También se se requiere tener la versión de python entre 3.5 y 3.8 por su compatibilidad con tensorflow 2.2 y posteriores. En nuestro caso estamos trabajando con python 3.6.9 y usamos un sistema operativo que es una distribución de Linux llamada Ubuntu con la versión 18.04.5 LTS con

un kernel 5.4.0, así que debemos tener las versiones como se puede ver en la Ilustración 29, 30 y 31.

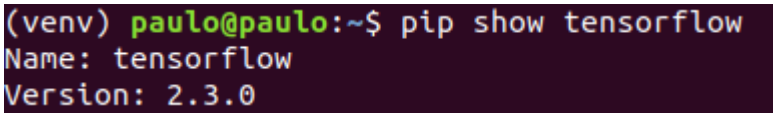


Ilustración 29: Versión de tensorflow

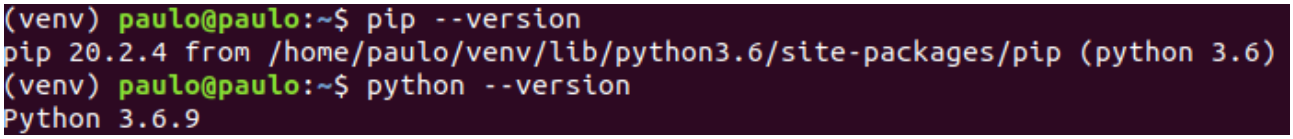


Ilustración 30: Versión de pip y python

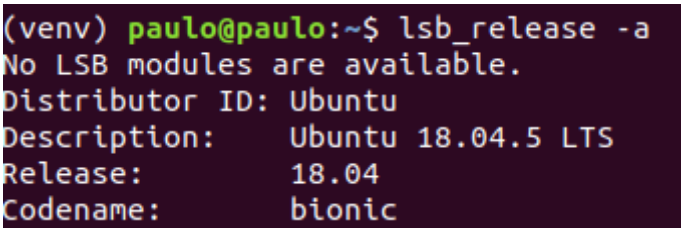


Ilustración 31: Versión de nuestra distribución Linux

3.1.3 Compatibilidad con GPU

Requisitos de software

- Controladores de GPU de NVIDIA: CUDA 10.1 requiere la versión 418.x o posterior.
- Kit de herramientas CUDA TensorFlow es compatible con CUDA® 10.1 (TensorFlow >= 2.1.0)
- CUPTI incluye el Kit de herramientas CUDA®.
- SDK de cuDNN 7.6

Por lo que lo primero que debemos tener es una tarjeta grafica NVIDIA ya que tensorflow trabaja con esas tarjetas graficas. En mi caso tengo una tarjeta grafica NVIDIA Ge-force GTX 950M la cual tiene una capacidad de computo 5.0 pero si no lo tenemos claro podemos consultarlo en la pagina de GPU CUDA y mirar si tenemos una tarjeta grafica compatible como observamos en la Ilustración 32.

Productos GeForce y TITAN compatibles con CUDA

Productos GeForce y TITAN

GPU	Capacidad de cómputo
GeForce RTX 3090	8,6
GeForce RTX 3080	8,6
GeForce RTX 3070	8,6
NVIDIA TITAN RTX	7.5
Geforce RTX 2080 Ti	7.5
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
NVIDIA TITAN V	7.0
NVIDIA TITAN Xp	6.1
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1

Productos portátiles GeForce

GPU	Capacidad de cómputo
Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
GeForce GTX 1080	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1
GeForce GTX 980	5.2
GeForce GTX 980M	5.2
GeForce GTX 970M	5.2
GeForce GTX 965M	5.2
GeForce GTX 960M	5,0
GeForce GTX 950M	5,0

Ilustración 32: Graficas compatibles con CUDA

Como tenemos una tarjeta grafica compatible hemos procedido a seguir con el siguiente paso, se ha descargado el controlador mas reciente desde la pagina https://www.nvidia.com/download/index.aspx?lang=en-us que nos proporciona el fabricante nvidia y hemos procedido a la instalación de CUDA 11 con una versión de controlador r450.66 que corresponde al controlador mas reciente para esta tarjeta grafica.

Antes de realizar esta instalación hemos consultado la compatibilidad con el SDK de CuDNN, información que podemos encontrar en la pagina de tensorflow, y como es compatible con CuDNN 8.0.2 hemos procedido a la instalación. Dicha compatibilidad la encontraremos en un tabla que nos proporciona la documentación de NVIDIA CUDNN documentation como observamos en la Ilustración 33. Debemos tener en cuenta estos aspectos por que si no instalamos una versión de CUDA, compatible con CUDNN y la versión del controlador tendremos que desintalar todo y empezar desde cero ya que suele dar problemas.

1.2. cuDNN 8.0.2 - 8.0.3

Versiones de controlador de hardware NVIDIA, CUDA y CUDA compatibles para las versiones de cuDNN 8.0.2 y 8.0.3.

Tabla 2. Hardware NVIDIA, CUDA y controlador CUDA compatibles

Hardware NVIDIA compatible	Versión CUDA	Capacidad de cómputo CUDA	Versión del controlador CUDA
• GPU A100 / GA100 basada en arquitectura NVIDIA Ampere • Turing • Volta • Pascal • Maxwell • Kepler	CUDA 11.0	SM 3.5 y posterior	r450

Ilustración 33: Tabla de compatibilidad CUDA, controlador, y CP

Con el siguiente comando podremos ver la versión de CUDNN que tenemos instalado en nuestro sistema, tal y como se observa en la Ilustración 34.

```
#define CUDNN_MAJOR 8
#define CUDNN_MINOR 0
#define CUDNN_PATCHLEVEL 2
--
#define CUDNN_VERSION (CUDNN_MAJOR * 1000 + CUDNN_MINOR * 100 + CUDNN_PATCHLEVEL)

#endif /* CUDNN_VERSION_H */
```

