UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Aprendizaje Profundo para Verificación Robusta de Identidad en Entornos Móviles

Roberto Rodríguez Rodríguez

Julio, 2019

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Aprendizaje Profundo para Verificación Robusta de Identidad en Entornos Móviles

Autor: Roberto Rodríguez Rodríguez

Tutor: Roberto Rodríguez Echeverría

Co-Tutor/es: Jorge Perianez Pascual

Tribunal Calificador

Presidente: Pedro Clemente

Secretario: José María Conejero

Vocal: Álvaro Prieto Ramos

ÍNDICE GENERAL DE CONTENIDOS

1. ÍNDICE DE TABLAS

2. ÍNDICE DE FIGURAS

3. RESUMEN

En este Trabajo Fin de Grado se realiza el estudio y parte de la implementación de un sistema de verificación de identidad es dispositivos móviles.

Gran parte del estudio se basa en aprender desde cero todos los conceptos necesarios para poder llegar a tener un buen resultado, desde las bases del aprendizaje automático hasta el estudio de arquitecturas de redes neuronales profundas siamesas.

El objetivo del trabajo es la realización de una aplicación en la que mediante fotografías podamos recoger la foto de un documento de identidad de un usuario y posteriormente compararla con una foto tomada por el usuario en el momento, comprobando si son o no la misma persona.

El trabajo está divido en tres partes bien diferenciadas, la parte de estudio en la que se repasan la mayoría de los conceptos aprendidos y necesarios para el entendimiento del resto del trabajo, la parte de extracción facial de documentos de identidad y la parte de correspondencia facial de la foto extraída y una foto tomada por el usuario en el momento.

La implementación se desarrollado hasta donde ha sido posible, primando la calidad de lo que se tenía antes que la cantidad de cosas desarrolladas es por esto por lo que se ha terminado el trabajo con una red neuronal capaz de extraer las fotos de documentos de identidad, dejando como trabajo futuro la implementación de la red siamesa y la aplicación Android, aunque estos procesos han sido estudiados y documentados en este documento.

4. INTRODUCCIÓN

Este trabajo surge de la colaboración con la empresa extremeña Mobbeel, con los cuales he cooperado durante todo el proceso.

A raíz de la necesidad actual que hay en la sociedad de la tecnología, cada vez son más las aplicaciones en las que es necesario comprobar la identidad de un usuario.

Ya sea para abrir una cuenta de banco online, para acceder a una aplicación de casas de apuestas, o para hacer una incorporación en alguna plataforma.

No sólo es un proceso importante por parte del usuario, también por parte de las empresas para asegurarse de que todos los usuarios que acceden a sus servicios tienen, por ejemplo, la edad adecuada.

Harán falta sistemas que sean capaces de leer los datos de un documento de identidad, número de documento, nombre de la persona, edad, etc. Simultáneamente comprobar que la persona que trata de identificarse sea la misma que la que aparece en el documento de identidad.

Esto puede parecer algo trivial, ya que lo hacemos de forma innata, sin embargo, crear un sistema que sea capaz de realizar todo esto puede ser un gran reto.

Aquí es cuando entran en juego las redes neuronales, que tratan de imitar el comportamiento del cerebro humano y que son muy buenas para hacer identificación de objetos y detecciones sobre imágenes debido a todas las capas de abstracción que consiguen y que serían muy complicado de igualar con los métodos tradicionales.

Es sobre esta idea en la que se basa el desarrollo de este proyecto, el estudio de una posible aplicación que solucione este tipo de ocasiones usando aprendizaje profundo.

En esta memoria se explicará todo el proceso realizado, las bases del aprendizaje automático, las redes neuronales y arquitecturas usadas. Así como todo el proceso de elaboración de las bases de datos y entrenamientos.

4.1. Contexto del proyecto y conocimientos previos

A la hora de comenzar el proyecto contaba con todos los conocimientos adquiridos a lo largo de todas las asignaturas del grado, y alguna investigación que había

realizado por mi cuenta de forma desinteresada. Pero se puede considerar que mis conocimientos en la materia eran prácticamente nulos, por lo que el trabajo se plantea desde el inicio, desde lo más básico para poder entender perfectamente que se estaba realizando en cada paso que se daba.

5. OBJETIVOS

El objetivo principal y general de este estudio es explorar el uso de las redes neuronales profundas en el ámbito de la verificación de identidad en dispositivos móviles. Los objetivos concretos son:

- Estudio de técnicas de aprendizaje automático.
- Estudio de técnicas de aprendizaje profundo.
- Estudio de arquitecturas de redes convolucionales orientadas a la extracción facial en documentos de identidad.
- Estudio de arquitecturas de redes siamesas orientadas a la correspondencia facial entre dos fotos.
- Realización de los modelos necesarios poniendo en práctica todo lo estudiado que resuelvan la extracción facial y la correspondencia facial.
- Implementación de una aplicación Android que haga uso de los modelos desarrollados.

6. ESTADO DEL ARTE

Como este proyecto se divide en dos partes sin contar con el estudio, una primera parte de extracción facial y una segunda de verificación facial, hay que tratar estos dos temas por separado en los antecedentes encontrados.

Tal y como se puede leer en [1], a la hora de hacer clasificación de imágenes, y reconocimiento facial las redes neuronales convolucionales son los métodos que mejores resultados arrojan en los últimos años, lo cual corrobora la idea original del proyecto de hacer uso de aprendizaje profundo.

En [2], se expone FaceNet una solución por parte de unos ingenieros de Google para reconocimiento facial que proclama en el documento tener un 99.63% de precisión. Explican gran cantidad de detalles de cómo se ha desarrollado el entrenamiento de la red, desde la cantidad de datos utilizada hasta cómo se realiza el cálculo de la pérdida. Desde luego es un documento que permite comprender gran parte de las necesidades que requiere una red neuronal a la hora de hacer detección facial.

En [3], se realiza una revisión y evaluación del aprendizaje profundo para detección de objetos en imágenes. Se llega a la conclusión de que los métodos tradicionales son inferiores a las alternativas realizadas con aprendizaje profundo. Se indica que las redes entrenadas para este propósito son un poco especiales con respecto a su comportamiento y cómo tratar con ellas. Se muestran también las arquitecturas genéricas de redes neuronales para detección de objetos y algunas modificaciones para hacerlas mejores, especializándolas en detección facial, en los cuales obtienen muy buenos resultados.

Finalmente, en [4] se plantea una solución usando redes siamesas para comprobar si dos firmas eran la misma. Aunque el caso de uso no sea el mismo, muchas de las ideas originales del documento pueden ser extrapoladas. En el artículo original se obtiene resultados del 95.5% de acierto, lo cual prueba de la validez de este tipo de arquitecturas para estas tareas y que podemos intentar seguir este ejemplo.

Ninguno de los trabajos anteriormente mencionados están expresamente pensados ni planteados para su uso en entornos móviles.

7. METODOLOGÍA

El método seguido para la realización del proyecto ha sido dividido en dos partes. Primero una investigación para conocer los elementos con los que tratamos y posteriormente una fase de desarrollo en la que se trata de poner en práctica todo lo aprendido.

7.1. Introducción y técnicas básicas de aprendizaje automático

El estudio se realiza siguiendo diferentes fuentes como [10] [11] [25] y [26]

El aprendizaje automático es una herramienta valiosa para los ingenieros para poder sacar provecho de sus datos.

Primero, es una herramienta que nos permitirá reducir el tiempo de programación. Por ejemplo, imaginemos que queremos escribir un programa para corregir errores ortográficos, esto se podría hacer con un algoritmo iterativo normal añadiendo muchas reglas generales de la ortografía española. O podríamos usar un programa de aprendizaje automático, introducirle algunos ejemplos y obtener un programa más confiable en una fracción del tiempo.

Segundo, permite personalizar los productos para determinados grupos específicos de personas. Imaginemos de nuevo que hemos creado el corrector ortográfico escribiendo el código a mano, y que, ahora queremos tener versiones para otros idiomas. Habría que empezar de cero con cada idioma, lo que llevaría mucho esfuerzo. Sin embargo, si este corrector fuera creado con aprendizaje automático, hacer diferentes versiones sólo pasaría por recopilar datos en ese nuevo idioma, e introducirlo al mismo modelo de aprendizaje automático.

Tercero, el aprendizaje automático permite resolver problemas que no sabríamos hacer a mano. Hay habilidades que tenemos los humanos, que damos por hechas y que son sencillas (a priori), pero que, si nos dedicamos a intentar replicar en un programa, no sabríamos por donde empezar. Por ejemplo, detectar un elemento en una imagen o reconocer una voz en una pista de audio. Pero estas tareas son las que los algoritmos de aprendizaje automático hacen muy bien, ya que, no hay que decirle

al algoritmo lo que tiene que hacer, basta con mostrarle muchos ejemplos, y, a partir de estos, se realiza la tarea.

No todo son ventajas, al principio el aprendizaje automático resulta complejo de asimilar, ya que estamos entrenados para pensar de forma lógica y matemática. Usamos afirmaciones para probar si las propiedades de nuestro programa son correctas. Con el aprendizaje automático, el enfoque cambia a un enfoque natural, hay que hacer observaciones, hay que usar la estadística, no la matemática, para analizar los resultados.

7.1.1. Terminología clave del aprendizaje automático

Primero deberemos saber qué es el aprendizaje automático (de ahora en adelante AA), una definición sería:

Los sistemas de AA aprenden cómo combinar entradas para producir predicciones útiles sobre datos nunca vistos.

Para continuar vamos a repasar la terminología básica utilizada en el aprendizaje automático:

- **Etiquetas**: valor que estamos prediciendo, es decir, variable **y** en la regresión lineal simple. Por ejemplo, el tipo de animal que se encuentra en una imagen, el significado de un clip de audio, o cualquier cosa.
- **Atributos**: variable de entrada al algoritmo, es decir, variable **x** en la regresión lineal simple. Un algoritmo de aprendizaje automático siempre podría usar un solo atributo, mientras que otro más sofisticado podría hacer uso de millones de atributos.
- **Ejemplo**: es una instancia de datos en particular. Hay dos categorías:
    - Ejemplo etiquetado: incluye tanto atributos como la etiqueta. Los ejemplos etiquetados son usados para entrenar el modelo.
    - Ejemplo sin etiqueta: contiene atributos, pero no la etiqueta.

      Los ejemplos etiquetados son usados para entrenar el modelo. Una vez el modelo ha sido entrenado, el modelo se usa para predecir la etiqueta en los ejemplos sin etiqueta.

- **Modelo**: define la relación entre atributos y etiquetas. Una vez el modelo ha sido entrenado podrá asociar de manera muy definida determinados atributos con las etiquetas. Los modelos tienen dos fases:
    - Entrenamiento: proceso en el cual se muestra al modelo ejemplos etiquetados y permites que este aprenda gradualmente las relaciones entre los atributos y la etiqueta.
    - Inferencia: significa aplicar el modelo entrenado a ejemplos sin etiqueta. Lo que es lo mismo, usas el modelo entrenado para realizar predicciones.

7.1.2. Entrenamiento y Pérdida

Entrenar un modelo simplemente significa aprender valores correctos para todas las ponderaciones y las ordenadas al origen de los ejemplos etiquetados.

En un algoritmo de AA, el algoritmo construye un modelo al examinar varios ejemplos e intentar encontrar un modelo que minimice la pérdida. Este proceso se denomina **minimización del riesgo empírico.**

La pérdida es una forma de medir las predicciones incorrectas. Esto quiere decir que la **pérdida** es un número que indica qué tan incorrecta fue la predicción del modelo en un solo ejemplo. Si la predicción del modelo es perfecta, la pérdida es cero, de lo contrario, la pérdida es mayor. El objetivo de entrenar un modelo es encontrar un conjunto de ponderaciones y ordenadas en el origen que, en promedio, tengan pérdidas bajas en todos los ejemplos. En un ejemplo de una regresión lineal simple, esto se puede ver muy fácilmente en unos gráficos como los de la imagen siguiente:

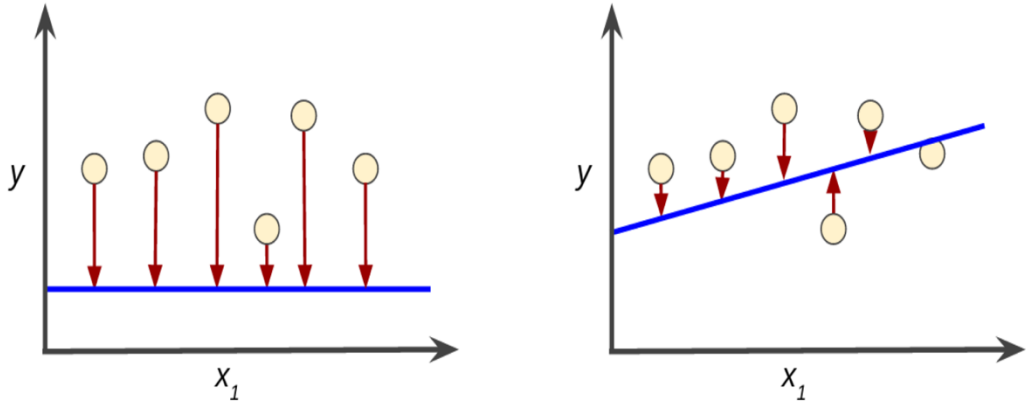


Figura 1: Comparación entre modelo con alta pérdida y modelo con baja pérdida

En la Figura 1 se ven las predicciones de dos modelos, en las cuales, la línea azul representa las predicciones, la flecha roja representa la pérdida, y el punto amarillo el valor real.

Como se puede observar el modelo de la izquierda tiene más perdida que el modelo de la derecha, con una pérdida más baja. Claramente, el modelo de la derecha es un modelo de predicción mucho más acertado que le de la izquierda, pues su pérdida es menos.

