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 (name of the supervisor/s of the master's thesis)

Month, year

Thanks to...

Abstract

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

Keywords

keyword1, keyword2, keyword3, ...

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 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 na salida.

Cada entrada (**x**) tiene asociado un peso (**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 (**f**) a la suma ponderada de las entradas como muestra la figura:

Además de los elementos ya comentados tendremos un término de sesgo (**b**), cuyo objetivo es proveer cada neurona con un valor entrenable que no dependa de la entrada.

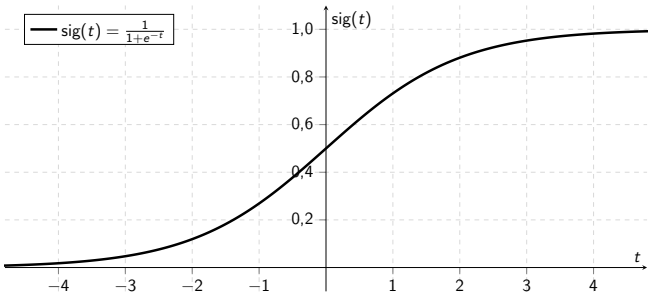
La salida de la neurona (**Y**) se calculara de la siguiente manera:

$$Y = f\left(\sum_i \omega_i x_i + b\right)$$

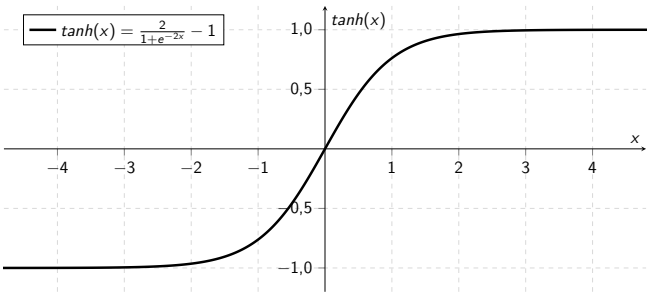
Llamaremos a **f** la **función de activación**, es una función (no lineal) diferenciable cuyo propósito es introducir no linealidad a la salida de la neurona. Ésto permite adaptar el modelo a problemas reales, puesto a que estos raramente son lineales.

Las funciones de activación más utilizadas son:

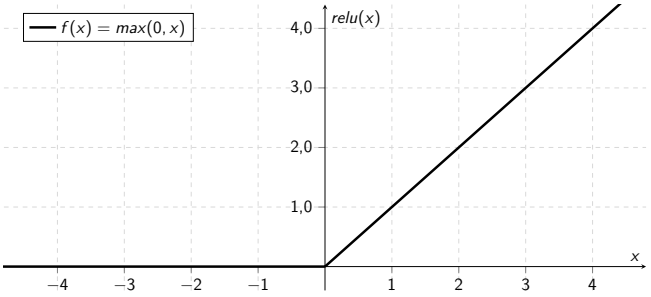
- Sigmoide: $f(x) = \frac{1}{1+e^{-x}}$



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



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



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.

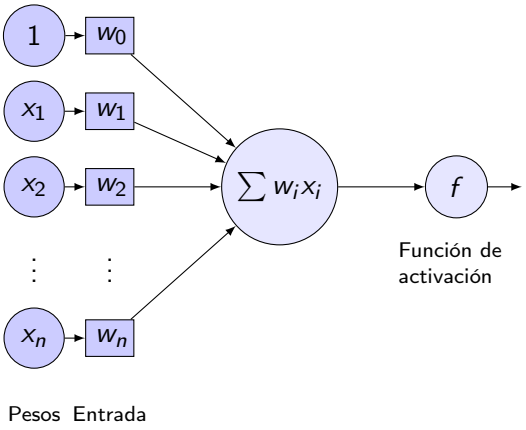


Figura 1: Ejemplo de una neurona

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

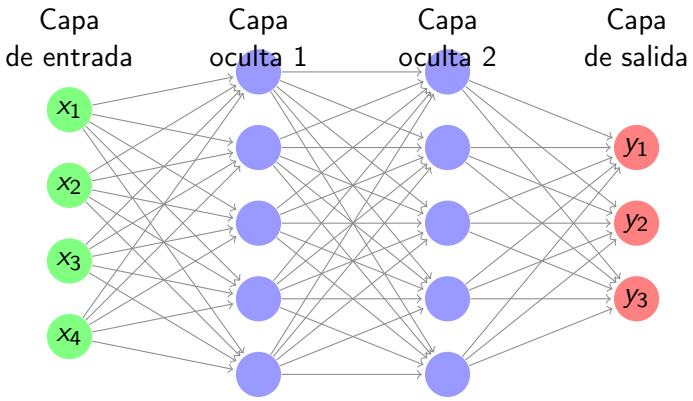


Figura 2: Ejemplo de red neuronal

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.

