Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Degree in Mathematics

Bachelor's Degree Thesis

Wordnet y Deep Learning: Una posible unión

Raquel Leandra Pérez Arnal

Supervised by Dario Garcia Gasulla y Claudio Ulises Cortés García

Enero, 2017

Thanks to...

Abstract

This should be an abstract in english, up to 1000 characters.

Keywords

keyword1, keyword2, keyword3, ...

Índice

1. Conocimientos previos

1.1 Redes neuronales

Una **red neuronal** es un modelo computacional inspirado en la forma de procesar información de las neuronas del cerebro. Las redes neuronales han generado una significativa cantidad de investigación e industria gracias a sus resultados en reconocimiento del habla, visión por computador y procesamiento de texto. Para dejar claro su funcionamiento explicaré primero el funcionamiento de una sola neurona, para después unirlo con la organización por capas de varias neuronas y finalmente explicar el algoritmo de *backpropagation*.

1.1.1 Una neurona

Definición 1.1. La unidad de computación básica de una red neuronal es una **neurona**, también llamada nodo o unidad. Ésta, recibe una entrada sobre la que le aplica una función para generar una salida.

Cada entrada ($\mathbf{x}$) tiene asociado un peso ($\mathbf{w}$), que es aprendido en base a su importancia relativa a otras entradas, a la salida y al objetivo de la red. La neurona aplica una función ($\mathbf{f}$) a la suma ponderada de las entradas.

Además de los elementos ya comentados tendremos un término de sesgo (w_0), cuyo objetivo es proveer cada neurona con un valor entrenable que no dependa de la entrada y facilitar el ajuste del modelo a los datos.

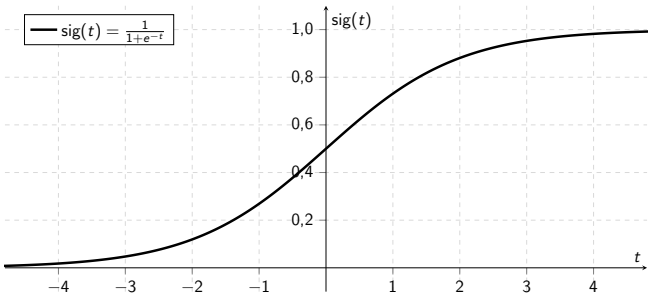
La salida de la neurona ($\mathbf{y}$) se calculara de la siguiente manera:

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = f\left(\sum_{i=1}^{N} \omega_i x_i + w_0\right)$$

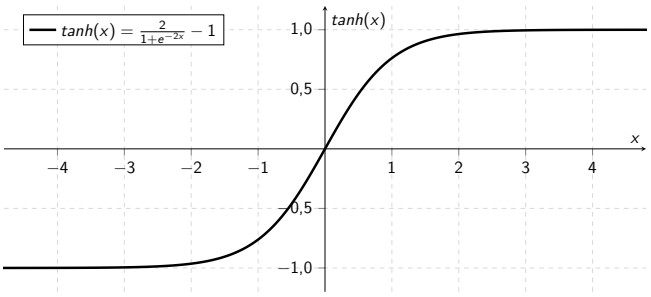
Llamaremos a $\mathbf{f}$ la **función de activación**, es una función (no lineal) diferenciable cuyo propósito es introducir no linealidad a la salida de la neurona. Ésto permite adaptar el modelo a problemas reales, puesto a que estos raramente son lineales.

Las funciones de activación más utilizadas son:

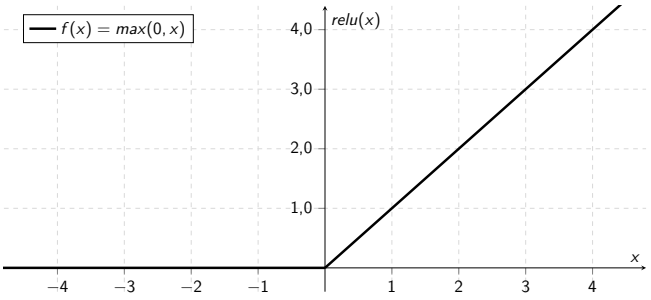
- Sigmoide: $f(x) = \frac{1}{1+e^{-x}}$



- Tanh: $f(x) = \frac{2}{1+e^{-2x}} - 1$



- ReLU: $f(x) = max(0, x)$



En la figura 1 podemos ver un ejemplo visual del comportamiento de una sola neurona.

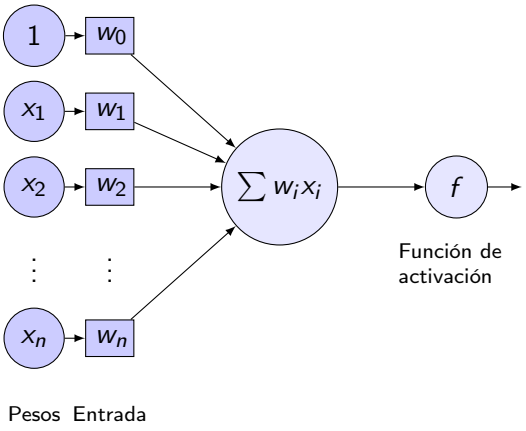


Figura 1: Ejemplo de una neurona

Dotando una neurona de una función de pérdida en la salida podríamos convertirla en un clasificador lineal, pero la verdadera potencia del método viene dada al unir diversas neuronas en lo que llamaremos capas.

1.1.2 Organización por capas

Las redes neuronales se modelizan como una colección de neuronas conectadas en un grafo acíclico, comúnmente organizado por capas. El tipo más común de capa es la capa completa(*fully-connected layer*), en el que todas las neuronas de una capa están conectadas con las de la capa posterior a ésta, mientras que

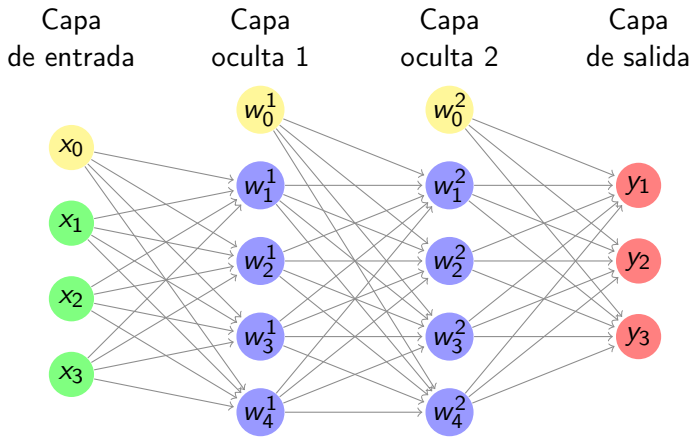


Figura 2: Ejemplo de red neuronal

no comparten ninguna conexión con las de su propia capa, como muestra la figura 2. La red neuronal más básica la podemos dividir en:

1. Neuronas de entrada: Proveen la red de información del exterior, consisten en los datos de entrada, todo su conjunto se conoce como la capa de entrada.
2. Neuronas ocultas: Las neuronas ocultas no tienen conexión directa con el exterior. Solo calculan y transfieren información de la entrada a la salida. La colección de las neuronas ocultas se denomina las capas ocultas.
3. Neuronas de salida: Colectivamente denominados capa de salida y son responsables de generar la salia de la red neuronal calculada a partir de la entrada y procesada por varias capas.

Matemáticamente la notación (tomada de [1]) varía ligeramente al añadir capas. Si tenemos una muestra de tamaño N y una red con M capas,definimos las activaciones $(a_j^{(k)})$ para cada neurona j de la capa k como:

$$a_j^{(k)}(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{N} w_{ji}^{(k)} x_i + w_{j0}^{(k)}$$

Ecuación 1

Donde $w_j i$ es el peso que va de la neurona i a la neurona j y $w_j 0$ es el sesgo.

Si tomamos $z_j^{(k)} = f(a_j^{(k)})$ podemos expresar las activaciones de la siguiente capa como:

$$a_j^{(k+1)}(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{N} w_{ji}^{k} z_j^{(k)} + w_{j0}^{(k)}$$

Ecuación 2

Después de procesar todas las capas la salida de la red sería:

$$\mathbf{y}(\mathbf{x},\mathbf{w}) = \mathbf{a}^{(M)} = \sum_{i=1}^{N} w_{ji}^{(M-1)} z_j^{(M-1)} + w_{j0}^{(M-1)} = \sum_{i=1}^{N} w_{ji}^{(M-1)} \left(\sum_{i=1}^{N} w_{ji}^{(M-2)} z_j^{(M-2)} + w_{j0}^{(M-2)} \right) + w_{j0}^{(M-1)}$$

Ecuación 3

Donde, para calcular la salida, habría que calcular todos los sumatorios encadenados hasta llegar a la entrada.

De esta forma nos queda una estructura formada por pesos ordenados en distintos tipos de capas. Cada peso de la red es un parámetro que necesitamos que la red aprenda, y de esta forma corresponda con el resultado esperado (permitiendo un cierto error). Para ello utilizaremos el algoritmo de *backpropagation*, que explicaremos en el siguiente apartado.

1.1.3 Entrenar la red: Backpropagation

Finalmente, queda explicar como entrenar los distintos parámetros para que se adapten a los datos, para ello se utiliza **backpropagation**.