Ilustración 34: Versión CUDNN

Con el comando nvidia-smi podremos ver el controlador instalado y la versión de CUDA como podemos observar en la Ilustración 35.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.66       Driver Version: 450.66       CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GeForce GTX 950M    Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   45C    P0    N/A /  N/A |    420MiB /  4046MiB |      5%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1003      G   /usr/lib/xorg/Xorg                184MiB |
|    0   N/A  N/A      1558      G   /usr/bin/gnome-shell              131MiB |
|    0   N/A  N/A      2241      C   ...ffice/program/soffice.bin       28MiB |
|    0   N/A  N/A      2439      G   ...AAAAAAAAA= --shared-files       69MiB |
+-----------------------------------------------------------------------------+
```

Ilustración 35: Versión de CUDA y controlador

4.Descripción del sistema

Para abordar el problema de la localización y detección de monedas en una secuencia de video, se utilizaran la redes neuronales convolucionales, que son las adecuadas para este tipo de problemas: clasificación a partir de imágenes.

En este apartado vamos a describir con detalle el sistema que se ha usado para la resolución del problema que tenemos como objetivo en el proyecto.

4.1 Visión General

Nuestro sistema estará compuesto de las siguiente partes:

- Modelo machine learning
  En este apartado nos centraremos en el entrenamiento y la creación de un buen modelo de la red neuronal convolucional para la clasificación de monedas, incluyendo los resultados de dicho entrenamiento.

- Toma de imágenes Y detección de objetos
  En este apartado se explicara como se ha tomado las imágenes mediante la captura de cada fotograma que se muestra por la cámara, también se explicaran la técnicas usadas mediante rangos de colores HSV, en un fondo blanco y con una luminosidad adecuada y la búsqueda de contornos de cada objeto para su posterior detección.

- Muestra de resultados:
  En este apartado mostraremos el resultado de nuestra aplicación, veremos sobre las imágenes la correcta predicción de la monedas.

4.2 Modelo Machine Learning

En este apartado explicaremos el diseño de la parte del entrenamiento que es independiente del sistema final de detección y predicción. Como sabemos el primer paso para la creación de nuestro sistema, es el entrenamiento de una red neuronal convolucional para la clasificación de 3 clases de monedas que corresponden al valor de 5 céntimos, 50 céntimos y 1 euro. En este apartado nos centraremos en la descripción de la red que vamos a usar en la ejecución de este proyecto ya que como se ha explicado anteriormente, el algoritmo de aprendizaje de este tipo de redes nos permite extraer características de cada clase a partir de un conjunto de entrenamiento. Se modifica los pesos de cada neurona en funcion de una funcion de perdida y sus valores se calculan mediante le método de backpropagation de aprendizaje supervisado. Este método consta de dos etapas principales:

Para cada elemento del conjunto de entrenamiento se calcula a la clase que pertenece según lo valores que tienen los pesos en ese momento dando como buena dicha clasificación mediante una función de error, para luego comparar la clasificación realizada con la clase a la que pertenece realmente. Una vez que se ha obtenida el error cometido, las salidas del error se propagan hacia atrás partiendo de la capa de salida hacia todas la neuronas de la capa oculta que contribuyen directamente a la salida. Este proceso se repite , capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total y así con este proceso repetitivo los pesos se irán actualizando para optimizar la función de error.

4.2.1 Set de datos

El primer paso para nuestro sistema es la elección de un set de datos que contendrá imágenes de las monedas a clasificar. En este proyecto clasificaremos 3 tipos de monedas: las un euro, cincuenta céntimos y cinco céntimos. Para que el modelo pueda generar un modelo correcto y eficaz debemos tener en cuenta que estas imágenes deben estar balanceadas en cantidad. Esto significa que debemos tener cantidades similares de monedas de cada tipo para el entrenamiento.

También debemos tener en cuenta que este set de entrenamiento debe contener imágenes con una variedad de aspecto como:

- Imágenes desde diferentes ángulos.
- Distintos fondos.
- Rotaciones de la imágenes.
- Diferentes contrastes de luz.

Para abordar este proyecto hemos tomado 1500 imágenes de dichos tipos de monedas mencionados anteriormente, por lo que cada tipo contendrá 500 imágenes por moneda. Para cargar la ruta de nuestro directorio de imágenes, Se utiliza la siguiente instrucción para luego generar nuestro set de datos en el siguiente paso como vemos en la Ilustración 36.

```
#ruta del set de datos de imagenes
data_dir = '/home/paulo/Escritorio/tfg/imagenes3'
data_dir = pathlib.Path(data_dir)
```

Ilustración 36: Ruta del set de imágenes

El contenido de la ruta que contiene nuestras imágenes esta distribuido como se observa en la Ilustración 37.

```
Escritorio/tfg/imagenes3
├── cinco
├── cincuenta
└── euro
```

Ilustración 37: Árbol de dependencia del directorio

Representamos 16 imágenes que contiene este directorio en la Ilustración 38.

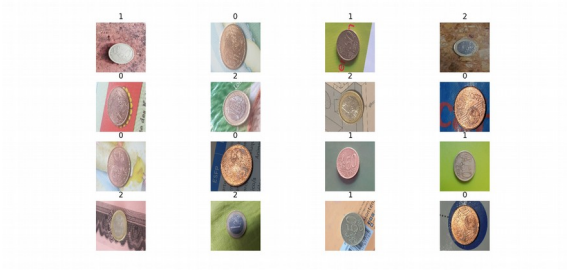


Ilustración 38: Imágenes de las monedas contenidas en los directorios

Una parte importante del set de entrenamiento es que vamos a dividir este set de datos usando el 80% para el entrenamiento y el 20% para el testeo o validación. Este 20% nos permitirá evaluar el desempeño del algoritmo y también nos permite detectar fácilmente problemas de underfitting y overfitting.

Utilizando la API funcional de Keras en Tensorflow mediante una función llamada image_dataset_from_directory() como vemos en la Ilustración 39 y 40, vamos a generar el set de datos con esta distribución desde archivos de imagen del directorio de trabajo. La estructura del directorio se mantiene organizada. A esta función le pasaremos las dimensiones de la imagen, la ruta del directorio, el tamaño del lote de imágenes, semilla de aleatoriedad, y validation_split, ya que esta función devolverá un conjunto de imágenes de los subdirectorios que encuentre en su estructura con su correspondiente etiqueta. El parámetro validation_split lo que hace es indiciar el porcentaje de este conjunto de imágenes que se va usar para el set de validación. En esta función no hemos indicado otro parámetro llamado label_mode que por defecto viene definido por un numero entero lo que significa que las etiquetas están codificadas con números enteros. Las etiquetas correspondientes son para cinco 0, cincuenta 1 y euro 2.

```
#Cargamos nuestro set de datos de entrenamiento
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,   #ruta de las imagenes
  validation_split=0.2, #20% de las imagenes las usaremos para validacion
  subset="training", #entrenamiento
  seed=123,  #Semilla aleatoria opcional para barajar y transformaciones
  image_size=(img_height, img_width), #tamaño de las imagenes
  batch_size=batch_size) #lote de imagenes
```

Ilustración 39: Generación del set de entrenamiento

```
#Cargamos nuestro set de datos de validacion
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

Ilustración 40: Generación del set de validación

Por lo que como resultado se obtiene la separación de datos como se muestra en la Ilustración 41.