Hay varias formas de medir la pérdida, y estos métodos varían en función del modelo a tratar. Por ejemplo, si usamos un modelo como el de la imagen anterior, un modelo de regresión lineal, se puede usar la llamada **pérdida al cuadrado** (también conocida como L2). Se calcularía de la siguiente forma:

$$L2 = (observation - prediction(x))^2$$

Esto nos vale para calcular el error de predicción sobre un ejemplo, pero por lo general queremos saber una media de los errores, para ello, se suele usar el **error cuadrático medio (ECM)**, el cual es el promedio de la pérdida al cuadrado de cada ejemplo. Lo que es lo mismo:

$$ECM = \frac{1}{N} \sum_{(x,y) \in D} (y - predicition(x))^2$$

Donde:

- (x, y) es un ejemplo en el que:
    - **x** es el conjunto de atributos que el modelo usa en las predicciones.
    - **y** es la etiqueta del ejemplo
- prediction(x) es un atributo de las ponderaciones y las ordenadas al origen en combinación con el conjunto de atributos x.
- D es el conjunto de datos que contiene muchos ejemplos etiquetados, que son los pares (x, y).
- N es la cantidad de ejemplos en D.

7.1.3. Reducción de pérdida

Para poder reducir la pérdida, necesitamos algún tipo de guía, algo que nos indique en qué punto estamos y tras modificar el conjunto de hiperparámetros con los que trabajemos en el modelo, nos diga si estamos con más o menor pérdida que antes.

Una forma de obtener una dirección es el calculo de la gradiente, la derivada de la función de pérdida con respecto a los parámetros del modelo.

Para funciones de pérdida simples, como la pérdida al cuadrado, la derivada es fácil de calcular, y nos proporciona una forma muy eficaz de actualizar los parámetros de los modelos.

Si al calcular la gradiente es negativo, indica que la dirección para actualizar los parámetros de los modelos es la adecuada. Se realiza un paso en esa dirección, obtenemos una nueva versión del modelo y se vuelve a calcular el gradiente y se repite.

Pero ¿cómo de grande tiene que ser el paso que demos? Esto se determina por la **tasa de aprendizaje.** Es un hiperparámetro que podemos modificar. Si la tasa de aprendizaje es muy pequeña, daremos muchos pasos de gradiente, lo cual requiere mucho cálculo para alcanzar el valor mínimo.

Sin embargo, si la tasa de aprendizaje es muy alta, daremos un paso muy grande en dirección del gradiente negativo. Puede ocurrir que superemos el valor mínimo y

alcancemos un punto en donde la pérdida sea incluso más grande que antes. Si trabajamos con modelos con más dimensiones, puede provocar divergencia en el modelo.

Realizar el cálculo de gradientes es algo teórico que implicaría una gran cantidad de poder de computación. Por ello, en la práctica lo más común es **realizar descenso de gradientes de minilote**, lo que implica que, en lugar de calcular todas las gradientes de todos los ejemplos, se usa un lote pequeño, de entre diez y mil ejemplos para dar los pasos.

7.1.3.1. Uso de reducción de pérdida

A la hora de usar la reducción de pérdida en un modelo de AA, hay que usar un enfoque iterativo.

La siguiente figura muestra el proceso iterativo de prueba y error para entrenar modelos:

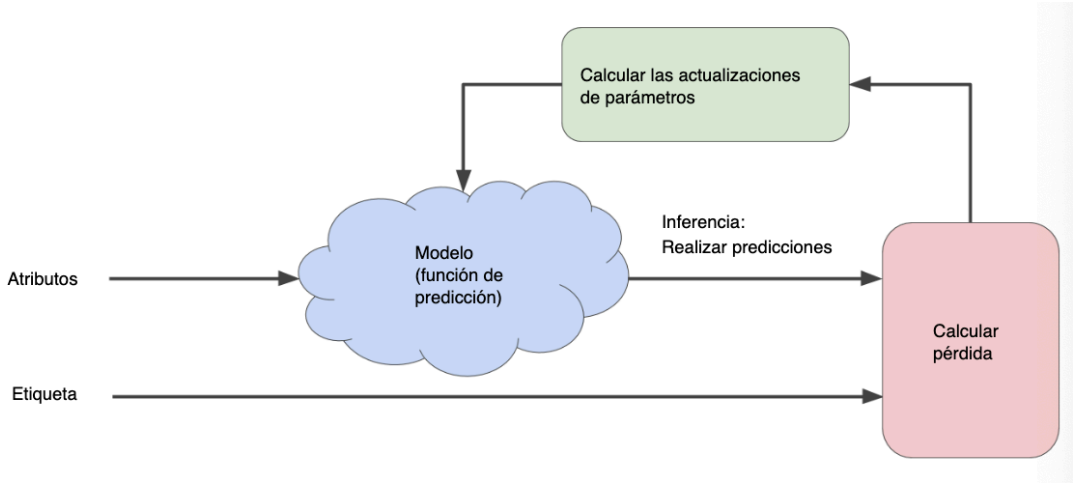


Figura 2: Esquema reducción de pérdida

El modelo toma uno o más atributos (x) como entrada y devuelve una predicción (y') como resultado. Si esto fuera un modelo de regresión simple la función que describiría este modelo sería:

$$y' = b + w_1 x_1$$

Aquí nos planteamos qué valores iniciales debemos establecer para b y w_1 inicialmente. En el caso de la regresión simple, estos valores de inicio no son importantes, se podrían elegir al azar.

Una vez tenemos los valores establecidos (nube azul en la imagen) se realiza inferencia y se procede al cálculo de la pérdida del modelo. En este paso se ve qué resultado da la predicción (y') y se comprueba con la etiqueta correcta (y) de los atributos (x) probados.

Finalmente, se realiza la función de **actualización de parámetros** de la imagen. Aquí el sistema de AA examina el valor de la función de pérdida y genera nuevos valores a los parámetros (b y w_1). En este punto se realizan una serie de comprobaciones, generan nuevos valores y el sistema de AA vuelve a evaluar todos los atributos con todas las etiquetas, obteniendo un nuevo valor de la función de pérdida, generando nuevos parámetros…

El aprendizaje continúa iterando hasta que el algoritmo descubre los parámetros del modelo con la pérdida más baja posible. En general, se itera hasta que la pérdida general deja de cambiar, cuando esto ocurre, se dice que el modelo ha **convergido**.

7.1.3.2. Descenso de gradientes

Basado en el trabajo publicado en [5].

Antes nos referíamos a "una serie de comprobaciones" que se realizaban en la caja verde del proceso iterativo de reducción de pérdida. Estas comprobaciones no es mas que el descenso de gradiente. Si bien a la hora de comenzar el entrenamiento de un modelo los valores iniciales de b y w_1 pueden ser aleatorios (normalmente 0), a la hora de modificar estos parámetros debemos de ir acercándonos a un punto en el que el modelo converja y la pérdida sea mínima. Para esto, las modificaciones de los parámetros no pueden ser aleatorias y necesitamos algo que nos guíe en alguna dirección (aumentar o reducir los parámetros).

Para esta tarea se realiza el descenso de gradientes. Si pudiéramos calcular la pérdida de todos los valores posibles de w_1 (lo cual es imposible pues son infinitos), tendríamos un gráfico como el que se muestra en la Figura 3:

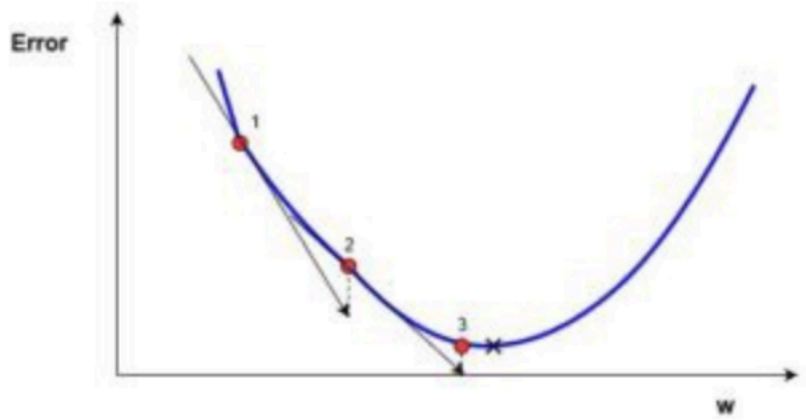


Figura 3: Esquema reducción gradiente

El valor de pérdida para los valores iniciales elegidos en el modelo sería el punto rojo que se encuentra con menor valor en eje de w.

La línea azul representa la pérdida para cada valor de w. Al principio empieza siendo alta, pues los valores de los parámetros han sido aleatorios. Si el problema es convexo (no tiene por qué serlo), la pérdida siempre tiene esta forma.

Como calcular todos los valores de pérdida para todos los posibles valores de w no es posible, se usa el descenso de gradientes, la cual funciona de la siguiente manera:

- Se elige un valor de inicio (punto de partida) para w. Como ya hemos visto anteriormente, esto puede ser un valor aleatorio. Esto nos daría en la gráfica el punto rojo que se encuentra más a la izquierda.
- El algoritmo de descenso de gradientes calcula la gradiente de la curva de pérdida en el punto de partida. Una gradiente es un vector de derivadas parciales (indica por dónde es más cerca o más lejos). Siendo vectores, tiene dirección y magnitud (vector que pasa por el punto rojo de pérdida). El algoritmo toma un paso en dirección de la gradiente negativa para reducir la pérdida lo más rápido posible.

7.1.3.3. Tasa de aprendizaje

Ya se ha visto que el vector de gradiente tiene una dirección y una magnitud. Los algoritmos de descenso de gradiente multiplican la gradiente por un escalar conocido como **tasa de aprendizaje** para determinar el siguiente punto.

La tasa de aprendizaje es uno de los hiperparámetros que tenemos para poder controlar y ajustar los algoritmos de AA. La mayor parte del tiempo del desarrollo de un programa usando AA es usada en ajustar la tasa de aprendizaje. Si es muy pequeña el aprendizaje tardará mucho tiempo. Por el contrario, si es muy grande, el siguiente punto rebotará al azar eternamente en la parte inferior sin nunca llegar a converger.

Para cada problema de regresión siempre hay una tasa de aprendizaje **con valor dorado**. El valor dorado está relacionado con qué tan plana es la función de pérdida. Si sabemos que el gradiente de pérdida es pequeño, deberíamos usar una tasa de aprendizaje mayor, que compensa este gradiente pequeño y dará como resultado un paso más grande.

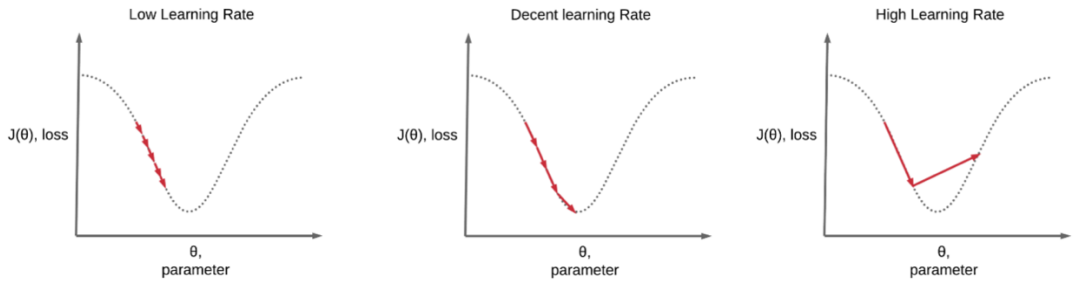


Figura 4: Comparación distintas tasas de aprendizaje

7.1.4. Generalización

Ajustar los hiperparámetros de entrenamiento no es suficiente para el AA. A la hora de realizar un modelo de AA nos interesa la generalización, es decir la capacidad de este modelo a adaptarse de manera adecuada a datos nuevos nunca vistos.

A la hora de entrenar un modelo, iremos viendo como la pérdida va siendo cada vez menor. Si intentamos tener una exactitud del 100% en los datos de entrenamiento, estaríamos modificando el modelo de manera mucho más compleja, adaptándola al

conjunto de datos de entrenamiento en vez de ser más precisa en la realidad de los datos que representan este conjunto de datos. El modelo tendrá una precisión del 100% sobre los datos de entrenamiento, pero no será bueno a la hora de predecir nuevos atributos nunca vistos.

Este problema es conocido como sobreajuste.

Figura 7: Conjunto de datos sobre el espacio

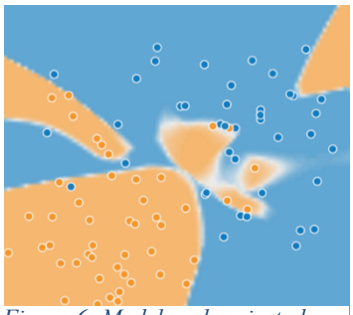


Figura 6: Modelo sobreajustado sobre los datos

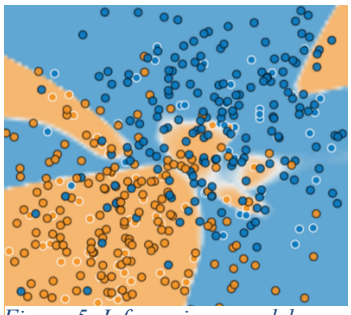


Figura 5: Inferencia en modelo sobreajustado

En las tres imágenes anteriores se puede apreciar el sobreajuste. En la Figura 7, intuitivamente podemos imaginar una línea divisoria de arriba hacia abajo y de izquierda a derecha, dejando la parte inferior izquierda para datos naranjas y la superior derecha para los datos azules, lo que podría resultar en un modelo con una precisión bastante buena.

¿Qué ocurre si seguimos entrenando sobre los mismos datos buscando el 100% de precisión? Pues que acabamos con un modelo complejo como el de la Figura 6, que describe muy bien el conjunto de datos de entrenamiento, pero que, si miramos la a la Figura 5, que muestra otros datos con los que no se ha entrenado, podemos ver que

la precisión es bastante inferior a la que sería con el modelo simple que se plantea anteriormente.

Esto ocurre porque siempre podemos tener datos anómalos o características anómalas que no tengan el resto de los datos, haciendo que el modelo se ajuste a estas peculiaridades y empeore el rendimiento con los datos nuevos.

Para poder probar que no hay sobreajuste en nuestro modelo, generalmente se divide el conjunto de datos etiquetado que tengamos en dos:

- Conjunto de entrenamiento: subconjunto usado para el entrenamiento del modelo.
- Conjunto de prueba: subconjunto usado para probar el modelo.

Un buen rendimiento en el conjunto de prueba es un indicador útil de buen rendimiento en los datos nuevos en general siempre que el conjunto de prueba sea lo suficientemente grande.

