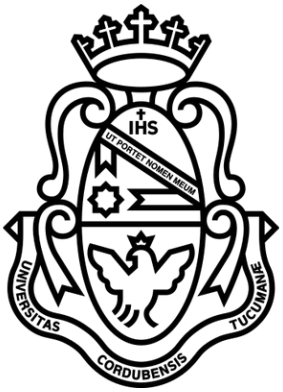
Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía, Física y Computación



Trabajo Especial de Licenciatura en Ciencias de la Computación

Entrenamiento de modelos de aprendizaje profundo mediante autosupervisión

Rubén Ezequiel Torti López

Director: Jorge Adrián Sánchez

Agosto 2017

Resumen

Dentro del campo del aprendizaje automático, una clase de técnicas conocidas como Deep Learning (DL) han cobrado particular relevancia, ya que mediante su utilización se han conseguido mejoras muy significativas respecto de métodos tradicionales. Una desventaja de los modelos basados en DL es que usualmente cuentan con más parámetros que cantidad de elementos en los conjuntos de datos de entrenamiento. En el caso particular de la clasificación de imágenes por contenido, si bien existen grandes conjuntos de datos anotados disponibles, su generación para problemas en otros dominios es muy costosa. Se propone en este trabajo una manera alternativa al entrenamiento de esta clase de modelos inspirada en cómo los organismos vivientes desarrollan habilidades de percepción visual: moviéndose e interactuando con el mundo que los rodea. Partiendo de la hipótesis de que se puede usar la información del movimiento propio (rotación y traslación en los ejes X,Y,Z) como método de supervisión, Agrawal et al. [1] ya han demostrado que es posible obtener buenos resultados entrenando con menos imágenes anotadas que lo usual. Se validan experimentalmente los resultados de este método de entrenamiento con respecto a los del estado del arte en tareas de clasificación en distintos dominios.

Índice

Capítulo 1

1. Introducción

Para los seres humanos, percibir el mundo que nos rodea es una tarea fácil. Millones de años de evolución nos han dotado con un sistema visual altamente sofisticado que nos permite reconocer patrones muy complejos del mundo tridimensional en el que vivimos. Distinguir formas, sombras y color, o incluso cosas más generales, como movimiento, potenciales amenazas y rostros de conocidos, son algunas de las actividades que nuestros cerebros realizan casi de manera inconsciente. Y si bien estas tareas pueden ser fáciles para nosotros, no es tal el caso para las computadoras.

Richard Szeliski caracteriza a la visión por computadoras como un *problema inverso*, es decir, se busca describir el mundo que uno ve en una o más imágenes y reconstruir sus propiedades tales como forma, iluminación y distribuciones de color [47]

Una de las áreas principales de de la visión por computadoras es la clasificación de imágenes. Es decir, dada una imagen y un conjunto de categorías, determinar a cual de las categorías pertenece esa imagen. Tener un buen entendimiento de los algoritmos de clasificación es crucial para desarrollar otras tareas dentro de la visión por computadoras dado que muchos problemas pueden ser reducidos a clasificación: detección de objetos, descripción de imágenes y segmentación entre otros.

Sin embargo, la clasificación de imágenes es un problema más difícil de lo que aparenta. Una imagen es solamente un conjunto de números (llamados *píxeles*). Surge la pregunta entonces, ¿cómo darle significado a un conjunto de números?. Se podría pensar en elaborar alguna métrica de distancia con los píxeles de otra imagen cuya categoría sea conocida, y si la distancia es menor a un cierto umbral sabríamos que ambas pertenecen a la misma clase. Puesto en otras palabras, lo que hicimos fue elegir una *representación* de la imagen basada en sus píxeles.

Sin embargo esta representación carece de robustez, ya que el menor cambio de *iluminación* podría alterar las métricas y confundir a nuestro modelo. No solamente eso, los objetos en las imágenes podrían estar parcialmente ocultos por el ambiente (*oclusión*), o en posiciones diferentes (*deformación*), o ser exactamente iguales, pero variar en colores y pequeños detalles (*variaciónintraclase*).

Hay mejores métodos para representar imágenes: representaciones basadas en color [39] [13] [40] [21], textura [30] [16] y descriptores locales [29] [42] [36] [3] [10] entre otros. Todos ellos fueron pensados para atacar problemas

puntuales en diversas áreas del procesamiento de imágenes, lo que significa que utilizarlos para generar una representacion de una imagen requiere un diseño manual y adaptado al dominio de cada problema en particular.

Una vez se tiene una representación adecuada de la imagen, podemos encarar el problema antes mencionado de clasificación. Numerosos métodos se han utilizado a lo largo de la historia: *máquinas vectores de soporte* [37], árboles de decisión [12], *bolsas de palabras* [9] entre otros. Comúnmente el proceso consiste en obtener buenas representaciones mediante los métodos manuales previamente mencionados y luego entrenar los algoritmos.

Sin embargo, en los últimos años todos los algoritmos relativos a la clasificación de imágenes fueron ampliamente superados por las Redes Neuronales Convolucionales. Desde el año 2010, todos los equipos ganadores del desafío ImageNet usaron Redes Neuronales Convolucionales, cada vez con resultados más precisos [22]. Un aspecto radicalmente distinto entre las redes neuronales convolucionales y los métodos mencionados anteriormente es que las primeras pueden obtener una representación de la imagen por si mismas. Esto significa que pueden aprender representaciones que sean más útiles al dominio del problema.

A pesar de haber tenido su golpe de popularidad en los últimos años, los primeros esbozos de modelar redes neuronales artificiales datan de 1958, cuando Frank Rosenblatt ideó el *Rosenblatt58theperceptron:*, un algoritmo para reconocimiento de patrones basado en una red de dos capas de aprendizaje [41]. Sin embargo, en 1969 se estableció que el poder de cómputo disponible en ese entonces no bastaba para poder entrenar y correr grandes redes neuronales, implicando que el área se estancara durante años [34]. Tan así es, que recién en 2006, con el abaratamiento de costos en hardware de alto desempeño se pudieron implementar arquitecturas más complejas (no necesariamente nuevas) y redes neuronales más profundas, algo que se conoce como Aprendizaje Profundo (*Deep Learning*).

Para poder entrenar Redes Neuronales Profundas, es necesario contar con un conjunto de datos anotados muy grande, cuyos tamaños pueden ir de las decenas de miles hasta millones de imágenes. Generalmente se requiere un gran esfuerzo humano para etiquetar tantas imágenes, por lo que los conjuntos de datos de entrenamiento lo suficientemente grandes suelen ser escasos, por ejemplo ImageNet ILSVRC'12 [22] (1 millón de imágenes distribuidas en 1000 clases), COCO (300 mil imágenes con 2.5 millones de objetos segmentados manualmente) [28] o SUN397 [51] (130 mil imágenes distribuidas en 397 clases).

Actualmente contamos con una increíble cantidad de imágenes para etiquetar. Para 2016, Cisco estimaba que el 51 % del tráfico de Internet iba a

provenir de dispositivos WiFi, tales como celulares, *tablets*, *smart TVs*, etc. Más aún, se estima que para el 2019 el 80 % del tráfico IP va a ser en forma de píxeles (multimedia), superando al 67 % que existente en 2014 [8]. Como tendencia podemos nombrar a Youtube, donde la mitad de los videos son subidos desde dispositivos móviles [53]. Por otro lado, las redes sociales más populares como Facebook, Flickr o Instagram almacenan una gigantesca cantidad de imágenes, contando la última con más de 80 millones de imágenes subidas por día [23]. Podríamos concluir que hay suficientes recursos para generar buenos conjuntos de datos, pero es prácticamente imposible contar con los recursos suficientes para anotar tantos videos, imágenes y demás contenido multimedia.

Una posible solución al entrenamiento de redes profundas cuando no se puede anotar una gran cantidad de datos es la propuesta por Agrawal et al. [1]. En la misma proponen utilizar información odométrica -inclinación, movimiento, rotación- disponible en agentes móviles (giróscopos, acelerómetros, etc.) para *preparar* los modelos de redes neuronales profundas, y luego realizar una transferencia de aprendizaje sobre un conjunto de datos anotados que se desee, obteniendo la ventaja de no necesitar un conjunto tan grande para lograr buena precisión. Este trabajo final tiene como objetivo evaluar experimentalmente las metodologías propuestas en Ref. [1].

El trabajo está organizado como sigue: en la Sección 2 se introducen conceptos del aprendizaje automático y las redes neuronales artificiales para concluir con redes convolucionales, en la Sección 3 se presenta un método para entrenar modelos de aprendizaje profundo mediante redes siamesas, que luego se aplicará a los experimentos en la Sección 4. Finalmente, las secciones 5 y 6 presentan una discusión final y trabajo a futuro.

Capítulo 2

2. Marco Teórico

2.1. Introducción

En esta Sección se introducirán conceptos de aprendizaje automático sobre los que definiremos diferentes algoritmos comunes en tareas de entrenamiento supervisado, para luego hacer foco en redes neuronales. Se definirán conceptos básicos como clasificadores lineales y funciones de costo junto con el algoritmo más popular utilizado en la minimización de las mismas, el descenso de gradiente estocástico. Introduciremos además el algoritmo de *retropropagación* (*backpropagation* en inglés), que es una manera eficiente de computar los gradientes durante el entrenamiento de un modelo cuando la cantidad de parámetros es muy grande.

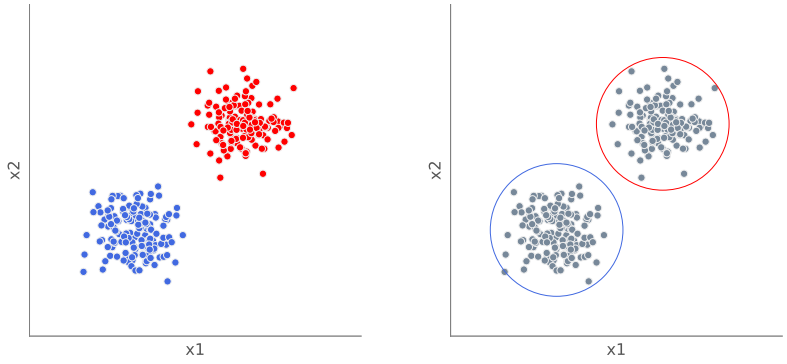
Se presentarán además los problemas principales a la hora de entrenar un modelo, como el sobreajuste, y métodos para evitarlos.

Este Capítulo concluye con una introducción a redes neuronales convolucionales y una breve mención de las arquitecturas de redes más conocidas, una de las cuales (*AlexNet*) es la base de las arquitecturas utilizadas en los experimentos de la Sección 4.

2.2. Aprendizaje Automático

Las técnicas de aprendizaje automático tienen como objetivo identificar patrones en conjuntos de datos utilizando herramientas de la estadística, teoría de la información, cálculo y optimización entre otras.

El aprendizaje automático adquiere relevancia cuando las tareas que se desean automatizar son demasiado complejas para programarse directamente. Como ejemplo tomemos la tarea de verificación de rostros. Supongamos que queremos crear un sistema que genere representaciones de imágenes de caras para luego diferenciarlas. El sistema debe tener en cuenta detalles como variaciones en sombra, color, orientación, por no mencionar las diferentes características que hay que extraer de una cara para diferenciarla de otra (arrugas, prominencias, etc.) [48]. Se puede ver que son demasiados detalles y combinaciones a tener en cuenta como para programar cada caso posible manualmente, por lo que un sistema que utilice aprendizaje automático que aprenda a diferencias esas características por nosotros podría ser una mejor opción.



(a) Aprendizaje supervisado (b) Aprendizaje no supervisado

Figura 1: Ejemplos simples de la organización de los datos en problemas de aprendizaje supervisado vs. no supervisado.

Hay tres áreas principales en las que podemos dividir el aprendizaje automático:

Aprendizaje supervisado Cuando se poseen anotaciones o alguna clase de etiqueta sobre los datos a aprender, hablamos de aprendizaje supervisado. Retomando el caso del verificador de rostros, las etiquetas serían el nombre o algún identificador de cada persona y nuestro clasificador aprendería a diferenciar las caras tomando como referencia las anotaciones.

Aprendizaje no supervisado Cuando los datos no están categorizados de antemano hablamos de aprendizaje no supervisado. Por ejemplo, si se contara con una lista de casas con sus respectivos precios, su área en metros cúbicos y cantidad de habitaciones y quisiéramos encontrar alguna relación entre ellas. Si tomamos como ejemplo la tarea de clasificación, en la Figura 1 se puede observar la diferencia entre los datos utilizados en aprendizaje supervisado (dos clases de elementos) y los datos de aprendizaje no supervisado (no se conocen las categorías de antemano y el algoritmo debe encontrar los patrones en los datos).

Aprendizaje por refuerzo Cuando el agente debe interactuar con su entorno para cumplir algún objetivo (por ejemplo manejar un vehículo [43] o ganar un videojuego [32]) y debe aprender a partir de las recompensas y penalizaciones que surgen mientras explora el espacio del problema.

2.2.1. El problema de clasificación

Supongamos que contamos con una población donde cada elemento pertenece a alguna clase de un conjunto de K clases en total. En este contexto, la tarea de clasificación consiste en determinar a qué clase pertenece una observación nueva.

Imaginemos que queremos clasificar imágenes, es decir, asignar una etiquetas a representaciones de imágenes desconocidas. Para ello vamos a definir una función f que mapee estas representaciones $\boldsymbol{x}$ a puntajes (*scores*) para cada etiqueta. Tomemos por ahora como representación de una imagen a sus píxeles. Supongamos que contamos con un conjunto de datos de imágenes $\boldsymbol{x_i} \in R^D$, donde $i = 1 \cdots N$, D es la dimensión de cada representación de una imagen y $y_i = 1 \cdots K$ es la etiqueta asociada. Es decir, tenemos N imágenes y K categorías.

Definamos ahora una función $f \colon R^D \mapsto R^K$ como un mapeo entre píxeles y *scores*:

$$f(\boldsymbol{x_i}, \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{W}\boldsymbol{x_i} + \boldsymbol{b} \tag{1}$$

Lo que acabamos de definir es un clasificador lineal. Un clasificador lineal combina linealmente las características (o *features*) de los datos de entrada para determinar a que clase pertenecen los mismos, y usualmente es entrenado mediante técnicas de aprendizaje supervisado.

Asumimos que la imagen $\boldsymbol{x_i}$ es un vector de una sola columna $[D \times 1]$, $\boldsymbol{W}$ es una matriz $[K \times D]$ y $\boldsymbol{b}$ es otro vector $[K \times 1]$. A menudo la matriz $\boldsymbol{W}$ es llamada los *pesos* de f, y a $\boldsymbol{b}$ el *vector de sesgo* dado que influencia los *scores* de salida, pero sin interactuar con $\boldsymbol{x_i}$.

