Visión por Computador - Práctica Final
Uso de redes siamesas para la clasificación de fotografías de ropa (Zalando)

Sergio Quijano Rey
sergioquijano@correo.ugr.es

Alejandro Borrego Mejías
alejbormeg@correo.ugr.es

23 de enero de 2022

Índice

Índice de figuras

Índice de cuadros

1. Abstract

En este proyecto nos proponemos profundizar en el conocimiento de las Redes Siamesas y su funcionamiento para un problema de clasificación multiclase con KNN. Para el entrenamiento de la red emplearemos la función de error Triplet Loss y probaremos distintas técnicas de elección de triples, en concreto usaremos tripes aleatorios y triples *"difíciles"* online. Por otra parte probaremos distintas arquitecturas para la red comenzando con una ResNet-18 de complejidad mucho mayor a la de la base datos y posteriormente usaremos una red mucho más sencilla que hemos denominado Light Model con la que mejoraremos sustancialmente los resultados obtenidos hasta el momento.

2. Introducción

2.1. Problema a resolver

En esta práctica vamos a trabajar con el uso de redes siamesas y la función de pérdida *Triplet Loss*. Por tanto, calcularemos un *embedding* de la base de datos y probaremos distintas técnicas de elección de triples como son los tripes aleatorios y los triples difíciles online. Adaptaremos dicha red para realizar una tarea de clasificación usando el algoritmo *k-NN*.

Realizaremos dos experimentos. El primero de ellos basado en el entrenamiento de la red usando triples aleatorios. El segundo, basado en el entrenamiento de la red usando triples difíciles, calculados de forma *online* por cada *minibatch*.

2.2. Elección del *dataset*

El dataset que hemos utilizado para la realización de este proyecto se denomina *Fashion-MNIST* [1] y nos proporciona un total de 60000 imágenes 28×28 en escala de grises de ropa de distintos tipos (en concreto de 10 clases distintas).

Esta base de datos surge como una alternativa al famoso conjunto de datos *MNIST*. Los autores de la base de datos buscaban mantener la estructura de la base de datos original (tamaño de las imágenes, número de clases, tamaños de los conjuntos de entrenamiento y test, ...) a la vez que proponer un mayor reto que el reconocimiento de dígitos [2].

Más adelante, en *"3.3. Adecuación de la arquitectura de red a la base de datos escogida"*, discutiremos sobre la idoneidad de la arquitectura de red teniendo en cuenta la base de datos que acabamos de presentar.

El código `Pytorch` descarga automáticamente el conjunto de datos, así que no vemos la necesidad de guardar la base de datos en la consigna para ser consultado. Basta con ejecutar el código de la carga de datos para disponer de dicho conjunto de datos. Para más consultas sobre la base de datos, se puede consultar la documentación oficial [1].

2.3. Motivación

Hemos elegido este proyecto por nuestro interés en el estudio de redes Siamesas. Queríamos profundizar en su funcionamiento, en concreto, en las dificultades que supone la generación de triples para el correcto entrenamiento de la red a través de la función de error *Triplet Loss*. De hecho, y como comentaremos más adelante, esta ha sido la parte más complicada del proyecto, pues escribir un código lo más eficiente posible ha sido crítico para reducir drásticamente los tiempos de ejecución.

La elección de la base de datos viene motivada por la actualidad de los datos que en ella se presentan y por las características de la misma. En concreto, es una base de datos lo suficientemente sencilla como para permitirnos realizar la experimentación aquí presentada en tiempos razonables, a la vez que tratábamos un problema más o menos complejo de resolver.

2.4. Objetivos a realizar

Los objetivos que nos proponemos en este proyecto son:

1. Aplicar *Transfer Learning*. Durante todos los experimentos, usaremos una red *ResNet18* pre-entrenada sobre *ImageNet* que modificaremos ligeramente (en la capa de salida) y sobre la que realizaremos *fine tuning*.
2. Entrenamiento de la red usando triples aleatorios. Esperamos obtener resultados muy malos. Sin embargo, tomamos esto como *baseline*, con el que compararemos cuando propongamos una mejor forma de tomar los triples con los que entrenamos.
3. Entrenamiento de la red usando triples difíciles, calculados de forma *online* dentro de cada *minibatch*. Con esto esperamos obtener una mejora drástica frente al uso de triples aleatorios.
4. Adaptación de ambas redes (la entrenada con triples aleatorios y la entrenada con triples difíciles) a una tarea de clasificación usando *k-NN* sobre el *embedding* que las redes siamesas calculan.

2.5. Otros detalles

Al principio del *Notebook* definimos una variable `RUNNING_ENV` que indica en que tipo de entorno nos encontramos. Cuando `RUNNING_ENV == "local"`, indicamos que estamos corriendo el *Notebook* en local, y cuando `RUNNING_ENV == "remote"` indicamos que estamos corriendo en *Google Colab*. Así podemos controlar las pequeñas diferencias entre estos dos entornos de desarrollo. Principalmente son dos las diferencias:

- Cuando corremos en *Google Colab* tenemos que introducir un código de verificación
- Las rutas de la carpeta donde guardamos las imágenes son diferentes

Además, al estar usando `Pytorch` como librería principal, hemos tenido que escribir mucho código para llevar a cabo tareas básicas (como el bucle del entrenamiento, funciones para mostrar

el progreso del entrenamiento conforme se está realizando, funciones para evaluar un clasificador, ...) Por ello hemos decidido separar el código de la siguiente forma:

- Una carpeta `lib/` en la que tenemos ficheros `.py` con el código básico del que ya hemos hablado
- Un *notebook* `Notebook.ipynb`, en el que realizamos todo el trabajo interesante. Por ejemplo, aquí definimos todos los hiperparámetros que usamos finalmente, las funciones de pérdida usadas, la forma de calcular los triples, la evaluación del modelo, ...

