

Proyecto final:

Redes Siamesas para identificación de habitaciones Airbnb



Nombre Alumno 1º: Antonio José Aroca Aguilar

Nombre Alumno 2º: José Antonio Díaz Ramírez

E-mail 1º: antonioaroca@correo.ugr.es

E-mail 2º: joseadr@correo.ugr.es

Asignatura: Visión por Computador

Profesor: Nicolás Pérez de la Blanca Capilla

Curso: 2020/21

Índice

Problema y su enfoque	2
Metodología aplicada	2
Obtener información del Dataset	2
Parejas	3
Submodelos	5
Submodelo 1	5
Submodelo 2	6
Submodelo 3	6
Modelo Siames	6
Predicción	8
Resultados finales	10
CONCLUSIÓN	16
Bibliografía	17

Problema y su enfoque

El problema a tratar consiste en un conjunto de imágenes de distintas habitaciones extraídas de la página web Airbnb, dedicada al alquiler de viviendas, gracias a un Dataset ya creado por un usuario en Kaggle. Este dataset contiene imágenes de distintas habitaciones en pisos de diferentes ciudades, las cuales pueden tener 1,2,3,4,5 o hasta 6 imágenes tomadas de una misma habitación en distintas perspectivas. El objetivo es entrenar una red siamesa que sea capaz, dada una pareja de imágenes del conjunto, determinar si pertenecen a la misma habitación o no, usando como modelos de redes los implementados a lo largo del curso y programar la ,ya estudiada en la parte teórica de la asignatura, red siamesa.

Para ello adaptaremos la base de datos a nuestro problema formando las parejas necesarias, crearemos varios submodelos para comprobar cuánta efectividad tiene cada uno de ellos a la hora de predecir el resultado, e intentar mejorarlo usando técnicas empleadas para las redes neuronales.

Metodología aplicada

Obtener información del Dataset

La información obtenida en el Dataset viene distribuida en dos carpetas, Training y Test, las cuales, a su vez, se ramifican en varias carpetas que albergan muchas imágenes. Estas carpetas presentan como nombre el tipo de habitación de las imágenes que se incluyen en ellas, por ejemplo, "backyard", "basement". Las imágenes están nombradas siguiendo un patrón claro, los nombres están formados de la siguiente forma: "ciudad_codigohabitacion_nºimagen", siendo el código de habitación único para cada habitación distinta y el número de imagen se refiere al número de foto captada sobre la misma habitación.

Para realizar una correcta lectura de estas imágenes hemos implementado una serie de funciones, aunque la que realmente realiza la lectura de los datos es cargarImagenes(path). Esta función recibe como entrada la dirección a la carpeta principal que alberga las imágenes, bien sea la carpeta "Training Data" o "Test Data". La función hará uso del módulo "os" para obtener el nombre de todas las carpetas que se encuentren dentro del path proporcionado ya que como se ha comentado anteriormente las imágenes están distribuidas a lo largo de varios directorios. Con esta información la función se meterá en todas estas carpetas e irá cargando todas las imágenes que encuentre. Al mismo tiempo se introducen en un diccionario que tendrá como clave la clase a la que pertenece la imagen y en cada entrada del diccionario encontraremos una lista con todas las imágenes que pertenezcan a esa clase.

Como el problema que tratamos de resolver no es simplemente identificar si dos habitaciones representan la misma estancia (dormitorio, cocina, etc), sino que queremos identificar si son dos fotos pertenecientes a la misma habitación pero tomadas desde un

ángulo distinto, no podemos tomar como clase el nombre de los directorios que contienen las imágenes. Por lo tanto, las clases con las que trabajaremos serán el nombre de estos directorios seguidos por el nombre de las imágenes exceptuando el nºImagen.

Una vez hemos obtenido los diccionarios pertenecientes al test y al training, y al tratarse de un problema que vamos a resolver mediante el uso de redes siamesas, hemos decidido eliminar aquellas clases que solamente tengan una única imagen ya que no vamos a poder formar parejas de imágenes que pertenezcan a esa clase. Para ello hemos utilizado la función eliminarEjemplosNoValidos(diccionario) que irá recorriendo el diccionario comprobando que se cumple esta condición.

Como veremos posteriormente, en el apartado “Parejas”, hemos diseñado distintas maneras de formar las parejas, algunas de las cuales necesitan que las imágenes no estén almacenadas en un diccionario, sino en otra estructura de datos y para poder presentar los datos correctamente a estas funciones hemos decidido adaptar las funciones cargarDatos y leerImagenes que se nos proporcionaron para la realización de la práctica 2 de esta misma asignatura. La segunda de estas funciones almacenará todas las imágenes en una única lista guardando en otra lista del mismo tamaño la clase a la que pertenece cada imagen de tal forma que la imagen almacenada en la posición i de la primera lista pertenece a la clase almacenada en la dirección i de la segunda lista. Finalmente la función cargarDatos haciendo uso de estas dos nuevas estructuras proporcionadas por la función anterior barajará los datos y nos presentará las clases en forma de matriz. Esta matriz tendrá un tamaño de nxm donde n es el número de imágenes y m el número de clases y será de la siguiente forma :

```
[[000...1...000]
 [000...1...000]
 ...
 [000...1...000]
 [000...1...000]]
```

donde cada fila representa una imagen y cada columna una clase. Las filas son por lo tanto sucesiones de 0s con un único 1 indicando la clase a la que pertenece la imagen. Para poder hacer esta transformación hemos tenido que convertir las clases a números previamente para poder utilizar la función to_categorical finalmente.

Parejas

Como ya se ha mencionado en esta memoria, el objetivo es trabajar con redes siamesas para detectar si dos imágenes representan distintas perspectivas de una misma habitación, por lo tanto uno de los aspectos más importantes es formar parejas de imágenes de una manera correcta puesto que la red trabajará con duetos.