7.1.5. Conjunto de entrenamiento y prueba.

Por lo general se parte de un único conjunto de datos etiquetados para el entrenamiento del modelo, cuanto mayor sea este mejor. Pero tal y como se ha explicado anteriormente, necesitamos dos conjuntos separados. A la hora de dividirlos hay que tener dos cosas en cuenta por cada conjunto:

- Que sea lo suficientemente grande como para generar resultados significativos desde el punto de vista estadístico.
- Que sea representativo de todo el conjunto de datos, es decir, no hagamos un conjunto de prueba con características diferentes al del conjunto de entrenamiento.

Es muy importante que **nunca** se usen los datos de prueba en el entrenamiento, pues el modelo se ajustaría a estos últimos y los resultados no serían de valor.

7.1.6. Conjunto de validación

Con lo que se conoce hasta este punto, el flujo de entrenamiento sería algo como lo siguiente:

- Entrenar el modelo con el conjunto de entrenamiento.
- Evaluar el modelo con el conjunto de prueba.
- Ajustar el modelo en función de los resultados con el conjunto de prueba.
- Tras varias modificaciones, seleccionar el modelo que mejor se desempeñe con el conjunto de prueba.

La división del conjunto de datos en dos es un gran paso adelante contra el sobreajuste, sin embargo, la mayoría de las ocasiones no es suficiente, se puede reducir en gran medida el sobreajuste realizando una tercera partición en los datos.

El **conjunto de validación** se introduce para evaluar los resultados del conjunto de entrenamiento. Modificando el flujo de entrenamiento a lo siguiente:

- Entrenar el modelo con el conjunto de entrenamiento.
- Evaluar el modelo con el conjunto de **validación.**
- Ajustar el modelo en función de los resultados con el conjunto de **validación**.
- Tras varias modificaciones, seleccionar el modelo que mejor se desempeñe con el conjunto de **validación**.
- Confirmar los resultados con el conjunto de **prueba**.

Este flujo obtiene mejores resultados pues se crea menos exposición al conjunto de prueba durante el entrenamiento.

7.1.7. Representación: Ingeniería de atributos

Los proyectos de AA se enfocan en la representación, a diferencia de la programación tradicional que se centra en el código, por lo que, una forma de perfeccionar los modelos es añadiendo y mejorando atributos.

A la hora de entrenar un modelo, no podemos darle un conjunto de datos sin procesar, hay que pasarle un **vector de atributos.** Esto es la ingeniería de atributos pasar de datos sin procesar a vectores de atributos que poder dar a los modelos.

Hay tipos de datos que no requieren de codificaciones especiales, como datos numéricos, por ejemplo, el número de ruedas de un vehículo. Pero hay otro tipo de datos de valores categóricos que si que requieren de más trabajo. Un ejemplo de esto sería el color del vehículo, que puede tener varios valores. Estos colores pueden ser cadenas de caracteres que los modelos de AA no pueden multiplicar por pesos aprendidos. Aplicando ingeniería de atributos podemos pasar estos valores de los atributos a números. Podemos asignar a cada color un valor numérico único para esa marca.

Sin embargo, si incorporamos estos números índice directamente en nuestro modelo, se generan dos problemas:

- El modelo aprenderá un peso único que se aplicará a todos los colores. Por ejemplos si el modelo aprende un peso 5 para el atributo color, luego se multiplicará por 0 para una, 1 para la siguiente, 2 para la tercera… Consideremos un modelo que prediga el precio del vehículo usando el color como atributo. Es poco probable que haya un ajuste lineal en el precio basado en el color del vehículo. Nuestro modelo necesita flexibilidad par aprender diferentes pesos para cada calle, que se irán agregando al precio estimado usando los otros atributos.
- No se están contemplando los casos en los que el color pueda tener múltiples valores. Por ejemplo, los vehículos que tienen el techo de un color y el resto del cuerpo de otro y no hay forma de codificar esa información en el valor color si este contiene un solo índice.

Para eliminar estas restricciones, podemos crear un vector binario para cada atributo categórico de nuestro modelo que represente los valores de la siguiente manera:

- En el caso de los valore que aplican al ejemplo, establecer los elementos correspondientes al vector en 1.
- Establecer todos los demás elementos a 0.

La longitud de este vector será igual a la cantidad de elementos del vocabulario. Esta representación se denomina codificación **one-hot** cuando un único valor es 1 y codificación **multi-hot** cuando varios valores son 1.

Esto crea, en la práctica, una variable booleana para el valor de cada atributo. En este caso, si un vehículo es de color rojo, el valor binario sería 1 únicamente para el color rojo. De esta manera, el modelo usa únicamente el peso para el color rojo.

De manera similar, si un vehículo tiene dos colores, entonces los dos valores binarios se establecen en 1 y el modelo usa ambos pesos respectivos.

¿Qué ocurre si tenemos 1 millón de colores distintos y queremos incluirlos todos como valores de color? Crear un vector binario con 1 millón de elementos, en los que sólo 1 o 2 son verdaderos sería muy ineficaz respecto al almacenamiento y al tiempo de procesamiento. En este caso, una solución común sería usar una **representación dispersa** en la que solo se almacenan los valores que no son cero. En las representaciones dispersas, se aprende un peso modelo independiente para cada valor de atributo, igual que se explica anteriormente.

7.1.7.1. Cualidades de buenos atributos

No todos los valores que nos encontramos en los datos sin procesar tienen porqué ser necesariamente buenos o representativos. Es por esto por lo que debemos analizarlos y elegir qué tipos de valores son los que necesitamos en los vectores de atributos.

Deberíamos de evitar los valores de **atributos discretos con poco uso**. Los atributos buenos deben aparecer al menos 5 veces en un conjunto de datos. Esto permite que un modelo aprenda cómo se relaciones este valor de atributo con la etiqueta. Es decir, tener muchos ejemplos con el mismo valor discreto le permite al modelo ver el atributo en diferentes escenarios y, a su vez, determinar cuándo es una buena predicción para la etiqueta.

Si vemos que un valor de un atributo aparece solo una vez o muy de vez en cuando, el modelo no podrá realizar predicciones basadas en ese atributo, por lo que es incluso mejor no incluirlo.

Debemos intentar buscar los **significados claros y evidentes**, cada atributo debería tener un significado claro y evidente para todas las personas que vean el proyecto.

No es recomendable medir el tiempo en milisegundos si estamos representando la edad de un vehículo que puede llegar a alcanzar varias décadas, mejor hacerlo en años.

No debemos incluir valores "**mágicos**" en los atributos, por ejemplo, si tenemos un atributo de punto flotante entre 0 y 1, no debemos tener valores de este atributo como -1. Los atributos deben tener solo calificaciones de calidad nunca valores mágicos. Si queremos indicar la existencia o no de un atributo, siempre se puede crear otro atributo que indique si ese atributo tiene un valor relevante para ese ejemplo o no.

7.1.7.2. Limpieza de datos

A la hora de desarrollar un modelo de AA, se dedica mucho tiempo a desechar ejemplos malos y preparar los que sirven. Unos pocos de ejemplos no válidos pueden arruinar todo el conjunto de datos.

Hay varias cosas que realizar a la hora de limpiar los datos de un conjunto de datos.

- **Ajustar valores de atributos**: convertir los valores de atributos en coma flotante de su rango natural (por ejemplo, de 100 a 900) al rango estándar (de 0 a 1 o -1 a +1). Si un conjunto de atributos consiste solo en una única función, el ajuste no ofrece ningún beneficio real. Sin embargo, si el conjunto de atributos consiste en varios atributos, el ajuste puede acelerar la convergencia del descenso de gradientes, evitar la "trampa de N/A", en la que un modelo se convierte en NaN (por ejemplo exceder el rango de precisión de punto flotante) y por las operaciones matemáticas el resto de los números en el modelo se convierten en NaN, y permite que el modelo aprenda las ponderaciones correspondientes para cada atributo. Sin el ajuste, el modelo prestaría más atención a los atributos con un rango más amplio.
- **Manejo de valores atípicos extremos**, por ejemplo, si queremos realizar un modelo que prediga el precio de un vehículo, deberíamos excluir del conjunto de datos coche de lujo, los cuales son pocos y suponen una gran diferencia con la mayoría. Para eliminar estos valores atípicos podemos usar alguna de las siguientes técnicas:
    - Podemos usar como valor el logaritmo de cada valor, de esta manera reducimos proporcionalmente todos los valores.

  - Si el ajuste logarítmico todavía tiene cola, podemos fijar un valor límite. De esta manera, todos los valores que superen un determinado valor no son eliminados, si no que, tendrán el valor límite.

- **Discretización** para atributos cuya representación no tenga sentido en coma flotante y queramos representarlo por rangos de valores. Creando vectores como si de valores categóricos se tratara.
- **Arrastre:** consiste en eliminar ejemplos incorrectos cuando los detectemos. Alguno de estos datos puede ser incorrectos por alguna de estas razones:
  - Valores omitidos: que a alguien se le haya olvidado ingresar el valor de un atributo en un ejemplo.
  - Ejemplos duplicados.
  - Etiquetas incorrectas.
  - Valores de atributos incorrectos.

7.1.8. Regularización: Simplicidad

La regularización es otro de los aspectos claves del AA, junto con la reducción de la pérdida de entrenamiento.

Cuando entrenamos un modelo, cuántas más iteraciones hagamos, más sobreajuste tendremos y por lo tanto más se reduce la pérdida en datos de entrenamiento, en algún punto la pérdida convergerá y no bajará más. Pero en algún punto, la pérdida sobre los datos de validación comenzará a subir. Esto se debe a que se produce el sobreajuste del que hablamos anteriormente.

Para evitar este sobreajuste, usamos la regularización y sus diferentes estrategias, como la interrupción anticipada, que detiene el entrenamiento en la convergencia de datos. También se puede intentar penalizar la complejidad del modelo que es lo que veremos en un momento.

Hasta ahora nos hemos enfocado en obtener los datos correctos y minimizar el riesgo empírico, pero ahora empezaremos a usar lo que se denomina minimización del riesgo estructural para equilibrar la pérdida y la complejidad del modelo.

Pero ¿cómo medimos la complejidad del modelo? Esto podemos hacerlo de varias formas. Una es usar ponderaciones pequeñas sin afectar los ejemplos correctos, esta ponderación puede venir dada por una regularización N2, L2… El primer término será el de pérdida que depende de los datos de entrenamiento, el segundo será el de la regularización que no depende de los datos, este sólo indica que necesita un modelo más simple, es como una lambda que equilibra los dos términos.

7.1.8.1. Regularización L2

En la Figura 8 se muestra un modelo en el que la pérdida en entrenamiento se reduce gradualmente, pero la pérdida en validación eventualmente aumenta. Lo que es lo mismo, es un gráfico que muestra sobreajuste a los datos de entrenamiento.

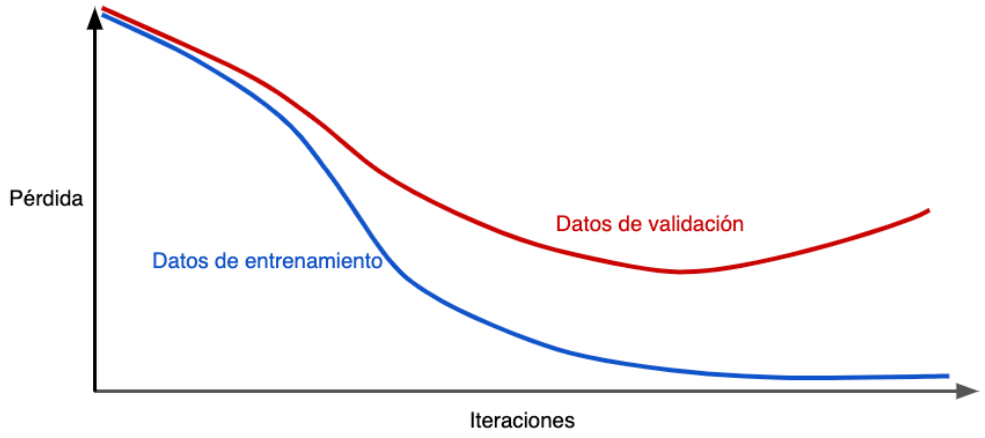


Figura 8: Gráfico en el que se aprecia sobreajuste

Para poder prevenir esto, pasamos de intentar reducir la pérdida de esta manera:

$$minimize(Loss(Data|Model))$$

A algo como la siguiente fórmula:

$$minimize(Loss(Data|Model) + complexity(Model))$$

Nuestro algoritmo de optimización es ahora una función de dos términos, el término de **pérdida** que mide qué tan bien se ajusta el modelo a los datos, y el término de **regularización**, que mide la complejidad del modelo.

Hay dos formas comunes de interpretar la complejidad del modelo:

- Como una función de las ponderaciones de todos los atributos que contiene.
- Como una función de la cantidad total de atributos con ponderaciones que no sean cero.

Podemos cuantificar la complejidad mediante la fórmula de la **regularización L2**, que define se define de la siguiente manera:

$$L_2 regularization = ||\omega||_2^2 = \omega_1^2 + \omega_2^2 + \cdots + \omega_n^2$$

No es más que la suma de los cuadrados de todas las ponderaciones de atributos.

En esta fórmula, las ponderaciones cerca de cero tienen poco efecto en la complejidad del modelo, mientras que las ponderaciones atípicas pueden tener un gran impacto.

7.1.8.2. Lambda

Para poder definir el impacto general de la regularización, se suele multiplicar su valor por un escalar conocido como **lambda** (o tasa de regularización), es decir se modifica la función a lo siguiente:

$$minimize(Loss(Data|Model) + \lambda\, complexity(Model))$$

7.1.8.3. Regularización L2

El efecto que tiene la regularización L2 en un modelo es el siguiente:

- Lleva los valores de las ponderaciones hacia 0 (sin llegar a él).
- Lleva la media de las ponderaciones hacia 0, con distribución normal (en forma de campana gaussiana).

Aumentar el valor de lambda aumenta el efecto de la regularización. Es por esto por lo que hay que saber elegir este valor para lograr el balance adecuado entre la simpleza y el ajuste de los datos de entrenamiento. Si el valor es demasiado alto, el modelo será simple, pero hay riesgo de subajuste. El modelo no aprenderá lo suficiente sobre los datos de entrenamiento para hacer predicciones precisas. Si por el contrario el valor de lambda es demasiado bajo, el modelo será complejo, pero hay riesgo de sobreajuste de datos.

7.1.9. Regresión logística

A la hora de hacer predicciones es posible que obtengamos resultados anómalos, para controlar esto debemos introducir un método que interprete los valores como probabilidades entre 0 y 1, y nunca se salga de este rango.