Por tanto nos queda una estructura formada por pesos ordenados en distintos tipos de capas. Cada peso de la red es un parámetros 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.

1.1 Entrenar la red: Backpropagation

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

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, la predicción dada por la red se compara con la salida esperada y se calcula un error utilizando la función de pérdida (por ejemplo la de mínimos cuadrados). La pérdida calculada se utiliza para actualizar los pesos de la última capa, buscando los valores de éstos que la minimizan.

TODO: Añadir un ejemplo, explicar bien backprop

La complejidad reside en optimizar los pesos de las capas que no están conectadas directamente con la salida. Para resolver esto backpropagation utiliza la regla de la cadena, que permite calcular derivadas de capas previas, y de esta forma actualizar los pesos de las capas restantes.

Tradicionalmente, el cambio de pesos se ha computado usando el algoritmo de optimización llamado *Stochastic Gradient Descent*, un método iterativo de minimización.

De esta forma, aplicando backpropagation iterativamente podemos estimar los pesos de todas las capas de una red neuronal para un problema dado.

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[4].

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:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{1}$$

Equación 1. Convolución

Definición 1.4. Una **función de kernel**

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 completa 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.

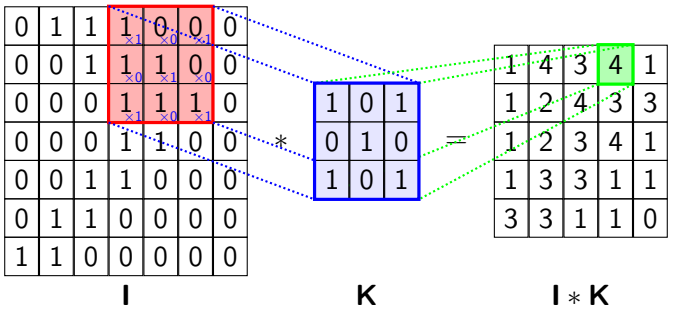
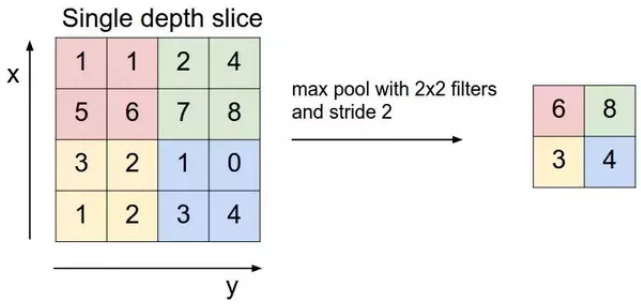


Figura 3: Ejemplo de la forma de operar una convolución sobre una imagen.

En la figura 3 podemos ver un ejemplo del funcionamiento de una convolución con un kernel de tamaño 9 sobre una imagen, en el ejemplo podemos observar como se comparten los pesos a la hora de calcular el resultado.

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 completa, 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.



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 4, 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*.

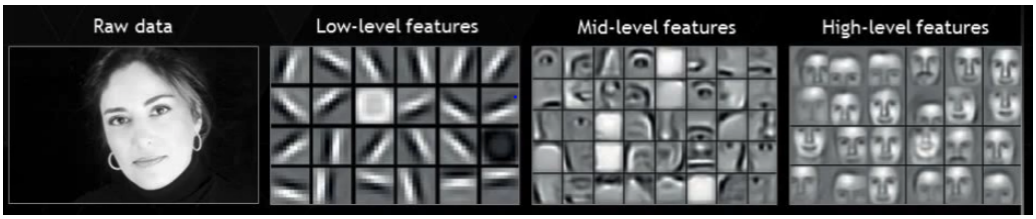


Figura 4: Ejemplo de features

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.
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.