Para entender mejor a los clasificadores lineales, podemos verlos de la siguiente manera: si la imagen tiene 32×32 píxeles y la representamos con un vector columna de dimensión D (en este caso $D = 1024 = 32 \times 32$), entonces en ese espacio *D-dimensional* la imagen es solamente un punto. Como se observa en la Figura 2 de manera simplificada, nuestro clasificador lineal define un hiperplano que separa cada clase dentro de ese espacio multidimensional.

Más adelante veremos cómo definir $\boldsymbol{W}$ y $\boldsymbol{b}$ para obtener un buen clasificador.

2.2.2. Entrenamiento

En el caso de los clasificadores lineales, entrenar un modelo se traduce en encontrar buenos valores de $\boldsymbol{W}$ y $\boldsymbol{b}$ que minimicen el *error* de acuerdo a algún criterio sobre el conjunto de entrenamiento.

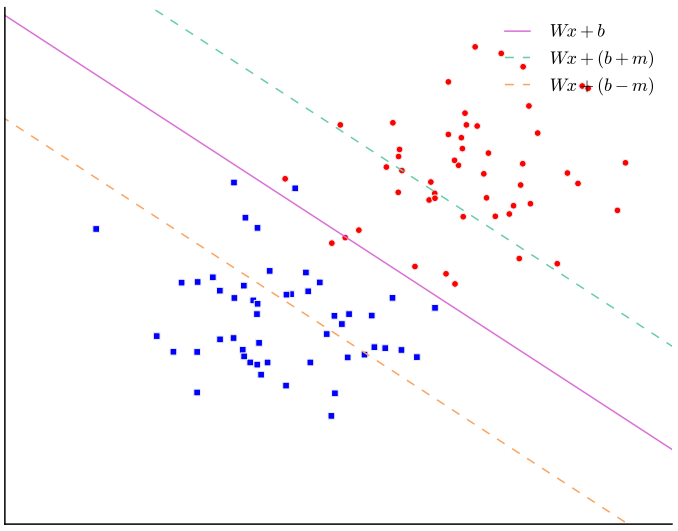


Figura 2: Visualización de un clasificador lineal binario. Cada punto representa una muestra en un espacio de dimensión $D = 2$ con $K = 2$ categorías. En este ejemplo el objetivo es establecer un hiperplano que defina un límite entre ambas clases, definido por la ecuación $f(\boldsymbol{x_i}, \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{W}\boldsymbol{x_i} + \boldsymbol{b}$. A modo de ejemplo están graficados dos clasificadores lineales más con el *vector de sesgo* ligeramente modificado. Se puede observar que $\boldsymbol{b}$ no afecta al clasificador sino que simplemente lo traslada a lo largo de las dimensiones.

Es muy común, cuando se cuenta con un conjunto de datos lo suficientemente grande, dividirlo en al menos 3 subconjuntos disjuntos: uno para entrenar el modelo, un segundo para validar el modelo durante el entrenamiento y un tercero para probar el modelo una vez entrenado. Es importante que esta división sea disjunta para poder evaluar qué tan bien generaliza nuestro modelo a nuevos datos (ver problema de *sobreajuste* en la Sección 2.4.3).

A grandes rasgos, podemos describir el proceso de entrenamiento de un clasificador de la siguiente manera:

1. Primero se mide el error actual del modelo con un subconjunto de datos de entrenamiento
2. Luego se actualizan los parámetros del clasificador ($\boldsymbol{W}$ y $\boldsymbol{b}$) para reducir ese error
3. Se repiten los pasos anteriores hasta lograr la convergencia del modelo

Por lo tanto hay dos aspectos a tener en cuenta antes de entrenar un modelo: cómo medir efectivamente la tasa de error y cómo actualizar sus parámetros para minimizar la misma. Para el primer caso se define lo que se llama una *función de pérdida o costo*, mientras que para el segundo analizaremos una técnica muy utilizada en aprendizaje automático denominada *descenso de gradiente*. Esto no significa que sea la única alternativa para entrenar modelos, pero al ser ampliamente utilizada en redes neuronales artificiales será la única que analizaremos.

2.2.2.1. Función de costo Una función de costo define un criterio de optimalidad que nos ayuda a saber que tan bien o mal está actuando nuestro clasificador. Es decir, si la tasa de error del clasificador es muy alta, el costo o la *pérdida* será muy alta y viceversa.

Sea L la función de costo de la predicción la clase de x_i cuando su etiqueta de clase asociada es y_i utilizando la función f con parámetros W, y supongamos que se cuenta con m datos de entrenamiento. Entonces el costo total de $f(x_i; W)$ para todo el conjunto de datos es:

$$L(\boldsymbol{W}) = \frac{1}{m} \sum_{i}^{m} L(f(\boldsymbol{x_i}; \boldsymbol{W}), y_i) \tag{2}$$

2.2.2.2. Descenso de Gradiente Ya contamos con una función para medir que tan bien o que tan mal está comportándose nuestro modelo, la *función de pérdida*. Como se puede observar, esta función depende de nuestro $\boldsymbol{W}$ y las imágenes (o *features* de entrenamiento que estemos usando). Nosotros no tenemos control sobre nuestro conjunto de datos de entrenamiento, pero sí podemos modificar los parámetros de $\boldsymbol{W}$ para producir la menor pérdida posible.

Para entender el algoritmo de descenso de gradiente tomemos un escenario hipotético: imaginemos por un momento que una persona con los ojos vendados está atrapada entre montañas y busca llegar al valle. Una manera de llegar al valle es probar dando un pequeño paso a su alrededor, y "sentir"hacia donde desciende más rápido la montaña, sólo valiéndose de la información local para moverse. Cuando finalmente esté seguro hacia donde descender, dará varios pasos en esa dirección, se detendrá y volverá a observar. Sabemos que eventualmente llegará al fondo del valle, pues lo único que tiene que hacer es seguir bajando por la pendiente de la montaña.

Formalmente, la pendiente de la montaña es la pendiente de la función de costo L que estemos utilizando y la dirección hacia donde bajar se corresponde con la dirección negativa del gradiente de L en ese punto, ya que la función *decrece* en el sentido opuesto que indica el gradiente. En otras palabras, lo que estamos haciendo es buscar el mínimo de L, y en consecuencia, el conjunto de parámetros de $\boldsymbol{W}$ que minimicen el costo.

El descenso de gradiente (SGD por sus siglas en inglés) se utiliza para optimizar los pesos partiendo de la premisa que el modelo es diferenciable localmente (o se puede aproximar su derivada) con respecto a $\boldsymbol{W}$. Dado que queremos minimizar la función de costo L, lo que vamos a hacer es calcular su gradiente ∇L respecto a cada parámetro y luego modificar cada uno ligeramente con el objetivo de acercarlo al mínimo en la función. Entonces, si W_n son nuestros parámetros en el paso n del entrenamiento, calculamos W_{n+1} de la siguiente manera:

$$\boldsymbol{W_{n+1}} = \boldsymbol{W_n} - \epsilon \frac{1}{m} \sum_{i}^{m} \nabla_{\boldsymbol{W_n}} L(f(\boldsymbol{x_i}; \boldsymbol{W_n}), y_i) \tag{3}$$

Donde ϵ es conocido como la *tasa de aprendizaje* y m es la cantidad de elementos en el conjunto de datos. Notar que la tasa de aprendizaje determina una fracción del gradiente a sustraer. En la ecuación se puede observar que se modifican los parámetros con respecto a la dirección opuesta al gradiente, dado que el mismo indica la dirección de crecimiento de una función pero nosotros queremos encontrar un mínimo (Figura 3).

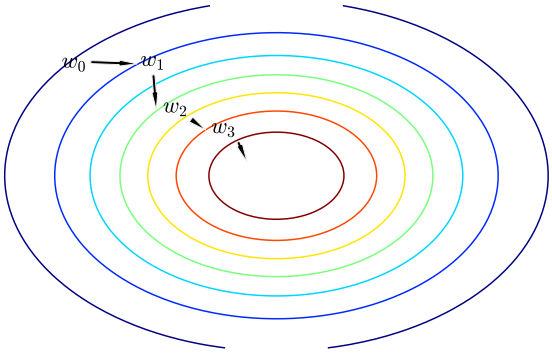


Figura 3: Descenso de gradiente. Una función de clasificación f alcaza un mínimo en su parámetro w a medida que se actualiza su valor mediante la substracción del gradiente calculado en ese parámetro.

Lo más comun al utilizar SGD es mediante una técnica llamada *descenso de gradiente por mini-batch*, que se basa en calcular el gradiente de un subconjunto del total de datos (llamado *mini-batch*) y actualizar $\boldsymbol{W}$ al final de cada iteración. Esto es muy útil dado que es computacionalmente costoso calcular el gradiente de todo un conjunto de datos con miles de imágenes a la vez y calcular el gradiente de un *batch* aproxima bastante bien el gradiente del total [11].

2.2.2.3. Clasificador *Softmax* Antes de saltar de lleno a las redes neuronales artificiales vamos a describir brevemente un tipo de clasificador muy utilizado en las mismas, el clasificador *Softmax*.

La función *Softmax* tiene la siguiente forma:

$$\sigma(\boldsymbol{x})_j = \frac{e^{f(\boldsymbol{x};\boldsymbol{W})_j}}{\sum_k e^{f(\boldsymbol{x};\boldsymbol{W})_k}} \tag{4}$$

Se puede observar que sus resultados son siempre positivos y normalizados respecto a todas las salidas, por lo que su resultado es siempre un número entre 0 y 1. Por lo tanto, *Softmax* se puede interpretar como la *probabilidad* de x de pertenecer a cada una de las clases k.

Si nuestro problema de clasificación cuenta con K clases, podemos codificar cada anotacion y_i de nuestro conjunto de datos como un vector de longitud K donde todos los elementos son 0 y el elemento del índice correspondiente a la clase de $\boldsymbol{x_i}$ es 1. Esta representación de las etiquetas se denomina *one-hot encoding*.

Ahora bien, si queremos obtener la probabilidad de y_i a partir del resultado de *Softmax*, simplemente debemos computar el producto interno entre ambos vectores:

$$\sigma(f(\boldsymbol{x_i}, \boldsymbol{W})_{y_i}) = \langle\, f(\boldsymbol{x_i}, \boldsymbol{W}), y_{\text{codificado}} \rangle \tag{5}$$

La función de costo utilizada comunmente con *Softmax* es el negativo de la función de verosimilitud (o *log-likelihood* en inglés). Sea $\sigma(f(\boldsymbol{x_i}, \boldsymbol{W})_{y_i})$ la probabilidad computada mediante *Softmax* para la clase y_i, entonces *log-likelihood* se expresa:

$$L_i = -\log\left(\frac{e^{\sigma(f(\boldsymbol{x_i},\boldsymbol{W})_{y_i})}}{\sum_j e^{\sigma(f(\boldsymbol{x_i},\boldsymbol{W})_j)}}\right) \tag{6}$$

Observar que si el clasificador se equivocó al predecir (o sea, asignó una probabilidad p muy baja a la clase correspondiente y_i) entonces el costo

será muy alto. Como $log(p) \to -\infty$ cuando $p \to 0$, intuitivamente podemos ver que $-log(p) \to \infty$ cuando $p \to 0$. En cambio si clasificador predijo con más probabilidad la clase y_i, significa que p es más cercano a 1 y por ende, $-log(p)$ es más cercano a 0.

Nuevamente, notar que la pérdida total de nuestro conjunto de datos en un determinado paso del entrenamiento es el promedio de las pérdidas de cada elemento del conjunto.

2.3. Redes Neuronales Artificiales

Hasta ahora analizamos clasificadores lineales y un tipo particular llamado softmax. Si conectáramos la salida de un clasificador lineal $\boldsymbol{s_1 = W_1 x + b_1}$ con la entrada de otro clasificador $\boldsymbol{s_2 = W_2 y + b_2}$, entonces obtendríamos un tercero:

$$\boldsymbol{s_3 = W_2(W_1 x + b_1) + b_2 = (W_2 W_1)x + (W_2 b_1) + b_2} \tag{7}$$

$$\boldsymbol{s_3 = W_3 x + b_3} \tag{8}$$

Es fácil hacer un chequeo de dimensiones para ver que efectivamente podemos "colapsar" las matrices $\boldsymbol{W_2}$ y $\boldsymbol{W_1}$ en una sola, por lo cual terminamos con otro clasificador lineal.

Notemos que por más que combinemos miles de clasificadores lineales vamos a obtener un nuevo clasificador también lineal. Una manera de romper la linealidad de estas "capas" de clasificadores es, por ejemplo, agregar lo que se llama *función de activación*:

$$s = \boldsymbol{W_2}\,\text{máx}\,(0, \boldsymbol{W_1 x + b_1}) + \boldsymbol{b_2} \tag{9}$$

Lo que acabamos de definir es una red neuronal básica de dos capas, de una neurona cada una.

En la Figura 4 vemos un modelo formal de una neurona estándar, en el que las entradas x_i interactúan multiplicativamente con los pesos $_i$. Luego, esos resultados se suman junto con un vector de sesgo y sobre eso se computa lo que se llama *función de activación* que decide si transmitir o no la salida.

2.3.1. Funciones de activación comunes

Se han propuesto varias funciones de activación a lo largo de los años. Nos concentraremos en las unidades *ReLU* (*Rectifier Linear Unit* en inglés), actualmente muy populares en las redes convolucionales.

Hay tres tipos de rectificadores lineales:

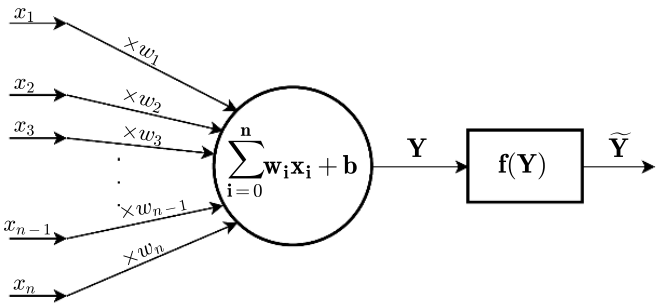


Figura 4: Esquema de una neurona artificial.

2.3.1.1. ***ReLU*** Una unidad ReLU establece un umbral en 0 a la salida de la neurona. Es decir, la activación de una neurona va a ser 0 si su salida fue negativa o un numero positivo en caso contrario (Figura 5.a):

$$f(x) = \max(0, x) \tag{10}$$

Se puede observar que requiere muy pocas operaciones, además está comprobado empíricamente que los modelos que utilizan *ReLU* convergen hasta 6 veces más rápido [25] que con otras funciones de activación (como la *sigmoide* [17] y *tangente hiperbólica*).