Esta idea se llama **regresión logística**. Esto nos ayuda ya que muchos problemas exigen como resultado el cálculo de una probabilidad. Usando la regresión logística podemos calibrar los cálculos de las probabilidades como necesitemos según el caso.

7.1.9.1. Cálculo de probabilidades

Para conseguir que la regresión logística pueda garantizar que todos los valores se encuentran entre 0 y 1, se usa una función sigmoidea que es definida como sigue:

$$y = \frac{1}{1 + e^{-z}}$$

Y cuya representación se puede ver en la Figura 9:

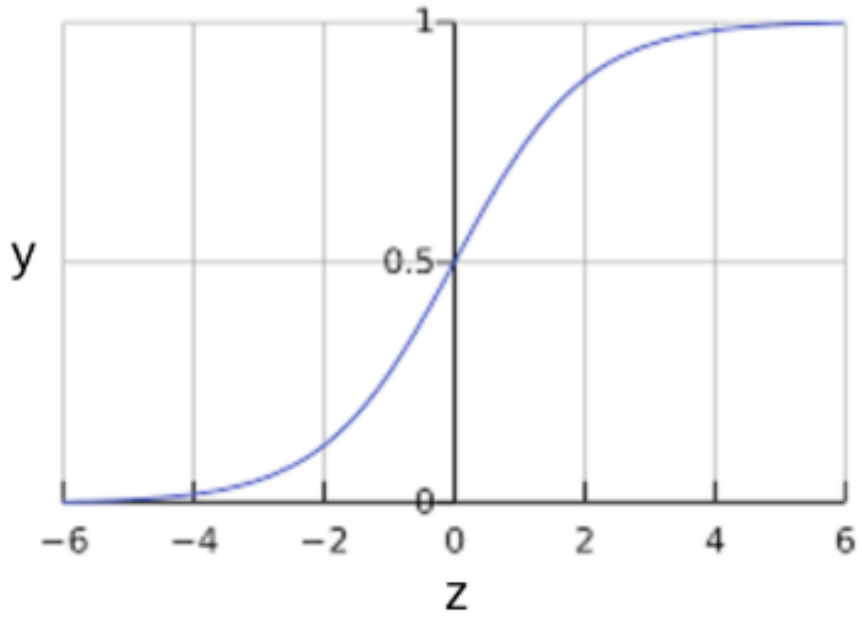


Figura 9: Representación función sigmoidea

Z representa el resultado de la capa lineal de un modelo entrenado con regresión logística, la función sigmoidea(z) generará un valor entre 0 y 1 siguiendo la siguiente función:

$$y' = \frac{1}{1 + e^{-(z)}}$$

Donde:

- y' es el resultado del modelo de regresión logística para un ejemplo concreto.
- z es $b + w_1x_1 + w_2x_2 + \ldots + w_nx_n$
    - Los valores w son los pesos aprendidos del modelo, y b es la ordenada en el origen.
    - Los valores x son los valores de atributo para un ejemplo concreto.

7.2. Introducción al aprendizaje profundo

El aprendizaje profundo es un subcampo del aprendizaje automático que hace uso de algoritmos inspirados por la estructura y funcionamiento del cerebro humano. Estos algoritmos son llamados redes neuronales. No dejan de ser versiones mucho más sofisticadas de las combinaciones de atributos de AA, ya que las redes neuronales aprenden las combinaciones de atributos que necesitan.

7.2.1. Concepto de red neuronal

Las redes neuronales son un tipo de algoritmos de aprendizaje automático que imitan el funcionamiento del cerebro humano. Consisten en un conjunto de unidades de computación que reciben el nombre de neuronas, conectadas entre sí para transmitir información. La información de entrada atraviesa la neurona, la cual modifica la entrada y produce una salida que es pasada a la siguiente neurona.

Las comunicaciones entre neuronas tienen una serie de pesos, estos pesos multiplican el valor de salida de una neurona antes de pasárselo a la siguiente, incrementando, reduciendo o inhibiendo los valores. De esta forma podemos dar más importancia a según qué patrones detecte una neurona, por ejemplo.

Dentro de cada neurona, hay una serie de pesos, estos pesos indican cómo modificar la entrada de la neurona para dar una salida. Variando estos valores, modificamos la forma en la que se toman las decisiones.

Obviamente una neurona por sí sola no sirve de nada, al final se trabaja con redes más complejas, como la de la Figura 10:

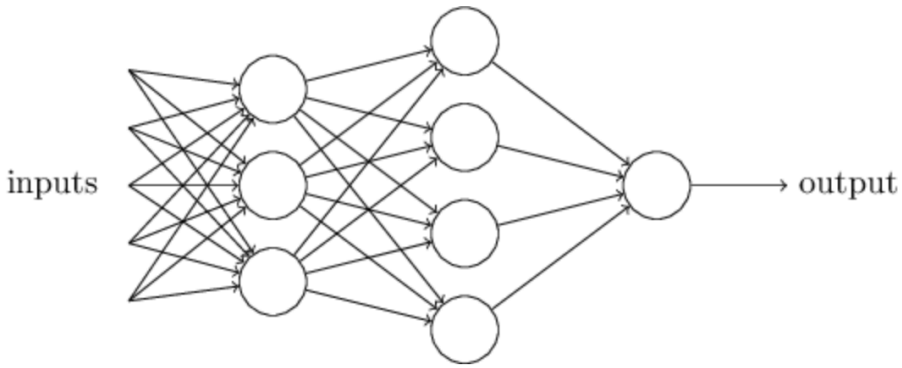


Figura 10: Ejemplo red neuronal con 3 entradas y 1 salida.

En el modelo visto en la figura, la primera línea de neuronas se conoce como **capa de entrada**. En este caso, se ocupa de hacer 3 decisiones, creando unos pesos para la entrada. La segunda capa, hace las decisiones basándose en las salidas de la capa de entrada multiplicados por unos pesos. Estas capas se conocen como **capas ocultas.** De esta manera las neuronas de la segunda capa pueden tomar decisiones en un nivel más complejo y más abstracto que las neuronas de la capa de entrada. Finalmente, las neuronas que realizan las decisiones más abstractas son las que se encuentran en la **capa de salida**, proporcionando el resultado de la red.

Mientras que las capas de entrada y de salida pueden tener una arquitectura un poco lógica (una neurona por atributo de entrada, y una neurona por cada resultado dado, respectivamente), el diseño de las capas ocultas puede llegar a ser muy complejo, no hay una serie de reglas heurísticas para saber cómo diseñar estas capas.

Se ha mencionado que los valores de una capa pasan a la siguiente, este proceso es conocido como "**FeedForward**", esto significa que no hay bucles en el grafo definido en la red. Sin embargo, hay otros tipos de redes llamadas **redes neuronales recurrentes.**

Este tipo de algoritmos son especiales porque pueden realizar combinaciones de atributos automáticamente, lo que las hace ideales para problemas no lineales. Estas combinaciones se forman durante la fase de entrenamiento de la red en la cual se establecen los pesos de las conexiones y son los que modificarán las salidas de la red finalmente.

7.2.2. Concepto de red neuronal profunda

Todo lo que se expone, parte de los conocimientos adquiridos leyendo y analizando [18] [19] [20] [21] [22] y [23].

Si queremos que una red nos responda a una pregunta compleja sobre una imagen, por ejemplo, si hay una cara en ella, la red deberá descomponer esta pregunta en otras preguntas más simples y que sean más fáciles de responder. Estas últimas preguntas en algún punto serán a nivel de píxel. Esto se hace con una serie de muchas capas, con las capas iniciales respondiendo a preguntas muy simples y capas

posteriores construyendo una jerarquía de conceptos todavía más complejos y abstractos. Este tipo de redes neuronales se llaman profundas.

7.2.2.1. Cálculo de gradiente

Para el cálculo de gradientes en redes neuronales profundas se usa un algoritmo llamado "BackPropagation". Este algoritmo es el principal mecanismo mediante el cual una red neuronal aprende. Se conoce como propagación al paso de mensajes de una neurona a otra.

Como se ha visto anteriormente, una red neuronal propaga la señal de entrada mediante sus parámetros hacia las capas ocultas, y posteriormente hay una **propagación hacia atrás** [18] con información sobre la pérdida, en orden inverso al de la red, para que se puedan modificar los parámetros. Esto ocurre en los siguientes pasos:

- La red hace una predicción sobre unos datos usando sus parámetros.
- Se evalúa la red con una función de pérdida.
- El error obtenido se propaga hacia atrás para ajustar los parámetros y mejorar la pérdida.

7.2.2.2. Dropout

El Dropout es una técnica de prevención de sobreajuste de la red radicalmente diferente a la regularización. Dropout no modifica la función de costes, lo que hace es modificar la red en sí.

Supongamos que entrenamos una red, tenemos un data de entrada x y la correspondiente salida deseada y. Normalmente, entrenaríamos la red con la propagación hacia delante de x por la red, y después aplicando propagación hacia atrás para optimizar la pérdida. Con Dropout este proceso se modifica.

Se empieza borrando aleatoriamente (y temporalmente) la mitad de las neuronas ocultas de la red, dejando las de entrada y las de salida sin tocar.

Se propaga ahora la entrada x por la red modificada, y después se propaga el error hacia atrás, también modificando la red. Después de esto sobre un conjunto de datos,

se actualizan los pesos y parámetros de la red. Se sigue repitiendo el proceso, primero restaurando las neuronas borradas anteriormente, y continuando eligiendo otras neuronas aleatorias que serán borradas.

Este proceso es un poco extraño al principio. Pensemos en un entrenamiento de una red neuronal normal sin Dropout. En particular, imaginemos un entrenamiento de varias redes neuronales distintas, todas usando el mismo conjunto de datos de entrenamiento. Por supuesto, cada una finalmente puede dar resultados distintos a las otras. Cuando esto ocurre podemos usar algún tipo de media para decidir que salida elegir como válida. Si entrenamos 5 redes, y 3 de ellas están clasificando una imagen con un tipo, posiblemente sea ese tipo. Las otras 2 puede que estén cometiendo un error. Este tipo de esquemas son según que casos una forma muy eficaz de reducir el sobreajuste de una red. La razón es porque diferentes redes pueden sobreajustarse de diferentes formas y haciendo una media puede eliminar este sobreajuste.

Al final el Dropout no hace otra cosa que simular el entrenamiento de distintas redes en una misma red. Por lo que el efecto del Dropout es como hacer la media de una gran cantidad de redes distintas.

7.2.2.3. Problemas entrenamiento redes neuronales profundas

A veces, durante el entrenamiento de una red neuronal puede ocurrir que las capas finales de la red estén aprendiendo bien, mientras que las primeras capas se queden estancadas en el aprendizaje sin casi aprender nada. Este estancamiento no es simplemente mala suerte. Es mas, este descenso en el aprendizaje viene dado por el uso de técnicas basadas en el uso de gradientes para el aprendizaje.

También puede ocurrir que sean las primeras capas la que aprenda, y las últimas las que queden estancadas. De hecho, vamos a ver que hay una inestabilidad intrínseca asociada al aprendizaje basada en el descenso de gradientes en redes neuronales profundas. Esta inestabilidad es la que causa el estancamiento.

Para saber qué ocurre en el proceso, tenemos que visualizar cómo aprende la red.

La red se inicializa con valores aleatorios, por lo tanto, no es sorprendente saber que hay mucha variación entre cómo de rápido cada neurona aprende, las que tengan un valor aleatorio más cercano del correcto aprenderán más lentamente que las que tengan un valor muy separado. Pero hay un detalle a tener en cuenta y es que las

gradientes siempre tienden a ser menores cuanto más nos movemos hacia atrás, lo que hace que las capas iniciales no aprendan tan rápido como las finales. Este fenómeno tiene el nombre de **problema de fuga de gradiente** [17].

Mientras que si es al contrario, son las primeras capas las que tienen mayor gradiente que las últimas se conoce como **problema del gradiente explosivo**.

La raíz del problema es la inicialización de pesos y parámetros aleatoria de la red al inicio del entrenamiento. Estos valores suelen ser cercanos a 0, y desde ahí la red los va modificando hacia arriba. Pero cuando se empieza con valores cercanos a 0, al hacer la propagación hacia atrás, se multiplica el valor de gradiente por este valor cercano a 0, reduciéndolo con cada paso.

Esto implica que cuanto menor sea la gradiente más difícil será para la red el actualizar los pesos y más tardará en llegar al resultado final.

Las soluciones a estos problemas pueden ser, para fuga de gradiente.

- Inicializar los pesos de forma que se minimice el potencial de la fuga.
- Usar Long Short-Term Memory Networks (LSTMs) [16]

Para solucionar problemas de gradiente explosivo:

- Para la propagación hacia atrás en algún punto, lo cual no es normalmente óptimo porque no todos los pesos se actualizarán.
- Penalizar o reducir artificialmente la gradiente.
- Poner un límite máximo a la gradiente.

7.2.3. Concepto de red convolucional

Este tipo de redes están especialmente adaptadas para clasificar imágenes. Las redes convolucionales están basadas en tres ideas básicas: campos locales de recepción, pesos compartidos y pooling.

7.2.3.1. Campos receptivos locales

En las redes convolucionales es más simple pensar en las capas de neuronas como matrices de neuronas más que como vectores, es decir en vez de tener una capa de

1x16 neuronas, es mejor pensar en una capa de 4x4 neuronas. Cada neurona correspondiente a un conjunto de pixeles dependiendo de la cantidad de estos que tenga la entrada.

Como es normal, conectaremos las neuronas de entrada con las neuronas de la segunda capa, pero en este caso, no conectaremos cada una de las neuronas de la primera capa con cada una de las de la segunda, ahora sólo haremos conexiones en regiones pequeñas y localizadas de la imagen de entrada.

Por ejemplo, si tenemos una capa de entrada de 25x25 neuronas, conectaremos regiones de 5x5 con la siguiente capa oculta. Esta región se llama campo receptivo local. Cada conexión aprende un peso. Se puede pensar que cada neurona oculta está aprendiendo a analizar su campo receptivo local.

Si recorremos toda la matriz de neuronas de entrada con campos receptivos locales, tendremos una primera capa oculta cuyo tamaño será menor que la capa de entrada.