La idea general del algoritmo consiste en obtener las derivadas parciales de la función de error respecto a los pesos utilizando la regla de la cadena. Con el objetivo de, una vez tengamos las derivadas, utilizar un algoritmo de minimización sobre la función de error.

Sea $E(\mathbf{w})$ nuestra función de error,que depende de los pesos $\mathbf{w}$, y $\mathbf{t}$ el vector objetivo, es decir, los valores que debería dar la red neuronal dada la entrada $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$.

Si tomamos el error de mínimos cuadrados tenemos:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$$

Ecuación 4

Que, como vimos en la ecuación 3, depende de los pesos de todas las capas. Esta estructura nos dificulta minimizar la función de una forma eficiente.

Para facilitar la notación, definimos $z_0 = 1$, $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$ y $E_n(w) = \frac{1}{2} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$, de esta forma tenemos:

$$a_j^{(k)} = \sum_{i=0}^{N} w_{ji} z_i^{(k)} \qquad E_n = \frac{1}{2} \sum_h (y_{nh} - t_{nh})^2 \qquad E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

Ecuación 5. Simplificaciones que utilizaremos

Donde $a_j^{(k)}$ a la activación de la neurona j de la capa k y el índice h corresponde a cada una de las coordenadas de los vectores $\mathbf{y}$ y $\mathbf{t}$.

Para calcular las derivadas de $E(\mathbf{w})$, calcularemos cada una de las de E_n. Como vemos en la siguiente ecuación:

$$\frac{\partial E_n}{\partial w_{ji}^{(k)}} = (y_{nj} - t_{nj}) \frac{\partial y_{nj}}{\partial w_{ji}^{(k)}}$$

Ecuación 6

Donde n es el índice de la muestra, j es el índice de la neurona y k es el índice de la capa.

Utilizando la ecuación 3, podemos calcular el valor que corresponde a la derivada del error respecto a los pesos de la última capa (que depende de la penúltima), sin embargo el cálculo de la derivada respecto a los pesos de capas anteriores se dificulta.

$$\frac{\partial E_n}{\partial w_{ji}^{(M)}} = (y_{nj} - t_{nj}) z_j^{(M-1)}$$

Ecuación 7

Para continuar desarrollando introducimos la siguiente notación:

$$\delta_j^{(k)} = \frac{\partial E_n}{\partial a_j^{(k)}}$$

Ecuación 8

Donde las δ se suelen llamar errores. Además si calculamos la derivada de las activaciones respecto a los pesos utilizando la simplificación de la fórmula 5 tenemos:

$$\frac{\partial a_j^{(k)}}{\partial w_{ji}^{(k)}} = z_i^{(k)}$$

Ecuación 9

Utilizando la regla de la cadena sobre $\frac{\partial E_n}{\partial w_{ji}^{(k)}}$ y utilizando 8 y 9 tenemos:

$$\frac{\partial E_n}{\partial w_{ji}^{(k)}} = \frac{\partial E_n}{\partial a_j^{(k)}} \frac{\partial a_j^{(k)}}{\partial w_{ji}^{(k)}} = \delta_j^{(k)} z_i^{(k)}$$

Ecuación 10

Utilizando 8 con los nodos de salida y 4, tenemos:

$$\delta_h^{(M)} = y_h - t_h$$

Ecuación 11

El resto de δs se pueden calcular utilizando la regla de la cadena sobre 8 y el valor de delta de la capa final 11. De esta forma tenemos:

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_l \frac{\partial E_n}{\partial a_l} \frac{\partial a_l}{\partial a_j}$$

Ecuación 12

Donde la l incluye todas las neuronas a las que está conectada j. Finalmente, uniendo todo obtenemos la **fórmula de backpropagation**:

$$\delta_j = f'(a_j) \sum_l w_{lj} \delta_l$$

Ecuación 13

En resumen, el algoritmo de *backpropagation* empieza con un *fordward pass*(una evaluación en la que se procesa la entrada desde el principio hasta generar la salida de acuerdo con el estado actual de la red) por toda la red, con el que se calcula la y_n y todas las activaciones $a_{ij}^{(k)}$, la predicción dada por la red se compara con la salida esperada (t_n) y se calcula un error (δ) para los pesos de la salida, luego se propaga este error utilizando 13. Finalmente utilizando 10 obtenemos los valores de las derivadas para todos los pesos.

Una vez calculadas las derivadas, se utiliza el *Stochastic Gradient Descent*, o otros métodos de minimización, para obtener los valores de los pesos que minimizan el error. El *Stochastic Gradient Descent* [1] es un método iterativo que, dada una función a minimizar $E(w)$ obtiene un mínimo local.

Cada nuevo valor para $w^{(i+1)}$ (nótese que en este caso utilizamos el super-índice para indicar la iteración a la que corresponde w, no como índice de la capa) se calcula de la siguiente forma (partiendo de una inicialización aleatoria):

$$w^{(i+1)} := w^{(i)} - \lambda \nabla E(w^{(i)})$$

Donde λ es un hiper-parámetro a añadir a nuestro modelo y $\nabla E(w^{(i)})$ es el gradiente de E, cuyos términos hemos calculado utilizando *backpropagation*. Nótese que al ser una optimización en un espacio multi-dimensional (por cada capa un peso por neurona, siendo cada peso un parámetro a optimizar), se requiere muchas muestras y ver varias veces cada muestra para minimizar el error.

1.2 Redes convolucionales

Definición 1.2. Una **red convolucional** es una secuencia de capas de diferentes tipos, donde, al menos una de las capas es una capa convolucional.

Las redes convolucionales son muy similares a las redes neuronales explicadas, las mayores diferencias son que contienen un tipo especial de neuronas (las neuronas convolucionales) y que asume explícitamente que la entrada son datos con una estructura concreta(como por ejemplo imágenes), lo que permite añadir ciertas propiedades a la arquitectura. Esto permite hacer un *fordward pass* más eficiente de implementar y reducir significativamente la cantidad de parámetros.

Una neurona convolucional es aquella que opera con una convolución[2].

Definición 1.3. Una **convolución** es una aplicación matemática cuya entrada son dos funciones de valores reales (**x** y **w**) y cuya salida es otra función (**s**).

Usualmente se denota como $s(t) = (x * w)(t)$.

En el contexto de redes convolucionales llamaremos a x *entrada*, a w kernel y a la salida s *feature map*.

Nótese que la definición no depende de la dimensión del espacio, es decir, que podemos definir una convolución para tantas dimensiones como necesitemos. En el caso concreto de una imagen I (de dos dimensiones) y un kernel K, la convolución quedaría como:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Ecuación 14. Convolución

Definición 1.4. Una **función de kernel** es aquella que cumple:

$$K(x,x') = \Phi(x)^T \Phi(x')$$

Donde $x, x' \in X$, siendo X un conjunto, y Φ es una función de base tal que $\Phi(x)^T \Phi(x') \in \mathbb{R}$. En el caso de las redes convolucionales se suele utilizar un *kernel* lineal.

Típicamente las neuronas convolucionales tienen una entrada limitada, es decir están conectadas solo a un subconjunto de las neuronas de la capa anterior, de esta forma cada neurona se centra solo en parte de la entrada, lo que significa el aprendizaje de las neuronas individuales cuando hay entradas de muchas dimensiones. En contraste, las neuronas de una capa *fully-connected* están conectadas a todas las neuronas de la capa anterior. De esta forma se reducen la cantidad de parámetros de la red.

Por su conectividad limitada, las capas convolucionales pueden centrarse en una parcela particular de la entrada. El conjunto de pesos (que sería equivalente al kernel comentado en la definición 1.3) aprendido por esta parcela puede ser relevante para las otras parcelas de la entrada, por tanto, podemos definir neuronas similares que utilicen el mismo kernel, pero que se enfoquen en otra parte de la entrada. Esta idea se conoce como "weight sharing",por que varias neuronas de la misma capa están definidas por un conjunto común de pesos, esto permite tener una considerable cantidad de neuronas utilizando los mismos parámetros. Esto puede ayudar a detectar los mismos patrones en diferentes partes de la entrada. Por ejemplo si tenemos una neurona que detecta ojos, al pasarla por una foto con una cara se activará dos veces.

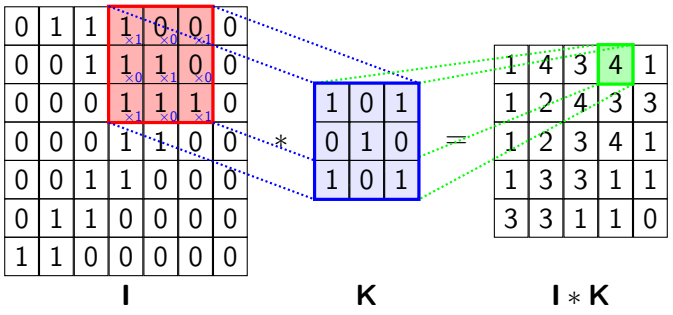


Figura 3: Ejemplo de la forma de operar una convolución sobre una imagen.

En la figura 3 podemos ver un ejemplo del funcionamiento de una convolución con un kernel de tamaño 9 sobre una imagen, podemos observar como se comparten los pesos a la hora de calcular el resultado. Dada una sección de la red, el *kernel* multiplica cada uno de sus valores por el valor correspondiente de ésta, para finalmente, sumar el resultado de todos los productos.