Inicialmente creamos una primera función make_pairs_v1 que recibe una lista con todas las imágenes y una matriz indicando las clases a las que pertenecen esas imágenes. El objetivo principal que persigue esta función es crear una pareja positiva (cuyas dos imágenes representen la misma habitación) y una negativa (cuyas dos imágenes representen habitaciones distintas) para cada una de las imágenes. Para ellos hemos obtenido inicialmente en que posiciones de la lista se almacenan todas las imágenes que pertenecen a la misma clase que la imagen actual con la que estamos trabajando en cada

momento. De entre todas estas imágenes que pertenecen a la misma clase que la actual escogemos una aleatoriamente asegurándonos de que no sea la propia imagen actual y formamos la pareja positiva con estos dos retratos. A continuación escogemos otra imagen aleatoria de entre todas las que no pertenezcan a la misma clase que la actual para formar la pareja negativa. Estas dos parejas se introducen en una lista indicando en otra un 1 para la imagen positiva y un 0 para la negativa. Esta otra lista es la que usaremos como etiquetas a la hora de entrenar ya que es la que nos dice si la pareja la consideramos positiva o negativa. Repetimos este proceso con todas las imágenes de la lista y así conseguimos formar todas las parejas correctamente.

Inicialmente trabajamos con este modelo de parejas, hasta que nos dimos cuenta de que las distintas redes podrían aprender mejor si realizamos un aumento de los datos, es decir, aumentar la cantidad de parejas durante el aprendizaje. Con este argumento nos decidimos a hacer la función `make_pairs_v2`, que realiza el mismo proceso que la versión anterior, con la diferencia de que ahora vamos a guardar dos parejas de imágenes que no pertenezcan a la misma clase por cada una de las distintas imágenes que tenemos.

Rápidamente nos dimos cuenta de que esto no era una solución buena, ya que los modelos cogían mucha predisposición a decir que dos imágenes no pertenecen a la misma clase, ofreciendo un valores de similarity bajos. Conseguimos mejores resultados en las parejas negativas a costa de perder prestaciones en las positivas, de cierta manera estábamos pegandonos más a las parejas negativas y para tratar de solucionar este problema se nos ocurrió que podríamos formar una tercera versión, el `make_pairs_v3`. El objetivo de esta función es crear el mismo número de parejas positivas que negativas, creando dos de cada tipo por cada imagen.

Con esta tercera versión mejoraron los resultados respecto a las versiones anteriores, aunque tenía un inconveniente, requiere una gran cantidad de memoria RAM y colab no podía completar la ejecución en algunas ocasiones. Además nos daba la sensación viendo las gráficas que teníamos un sobreajuste más alto de lo deseable, posteriormente confirmamos nuestras sospechas mostrando algunas imágenes junto con sus predicciones, nos dimos cuenta de que se repetían parejas, esto es que en aquellas clases que solamente tienen dos imágenes estábamos creando las parejas (i,j) y (j,i) siendo ' i ' y ' j ' las dos únicas imágenes pertenecientes a una clase y esto nos estaba produciendo este overfitting.

Para evitar esto, y reducir la cantidad de parejas durante el entrenamiento para asegurar tener suficiente RAM en todo momento, decidimos dar un cambio de sentido y tirar por otra línea distinta. Así es como decidimos implementar la función `hacerParejas`. Esta función recibe el diccionario creado al leer las imágenes directamente y lo que hace es crear tantas parejas positivas distintas como nos sea posible con las imágenes que tenemos. Esto es si para una clase tenemos 5 imágenes generamos las parejas $(1,2)$, $(1,3)$, $(1,4)$, $(1,5)$, $(2,3)$, $(2,4)$, $(2,5)$, $(3,4)$, $(3,5)$, $(4,5)$. Llevando siempre la cuenta de cuántas parejas positivas hemos creado con estas imágenes para crear el mismo número de parejas negativas con ellas y así equilibrar los resultados y que no se peguen a una de las dos opciones como pasaba con la segunda versión. Tras crear todas las parejas, las barajamos para incluir aleatoriedad en cuanto al orden se refiere.

Con esta nueva función conseguimos mejorar los resultados aún más y conseguimos también reducir tanto el overfitting producido por la repetición de parejas, como la cantidad de parejas formadas lo que reducía la memoria RAM necesaria. Este es el modelo final de creación de parejas con el que nos hemos quedado ya que nos produce unos resultados mejores que todos los modelos anteriores siendo más rápido y sin requerir una gran cantidad de recursos al trabajar con diccionarios en lugar de con arrays y listas.

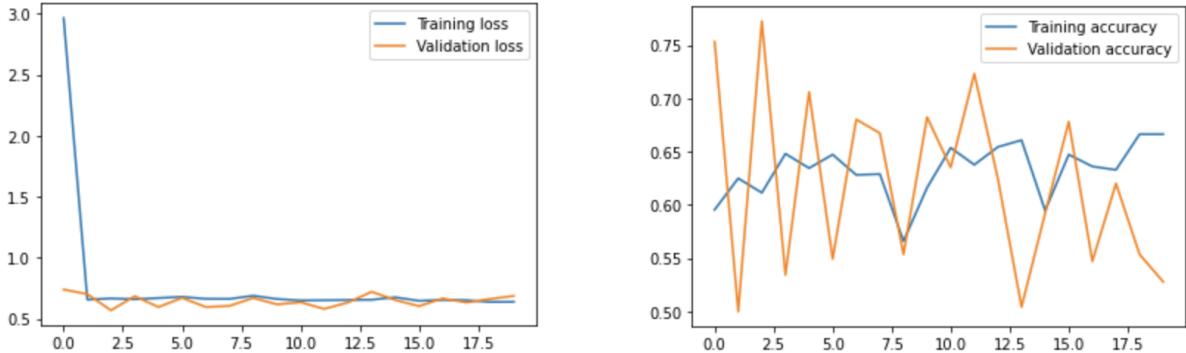
Submodelos

Submodelo 1

Una vez tenemos hechas las parejas hay que implementar el modelo en el que extraemos el vector de características de cada imagen para después poder compararlas en el modelo siames. La primera tentativa a solucionar el problema lo hacemos con un modelo de red neuronal con capas de convolución e implementando las técnicas aprendidas en la práctica 2 del curso, añadiéndole capas como pooling o de dropout. Para ello empezamos con una capa de convolución 2D seguida por activación relu, con 64 filtros, un tamaño de kernel de (2,2) y padding “same”. Un MaxPooling2D de un (2,2) en el que se queda con el valor máximo en cada cuadrado de 2x2 para cada pixel y el Dropout con un 0.3 para que distribuya la información a un 70% de las “neuronas” de las capas siguientes.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 256, 256, 3)]	0
conv2d (Conv2D)	(None, 256, 256, 64)	832
max_pooling2d (MaxPooling2D)	(None, 128, 128, 64)	0
dropout (Dropout)	(None, 128, 128, 64)	0
conv2d_1 (Conv2D)	(None, 128, 128, 64)	16448
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 64)	0
dropout_1 (Dropout)	(None, 64, 64, 64)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 64)	0
dense (Dense)	(None, 274)	17810
<hr/>		
Total params:	35,090	
Trainable params:	35,090	
Non-trainable params:	0	

Intentamos añadirle más profundidad a la red, cambiar los parámetros en las capas de convolución, pooling o dropout, pero el resultado obtenido no variaba mucho del siguiente:



Viendo que por mucho cambio que intentamos hacer, los resultados son siempre los mismos, con poca mejoría desde que empieza a entrenar hasta que termina la última época, y no siempre va a mejorar como vemos en las gráficas y comentaremos más tarde.