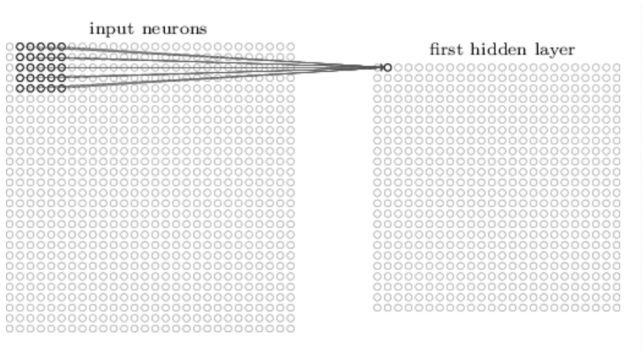


Figura 11: Conexión entre capa de entrada y primera capa oculta

7.2.3.2. Pesos y parámetros compartidos

Como se ha dicho, cada neurona oculta tiene un peso y 5x5 parámetros conectado a su campo receptivo local, y estos mismos valores serán usados para cada una de las neuronas ocultas. Esto significa que todas las neuronas en la primera capa oculta detectarán exactamente el mismo patrón, solo que en distintas posiciones de la imagen.

Para hacer reconocimiento de imágenes se necesitarán más de una capa con patrones, por lo que una capa convolucional completa consta de varias capas, por lo que es normal encontrar capas descritas de la siguiente manera: 3x24x24 neuronas. Cada una de estas capas se conocen como “feature map”.

7.2.3.3. Capas de pooling

Por lo general las capas de pooling se encuentran inmediatamente después de capas de convolución. Lo que hacen las capas de pooling es simplificar la información que tienen como salida las capas de convolución.

Una capa de pooling coge la salida de cada feature map de la capa de convolución y prepara un feature map condensado. Lo que es lo mismo, reduce este feature map a un tamaño más pequeño.

Como una capa de convolución suele tener más de un feature map, se aplica pooling a cada feature map por separado, lo que implica que cada capa de pooling que se tenga estará compuesta por el mismo número de feature map que en la capa de convolución anterior.

Lo que está haciendo es detectar que se ha encontrado un patrón, sin importar tanto dónde.

7.2.3.4. Juntando todo

Si juntamos todo tendremos algo como lo que se muestra en la Figura 12:

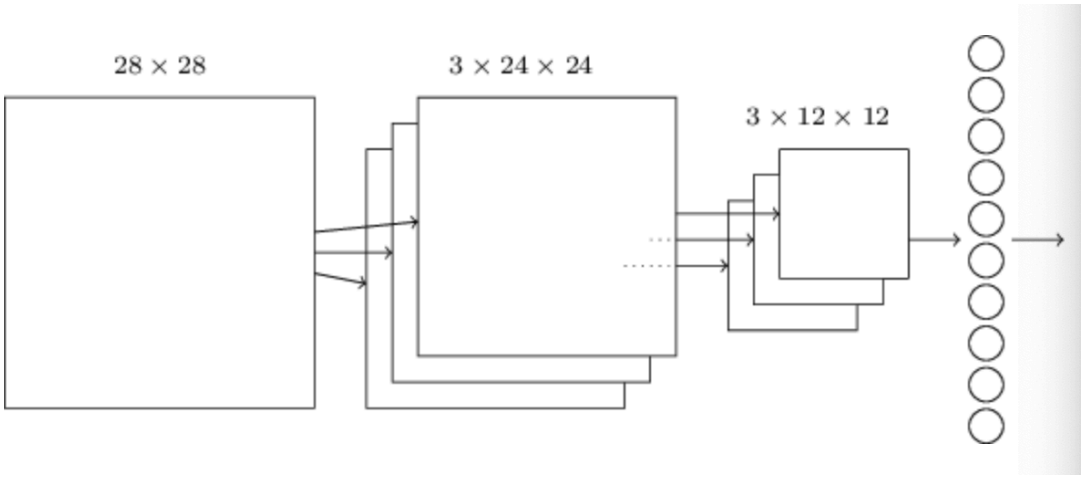


Figura 12: Red neuronal convolucional

La red comienza con una capa de entrada de 28x28 neuronas, las cuales son usadas para codificar los píxeles de las imágenes. Después le sigue una capa de convolución que usa campos receptivos locales de 5x5 y 3 feature maps. El resultado son 3x24x24

capas ocultas de neuronas. El siguiente paso es la capa de pooling, aplicando regiones de 2x2, a los 3 feature maps. El resultado es una capa oculta de 3x12x12 neuronas.

La capa final es una capa completamente conectada. Esto quiere decir que todas las neuronas de la capa anterior están conectadas con cada una de las neuronas de la última capa.

7.2.4. Non Maximum Suppresion

En las redes neuronales para detección de objetos se da una situación un poco peculiar. Casi con seguridad, se encontrarán distintas cajas que contengan al objeto, algunas mas altas, algunas mas anchas, algunas un poco descentradas… Podemos encontrarnos con una situación en la que tenemos un total de 6 detecciones, y todas ellas válidas, pero no queremos que el algoritmo nos de 6 posibilidades, nosotros queremos sólo una.

Para solucionar este "problema" existen algoritmos llamados Non-Maximum Suppresion (NMS).

Estos algoritmos lo que hacen es recibir todas las cajas, y calcular una que sea la media de las demás.

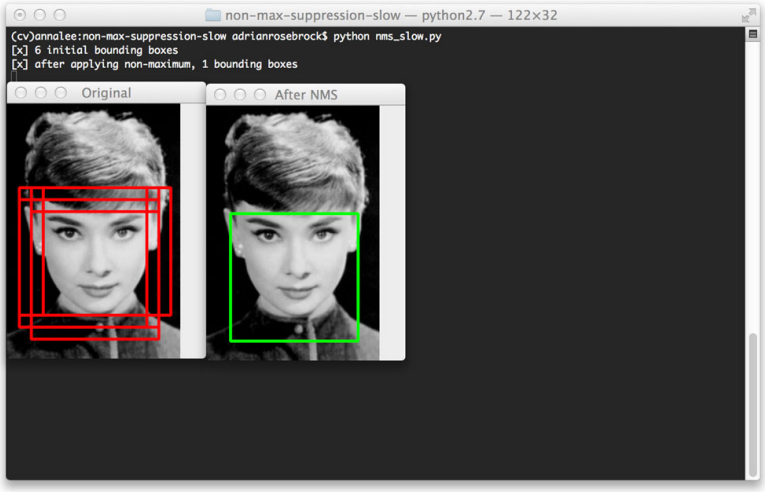


Figura 13: Demostración algoritmo NMS

7.2.5. Técnicas de aumento de datos

Las técnicas de preprocesado y aumento de datos son utilizadas cuando el conjunto de datos del que disponemos no es suficiente para realizar un entrenamiento completo de una red.

Principalmente se basan en modificar las imágenes que tenemos originales tratando de hacer creer a la red que son nuevas imágenes intentando evitar el sobreajuste.

Alguna de estas técnicas puede ser el **volteo** de las imágenes respecto del eje de las Y. De esta forma tendremos una imagen espejo de la original que podrá ser utilizada para el entrenamiento.

Según qué casos, podremos usar también imágenes **giradas** 90º y múltiplos de 90º si sobrepasar los 360º con lo que volveríamos a la imagen original.

Por otro lado, podemos hacer **zoom** en las imágenes, recortando los bordes un poco, con lo que tendremos una imagen con el objeto que queramos detectar un poco más grande. Siguiendo un poco con esta idea, podemos intentar **trasladar** el objeto dentro de la imagen, si está en el centro, moverlo a izquierda y derecha.

Podemos añadir **ruido gaussiano** a la imagen lo cual haría que las imágenes no tuvieran el mismo patrón, y que podría ayudar a evitar el sobreajuste.

7.2.6. Entornos de trabajo de aprendizaje profundo

7.2.6.1. Tensorflow

Tensorflow es una librería software Open Source para aprendizaje automático. Fue originalmente desarrollado por Google, y es actualmente una de las librerías de AA más populares.

Tensorflow es multiplataforma. Puede funcionar en casi cualquier CPU o GPU incluyendo las móviles.

Los componentes básicos de Tensorflow son dos:

- Tensores: matemáticamente un tensor es un vector N-dimensional, lo que significa que un tensor puede representar conjuntos de datos N-dimensionales. Por ejemplo, la Figura 14:

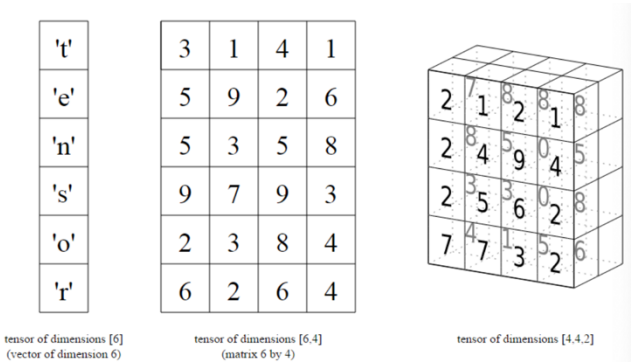


Figura 14: Tensores simplificados

- Grafos computacionales (flow): hace referencia a un grafo no cíclico en el cual cada nodo representa una operación como suma, restas, etc. Y cada resultado de operación forma un nuevo tensor.

Para el objeto de este proyecto, no queremos trabajar tan a bajo nivel como Tensorflow nos requiere, es por ello por lo que a priori no es la mejor opción.

7.2.6.2. Keras

Keras es la API de alto nivel oficial de Tensorflow. Se centra mucho más en la experiencia de usuario, manteniendo las propiedades multiplataforma de Tensorflow y haciendo mucho más fácil el desarrollo de modelos. Es por esto por lo que tiene una gran adopción en la industria y en la comunidad.

Keras ofrece APIs simples y consistentes, minimizando el número de acciones que debe hacer el usuario para los casos de uso comunes, y proporciona mensajes de error claros y concisos sobre los errores producidos por el usuario.

Esto hace que Keras sea fácil de usar y fácil de aprender, dejando al usuario centrarse en las ideas y no en la ejecución.

7.2.6.3. Tensorflow Object Detection API

Tensorflow Object Detection API es una librería construida sobre Tensorflow. Se puede usar para hacer detecciones, con cajas delimitadoras, de objetos en imágenes o vídeos. Todo esto sobre una serie de modelos preentrenados que hay disponibles o mediante modelos que podemos crear de cero.

7.3. Estudio de redes convolucionales para la detección y extracción facial en documento de identidad

El objetivo en este apartado es ser capaz de obtener la foto del documento de identidad de este. Este proceso debe darnos como resultado las coordenadas de un cuadrado dentro del cual se encuentre la cara, y ya con estas poder hacer el recorte de la foto original para poder usarla en la segunda parte del proceso.

7.3.1. Estudio y aplicación de técnicas de preprocesado y aumento de datos

De todas las técnicas vistas anteriormente, dos serían las que se podrían aplicar sin poner en riesgo los ejemplos en nuestro caso de uso:

- Volteo: esto puede ser muy útil en nuestra aplicación, ya que, al tener DNI de distintos tipos, y queriendo extraer la foto, podemos encontrarla según en qué tipo en un lado o en otro del DNI, por lo que tener las imágenes volteadas sería una gran forma de aumentar los datos
- Ruido gaussiano: aportaría algo de protección contra el sobreajuste, lo cual nunca está de más.

Por otro lado, se descartan las otras opciones por lo siguiente:

- Rotación: no aplicable pues todas nuestras imágenes del conjunto de datos tienen la misma orientación y modificar esto podría confundir a la red.
- Zoom y traslación: no son válidas para nuestro caso, ya que los bordes del DNI pueden ayudar a la red a encontrar la posición de la cara, por lo que mejor no usar esta técnica

7.3.2. Evaluación de redes neuronales

Como el autor escribía en [1], es posible que gran parte de la culpa de que se estén llegando a números como el 99.63% de precisión FaceNet en reconocimiento facial venga dado por unos pobres test de rendimiento. Para tratar de tener un resultado objetivo, en este proyecto se hará uso de las "COCO Detection metrics" [12]. Estas métricas son ampliamente conocidas y usadas en el ámbito de la detección de objetos usando Tensorflow y son peculiares ya que suelen ser muy estrictas en sus valoraciones. Para poner un ejemplo, modelos de mucho mayor tamaño, suelen tener unos valores como los que se muestran en la Tabla 2 y la Tabla 3:

Tabla 1: Comparativa métricas COCO en distintos tipos de redes

Method	data	Avg. Precision, IoU:			Avg. Precision, Area:			Avg. Recall, #Dets:			Avg. Recall, Area:		
		0.5:0.95	0.5	0.75	S	M	L	1	10	100	S	M	L
Fast [6]	train	19.7	35.9	-	-	-	-	-	-	-	-	-	-
Fast [24]	train	20.5	39.9	19.4	4.1	20.0	35.8	21.3	29.5	30.1	7.3	32.1	52.0
Faster [2]	trainval	21.9	42.7	-	-	-	-	-	-	-	-	-	-
ION [24]	train	23.6	43.2	23.6	6.4	24.1	38.3	23.2	32.7	33.5	10.1	37.7	53.6
Faster [25]	trainval	24.2	45.3	23.5	7.7	26.4	37.1	23.8	34.0	34.6	12.0	38.5	54.4
SSD300	trainval35k	23.2	41.2	23.4	5.3	23.2	39.6	22.5	33.2	35.3	9.6	37.6	56.5
SSD512	trainval35k	**26.8**	**46.5**	**27.8**	**9.0**	**28.9**	**41.9**	**24.8**	**37.5**	**39.8**	**14.0**	**43.5**	**59.0**

Tabla 2: Comparativa métricas COCO en distintos tipos de redes

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [3]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [6]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [4]	Inception-ResNet-v2 [19]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [18]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [13]	DarkNet-19 [13]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [9, 2]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [2]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [7]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [7]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

Si algo llama la atención es que no muchas arquitecturas son capaces de superar los 50 puntos en casi ningún apartado, a pesar de que son arquitecturas probadas, estudiadas y con resultados muy buenos [13] [14]. Esto indica que estas métricas son muy precisas a la hora de encontrar posibles errores.