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:

- 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** La primera estrategia consiste en inicializar los datos desde un estado no aleatorio (tomando los pesos ya entrenados). 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** 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, 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 *transfer learning for features extraction* [6].

2. Trabajo Relacionado

2.1 Full-Network embedding

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 5. El resto de capas se descartan por "ser poco probable que contengan una representación mejor", 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*.

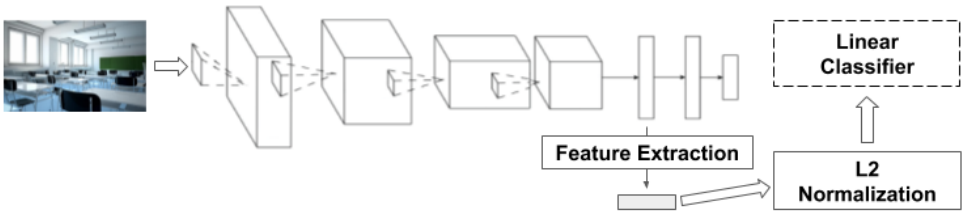


Figura 5: Estructura básica que se suele utilizar en *feature extraction*

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 completas).
- 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 completas 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 hace falta una estandarización. El valor estandarizado de cada categoría se obtiene calculando la media y la desviación típica del conjunto de datos de entrenamiento. De esta forma se pueden integrar los distintos valores en el *embedding*.
- Finalmente se aplica una **discretización de características** para evitar problemas al intentar explorar un espacio de dimensión tan grande. 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

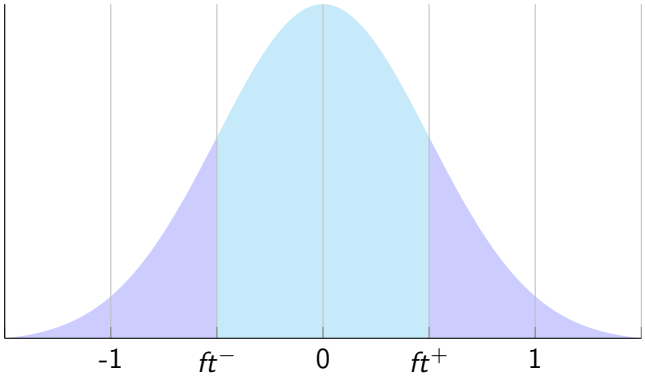


Figura 6

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 6. Para encontrar los límites se basan en [6], 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 [6] separan las características en tres conjuntos: *característico por presencia*, *no característico* y *característico por ausencia*.

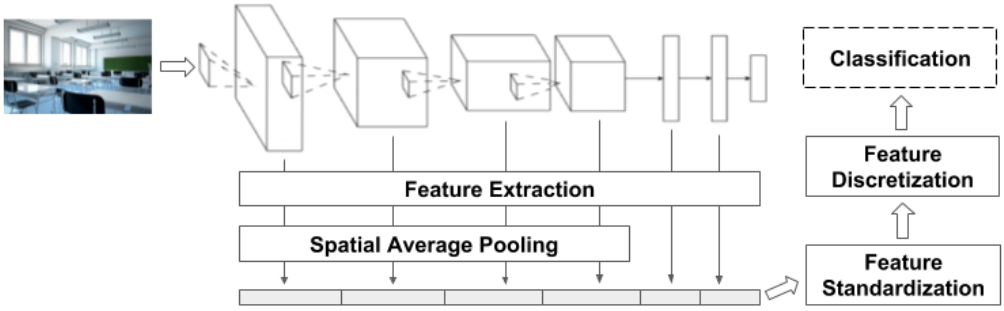


Figura 7: Estructura del *full-network embedding*

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, gato y minino).
- **Hiponimia**: una palabra es hipónimo de otra si su significado es más específico que el de ésta (ej, silla es hipónimo de mueble).
- **Hipernimia**: una palabra es hiperónimo de otra si su significado es menos específico que el de ésta (ej, perro es hiperónimo de dálmata).