Submodelo 2

Ante los resultados obtenidos con el submodelo anterior, pensamos que podría ser buena idea utilizar un modelo ya existente y que se adapte bien a muchos problemas de distintas naturalezas. Es por esto que usamos Resnet50 del módulo de Keras de aplicaciones para extraer las características de las imágenes preentrenada con imagenet usando capas de average pooling y sin incluir las capas fully-connected del principio, y finalizamos el submodelo añadiendo una capa Dense con 274, que son el número de habitaciones únicas que encontramos en el conjunto de Training. Para esta versión no reentrenamos las capas de Resnet50 haciendo uso de lo empleado en la práctica 2, recorriendo todas las capas de forma que su atributo trainable lo pasamos a False.

Submodelo 3

Este tercer submodelo que hemos implementado es un intento de mejorar el modelo anterior. Tras hacer varias pruebas añadiendo distintas cosas, como early-stopping, aumentar la profundidad con más capas, fine-tunning, etc, nos dimos cuenta de que no estábamos consiguiendo los resultados esperados ya que no conseguíamos obtener mejores predicciones que con Resnet50. Debido a esto hemos dejado este tercer submodelo simplemente con fine-tunning para que quede constancia tanto en la memoria como en el código de alguno de los experimentos aunque como veremos posteriormente, no obtendremos mejores resultados. En este submodelo se usa Resnet50 reentrenando todas las capas, especificando un 0 en el parámetro adecuado, o un número determinado de ellas, indicando este número en ese mismo parámetro.

Modelo Siames

En el apartado anterior hemos explicado algunos de los modelos que hemos implementado, los que hemos creído más interesantes y por ello aparecen en el código de la práctica, pero estos modelos representan distintas redes convolucionales, ahora debemos de montar la red siamesa que vamos a utilizar así como explicar cómo hemos llevado a cabo su entrenamiento.

Antes de todo esto, lo primero que tenemos que hacer es, puesto que el conjunto test no se debe de tocar en el entrenamiento ni antes de la predicción, generar un conjunto de validación para utilizarlo durante el entrenamiento. Este conjunto será un subconjunto del train y para crearlo hemos implementado la función formarValidation que recibe las imágenes del conjunto de entrenamiento, sus etiquetas y la probabilidad entre 0 y 1 de que una imagen se quede en este conjunto de entrenamiento, siendo la probabilidad de que pase al conjunto de validación de uno menos esa probabilidad anterior.

Con los conjuntos train y validation ya formados correctamente, pasamos a crear nuestro modelo de red siamesa y su posterior compilación y entrenamiento. Para realizar todas estas tareas se ha creado una nueva función completarCompilarEntrenarModelo(model, tamImagenes,par_x_train_final,par_y_train_final,par_x_val,par_y_val). Esta función lo primero que hará será instanciar tensores de keras para ambas imágenes de cada pareja mediante la función Input() del módulo tf.keras indicando el tamaño de las imágenes. Con ayuda de esta función obtenemos una representación de los vectores de características que obtendría el modelo para cada imagen y con estos dos vectores y la función euclidean_distance obtenemos la distancia euclídea que hay entre ellos. Finalmente solo nos queda completar la red siamesa con una última capa Dense con activación sigmoidal para obtener valores entre 0 y 1 (el porcentaje de similitud entre ambos vectores).

Con la red siamesa creada, lo siguiente que debemos hacer es compilarla y para ello definir un optimizador es un paso esencial. En nuestro caso hemos utilizado el gradiente descendente estocástico SGD() con una tasa de aprendizaje de 0.01 ya que es el más utilizado a lo largo tanto de este curso como de cursos anteriores, produciendo siempre buenos resultados. Aunque esto no significa que no hayamos investigado con otros optimizadores, en concreto se ha probado con Adam(), el cual es otro método de gradiente descendente estocástico, solo que está basado en momentos de primer y segundo orden. El motivo por el que se decidió probar este optimizador es que en algunos de los artículos y documentos analizados y estudiados previamente a la realización de este proyecto, utilizan este optimizador en sus redes siamesas obteniendo resultados mejores al SGD. Sin embargo, en nuestro caso no fue así, por lo que nos quedamos con el SGD(). Al realizar la compilación del modelo también establecemos como función de pérdida el binary_crossentropy. La selección de esta función de pérdida se debe a que en los artículos estudiados que trabajan con redes siamesas en las que tenemos que decidir entre dos clases, tenemos que decidir si pertenecen a la misma clase o no, y en las que trabajamos con parejas de imágenes es una de las más utilizadas por su sencillez y buenos resultados obtenidos.

Una vez compilada, debemos entrenar nuestra red. Como ya sabemos esto se hace con el método fit, que nos permite además obtener un historial con los valores de la pérdida y el accuracy en las distintas etapas. Para poder mostrar este proceso de aprendizaje se hace uso de la función mostrar_hist, proporcionada para la realización de la segunda práctica de la esta asignatura.

Predicción

Para poder comprobar como de buenos han sido los resultados obtenidos implementamos el procedimiento predecir(modelo, parejas_imagenes,parejas_etiqueta), en el que el modelo

es el modelo que hemos entrenado en el paso anterior, parejas_imagenes será una lista que contendrá las parejas de imágenes que hemos formado para el conjunto de fotos en Test y parejas_etiqueta otra lista que tendrá 1 si la pareja de imágenes es positiva (pertenece a la misma habitación) o 0 si la pareja es negativa (son distintas habitaciones).