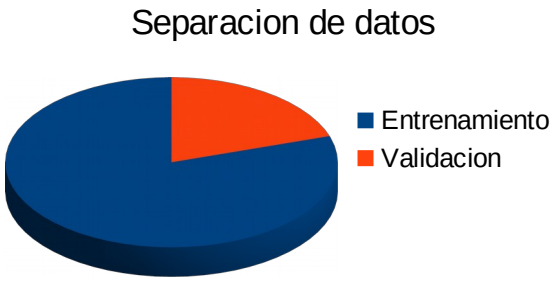


Ilustración 41: Distribución del set de datos

Antes de seguir con el siguiente paso, como dijimos anteriormente nuestro set de entrenamiento debe contener una cantidad suficientes de imágenes para el entrenamiento y con una cierta variedad, por lo que vamos a usar una funcion de la API funcional de Keras en Tensorflow para aumentar estos datos y así evitar el fenómeno de sobreajuste y subajuste como se puede ver en la Ilustración 42. El aumento de datos se refiere a que se generan imágenes nuevas y diferentes a partir del set de entrenamiento mediante transformaciones aleatorias como el zoom, rotación y cambio de contraste.

```
data_augmentation = keras.Sequential(
  [
    layers.experimental.preprocessing.RandomFlip("horizontal", input_shape=(img_height, img_width,3)),
    layers.experimental.preprocessing.RandomRotation(0.4),
    layers.experimental.preprocessing.RandomContrast(0.2),
    layers.experimental.preprocessing.RandomZoom(0.2),
  ]
)
```

Ilustración 42: Aumento de datos

Las imágenes generadas por el aumento de datos se puede ver en la Ilustración 43.

Ilustración 43: Resultado del aumento de datos

4.2.2 Arquitectura CNN

Para el entrenamiento se ha tomando un conjunto de datos de entrenamiento de imágenes mencionados anteriormente, las cuales tienen dimensiones de 150x150x3, ya que debemos tener en cuenta que trabajaremos en este proyecto con imágenes de color por lo que tendremos 3 canales correspondiente a los BGR (Blue,Green,Red).

La arquitectura que se ha tomado para el entrenamiento de esta CNN viene definida por las siguiente capas como se observa en la Ilustración 44:

1. Capa de entrada:
   Esta capa debe ser del mismo tamaño de las dimensión de la imagen, que en nuestro caso dichas dimensiones son de 150x150x3, esta primera capa corresponde a la entrada que es la imagen.

2. Capa Rescaling:
   Esta capa corresponde al re-escalado que le vamos hacer a la imagen, no lo hemos comentado antes pero es una parte importante ya que en vez de trabajar con valores de 0-255, trabajaremos con valores en cada píxel comprendo entre 0-1 ya que no es ideal para una red neuronal trabajar con valores grandes así que debemos intentar que los valores de entrada sean pequeños para reducir la magnitud del calculo computacional.

3. Primera capa convolucional:
   Como mencionamos antes cada capa convolucional consta de ciertos ciertos hiperparametros, de los que dependerá también el volumen o profundidad de la salida que obtengamos en este capa.
   En este caso se ha efectuado una convolucion sobre la imagen con 64 filtros de tamaño 3x3 con un strides=1 por lo cual desplazaremos el filtro píxel a píxel por la imagen y con un padding= “same” esto significa como mencionamos anteriormente que los bordes de la imagen se rellenaran con 0 para obtener una salida después de la convolucion con la misma dimensión, y una función de activación Relu.

4. Primea capa Pooling:

   Se realiza una operación MaxPooling en este caso, con un tamaño de Pooling 2x2, el tamaño de dicha región se ira desplazando quedándose con los valores máximo de cada región recorrida, por lo que reduciremos la imagen a la mitad quedándonos con dichos valores máximos pero su profundidad se mantiene intacta.

   A continuación se repite el mismo proceso, capa de convolucion y maxpooling por lo que solo describiremos sus parámetros ya que es un poco repetitivo.

5. Segunda capa convolucional:

   Aplicamos 128 filtro de tamaño 3x3 con un strides=1, padding = same y con una función de activación Relu.

6. Segunda capa Pooling:

   Aplicamos una capa MaxPooling con un tamaño 2x2.

7. Tercera capa convolucionales:

   Aplicamos 128 filtros de tamaño 3x3 con un strides=1, padding= same y con una función de activación Relu.

8. Tercera capa Pooling:

   Aplicamos una capa MaxPooling con un tamaño 2x2.

9. Cuarta capa convolucional:

   Aplicamos 256 filtros de tamaño 3x3 con un strides=1, padding = same y con una función de activación Relu.

10. Cuarta capa Pooling:

    Aplicamos una capa MaxPooling con un tamaño 2x2.

11. Quinta capa convolucional:

    Aplicamos 512 filtros de tamaño 3x3 con strides=1, padding=same y con una función de activación Relu.

12. Quinta capa Pooling:

    Aplicamos MaxPooling con un tamaño pooling 2x2.

13. Primera capa de abandono:

    Añadimos esta capa Dropout, para eliminar aleatoriamente una cantidad de unidades de salida de la capa durante el proceso de entrenamiento, esto ayuda a que la red neuronal no solo aprenda por un solo camino para llegar al resultado, en nuestro caso eliminamos un 30%.

14. Capa de aplanamiento:

    Esta capa toma los elementos de una matriz de imágenes de entrada en un array plano.

15. Capa Fully Connected (Dense):

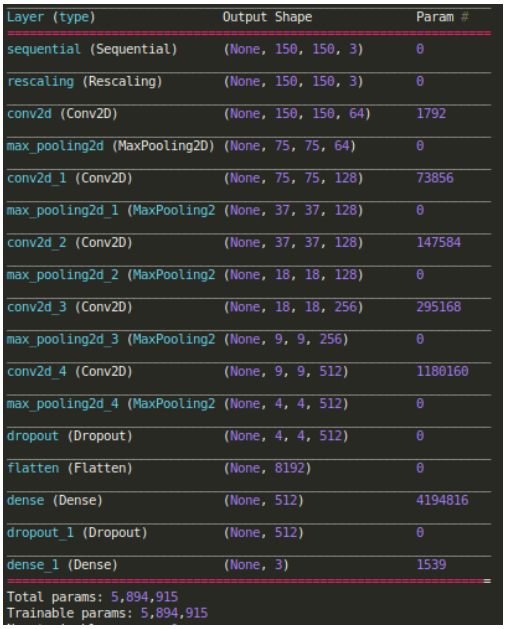
    Vamos a conectar todas las neuronas resultantes de la salida de la capa de aplanamiento a esta para su conectarla a la ultima capa que sera la capa de salida para su posterior clasificación.

16. Segunda capa de abandono:

    Desactivamos el 50% de la capa anterior aleatoriamente para que aprenda por otros caminos al azar.

17. Ultima capa de salida o clasificación:

    Devuelve la categoría en la que se ha clasificado la imagen, ademas de la función de perdida para el método backpropagation mencionado anteriormente, con un función de activación softmax que es utilizada para problemas de clasificación con mas de 2 categorías.