Finalmente queda comentar los tipos de capa más comunes en una red convolucional:

- Capa convolucional, capas formadas por neuronas convolucionales, la mayoría de capas de una red convolucional serán de este tipo.
- Capa *fully-connected*, es una capa donde todas las neuronas están conectadas entre ellas. Se suele utilizar como capa de clasificación.
- *Pooling Layer*, consiste en una capa que aplica una reducción matemática a su input(como una media o un max). El objetivo es aportar un cierto grado de invariancia espacial (imágenes equivalentes en diferentes posiciones), además reducen el tamaño de la salida de la capa, lo que provoca una reducción de la complejidad de la red. Un ejemplo de *pooling layer* sería la figura 4.

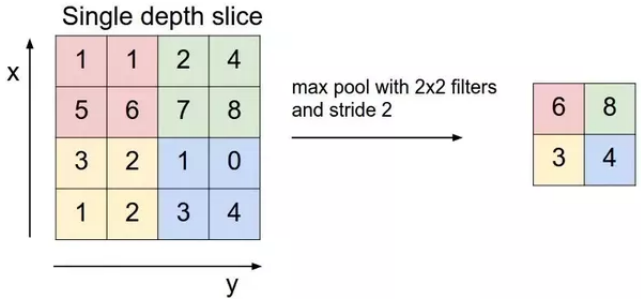


Figura 4

Otro concepto que utilizaremos a lo largo del documento es el de *feature*.

Definición 1.5. Llamaremos **feature** a los distintos subconjuntos de neuronas que ayudan a una red a cumplir con la tarea para la que fue diseñada, se generan al entrenar los pesos y pueden tener distintos grados de abstracción.

En redes convolucionales, la disposición por capas permite representar la información más compleja a partir de otra más simples. Por tanto puede acabar extrayendo *features* abstractas. En la figura 5 podemos

ver un ejemplo de *features* de distinto nivel de abstracción. Por ejemplo, si queremos clasificar diferentes animales, la presencia de *pico* sería una *feature*.

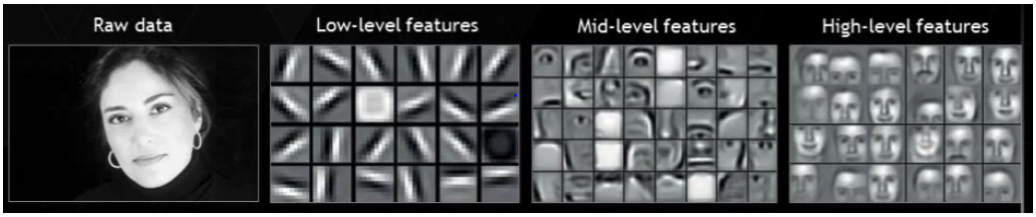


Figura 5: Ejemplo de features

1.3 Transfer Learning

En la práctica pocas personas entrenan una red profunda desde cero (con inicialización aleatoria), puesto a que éstas tienen unos requisitos bastante exigentes a la hora de ser entrenadas.

Los mayores problemas que puedes encontrar son:

1. A causa de la gran cantidad de parámetros a entrenar que tiene una red profunda necesitas un **conjunto de datos de gran tamaño**.
2. El **coste computacional** de entrenar la red, determinado por el número de parámetros, que en el caso de redes convolucionales pueden ser del orden de cientos de miles.
3. La búsqueda de **hiper-parámetros** (valores de los que depende el modelo) óptimos.
   La forma de obtener los hiper-parámetros es, entrenar la red para cada uno de los hiper-parámetros que hayamos considerado y, utilizando una partición de validación (o el método de *cross-validation*) comparar los distintos resultados.
   Ésto incrementa los problemas *1*, al necesitar una partición de validación necesitaremos más datos, y *2*, el coste computacional se multiplica por cada vez que haya que entrenar la red, que puede ser varias veces por cada hiper-parámetro que tengamos.

Por tanto, es común pre-entrenar una red convolucional en un conjunto de datos significativamente grande y usar la red convolucional como inicialización o como extractor fijo de características de la tarea de interés (como vemos en [3]).

Definición 1.6. Llamaremos **transfer learning** al campo de estudio que reutiliza el lenguaje de representación de un problema(que llamaremos problema origen o *Source*) para resolver otro(que llamaremos objetivo o *Target*).

A la hora de utilizar **transfer learning** tenemos dos componentes base: Un dominio $\mathcal{D}$ definido por un conjunto de instancias de datos con una distribución de probabilidades y una tarea $\mathcal{T}$, definida a partir de un conjunto de clases y una función objetivo.

Para un problema de *transfer learning* usaremos la notación(introducida en [4]):

- Problema origen: $(\mathcal{T}_{\mathcal{S}}, \mathcal{D}_{\mathcal{S}})$
- Problema objetivo: $(\mathcal{T}_{\mathcal{T}}, \mathcal{D}_{\mathcal{T}})$

Los dos casos más comunes de transfer learning son:

- **Fine-tuning**([5]): La primera estrategia consiste en inicializar los datos desde un estado no aleatorio (tomando los pesos ya entrenados), y a continuación, entrenar la red sobre estos datos. De esta manera puedes reducir significativamente el conjunto de datos necesario para entrenarla, sin embargo, sigue necesitando tiempo para optimizar los múltiples hiper-parámetros involucrados en el proceso y una cantidad significativa de recursos computacionales.
- **Feature Extraction**([4]) Consiste en procesar un conjunto de datos a través de una red neuronal ya entrenada y extraer valores de activación para que puedan ser utilizados por otro mecanismo de aprendizaje. Este método es aplicable a conjuntos de datos de cualquier tamaño([6], [7]), puesto a que cada dato es procesado independientemente. Además tiene un menor coste computacional, ya que no tiene que entrenar la red y no requiere la optimización de hiper-parámetros. Por estos motivos las aplicaciones de *transfer learning for feature extraction* están limitadas solo a las capacidades de los métodos que utilices encima de la representación profunda obtenida.

En nuestro caso nos centraremos en un tipo concreto de *transfer learning for features extraction* [8], que explicaremos más adelante en la sección2.1.

2. Trabajo Relacionado

2.1 Full-Network embedding

En esta sección explicaré el *full-network embedding* presentado en [11].

En general en *transfer learning for feature extraction* es común tomar los valores de activación de una sola capa cercana a la salida, como podemos ver en la figura 6, donde el proceso de *feature extraction* solo se hace en las últimas capas, *fully-connected*, mientras que las capas convolucionales(todas las anteriores) permanecen intactas. El resto de capas se descartan por "ser poco probable que contengan una representación mejor"[9], sin embargo es conocido que todas las capas de una red profunda pueden contribuir a caracterizar los datos de diferentes maneras. Esto implica que la representación más versátil y rica que puede ser generada por un proceso de *features extraction* debe incluir todas las capas de la red,es decir, debe definir un *full-network embedding*.

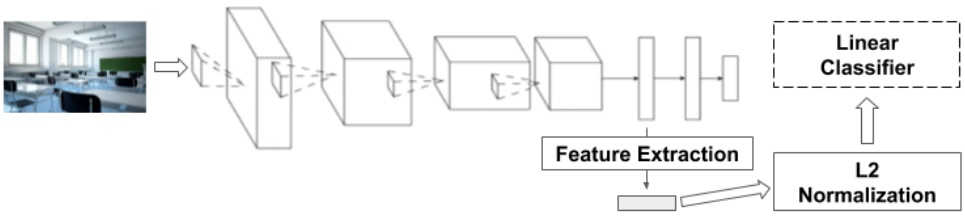


Figura 6: Estructura básica que se suele utilizar en *feature extraction*

Dado un conjunto de datos *t1*, queremos representarlo en el lenguaje aprendido para una tarea *t0*. Para ello el *full-network embedding* se divide en 4 pasos:

- El primer paso es hacer un **forward pass** de cada instancia de datos de *t1* a través del modelo entrenado en *t0*, guardando todos los valores de activación de cada capa de la red, (tanto las convolucionales como las *fully-connected*).
- El segundo paso consiste en un **spatial average pooling** en los filtros convolucionales. Como se ha explicado en el capítulo sobre redes convolucionales, un filtro convolucional genera varias activaciones para una entrada, para darle la capacidad de obtener información espacial. El objetivo de este paso es obtener un solo valor de activación para cada filtro, evitando así un aumento de la dimensionalidad. Los valores resultantes se concatenan a los de las capas *fully-connected* en un solo vector, para generar el *embedding* completo.
- El tercer paso es una **estandarización de características**. Puesto a que los valores del *embedding* provienen de distintos tipos de neuronas en distintos puntos de la red, la distribución de las activaciones pueden variar mucho y si se concatenasen todas algunas dominarían sobre las otras.

  El valor estandarizado de cada categoría se obtiene calculando la media y la desviación típica del conjunto de datos de entrenamiento, es decir, que se estandariza según el contexto de la feature en todo el conjunto de datos. De esta forma se pueden integrar los distintos valores en el *embedding*.