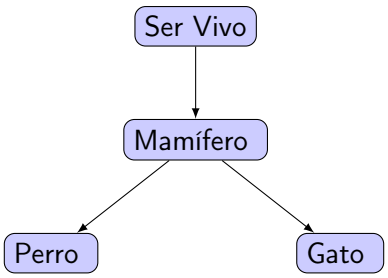


Figura 8: Ejemplo de hipónimos

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 21841 *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 1000 categorías diferentes.
- Un conjunto de datos de validación de 50000 muestras y 1000 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, $\mathcal{D}_T$ = 50000 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 50000 muestras por 12416 características, 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.

Por ejemplo 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.

Las características están ordenadas en capas de la siguiente manera:

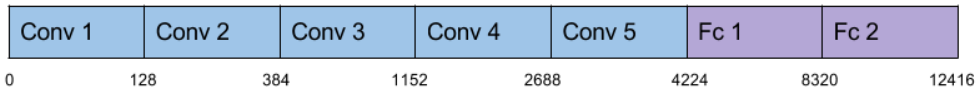


Figura 9: La disposición de las características por capas

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

(a) Seres vivos (b) Objetos

Figura 10: 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, para poder comparar el comportamiento del embedding en ambos casos.

3.1 Objetivos

Partiendo de los datos comentados y del trabajo publicado en [3] tomamos varios **objetivos** a partir de los que desarrollar el trabajo.

- Analizar el *embedding* dado y elcomportamiento de las *features* en las distintas capas.
- Analizar si hay alguna relación entre los embeddings de las diferentes clases y synsets relacionados con ellas (sus hipónimos e hipérnimos).

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 Embedding

Primero de todo estudiaremos las estadísticas generales del *embeddings*.

Para empezar comparamos la cantidad de elementos de cada *feature* en los distintos *embeddings*.

En la figura 11 podemos observar que la cantidad total de -1 es significativamente mayor. 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.

En la figura 12 tenemos la distribución de las imágenes por categoría, es decir cuantas imágenes tienen una cierta cantidad de *features*. En las tres categorías se ven dos distribuciones bien diferenciadas, lo que

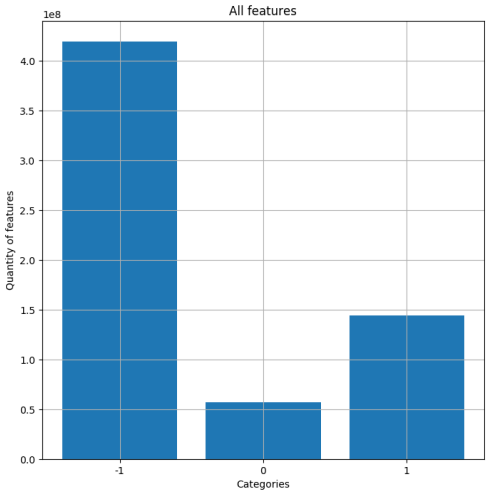
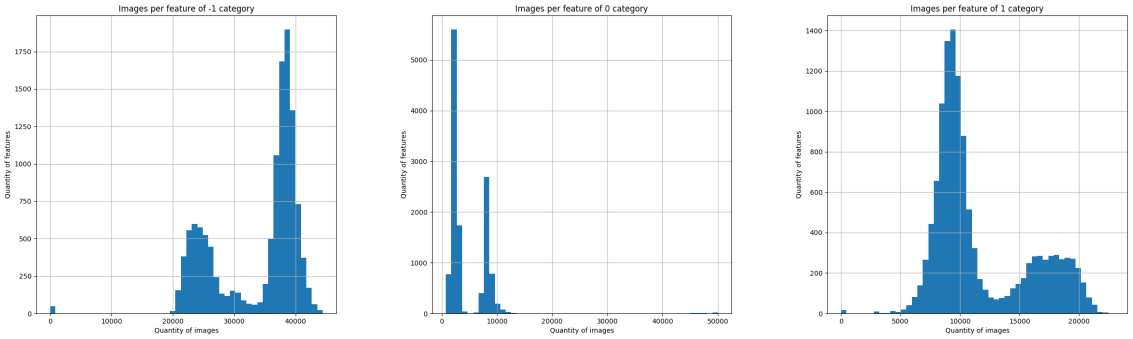


Figura 11



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

Figura 12: Imágenes por categoría

hace plantearse si depende de los dos tipos de capas diferentes del *embedding* (las convolucionales y las completas).