En último lugar, cabe destacar que estamos guardando los modelos entrenados en disco. Esto lo controlamos con el parámetro global `USE_CACHED_MODEL`, en la sección de hiperparámetros. Cuando esta variable vale `False`, se realiza el entrenamiento de los modelos y se guardan en memoria. Cuando la variable vale `True`, no se realiza el entrenamiento y se cargan los modelos desde disco. Esto ha sido crítico para poder realizar experimentaciones rápidas sin tener que volver a entrenar una y otra vez los mismos modelos bajo las mismas condiciones. Además, cuando nos hemos encontrado con problemas con el consumo de memoria *GPU*, entrenar, reiniciar y cargar el modelo desde disco ha sido una solución temporal (finalmente resolvimos estos problemas por otro camino, usando adecuadamente la instrucción `torch.no_grad()`)

3. Fundamentos Teóricos

3.1. *Redes Siamesas*

Una red siamesa es una arquitectura en la que se tienen dos redes idénticas, con el objetivo que, dadas dos imágenes de entidades iguales, obtengan dos representaciones cercanas. Y al revés, dadas dos entidades distintas, se debe obtener dos representaciones distantes.

Al trabajar con dos redes idénticas, lo que realmente manejamos es una única red que calcula representaciones de imágenes pasadas como entrada. Usaremos dicha red para calcular representaciones de pares o triples (en nuestro caso concreto, trabajaremos sobre todo con triples) de imágenes dadas como entrada.

Aclaramos lo que entendemos por representaciones cercanas y distantes. En primer lugar, la red calculará, dada una imagen de entrada 28×28, un *embedding* de la imagen. Esto es, una representación vectorial es un espacio $\mathbb{R}^d$ con $d \ll 28 * 28$ (ie. $d = 2$, $d = 4$, ...).

Con esto, dadas dos imágenes x, y, sus representaciones $f_\theta(x), f_\theta(y)$ serán cercanas cuando

$$||f_\theta(x) - f_\theta(y)||_2 \approx 0$$

Del mismo modo serán lejanas cuando

$$||f_\theta(x) - f_\theta(y)||_2 \gg 0$$

3.2. *Triplet Loss*

Esta será la función de pérdida que vamos a usar para optimizar nuestra red siamesa. A continuación introducimos el por qué de su uso.

Durante el entrenamiento, la red verá como ejemplos triples. Esto es, triples de la forma *anchor*, *positive*, *negative*. *anchor* será una imagen arbitraria, mientras que *positive* será una imagen de la misma clase que *anchor* y *negative* será una imagen de otra clase. Queremos que la distancia entre ancla y positivo sea pequeña, y entre ancla y negativo sea grande. Por tanto, lo que queremos es:

$$||f_\theta(anchor) - f_\theta(positive)||^2 \leq ||f_\theta(anchor) - f_\theta(negative)||^2$$

luego

$$||f_\theta(anchor) - f_\theta(positive)||^2 - ||f_\theta(anchor) - f_\theta(negative)||^2 \leq 0$$

Para evitar la trivialidad $f_\theta(img) = 0; \forall img$ añadimos un margen $\alpha > 0$ de la siguiente forma:

$$||f_\theta(anchor) - f_\theta(positive)||^2 - ||f_\theta(anchor) - f_\theta(negative)||^2 + \alpha \leq 0$$

Con esto, definimos una función de pérdida que solo se activará cuando la desigualdad no se mantenga, con lo que:

$$Loss(a, p, n) := max(||f_\theta(a) - f_\theta(p)||^2 - ||f_\theta(a) - f_\theta(n)||^2 + \alpha, 0)$$

3.3. Adecuación de la arquitectura de red a la base de datos escogida

Las redes siamesas se han usado sobre todo en tareas de reconocimiento de caras y en verificación de firmas a mano [3]. Estos dos problemas presentan las siguientes características fundamentales:

- La red tiene que distinguir si dos imágenes se corresponden a la misma entidad o no (dos fotografías de dos personas se corresponden a la misma, dos firmas se han realizado por la misma persona)
- La red va a ver ejemplos de entidades no vistas durante el entrenamiento. Por ejemplo, tiene que distinguir si dos fotos de personas son de la misma persona, sin haber visto fotografías de esa persona previamente. En modelos de clasificación clásicos la red puede enfrentarse a fotografías nunca vistas, pero de una clase de la que ha visto otras fotografías. Así, esta es una diferencia fundamental con otros modelos de *machine learning* con los que hemos trabajado
- Potencialmente, la red tiene pocos ejemplos de cada entidad de los que aprender.

En nuestra base de datos no se presentan las características fundamentales que hemos expuesto. La red se va a enfrentar a imágenes de clases que ya hemos visto previamente (fotografías nuevas de botas, por ejemplo. Pero la red ha sido entrenada con otras fotografías de otros tipos de botas). Además, tenemos muchos ejemplos de cada una de las diez clases con las que trabajamos.

Entonces, ¿por qué trabajamos con esta base de datos? En un primer momento, exploramos bases de datos más adecuadas para este problema, que no tuviesen que ver con el reconocimiento de caras o de firmas manuales. Por ejemplo, con una base de datos de figuras *Lego* [4]. Sin embargo, esta base de datos era demasiado compleja, tal y como nos indicó el profesor de teoría de la asignatura (nos indicó el problema de que muchas figuras salían en posturas demasiado diferentes, como por ejemplo, de cara a la cámara y de espaldas). Nos encontramos con problemas parecidos buscando bases de datos más adecuadas a las características fundamentales presentadas previamente.

Por tanto, escogemos usar esta base de datos porque, a pesar de no ser adecuado el uso de la arquitectura para este problema, podemos profundizar en todos los aspectos expuestos en *"2.3. Motivación"*. De hecho, y como se comentará más adelante, la base de datos ha supuesto una gran dificultad en lo que tiempos de entrenamiento se refiere. Con esto, dudamos mucho de haber sido capaces de realizar un estudio en una base de datos más adecuada teniendo en cuenta los recursos computacionales de los que disponemos (principalmente, los que *Google Colab* nos otorga).

En *"4.4. Conclusiones"* vemos que con nuestra técnica *baseline* se obtienen resultados muy pobres. Por lo tanto, la base de datos escogida no supone un problema trivial para la arquitectura de red que estamos empleando.

Una posible mejora que no exploramos por falta de tiempo es realizar el entrenamiento considerando 9 de las 10 clases, y evaluar el comportamiento sobre imágenes de la clase que nunca ha visto la red. Por ejemplo, nunca mostrar a la red imágenes de botas, y en evaluación, comprobar si la red es capaz de distinguir efectivamente imágenes de botas del resto de clases que sí ha visto durante el entrenamiento.