Una desventaja de las *ReLU* es que pueden provocar la "muerte" de neuronas durante el entrenamiento. El problema esta relacionado con el método más común de entrenamiento de redes neuronales, *backpropagation*. El algoritmo de *backpropagation* será explicado en la Sección 2.4.5, pero por ahora pensemos que el proceso de optimización de $\boldsymbol{W}$ implica restar un porcentaje del gradiente de la función de costo en $\boldsymbol{W}$. Si el gradiente es muy grande entonces los pesos sobre los que se realice la actualización terminarán siendo muy pequeños (negativos). Como consecuencia, la unidad *ReLU* no volverá a activarse, pues sus valores de entrada siempre van a ser valores negativos.

Esta situación puede agravarse si la tasa de aprendizaje es muy grande.

Una vez que la ReLU alcanza este estado, es improbable que vuelva a activarse, dado que su gradiente (aproximado por lo que se llama el *subgradiente*) en 0 es también 0, por lo que un entrenamiendo mediante descenso de gradiente y *backpropagation* no va a modificar los pesos locales, dejando a esa neurona "muerta".

2.3.1.2. *Leaky ReLU* Se puede solucionar el problema de la muerte de neuronas agregando una pequeña pendiente negativa (de 0.01 por ejemplo) en los valores negativos de la *ReLU*. Esta función de activación es la que se conoce como *Leaky ReLU* [6] (Figura 5.b):

$$L(x_{t_1}, x_{t_2}, W) = \begin{cases} D(x_{t_1}, x_{t_2}), & \text{si} |t_1 - t_2| \leq T \\ 1 - \text{máx}\,(0, m - D(x_{t_1}, x_{t_2})), & \text{si} |t_1 - t_2| > T \end{cases} \tag{11}$$

$$f(x) = \begin{cases} \alpha x, & \text{si} x < 0 \\ x, & \text{si} x >= 0 \end{cases} \tag{12}$$

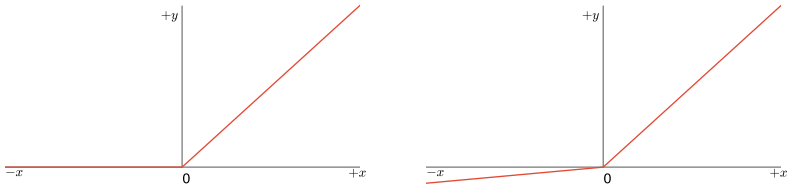
De esta manera nos aseguramos que al menos un pequeño gradiente fluya durante *backpropagation* cuando la neurona emite resultados negativos, permitiendo que se normalicen los pesos a mediano/largo plazo. Sin embargo no está demostrado que las *Leaky ReLU* presenten una mejora sustancial en el entrenamiento de las redes, por lo que las *ReLU* convencionales siguen siendo ampliamente usadas.

2.3.1.3. *Maxout*

$$f(x) = \text{máx}\,(w_1^T x + b_1, w_2^T x + b_2) \tag{13}$$

Maxout [15] es una generalización de las funciones *ReLU*, y obtiene lo mejor de los dos mundos: por un lado la forma lineal y no saturante de las *ReLUs* y por el otro evita el problema de la muerte de neuronas. A pesar de ello tiene la desventaja de duplicar los parámetros para cada neurona, lo cual no siempre es deseable, pues implica más tiempo de entrenamiento y más consumo de memoria y recursos, sobre todo en redes profundas.

Notar que una *ReLU* normal es básicamente una *maxout* con $w_1, b_1 = 0$.



(a) *ReLU* (b) *Leaky ReLU*

Figura 5: *ReLU* vs. *Leaky ReLU*. Se puede observar la pendiente negativa en 5b para $x < 0$, la cual produce un gradiente $\neq 0$ y evita la muerte de neuronas.

2.4. Entrenamiento de redes neuronales artificiales

Entrenar una red neuronal artificial no es muy distinto a entrenar un clasificador lineal. Necesitamos definir una función de pérdida y un método para ajustar los parámetros. Veremos además, como en otras tareas de aprendizaje automático, que hay que tener en cuenta el formato de los datos de entrada al model (tal vez eliminar ruido o redundancia, normalizar las dimensiones). Esta tarea se denomina preprocesamiento de datos.

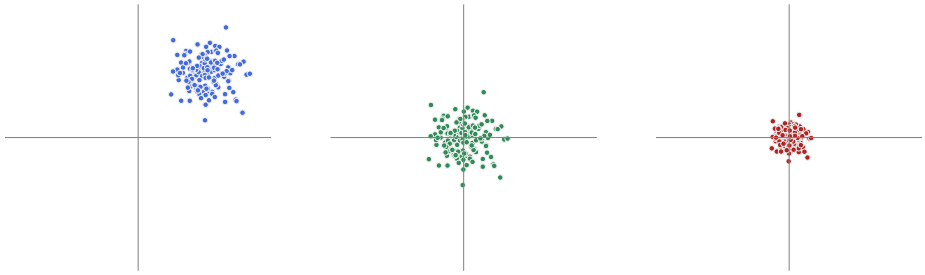
También analizaremos varias técnicas para evitar el sobre-ajuste de modelos. El sobre-ajuste surge cuando un modelo aprende "ruido" y detalles particulares del conjunto de datos en vez de características generales que ayuden a la tarea de clasificación.

Finalizaremos esta sección con un análisis de la organización interna de las redes neuronales artificiales y qué algoritmos se utilizan para entrenar.

2.4.1. Preprocesamiento de datos

Antes de comenzar con el entrenamiento de una red neuronal artificial es conveniente analizar los datos y si es necesario normalizarlos para que todos estén en el mismo rango de valores.

El preprocesamiento de datos, como alinear imágenes o normalizar valores ayuda a una mejor convergencia de los modelos. Las dos técnicas más comunes de preprocesamiento de datos para redes neuronales son la substracción de la media y la normalización.



(a) Original (b) Substraer media (c) Normalizacion

Figura 6: Se puede observar en 6b cómo al substraer la media de 6a logramos "centrar" nuestro conjunto de datos. En 6c podemos apreciar los resultados de la normalización de datos, logrando que todos los elementos pertenezcan al mismo rango de valores.

2.4.1.1. Substracción de la media Como su nombre lo indica, se le resta la media a cada elemento del conjunto de datos con el objetivo de *centrar* los datos alrededor del origen en todas las dimensiones (Figura 6b). Si hablamos de un conjunto de datos de imágenes esto equivale a restarle el valor medio de los píxeles a cada píxel de la imagen de entrada.

2.4.1.2. Normalización Una manera de normalizar los datos es dividir cada dimensión por su desviación estándar una vez que haya sido centrada en cero. De esta manera se logra que las dimensiones tengan aproximadamente la misma escala (Figura 6c). Notar que en general los píxeles tienen valores en el rango de 0 a 255, por lo que sus dimensiones ya se encuentran en escalas parecidas y cuando se trabaja con redes convolucionales no es estrictamente necesario normalizar los datos de entrada.

2.4.1.3. Otras maneras de preprocesar datos A la hora de entrenar redes convolucionales importan dos cosas: la calidad de los datos y la cantidad. Es necesario que además de preprocesar los datos con las técnicas usuales (substracción de media por ejemplo), se tengan en cuenta aspectos de más alto nivel. Por ejemplo, si estuvieramos entrenando una red de reconocimiento de rostros, puede ser mejor contar con un conjunto de datos de caras alineadas en vez de uno de caras en diferentes posiciones y ángulos que tenga más ruido. De esa manera vamos a lograr que la red aprenda mejor qué aspectos extraer de las imágenes.

Además no siempre se puede contar con un dataset de millones de imágenes para entrenar nuestra red, por lo que suele ser necesario aumentar

(a) Original (b) Alineacion (c) Color (d) Traslaciones

Figura 7: Diferentes formas de aumentar datos. Si trabajamos con rostros es muy común alinearlos, por ejemplo, sobre el eje que conforman los ojos 7b. Traslaciones, recortes en la imagen y cambios en el brillo, contraste y color son otras técnicas muy usadas (7c y 7d).

nuestros datos con técnicas de *data augmentation*: repetir la misma imagen pero con diferentes variaciones en el color, brillo, saturación, incluso hacer leves desplazamientos y rotaciones, como se puede observar en la Figura 7.

2.4.2. Inicialización de pesos

A la hora de inicializar los pesos es escencial romper con la simetría. Imaginemos que inicializamos todos los pesos en 0, algo que podría parecer razonable. En una capa completamente conectada, entonces todas las neuronas van a recibir el mismo valor de entrada 0 ($f(x) = \sum_i w_i x$ con $w_i = 0$) por lo que sus salidas van a ser todas iguales y por ende los gradientes que se calculen serán los mismos, produciendo que los pesos se actualicen todos iguales y la red no aprenda.

En cambio podemos inicializar los pesos con pequeños números aleatorios cercanos a cero. Una opción común es utilizar una distribución gaussiana con media cero y desviación estándar 0.01. Este método, si bien es bastante *ad-hoc*, es bastante usado. Hay muchas otras maneras más sofisticadas de inicializar los pesos de una red, pero su análisis escapa al alcance de este trabajo.

2.4.3. Evitando el sobre-ajuste: Regularización y Dropout

Cuando se aprende un modelo sobre un conjunto de datos, puede surgir el problema del *sobre-ajuste*, más conocido por su nombre en inglés *overfitting*. El *overfitting* significa que nuestro modelo ajustó sus parámetros demasiado bien al conjunto de datos de entrenamiento, provocando que aprendiera detalles insignificantes del mismo, principalmente *ruido* aleatorio. Como consecuencia, cuando se lo aplica en un conjunto de datos nuevo, el modelo

presenta un bajo rendimiento. En contrapartida al *overfitting*, a veces puede pasar que nuestro modelo aprendió pocas características de nuestro conjunto de entrenamiento y termina siendo muy genérico e inflexible a la hora de ser aplicado en un conjunto nuevo, obteniendo también baja precisión

A modo de ejemplo, en la Figura 8a se puede observar un gran sesgo en el caso de *underfitting*, que si bien permite una mayor generalización no logra distinguir el límite entre ambas clases, lo cual se traduce en menor precisión a la hora de evaluar el modelo. En la Figura 8c observamos un típico caso de *overfitting* con mucha variación y sensibilidad a los datos de entrenamiento, lo cual implica poca generalización a nuevos datos, mientras que en la Figura 8b se observa un buen ajuste del conjunto de datos.

Queremos elegir los mejores parámetros de $\boldsymbol{W}$ para evitar estos problemas, y eso lo podemos hacer agregando una penalidad de regularización $R(W)$. Lo que buscamos con esto es poner preferencias para algunos conjuntos de $\boldsymbol{W}$ sobre otros.

De esta manera, nuestra función de costo ahora cuenta con dos componentes, la función de costo propiamente dicha y la *componente de regularización*. Sea λ un número real (*término de regularización*), entonces nuestra nueva función de costo es:

$$L = \frac{1}{N} \sum_i L_i + \lambda R(W) \tag{14}$$

Notar que la pérdida total es el promedio de las pérdidas de cada imagen, y que la penalización de la regularización sólo se suma una vez.

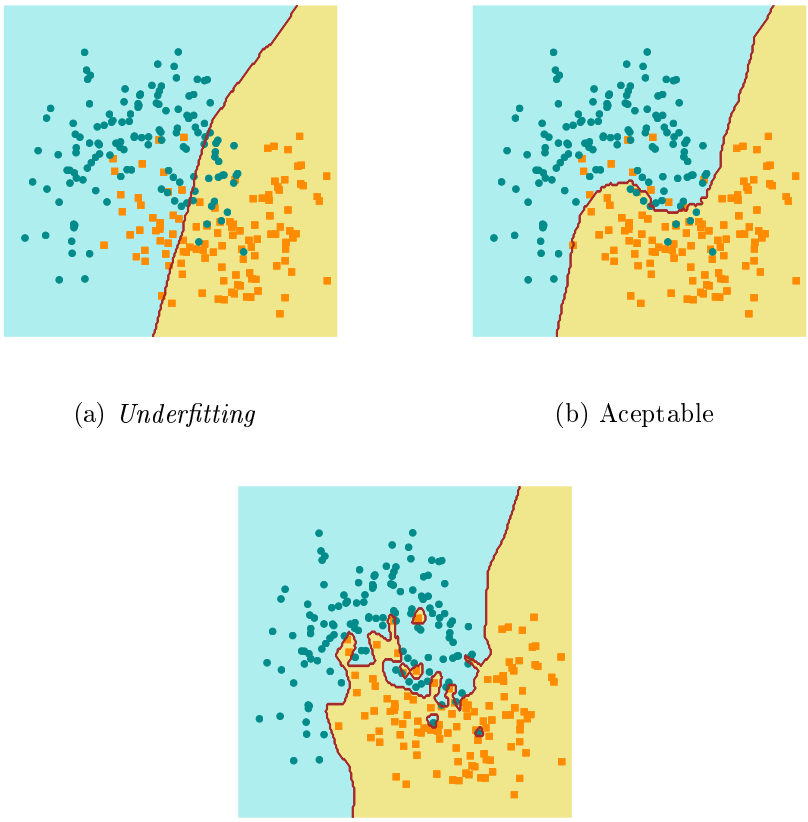
Las técnicas de regularizacion más usadas son:

2.4.3.1. Regularización de norma L2 (*weight decay*) Sea $\mathbf{W}$ la matriz de parámetros, podemos calcular su norma de Frobenius (norma L2 generalizada a matrices) como:

$$||\mathbf{W}||_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} w_{ij}^2} \tag{15}$$

Definimos la normalización L2 como $\frac{1}{2}\lambda||\mathbf{W}||_F^2$ donde λ es la tasa de regularización.

Una buena propiedad de la regularizacion es que al penalizar los pesos grandes, obliga a $\boldsymbol{W}$ a generalizar y contemplar todas las clases a la hora de clasificar. De esa manera, nuestro clasificador final va a tomar en cuenta



(a) *Underfitting*

(b) Aceptable

(c) *Overfitting*

Figura 8: Ejemplo de *overfitting* 8a, *underfitting* 8b y un buen ajuste al conjunto de datos 8c.

todas las dimensiones de entrada (algunas con más o menos probabilidad) sin dar prioridad a una sola.