Layer (type)	Output Shape	Param #
sequential (Sequential)	(None, 150, 150, 3)	0
rescaling (Rescaling)	(None, 150, 150, 3)	0
conv2d (Conv2D)	(None, 150, 150, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 75, 75, 64)	0
conv2d_1 (Conv2D)	(None, 75, 75, 128)	73856
max_pooling2d_1 (MaxPooling2	(None, 37, 37, 128)	0
conv2d_2 (Conv2D)	(None, 37, 37, 128)	147584
max_pooling2d_2 (MaxPooling2	(None, 18, 18, 128)	0
conv2d_3 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_3 (MaxPooling2	(None, 9, 9, 256)	0
conv2d_4 (Conv2D)	(None, 9, 9, 512)	1180160
max_pooling2d_4 (MaxPooling2	(None, 4, 4, 512)	0
dropout (Dropout)	(None, 4, 4, 512)	0
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 512)	4194816
dropout_1 (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 3)	1539

Total params: 5,894,915
Trainable params: 5,894,915

Ilustración 44: Red neuronal convolucional

Como podemos ver en la imagen esta arquitectura tiene un total de 5,894,915 parámetro debido al elevado numero de conexiones que tenemos en nuestra red. El numero de estos parámetro se calculan de la siguiente manera:

Primera capa convolucional → (3x3x3+1)x64 =1792
Segunda capa convolucional → (3x3x64+1)x128 =73856
Tercera capa convolucional → (3x3x128+1)x128 = 147584
Cuarta capa convolucional → (3x3x128+1)x256 = 295168
Quinta capa convolucional → (3x3x256+1)x512 = 1180160
Capa Fully Connected (Dense) → (4x4x512+1)x512 =4194816
Ultima capa → (512+1)x3 =1539

En el proceso de entrenamiento es necesario definir una serie de parámetros que calcula su funcionamiento.

4.2.3 Batch Size

Este parámetro es un subconjunto de datos que se entrenara o evaluara en una época, en nuestro caso utilizamos un tamaño de lote igual a 16, también hay que tener en cuenta cuanto mas grande sea este tamaño mas espacio ocupara en memoria.

4.2.4 Épocas

Es el numero entero para entrenar el modelo, lo que significa que una época es una iteración sobre la totalidad de los datos proporcionados para el entrenamiento, el modelo se entrenara como mucho hasta este numero final de épocas, en nuestro entrenamiento tenemos este parámetro definido por un valor de 500 épocas.

4.2.6 Precarga

Se hará una capacitación previa, que se va a utilizar para desacoplar el momento en que se producen los datos del momento en que se consumen los datos, la transformación utiliza un subproceso en segundo plano y un búffer interno para obtener elementos de un conjunto de entrada antes de que se soliciten.

Este valor lo podemos ajustar manualmente o mediante una función tf.data.experimental.ATUTONE() por lo que solicitara el tiempo de ejecución de los tf.data (set de datos) que ajuste el valor dinámicamente durante el tiempo de ejecución, es una forma de mejorar el rendimiento para las operaciones en memoria.

4.2.7 Cache

Debido a esta instrucción la segunda iteración del conjunto de datos se cargara datos de la cache en memoria, lo que conllevara a un ahorro de tiempo si el preprocesamiento de datos es complejo. Debemos tener en cuenta que para grandes conjunto de datos esto puede ser muy pesado para la memoria. En este caso trabajamos con un btachsize de 16 por lo que no se considera un tamaño excesivo.

4.2.8 Shuffle

Baraja las muestras para tener siempre un orden aleatorio de muestras alimentadas a la red. Dentro de este parámetro definimos un batch_size que es el numero de elementos en el búffer aleatorio como vemos en la Ilustración 45. Por lo que, la función llena el búffer y luego muestra aleatoriamente. Se necesita un búffer lo suficientemente grande para barajar correctamente, pero debemos tener en cuenta el

equilibrio con el consumo de memoria. En este caso hemos definido uno 1000 ya que no es excesivamente grande.

```
AUTOTUNE = tf.data.experimental.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Ilustración 45: Shuffle

4.2.9 Configuración del modelo para el entrenamiento

Mediante una función de la API funcional de Keras en Tensorflow llamado compile vamos a configurar los parámetros para el entrenamiento tales como el optimizador a utilizar, la función de perdida probabilista, y las métricas para la evaluación del modelo de entrenamiento.

Se va usar un optimizador Adam que es un método de descenso del gradiente estocástico que se basa en la estimación adaptativa de momentos de primer y segundo orden, por lo que su valor predeterminado de la tasa de aprendizaje es 0.001.

A continuación se puede ver los parámetros predeterminados del optimizador en la Ilustración 46.

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
)
```

Ilustración 46: Parámetros del optimizador Adam

Este optimizador es el ultimo desarrollado por Keras y esta muy orientado para problemas complejos de clasificación de imágenes, es por ello que se hace uso de ella.

La lista de métricas que el modelo evaluara durante el entrenamiento y las pruebas, sera el porcentaje de aciertos que tenemos con el set de entrenamiento y el set de validación, ya que evaluaremos el modelo mediante su precisión.

El propósito del parámetro loss es calcular la cantidad que un modelo debe buscar minimizar durante el entrenamiento. Definiendo este parámetro con la funcion SparseCategoricalCrossentropy() se calcula la perdida de entropía cruzada entre las etiquetas y las predicciones. Esta funcion se usa cuando hay mas de dos clases a clasificar y cuando las etiquetas están codificadas con números enteros.

Por lo que la función Compile estará definida de la siguiente manera como se observa en la Ilustración 47.

```
#Optimizacion del error y metricas para la precision
model.compile(optimizer='Adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Ilustración 47: Funcion compile

4.2.10 Earlystopping

En este funcion, se establecen parámetros como la métrica que se va a monitorear que en este caso sera la perdida y el numero de épocas en la cual no se ha apreciado una mejora. Por lo que, si pasado este numero de épocas y la medida de la métrica no ha mejora durante el entrenamiento se detendrá. Otro parámetro verbose para que nos muestre por pantalla el numero de la época en la que se ha detenido el entrenamiento, el tipo de modo sera “mín” debido a que la métrica monitoreada es la perdida, y un ultimo parámetro llamado restore_best_weights para restaurar los pesos del modelo de la época con el mejor valor de la cantidad monitoreada. (ver Ilustración 48)

```
#vamos a para el numero de iteraciones cuando alcanze un error minimo
early_stop = EarlyStopping(monitor='val_loss', patience=55, verbose=1, mode='min', restore_best_weights=True)
```

Ilustración 48: Detección temprana de overfitting

4.2.11 ReduceLROnPlateau

Con esta función lo que hacemos es reducir la tasa de aprendizaje cuando una métrica haya dejado de mejorar. Es parecida a la anterior función ya que debemos especificar también la métrica a monitorear , seguido de un factor de decrecimiento para la tasa de aprendizaje si esta no ha mejorado en un numero determinado de épocas definido por el parámetro patience y estableciendo el mínimo limite inferior de la tasa de aprendizaje. (ver Ilustración 49)

```
#Reducimos la tasa de aprendizaje si no mejora
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)
```

Ilustración 49: Reducción de la tasa de aprendizaje

4.3 Fase De Entrenamiento

4.3.1 Entrenamiento CNN

Por ultimo vamos a entrenar el modelo para un numero fijo de épocas que serán 500 épocas. Para ello, se hará uso de otra función de la API funcional de Keras en Tensorflow llamada fit() como se observa en la Ilustración 50, por lo que a esta función se le va a pasar el set de entrenamiento generado anteriormente en el apartado Set de datos, también el set de validación para poder evaluar el modelo a medida que se va a entrenar, con un parámetro verobse=2 lo cual significa que se va a mostrar por pantalla el resultado de las métricas de precisión y perdida cuando acabe cada época, y otra función callbacks donde introduciremos las variables que contienen la función Eartlystopping y ReduceLROnPlateau mencionados anteriormente.