3.2 Synsets

Como comentamos en la sección anterior los *synsets* que hemos tomado son 10. 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 13. 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.

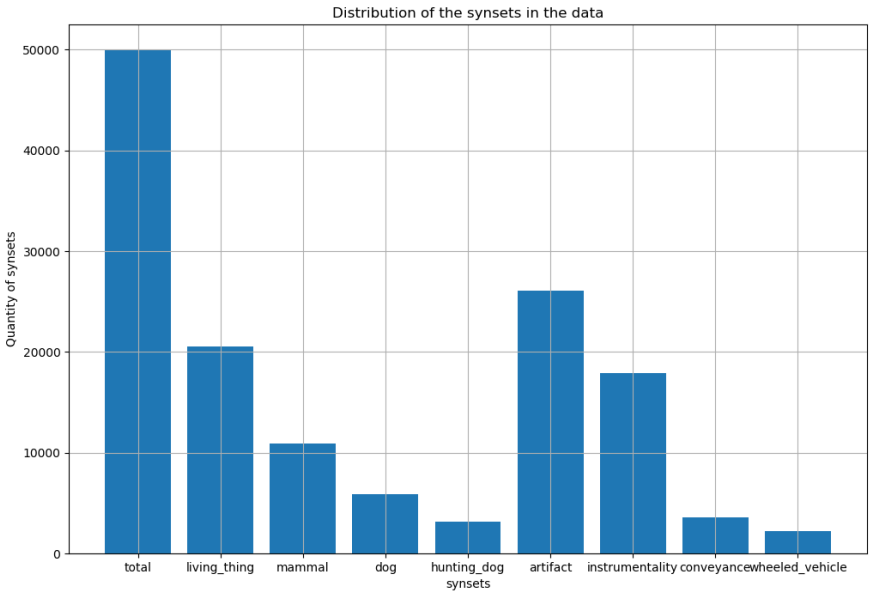


Figura 13

3.3 Hipótesis iniciales

Por la naturaleza del problema y lo que hemos visto al estudiar la matriz de embedding aparecen las siguientes hipótesis:

- Las características se distribuyen de diferente manera en los layers convolucionales y los completos.
- Cuanto más concreto es un synset, debería haber más features representativas, tanto por ausencia como por presencia.
- Cuanto más profundo es el layer, debería haber más features representativas, tanto por ausencia como por presencia.
- Se puede ver una relación entre los embeddings de synsets hipónimos.

4. Análisis

4.1 De wordnet a full network embedding

En este apartado explicaré:

- Las distribuciones de las imágenes por feature son diferentes para conv y fc, y mantienen la forma entre embeddings.
- Las distribuciones de las imágenes por feature por layer por synset se conservan.
- Explicar los resultados de las matrices de cambio.
- Cuanto más concreto es el synset mayor proprción de 1.

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.

5. Estructura del TFG:

- **Introducción (5 pags):**
  - FNN (Explicar neural networks, me puedo inspirar en `https://upc-mai-dl.github.io/mlp-convnets-the`
  - CNN
  - EMbeddings (Transfer Learning)
- **Related Work (10 pags):**
  - FNE
  - Wordnet
  - imagenet
- **Approach**
  - Hipotesis iniciales
  - statistics
  - insights
- **Analysis**
  - De wordnet a fne: Dado el embedding hemos encontrado patrones con los distintos synsets que concuerdan con las hipótesis
  - De fne a wordnet: Dado el árbol sintáctico y los patrones anteriores podemos generar una distancia con la que representamos lo parecidos que son los siynsets para la imagen.

Referencias

[1] https://upc-mai-dl.github.io/mlp-convnets-theory/

[2] http://cs231n.stanford.edu/.

[3] Dario Garcia-Gasulla, Armand Vilalta, Ferran Parés, Jonatan Moreno, Eduard Ayguadé, Jesus Labarta, Ulises Cortés, Toyotaro Suzumura, *An Out-of-the-box Full-network Embedding for Convolutional Neural Networks.*En arXiv:1705.07706,2017, https://arxiv.org/abs/1705.07706.