2.4.3.2. Regularización de norma L1 Similar a la anterior, sólo que se le adiciona $\lambda||\mathbf{W}||_1$ a la función objetivo, donde $||\mathbf{W}||_1$ es la norma L1 de una matriz:

$$||\mathbf{W}||_1 = \sum_{i=1}^{n}\sum_{j=1}^{m}|w_{ij}| \tag{16}$$

Los pesos tienden a converger a cero bajo la regularización L1. En general se prefiere la regularización L2 por obtenerse mejores resultados.

2.4.3.3. *Dropout* La técnica de *dropout* [45] consiste en mantener activa una neurona con una probabilidad p (a veces $1-p$). Esta técnica se aplica solamente durante el *entrenamiento* de las redes neuronales.

Si consideramos una red neuronal con L capas, sea $l \in \{1, \ldots, L\}$ el índice de cada capa oculta de la red. Sea $\boldsymbol{z}^{(l)}$ el vector de entrada a la capa l, $\boldsymbol{y}^{(l)}$ el vector de salidas de la capa l ($\boldsymbol{y}^{(0)} = \boldsymbol{x}$ son los datos de entrada a la red). $W^{(l)}$ y $\boldsymbol{b}^{(l)}$ son los parametros de la capa l. Dada una neurona i de la capa l, la operacion de *feed-forward* (o sea, cuando se procesa una imagen a traves de todas las capas de la red) de la red puede ser descripta como:

$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)}\boldsymbol{y}^l + b_i^{(l+1)}, \tag{17}$$

$$y_i^{(l+1)} = A(z_i^{(l+1)}) \tag{18}$$

Donde A es una función de activación. Si ahora agregamos *dropout*:

$$r_j^{(l)} \sim Bernoulli(p), \tag{19}$$

$$\tilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} \odot \boldsymbol{y}^{(l)}, \tag{20}$$

$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)}\tilde{\boldsymbol{y}}^l + b_i^{(l+1)}, \tag{21}$$

$$y_i^{(l+1)} = A(z_i^{(l+1)}) \tag{22}$$

Aquí $\odot$ denota el producto elemento a elemento y $\boldsymbol{r}^{(l)}$ es un vector de variables aleatorias de Bernoulli con probabilidad p de ser 1. Para cada capa

se calcula este vector $\boldsymbol{r}^{(l)}$ y luego se lo multiplica elemento a elemento por $\boldsymbol{y}^{(l)}$ para reducir la cantidad de neuronas activas, obteniendo como resultado $\tilde{\boldsymbol{y}}^{(l)}$ que a su vez va a ser la entrada de la capa siguiente.

En un entrenamiento sin *dropout*, la red actualiza todas sus neuronas en cada iteración (Figura 9-izquierda), mientras que utilizando *dropout* se eliminan neuronas aleatoriamente y se utiliza una *subred* de la original, lo cual impide a las neuronas co-adaptarse entre si (Figura 9-derecha). La co-adaptación ocurre cuando dos o más neuronas consecutivas dependen mucho entre ellas para detectar *features*, en vez de que cada neurona busque un tipo particular de *feature*.

Otra manera de pensarlo es la siguiente: una red neuronal con n neuronas puede ser vista como una colección de 2^n *subredes* que comparten todas los mismos pesos ($\boldsymbol{W}$), por lo cual la cantidad total de parámetros sigue siendo a lo sumo $O(n^2)$. Para cada elemento en el conjunto de entrenamiento se elige una de estas 2^n redes y apenas se la entrena. Esto es comparable a entrenar distintos modelos y luego promediar sus predicciones, algo que en general es muy útil en aprendizaje automático pero rara vez se hace en aprendizaje profundo debido a que cada modelo tarda mucho en entrenarse y es muy tedioso elegir buenos hiperparámetros.

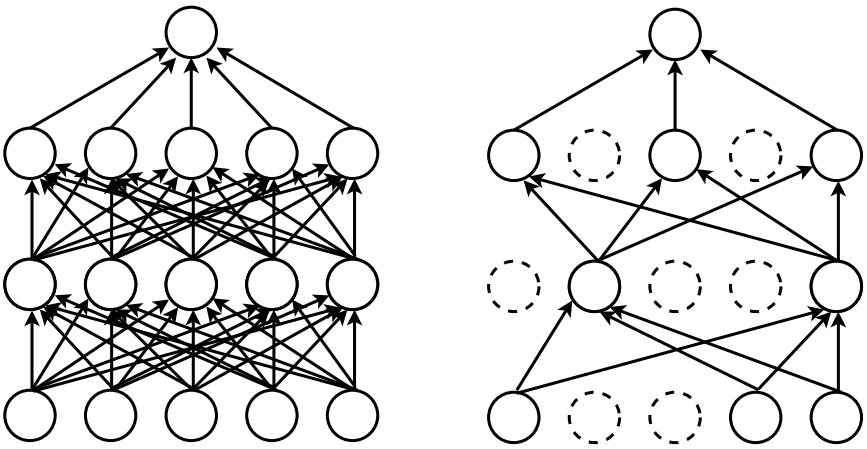


Figura 9: A la izquierda, una red neuronal normal; a la derecha, una red neuronal luego de aplicar *dropout*.

2.4.4. Organización de las redes neuronales

Las redes neuronales estan organizadas como un grafo acíclico de neuronas, donde las salidas de unas se transforma en la entrada de otras. Las

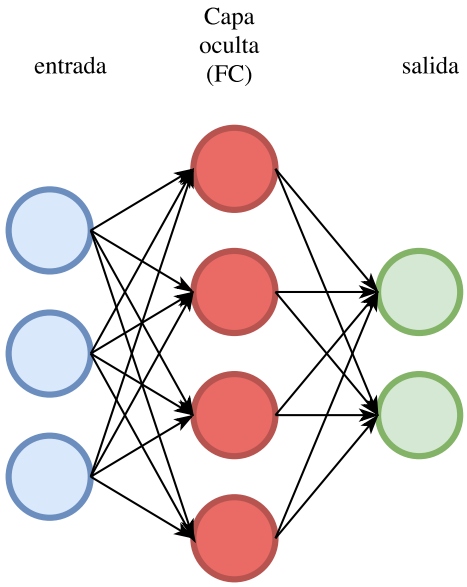


Figura 10: Diagrama de una red neuronal con entrada de dimensión $D = 3$, una capa oculta completamente conectada y dos neuronas de salida.

neuronas se organizan en distintas capas conectadas, de esa manera los cálculos se hacen con operaciones entre matrices, algo que no podríamos hacer tan fácil si tuvieramos neuronas conectadas aleatoriamente entre ellas.

El tipo más común de capa de neuronas es la capa *totalmente conectada* (de ahora en más FC, abreviación de su nombre en inglés *Fully Connected*), en donde cada neurona de la capa anterior se conecta con todas las neuronas de la capa siguiente, pero no comparten conexiones dentro de la misma capa.

Usualmente no se cuenta a la capa de entrada de las redes como una capa más, y la capa de salida no tiene funciones de activación, dado que generalmente representan las puntuaciones de cada clase (en clasificación) o alguna métrica (en regresión). Las redes neuronales suelen tener una o más capas intermedias entre la entrada y la salida, denominadas *capas ocultas* (Figura 10).

Las redes neuronales se entrenan partiendo del principio del descenso del gradiente que se explicó en la Sección 2.2.2.2. Notemos que L es una función que depende de las imágenes de entrada $\boldsymbol{x_i}$, $\boldsymbol{W}$ y $\boldsymbol{b}$. Sin embargo, como ya dijimos, el conjunto de datos de entrenamiento es algo fijo en nuestro modelo, por lo que sólo nos interesa calcular el gradiente sobre $\boldsymbol{W}$ y $\boldsymbol{b}$ para poder actualizar sus parámetros. Ahora bien, derivar una función con millones de

parámetros (cantidad que suelen tener las redes neuronales artificiales) es computacionalmente costoso, por lo que para actualizar $\boldsymbol{W}$ con los nuevos pesos se utiliza el algoritmo *retropropagación* (*backpropagation* en inglés).

2.4.5. *Backpropagation*

La retropropagación (o *backpropagation* en inglés) es un algoritmo utilizado para computar eficientemente el gradiente.

Para lograr una intuición del mismo empecemos por recordar la definición de derivada de una función. Sea $g : \mathbb{R} \to \mathbb{R}$, entonces su derivada se expresa como:

$$\frac{dg(x)}{dx} = \lim_{h \to 0} \frac{g(x+h) - g(x)}{h} \tag{23}$$

Si el dominio de g estuviera en $\mathbb{R}^n$, se calculan derivadas parciales y llamamos *gradiente* es simplemente un vector conteniendo derivadas. Por ejemplo, sea $n = 2$ y por lo tanto g una función que toma dos parámetros x e y, entonces el gradiente de g es $\nabla g = [\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}]$

Usualmente podemos derivar con métodos numéricos, pero es lento y sólo aproxima los resultados. Veremos más adelante que la función L de las redes neuronales suele tener decenas de millones de parámetros, y realizar tantas operaciones para una sola actualización de $\boldsymbol{W}$ no es conveniente. En la práctica usaremos el cálculo analítico del gradiente, en el cual derivamos una fórmula directa que es muy rápida de computar valiéndonos de la *regla de la cadena*.

La *regla de la cadena* nos ayuda a descomponer el cálculo del gradiente de expresiones complejas en pequeños pasos. Tomemos por ejemplo una función $g : \mathbb{R}^3 \to \mathbb{R}$:

$$g(x, y, z) = \frac{x}{y^2} + z \tag{24}$$

Si quisiéramos obtener su gradiente en x, y, z de la manera tradicional calculando el cociente de $g(x+h) - g(x)$ con h cuando $h \to 0$ deberíamos realizar muchos cálculos computacionalmente costosos. En cambio, podemos ver a g como una composición de funciones:

$$g(x, y, z) = \frac{x}{y^2} + z = \frac{x}{p} + z = q + z \tag{25}$$

Y calcular su gradiente valiéndonos de la *regla de la cadena*:

$$\frac{\partial g}{\partial q} = 1 \tag{26}$$

$$\frac{\partial q}{\partial p} = \frac{-x}{p^2} = \frac{-x}{y^4} \tag{27}$$

$$\frac{\partial q}{\partial x} = \frac{1}{y^2} \tag{28}$$

$$\frac{\partial p}{\partial y} = 2y \tag{29}$$

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial q}\frac{\partial q}{\partial x} = \frac{1}{y^2} \tag{30}$$

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial q}\frac{\partial q}{\partial p}\frac{\partial p}{\partial y} = \frac{-2x}{y^3} \tag{31}$$

$$\frac{\partial g}{\partial z} = 1 \tag{32}$$

Ahora podemos estructurar nuestro algoritmo de optimización en dos pasos: primero, evaluamos nuestra función L en los parámetros actuales (*forward pass*). Luego, partiendo de ese resultado calculamos el gradiente en cada variable utilizando la *regla de la cadena*. En la Figura 11 se pueden observar los valores computados para este ejemplo, en el caso particular de $x = -3$, $y = 2$, $z = 4$. El *forward pass* se marca en verde y los gradientes computados mediante la regla de la cadena (*backpropagation*) en rojo. De esta manera "propagamos" el error de la predicción hacia atrás (*backpropagation*) para luego corregir los pesos mediante, por ejemplo, el algoritmo de Descenso de Gradiente Estocástico que ya vimos.

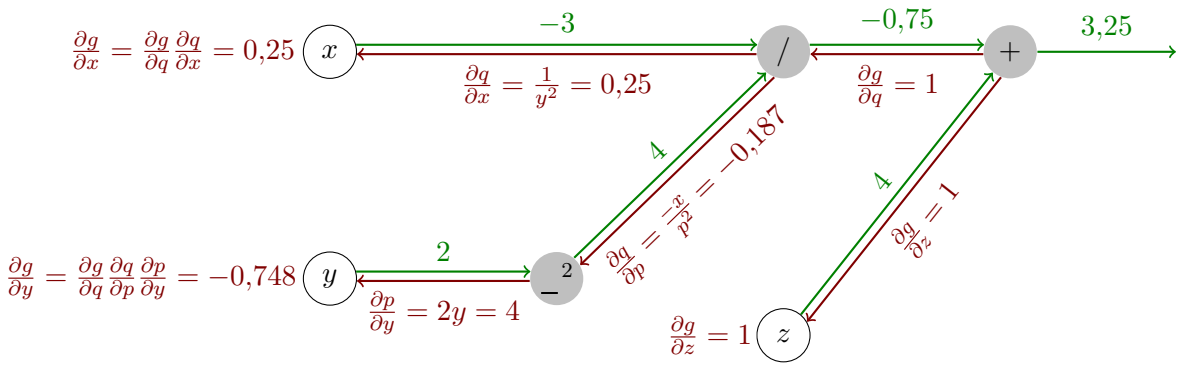


Figura 11: Grafo de computación durante la aplicación de *backpropagation*.

Una vez que contamos con el gradiente, actualizamos los parámetros de L restándole un porcentaje del gradiente negativo calculado (negativo porque queremos ir en dirección opuesta a donde crece la función, o sea, ir a su mínimo). Ese porcentaje es llamado *tasa de aprendizaje* (*learning rate*) y suele ser uno de los parámetros más difíciles de elegir, ya que la calidad y rapidez de aprendizaje dependen de él.

Notemos que el algoritmo *backpropagation* requiere que la función a optimizar sea derivable. Esto incluye a las funciones de activación previamente mencionadas, como las unidades ReLU. Si bien la derivada de una unidad ReLU no está definida en 0, puede ser aproximada mediante:

$$f'(x) = \begin{cases} 0, & \text{si } x \leq 0 \\ 1, & \text{si } x \geq 0 \end{cases} \tag{33}$$

Lo cual implica que es posible utilizar *backpropagation* en redes neuronales con activaciones ReLU.

Idealmente computaríamos el gradiente sobre todo el conjunto de datos, actualizaríamos los parámetros, y repetiríamos el ciclo hasta lograr la convergencia. Sin embargo los conjuntos de datos para entrenar las redes neuronales suelen tener cientos de miles o incluso millones de imágenes, por lo cual se utiliza una técnica llamada *Descenso de Gradiente Estocástico con mini-batches* o simplemente *SGD* por sus siglas en inglés, en el cual se calcula el gradiente para una cantidad predeterminada de imágenes (*mini-batches*), se actualizan los parámetros y se vuelve a repetir el ciclo con otro subconjunto

distinto. Esto parte de la suposición de que todas las imágenes del conjunto de datos estan correlacionadas entre sí. El tamaño de los *mini-batches* no es estrictamente un hiperparámetro que uno pueda validar durante el entrenamiento, sino que más bien depende del hardware sobre el que se esté entrenando la red (en general se eligen potencias de dos por cuestiones de eficiencia). No obstante puede haber casos en los que se elija un tamaño de *mini-batch* chico para tener más varianza en los datos de entrada de la red y evitar que la se caiga en un mínimo local.