Entonces se recorrerá cada pareja de imágenes Test para introducirlas en el método predict del modulo Keras, este nos devolverá un número entre 0 a 1 que nos indicará la probabilidad de que la pareja de imágenes pertenezca a la misma habitación. Para calcular el porcentaje de acierto de nuestro modelo se pondrá un umbral, al principio fue 0.5 pero tras muchas pruebas observando las distintas probabilidades predecidas lo mejor es bajarlo a 0.4, para determinar que parejas de imágenes determina el modelo que son de la misma habitación y cuales no en función de si supera el umbral o no. Nos guardaremos 4 variables en las que guardamos los resultados bien obtenidos tanto para las parejas positivas como para las negativas, y el nº de errores cometidos en predicciones para ambas situaciones, con ello podremos formar la matriz de confusión y calcular el porcentaje de acierto.

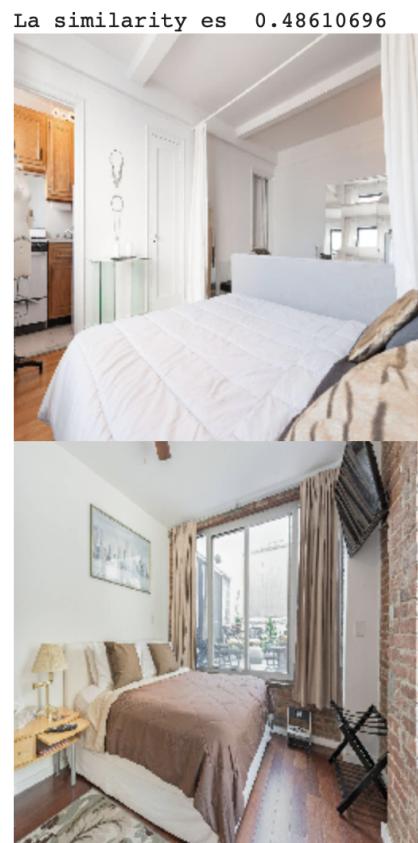
Con el método generarMatrizConfusion formamos la matriz de confusión, para el eje y tenemos las predicciones con la primera fila para las parejas negativas y la segunda para las parejas positivas, y en cuanto al eje x se tiene que todas las parejas positivas se encuentran en la primera columna, y las parejas negativas en la segunda columna, dejando así en la diagonal principal de la matriz todos los aciertos de nuestro modelo al predecir en Test y en la diagonal secundaria los fallos.



Como podemos observar en las dos primeras parejas de imágenes empezando por la izquierda, tenemos dos ejemplos de parejas positivas en las que la predicción da como positivas, en la primera de ellas se muestra la misma habitación desde distintas perspectivas, dando lugar a que solo en una de ellas aparezca la ventana o como en la otra que aparece una silla y la cocina que se encuentra pegada a la habitación, a pesar de ello encuentra un 59% de probabilidad de que sean la misma superando el 40% del umbral que

hemos fijado. Para la segunda pareja de imágenes se ve mucho más claro porque hay pocos cambios entre una imagen y la otra y encuentra un 77% de probabilidad de que sea la misma habitación. Las dos parejas de imágenes a la derecha dan ejemplo a casos negativos acertados, de forma que los porcentajes son menores al 40% establecido, siendo en la primera dos habitaciones con elementos parecidos entre ellas, colores, brillo y demás pero identifica que solamente se parecen en un 20%, sin embargo las últimas parejas de imágenes no se parecen en nada, siendo una cocina y una habitación, en el que su probabilidad de ser la misma habitación es de un 8%.

A continuación se muestran dos parejas de imágenes con un porcentaje similar pero con resultados distintos, ya que en la primera imagen con un 47% acierta que son la misma habitación, pero sin embargo en la siguiente con un 48% se equivoca al predecir que son la misma habitación, ambas son dormitorios y tienen las paredes blancas, pero a ojos de una persona la primera pareja de imágenes tendría mayor probabilidad de pertenecer a la misma habitación que la segunda pareja.



Resultados finales

Una vez hemos explicado en los apartados anteriores las distintas técnicas utilizadas y el enfoque que hemos seguido para la realización de la práctica, así como las distintas funciones implementadas. En este apartado expondremos los distintos resultados parciales y los finales que hemos obtenido en las pruebas realizadas.

Los primeros resultados que queremos mencionar son los obtenidos con el primer submodelo explicado en apartados anteriores y la función make_pairs_v1. Este submodelo, como ya se ha mencionado está formado por unas capas que hemos ido añadiendo basándonos en el modelo que se puede encontrar en la siguiente dirección:

<https://www.pyimagesearch.com/2020/11/30/siamese-networks-with-keras-tensorflow-and-deep-learning/>. Con este modelo hemos obtenido una historia de su función de pérdida y accuracy durante el aprendizaje tal cual se puede observar en séptima página de esta memoria.

En estas gráficas observamos como la pérdida del training se mantiene bastante estable, sin subidas y bajadas bruscas, aunque a un valor demasiado alto para nuestros intereses, por encima del 0.5 . La validación sin embargo, es distinta, no es tan estable pero en muchos momentos mejora al valor del training en este caso.

Observando el accuracy podemos ver, por contra, que ambos conjuntos son excesivamente irregulares, dando saltos muy grandes en cada época sin ningún sentido aparente. Aunque sí es cierto que parece que el training tiene una ligera tendencia a crecer y la función perteneciente a la validación todo lo contrario, una tendencia a perder prestaciones.