3.4. Métrica *Silhouette*

Usaremos esta métrica para medir la calidad de los clusters obtenidos con los modelos entrenados. Una buena clusterización es aquella en la que los elementos del mismo cluster están muy cerca entre sí (distancia intra-cluster pequeña) y los elementos de clusters distintos están lejos entre sí (distancia inter-cluster grande). Todo esto se resume en esta métrica.

Dado un punto i, definimos su índice de Silhouette $s(i)$ como:

$$s(i) := \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

donde $a(i)$ es la distancia media del punto i al resto de puntos del mismo cluster

$$a(i) := \frac{1}{|C_i|} \sum_{j \in C_i} d(i, j)$$

y $b(i)$ es la mínima distancia del punto i a un punto de un cluster distinto:

$$b(i) := \min_{j \in C_k; C_k \neq C_i} d(i, j)$$

Al ser $a(i), b(i)$ reales positivos, es claro que $s(i) \in [-1, 1]$. Como queremos que $a(i)$ sea pequeño y $b(i)$ sea grande, el mejor valor para $s(i)$ es 1.

Una vez que hemos definido el índice de Silhouette para un punto, definimos la métrica de Silhouette para toda una clusterización como la media de los índices de Silhouette punto a punto:

$$s = \frac{1}{n} \sum s(i)$$

Cuando $s \approx 0$ significa que las fronteras entre los distintos clusters no están muy diferenciadas.

4. Entrenamiento usando triples aleatorios

4.1. Detalles de implementación

Para la creación de los triples aleatorios, diseñaremos un `Dataset` de `Pytorch` para definir dichos triples aleatorios. Pasaremos de tener un conjunto de datos formado por pares (Imagen, Etiqueta) a tener un conjunto de datos formado por triples (Ancla, Positivo, Negativo). Notar que en dicha transformación las etiquetas pierden el protagonismo, lo que será clave a la hora de adaptar el bucle de entrenamiento, que ignorará dichas etiquetas por completo.

En dicho *dataset* definiremos un tamaño del conjunto de datos arbitrario. Podríamos considerar todos los triples del conjunto de datos. Pero esto supondría un *dataset* de tamaño intratable. Además, dejaría de ser un conjunto con triples aleatorios y pasaría a ser un conjunto con todas las combinaciones de triples.

Además de esto, el único detalle interesante es el pre-cómputo de las listas de índices en función de la clase. Es decir, tenemos acceso a los índices de las imágenes de cada clase precomputados, para que el cálculo de los triples sea más rápido.

Una vez hecho esto, el bucle de entrenamiento es el usual. Pasamos a la red *minibatches* conteniendo los triples, la red calcula su representación, y aplicamos la función de pérdida a cada triple. Dicha función de pérdida está implementada en la clase `TripletLoss` y `TripletLossCustom` (la última se encarga de manejar el error de todo un *minibatch*).

Utilizaremos para su entrenamiento *minibatches* de 32 triples y un total de 10 épocas.

Fijamos el tamaño del *embedding* a 2, y el margen a 0.001. Para optimizar el modelo usamos *Adam* con un *learning rate* inicial de 0.001. Esto porque en *"5.2.* Hyperparameter tuning*"*, hemos hecho una exploración de parámetros para el modelo que usa triples difíciles online, llegando a que los parámetros óptimos (para ese modelo) son dichos valores. Lo ideal habría sido realizar una exploración de parámetros para los triples aleatorios. Sin embargo, por tratarse del modelo que vamos a usar de base con el que comparar, y por lo lento del proceso de exploración de parámetros (esto lo comentaremos en *"5.2.* Hyperparameter tuning*"*), nos conformamos con los parámetros encontrados para el otro modelo.

4.2. Resultados del entrenamiento

La curva de aprendizaje se muestra en la siguiente gráfica:

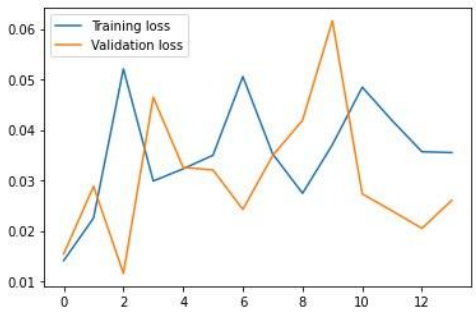


Figura 1: Curva de aprendizaje para el entrenamiento del modelo usando triples aleatorios. En azul, la función de pérdida en entrenamiento. En naranja, la función de pérdida en validación

En dicha curva de aprendizaje se muestra un claro comportamiento errático en el aprendizaje. Esto puede deberse a los siguientes motivos:

- El valor del *learning rate* es demasiado alto y por eso el ajuste del error con *back propagation* provoca dicho comportamiento errático
- No hemos entrenado durante suficientes épocas. Si entrenásemos durante más épocas, podríamos ver una tendencia global no tan errática
- El uso de triples aleatorios no es adecuado para optimizar la red, y por ello obtenemos dicho comportamiento errático

Comprobamos que no hemos usado pocas épocas entrenando durante 50, en vez de 10. La curva de aprendizaje en este caso es la siguiente:

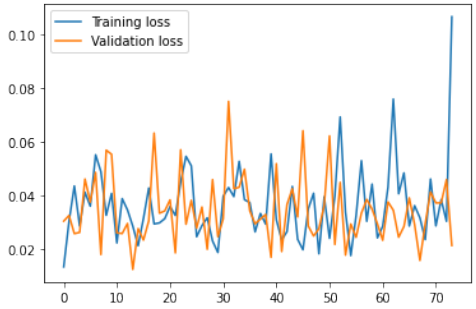


Figura 2: Curva de aprendizaje para el entrenamiento del modelo usando triples aleatorios. En este caso, en vez de entrenar durante 10 épocas, entrenamos durante 50

Seguimos observando el mismo comportamiento errático, y por tanto, descartamos que dicho comportamiento se explique por el uso de insuficientes épocas en el entrenamiento. Respecto al *learning rate*, mostramos ahora el resultado de entrenar durante 10 épocas con un *learning rate* mucho más bajo (en concreto, 0.0001). La curva de aprendizaje en este caso es la siguiente:

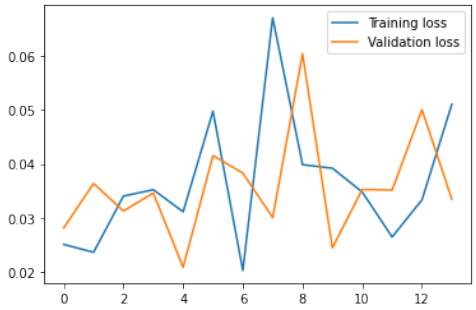


Figura 3: Curva de aprendizaje para el entrenamiento del modelo usando triples aleatorios. En este caso, en vez de entrenar usando un *learning rate* de 0.001, usamos 0.0001. De nuevo, estamos entrenando durante 10 épocas

El comportamiento errático sigue mostrándose con claridad. Por tanto, descartamos las dos primeras hipótesis para explicar el comportamiento errático, y consideramos la más probable la última: el uso de triples aleatorios no es adecuado para la optimización de la red.