2.4.5.1. Transferencia de aprendizaje Entrenar un modelo con un tipo específico de problema y luego utilizar su *conocimiento* para resolver otro problema nuevo, tal vez incluso en un área distinta a la que fue pensado originalmente, es lo que se llama transferencia de aprendizaje. Esta técnica ha cobrado importancia en *deep learning* dado que a menudo las arquitecturas son muy complejas y se tardan semanas en lograr la convergencia deseada, por lo que contar con modelos preentrenados sobre los cuales se puedan ajustar ligeramente los parámetros para resolver un nuevo problema es una ventaja. La mayoría de los *frameworks* modernos para implementar redes neuronales soportan realizar transferencia de aprendizaje utilizando modelos pre-entrenados.

2.5. Redes Neuronales Convolucionales

Podemos representar a una imagen como una función $f : \mathbb{R}^2 \to \mathbb{R}^n$ con n igual número de canales que pueda tener (usualmente 1 ó 3). Es decir, toma un punto en coordenadas cartesianas y devuelve la intensidad en ese punto.

El caso general de un *operador* de procesamiento de imágenes es el de una función que toma una o más imágenes de entrada y produce una imagen de salida. Para el caso discreto en el que el dominio consiste de un número finito de píxeles donde representamos a cada píxel como su posición en la imagen, $\boldsymbol{x} = (i, j)$, podemos expresar a un operador de píxeles como:

$$g(i, j) = h(f(i, j)) \tag{34}$$

Un operador muy utilizado en la visión por computadoras es el *filtro lineal*. El mismo es un tipo de *operador local* dado que usa el conjuto de píxeles en la vecindad de uno para determinar su nuevo valor. En un filtro lineal cada píxel de salida se determina como la suma ponderada de los valores de entrada:

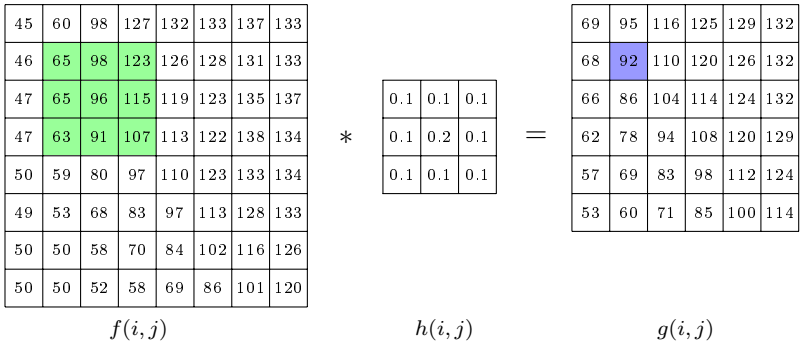


Figura 12: Convolución. La imagen en la izquierda se convoluciona con el filtro h para dar lugar a la imagen de la derecha. El píxel marcado en azul es el resultado de la combinación lineal de h con el *parche* verde en la imagen original.

$$g(i,j) = \sum_{k=-m}^{k=m} \sum_{l=-m}^{l=m} f(i+k, j+l) h(k,l) \tag{35}$$

Donde las entradas en el *kernel* o *máscara* de pesos $h(k,l)$ son los *coeficientes del filtro*. El operador en la ecuación Ecuación 35 se denonima *correlación*, y puede ser escrito como:

$$g = f \bigotimes h \tag{36}$$

Una variante de la Ecuación 35 es invirtiendo el signo de los *offsets*. Eso equivale a reflejar el filtro en ambas dimensiones y se denomina *convolución*:

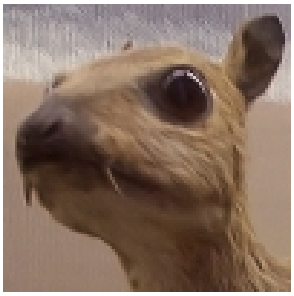
$$g(i,j) = \sum_{k=-m}^{k=m} \sum_{l=-m}^{l=m} f(i-k, j-l) h(k,l) \tag{37}$$

Que también puede ser escrito como:

$$g = f * h \tag{38}$$

En la Figura 12 se puede observar un ejemplo sobre el uso de este operador en una imagen de un solo canal donde la intensidad de cada píxel está representada con un número.

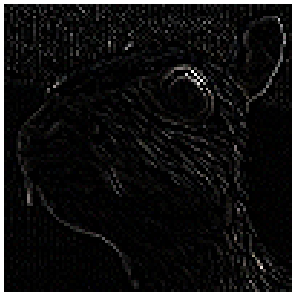
Dado que la convolución en el caso discreto es una combinación lineal, su resultado para cada elemento $f(i,j) \in \mathbb{R}^n$ es un escalar en $\mathbb{R}$. Notar que la Ecuación 37 puede ser generalizada a más dimensiones, por lo que el mismo concepto aplica también a imágenes con n canales.



(a) A

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

(b) *Kernel* K



(c) $A * K$

Figura 13: Es muy común convolucionar imágenes con ciertos *kernels* para detectar bordes en las mismas.

Si convolucionamos todos los píxeles de la imagen de entrada obtendremos una nueva imagen de un canal. Para aplicar la convolución a la totalidad de una imagen se deben tener en cuenta ciertas configuraciones espaciales. Una de ellas es el *stride*, o cantidad de *pasos* cada cuales convolucionar. Cuando se aplica el filtro convolucional a cada píxel de la imagen, se tiene un *stride* igual a 1, pero también podría elegirse aplicar el filtro cada dos o más píxeles. Además, al aplicar el filtro convolucional a lo largo y alto de la imagen surgen casos en los que el filtro se aplica a píxeles fuera de los bordes de la misma, por lo que se agrega un borde artificial mediante la técnicas de *padding*.

Una buena propiedad de la convolución es que toma ventaja de la *localidad* (vecinos) de cada píxel, que usualmente son los más relacionados, y produce una salida *comprimiendo* esa información local en un escalar. De esta manera se puede utilizar una convolución para extraer *features* de una imagen valiéndose de su estructura espacial. Otra buena propiedad es el principio de *invarianza traslacional*, que básicamente significa que la respuesta del operador no depende explícitamente del lugar en la imagen donde se aplique.

Las redes convolucionales cuentan con los mismos artefactos que las redes convencionales que ya discutimos (neuronas con pesos, funciones de pérdida, capas completamente conectadas). Incluso los mismos métodos de entrenamiento pueden ser aplicados. La diferencia radica en que las redes convolucionales asumen que están trabajando con imágenes, lo que permite optimizar la arquitectura de las mismas, reduciendo parámetros y mejorando el proceso de aprendizaje.

Imaginemos por un momento que quisieramos aprender a clasificar un

conjunto de imágenes de 200x200 píxeles con 3 canales de colores. Eso nos da una dimensión de entrada de 200x200x3, por lo que una neurona completamente conectada en la primer capa oculta tendría 120000 conexiones y por ende esa misma cantidad de pesos a entrenar. Si tenemos en cuenta que seguramente vamos a requerir más de una neurona (comúnmente cientos de ellas en una misma capa) podemos concluir que este enfoque no escala bien para el procesamiento de imágenes.

Una red neuronal convolucional utiliza capas con filtros convolucionales. Además de las ventajas de invarianza traslacional y localidad antes mencionadas, un filtro convolucional utiliza pocos parámetros.

Tal como hicimos con las redes neuronales convencionales, en la siguiente sección analizaremos los tipos de capas de una red neuronal convolucional. Al final de la Sección se dará una descripción general de la arquitectura de una red convolucional y varios ejemplos de redes convolucionales conocidas.

2.5.1. Capas de una red convolucional

2.5.1.1. Capas de Entrada y Salida Las redes convolucionales manipulan arreglos tridimensionales. Eso significa que para cada capa hay un volumen de entrada y un volumen de salida.

La capa de entrada de una red es la que provee la imagen original como un volumen de pixeles. Notar que si una imagen tiene un alto y ancho de 256 pixeles y tres canales de colores (RGB) entonces el volumen de salida de esta capa va a ser $256 \times 256 \times 3$.

En general la capa de salida de una red suele ser una capa completamente conectada con los puntajes de cada clase en el caso de tareas de clasificacion o con un vector de numeros reales en el caso de tareas de regresion. Para clasificación también es muy común utilizar un clasificador (por ejemplo, *Softmax*) que transforme los puntajes obtenidos en probabilidades normalizadas.

2.5.1.2. Capas Convolucionales Una capa convolucional consta de un conjunto de filtros (o *kernels*) cuyos parámetros se pueden aprender. En general cada filtro es pequeño a lo ancho y alto, pero se aplica a toda la profundidad del volumen de entrada (ej.: los tres canales RGB). Notar que el volumen de entrada puede bien ser una imagen o bien el volumen de salida con las activaciones de otra capa.

Como se puede observar en la Figura 14 estos filtros se convolucionan a traves del ancho y alto del volumen de entrada, produciendo un mapa de activaciones en 2-D para cada filtro. Si "apilamos" los mapas de activaciones de todos los filtros de una capa convolucional, obtenemos un *volumen* de

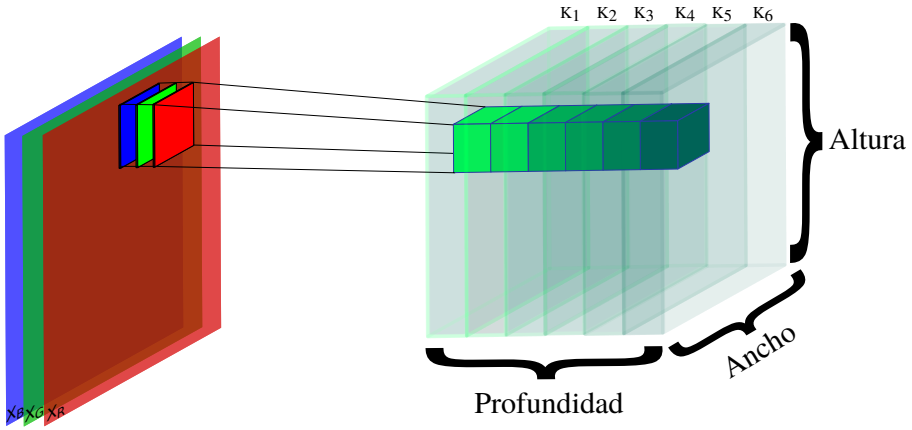


Figura 14: Ejemplo de una capa convolucional cuyo volumen de entrada es una imagen (x) con 3 canales RGB. La capa cuenta con 6 filtros y cada uno convoluciona con la imagen a lo largo y ancho.

salida. De esta manera cada elemento en el volumen de salida puede ser interpretado como la salida de una neurona conectada a una pequeña region de los datos de entrada, la cual comparte parámetros (pesos) con las otras neuronas que corresponden al mismo filtro.

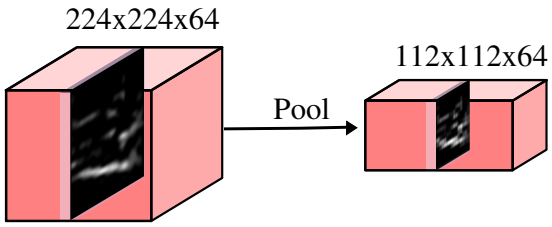
Esta conexión a una pequeña región en los datos de entrada es denominada *campo receptivo*. Es importante notar que los *campos receptivos* son locales en una pequeña área en cuanto al ancho y alto de la entrada, pero abarcan la totalidad de la profundidad del volumen de entrada.

Otros hiperparámetros relacionados con las capas convolucionales son la cantidad de filtros (K), el espacio en píxeles entre cada aplicacion de los filtros (*stride*) y por ultimo el relleno con ceros o *zero-padding*, donde se le agrega un "marco" de 1 o más ceros a la entrada de la capa.

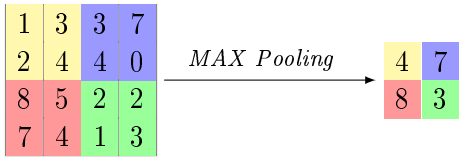
2.5.1.3. *Pooling* Con el objetivo de reducir la cantidad de parámetros y en consecuencia las dimensiones de las representaciones obtenidas de la imagen de entrada, se suelen intercalar capas *pooling* en reducen la dimensión espacial de sus entradas.

Reducir la cantidad de parámetros no sólo favorece la rapidez del entrenamiento sino también ayuda a controlar el *overfitting*. Lo más común es insertar capas de *pooling* luego de capas convolucionales.

Un método de *pooling* muy usado es *MAX Pooling*, en el cual se calcula el máximo de un área local (generalmente 2×2 ó 3×3) en el ancho y



(a)



(b)

Figura 15: Las capas de *Pooling* reducen las dimensiones espaciales. Notar en 15a que la profundidad del volumen se mantiene intacta. En 15b observamos cómo funciona una capa *MAX Pooling* con un un *stride* de 2.

largo del volumen de entrada y a través de cada una de las "rodajas" que conforman la profundidad del volumen, como se puede ver en la Figura 15b. El área local está definida por el tamaño del *stride*. De esta manera se reduce espacialmente la entrada, pero no su profundidad.

2.5.1.4. Capas Completamente Conectadas (Fully-Connected) Como su nombre lo indica, cada neurona de esta capa tiene conexiones a todas las salidas (o activaciones) de la capa anterior. Por lo tanto sus activaciones se pueden calcular con una multiplicacion de matrices junto con el calculo del *bias*, como ya se vio para las redes neuronales convencionales.

2.5.2. Arquitecturas conocidas de redes convolucionales

Normalmente una red convolucional esta compuesta de varias capas convolucionales (CONV), capas de *pooling* (POOL), capas completamente conectadas (FC por sus siglas en ingles) y funciones de activacion, generalmente rectificadores lineales (RELU).

El patrón usual en redes convolucionales es generar *bloques* con una o más capas CONV seguidas de capas RELU seguidas de capas de *pooling*. Luego es común utilizar capas FC hasta reducir las dimensiones a las dimensiones de salida, que en el caso de clasificacion son las probabilidades de cada clase.

A lo largo de los años ha habido varias arquitecturas de redes convolucionales que cuentan con nombre propio, como por ejemplo LeNet [27], creada en los 90 por Yann LeCun y utilizada para el reconocimiento de dígitos manuscriptos. Esta red fue utilizada con éxito para leer códigos postales y cheques bancarios.