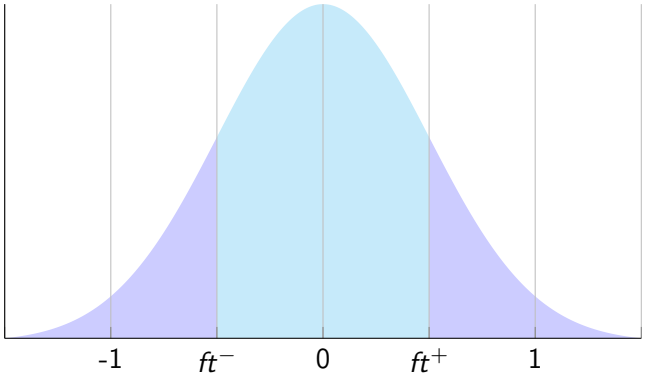


Figura 7

- Finalmente se aplica una **discretización de características** para evitar problemas al intentar explorar un espacio de dimensión tan grande(una sola capa convolucional puede llegar a tener del orden de 100,000,000 parámetros, por ejemplo [10]). Ésta discretización consiste en tomar unos límites que dependen de los datos, ft^- y ft^+ y reemplazar los valores que tienen un valor atípicamente bajo (todos los valores $x < ft^-$) por un -1, los que tienen un valor típico (todos los valores $ft^- < x < ft^+$) por un 0 y los que tienen un valor atípicamente alto (todos los valores $ft^+ < x$) por un 1, como podemos ver en la representación gráfica 7.

  Para encontrar los límites, los autores se basan en [8], donde dan una visión estadística sobre como evaluar la importancia de las diferentes características de una red convolucional comparando las activaciones para una clase concreta respecto al resto de clases del conjunto de datos. De esta forma los autores de [8] separan las características en tres conjuntos: *característico por presencia*, *no característico* y *característico por ausencia*.

Un ejemplo de la separación comentada en la *discretización de características* sería: si tenemos como datos objetivo un conjunto de coches tendremos que las features que se activen con las ruedas no serán representativas, puesto a que todos los coches tienen, las categorías que se activen con el techo de un coche serán representativas por ausencia para clasificar descapotables mientras que las que se activen para la capota serán representativas por presencia.

En la figura 8 podemos ver un diagrama de como se desarrollarían los diferentes pasos. Nótese la diferencia con la figura 6. Mientras que en la figura 6 solo se modificaban las capas finales, en esta se modifican también las convolucionales, dando lugar al *full-network embedding* explicado.

2.2 Wordnet

Wordnet es una base de datos que contiene nombres, verbos, adjetivos y adverbios en conjuntos de sinónimos (que llamaremos *synsets*). Los synsets están conectados entre ellos por medio de relaciones conceptuales, semánticas y léxicas. Utilizando los *synsets* y sus relaciones, se puede generar un grafo que puede ser utilizado para distintos objetivos, como lingüística computacional y procesamiento del lenguaje natural.

En concreto utilizaremos las relaciones de:

- **Sinonimia** : dos palabras son sinónimos si tienen el mismo significado (ej, pelo y cabello, figura 9a).

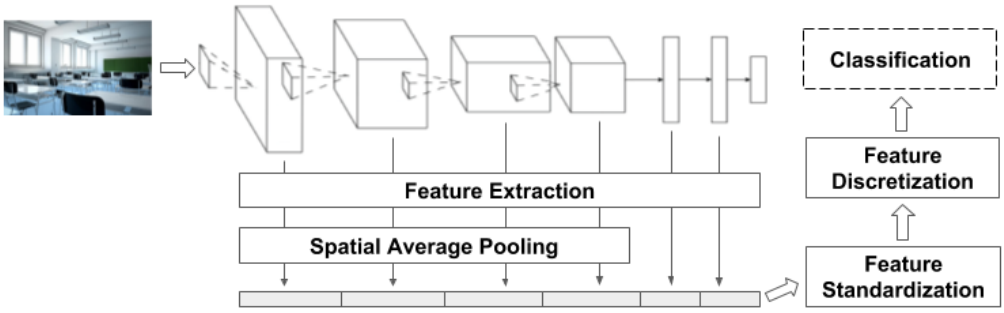
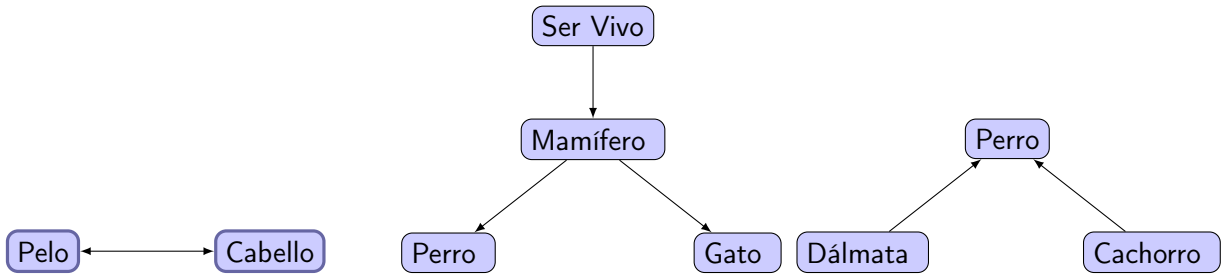


Figura 8: Estructura del *full-network embedding*

- **Hiponimia**: una palabra es hipónimo de otra si su significado es más específico que el de ésta (ej, mamífero es hipónimo de ser vivo, figura 9b).
- **Hipernimia**: una palabra es hiperónimo de otra si su significado es menos específico que el de ésta (ej, perro es hiperónimo de dálmata, figura 9c).



(a) Ejemplo de Sinónimos (b) Ejemplo de Hipónimos (c) Ejemplo de Hiperónimos

Figura 9

2.3 Imagenet

Imagenet es una base de datos de imágenes organizada utilizando la jerarquía de *wordnet*. Su principal objetivo es dotar a los investigadores en campos relacionados con visión artificial de una base de datos a gran escala con la que poder trabajar. Actualmente consta de 14,197,122 imágenes y 21,841 *synsets* indexados.

Para el full-network embedding utilizaron el subconjunto correspondiente al reto de *imagenet* de 2012 de reconocimiento de imágenes.

Éste consta de:

- Un conjunto de datos de entrenamiento de 1.2 millones de muestras y 1,000 categorías diferentes.
- Un conjunto de datos de validación de 50,000 muestras y 1,000 categorías.

Entre las posibles categorías podemos encontrar *synsets* de diferentes niveles de especificación, por ejemplo tenemos las categorías: perro, dálmata, pastor alemán...

3. Enfoque

En nuestro caso estudiaremos un *full-network embedding* obtenido a partir de la tarea origen ($\mathcal{T}_S$ =Clasificación de imágenes, $\mathcal{D}_S$ = 1.2 M imágenes de imagenet con sus correspondientes clases) y una tarea de destino ($\mathcal{T}_T$ =Clasificación de imágenes, $\mathcal{D}_T$ = 50,000 imágenes de imagenet con sus correspondientes clases). Es decir, partiendo de una red convolucional entrenada con todo el conjunto de datos de entrenamiento de imagenet, hemos generado un lenguaje de representación para el conjunto de datos de validación de imagenet. Lo que resulta en el *full-network embedding* con el que trabajaremos.

El *full-network embedding* consiste en una matriz de tamaño 50,000 muestras por 12,416 características, como observamos en la figura 10, cuyos valores están en $\{-1, 0, 1\}$. Para cada muestra también tenemos su correspondiente clasificación(un valor entre 0 y 999 que representa la clase de imagenet con la que fue clasificado).

Es importante remarcar el significado de las diferentes categorías.

- Si el valor para una imagen concreta es 1 significa que esta es representativa por presencia. Es decir, que dentro del contexto del conjunto de datos *Target*, es una *feature* característica de una clase concreta.
- Si el valor para una imagen concreta es 0 significa que esa característica no es representativa.
- Si el valor para una imagen concreta es -1 significa que es representativa por ausencia.

Hemos de tener en cuenta que estas definiciones son con respecto a $\mathcal{D}_T$, es decir, respecto al nuevo espacio de representación.

Las características están ordenadas en capas de la siguiente manera:

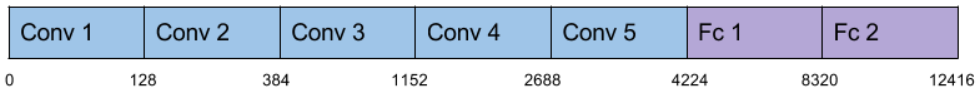


Figura 10: La disposición de las características por capas

Y finalmente, de todos los *synsets* posibles tomamos dos subconjuntos diferentes:

(a) Seres vivos (b) Objetos

Figura 11: Conjuntos de synsets que estudiaremos

Hemos tomado estos dos conjuntos por estar distribuidos de una forma bastante uniforme entre las clases, como veremos en el capítulo 3.2, y por ser de dos "ramas" lo más diferentes posibles,además están en sitios parecidos respecto a la jerarquía global, y por tanto representan niveles de abstracción parecidos.

3.1 Objetivos

Partiendo de los datos comentados y del trabajo publicado en [11] tomamos varios **objetivos** a partir de los que desarrollar el trabajo.

- Analizar el *embedding* dado y el comportamiento de las *features* en las distintas capas. Para ello se realizará un estudio estadístico inicial de la matriz de *embeddings*. De este estudio se analizarán los diferentes puntos de interés, y finalmente se hará un estudio más detallado de estos.