Notar que en el *Notebook* entregado usamos, tanto para el margen como para el *learning rate* los parámetros originales. Si se quieren obtener los resultados que hemos mostrado para descartar hipótesis basta con modificar dichos parámetros en la celda con todos los hiperparámetros del *Notebook*.

Obtenemos un valor de la función de pérdida, en el conjunto de test, de 0.0621. Dicho valor se consigue usando la función `test_model`, que se encuentra en nuestra librería, en el fichero `core.py`. Más adelante, en *"4.3.1. Resultados de clasificación"*, mostraremos más métricas del modelo obtenido, una vez que lo hayamos adaptado a una tarea de clasificación.

4.3. Adaptación a una tarea de clasificación

Una vez entrenada la red, disponemos de una red que transforma imágenes en un espacio $\mathbb{R}^d$ con d pequeña. Además, tenemos la propiedad fundamental que ya hemos comentado varias veces: imágenes de la misma clase tienen representaciones cercanas, e imágenes de clases distintas tienen representaciones lejanas.

Por ello, parece adecuado usar el algoritmo *k-NN* para adaptar nuestra red, que simplemente calcula cierta representación de las imágenes, para una tarea de clasificación. Esto es lo que hacemos en la clase `EmbeddingToClassifier`, que toma una red de este tipo y realiza la adaptación que ya hemos comentado.

Para ello, toma la red y el conjunto de imágenes que usaremos como base de instancias para *k-NN*. Calculamos y almacenamos las representaciones de todas las imágenes. Cuando llega un nuevo ejemplo (una nueva imagen), calculamos su representación y aplicamos *k-NN* sobre nuestro conjunto almacenado de representaciones.

En nuestro caso, usaremos $k = 3$. Los resultados de realizar esta adaptación a la tarea de clasificación se muestran a continuación.

4.3.1. Resultados de clasificación

Nos aprovechamos de los cálculos que realizamos en la adaptación del modelo a clasificación para mostrar la gráfica del *embedding* obtenido sobre el conjunto de entrenamiento:

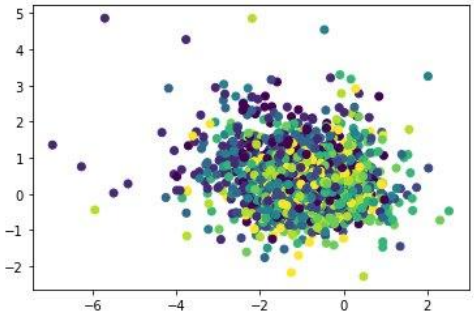


Figura 4: Gráfica del *embedding* calculado por la red sobre el conjunto de entrenamiento. Los colores muestran la clase asociada a cada uno de los puntos que mostramos

Como se puede ver en el gráfico anterior, las distintas clases se hallan muy juntas entre sí y entremezcladas, sin una separación clara por clase, por lo que la red no está separando correctamente las distintas clases del dataset. Este comportamiento era esperable debido a los malos resultados obtenidos previamente en el entrenamiento, y que mostrábamos en *"4.2. Resultados del entrenamiento"*

Por otra parte, si ahora extraemos algunas métricas como son el *accuracy* del clasificador, el Área bajo la curva ROC o la métrica Silohuette sobre el modelo de clasificación KNN obtenemos lo siguiente:

Dataset	Accuracy	Área bajo la curva ROC	Silhouette
train	0.125	0.5562	-0.1894
test	0.1042	0.5075	-0.1014

Cuadro 1: Métricas adicionales del modelo entrenado usando triples aleatorios

Como podemos observar, el Accuracy obtenido tanto en entrenamiento como en test por el clasificador es muy bajo, de un 10 %. Por otro lado el Área bajo la curva ROC es de un 0.5, lo que nos indica que el comportamiento que tiene el clasificador es prácticamente aleatorio. Finalmente en la métrica Silohuette obtenemos un -0.1014 lo que nos indica que las fronteras de las distintas clases se encuentran entremezcladas, es decir, los elementos de las distintas clases no están bien separados unos de los otros. No obtenemos *clusters* de calidad en el *embedding* calculado.

En definitiva, estamos obteniendo unos resultados muy malos debido a una red que calcula un *embedding* muy pobre. Como el algoritmo *k-NN* depende fundamentalmente de la calidad de dicho *embedding*, era de esperar, a la vista de los resultados de *"4.2. Resultados del entrenamiento"*, un comportamiento muy malo en la adaptación a clasificación.

4.4. Conclusiones

Los resultados obtenidos por el modelo de clasificación usando la Red Siamesa entrenada con triples aleatorios son muy pobres, como hemos mostrado tanto en *"4.2. Resultados del entrenamiento"* como en *"4.3.1. Resultados de clasificación"*. Claramente no se pueden tomar como una solución al problema planteado en este proyecto.

No obstante este modelo constituye un buen punto de partida sobre el cual mejorar los resultados obtenidos. En *"4.2. Resultados del entrenamiento"* hemos defendido la hipótesis de que el comportamiento errático está causado por la inconveniencia de los triples aleatorios como mecanismo de selección de dichos triples. Por tanto, veremos más adelante si un mecanismo más adecuado de selección de triples consigue optimizar de forma correcta la red con la que estamos trabajando.

Cabe destacar de nuevo que no hemos realizado una exploración de los hiperparámetros que definen el modelo y su entrenamiento. Principalmente, el valor del margen para la función de pérdida, el *learning rate* y la dimensión del *embedding* resultante. No tenemos tiempo para realizar una exploración que requiere mucho tiempo de cómputo. Además, consideramos que este mal comportamiento no se va a corregir con una mejor selección de parámetros.