En 2012, el ganador del desafío ImageNet ILSVRC, Alex Krizhevsky, obtuvo un 16 % de error utilizando una arquitectura llamada AlexNet [25]. Su arquitectura es muy similar a la de LeNet, aunque mas profunda y fue una de las primeras en concatenar varias capas CONV antes de una capa de *pooling*. El desafío ImageNet consiste en varias tareas de procesamiento de imágenes, entre ellas la clasificación de 1000 clases distintas de objetos.

Los ganadores del ISLVRC 2013 utilizaron una red llamada ZFNet [54] con un error del 11.2 % en ImageNet. Su propuesta era básicamente una modificación en los hiperparámetros y capas convolucionales de AlexNet.

En la misma competencia ILSVRC del 2014 se dieron a conocer dos redes, GoogLeNet [46] con un error del 6.67 % y VGGNet [44] con un error del 7.32 %. Ambas demostraron que la profundidad de la red es una característica crítica a la hora de obtener buenos resultados. Si bien GoogLeNet fue la ganadora ese año, luego se demostró que VGGNet es superior en muchas tareas de transferencia de aprendizaje, por lo que es más popular que GoogLeNet y se pueden encontrar muchos modelos ya preentrenados.

Finalmente, ResNet (Residual Network) [18], la red ganadora del ILSRVC 2015, cuenta con nada menos que 152 capas (VGGNet cuenta con 19) y obteniendo un error de 3.57 % en el top-5. A finales de 2016 se publicó una nueva variante de ResNet denominada DenseNet [20] que incluye conexiones entre todas las capas de la misma, obteniendo resultados de estado del arte pero consumiendo menos memoria y tiempo de cómputo.

Capítulo 3

3. Entrenamiento mediante tareas pretexto

3.1. Introducción

Basándonos en los conceptos previamente explicados, analizaremos en esta Sección una alternativa mediante entrenamiento no supervisado al problema de clasificación con conjuntos de datos con muchas categorías pero pocos elementos por categoría. Dicho método es el entrenamiento mediante *arquitecturas siamesas*. Se presentan motivaciones y fundaciones teóricas de los modelos siameses asi como también las distintas formas de definir funciones de costos para este paradigma, centrándonos principalmente en tareas de pretexto. Se desarrollarán además las dos funciones objetivo utilizadas en los experimentos del Capítulo 4: *Slow Feature Analysis* y *Automovimiento*.

3.2. Tareas de pretexto y autosupervisión

Los métodos discriminativos tradicionales generalmente requieren conocer las categorías del conjunto de datos de antemano y suelen estar limitados a un número reducido de categorías (entre 100 y 1000). Eso se convierte en una limitación para ciertas tareas donde la cantidad de clases es muy grande, como la verificación de rostros o motores de búsqueda visuales, donde se cuenta con pocos elementos por clase o sólo algunas clases son conocidas durante el entrenamiento. Generalizar bien para todas las clases se convierte entonces en una tarea muy cara (es necesario contar con una cantidad aceptable de ejemplos por clase) o prácticamente imposible.

A pesar de estas limitaciones todavía es posible aprender buenas representaciones mediante lo que se denomina tareas de pretexto. Las tareas de pretexto no son útiles en si mismas, pero sirven para aprender representaciones generales que puedan ser aplicadas a otro problema. Ejemplos de tareas de pretexto en el contexto del aprendizaje automático aplicado a imágenes incluyen reconstruir la imagen de entrada [19], predecir píxeles en una secuencia de video [50], ordenar temporalmente cuadros de videos [31] y ordenar parches en imágenes estáticas [35]. Las etiquetas para tareas de pretexto son fáciles de obtener e incluso de elaborar a partir del conjunto de datos. El desafío consta de diseñar una tarea de pretexto que permita aprender representaciones generales tomando en cuenta las características que se desea extraer de las imágenes.

De esta manera, las tareas de pretexto ofrecen un método alternativo

de aprendizaje supervisado. Más aún, es posible generar etiquetas para las tareas de pretexto algorítmicamente y así evitar anotar manualmente los ejemplos. Noroozi y Favaro [35] crean rompecabezas de 3×3 a partir de imágenes y fuerzan a su modelo a aprender el orden de las piezas. Otras tareas autosupervisadas incluyen predecir colores a partir de la luminancia [26] o predecir sonidos en secuencias de videos [38].

Un factor común a todos estos ejemplos es la hipótesis de que el modelo aprenderá características de alto nivel para poder realizar su tarea.

3.3. Arquitecturas siamesas

Cuando se cuenta con muchas clases y pocos elementos de entrenamiento por cada una se suele resolver el problema de la clasificación mediante métodos basados en distancias. Dichos métodos consisten en computar una métrica de similaridad entre las representaciones y un conjunto de prototipos cuyas representaciones hayan sido previamente computadas. Para aplicar esta técnica es necesario contar con un método para obtener buenas representaciones sobre el dominio del problema sin necesidad de contar con la información de las categorías.

Se debe buscar entonces una función que para elementos semánticamente similares en el espacio de entrada produzca representaciones que también sean similares en el espacio de salida. Esta similaridad entre las representaciones puede obtenerse de acuerdo a alguna métrica de distancia (distancia euclidea por ejemplo).

Sea entonces $G_W(x)$ una familia de funciones con parámetros W donde x pertenece al espacio de entrada, buscamos encontrar W tal que para una medida de similitud D, $D(G_W(x_1), G_W(x_2))$ sea pequeña si x_1 y x_2 pertenecen a la misma categoría y grande en caso contrario. Una manera de optimizar los parámetros W es estableciendo una función de costo que minimice $D(G_W(x_1), G_W(x_2))$ tomando en cuenta los casos recien descriptos.

Notemos que la función G_W no debe cumplir ninguna condición en particular más allá de que sea derivable (o su derivada pueda ser aproximada) en W.

Si creamos un modelo que tome dos entradas y optimice $D(G_W(x_1), G_W(x_2))$, lo que nos queda es una *arquitectura siamesa*. En nuestro dominio particular, G_W son redes convolucionales. Su elección radica en que son capaces de generar buenas representaciones de imágenes a pesar de la gran variación en color y geometría de las mismas.

Una red neuronal convolucional siamesa entonces consiste de dos redes idénticas cuyos parámetros W son compartidos. La arquitectura toma un

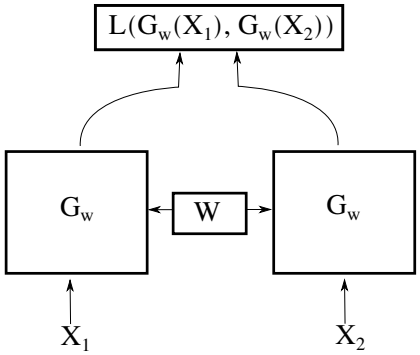


Figura 16: Esquema de la arquitectura de una red siamesa. G_W es un conjunto de funciones, en este caso una red convolucional, con parámetros W. Ambas redes comparten los parámetros. Luego una función de pérdida L analiza la similitud entre X_1 y X_2

par de imágenes (x_1, x_2) y computa si ambas pertenecen o no a la misma categoría o si son similares de acuerdo a algún criterio. Para ello, las representaciones obtenidas de ambas redes son redireccionadas a una función de costo que calcula alguna métrica respecto a las entradas (Figura 16). Observemos que esto puede incluso ser establecido como una tarea de pretexto en el cual la función objetivo puede no ser la misma que la del dominio final del problema.

Las redes siamesas fueron utilizadas por primera vez con éxito para la verificación de huellas digitales [2] y de firmas [5]. Se han logrado también buenos resultados en verificación de rostros [7] y más recientemente en tareas de clasificación [24] y recomendación de productos [4].

3.3.1. Funciones de costo aplicadas a redes siamesas

Qué función de costo utilizar y cómo generar los pares de entrenamiento para redes siamesas son opciones que dependen del dominio del problema.

Para funciones de costo suele considerarse la penalización sobre alguna medida de similitud entre las representaciones aprendidas por la red. En ese caso los pares se arman con ejemplos negativos/positivos (es decir, de imágenes no similares/similares). Una manera de probar que las representaciones obtenidas por la red siamesa pueden generalizar bien a otros dominios es realizar transferencia de aprendizaje de los parámetros aprendidos a un nuevo problema (clasificación, búsqueda de imágenes, detección, etc).

Nuevamente, buscamos minimizar $D(G_W(x_1), G_W(x_2))$ para los elementos que sean semánticamente similares en el espacio de entrada (*pares genui-*

nos). Pero además queremos que los elementos que no son similares queden separados en nuestro espacio de representación. Por consiguiente, es necesario que la función de costo tenga un término de *contraste* que asegure que no sólo se minimicen las distancias de los pares similares sino que se aumenten las de los pares distantes semánticamente (*pares impostores*) [7]. Asignemos una etiqueta y a cada par, donde $y = 1$ para los pares impostores e $y = 0$ para los pares genuinos. Definimos entonces una *función de costo de contraste* (*contrastive loss* en inglés) para un par x_1, x_2 de la siguiente manera:

$$L(W, (y, x_1, x_2)) = (1 - y)L_{gen}(G_W(x_1), G_W(x_2)) + yL_{imp}(G_W(x_1), G_W(x_2)) \tag{39}$$

Donde L_{gen} es la función de costo parcial para un par genuino y L_{imp} su correspondiente para un par impostor.

3.3.2. Otros enfoques para entrenar redes siamesas

Si bien mencionamos que en las redes siamesas se suelen comparar las representaciones generadas utilizando métricas de similitud, también podemos definir nuestra función de costo de diferente manera. Se podría, por ejemplo, intentar predecir cuando dos parches (secciones de una imagen) pertenecen o no a la misma imagen, o cuando dos objetos distintos pertenecen a la misma clase, o cuando dos fotos son la misma a pesar haber sufrido alguna transformación (traslacion, rotación, etc). Todos estos enfoques pueden ser pensados como tareas de clasificación donde el objetivo es predecir la categoría de similitud a la que pertenece cada par.

En el caso de la predicción de las transformaciones en la imagen como la traslación, la información sobre el cambio de posición respecto a un punto inicial puede ser obtenida mediante sensores (giróscopos, acelerómetros, etc) y se denomina *información odométrica*.

La información odométrica es ampliamente utilizada en robótica para estimar la posición relativa de un agente respecto a una posición inicial. Por otro lado, agentes biológicos también utilizan su sistema perceptual para ubicarse en el ambiente y poder lograr sus objetivos. Cabe preguntarse si toda la información que el agente obtiene mediante su movimiento por el ambiente (*automovimiento*) puede ser utilizada por el mismo como una fuente de supervisión en la tarea de aprender representaciones perceptuales.

Nos enfoquemos en la tarea de percepción visual. En este caso, la pregunta se puede reformular de la siguiente manera: ¿Puede un agente móvil

valerse de la información obtenida de su propio movimiento para supervisar el aprendizaje de buenas representaciones visuales de su entorno?

Cuando hablamos de buenas representaciones hacemos hincapié en dos aspectos: (1) la capacidad de poder realizar múltiples tareas visuales y (2) la habilidad de poder aprender nuevas tareas visuales basándose en pocos ejemplos.

Agrawal et al. [1] proponen que relacionando los estímulos visuales con la información de automovimiento pueden aprenderse buenas representaciones visuales. De esa manera el agente móvil puede ser visto como una cámara que se mueve por el ambiente y la información del automovimiento es entonces la misma que la del movimiento de la cámara. Bajo este enfoque, **el problema de relacionar los estímulos visuales con el automovimiento puede ser establecido como el problema de predecir las transformaciones de la cámara en pares de imágenes a lo largo del movimiento de la misma**.

Luego, el problema puede ser implementado con una arquitectura de redes convolucionales siamesas cuya tarea sea la de predecir las transformaciones entre dos imágenes de entrada. En Ref. [1] se prueba para algunos casos que las representaciones obtenidas al entrenar la red siamesa de manera no supervisada mediante la información de automovimiento son competitivas con las del estado del arte que utilizan métodos supervisados de clasificación cuando son entrenadas con pocas imágenes por categoría.

En la siguiente Sección explicaremos y evaluaremos diferentes métodos de entrenamiento mediante información odométrica.

3.4. Entrenamiento de modelos de aprendizaje profundo utilizando información de automovimiento

Queremos evaluar que nuestro algoritmo no supervisado de automovimiento funciona mejor que un entrenamiento supervisado cuando se cuenta con pocas imágenes por categoría.

Para evaluar la calidad de las representaciones utilizaremos el paradigma de transferencia de aprendizaje. Esto es, entrenaremos las arquitecturas siamesas y luego utilizaremos los parámetros aprendidos para entrenar un modelo sobre un nuevo dominio de problema.

Además, compararemos las representaciones mediante automovimiento con otra técnica no supervisada utilizada en arquitecturas siamesas, *Slow Feature Analysis*. Se concluye eventualmente que la técnica de automovimiento es superior a *Slow Feature Analysis*.

En lo que resta del Capítulo se presentarán las dos técnicas de entrenamiento no supervisado que se utilizarán en los experimentos en 4.

3.4.1. Aprendizaje por *Slow Feature Analysis*

Para analizar que tan bueno es aprender representaciones mediante automovimiento compararemos la calidad de las representaciones obtenidas con ese método contra las representaciones obtenidas mediante un tipo de función de contraste denominado *Slow Feature Analysis* (SFA).

SFA es una técnica de aprendizaje no supervisado que se basa en que las características relevantes (*features*) cambian poco en una ventana de tiempo pequeña. Aplicado a nuestro dominio donde un agente se mueve a través de su entorno, sean x_{t_1} y x_{t_2} imágenes tomadas en tiempos t_1 y t_2 respectivamente. Computamos su similitud mediante alguna métrica D. Si las imágenes están cercanas en el tiempo - es decir $|t_1 - t_2| \leq T$ para algún intervalo de tiempo T - son consideradas similares y su distancia debería ser pequeña, caso contrario son consideradas diferentes y su distancia debería ser mayor a algun margen preestablecido. Podemos entonces establecer una función de costo con el objetivo de penalizar las representaciones que son parecidas pero estan lejanas en el espacio de representación y a su vez penalizar las que son distintas pero se encuentran cercanas en ese mismo espacio. Definimos entonces a la función de costo SFA como:

$$L(x_{t_1}, x_{t_2}, W) = \begin{cases} D(f(x_{t_1}, W), f(x_{t_2}, W), & \text{si} |t_1 - t_2| \leq T \\ \max(0, m - D(f(x_{t_1}, W), f(x_{t_2}, W))), & \text{si} |t_1 - t_2| > T \end{cases} \tag{40}$$

Donde W son los parámetros compartidos de las redes siamesas y m es el margen mínimo que debe separar a las representaciones distintas. En los experimentos realizados la distancia D se eligió como la norma L2 de las representaciones.