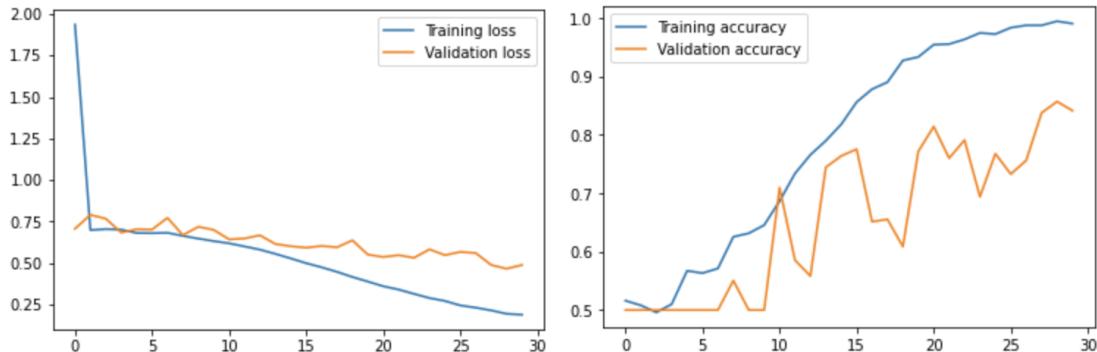
Viendo estas gráficas que acabamos de comentar, no parece lógico tener esperanzas en que con este modelo con los parámetros introducidos y la formación de parejas escogida vayamos a tener una producción aceptable. En concreto hemos obtenido un 52.36% de acierto, esto quiere decir que hemos sido capaces de identificar si dos imágenes pertenecen a la misma habitación en poco más de la mitad de los casos disponibles. Esta predicción no es nada buena ya que estamos muy por debajo de lo que sería aceptable. Para observar más claramente los resultados hemos obtenido la matriz de confusión:

		0	1
0	13.000	2.000	
1	220.000	231.000	

En esta matriz podemos observar en la diagonal principal los casos acertados y en la secundaria los fallados. También podemos ver en la primera fila las veces que ha dicho que una pareja era negativa y en la segunda fila las veces que ha predicho nuestro modelo que era positiva. En las columnas tenemos por como era realmente esas parejas. Una vez dicho esto, observamos como este modelo tiene una excesiva tendencia a predecir que dos imágenes sí pertenecen a la misma habitación, solo ha dicho en un 3% de los

casos que la pareja era negativa. Esto nos lleva a ese 52% de acierto únicamente, ya que sabemos que en este caso tenemos un 50% de parejas positivas y un 50% de duetos negativos. Esta tendencia no ha sido única con este modelo y sus intentos de mejora que hemos llevado a cabo, de hecho, hemos podido observar en todas las ocasiones que hemos predicho como se mantenía este efecto, sin llegar a alcanzar nunca un porcentaje de acierto mayor al 60% y con mucha predisposición a predecir un 1 como clase de las parejas. Esto es así, hasta el punto de que en una ocasión este modelo predijo que en todos los casos las imágenes representaban la misma habitación y de este modo se aseguraba un porcentaje del 50%.

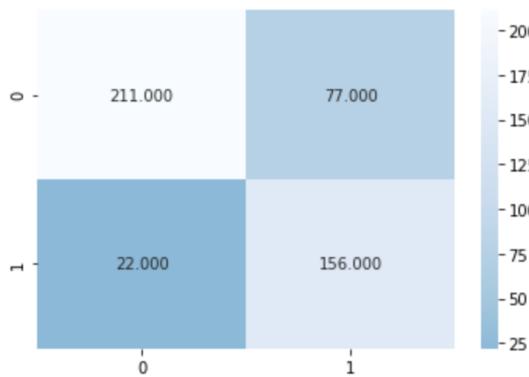
A continuación, decidimos cambiar el enfoque y trabajar con el modelo resnet50, como ya se ha explicado anteriormente. Al entrenar este modelo con las mismas parejas al caso anterior obtuvimos las siguientes gráficas:



En la primera gráfica, la que nos presenta información sobre la función de pérdida vemos como mientras el entrenamiento si ha conseguido mejorar respecto al caso anterior y no mantenerse constante sino que tiene una curva suave descendente llegando a valores inferiores a 0.25, el validation ha mantenido el comportamiento anterior.

El accuracy sin embargo es distinto, las dos rectas que representan el comportamiento de ambos conjuntos ahora son claramente ascendentes. A pesar de esto, la representación del validation es demasiado inestable y presenta muchos altibajos.

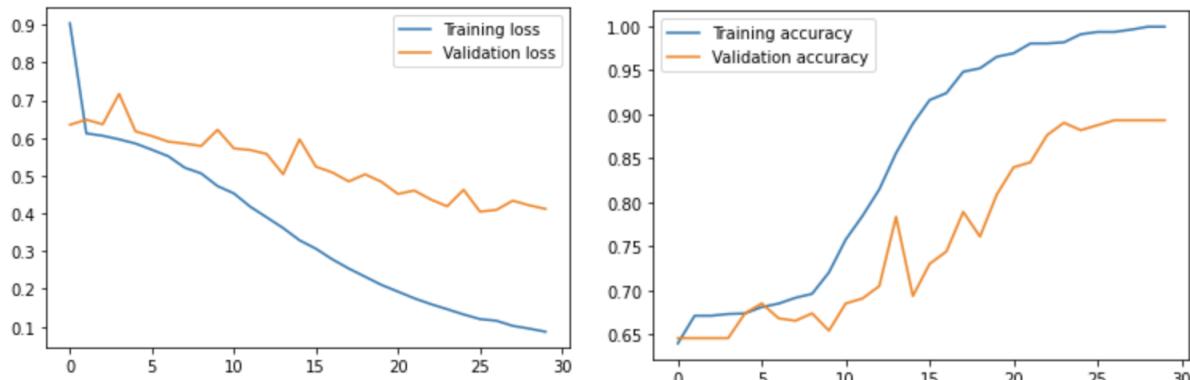
A pesar de esto, las gráficas nos hacen pensar lo que ya esperábamos, esto es, que vamos a obtener una mejora sustancial de los resultados al predecir utilizando este modelo resnet50. De hecho, obtenemos un acierto del 77.10% con una matriz de confusión como la siguiente:



Lo primero que apreciamos en esta imagen es que ha desaparecido ese comportamiento de decir que sí son iguales con tanta facilidad, o en la mayoría de los casos.