- Analizar si hay alguna relación entre los *embeddings* de las diferentes clases y *synsets* relacionados con ellas (sus hipónimos e hipérnimos). Para ello se realizarán las estadísticas del comportamiento de los *synsets* de la figura 11 con respecto a la matriz de *embeddings*, con intención de profundizar en los puntos de interés que se encuentren.

3.2 Estadísticas

La finalidad de este capítulo es hacer un estudio inicial de los datos de los que se dispone, es decir, de los correspondientes a los distintos *embeddings* y a los *synsets* que hemos tomado para el estudio.

3.2.1 Synsets

Como comentamos en la sección anterior los *synsets* que hemos tomado son 11. El criterio que hemos utilizado para decidir que una categoría pertenece a un *synset* es que sea hipónimo de esta, por tanto es coherente que los *synsets* más generales tengan más imágenes.

Además, ambas familias(la de seres vivos y objetos), se distribuyen de forma parecida(cada *synset* hijo tiene aproximadamente la mitad de imágenes que su *synset* padre), como podemos ver en la figura 12. Elegimos dos familias de *synsets* que cumplieran esto para que al comparar los resultados se distorsionaran lo menos posible por la cantidad de imágenes pertenecientes a ellas.

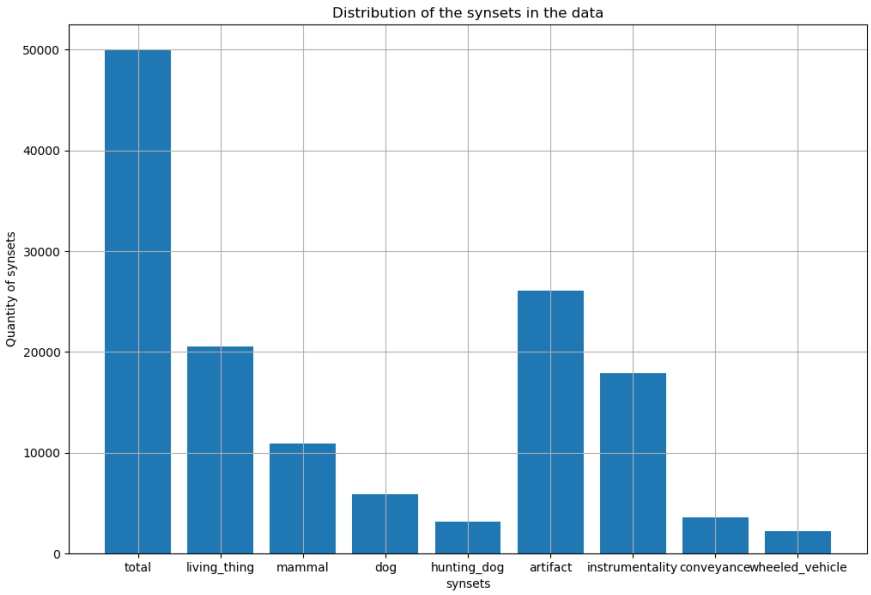


Figura 12: Distribución de los *synsets* en el embedding

3.2.2 Visión de conjunto de del Embedding e Hipótesis iniciales

Primero de todo estudiaremos las estadísticas generales del *embeddings*. Para empezar comparamos la cantidad de elementos de cada *feature*.

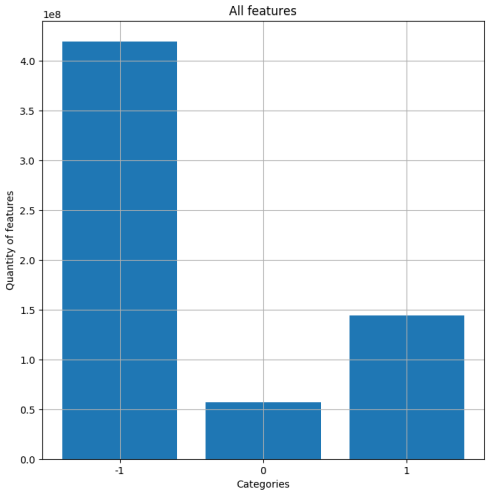
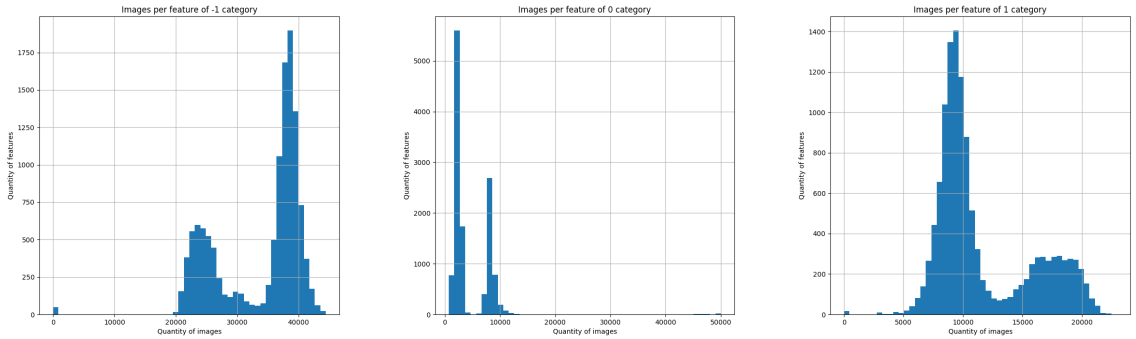


Figura 13: Cantidad de *features* de cada categoría

En la figura 13 vemos la cantidad de *features* de cada categoría. Podemos observar que la cantidad total de -1 es significativamente mayor, esta distribución se conserva(en general) para los distintos *synsets*, independientemente de lo específicos que sean. Teniendo en cuenta que al hacer *transfer learning* se suele

pasar de un conjunto de datos más general a uno más específico que haya más *features* características por ausencia es coherente. Más adelante veremos también como se distribuyen las categorías entre las diferentes capas del *embedding*.

Observamos también como se distribuyen las *features* respecto al conjunto de imágenes que tenemos.



(a) Categoría -1 (b) Categoría 0 (c) Categoría 1

Figura 14: Imágenes por categoría

En la figura 15 tenemos la distribución de las imágenes por categoría, es decir cuantas imágenes tienen una cierta cantidad de *features*. La idea de generar esta gráfica era ver si podíamos encontrar alguna distribución concreta o patrón.

Al estudiar la matriz de embeddings nos planteamos las siguientes hipótesis:

1. Las características se distribuyen de diferente manera en los layers convolucionales y los completos. Esta hipótesis proviene de la gráfica 15, donde en las tres categorías se ven dos distribuciones bien diferenciadas. Otro motivo para pensar que las características se distribuyen de forma diferente es que las capas *fully-connected* tienen más *features* que las convolucionales.
2. Cuanto más concreto es un *synset*, debería haber más features representativas, tanto por ausencia como por presencia.
3. Cuanto más profundo es el layer, debería haber más features representativas, tanto por ausencia como por presencia. Esta hipótesis viene de que en una red convolucional cuanto más profunda es una capa más concretas suelen ser las features con las que se activa, como vimos en la figura 5.
4. Se puede ver una relación entre los embeddings de synsets hipónimos. La idea sería que dada una imagen perteneciente a un *synset*, compartiría *features* características con sus hipónimos.

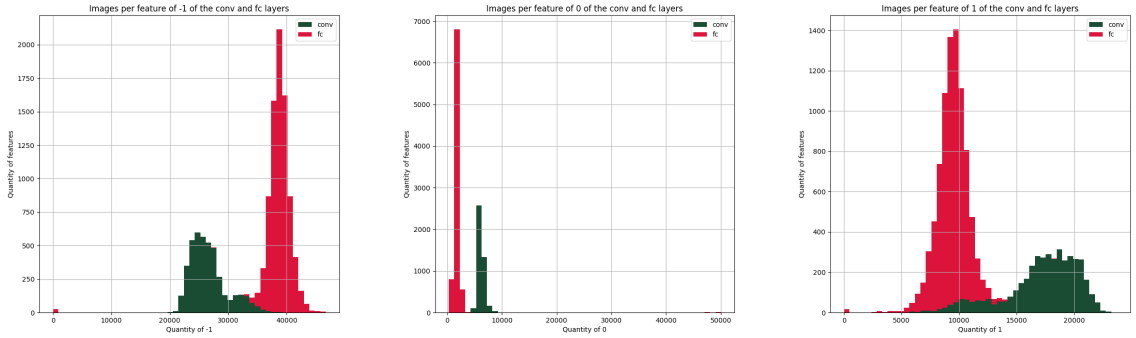
4. Análisis

4.1 De wordnet a full network embedding

En este apartado explicaré el estudio hecho para intentar aceptar o revocar las hipótesis comentadas. Empezaré con la distribución por tipo de capa en el *embedding* general, para luego concretar estudiando el comportamiento por capa. Finalmente explicaré los descubrimientos relacionados con los *synsets*.

4.1.1 Distribución por tipo de capa

Para comprobar si la hipótesis 1 es cierta calculamos las gráficas de la figura 15 distinguiendo entre las capas convlolucionales y *fully-connected*, obteniendo las siguientes gráficas:



(a) Categoría -1 (b) Categoría 0 (c) Categoría 1

Figura 15: Imágenes por *feature*

En ellas podemos ver claramente la distinción que suponíamos. Sobretodo para la categoría 0, hay una clara separación entre el comportamiento de los diferentes tipos de capa. En parte esto se debe a que hay más *features* en las capas *fully-connected*, por eso sus gráficas correspondientes son más altas, además en éstas podemos ver una menor variabilidad, que concuerda con el hecho de que solamente son 2 capas *fully-connected*, contra las 5 convolucionales. También vemos que el grueso de las features características (tanto por presencia como por ausencia) se encuentran en las capas *fully-connected*, cosa que concuerda con el hecho de que sean estas capas las que dan la clasificación final.