3.4.2. Aprendizaje por clasificación de transformaciones en pares de imágenes

Analizar el automovimiento se plantea como una tarea de clasificación en donde las redes tienen que predecir qué transformación fue efectuada en cada uno de los ejes del ambiente del agente (X, Y, Z).

Sin embargo, estas transformaciones viven en un espacio continuo, por lo que se discretizan las traslaciones y rotaciones por rangos de valores y se

utiliza 3 clasificadores, uno para cada eje. Cada clasificador tiene un número determinado de categorías correspondientes al resultado de discretizar las transformaciones antes mencionadas.

Para ilustrar esto, si un agente se mueve entre los valores L y U (medidos en metros o píxeles por ejemplo) con respecto al eje X y un punto de origen establecido, podemos discretizar todas las posibles transformaciones en N clases distintas donde cada clase comprende un rango de números reales $(U - L + 1)/N$.

Luego, cada clasificador computa su función de costo independiente y el costo total de un par de imágenes se obtiene sumando los costos de las predicciones para cada eje.

Capítulo 4

4. Experimentos

4.1. Introducción

En esta sección se presentan todos los resultados experimentales realizados sobre arquitecturas siamesas con información odométrica.

Todos los experimentos realizados se encuentran en el repositorio GitHub del autor [49]. Dentro del repositorio se encuentra toda la documentación pertinente a la descarga de los conjuntos de datos y la reproducción de los experimentos. Parte del tiempo fue dedicado a tratar de que los experimentos sean reproducibles por otras personas, por lo que se trató de seguir buenas prácticas a la hora de modularizar y organizar los el código.

Los experimentos fueron desarrollados en su mayoría en el lenguaje Python, excepto por algunos scripts de preprocesamiento que originalmente fueron creados en C++. Se presentará primero una prueba de concepto con el conjunto de datos MNIST [33] para luego mostrar los resultados sobre los conjuntos de datos KITTI [14], SUN397 [51] e ILSVRC 2012 [22].

4.2. Prueba de concepto con MNIST

4.2.1. Conjunto de datos

El conjunto de datos MNIST [33] cuenta con 60000 imágenes de caracteres numéricos manuscritos para entrenamiento, más 10000 imágenes para evaluación. La dimensión de cada imagen es $1 \times 28 \times 28$. Para el entrenamiento mediante *automovimiento* se crearon pares de imágenes compuestos de la imagen original y la imagen con transformaciones en los ejes X, Y, Z. Las transformaciones en X e Y son traslaciones de 3 píxeles, mientras que la rotación en Z varía entre los -30° y los 30°. Tanto las rotaciones como las traslaciones son números enteros. Para cada par creado las transformaciones se eligen de manera aleatoria uniforme.

Para SFA, se consideran *temporalmente cercanos* a los pares cuyas transformaciones en X e Y estuvieran en el rango de -1 y 1 píxeles y en Z estuvieran entre -3° y 3°.

En la Figura 17 se pueden observar varios pares de imágenes generadas durante la creación de la base de datos que luego fue usada para entrenar la red siamesa.

Figura 17: Ejemplo de transformaciones aplicadas a las imagenes de MNIST. Cada par contiene la imagen original y la transformada con traslaciones en los ejes X e Y, y con rotación en Z.

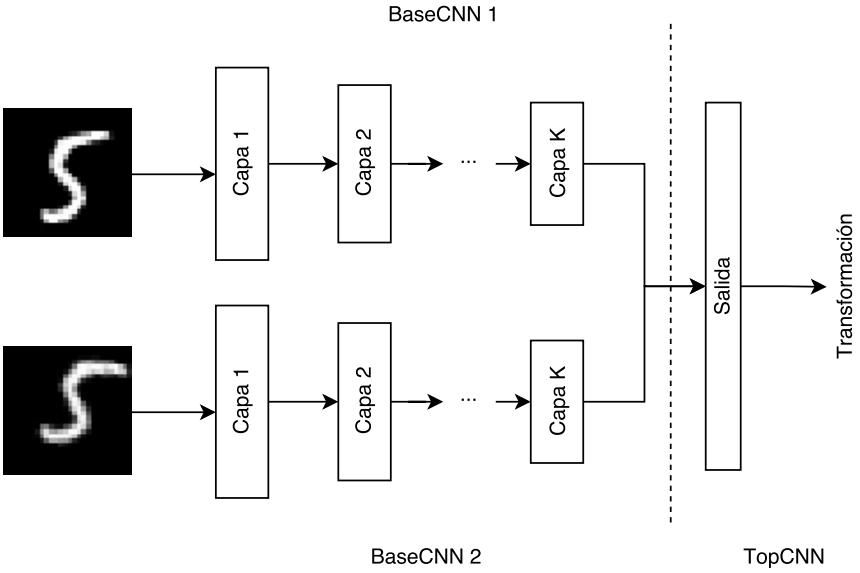


Figura 18: Esquema de una red siamesa que tiene como objetivo aprender las transformaciones en los caracteres de MNIST.

4.2.2. Nomenclatura

Cada componente de la red siamesa se denotará con BCNN por sus siglas en inglés *Base CNN*. Las *features* extraídas de estas redes serán concatenadas y pasadas a otra red llamada TCNN (*Top CNN*). En la TCNN es donde colocaremos nuestras funciones de costo. Las BCNN son las redes sobre las que luego haremos transferencia de aprendizaje. Se puede observar un esquema de la arquitectura en la Figura 18.

Para dar una idea de las arquitecturas de cada red de manera sencilla vamos a usar las siguientes abreviaciones:

- Ck para una capa convolucional con k filtros cuadrados.

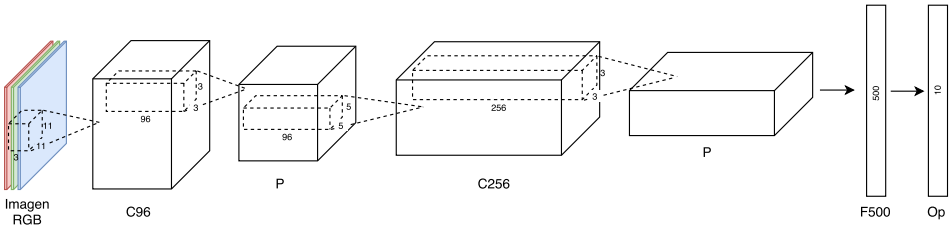


Figura 19: Ejemplo de una arquitectura de red convolucional definida como C96-P-C256-F500-Op, donde la cantidad de clases de salida es 10. Recordar que luego de cada capa convolucional se utilizan unidades ReLU y luego de cada capa completamente conectada se agrega una capa de *Dropout*.

- Fk para una capa completamente conectada (FC) con salida de dimensión k.
- P para una capa de *Pooling*. A menos que se diga lo contrario, siempre usaremos *MAX Pooling*.
- D para una capa *Dropout*.
- Op para la capa de salida. En general nuestras capas de salida van a estar conformadas por una Fk (k es el número de clases) seguidas por una capa Softmax.

Se agregan además rectificadores lineales ReLU luego de cada capa convolucional y cada cada FC.

A modo de ejemplo, en la Figura 19 podemos ver una red definida como C96-P-C256-P-F500-F10.

4.2.3. Descripción de la red

La arquitectura utilizada para las BCNN fue C96-P-C256-P, y para la TCNN se eligió F1000-D-Op. Tener en cuenta que para el caso del *automovimiento* es necesario utilizar una combinación FC-Softmax para calcular la pérdida en cada una de las transformaciones.

Para transferencia de aprendizaje se añadió F500-D-F10-Softmax a una BCNN.

4.2.4. Entrenamiento y evaluación

Las redes siamesas se pre-entrenaron durante 40000 iteraciones con una tasa de aprendizaje de 0.01. Se utilizaron márgenes m de 10 y 100 para

SFA por ser los que mejores resultados lograron. En ambas redes la tasa de aprendizaje se reduce a la mitad cada 10000 iteraciones. El tamaño del *mini batch* fue de 125, lo cual equivale a procesar 5 millones de pares de imágenes durante las 40000 iteraciones del entrenamiento.

La etapa de transferencia de aprendizaje se hizo con 4000 iteraciones a una tasa de aprendizaje constante de 0.01.

Durante la transferencia de aprendizaje se estableció a cero la tasa de aprendizaje de las capas convolucionales. De esa manera se evita que se modifiquen los parámetros aprendidos mediante automovimiento y así poder evaluar la caliidad de las representaciones mediante un clasificador.

En el Cuadro 1 se puede observar la exactitud obtenida mediante la transferencia de aprendizaje con 100, 300, 1000 y 10000 imágenes de los dos métodos utilizados (automovimiento y SFA) y una comparación con un entrenamiento desde cero utilizando esa misma cantidad de imágenes.

	datos entrenamiento			
Método	100	300	1000	10000
Desde cero	42	70	82	97
SFA(m=10)	52	71	77	82
SFA(m=100)	58	73	80	88
Automovimiento	75	90	92	99

Cuadro 1: Exactitud porcentual de los dos métodos de pre-entrenamiento utilizados (SFA y automovimiento) de acuerdo a la cantidad de elementos de entrenamiento: 100, 300, 1000 y 10000.

Se puede observar que entrenar mediante automovimiento presenta una performance claramente superior a entrenar una red desde cero con la misma cantidad de imágenes en los casos en los que el conjunto de datos es relativamente pequeño. Es también superior al entrenamiento utilizando *Slow Feature Analysis*, y dado que no se modificaron los pesos de las capas convolucionales aprendidas durante el pre-entrenamiento, podemos concluir que las *features* aprendidas son buenas y logran captar las representaciones necesarias para el domino del problema en cuestión. El siguiente paso es verificar que efectivamente las *features* aprendidas se puedan aplicar a diferentes dominios de problemas y sean lo suficientemente generalizables.

Figura 20: Ejemplo de imágenes extraídos de KITTI. Las transformaciones en X, Y y Z se obtuvieron de las anotaciones provistas por los creadores del conjunto de datos. Para las rotaciones en el eje Y se calculó el ángulo de Euler correspondiente al cambio entre dos *frames*.

4.3. Pruebas con KITTI

4.3.1. Conjunto de datos

El conjunto de datos KITTI [14] consiste en 11 secuencias que registran el movimiento de un automóvil en una ciudad. Además de proveer cuadros de video, se encuentra la información odométrica recolectada por sensores montados en el automóvil. Esa misma información es la que usa Agrawal et al. [1] a la hora de computar las transformaciones en la cámara entre pares de imágenes, y es la que intentaremos reproducir en esta sección.

Se asume que la dirección a la que apunta la cámara es el eje Z y el plano de la imagen es el plano XY (ejes horizontales y verticales). Dado que las transformaciones más significativas de la cámara ocurren en los ejes Z/X (a medida que el automóvil avanza por la calle) y sobre el eje Y (cuando el automóvil gira), sólo se tomaron en cuenta esas tres dimensiones a la hora de analizar las transformaciones.

Nuevamente, la predicción de transformaciones se establece como una tarea de clasificación, esta vez con 20 clases para las transformaciones en cada eje. Siguiendo los lineamientos originales del paper, los pares de entrenamiento se tomaron de cuadros separados a lo sumo por 7 cuadros intermedios. Similarmente, para entrenamiento por SFA se consideraron a los cuadros separados por ± 7 cuadros intermedios como similares.

Finalmente, las redes siamesas fueron entrenadas a partir de parches de 227×227 extraídos aleatoriamente de las imágenes originales de 1241×376 píxeles. No se aplicaron transformaciones extras más allá de las otorgadas por el movimiento de la cámara.

En la Figura 20 se pueden observar pares de imágenes utilizados durante el entrenamiento de las redes siamesas.

4.3.2. Odometria

KITTI provee anotaciones con las poses de la camara (es decir, su trayectoria) para las secuencias de imágenes. Estas poses estan dadas por una matriz de transformación de 3×4, siendo la última columna las traslaciones en X, Y, Z. De esa matriz se puede extraer el ángulo de Euler correspondiente a las rotaciones sobre el eje Y si asumimos que el primer bloque 3×3 es una matriz de rotación R.

4.3.3. Descripción de la red

La red utilizada como base de las BCNN está inspirada en las primeras 5 capas convolucionales de AlexNet [25], es decir, C96-P-C256-P-C384-C384-C256-P. La TCNN fue definida como C256-C128-F500-D-Op, con filtros convolucionales de 3×3.

4.3.4. Entrenamiento y evaluación

Se pre-entrenaron las redes siamesas por 60K iteraciones con un tamaño de *mini batch* de 125 y una tasa de aprendizaje inicial de 0,001, reducida en un factor de 2 cada 20K iteraciones. La transferencia de aprendizaje se hizo durante 10K iteraciones a una tasa de aprendizaje constante de 0.001. Para diferenciar los distintos entrenamientos, al modelo entrenado con SFA lo vamos a llamar KITTI-SFA y al entrenado con automovimiento, KITTI-EGO.

Para tener un baseline adecuado, se entrenó AlexNet con el conjunto de datos ILSVRC'12 desde cero utilizando 20 y 1000 imágenes por clase. El conjunto de datos ILSVRC es el usado en la competencia anual de Imagenet y contiene mil clases de objetos distintas. Dichos modelos seran llamados ALEX-20 y ALEX-1000 respectivamente.

Para hacer una comparación justa con ALEX-20 y ALEX-1000, las redes siamesas fueron entrenadas con aproximadamente 20K pares de imágenes.

4.3.4.1. Evaluación utilizando el conjunto de datos SUN-397 El conjunto de datos SUN-397 [51] consiste de 397 categorías de paisajes interiores y exteriores y además provee 10 particiones del dataset para hacer *cross-validation*, pero debido a lo costoso que es entrenar redes neuronales convolucionales solo se utilizaron tres particiones. En la Figura 21 se pueden observar algunas de las clases que provee este conjunto de datos.

La evaluación se hizo midiendo la exactitud de clasificadores *Softmax* utilizando las *features* obtenidas de las salidas de las primeras 5 capas con-

Figura 21: Ejemplo de clases del conjunto de datos SUN-397: paisajes exteriores, interiores, construcciones, etc.

volucionales (nombradas L1-L5). Los resultados pueden verse en el Cuadro 2. En todos los casos de estudio, a medida que se utilizan más capas para extraer representaciones se observa que aumenta la exactitud de los modelos. Esto es porque la capacidad de las redes de aprender representaciones de las imágenes a varios niveles de abstracción aumenta mientras más capas y no linealidades contiene [54]. Como consecuencia, una red que sólo utilice representaciones de la primer capa convolucional será muy buena en la detección de características de bajo nivel (colores, bordes) mientras que una red que extraiga representaciones de la última capa convolucional será capaz de detectar variaciones más específicas a las clases de alto nivel con las que se haya entrenado.