7.3.2.1. Métricas que proporciona COCO

Expliquemos un poco que significan todos estos valores usados [9] en COCO metrics:

- Intersection over the Union (**IoU**): definido como el área de la intersección dividida por el área de la unión de una caja obtenida de la predicción de un modelo y la caja definida en la etiqueta del ejemplo.
- **Precisión**: definida como el numero de verdaderos positivos dividido por la suma de verdaderos positivos y falsos positivos
- **Recall**: definido como el número de verdaderos positivos divido por la suma del número de verdaderos positivos y falsos negativos

Estos son los conceptos básicos, pero las métricas que nos encontraremos usando el método de evaluación elegido son:

- Average Precision (**AP**): basada en la curva formada por la precisión y el Recall. Básicamente, la media de la precisión es la media de esta para todos los valores de Recall.
- Mean Average Precision (**mAP**): mientras que AP se basa para una única clase de detección, mAP es AP para múltiples clases de detección.
- Average Recall (**AR**): es una métrica numérica para compara el rendimiento de un detector. Básicamente es la media del Recall sobre toda la IoU contenida entre 0.5 y 1.0. Usando COCO Metrics se calculará para cada clase de detección.

  Usando COCO Metrics se calculan los valores para distintos valores de threshold, entre 0.5 y 1.0.

Conociendo todos estos conceptos, podremos ser objetivos a la hora de evaluar si nuestro rendimiento es aceptable para el caso de uso propuesto, pues, no sólo tendremos una serie de valores de rendimiento de nuestro modelo, si no que tendremos un punto de comparación con la Tabla 2 y la Tabla 3 para saber en qué puntos nuestra red es mejor y cuales peor.

7.4. Redes convolucionales para el cálculo de correspondencia facial

El objetivo en este apartado es encontrar una arquitectura de red capaz de tener como entrada dos imágenes, una la imagen extraída del documento de identidad, y otra una imagen de la misma persona. Y que la salida sea un valor que nos indique la similitud de las personas que hay en ambas fotos.

7.4.1. Redes siamesas

La teoría relacionada con este tipo de arquitecturas ha sido encontrada y estudiada previamente en [4].

Una red siamesa es una arquitectura de red neuronal avanzada. Se denominan siamesas por que tienen dos o más subredes internas iguales. Una red siamesa es algo como lo que se muestra en la Figura 15:

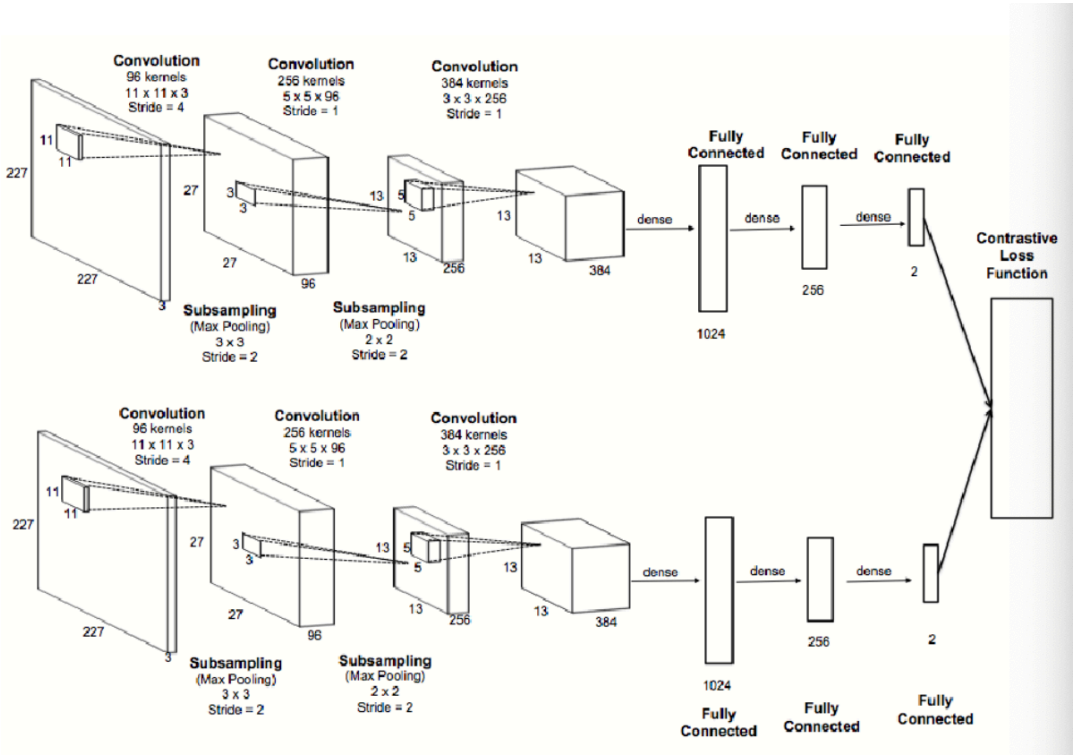


Figura 15: Ejemplo red siamesa

Es importante saber que no sólo las arquitecturas de las subredes son idénticas, si no que también los pesos deben ser compartidos entre ellas para que la red pueda ser llamada siamesa.

La idea principal tras las redes siamesas es que pueden aprender descriptores de información útiles que pueden ser usados más adelante para comparar las dos entradas de las respectivas subredes.

7.5. Implementación de redes neuronales en Android

El core de Tensorflow está escrito en C++. Como ya sabemos, Android está escrito en Java, por lo que, para poder hacer una aplicación Android que use un modelo de Tensorflow tendremos que hacer uso de JNI (Java Native Interface) para llamar a las funciones C++ como pueden ser loadModel o getPredictions.

Tendremos que generar un fichero con extensión **.so** (shared object) que es un archivo C++ compilado y un archivo con extensión **.jar** el cual consistirá en la API Java que hará las llamadas a el código nativo de C++.

Con esto haremos las llamadas a la API Java para realizar las acciones fácilmente.

7.5.1. Proceso de cuantización de redes

La teoría detrás del proceso de cuantización de una red neuronal se puede encontrar en [6]

A la hora de implementar una red neuronal en una aplicación, lo ideal sería no afectar al tamaño de la aplicación, lo cual es complicado si tenemos que incluir el modelo en la aplicación.

Este peso se reduce si cambiamos todos los pesos y parámetros del modelo de coma flotante con muchísima precisión, a pesos y parámetros con muy poca precisión (1bit).

En la práctica las variables no se cambian a 1 bit de precisión, se modifican a, por ejemplo 8 bits, ya que 1 bit modificaría considerablemente el rendimiento del modelo.

Como consecuencia, no sólo el peso de la red será menor, también mejorará el uso de memoria, y reducirá en gran parte el consumo de potencia.

Todo este proceso se realiza teniendo en cuenta que la precisión del modelo no puede variar mucho. Estas técnicas de cuantización se pueden realizar durante el proceso de entrenamiento o post entrenamiento.

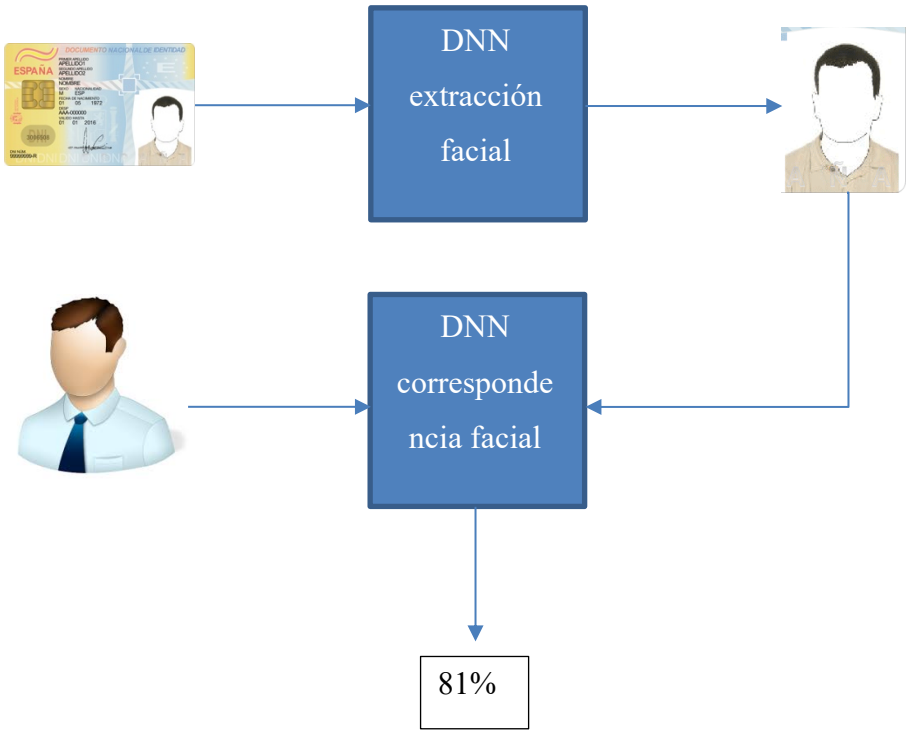
Si se cuantiza el modelo post entrenamiento, se puede producir perdida de precisión, particularmente en redes pequeñas.

8. IMPLEMENTACIÓN Y DESARROLLO

8.1. Planteamiento

Se plantea el desarrollo en dos fases:

- Extracción de la foto de un documento de identidad.
- Uso de la foto extraída del documento de identidad y compararla con una foto que haga el usuario en el momento.



8.2. Desarrollo de una red convolucional para la detección y extracción facial en documento de identidad

8.2.1. Conocimiento y evaluación de redes propuestas

Al inicio del desarrollo del TFG, se me propusieron una serie de redes ya pre-entrenadas que podrían serme de ayuda en el desarrollo de este trabajo. Todas estas son modelos que han sido entrenados haciendo uso de la Object Detection API de

Tensorflow. Esto nos viene muy bien ya que tendremos por defecto la detección con cajas contenedoras como necesitamos.

Los parámetros a los que vamos a dar preferencia a la hora de hacer la elección del modelo, teniendo siempre en cuenta que están destinadas a entornos móviles serían:

- Tamaño (en bytes) del modelo.
- Precisión.
- Tiempos de respuesta.

En esta comparativa no vamos a comparar las arquitecturas internas de los modelos pues nuestro objetivo no es modificar las redes de ninguna forma, si no a partir de los entrenamientos ya realizados sobre los modelos y reentrenarlos con nuestros datos, aprovechando todos los patrones que hayan sido capaces de aprender todas las neuronas de la red y que sea sólo la última capa la que modifique sus valores para detectar el objeto que queremos.

Para calcular la precisión de las redes, aplicado a nuestro caso de uso, se ha realizado un entrenamiento muy corto para ver cual se adaptaba mejor. Este entrenamiento constaba de 100 pasos, una tasa de aprendizaje de 0.004 y un tamaño de lote de 2.

Tabla 3: Comparativa arquitecturas

Modelo preentrenado	**Tiempo de inferencia**	**Precisión**	**Tamaño modelo**
Faster-inception-v2	0.9s	**0.71**	57,2 MB
Faster-ResNet50	0.2s	0.69	120,5 MB
Mask-Inception-v2	1.5s – 2s	0.61	67,1 MB
SSD-InceptionV2	0.26s	0.69	102 MB
SSD-Mobilenet-v1	0.35s	0.63	29,1 MB
SSD-Mobilenet-v2	**0.12s**	0.68	**19,1 MB**

Con los datos mostrados en la Tabla 1 podemos tomar una decisión. La arquitectura elegida para realizar el proceso de entrenamiento es la **SSD-Mobilenet-v2**.

Es elegida por los siguientes motivos:

- Es la más ligera de todas, lo cual será una ventaja al incluirla en las aplicaciones que queramos. No es viable añadir un modelo de 100MB a una aplicación, sin tener en cuenta el altísimo tiempo de respuesta que tendría un modelo de este tipo en un móvil con menor potencia de cálculo que un ordenador.
- El tiempo de inferencia es el menor, lo cual tiene sentido teniendo en cuenta que es la de menor tamaño, por lo tanto, la que menos capas tiene a la hora de pasar la imagen a través de ella.
- La precisión comparada con las otras no es muy inferior, a penas 0.3 con la que mejores valores ha obtenido, con lo que cuando entrenemos este valor debería mejorar mucho.

8.2.2. Preparación y anotación de una base de datos

Para el desarrollo de este proyecto se ha empleado una base de datos de imágenes de documentos de identidad españoles propiedad de Mobbeel.

Entre estas imágenes se pueden encontrar distintas versiones de este, en distintos formatos y orientaciones.

Este conjunto está forma por un 10% de DNI v1, 35% DNI v2 y 55% DNI v3. Tanto cara frontal como trasera.

8.2.2.1. Normalización de los datos.

Para poder obtener buenos resultados, se normalizan los datos de entrada. Observando el conjunto saco la conclusión de que la mayoría de las imágenes que tenemos tienen un ancho de 640 px y un alto de 403 px. En la mayoría de estas imágenes se encuentra recortado el documento de identidad, sin dejar ver nada más en la imagen, sin embargo, en otras se puede ver algo de fondo en las mismas, sobre todo en aquellas que fueron tomadas en orientación vertical.

Para obtener un conjunto de datos homogéneo, lanzo un primer script que elimine todas las fotos cuyo alto sea mayor que el ancho, de esta forma eliminamos las fotos con orientación vertical.

Una vez hecho esto, haciendo uso de OpenCV se reescalan todas las imágenes a un ancho de 640 px, dejando el alto libre para que no pierdan la proporción de alto/ancho original y no se deformen las imágenes de las caras.

Tras esto, se hace una inspección manual de las imágenes, borrando todas aquellas que se hayan podido escapar en pasos anteriores. Todas tiene el ancho requerido y todas ellas varían entre 400 px de alto y 405 px de alto. Llegados a este punto, decido que, en vez de recortar las imágenes, es preferible hacer un reescalado de estas, ya que siendo pocos píxeles de diferencia entre ellas apenas habrá deformación en las caras finales y conservo los bordes del documento que la red neuronal puede aprovechar a posteriori.

Una vez finalizado el proceso acabamos con una base de datos de 3135 imágenes homogéneas, que tienen tanto el mismo contenido como mismo tamaño de imagen.

8.2.2.2. Etiquetación de la base de datos.

Para etiquetar los ejemplos que tenemos, hago uso de un programa Open Source llamado LabelImg. Este programa permite abrir un directorio de imágenes, visualizar una por una, y mediante un visor, marcar con cajas los elementos que queramos en la imagen, etiquetándolo con un nombre, también a nuestra elección. Al pasar a la siguiente imagen, el programa genera un archivo XML con las coordenadas superior izquierda e inferior derecha de la caja marcada.