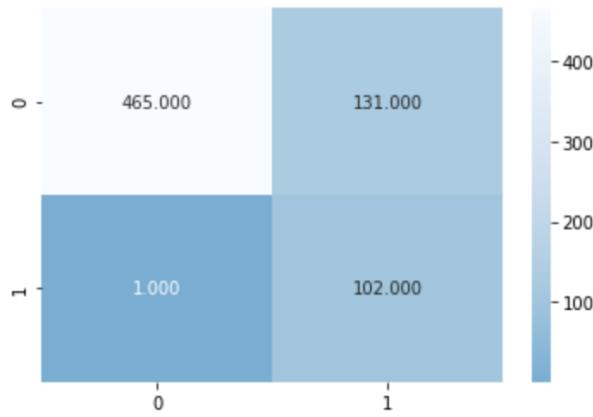
Aún así pensamos que este resultado anterior de un 77% es bastante aceptable aunque mejorable. Antes de empezar a mejorar el modelo, nos dimos cuenta, como ya se ha comentado, que los conjuntos de imágenes se podrían mejorar y con ello fomentar una mejor predicción. De ahí salió la idea de hacer la función make_pairs_v2 (el doble de

parejas negativas que positivas) y decidimos probarla con este modelo resnet con el que habíamos obtenido tan buenos resultados. Las gráficas serían las siguientes:



Poco hay que comentar respecto a la primera gráfica puesto que es muy similar a la anterior. No parece ni que hayamos bajado los valores de la función de pérdida para alguno de los dos conjuntos. Respecto a la segunda gráfica si se ve una cierta mejoría, sobre todo en el conjunto validación. Resulta curioso observar como ambas funciones empiezan más lentas que en el caso anterior, es decir, empiezan a subir más tarde, pero sin embargo, una vez que empiezan dan menos saltos y consiguen llegar a un valor más alto.

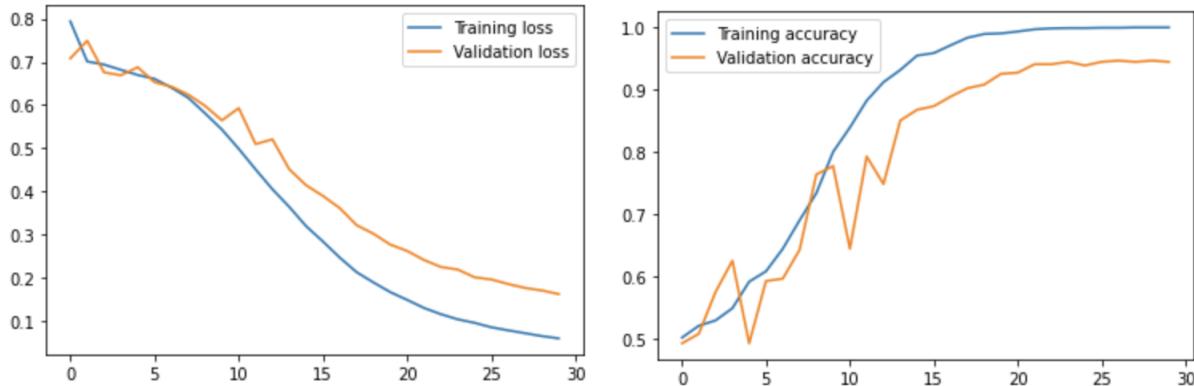
Este modelo obtiene un 81.11% de bondad en las predicciones, sin embargo, como



podemos observar en esta matriz, ahora tenemos una fuerte tendencia a decir que dos imágenes no representan la misma habitación.

A este hecho, una de las explicaciones que se nos ocurrieron es el tener el doble de parejas negativas que positivas a la hora de entrenar. Pensamos que nuestro modelo estaba ajustándose demasiado a este hecho ya que en el caso de duda hay más posibilidades de acertar diciendo que la pareja es negativa.

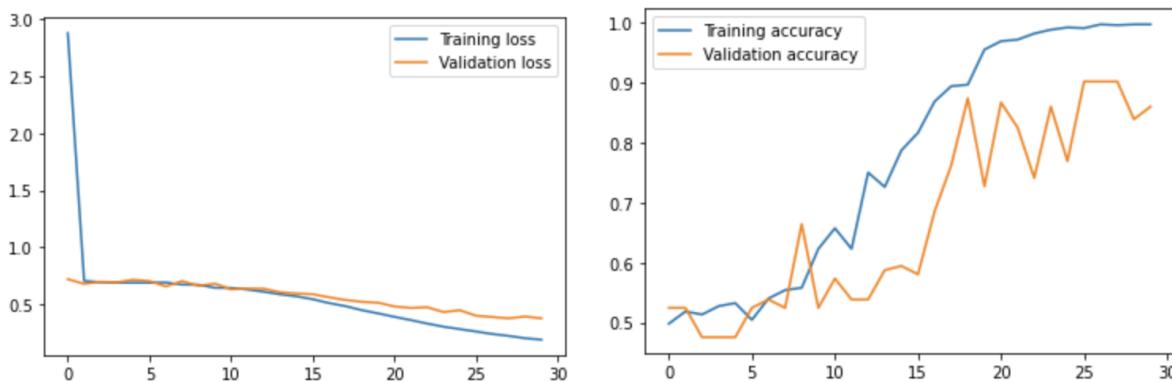
Esto nos llevó a la tercera versión de make_pairs, en el que volvemos a tener el mismo número de parejas positivas que negativas. Siendo este número el doble al de la primera versión, para así intentar mejorar los resultados aumentando la cantidad de datos de entrada. De este modo y con el mismo modelo conseguimos obtener una bondad en la predicción en torno al 80% de nuevo, al igual que ocurre en el caso recién explicado. Aunque siempre obtenemos sobre un 1% o 2% menos que antes. En las gráficas podemos observar lo siguiente:



Lo primero que llama la atención con un simple vistazo, es que los dos conjuntos presentan un comportamiento casi idéntico. Es decir, el conjunto train obtiene mejores valores como es normal puesto que estamos entrenando con él, pero el validation es extrañamente similar. Las mismas curvas, se mantiene a la misma distancia a lo largo del dominio, etc. Esto es debido a que hay un claro overfitting de los datos.