5. Entrenamiento usando triples *online*

5.1. Detalles de implementación

En este caso cambiaremos la estrategia para generar los triples (*anchor*, *positive*, *negative*). En el uso de triples aleatorios, creábamos de forma *offline* un conjunto de datos de triples, en el que ya no necesitábamos información de las etiquetas (pues va implícita en la estructura de triples). En este caso pasaremos a usar una estrategia *online*.

Para ello, realizaremos un bucle clásico de entrenamiento:

1. Tomamos un *minibatch* de imágenes etiquetadas,
2. Pasamos las imágenes por la red obteniendo los valores de salida
3. Con los valores de salida y las etiquetas, calculamos la función de pérdida
4. Propagamos hacia atrás la pérdida

Si en los triples aleatorios definíamos un *dataset* para manejar los triples, en este caso usaremos una función de pérdida de `Pytorch` para dicho manejo de los triples. Esta función de pérdida la definimos en la clase `OnlineTripletLoss`.

La estrategia es la siguiente. Por cada imagen del *batch*, calcularemos su triple más difícil dentro de dicho *batch*. Es decir, consideramos una a una las imágenes del *batch* como anclas. Dada un ancla, buscamos la imagen del *batch* de la misma clase más alejada (positivo difícil), y la imagen de distinta clase más cercana (negativo difícil). Con este triple difícil calculamos el *Triplet Loss*. Devolvemos la media de las pérdidas por cada triple.

Con esto conseguimos generar, de forma *online*, triples difíciles, pero no extremadamente difíciles, pues el espacio de búsqueda de positivos y negativos se restringe al *batch*. Para tener un buen compromiso en la dificultad, usaremos un tamaño de *batch* considerablemente grande, en concreto, de 1024. Además, los *minibatches* serán distintos entre épocas, y por tanto conseguimos ver combinaciones distintas entre épocas (esto no lo lograríamos usando minería de triples *offline*).

Para acelerar el entrenamiento, por cada *minibatch* que pasamos a la función de pérdida, realizamos pre-cómputos para que ciertas operaciones no se repitan por cada elemento del *minibatch*. En concreto, y como ya hemos hecho previamente, calculamos una lista de listas, en el que cada lista almacena los índices de las imágenes de una clase en concreto. También calculamos una lista de listas, en el que cada lista almacena los índices de todas las imágenes que no son de una clase en concreto. Esta estrategia hace que los tiempos de entrenamiento se reduzcan aproximadamente 10 veces.

Además, para que esta estrategia sea efectiva, es bueno usar *minibatches* de gran tamaño, para que el tiempo que perdemos pre-computando se amortice con muchos cálculos ahorrados. Y ya usábamos *minibatches* grandes para encontrar triples de suficiente dificultad.

5.2. *Hyperparameter tuning*

Como este va a ser nuestro modelo final, vamos a considerar también hacer un *Grid Search* de los parámetros más importantes del modelo, usando como técnica de validación *4-Fold Cross Validation*. Los parámetros más relevantes a explorar consideramos que son: el learning rate, la dimensión del embedding y el margin (el parámetro α de la fórmula de *TripletLoss*).

Los valores que, inicialmente, consideramos explorar son:

- Margen: $\{0.001, 0.01, 0.1\}$
- *Learning Rate*: $\{0.0001, 0.001, 0.01\}$
- Dimensión del *embedding*: $\{2, 3, 4\}$

Decimos que consideramos *"inicialmente"* explorar porque ciertas combinaciones no las hemos explorado, al ver que ciertos valores de los parámetros producían resultados muy malos, y por tanto, consideramos que no merecía seguir explorando una zona potencialmente mala.

Para evaluar cada combinación de parámetros del Grid vamos a emplear *4-fold Cross Validation*, dividiendo el conjunto de todas las imágenes de entrenamiento (un total de 48000) en 80 % para Training y 20 % para validación.

Lo ideal habría sido usar 5 o 10 *fold cross validation*, al menos 10 épocas de entrenamiento. Sin embargo, haciendo cálculos, llegamos a la conclusión de que esto no era viable. Estimamos que tardábamos un total de *10 minutos por época*.

En total queríamos explorar tres posibles valores de cada parámetro, lo que hace un total de *27 combinaciones*. Por otro lado se consideró emplear 10 épocas y la estrategia 5-fold, el tiempo estimado de entrenamiento era el siguiente:

$$\frac{10\,min * 10\,epocas * 5\,folds * 27\,parametros}{60\,min * 24\,horas} = 9.375 dias$$

Como esta cantidad de tiempo de entrenamiento no se podía asumir, pensamos en reducir el número de épocas a 7, pues consideramos que este número seguía siendo apropiado para sacar conclusiones:

$$\frac{10\,min * 7\,epocas * 5\,folds * 27\,parametros}{60\,min * 24\,horas} = 6.56 dias$$

Finalmente decidimos reducir el número de folds a 4 y hacer 4-fold Cross Validation, pero consideramos que ya no podíamos reducir ningún valor más sin perjudicar en exceso la búsqueda:

$$\frac{10\,min * 7\,epocas * 4\,folds * 27\,parametros}{60\,min * 24\,horas} = 5.25 dias$$

Podemos ver que sigue siendo un tiempo intratable. Sin embargo, disponemos de dos cuentas de *Google Colab* para explorar en paralelo, por lo que el tiempo se reduce a 2.625 días, aproximadamente. Lo seguimos considerando un tiempo no razonable, pero durante el proceso de

exploración supervisamos los resultados para podar ciertas zonas de exploración poco prometedoras de forma manual. Con ello, todo el conjunto de parámetros explorados y sus resultados se muestran en la siguiente tabla:

Embedding Dimension	Learning Rate	Margin	Loss
2	0.0001	0.01	0.007372344437
2	0.0001	1	0.4078908339
3	0.0001	1	0.354682561
2	0.0001	0.1	0.04381236387
3	0.0001	0.1	0.05420429958
4	0.0001	1	0.3357170224
2	0.0001	0.001	0.01420788498
2	0.001	1	0.344632715
3	0.0001	0.001	0.0181999413
4	0.0001	0.001	0.01199295989
2	0.001	0.001	0.001895822257
3	0.001	0.001	0.001268613858
4	0.001	0.001	0.001237501063
2	**0.01**	**0.001**	**0.0007436529413**
3	0.01	0.001	0.006536031724
4	0.01	0.001	0.001353246829
3	0.0001	0.01	0.02716585406