```
#entrenamos nuestro modelo
history = model.fit(train_ds, validation_data=val_ds, epochs=epochs, verbose=2, callbacks=[early_stop,reduce_lr])
```

Ilustración 50: Entrenamiento de la CNN

4.3.2 Resultados del entrenamiento

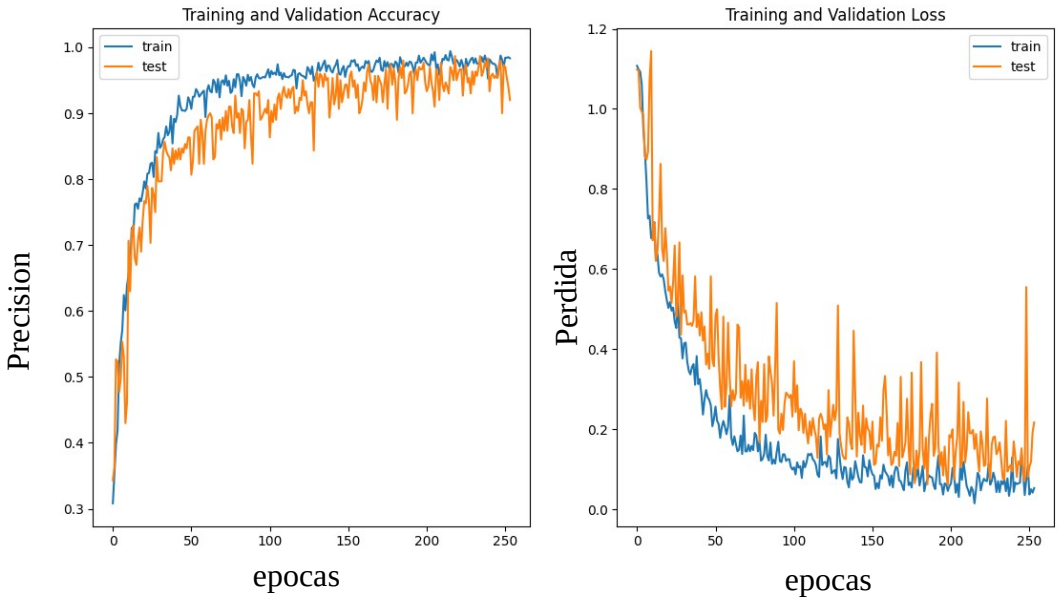


Ilustración 51: Resultado del entrenamiento

En las graficas de la Ilustración 51 se puede apreciar, a la izquierda, como la precisión de entrenamiento y de testeo aumenta un 98% lo que indica que el modelo esta bien encaminado, mientras que en la grafica de perdidas a la derecha se puede ver también que no hay problemas de overfitting ya que perdida en el conjunto de validación se va reduciendo a medida que el error de perdida del entrenamiento va cayendo a valores 0,09% , por lo que el modelo es aceptable para este tipo de problema, aunque también se puede apreciar ciertos picos debido a que algunas imágenes no las estará clasificando adecuadamente. Una mejora a futuro seria aumentar el set de datos a un set de datos mas grande y con mas variedades de imágenes de cada tipo. En este caso solo se ha usado 1500 imágenes por lo que parece que no es suficiente para el modelo, pero en si la arquitectura del modelo es la adecuada.

Como la distribución del set de datos ha sido de 80% y 20%, se tendrá un set de validación o testeo con 300 imágenes, por lo que se vera cuantas imágenes han sido bien clasificadas por el modelo mediante una matriz de confusión como se ve en la Ilustración 52.

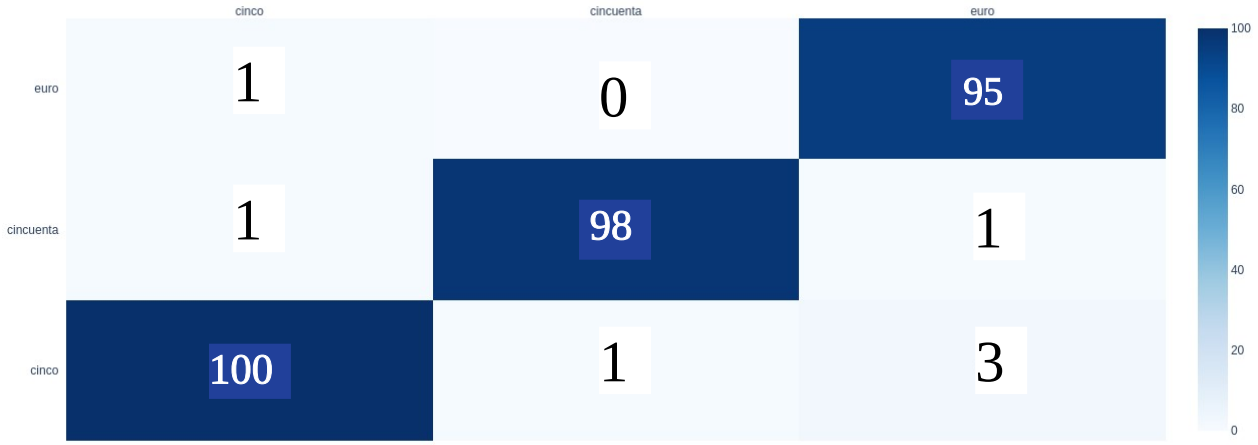


Ilustración 52: Matriz de confusión

Se puede observar que el modelo se comporta muy bien, aunque se ha obtenido algunas monedas mal etiquetadas respecto a otras pero observando la diagonal es bastante claro que el modelo a acertado casi en su totalidad las imágenes de validación respecto contra solo 7 imágenes mal etiquetadas.

Los porcentajes de las diferentes métricas tales como precisión, exhaustividad y puntaje F1, que se usan para su evaluación se observa en la Ilustración 55.

Clasification Report

	precision	recall	f1-score	support
cinco	0.98	0.96	0.97	104
cincuenta	0.99	0.98	0.98	100
euro	0.96	0.99	0.97	96
accuracy			0.98	300
macro avg	0.98	0.98	0.98	300
weighted avg	0.98	0.98	0.98	300

Ilustración 53: Informe de Clasificación

Se puede ver que el acierto es bastante elevado por lo que daremos este modelo como valido. Como se ha mencionado anteriormente uno de los aspectos necesarios para mejorar este modelo es el incremento del set de datos, ya que debido al tiempo no podemos contar con un set de datos mas complejo en cantidad y variedad.

Luego de ver estos resultados guardamos el modelo y pesos de la red neuronal en un formato .h5 para luego cargar el modelo y pesos en el algoritmos de predicción y detección de objetos.

4.4 Aplicación de predicción

Para la toma de imágenes como se observa en la Ilustración 56, se aplicaran ciertas técnicas con ayuda de la librería de OpenCV. El programa consiste en la captura de una serie de imágenes mediante una función cv2.videocapture() que crea un objeto de captura de video desde la cámara desde el que mediante la funcion read() dentro de un bucle infinito se capturan los diferentes fotogramas. Si después se pone la funcion imshow() dentro de este bucle después de cada una de las lecturas las visualizaríamos de manera secuencial. Uno de estos fotogramas se muestra en la Ilustración 54.

Ilustración 54: Toma de imágenes

Estas lineas de código se muestran en la instrucción de la Ilustración 55, 56 y 57.