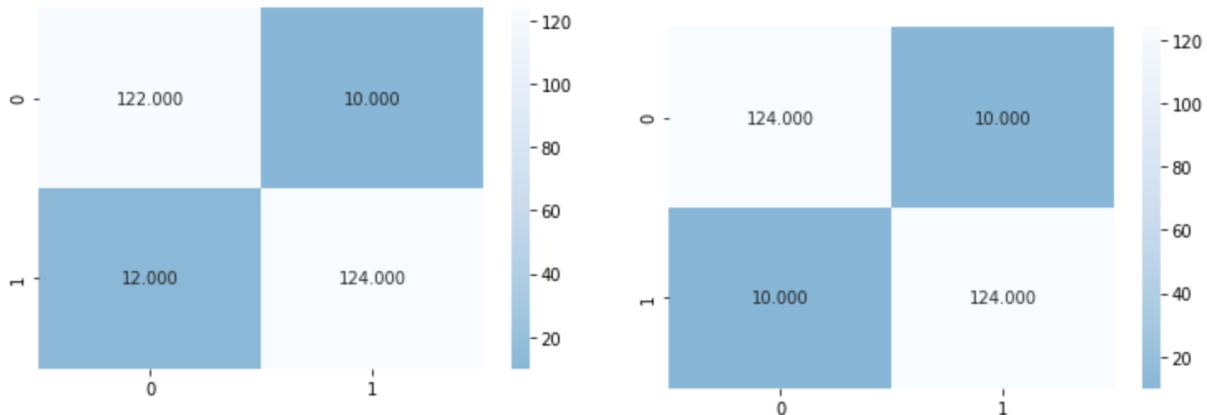
Estudiando a qué se podía deber este sobreajuste llegamos a la conclusión de que debido a nuestra manera de hacer las parejas estábamos repitiendo parejas, tal y como explicamos en secciones anteriores, lo que produce este fenómeno. Y no solo esto, sino que al dividir el train inicial en training y validation se da el caso con bastante probabilidad de que existan las mismas parejas en ambos conjuntos. Esto nos hizo cambiar completamente la forma de hacer parejas y nos llevó a la función hacerParejas que se ha explicado anteriormente. Pensamos con esta función que quizá sería más conveniente tener menos parejas, siempre que sea una cantidad suficiente, pero garantizar que no se repitan parejas y sobre todo, que no se repitan en dos conjuntos distintos como son el validation y el train.

Con esto llegamos a estos resultados obtenidos con este modelo resnet50 y esta forma de crear los datos de entrenamiento, validación y test.



En estas gráficas vemos un paso atrás, en cuanto a ajustarse las dos rectas, lo que significa que hemos conseguido reducir el sobreajuste. En la función de pérdida vemos como hemos obtenido unos valores un poco más altos, pero sin llegar a ser algo significativo que vaya a marcar las predicciones para mal. En la otra gráfica volvemos a tener cierta irregularidad en cuanto a los saltos que se dan en el conjunto de validación. Sin embargo el accuracy es muy bueno a pesar de estos saltos y la línea es claramente ascendente. En cuanto a las

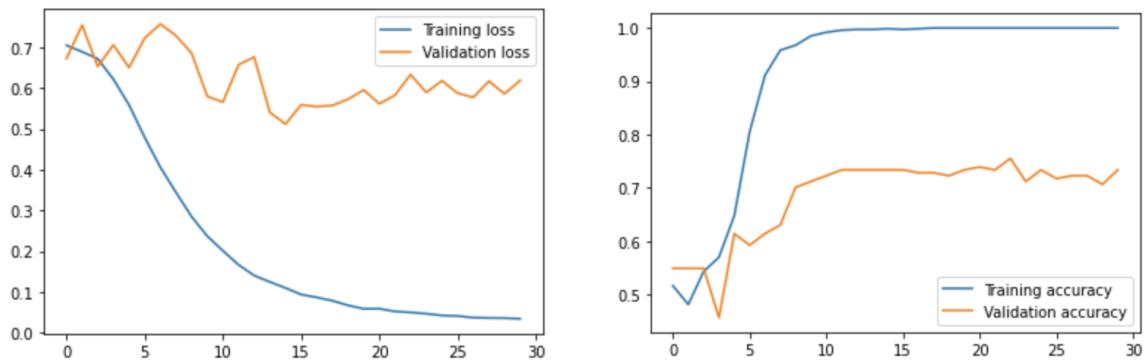
predicciones obtenidas obtenemos matrices de confusión del estilo a las siguientes en todas las iteraciones:



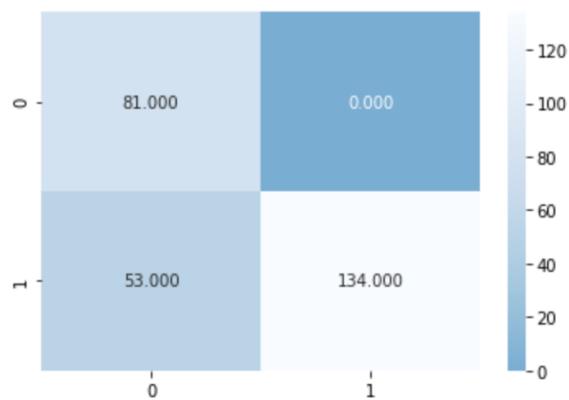
En estas matrices, podemos observar como el número de fallos(diagonal secundaria) son muy reducidos. De hecho obtenemos un acierto del 91.17% en la ejecución correspondiente a la matriz de la izquierda y un 92.53% en la ejecución correspondiente a la matriz de confusión de la derecha. A parte de obtener un porcentaje de acierto muy elevado, más de un 90% de acierto en todas las iteraciones, vemos como los fallos están compensados, es decir, no tenemos ninguna preferencia aparente a la hora de decir si es una pareja positiva o negativa. Tenemos prácticamente el mismo número de falsos positivos que de falsos negativos.

Tanto este como los otros resultados expuestos anteriormente y posteriormente a este es con un porcentaje del 20% para el conjunto de validación y umbral de 0.4, es decir, a partir de una similitud mayor a 0.4 entendemos que se corresponde con la misma habitación ambas imágenes. Pero estos valores no son casualidad, han sido el resultado del estudio realizado para escoger los mejores valores. Escogiendo un 30% de los datos del train para validación el porcentaje de acierto baja hasta el 90% por ejemplo, y con un umbral de 0.5 este porcentaje baja aún más hasta un 88% u 89% de acierto.