4.1.2 Comportamiento respecto a la profundidad

Una vez comprobado que los dos tipos de capa se comportan diferente, vamos a estudiar en que nivel se diferencian y a que profundidad. De esta forma veremos si podemos aceptar o desmentir la hipótesis 3.

Para ello hemos calculado la cantidad de *features* de cada tipo por capa respecto al *embedding* total. Como podemos observar en la gráfica 16. De esta forma se esperaba ver como se distribuyen las *features* por cada capa.

En la gráfica esperábamos encontrar una mayor proporción de *features* características por presencia en las capas finales, sin embargo podemos ver que la mayor proporción se encuentra en la cuarta capa

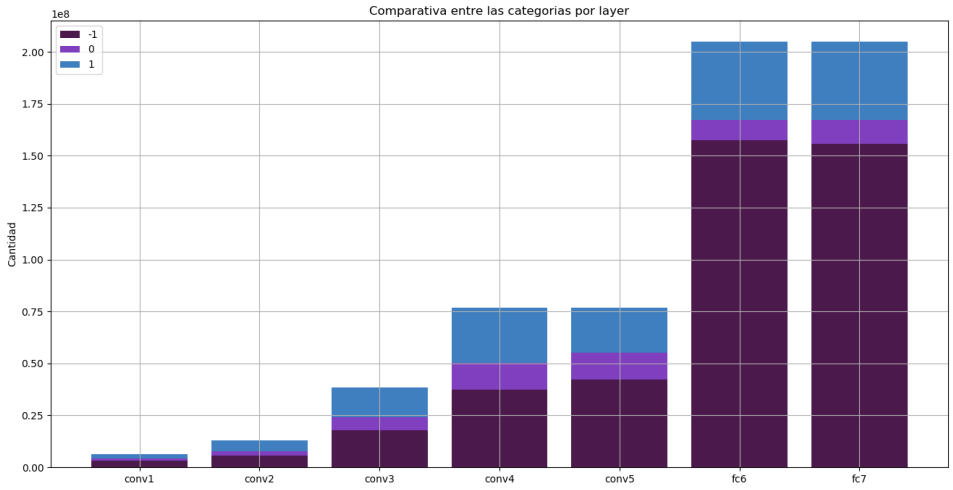


Figura 16: Cantidad de *features* de cada categoría por capa

convolucional. En esta gráfica podemos ver claramente que todas las capas, no solo las *fully-connected* contienen información importante de cara a caracterizar el espacio de representación, al contrario de lo que sostienen en [9].

4.1.3 Comportamiento de los synsets

A la hora de estudiar los *synsets* nos hemos topado con la dificultad de decidir el valor de una *feature* para el conjunto de todas las muestras que pertenecen al *synset*, puesto a que para diferentes muestras puede tener distintos valores, como podemos ver en el ejemplo de la figura 17.

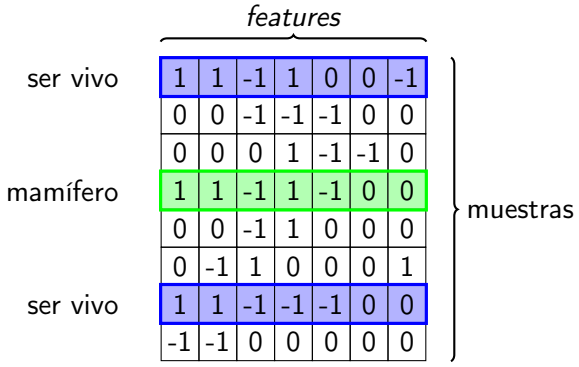


Figura 17: Muestra de una sección del embedding.

Para solucionar este problema hemos decidido tomar dos enfoques diferentes, por un lado hemos tomado la sub-matriz correspondiente a cada *synset*, por otro lado hemos tomado un vector representante para cada *synset* a estudiar.

Sub-matriz:

En este caso hemos tomado una matriz cuyas filas consisten en las muestras que pertenecen al *synset* que queremos estudiar y cuyas columnas corresponden a las *features*, como muestra el ejemplo de la figura 18.

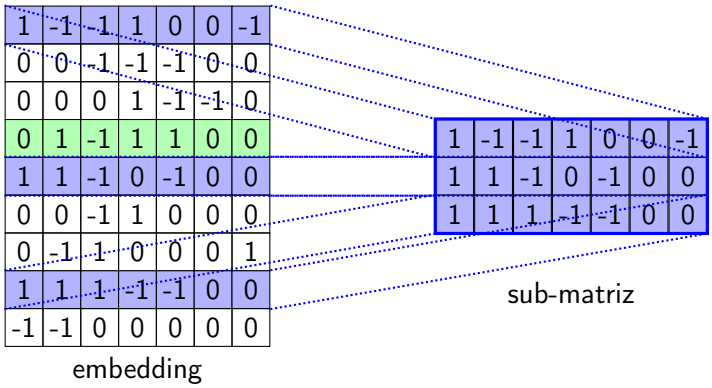
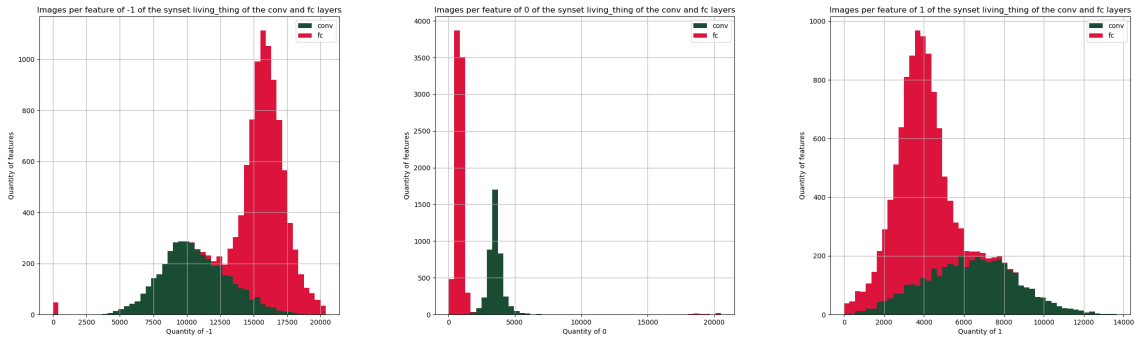


Figura 18: Ejemplo de una submatriz de un *synset*.

Este enfoque nos da información general sobre el comportamiento de los distintos *synsets* en el *embedding*.

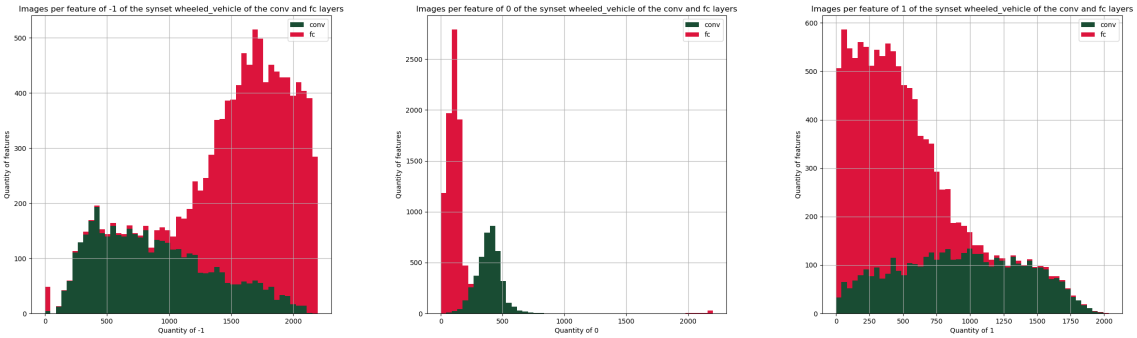
Utilizando la sub-matriz asociada a cada uno de los *synsets* hemos generado gráficas equivalentes a la de la figura 15. En este caso, las distribuciones se mantienen bastante en los *synsets* más generales, como podemos ver en la figura 19, mientras que en los menos generales se empiezan a distorsionar. Como podemos observar en la figura 20.



(a) Categoría -1 (b) Categoría 0 (c) Categoría 1

Figura 19: Imágenes por categoría del *synset* seres vivos

La diferencia entre los distintos grupos de gráficas (*synsets* más generales contra más específicos) se debe a que al ser una muestra más pequeña hay una mayor varianza. También se observa una disminución de la proporción de *features* no característica cuanto más concreto es el synset, esto nos da una idea de que la red convolucional se adapta al nivel de generalización que tiene un *synset*, por ejemplo, *perro* tendrá menos *features* no características (en proporción) que *mamífero*. Este hecho concuerda con nuestra hipótesis 2, en



(a) Categoría -1 (b) Categoría 0 (c) Categoría 1

Figura 20: Imágenes por categoría del *synset* vehículo

la que ahondaremos más adelante.