[4] https://en.wikipedia.org/wiki/Convolutional_neural_network#Convolutional

[5] Deep Learning Ian Goodfellow Yoshua Bengio Aaron Courville

[6] Dario Garcia-Gasulla, Ferran Parés, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, and Toyotaro Suzumura. On the behavior of convolutional nets for feature extraction. arXiv preprint arXiv:1703.01127, 2017.

[7] https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/neural_networks.html

A. Glosario

- Fordward pass:
- Sobreajuste:
- Kernel
- Deep Network:
- Hiperparámetros
- Valores de activación de una red neuronal

B. Código utilizado

```
class Data:
    """
    Esta clase consiste en los datos que voy a necesitar para hacer las estadísticas.
    Que no dependen de los synsets elegidos.

    Attributes:
        version (int): versión del embedding que utilizo puede ser 19, 25 o 31
        embedding_path (str): path
        layers (dict): Un diccionario tal que
            layers[string correspondiente al layer] = [inicio del layer, final del layer]

         labels ()

         :parameter version = Version del embedding que utilizo
    """

    def __init__(self, path, version=25):
        """

        :param version: Es la versión del embedding que queremos cargar (25,31,19)
        """
        self.version = version
        _embedding_path = "../Data/vgg16_ImageNet_ALLlayers_C1avg_imagenet_train.npz"
        self.imagenet_id_path = "../Data/synset.txt"
        if version == 25:
            _embedding = 'vgg16_ImageNet_imagenet_C1avg_E_FN_KSBsp0.15n0.25_Gall_train_.npy'
        elif version == 19:
            _embedding = 'vgg16_ImageNet_imagenet_C1avg_E_FN_KSBsp0.11n0.19_Gall_train_.npy'
        elif version == 31:
            _embedding = 'vgg16_ImageNet_imagenet_C1avg_E_FN_KSBsp0.19n0.31_Gall_train_.npy'
```

```
else:
    _embedding = path
    print('No has puesto un embedding válido, usando el de defoult (25)')
self.discretized_embedding_path = '../Data/Embeddings/' + _embedding
print('Estamos usando ' + _embedding[-20:-16])
embedding = np.load(_embedding_path)
self.labels = embedding['labels']
# self.matrix = self.embedding['data_matrix']
del embedding
self.dmatrix = np.array(np.load(self.discretized_embedding_path))
self.imagenet_all_ids = np.genfromtxt(self.imagenet_id_path, dtype=np.str)
self.features_category = [-1, 0, 1]
self.colors = ['#3643D2', 'c', '#722672', '#BF3FBF']
self.layers = {
    'conv1_1': [0, 64],  # 1
    'conv1_2': [64, 128],  # 2
    'conv2_1': [128, 256],  # 3
    'conv2_2': [256, 384],  # 4
    'conv3_1': [384, 640],  # 5
    'conv3_2': [640, 896],  # 6
    'conv3_3': [896, 1152],  # 7
    'conv4_1': [1152, 1664],  # 8
    'conv4_2': [1664, 2176],  # 9
    'conv4_3': [2176, 2688],  # 10
    'conv5_1': [2688, 3200],  # 11
    'conv5_2': [3200, 3712],  # 12
    'conv5_3': [3712, 4224],  # 13
    'fc6': [4224, 8320],  # 14
    'fc7': [8320, 12416],  # 15
    'conv1': [0, 128],  # 16
    'conv2': [128, 384],  # 17
    'conv3': [384, 1152],  # 18
    'conv4': [1152, 2688],  # 19
    'conv5': [2688, 4224],  # 20
    'conv': [0, 4224],  # 21
    'fc6tofc7': [4224, 12416],  # 23
    # 'all':[0,12416]          # 24
}
self.reduced_layers = {
    'conv1': [0, 128],
    'conv2': [128, 384],
    'conv3': [384, 1152],
    'conv4': [1152, 2688],
    'conv5': [2688, 4224],
    'fc6': [4224, 8320],
    'fc7': [8320, 12416]
```

```
    }

def __del__(self):
    self.embedding = None
    self.dmatrix = None
    self.version = None
    self.embedding_path = None
    self.layers = None
    self.labels = None
    self.features_category = None
    self.colors = None
    gc.collect()
```