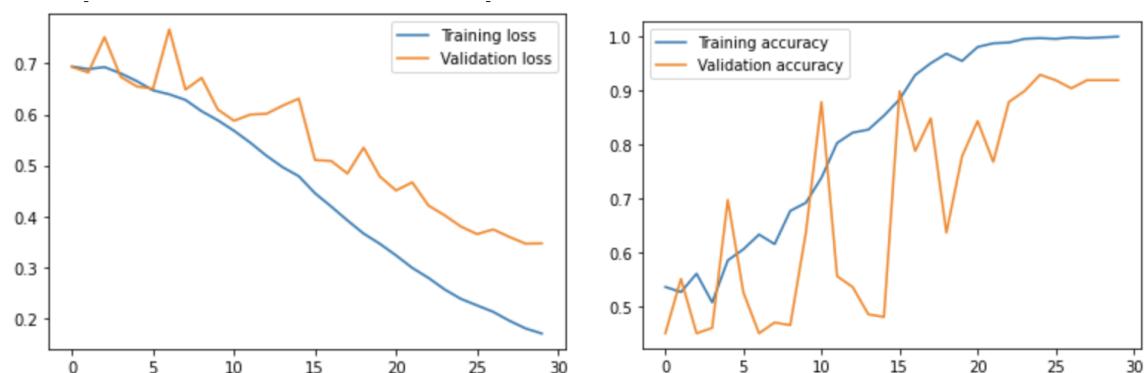
A pesar de tener unos resultados ya bastante buenos en la predicción buscamos a ver si con técnicas estudiadas como fine-tunning en resnet50, o añadirle más profundidad a la red con distintas capas como convolución podríamos mejorar algo más los resultados. Aquí tenemos una muestra de lo obtenido al entrenar el modelo con esta técnica, obteniendo unos resultados en el conjunto de validación peores que en la versión anterior, algo que se repetía al añadir mayor profundidad al modelo o emplear otras técnicas como early stopping, en la mayoría de casos esto sucedía por el overfitting, ya que en la quinta época ya alcanzábamos un valor de accuracy casi del 100% para el conjunto training mientras que en los casos anteriores no sucedía hasta las épocas 20-25, llegando a mejores resultados en validación.



El resultado obtenido en estos modelos rondaban entre un 75% a un 82%, en este particular un 80% en el que como podemos observar de todas las parejas negativas, fallaba en 53 parejas prediciendo que son la misma habitación cuando no lo son, un valor muy significativo ya que supone el 40% de las parejas negativas que las predice mal.

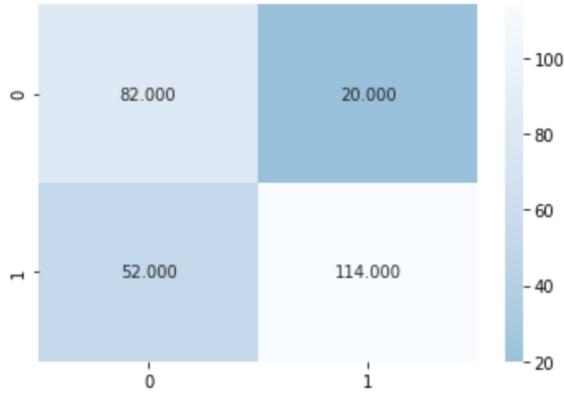


En otro intento por mejorar se usó la técnica de fine-tunning por capas, exactamente todas menos las 10 finales, que nos dió algo mejor en la práctica 2, y por lo que se puede observar da mejores resultados en las gráficas ya que no ocurre un sobreajuste hasta la parte final del modelo permitiendo una precisión mejor en el conjunto de validación, pero esto no dará buenos resultados al predecir como veremos a continuación.



Los resultados en Test son malos, encontrando un 73% de acierto en las parejas de imágenes, se puede explicar viendo las gráficas nuevamente, ya que esos altibajos en la

precisión del conjunto de validación indican que realmente depende mucho en este modelo de que imágenes esté evaluando para obtener buenos resultados o malos, resultando en Test con peores predicciones incluso que con las anteriores versiones de fine-tunning o mayor profundidad a la red.



CONCLUSIÓN

Tras todo este análisis llegamos a la conclusión que el mejor modelo es el que usaba la última versión del método para hacer las parejas, no siendo tan importante la cantidad como la calidad de estas, siempre y cuando la cantidad de parejas sea la suficiente, 928 parejas para el Training (de las cuales un 20% se usarán para la validación) y 268 para el Test. También destacar que no por complicar el modelo y añadir más parámetros entrenables (fine-tunning) o más capas para añadir profundidad a la red resultará en una mejora del modelo para predecir un problema concreto, en nuestro caso usando un modelo resnet50 entrenado con imagenet y añadiéndole una capa Dense nos da unos resultados excelentes, en el que tras muchas ejecuciones los resultados no varían más allá del 88% al 94% de acierto al predecir, y al intentar mejorarlo con redes más complejas no se conseguía no solo mejorarla, si no empeorarlos mucho.

Este problema ha estado centrado en otros papers o notebooks en la clasificación de las imágenes para clasificarlas según el tipo de habitación usando redes convolucionales y obteniendo unos resultados entre 60% y 75%, sin embargo nosotros hemos querido enfocar el problema a la identificación de dos fotografías que capturen la misma habitación. No obstante hemos podido observar como nuestra red siamesa obtenía un valor de similaridad mayor entre imágenes que mostraban habitaciones equivalentes (dormitorios, cocinas, ...) a entre imágenes que mostraban estancias completamente distintas. Por esto pensamos que podría ser interesante aplicar este modelo de redes siamesas al problema de clasificación que hemos mencionado.

Bibliografía

- <https://stackoverflow.com/questions/43977463/valueerror-could-not-broadcast-input-array-from-shape-224-224-3-into-shape-2>
- https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/load_img
- [https://en.wikipedia.org/wiki/Cropping_\(image\)](https://en.wikipedia.org/wiki/Cropping_(image))
- <https://jkjung-avt.github.io/keras-image-cropping/>
- <https://www.pyimagesearch.com/2020/11/30/siamese-networks-with-keras-tensorflow-and-deep-learning/>
- <https://www.pyimagesearch.com/2021/01/18/contrastive-loss-for-siamese-networks-with-keras-and-tensorflow/>
- <http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf>
- <https://stats.stackexchange.com/questions/388859/is-it-possible-to-give-variable-sized-images-as-input-to-a-convolutional-neural>
- <https://medium.com/@prabhnoor0212/siamese-network-keras-31a3a8f37d04>
- <https://keras.io/api/optimizers/>
- <https://keras.io/api/optimizers/adam/>