Cuadro 2: Resultados de la exploración de hiperparámetros usando *4-fold Cross Validation*. En negrita, la combinación de parámetros que mejor resultados proporciona

Como vemos en la anterior tabla, la mejor combinación de parámetros es la siguiente:

- Margen: 0.001
- *Learning Rate*: 0.01
- Dimensión del *embedding*: 2

5.3. Resultados del entrenamiento

La curva de aprendizaje, tras haber entrenado durante 10 épocas, se muestra en la siguiente figura:

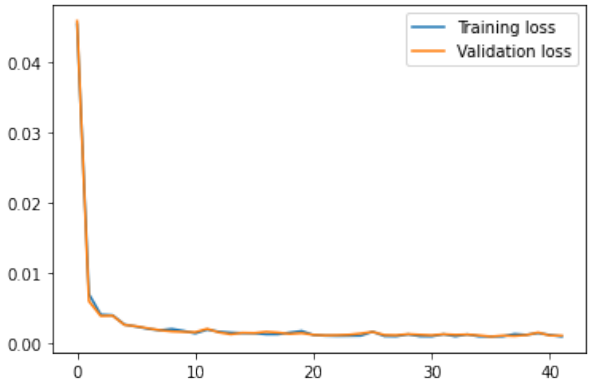


Figura 5: Curva de aprendizaje para el entrenamiento del modelo usando triples difíciles *online*. En azul, la función de pérdida en entrenamiento. En naranja, la función de pérdida en validación. En el eje x mostramos las veces que se calculan las métricas, que corresponden con 10 *batches*

En primer lugar, vemos que no hemos caído en problemas de sobre-aprendizaje. En segundo lugar, vemos que tras los 10 primeros *batches*, la red no mejora apenas. Por tanto, podríamos haber entrenado durante muchas menos épocas, obteniendo los mismos resultados.

En el conjunto de *test* obtenemos un valor de la función de pérdida de 0.00102. Nos aproximamos prácticamente al límite de la función de pérdida debido al margen escogido para dicha función de pérdida. Por tanto, en este sentido estamos satisfechos con los resultados obtenidos.

5.3.1. Resultados de clasificación

De nuevo, realizamos una adaptación de la red a una tarea de clasificación. Este proceso ya fue explicado en *"4.3. Adaptación a una tarea de clasificación"*, y por tanto no lo volvemos a explicar.

Comenzamos mostrando el gráfico del *embedding* obtenido en la siguiente figura:

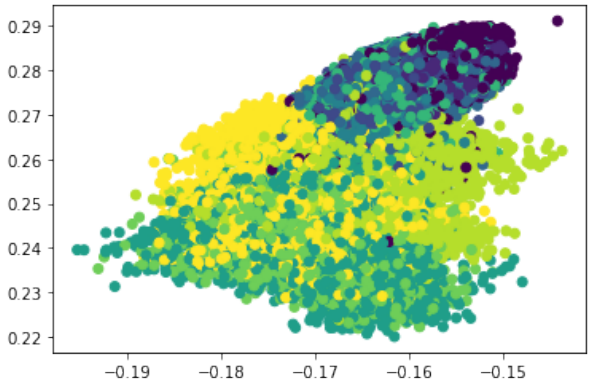


Figura 6: Gráfica del *embedding* calculado por la red sobre el conjunto de entrenamiento. Los colores muestran la clase asociada a cada uno de los puntos que mostramos

Se pueden observar que obtenemos *clusters* coherentes, pero que se solapan entre sí. Por tanto, esperamos obtener una métrica *Silohuette* muy mala, cercana a cero.

Ahora, en la siguiente tabla, mostramos algunas métricas obtenidas en la adaptación a clasificación:

Dataset	Accuracy	Área bajo la curva ROC	Silhouette
train	0.266	0.665	-0.095
test	0.266	0.663	-0.099

Cuadro 3: Métricas adicionales del modelo entrenado usando triples difíciles *online*

Podemos observar que tampoco se aprecia sobre-aprendizaje en la adaptación de la red a una tarea de clasificación. Sin embargo, los resultados siguen siendo muy malos. Era de esperar el mal valor de la métrica de *Silohuette*, por lo que hemos visto en la gráfica del *embedding*. Sin embargo, esperábamos obtener un mejor valor de *accuracy*. Teniendo en cuenta los resultados mostrados en *"1. Métricas adicionales del modelo entrenado usando triples aleatorios"*, estamos duplicando el *accuracy* pero manteniendo un valor muy bajo.

5.4. Conclusiones

Como hemos podido observar en este experimento, los resultados han mejorado algo en comparación con el modelo entrenado con Triples aleatorios, (estamos duplicando el *accuracy* del modelo) pero siguen siendo muy pobres y claramente no se podría tomar como una solución al problema de clasificación multiclase planteado.

Un claro problema, que vemos en *"6. Gráfica del* embedding *calculado por la red sobre el conjunto de entrenamiento. Los colores muestran la clase asociada a cada uno de los puntos que mostramos"*, es que obtenemos *clusters* coherentes (distancia *intracluster* pequeña), pero muy solapadas entre sí. Como comentaremos más adelante en *"7. Conclusiones"*, pensamos que modificando el valor del margen conforme avanzan las épocas de entrenamiento.

Ante estos resultados nos vemos obligados de nuevo a cambiar la estrategia, como comentamos en *"6.1. Motivación"*.

6. Entrenamiento usando una red mucho más simple

6.1. Motivación

Los malos resultados que ya hemos comentado en *"5.4. Conclusiones"* nos fuerza a buscar posibles mejoras al modelo entrenado. Probamos en el último momento con una arquitectura de red mucho más simple. Esto lo hacemos siguiendo nuestra intuición, pues el conjunto de datos es relativamente sencillo (no deja de ser una alternativa al famoso conjunto de datos *MNIST*). Y como comentaremos más adelante, conseguimos resultados mucho mejores con esta alternativa.