```
#iniciamos la camara
cam = cv2.VideoCapture(2)
```

Ilustración 56: Código captura de imágenes: creación objeto captura de video

```
#bucle infinito para cada fotograma
while (True):
    ret,frame = cam.read()
```

Ilustración 55: Código captura de imágenes: lectura de un fotograma del objeto de captura de video

```
#mostramos la camara por pantalla cada fotograma
cv2.imshow('camara1',frame)
#cerramos nuestra camara
k = cv2.waitKey(1) & 0xFF
if k==27:
    break
```

Ilustración 57: Código captura de imágenes:visualización del fotograma

Ahora que ya se sabe como capturar cada fotograma que muestra la cámara, el objetivo es separar cada una de las monedas para poder servir de entrada al sistema de clasificación. Para comenzar, se hará una detección de los píxeles correspondientes a una moneda mediante los valores de color HSV (tono, saturación y valor) como se puede observar en la Ilustración 58. Se define el rango de colores que se va utilizar para la segmentación de objetos sobre un fondo blanco.

```
#Rangos de colores HSV
rangomax = np.array([100,255,255])
rangomin = np.array([0,50,20])
```

Ilustración 58: Rangos HSV

Se Realiza una operación de umbralizacion usando la función cv2.inRange() de opencv para generar una mascara binaria como vemos en la Ilustración 59. Un píxel se establece a 255 si se encuentra dentro de los limites de color especificados anteriormente, de lo contrario el píxel se establece a 0. De esta manera devuelve una imagen binarizada apareciendo en blanco los objetos a partir del color que queremos detectar.

```
mascara = cv2.inRange(frame, rangomin, rangomax)
```

Ilustración 59: Creación de la mascara

En el proceso anterior, algunas partes del objeto, que en este caso son las monedas no serán detectadas por que en los colores HSV hay que tener en cuenta no solo el tono de color si no también valores como la luminosidad y la saturación que pueden cambiar de imagen a imagen. Se realizaran operaciones morfológicas para rellenar los posibles huecos y así cerrar el objeto binarizado o eliminar píxeles de ruido sobre fondo.

Como mencionamos anteriormente se realizara operaciones morfológicas con la función cv2.morphologyEx() como se ve en la Ilustración 60, como la apertura que no es mas que una erosión seguida de una dilatación, que es muy útil para eliminar el ruido. Seguidamente de una clausura que es lo contrario ya que no es mas que una dilatación seguida de erosión es útil para eliminar pequeños agujeros dentro del objeto. Este proceso se completara con otra operación de clausura, una dilatación para cerrar huecos de mayor tamaño y por ultimo dos clausuras mas.

```
mascara = cv2.inRange(frame, rangomin, rangomax)

#operaciones moroflogicas
opening = cv2.morphologyEx(mascara, cv2.MORPH_OPEN, kernel)
cv2.imshow('opening',opening)
cv2.imwrite('opening.jpg',opening)
close1= cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel)
cv2.imshow('close1',close1)
cv2.imwrite('close1.jpg',close1)
close2= cv2.morphologyEx(close1, cv2.MORPH_CLOSE, kernel)
cv2.imshow('close2',close2)
cv2.imwrite('close2.jpg',close2)
#dilatamos
dil2 = cv2.dilate(close2,kernel1,iterations = 1)
cv2.imwrite('dil2.jpg',dil2)
#operaciones morfologicas
close3= cv2.morphologyEx(dil2, cv2.MORPH_CLOSE, kernel1)
cv2.imshow('close3',close3)
cv2.imwrite('close3.jpg',close3)
close4= cv2.morphologyEx(close3, cv2.MORPH_CLOSE, kernel1)
cv2.imshow('close4',close4)
cv2.imwrite('close4.jpg',close4)
```

Ilustración 60: Operaciones morfológicas

El progreso de la imagen sera de la siguiente forma como vemos en la Ilustración 61. No es necesario obtener con precisión el tamaño de las monedas, ya que el objetivo final es definir una mascara que permita selección la porción de la imagen correspondiente con cada una de las monedas.

Ç

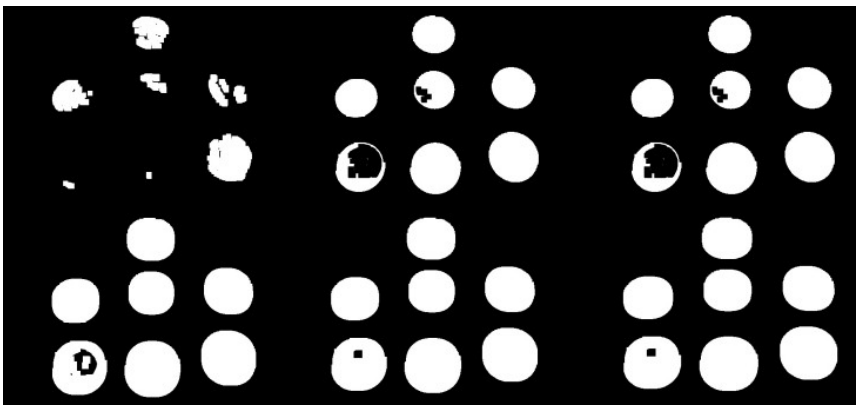


Ilustración 61: Evolución de la imagen binarizada

El siguiente paso es la detección de contornos mediante una función llamada cv2.finContours() como se puede observar en la Ilustración 62, donde pasamos como parámetros la imagen binarizada que hemos obtenido de las operaciones morfológicas. También le indicaremos que solo queremos buscar contornos externos a la función definiendo como parámetro cv2.RETR_EXTERNAL y un método de configuración para la detección cv2.CHAIN_APPROX_SIMPLE que lo que hace es guardar solo los segmentos verticales, horizontales y diagonales dejando solo sus puntos que forman dichos segmentos como por ejemplo si tenemos como objeto un cuadrado, solo tendremos sus 4 puntos de cada vértice.