En este caso, marco la foto del DNI en cada imagen, asignándole la etiqueta document_face. Este proceso debe hacerse a mano y para todas las imágenes que tenemos en la base de datos.

El uso de este programa fue clave a la hora de optimizar el tiempo de etiquetación de la base de datos, reduciéndolo de lo que habrían sido días de tedioso trabajo en horas, gracias a los distintos comandos de teclado que proporciona y que permite realizar cualquier acción con una mano mientras con la otra se seleccionan las cajas contenedoras en la imagen haciendo uso del ratón.

8.2.3. Generación conjunto entrenamiento y conjunto de validación.

Para el proceso de entrenamiento necesitamos 2 conjuntos de datos. Uno con el que realizar el entrenamiento y otro con el que ir validando el proceso.

Para ello hago uso del algoritmo "xml_to_csv.py" que se encuentra en [7] y paso todas las etiquetas de xml a un único CSV.

Una vez tenemos el CSV con las etiquetas, podemos proceder a realizar la separación de los conjuntos. Usamos el algoritmo "Split labels.ipynb" que podemos encontrar en [7].

Hay que modificar este algoritmo con los porcentajes que queremos para cada conjunto. En nuestro caso queremos un 80% de las imágenes en el conjunto de entrenamiento y el 20% restante en el conjunto de validación. Es importante también que indiquemos que queremos que se seleccionen las imágenes de cada conjunto aleatoriamente y no sean por orden.

El resultado serán 2 archivos CSV, uno con las etiquetas de entrenamiento y otro con las etiquetas de validación.

8.2.3.1. Generación archivos TFRecord.

La API "Object-Detection API" [8] de TensorFlow acepta como entrada archivos TFRecord que incluyen las imágenes y las etiquetas en el mismo archivo. Nosotros tenemos los datos separados, por un lado, el conjunto de imágenes y por otro, los archivos CSV de las etiquetas. Necesitamos un archivo TFRecord para el conjunto de entrenamiento y otro para el conjunto de validación.

Para poder generar estos archivos, hago uso de el algoritmo "generate_tfrecord.py" que se puede encontrar en [7]. Al algoritmo hay que pasarle el archivo CSV de las etiquetas y el directorio donde se encuentran las imágenes, y el nombre del fichero de salida que queramos.

8.2.4. Creación fichero labelmap

A la hora de hacer la transferencia de conocimiento, tenemos que indicar cuáles serán las nuevas etiquetas que tiene que reconocer el modelo, ya que el modelo preentrenado reconoce multitud de objetos.

Esto se hace creando un nuevo fichero llamado "labelmap.pbtxt" en el que se indica mediante objetos JSON el id que queramos darle a la etiqueta en cuestión y el nombre de esta etiqueta. Es muy importante que esta etiqueta coincida con el nombre que pusimos en LabelImg a las cajas marcadas de las imágenes.

8.2.5. Entrenamiento.

8.2.5.1. Configuración del entrenamiento

Junto con el modelo preentrenado, había un fichero llamado "pipeline.config".

En este fichero es donde se configura todo el entrenamiento del modelo. Está dividido en tres partes, una es la parte de configuración del modelo como tal, otra es la parte de configuración de hiperparámetros de entrenamiento y la última es la parte de configuración de la evaluación del modelo.

De la configuración del modelo, en este caso tenemos que modificar:

- Num_classes: indica el número de clases que va a identificar el modelo, nosotros sólo estamos interesados en obtener las fotos de las caras de los DNI, por lo que en nuestro caso será un 1.

Dentro de la configuración del modelo, podemos modificar independientemente los valores de regularización y activación tanto de el extractor de características como de el predictor de las cajas. Para este caso no ha sido necesario modificar estos valores.

También se puede modificar dentro de "post_processing" el algoritmo de Non Maximum Suppresion usado en la predicción de las cajas, por si fuera necesario.

En la configuración del entrenamiento los hiperparámetros utilizados finalmente han sido:

- Batch_size: 15

- Learning_rate: configurado para ir decayendo exponencialmente durante el entrenamiento. Empieza en 0.004.
- Num_steps: 8000.
- Fine_tune_checkpoint: aquí debemos indicar la ruta donde se encuentra el "model.ckpt" del modelo preentrenado que estamos reentrenando.

En la configuración de la evaluación lo único que ha sufrido modificación ha sido el número de ejemplos, que hay que indicar el número que tengamos.

Finalmente, tanto en el lector de entrada del entrenamiento, como en el lector de entrada de la evaluación, hay que indicar lo mismo. La ruta donde se encuentra el "labelmap.pbtxt" creado anteriormente y la ruta donde se encuentra el TFRecord a usar, cada uno el suyo.

8.2.5.2. Proceso de entrenamiento

Una vez está todo configurado, haciendo uso de "Object Detection API" de TensorFlow, que se puede encontrar en [8]. Se procede con el entrenamiento.

El proceso de entrenamiento ha sido realizado usando el algoritmo "train.py" que se encuentra dentro de la carpeta Legacy de [8]. No se usa la nueva versión de este algoritmo por que el modelo preentrenado usado se entrenó con este algoritmo y no es compatible con el nuevo.

Para ir vigilando el entrenamiento, se hace uso de Tensorboard, con el cual tenemos acceso a los datos en tiempo real de valores de pérdida, precisión de las cajas y demás.

8.2.6. Evaluación de resultados

Tras terminar el proceso de entrenamiento, tendremos en el directorio de entrenamiento varios "checkpoint" del proceso. Habrá que hacer una elección de qué "checkpoint" queremos convertir a modelo y por lo tanto utilizar finalmente. Para esto podemos fijarnos en los datos de pérdida y precisión que hemos ido obteniendo durante el entrenamiento y observando en Tensorboard.

Pero si queremos ser un poco más objetivos, podemos usar el algoritmo incluido en el repositorio "Object Detection API" [8] de Tensorflow, concretamente el algoritmo "eval.py" de Legacy dentro del paquete "Object Detection". Este algoritmo hace uso de la configuración de evaluación indicada en el archivo "pipeline.config" del modelo.

8.2.6.1. Prestaciones

Usando Tensorboard se puede obtener la pérdida durante el entrenamiento con exactitud. En el caso de los modelos entrenados usando "Object Detection API" de Tensorflow [8], obtenemos dos valores de pérdida pues son dos las características que buscamos en estos modelos.

Sin embargo, dadas las características de nuestro caso de uso, una de las dos nos es indiferente, la pérdida en clasificación (Figura 16), no es relevante puesto que sólo tenemos un tipo de objeto a encontrar. Este valor sería muy importante si quisiéramos detectar múltiples objetos dentro de la misma imagen.

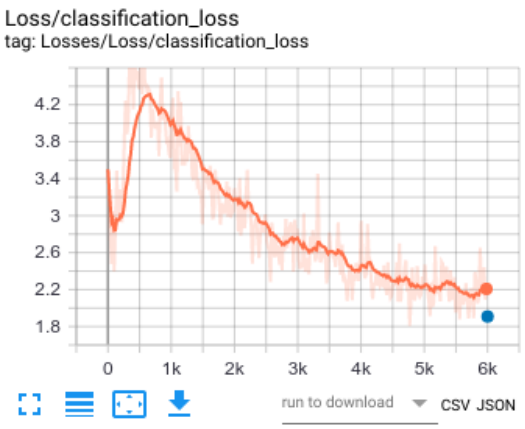


Figura 16: Pérdida en clasficación.

Mientras que la pérdida de localización si que es muy importante para nosotros ya que, es lo importante en nuestro caso de uso, cómo de bien localiza la cara dentro del documento.

Como se ve en la Figura 17, la pérdida baja mucho hasta quedarse estabilizada con ligeras fluctuaciones en torno a los 0.10 puntos de pérdida, bastante más bajo que los 0.50 puntos de pérdida que tenía el modelo al comenzar el entrenamiento.

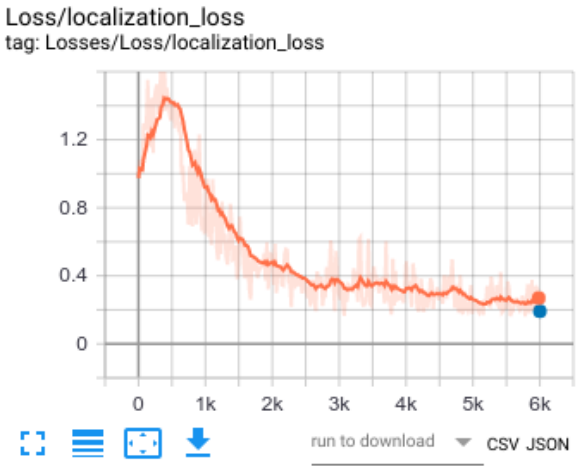


Figura 17: Pérdida en localización.

No sólo se obtienen valores de prestaciones desde Tensorboard, uno de los más importantes como es la precisión viene dada por las COCO Metrics [12]. Los valores obtenidos de la evaluación del modelo después del reentrenamiento son los que se pueden observar en la Tabla 4.

Tabla 4: Prestaciones modelo entrenado

AP 0.5 – 1.0	0.655
AP 0.5	0.867
AP 0.75	0.838
AR 0.5 - 1.0 maxDets: 1	0.801
AR 0.5 - 1.0 maxDets: 10	0.802
AR 0.5 - 1.0 maxDets: 100	0.803
mAP	0.655365
mAP 0.50	0.866728

mAP 0.75	0.837733
Classification_Loss	1.908290
Localization_Loss	0.191834

Tomando como referencia los valores vistos en la Tabla 3, donde se mostraban los valores de las métricas para una serie de arquitecturas conocidas, podemos decir que hemos obtenido unos buenos resultados.

Si nos fijamos los valores de AP y mAP son los mismos, esto se debe a que mAP es la media de AP para todos los objetos identificados. En nuestro caso sólo identificamos uno, por lo que tiene sentido esta igualdad.

Sabiendo que AP y mAP miden qué tan bien las predicciones son precisas, podemos ver que para valores de IoU de 0.5 y 0.75 los resultados de 0.86 y 0.83 (sobre 1) respectivamente son muy buenos comparados con los 61.1 y 44.1 (sobre 100) obtenidos por RetinaNet [14] usando las mismas métricas de evaluación.

Si nos fijamos ahora en los valores de AR, que indican la precisión que tiene el modelo en detectar todos los casos positivos, vemos que obtenemos un buen resultado también. Pues todos los campos están en 0.8 (sobre 1) y en comparación, una arquitectura como SSD512 [13] mucho más pesada y compleja, se queda en 24.8, 37.5 y 39.8 (sobre 100) respectivamente.

8.2.7. Generación grafo congelado

Teniendo el "checkpoint" seleccionado, deberemos generar el archivo "Protobuf" del modelo. Este fichero agrupa en sí mismo tanto el modelo como los pesos de cada variable del modelo, por lo que será este único fichero el necesario a la hora de ejecutar pruebas del modelo.

Para la generación de este fichero se hace uso del algoritmo "export_inference_graph.py" que hay dentro de la "Object Detection API" de Tensorflow. Basta con indicar el directorio donde se encuentra el archivo "pipeline.config" del entrenamiento, el prefijo del "checkpoint" elegido y el directorio de salida de este.

8.2.8. Tiempos de respuesta

Una vez tenemos el modelo congelado, podemos usar un script para probar la inferencia y visualmente evaluar el rendimiento de la red.

Hay que tener en cuenta que, ya que hasta el momento no hay una aplicación móvil en la que probar el modelo, las pruebas de tiempo de respuesta han sido realizadas en un ordenador portátil con las siguientes características:

- CPU: Intel Core i5, 2,6 GHz con dos 2 núcleos físicos y 4 lógicos.
- GPU: Integrada con la CPU Intel Iris.
- RAM: 8GB ram DDR3 1600MHz

El tiempo de respuesta final obtenido es de 0.15s por imagen evaluada.

8.2.9. Umbral

Es importante que a la hora de realizar un script con el que probar de forma práctica el desempeño del modelo, se tenga en cuenta que el modelo retorna un conjunto de predicciones, en nuestro caso 100.

Cuando el modelo devuelve estas predicciones, no sólo devuelve las coordenadas de la caja contenedora del objeto en cuestión, también devuelve un valor de la certeza de que el objeto detectado sea el que tiene que ser.

Es con este valor con el que tenemos que jugar, podríamos elegir como resultado la predicción con mayor certeza, sería una solución, sin embargo, se para esta implementación se ha seguido otro camino.

Una vez que el modelo devuelve el conjunto de 100 predicciones sobre la imagen, usamos un valor umbral para descartar todas aquellas que sean demasiado bajas, en este caso concreto 0.7 ha sido el punto en el cual ha rendido mejor.

Esto nos decrementará muchas de esas 100 predicciones originales, pero seguimos teniendo unas cuantas. En este punto aplicamos un algoritmo de NMS [15] para con todas esas cajas finales muy próximas todas ellas a una caja perfecta, generar una única que sea la “media” de todas.

Esta ha sido la mejor solución que he encontrado para tratar con todas las predicciones devueltas por el modelo al hacer una predicción.

Con esto también conseguimos protegernos contra los casos en los que no haya caras, ya que detectará una caja en cualquier parte de la imagen, pero con un porcentaje de certeza muy bajo, por lo que no se tendrá una caja contenedora en esa imagen.

8.3. Desarrollo de una red siamesa para la correspondencia facial.

Para nuestro caso de uso, es posible que, en vez de una red siamesa, sea conveniente hacer una red pseudo-siamesa.

La idea es que ambas subredes (serían 2 pues tenemos dos imágenes de entradas) compartieran la misma arquitectura, dos Mobilenet, por ejemplo, pero que no compartan los pesos, pues las dos imágenes no son iguales, una será una imagen de un documento de identidad, en blanco y negro y la otra será muy probablemente una foto en color y hecha con un móvil. De esta forma cada subred podría adaptarse a la imagen de entrada que tenga.

Finalmente coger una capa completamente conectada a las salidas de las dos subredes y que esta nos de un valor de confianza sobre si son las mismas personas o no, usando una regresión logística.

Para el desarrollo de esta red se utilizaría Keras, pues no se han encontrado redes siamesas preentrenadas y no nos son necesarias las cajas contenedoras que si que necesitábamos anteriormente.