Al haber explorado esta alternativa en el último momento, no tenemos tiempo para realizar todos los experimentos que habríamos considerado ideales. Entre ellos, haber realizado pruebas con la arquitectura de red (hemos usado la primera red que planteamos, no usamos técnicas de regularización sobre la red, por ejemplo), haber realizado exploración de los hiperparámetros con *Cross Validation*, ...

6.2. Detalles de implementación

Para este nuevo enfoque al problema hemos creado una CNN mucho más simple que la que estábamos utilizando hasta el momento que hemos denominado *LightModel*. La arquitectura de dicha red se muestra en la siguiente tabla:

Layer number	Layer Type	kernel size	Input | Output channels or neurons
1	Conv2D	3	1 | 4
2	Relu	-	-
3	Conv2D	3	4 | 8
4	Relu	-	-
5	Conv2D	3	8 | 16
6	Relu	-	-
7	Conv2D	3	16 | 32
8	Relu	-	-
9	MaxPooling	2	-
10	Flatten	-	32 | 1
11	Dense	-	3200 | 2

Cuadro 4: Descripción de la arquitectura del modelo *LightModel*

Como ya hemos comentado, esta es la primera arquitectura que diseñamos, y con la que conseguimos buenos resultados. Lo ideal habría sido probar con distintas modificaciones en la arquitectura para refinarla. Sin embargo, por falta de tiempo, nos conformamos con esta primera propuesta que da buenos resultados.

Por lo demás el resto de etapas de entrenamiento y posterior adaptación a un modelo de clasificación KNN serán iguales al las del los modelos anteriores, y por tanto, no volvemos a explicar este proceso.

6.3. *Hyperparameter tuning*

De nuevo, y como ya hemos comentado, por falta de tiempo no realizamos *Hyperparameter Tuning* con *Cross Validation* para este nuevo modelo, que habría sido lo ideal. Por ello, decidimos nosotros los parámetros que vamos a emplear.

Con esta nueva arquitectura decidimos mantener el *learning rate* del modelo anterior. En el *hyperparameter tuning* anterior vimos que variar este parámetro no era muy significativo. Además, estuvimos explorando el efecto del margen en el error que se obtenía durante el proceso de entrenamiento. Llegamos a la conclusión de que este error era siempre menor cuanto más pequeño fuera el valor del margin y que no se producía overfitting. Por tanto, decidimos establecer como valor final para el margen 10^{-5}.

6.4. Resultados del entrenamiento

Comenzamos mostrando la curva de aprendizaje obtenida, en la siguiente figura:

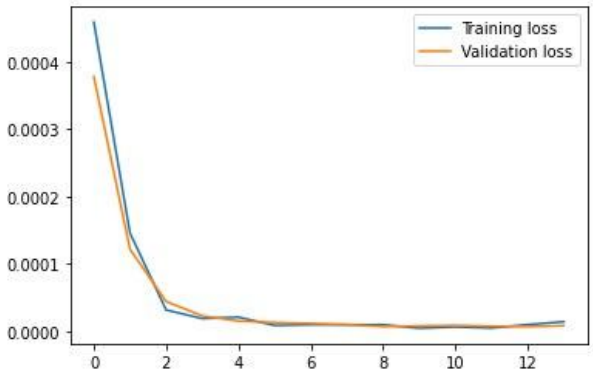


Figura 7: Curva de aprendizaje para el entrenamiento del modelo usando triples difíciles *online*. Además, usamos la nueva arquitectura de red. En azul, la función de pérdida en entrenamiento. En naranja, la función de pérdida en validación. En el eje x mostramos las veces que se calculan las métricas, que corresponden con 10 *batches*

Podemos ver que la red no comete sobreajuste, a pesar de que no hemos incluido elementos para tratar el sobre-aprendizaje en la arquitectura. Sin embargo, gracias a la simplicidad de la arquitectura, esto no supone un problema.

Hemos entrenado la red durante tres épocas, lo hemos considerado así debido a la gran rapidez con la que desciende el error al comienzo y porque tras la primera época la variación es mínima, por lo que entrenar durante más tiempo no iba a suponer una mejora considerable. En la curva de aprendizaje se ve claramente que a partir de los 40 primeros *batches* no hay mejora alguna.

Por otra parte, tras el entrenamiento, se obtiene un error en el conjunto de Test de 8.61×10^{-6}, que nos confirma que no ha habido sobreajuste en el entrenamiento, como ya sospechábamos por los resultados mostrados en la curva de aprendizaje.

6.4.1. Resultados de clasificación

Comenzamos mostrando, como hacíamos en secciones anteriores, la gráfica del *embedding* obtenido:

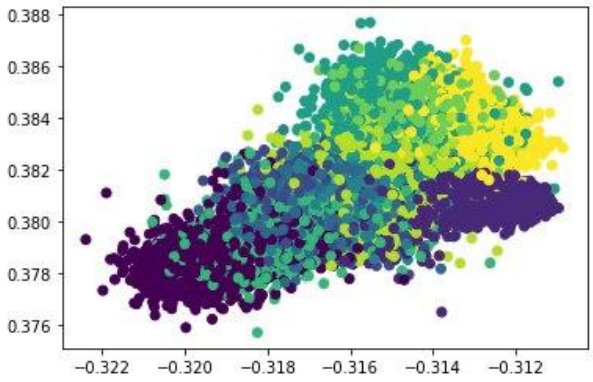
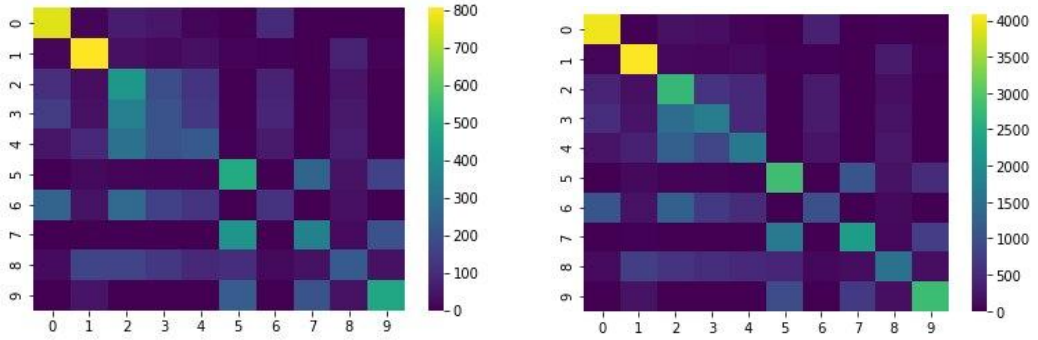


Figura 8: Gráfica del *embedding* calculado por la red sobre el conjunto de entrenamiento. Los colores muestran la clase asociada a cada uno de los puntos que mostramos

Como se puede observar las clases se encuentran mejor separadas entre sí que en los modelos anteriores, aunque sigue habiendo cierto grado de solapamiento entre clases. Es más, hay ciertas clases que parecen mezclarse con otras,

Mostramos ahora las matrices de confusión, pues puede ser que la red esté confundiendo clases que son parecidas (como con los distintos tipos de zapatos).