También se observan unos comportamientos simétricos entre las *features* características por presencia y por ausencia en todos los *synsets* y en la gráfica general. Este tipo de simetría es una buena señal, puesto a que la caracterización que hemos dado es abstracta, es decir, la definición de característica por ausencia o por presencia no viene dada por la red convolucional sino por un estudio externo a ésta.

Otro dato que podemos obtener es la cantidad de *features* de cada una de las categorías por *synset*, como vemos en la figura 21, simplemente sumando los diferentes valores de las sub-matrices.

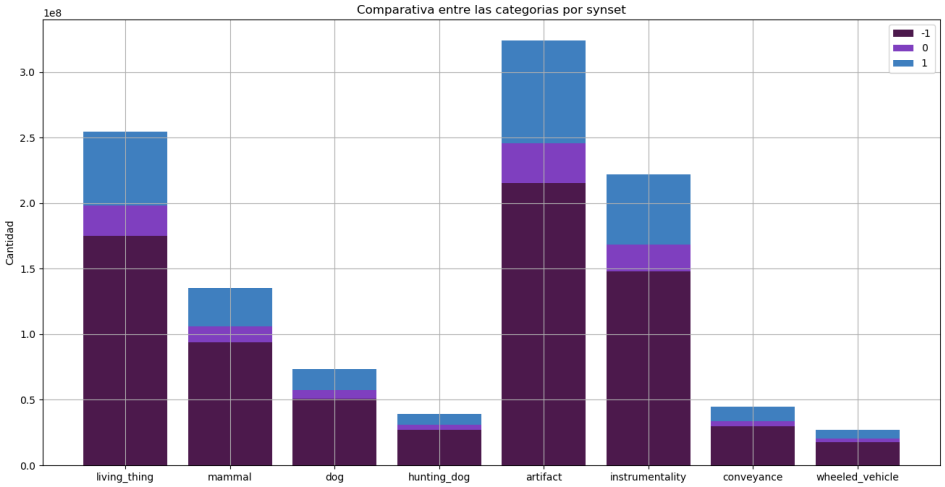


Figura 21: Cantidad de features por synset

Además utilizando esta sub-matriz hemos calculado la cantidad de *features* de cada tipo por capa, de la misma forma que calculamos los valores de la gráfica 16. En general no se ve mucha diferencia entre las gráficas de cada *synset* y la general(como podemos observar en la figura 22), esto implica que la cantidad de *features* con los distintos valores por capa es parecida independientemente del synset que tomemos y de

lo general que éste sea, cosa que va en contra de nuestra hipótesis 2.

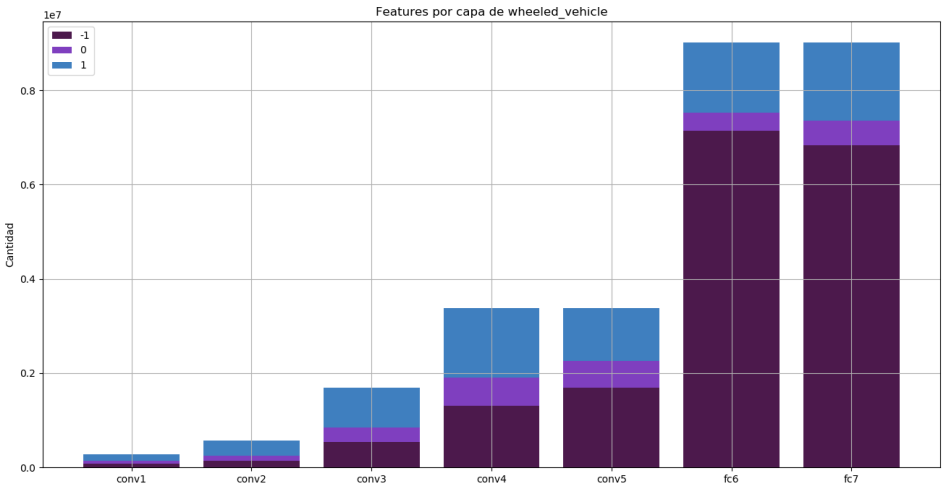


Figura 22: Cantidad de features por capa de vehículo

Mientras que la distribución de las imágenes por *feature* se difuminaba con *synsets* más concretos (figura 20), vemos que la proporción de *features* características se conserva, tanto por synset como por capa. Esto nos confirma que la difuminación que veíamos sea debida a una menor cantidad de muestras a la hora de calcular la gráfica y no a una distinción por *synset*.

Representante:

Con la sub-matriz hemos logrado una visión general del comportamiento de los *synsets* dentro del *embedding*, pero hay información que estamos perdiendo. Sería interesante saber si a nivel de *feature* diferentes *synsets* se comportan distinto, o de la misma forma, como parecían indicar los descubrimientos para sub-matrices. Por ejemplo que se concentrasen todos los 1 en un mismo conjunto de *features* para el mismo synset.

Para solventar este problema decidimos tomar un vector muestra para cada uno de los *synsets* de tamaño $1x12,416$ donde cada componente se toma como la categoría más frecuente para esa *feature* concreta, como muestra la figura 23.

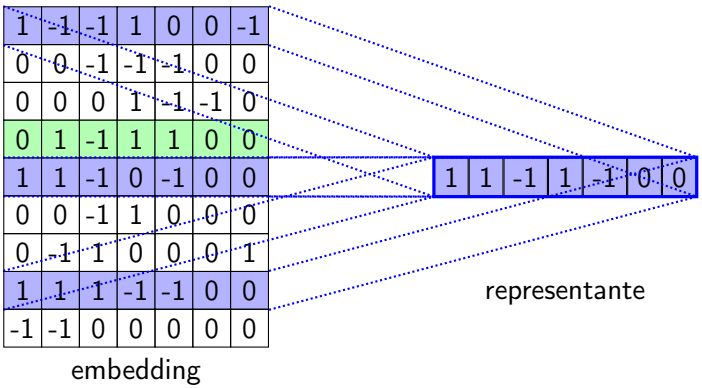


Figura 23: Ejemplo de un representante de *synset*.

De esta obtenemos una información más concreta respecto a cada *feature*, además podemos comparar los distintos *synsets* de una forma más efectiva. Por ejemplo, si una *feature* suele ser característica para mamíferos, quedará marcada en el representante, mientras que si tomáramos los valores de la proporción de las características respecto a la sub-matriz correspondiente al *synset* esta información se perdería, ya que la proporción de *features* no características es muy superior a la del resto de opciones.

Al calcular las gráficas anteriores, pero del representante, esta vez si que obtenemos diferencias.

Primero de todo observamos que en la gráfica 24, cuanto más concreto es el *synset* mayor proporción de features características por presencia tenemos, además que las *features* no características desaparecen prácticamente por completo. Esto implica que las *features* no características no sólo están en minoría para la mayoría de imágenes y *features*,sino que además no se concentran en conjuntos concretos de *features*.

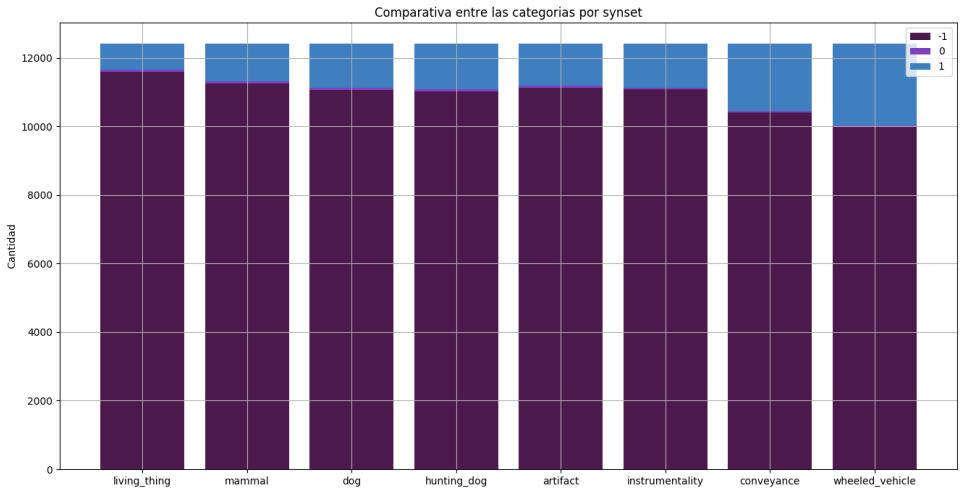


Figura 24: Cantidad de features de los representantes

Si además comparamos la gráfica 24 con la de 21 vemos que las features características aparecen en mayor proporción. Si no estuvieran concentradas, las *features* tomarían todas el valor -1, puesto a que es el

más común en el embedding.