```
#Buscamos contornos
(contornos,_) = cv2.findContours(close4, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
print("He encontrado {} objetos".format(len(contornos)))
```

Ilustración 62: Búsqueda de contornos

Bueno ahora se va a recorrer todos los contornos encontrados mediante un bucle, calcularemos el área de cada contornos y dejaremos que pasen la áreas mayor que 1200 y menores a 100000. El uso de los contornos se debe a que con la funcion cv2.boundingRect() vamos a obtener el rectángulo limitador devolviendo x, y como la coordenada superior del rectángulo en otras palabras la esquina superior de dicho rectángulo, y w, h que representara la anchura y la altura.

Una vez obtenido estos 4 parámetros, ya podemos pasar el siguiente paso que seria utilizar dicho cuadro limitador para recortar en cada fotograma cada moneda y guardar la imagen con la funcion cv2.imwrite().

A partir de aquí empieza a trabajar el modelo entrenado. Antes de entrar a esta zona del algoritmo debemos cargar el modelo y los pesos guardado en un formato .h5 mediante la funcion load_model() de Keras como se puede observar en la Ilustración 64 . Luego cargamos las imágenes recortadas que sera la moneda a predecir con la funcion load_img(). Después de haber cargado la imagen, se transforma la imagen en una matriz de valores que representa la imagen, seguidamente se añade una dimensión extra con np.expand_dims(), esto es para que se pueda procesar la información sin ningún problema. Luego llamamos a la red entrenada y realizamos la predicción sobre la imagen que le estamos pasando. Esta

funcion predict() nos devolverá una salida de dos dimensiones por ejemplo [[1,0,0]] por eso en el siguiente paso cuando seleccionamos array[0], estamos seleccionando la primera dimensión de esa predicción que de acuerdo al ejemplo seria [1,0,0] y por ultimo la respuesta con la funcion np.argmax() se obtiene como resultado el indice del valor máximo que sera la predicción, por lo que si la predicción era [1,0,0] no dará una respuesta con su correspondiente indice que sera 0 que pertenece a la primera clase. Luego vamos añadiendo dichos valores a una lista y también guardaremos en una lista las posiciones de dichos objetos localizados como vemos en la Ilustración 63.

```
#recorremos cada contorno
for c in contornos:
    area = cv2.contourArea(c)
    if area > 1200 and area < 1000000:
        (x, y, w, h) = cv2.boundingRect(c)
        #deteccion de obejtos
        cv2.circle(frame,(x+w//2,y+h//2),6,(0,0,100),-1)
        cv2.rectangle(frame, (x, y), (x + w + 4, y + h + 4), (0, 255, 0), 3, cv2.LINE_AA)
        #segmentacion de cada objeto
        recorte = frame1[y:y+h+3,x:x+w+3]
        cv2.imwrite("/home/paulo/Escritorio/tfg/cap/"+str(i)+".jpg", recorte)
        #Cargamos cada objeto detectado y iniciamos la prediccion
        foto = load_img('/home/paulo/Escritorio/tfg/cap/'+str(i)+'.jpg', target_size=(longitud, altura))
        foto = img_to_array(foto)
        foto = np.expand_dims(foto, axis=0)
        array = new_model.predict(foto)
        result = array[0]
        #resultado
        answer = np.argmax(result)


        #guardamos las posiciones de cada objeto en una lista
        location1.append(x)
        location2.append(y)
        answers.append(answer)
```

Ilustración 63: Detección y predicción del objeto

```
#caragmos nuestro modelo
longitud, altura = 150, 150
modelo = './graficas_redes_3class/modelo11.h5'
pesos_modelo = './graficas_redes_3class/pesos11.h5'
new_model = load_model(modelo)
new_model.load_weights(pesos_modelo)
```

Ilustración 64: Carga del modelo y los pesos de la red

Y por ultimo, con todas la predicciones y sus posiciones pasamos a escribir la predicción de cada moneda con la función cv2.puxtext() como vemos en la Ilustración

65 que nos sirve para escribir un texto, esa función es simple solo tenemos que pasar parámetros como la posición, el formato fuente, el grosor de la letra, y color.