(a) Conjunto de entrenamiento

(b) Conjunto de test

Figura 9: Matrices de confusión obtenidas en la adaptación a la tarea de clasificación

Podemos observar que las dos primeras clases son bien distinguidas por la red entrenada. Sin embargo, en el resto de clases, no tenemos tanta precisión. Es destacable la confusión que tiene la red entre las clases 5, 7 y 9. Estas clases se corresponden con las sandalias, zapatillas tipo *sneakers* y botas. Por tanto, la red tiene problemas para distinguir estos tres tipos de calzados.

Finalmente, mostramos las mismas métricas que hemos calculado en los modelos previos en la siguiente tabla:

Dataset	Accuracy	Área bajo la curva ROC	Silhouette
train	0.51175	0.8476	-0.0157
test	0.4101	0.7588	-0.0159

Cuadro 5: Métricas adicionales del modelo ligero

Como se puede ver, el *accuracy* mejora considerablemente con respecto a los modelos anteriores, logrando un 41 % en el conjunto de *test*. Además el *área bajo la curva roc* también experimenta una mejora considerable, pues alcanza un 0.7588 en *test*, lo que nos indica que comienza a ser un mejor clasificador que los modelos anteriores. Como era de esperar, pues hemos visto que las clases están muy solapadas, el valor de *Silhouette* es muy malo (cercano a cero)

6.5. Conclusiones

En primer lugar, el modelo planteado logra una mejora considerable respecto de los dos modelos previos. Sin embargo, los resultados siguen sin ser satisfactorios, y no podemos considerar el resultado obtenido como una solución al problema planteado.

En segundo lugar, consideramos que con esta alternativa se abre una nueva línea a investigar con la que se podría llegar a obtener un clasificador apropiado si se elige una arquitectura mejor de red (manteniendo la simplicidad) o haciendo una búsqueda exhaustiva de los distintos hiperparámetros como se hizo con el modelo anterior. De tener el tiempo suficiente exploraríamos esta línea al parecernos la más prometedora.

Finalmente, podemos concluir que es fundamental adecuar la complejidad de la arquitectura de red a la complejidad del *dataset* con el que se trabaja. Uno de los errores que hemos cometido en esta práctica ha sido trabajar con *ResNet-18*, una arquitectura demasiado compleja para el problema a resolver. Por tanto, y a pesar de ser una red pre-entrenada, nos ha hecho perder mucho tiempo sin ofrecer resultados satisfactorios.

7. Conclusiones

Finalmente, a modo de resumen de todas las conclusiones que se han obtenido hasta el momento:

En primer lugar, nuestra hipótesis inicial sobre el uso de triples aleatorios queda confirmada. Es una técnica rápida para la generación de triples pero que no da buenos resultados, y por tanto es inútil más allá de servir como *baseline* con el que comparar futuros modelos.

En segundo lugar, el uso de triples difíciles *online* mejoró sustancialmente el modelo. Una buena selección de triples es clave para que la red mejore efectivamente. Además, la estrategia *online* nos permite tanto seleccionar triples de forma relativamente rápida como equilibrar la dificultad de los triples escogidos.

Sin embargo, vemos que aunque haya una mejora sustancial, no es suficiente. Al menos en la adaptación de la red a clasificación multiclase. Por tanto, buscamos mejorar los resultados de nuestro modelo, llegando a la conclusión de que es conveniente el uso de una red definida por nosotros, ligera y sin pre-entrenar.

Con esta red, mucho más simple, obtenemos unos resultados todavía mejores, en todas las métricas. Por tanto, aunque en general no sea una buena idea entrenar de cero un modelo, debido a la simplicidad de la base de datos es algo positivo: adecuamos la complejidad del modelo a las características de la base de datos y el entrenamiento desde cero no supone demasiado tiempo de cómputo. A pesar de esto, la red no obtiene unos resultados aceptables cuando realizamos la adaptación a clasificación.

Un problema que hemos comentado previamente es todo el tiempo que hemos dedicado a la exploración de parámetros y optimización del modelo que usaba como base *ResNet-18*. Si todo ese tiempo lo hubiésemos dedicado a mejorar la arquitectura de la red simple, a explorar sus parámetros y a mejorar el entrenamiento, estamos bastante convencidos que habríamos obtenido resultados mucho mejores.

Y para finalizar, en todos los casos hemos visto que, a pesar de obtener *embeddings* aceptables, siempre ha habido una claro solapamiento entre las distintas clases. Una posible solución para este problema, que nos planteamos pero que no nos ha dado tiempo a implementar, es el uso de un margen adaptativo. Esto es, comenzar con un margen bajo y, con el paso de las épocas de entrenamiento, aumentarlo para forzar a que la red separe las distintas clases. De conseguir esto, pensamos que los resultados en la adaptación a clasificación habrían sido sustancialmente mejores.

8. Referencias

[1] "zalandoresearch/fashion-mnist: A mnist-like fashion product database. benchmark." https://github.com/zalandoresearch/fashion-mnist. (Accessed on 01/15/2022).

[2] "zalandoresearch/fashion-mnist: A mnist-like fashion product database. benchmark." https://github.com/zalandoresearch/fashion-mnist#why-we-made-fashion-mnist. (Accessed on 01/15/2022).

[3] "Siamese neural network - wikipedia." https://en.wikipedia.org/wiki/Siamese_neural_network. (Accessed on 01/15/2022).

[4] "Lego minifigures — kaggle." https://www.kaggle.com/ihelon/lego-minifigures-classification. (Accessed on 01/15/2022).