9. RESULTADOS Y DISCUSIÓN

En este punto se presentan los resultados obtenidos durante el desarrollo del proyecto.

Tras el entrenamiento y evaluación de la red entrenada, el siguiente paso es probar la red con ejemplos no etiquetas que la red nunca haya visto antes.

Para ello se crea una nueva base de datos, igualmente propiedad de Mobbeel, con la que poder realizar la prueba de la red. Este proceso es idéntico que el realizado a la hora de obtener los conjuntos de entrenamiento y evaluación del proceso de entrenamiento del modelo.

Por motivos de confidencialidad no puedo mostrar las pruebas, pero el resultado es realmente bueno. En todo el conjunto de prueba no se ha replicado ningún fallo de la red, en todos los casos probados se ha detectado una cara, con muy buena precisión en la localización. Todas las cajas contenedoras contenían las caras con márgenes aceptables, es decir, apenas se deja un margen de entre 10 y 15 píxeles de ancho/alto alrededor de la cara objetivo.

Ya que la red ha sido entrenada con las imágenes propiedad de Mobbeel, ni mi DNI ni mi permiso de conducción se encontraban entre ellas. Por lo que puedo enseñar una predicción sobre mi propio permiso de conducción para probar la red, y mostrar los resultados.

Los resultados son los siguientes:

Figura 18: Ejemplo detección sobre permiso de conducción

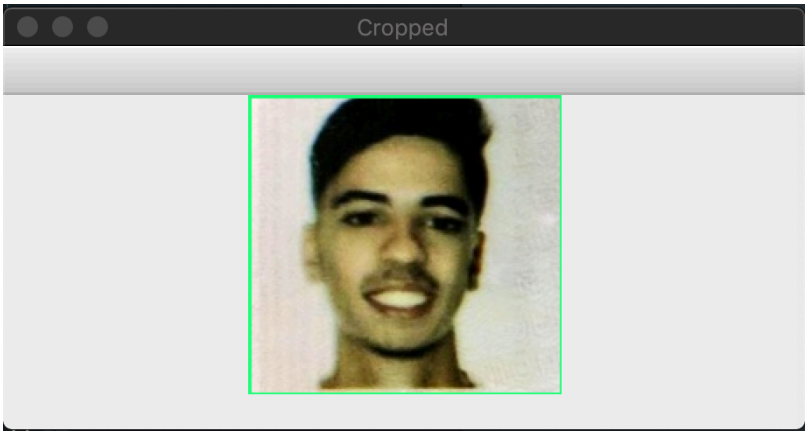


Figura 19: Ejemplo de recorte de detección.

El script de prueba utilizado aplica el método del umbral descrito con anterioridad, ese es el motivo por el cual vemos una única caja contenedora como resultado.

10. CONCLUSIONES Y TRABAJO FUTURO

Como conclusión general, se puede decir que se a lo largo del desarrollo del proyecto, a parte de conseguir los conocimientos partiendo de cero, se ha logrado desarrollar parte de un sistema de verificación de identidad.

Se ha podido comprobar que el trabajo con redes neuronales no es sencillo, y que los procesos de entrenamientos son largos y hay que hacer muchas pruebas hasta dar con los hiperparámetros claves para cada caso de uso.

En lo que respecta a lo desarrollado, los resultados obtenidos son satisfactorios para el caso de uso propuesto.

Si nos fijamos en el resto del estudio, considero que se ha llegado a un punto en el que la implementación no debería ser demasiado compleja siempre que se dispongan de los datos necesarios para el entrenamiento.

De manera más parcial se pueden destacar los siguientes logros:

- Se ha realizado con éxito el estudio de sistemas de AA, desde lo más básico y simple hasta arquitecturas complejas.
- Se ha conseguido desarrollar el estudio y decisiones sobre las mejores formas de realizar una red neuronal que extraiga caras en documentos de identidad.
- Se han obtenido buenos resultados en el entrenamiento de una red neuronal para extracción facial de un documento de identidad.
- Se ha conseguido desarrollar el estudio sobre una posible solución a el problema planteado de correspondencia facial entre dos fotos de un mismo usuario.

Para finalizar, se proponen los siguientes puntos como trabajo futuro:

- Realizar un entrenamiento con mayor cantidad de datos y horas.
- Generación y proceso de etiquetar un conjunto de datos para la correspondencia facial.
- Entrenamiento de una red neuronal siamesa que siga las indicaciones y siguiendo las conclusiones a las que se han llegado durante la realización de este proyecto
- Creación de la aplicación Android que integre todo el sistema.

11. BIBLIOGRAFÍA

[1] BALABAN, S., 2015. Deep Learning and face recognition: the state of the art [en línea]. [Consulta: 7 March 2019]. Disponible en: https://arxiv.org/pdf/1902.03524.pdf

[2] SCHROFF, F., KALENICHENKO, D., PHILBIN, J., 2015. FaceNet: A Unified Embedding for Face Recognition and Clustering. [En línea]. [Consulta: 4 March 2019]. Disponible en: https://arxiv.org/pdf/1503.03832.pdf

[3] ZHAO, Z., ZHENG, P., XU, S., WU, X., 2019. Object Detection with Deep Learning: A Review. [En línea]. [Consulta: 28 April 2019]. Disponible en: https://arxiv.org/pdf/1807.05511.pdf

[4] BROMLWY, J., GUYON, I., LECUN, Y., SÄCKINGER, E., SHAH, R., 1994, Signature Verification using a "Siamese" Time Delay Neural Network. [En línea]. [Consulta: 19 March 2019]. Disponible en: http://papers.nips.cc/paper/769-signature-verification-using-a-siamese-time-delay- neural-network.pdf

[5] SURYANSH, S., 2018, Gradient Descent: All You Need to Know. *Hackernoon* [En línea]. [Consulta: 12 March 2019]. Disponible en: https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

[6] HUBARA, I., COURBARIAUX, M., SOUDRY, D., EL-YANIV, R., BENGIO, Y., 2018, Quitized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. [En línea]. [Consulta: 3 April 2019]. Disponible en: http://www.jmlr.org/papers/volume18/16-456/16-456.pdf

[7] DATITRAN, 2017, Raccoon Detector Dataset. *Github* [En línea]. [Consulta: 1 de April 2019]. Disponible en: https://github.com/datitran/raccoon_dataset

[8] TENSORFLOW, 2017. Object Detection API. *Github* [En línea]. [Consulta: 3 April 2019]. Disponible en: https://github.com/tensorflow/models/tree/master/research/object_detection

[9] UI, JONATHAN, 2019. mAP (mean Average Precision) for Object Detection. *Medium* [En línea]. [Consulta: 8 April 2019]. Disponible en: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

[10] ADEXT, 2017. ¿Qué es machine learning? [Guía completa para principiantes]. *Adext Blog* [en línea]. [Consulta: 1 April 2019]. Disponible en: https://blog.adext.com/es/machine-learning-guia-completa

[11] An Introduction to Machine Learning Theory and Its Applications: A Visual Tutorial with Examples. *Toptal Engineering Blog* [en línea], 2014. [Consulta: 2 March 2019]. Disponible en: https://www.toptal.com/machinelearning/machine-learning-theory-an-introductory-primer

[12] COCO, 2015. Detection Evaluation. *CocoDataset* [en línea]. [Consulta: 1 May 2019] Disponible en: http://cocodataset.org/#detection-eval

[13] LIU, W., ANGUELOV, D., ERHAN, D., SZEGEDY, C., REED, S., FU, C., C. BERG, A., 2016, SSD: Single Shot Multibox Detector. [en línea]. [Consulta: 13 March 2019]. Disponible en: https://arxiv.org/pdf/1512.02325.pdf

[14] LIN, T., GOYAL, P., GIRSHICK, R., HE, K., DOLLAR, P., 2018, Focal Loss for Dense Object Detection. [en línea]. [Consulta: 13 March 2019]. Disponible en: https://arxiv.org/pdf/1708.02002.pdf

[15] ROSEBROCK, A., 2014, Non-Maximum Suppresion for Object Detection in Python. *PyImageSearch.* [en línea]. [Consulta: 16 March 2019]. Disponible en: https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python

[16] SUPERDATASCIENCE TEAM, 2018, Recurrent Neural Networks (RNN) – Long Short Term Memory (LSTM). *SuperdataScience.* [en línea]. [Consulta 13 March 2019]. Disponible en: https://www.superdatascience.com/blogs/recurrent-neural-networks-rnn-long-short-term-memory-lstm/

[17] SUPERDATASCIENCE TEAM, 2018, Recurrent Neural Networks (RNN) – The Vanishing Gradient Problem. *SuperdataScience.* [en línea]. [Consulta: 13 March 2019]. Disponible en: https://www.superdatascience.com/blogs/recurrent-neural-networks-rnn-the-vanishing-gradient-problem/

[18] A Beginner's Guide to Backpropagation in Neural Networks – A.I Wiki. *Skymind.ai* [en línea] [Consulta: 19 April 2019]. Disponible en: https://skymind.ai/wiki/backpropagation

[19] ROUSE, M., 2018, Deep Learning (Deep neural network). *SearchEntrepriseAI.* [en línea]. [Consulta: 13 April 2019]. Disponible en:

https://searchenterpriseai.techtarget.com/definition/deep-learning-deep-neural-network

[20] Deep Learning Tutorial for Beginners: Neural Network Classification – Guru99. *Guru99.com* [en línea] [Consulta: 13 April 2019]. Disponible en: https://www.guru99.com/deep-learning-tutorial.html

[21] THOMAS, C., 2019.The basics of Deep Neural Networks. *TowardsDataScience* [en línea]. [Cosulta: 13 April 2019]. Disponible en: https://towardsdatascience.com/the-basics-of-deep-neural-networks-4dc39bff2c96

[22] SHARMA, P., 2018. An Introductory Guide to Deep Learning and Neural Networks. *Analytics Vidhya* [en línea]. [Consulta: 16 April 2019]. Disponible en: https://www.analyticsvidhya.com/blog/2018/10/introduction-neural-networks-deep-learning/

[23] LIANG, J., 2018, An Introduction to Deep Learning. *TowardsDataScience* [en línea]. [Consulta: 16 April 2019]. Disponible en: https://towardsdatascience.com/an-introduction-to-deep-learning-af63448c122c

[24] DWIVEDI, D., 2018, Machine Learning For Beginners. *TowardsDataScience* [en línea]. [Consulta: 27 February 2019]. Disponible en: https://towardsdatascience.com/machine-learning-for-beginners-d247a9420dab

[25] KUNAL, J., 2015. Machine Learning Basics for a newbie. *Analytics vidhya* [en línea]. [Consulta: 27 February 2019]. Disponible en: https://www.analyticsvidhya.com/blog/2015/06/machine-learning-basics/

[26] MCCREA, N., 2014, An Introduction to Machine Learning Theory and Its Applications: A Visual Tutorial with Examples. *Toptal Developers* [en línea]. [Consulta: 3 March 2019]. Disponible en: https://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer

[27] TAHSILDAR, S., 2018, Machine Learning Basics. *Quant Insti Blog* [en línea]. [Consulta: 3 March 2019]. Disponible en: https://blog.quantinsti.com/machine-learning-basics/

[28] BROWNLEE, J., 2015, Basics Concepts in Machine Learning. *MachineLearningMastery* [en línea]. [Consulta: 5 March 2019]. Disponible en: https://machinelearningmastery.com/basic-concepts-in-machine-learning/

12. ANEXOS

12.1. Planificación del proyecto

La idea de colaboración con la empresa Mobbeel para el desarrollo del proyecto apareció por el mes de febrero. La planificación inicial fue destinar un mes y medio para la fase de estudio y desarrollo, dejando un mes para la realización de la memoria, el cual se vio reducido debido a exámenes.

Se han realizado varias reuniones con el tutor en las que se compartían los problemas que se iban encontrando a lo largo del desarrollo del modelo. Se habla solo del desarrollo del modelo ya que durante la fase de estudio estas reuniones no se llevaron a cabo, ya que había muchos conocimientos aún por asimilar.

Se realizó una reunión inicial en la cual se compartía con el tutor el objetivo del proyecto, una vez que se tuvo esto en claro, se definió un poco la hoja de ruta a seguir durante la fase de estudio.

Una vez se comenzó el desarrollo se llevaron a cabo 4 reuniones una a la semana en la que se compartía el avance realizado y los problemas surgidos a la hora del desarrollo y entrenamiento del modelo.

Aún así, se ha mantenido conversación en la plataforma Trello, en la cual se exponían dudas y avances que se iban haciendo, en la siguiente tabla se tienen en cuenta tanto las reuniones presenciales, como las conversaciones en dicha plataforma:

Tabla 5: Resumen de la planificación del proyecto

Fase	Reunión	Tema tratado
Estudio	06/02	Reunión inicial para planteamiento del proyecto.
	11/02	Comienzo estudio aprendizaje automático.
	20/02	Fin curso, y puesta en

		común de los conocimientos adquiridos
Desarrollo	20/02	Desarrollo ejemplos simples
	01/03	Comienzo de búsqueda de las alternativas existentes para reconocimiento facial
	06/03	Descarte de las soluciones FaceNet por las versiones de Tensorflow usadas para su entrenamiento.
	12/03	Encontrado nuevo repositorio con modelos preentrenados y uso de estos en el módulo DNN de OpenCV 4
	04/03	Primeros problemas con los datos de entrada y comienzo del preprocesamiento de datos
	12/03	Etiquetado de bases de datos finales
	19/03	Problemas de tiempo de inferencia con los modelos en OpenCV 4
	21/03	Cambio al uso de Tensorflow. Inferencias de 0.1s. Se empieza a usar NMS.
	4/04	Primeros resultados de

		entrenamiento. Insatisfactorios.
	9/04	Normalización de las imágenes para mejorar el entrenamiento
	21/05	Resultados finales detección facial.
	22/05	Comienzo estudio correspondencia facial
	29/05	Se termina el estudio y se concluye que es suficiente trabajo realizado para el TFG.
Documentación	22/06	Comienzo de la memoria.
	13/07	Revisión de memoria.
	16/07	Finalización de la memoria.

12.2. Scripts utilizados en el proyecto

Los scripts utilizados en la evaluación del proyecto y el script utilizado de NMS [15] se encuentra en un repositorio propio. Disponible en:
https://github.com/rrodriguze/DNN_Visual_Evaluation_Script