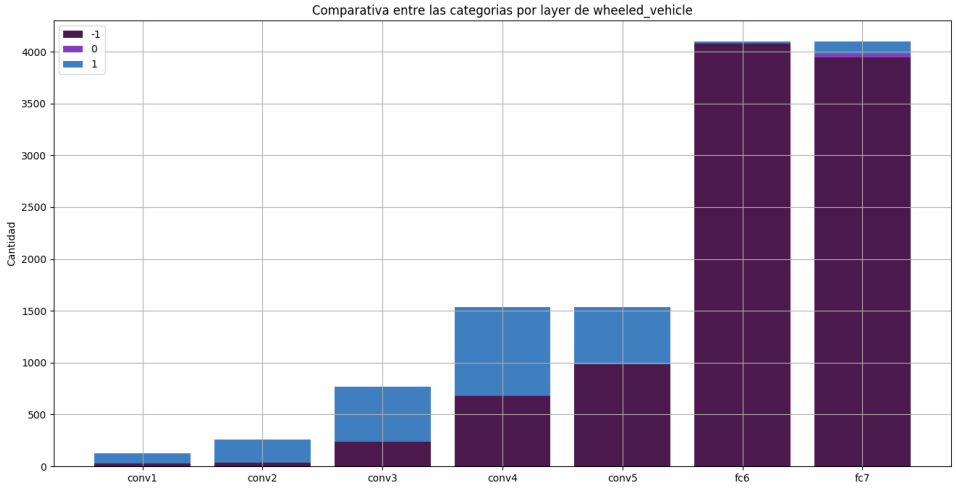


Figura 25: Cantidad de features de los representantes

Respecto a las proporciones por capa, para todos los synsets han salido resultados parecidos, siendo el más visible el correspondiente a Vehículo, como vemos en la 25. En este caso también vemos una desaparición de las *features* no representativas en todas las capas. Además la proporción de *features* características aumenta en las capas convolucionales, mientras que disminuye en las *fully-connected*. Lo que implica una dispersión de los valores característicos con respecto a los que no lo son.

Para comparar los *synsets* a un nivel todavía más concreto hemos generado diferentes matrices de cambio a partir de los representantes, éstas están definidas como vemos en la figura 26.

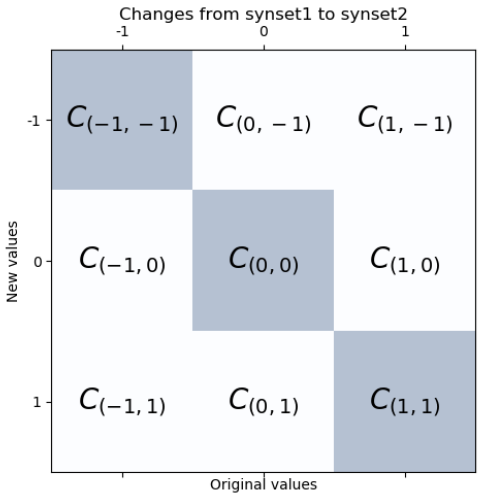


Figura 26: Matriz de cambios

Donde $C_{(i,j)}$ es la cantidad de valores que en el primer *synset* valían i y en el segundo valen j, comparando

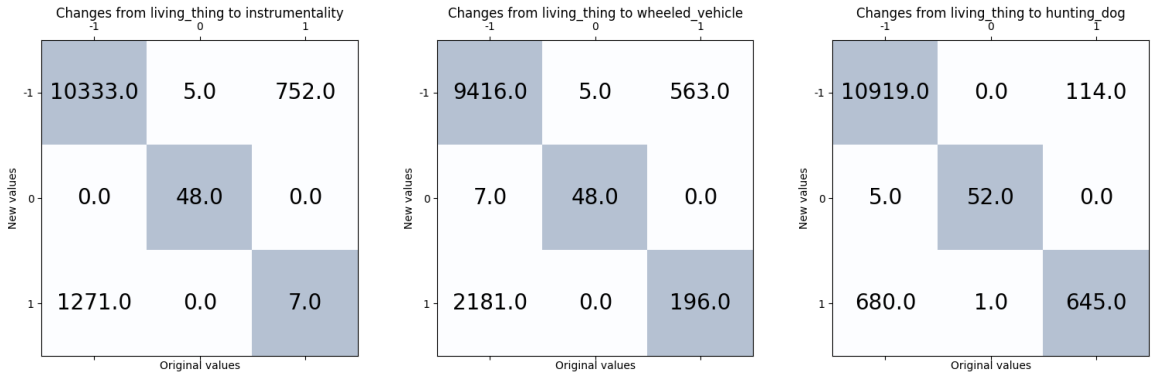
elemento a elemento. La suma total de todas las $C_{(i,j)}$ es 12,416, es decir, la cantidad total de *features* de un representante. Además, si sumamos todas las $C_{(i,j)}$ de una columna concreta obtenemos el total de *features* con valor *i* del primer *synset*, y de forma simétrica, si sumamos las filas obtenemos el total de *features* con valor *j* del segundo *synset*.

En la diagonal contienen la cantidad de valores que se conservan, es decir, que para el primer *synset* y el segundo tienen el mismo valor. En el resto de posiciones contiene la cantidad de *features* que en el primer *synset* tenían un valor y en el segundo pasa a tener otro.

En nuestro primer ejemplo de matrices de cambio (figura 27) podemos observar que, tanto la columna como la fila correspondiente al cero (las *features* no características) tiene unos valores muy bajos con respecto al resto. Esto se repite en todas las matrices para todos los *synsets*, puesto a que el total de *features* con este valor es muy bajo. Respecto al resto de valores, los explicaré por columnas para facilitar la comprensión.

En la columna de las *features* características por ausencia se acumulan la mayoría de los valores, como era de esperar. En la figura 27a, vemos una respetable cantidad de *features* que eran representativas por ausencia para seres vivos pasan a ser representativas por presencia en *instrumentos*, además, este fenómeno se acentua en *synsets* más concretos, como vemos en la figura 27b. Un ejemplo de este comportamiento podrían ser *features* que detectaran ruedas, a la hora de clasificar, sería representativo para seres vivos que no tuvieran ruedas, mientras que el *synset* de los instrumentos, contiene los vehículos, para los que esta *feature* sería característica por presencia. Para Perro de caza, sin embargo, la cantidad de *features* de valor -1 que pasan a valer 1, es significativamente menor. Las que hay podrían ser *features* bastante concretas, que se asociaran a perros de caza, pero no al conjunto general de seres vivos, como por ejemplo ser cierta raza de perro.

Finalmente en la columna de las *features* características por presencia, vemos en la figura en al que se conservan más es 27c, mientras que en el resto casi todas pasan a ser características por ausencia. De forma simétrica al comportamiento de la columna de las *features* características por ausencia, pero a menor escala.

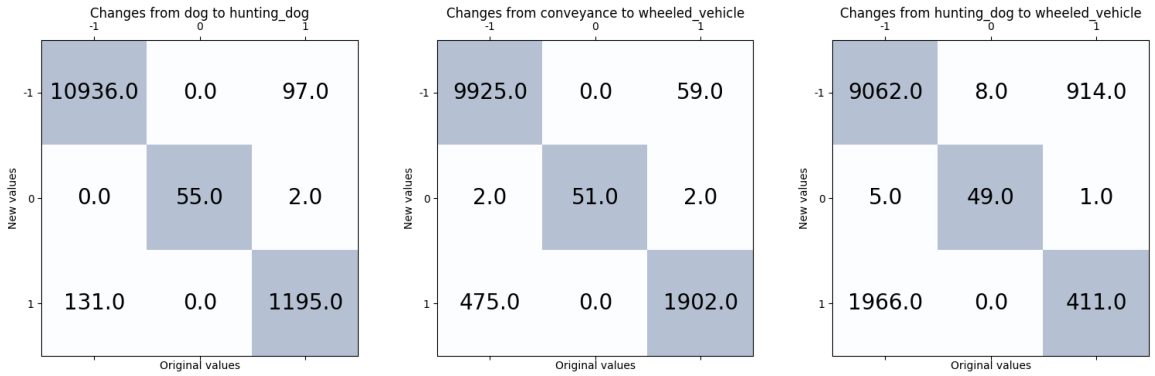


(a) Ser Vivo a Instrumento (b) Ser Vivo a Vehículo (c) Ser Vivo a Perro de Caza

Figura 27: Matrices de cambio de Ser Vivo

El siguiente ejemplo (28) es para ver el comportamiento en *synsets* más concretos. Podemos ver claramente que la red está captando la similitud entre Perro y Perro de caza, puesto a que casi todas las *features* se mantienen en la diagonal. Se comporta de forma parecida con Transporte y Vehículo. Mientras que entre Perro de Caza y Vehículo la mayoría de las *features* pasan de características a no características y viceversa.

Nótese la similitud de la figura 28c y la figura 27a, aun siendo en el primer caso los synsets más concretos y en el segundo los más generales, el comportamiento es parecido.



(a) Perro a Perro de caza (b) Transporte a Vehículo (c) Perro de caza a Vehículo

Figura 28: Matrices de cambio

4.2 Del full network embedding a wordnet

Explicación de la distancia definida entre los synsets, demostración de que es distancia, los grafos con las distancias.

Referencias

[1] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, New York :, 2006.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[4] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[5] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems 27*, pages 3320–3328, 2014.

[6] Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. Factors of transferability for a generic convnet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(9):1790–1802, 2016.

[7] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.

[8] Dario Garcia-Gasulla, Ferran Parés, Armand Vilalta, Jonathan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, and Toyotaro Suzumura. On the behavior of convolutional nets for feature extraction. *CoRR*, abs/1703.01127, 2017.

[9] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *CoRR*, abs/1310.1531, 2013.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] Dario Garcia-Gasulla, Armand Vilalta, Ferran Parés, Jonathan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, and Toyotaro Suzumura. An out-of-the-box full-network embedding for convolutional neural networks. *CoRR*, abs/1705.07706, 2017.