```
if answer == 0:
    cv2.putText(frame,'cinco', (x,y-5),1,1.5,(0,0,255),2)
if answer == 1:
    cv2.putText(frame,'cincuenta', (x,y-5),1,1.5,(0,0,255),2)
if answer == 2:
    cv2.putText(frame,'euro', (x,y-5),1,1.5,(0,0,255),2)
```

Ilustración 65: Escritura del resultado de la predicción

4.4.1 Visualización de resultados

Como podemos ver el resultado de nuestro sistema de detección y predicción mediante la red que hemos entrenado, los resultados son satisfactorios como observamos en las Ilustraciones 66 y 67.

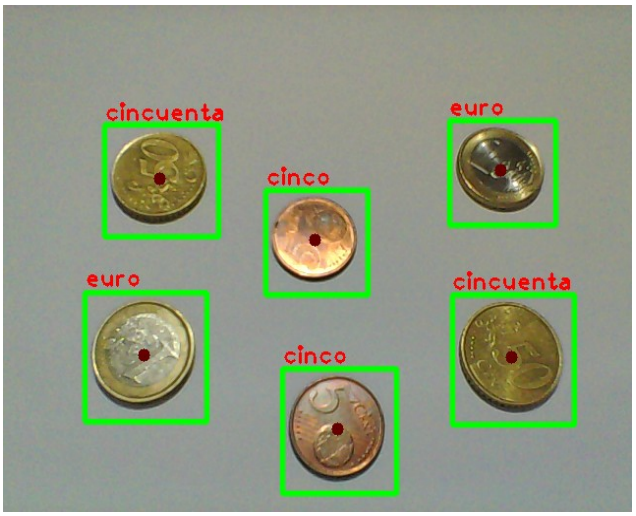


Ilustración 66: Visualización de la predicción

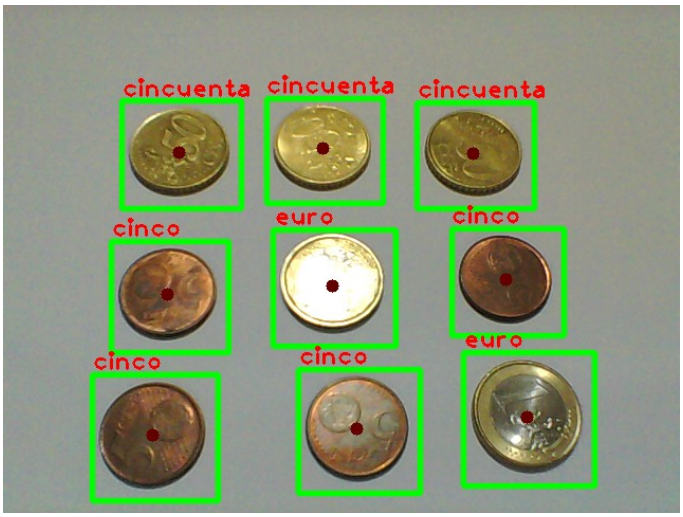


Ilustración 67: Visualización de la predicción

Se puede observar que el resultado es bueno, incluso podemos apreciar que con el reflejo ha logrado predecir correctamente la moneda, aunque no siempre sera así por que como podemos ver casi ni se puede apreciar muy bien los patrones de la moneda. Esta prueba se ha hecho con luz blanca para no tener ningún problema con alguna sombra que pueda causar otra luz.

En alguna prueba se ha observado que mayormente da una predicción correcta, aunque aveces puede predecir erróneamente y esto es debido a que tal vez el set de

entrenamiento que estamos usando no es lo suficiente para abordar este problema con una calidad de modelo al 100% pero sin embargo se puede concluir que el modelo es bastante aceptable (ver Ilustración 68).

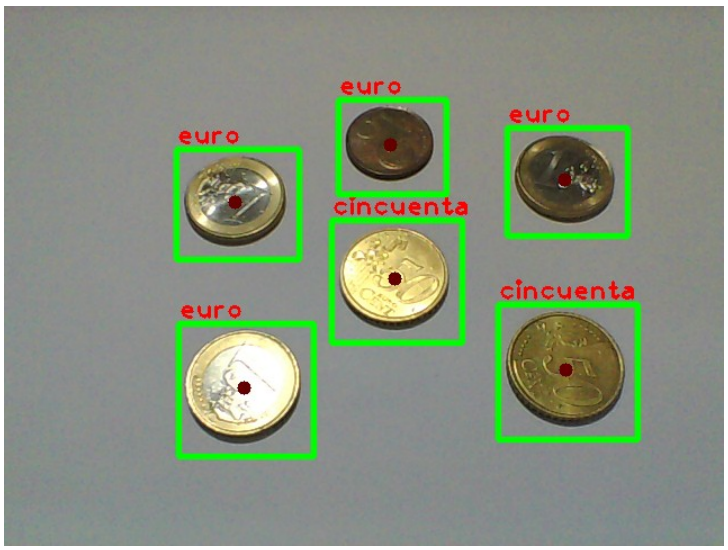


Ilustración 68: Visualización de predicción errónea

5.Conclusiones

En este proyecto hemos conseguido crear un set de datos de imágenes de tres tipos de monedas desde cero para la clasificación de imágenes, la cual hemos usado para el entrenamiento de la CNN resultando un modelo en que los valores de rendimiento y de perdida obtenidos superan los requerimientos mínimos para la predicción de dichos objetos. Podemos concluir que tenemos un modelo de calidad.

Para elaborar este set de datos hemos recopilado en principio 1000 imágenes para cada uno de los 3 tipos de clases de monedas, pero a medida que íbamos experimentado problemas de sobreajuste, hemos tenido que depurar dicho set de datos para eliminar algunas imágenes que parecían repetitivas ya que esto es muy malo para el entrenamiento, ya que es una de los factores por lo que se produce este fenómeno. También se han utilizado otras técnicas para combatir este fenómeno como el dropout, con un optimizador Adam el cual es un optimizador muy bueno proporcionado por Keras en Tensorflow y con un learning rate dinámico si se encontraba un punto de silla.

Por tanto, como conclusiones finales de este trabajo se puede subrayar:

- Se ha obtenido una buena base de datos de imágenes y se empleado técnicas de aumento de datos.

- Se han conseguido competencias necesarias para la utilización de los paquetes Keras y Tensorflow para la creación de una CNN de buen rendimiento para la clasificación de monedas.

- En un principio se intento abarcar el proyecto con los 8 tipos de clases de monedas existentes pero debido a la falta de tiempo y de ejemplares el modelo se tuvo que simplificar, y también hemos obtenido resultado esperados con la función de optimización “Adam”.

- Como posible lineas futuras a mejorar en estos aspectos, se podría proponer la construcción de un modelo con los 8 tipos de monedas. Para ello se podrían realizar un numero de experimentos con los diferentes tipos de optimizadores que existen. También seria necesario recolectar una buena base de datos de los ejemplares para el entrenamiento del modelo, ya que si no tenemos un numero adecuada de muestras en cantidad y variedad, vamos a experimentar problemas de underfitting y por mas que aumentemos los datos si no son distintos seguiremos experimentando este fenómeno.

6.Bibliografía

[1] “CNN – RNA Convolucional by” Numerrentur.
http://numerentur.org/convolucionales/

[2] “Red neuronal convolucional (CNN)” by Tensorflow.
https://www.tensorflow.org/tutorials/images/cnn

[3] “Guía para desarrolladores ” by Keras.
https://keras.io/guides/

[4] “Deep Learning Introducción practica con Keras” by Jordi Torres.
https://torres.ai/deep-learning-inteligencia-artificial-keras/
#Caso_de_estudio_reconocimiento_de_digitos

[5] “Hands-On Machine Learning with Scikit-Learn and Tensorflow” conpecpts, tools, and techniques to build intelligent systems by Aurélien Géron Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

[6] CS231n: Convolutional Neuronal Networks for Visual Recognition of Standford University.
https://cs231n.github.io/

[7]Convolutional Deep Belief Networks on CIFAR-10 by Alex Krizhevsky.
https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf

[8]NVIDIA CUDNN Documentation
https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html

[9]Image Segmentation by Tensorflow
https://www.tensorflow.org/tutorials/images/segmentation

[10]Compatibility with GPU by Tensorflow
https://www.tensorflow.org/install/gpu#linux_setup

[11]Funciones de Coste – Redes neuronales por Diego Calvo.
https://www.diegocalvo.es/funcion-de-coste-redes-neuronales/

[12]Image Thresholnding : documentation Opencv-python
https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html#adaptive-thresholding

[13] Que es overfitting & underfitting y como evitarlos by Enrique Blanco.
https://empresas.blogthinkbig.com/que-es-overfitting-y-como-evitarlo-html-2/

[14] The Frontier of SGD and its Variants in Machine Learning by Juan Du "2017 2nd International Conference on Mechatronics and Information Technology"(ICMIT 2017).

[15] Conceptos basico de Redes Neuronales.
http://grupo.us.es/gtocoma/pid/pid10/RedesNeuronales.htm#modeloneurona

[16]Una descripción general de los algoritmos de optimización del descenso de gradientes by Sebastián Ruder.
https://ruder.io/optimizing-gradient-descent/index.html#adadelta

[17]Convolución by Universidad de Oviedo.
https://www.unioviedo.es/compnum/laboratorios_web/laborat07c_convolucion/lab07c_convolucion.html