Del Cuadro 2 también concluimos que el pre-entrenamiento con automoviento no supera la performance que se obtiene al pre-entrenar la misma red con ILSVRC'12 con 1000 imagenes por clase. Sin embargo, sí equipara la performance de pre-entrenar la red con 20 imágenes por clase en ILSVRC'12. Además supera la performance de la red siamesa entrenada con SFA. Esto nos indica que cuando se tiene un conjunto acotado de datos de entrenamiento, se puede lograr entrenar un modelo que logre resultados similares al estado del arte si utilizamos redes siamesas entrenadas con automovimiento.

Método	#preentr.	#finet.	L1	L2	L3	L4	L5	#finet.	L1	L2	L3	L4	L5
ALEX-1000	1M	5	3.73	5.07	5.07	8.53	10.40	20	9.07	12.53	16.27	17.60	10.67
ALEX-20	20K	5	2.93	1.87	3.73	5.07	3.20	20	6.13	5.33	5.33	4.53	5.07
KITTI-SFA	20.7K	5	2.13	3.20	2.40	1.60	1.87	20	4.53	3.73	2.13	2.40	2.93
KITTI-EGO	20.7K	5	2.93	1.87	3.20	5.87	1.33	20	6.67	7.47	9.87	9.33	4.00

Cuadro 2: Exactitud de pre-entramiento con Egomotion y SFA comparado con un entrenamiendo de cero de AlexNet utilizando el conjunto de datos ILSVRC'12. L1-L5 significa que se agregó un clasificador a la salida de las capas convolucionales 1 a 5. Se reporta la exactitud de cada clasificador utilizando el conjunto de datos SUN397. La columna *finet* indica la cantidad de elementos por clase que se utilizaron de SUN397 para realizar la transferencia de aprendizaje de los pesos previamente optimizados.

Cabe destacar que hay un decaimiento en la exactitud reportada para algunas capas consecutivas, en particular para la última capa convolucional (L5) en KITTI-SFA y KITTI-EGO. La hipótesis es que durante el preentrenamiento se generó una co-adaptación frágil [52] entre esa capa convolucional y la siguiente, lo que derivó en representaciones menos precisas a la hora del *finetuning*. Cómo evitar la co-adaptación de capas o si eso es un problema inherente a la arquitectura o método de entrenamiento utilizados escapa al objetivo del experimento.

4.3.4.2. Evaluacion utilizando el conjunto de imagenes Imagenet ILSVRC 2012 Para evaluar que los filtros aprendidos mediante automovimiento son buenos para tareas de clasificación, se procedió a realizar transferencia de aprendizaje en todas las capaz convolucionales (es decir, reentrenar toda la red utilizando un nuevo conjunto de datos).

Se evaluó la exactitud de una red preentrenada con automovimiento contra una entrenada con *SFA* y una con pesos inicializados aleatoriamente. Para ello se utilizó el conjunto de datos ILSVRC 2012 utilizado en la competencia de Imagenet [22]. Dicho conjunto de datos cuenta con 1000 clases de objetos. Ejemplos de clases en este conjunto de datos se pueden ver en la Figura 22.

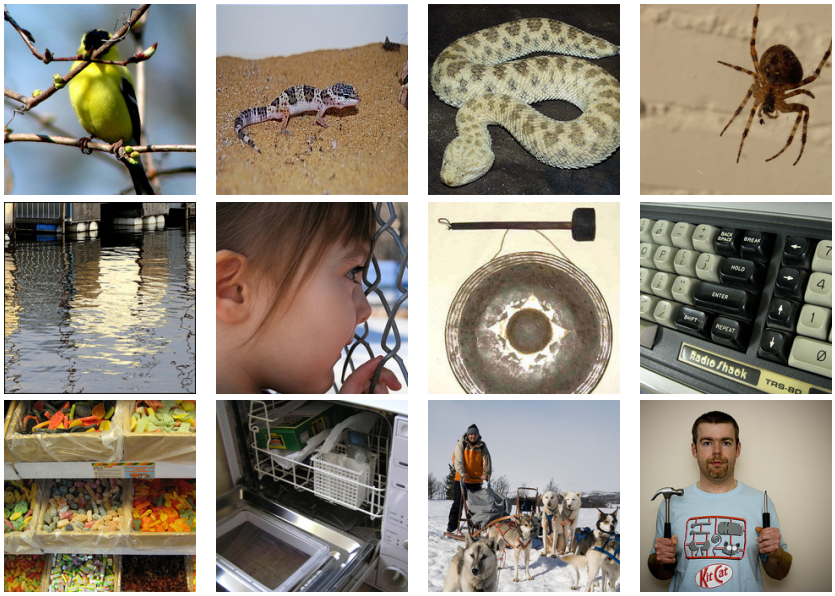


Figura 22: Ejemplo de clases del conjunto de datos ILSVRC'12.

Para el entrenamiento de las redes se utilizaron subconjuntons de todas las clases con 1, 5, 10, 20 y 1000 elementos por cada una. Los resultados se muestran en el Cuadro 3. Se puede observar que los pesos de una red pre-entrenada con automovimiento (KITTI-EGO) supera en todos los casos a los pesos inicializados aleatoriamente (ALEXNET), mientras que los pesos aprendidos mediante *SFA* (KITTI-SFA) presentan un rendimiento incluso peor que el de ALEXNET.

Método	1	5	10	20	1000
KITTI-EGO	0.49	1.27	2.14	4.13	20.8
KITTI-SFA	0.35	0.75	1.34	2.64	11.83
ALEXNET	0.45	0.95	1.91	3.69	18.35

Cuadro 3: Exactitud de los modelos aprendidos mediante redes siamesas (KITTI-EGO corresponde a la red de automovimiento, KITTI-SFA a la que utiliza *slow feature analysis*) contra una red cuyos pesos fueron inicializados aleatoriamente (ALEXNET). Se entrenaron las 3 redes utilizando el conjunto de datos ILSVRC 2012 con 1, 5, 10, 20 y 1000 imágenes por clase.

Capítulo 5

5. Conclusiones

Si bien actualmente se suele recurrir a la transferencia de aprendizaje de redes del estado del arte pre-entrenadas por otras personas (por ejemplo, por grandes empresas como Google, Facebook, o algunas universidades), puede haber casos en los que se requiera entrenar los pesos de una red desde cero. Y como ya se dijo, los conjuntos de datos para tareas específicas suelen ser muy reducidos como para obtener buenos resultados con redes convolucionales profundas.

En este trabajo se propuso reproducir algunos de los resultados de Agrawal et al. [1]. Se logró demostrar la hipótesis principal de ese trabajo en cada uno de los experimentos: que utilizando información de *automovimiento* libremente disponible se puede entrenar una red neuronal convolucional profunda y obtener resultados similares a los del estado del arte. Los experimentos propuestos aquí son todas tareas de clasificación, aunque bien podría extenderse este trabajo a otros problemas.

Se demostró que utilizando unos 20 mil pares de imágenes durante el pre-entrenamiento de una red (mediante redes siamesas) los resultados son equiparables a los que se obtendrían entrenando normalmente utilizando clasificación, lo cual hace al entrenamiento con *automovimiento* un candidato ideal para para los tipos de tareas mencionados.

6. Trabajo a futuro

Hay que destacar que todas las pruebas se hicieron con arquitecturas basadas en AlexNet, que ya resulta anticuada en lo que se refiere al estado del arte, donde las redes se han vuelto más profundas y complejas. Sin embargo puede haber casos en los que se busque una red menos profunda por cuestiones de rendimiento en velocidad, característica de los datos, etc. Queda como trabajo a futuro probar nuevas arquitecturas con la metodología propuesta.

Además sólo se probó la técnica propuesta en Agrawal et al. [1] con pocos conjuntos de datos por cuestión de tiempo (ILSVRC'12, SUN-397, MNIST, KITTI). Probar esta técnica en otros dominios de problemas queda como trabajo a futuro.

Referencias

[1] Agrawal, Pulkit, João Carreira y Jitendra Malik: *Learning to See by Moving*. CoRR, abs/1505.01596, 2015. http://arxiv.org/abs/1505.01596.

[2] Arrieta, Angélica González, Griselda Cobos Estrada, Luis Alonso Romero y Ángel Luis Sánchez Lázaro y. Belén Pérez Lancho: *Neural Networks Applied to Fingerprint Recognition*, páginas 621–625. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ISBN 978-3-642-02481-8. https://doi.org/10.1007/978-3-642-02481-8_91.

[3] Bay, Herbert, Andreas Ess, Tinne Tuytelaars y Luc Van Gool: *Speeded-Up Robust Features (SURF)*. Comput. Vis. Image Underst., 110(3):346–359, Junio 2008, ISSN 1077-3142. http://dx.doi.org/10.1016/j.cviu.2007.09.014.

[4] Bell, Sean y Kavita Bala: *Learning Visual Similarity for Product Design with Convolutional Neural Networks*. ACM Trans. Graph., 34(4):98:1–98:10, Julio 2015, ISSN 0730-0301. http://doi.acm.org/10.1145/2766959.

[5] Bromley, Jane, Isabelle Guyon, Yann LeCun, Eduard Säckinger y Roopak Shah: *Signature Verification Using a "Siamese"Time Delay Neural Network*. En *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, páginas 737–744, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. http://dl.acm.org/citation.cfm?id=2987189.2987282.

[6] Cheng, Jian, Xin Chen y Angeliki Metallinou: *Deep Neural Network Acoustic Models for Spoken Assessment Applications*. Speech Commun., 73(C):14–27, Octubre 2015, ISSN 0167-6393. http://dx.doi.org/10.1016/j.specom.2015.07.006.

[7] Chopra, Sumit, Raia Hadsell y Yann LeCun: *Learning a Similarity Metric Discriminatively, with Application to Face Verification*. En *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, páginas 539–546, Washington, DC, USA, 2005. IEEE Computer Society, ISBN 0-7695-2372-2. http://dx.doi.org/10.1109/CVPR.2005.202.

[8] *The Zettabyte Era—Trends and Analysis*. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html, Visitada en 2016-01-16.

[9] Csurka, Gabriella, Christopher R. Dance, Lixin Fan, Jutta Willamowski y Cédric Bray: *Visual categorization with bags of keypoints.* En *In Workshop on Statistical Learning in Computer Vision, ECCV*, páginas 1–22, 2004.

[10] Dalal, Navneet y Bill Triggs: *Histograms of Oriented Gradients for Human Detection.* En *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, páginas 886–893, Washington, DC, USA, 2005. IEEE Computer Society, ISBN 0-7695-2372-2. http://dx.doi.org/10.1109/CVPR.2005.177.

[11] Dekel, Ofer, Ran Gilad-Bachrach, Ohad Shamir y Lin Xiao: *Optimal Distributed Online Prediction using Mini-Batches.* CoRR, abs/1012.1367, 2010. http://arxiv.org/abs/1012.1367.

[12] Dumont, Marie, Raphaël Marée, Louis Wehenkel y Pierre Geurts: *Fast Multi-Class Image Annotation with Random Subwindows and Multiple Output Randomized Trees.* En *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, volumen 2, páginas 196–203. INSTICC, feb 2009. http://www.montefiore.ulg.ac.be/services/stochastic/pubs/2009/DMWG09.

[13] Flickner, Myron, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele y Peter Yanker: *Query by Image and Video Content: The QBIC System.* Computer, 28(9):23–32, Septiembre 1995, ISSN 0018-9162. https://doi.org/10.1109/2.410146.

[14] Geiger, Andreas: *Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite.* En *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, páginas 3354–3361, Washington, DC, USA, 2012. IEEE Computer Society, ISBN 978-1-4673-1226-4. http://dl.acm.org/citation.cfm?id=2354409.2354978.

[15] Goodfellow, Ian, David Warde-Farley, Mehdi Mirza, Aaron Courville y Yoshua Bengio: *Maxout Networks.* En Dasgupta, Sanjoy y David McAllester (editores): *Proceedings of the 30th International Conference on Machine Learning*, volumen 28 de *Proceedings of Machine Learning Research*, páginas 1319–1327, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. http://proceedings.mlr.press/v28/goodfellow13.html.

[16] Grigorescu, S. E., N. Petkov y P. Kruizinga: *Comparison of Texture Features Based on Gabor Filters*. Trans. Img. Proc., 11(10):1160–1167, Octubre 2002, ISSN 1057-7149. http://dx.doi.org/10.1109/TIP.2002.80426.

[17] Han, Jun y Claudio Moraga: *The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning*. En *Proceedings of the International Workshop on Artificial Neural Networks: From Natural to Artificial Neural Computation*, IWANN '96, páginas 195–201, London, UK, UK, 1995. Springer-Verlag, ISBN 3-540-59497-3. http://dl.acm.org/citation.cfm?id=646366.689307.

[18] He, Kaiming, Xiangyu Zhang, Shaoqing Ren y Jian Sun: *Deep Residual Learning for Image Recognition*. CoRR, abs/1512.03385, 2015. http://arxiv.org/abs/1512.03385.

[19] Hinton, G E y R R Salakhutdinov: *Reducing the dimensionality of data with neural networks*. Science, 313(5786):504–507, Julio 2006. http://www.ncbi.nlm.nih.gov/sites/entrez?db=pubmed&uid=16873662&cmd=showdetailview&indexed=google.

[20] Huang, Gao, Zhuang Liu y Kilian Q. Weinberger: *Densely Connected Convolutional Networks*. CoRR, abs/1608.06993, 2016. http://arxiv.org/abs/1608.06993.

[21] Huang, Jing, S. Ravi Kumar, Mandar Mitra, Wei Jing Zhu y Ramin Zabih: *Image Indexing Using Color Correlograms*. En *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, CVPR '97, páginas 762–, Washington, DC, USA, 1997. IEEE Computer Society, ISBN 0-8186-7822-4. http://dl.acm.org/citation.cfm?id=794189.794514.

[22] *ImageNet*. http://image-net.org/index, Visitada en 2016-01-24.

[23] *Press Page - Instagram*. https://www.instagram.com/press/?hl=en, Visitada en 2016-01-19.

[24] Koch, Gregory, Richard Zemel y Ruslan Salakhutdinov: *Siamese Neural Networks for One-shot Image Recognition*. En *ICML Deep Learning workshop*, 2015.

[25] Krizhevsky, Alex, Ilya Sutskever y Geoffrey E. Hinton: *ImageNet Classification with Deep Convolutional Neural Networks*. En *Proceedings of the 25th International Conference on Neural Information Processing*

Systems, NIPS'12, páginas 1097–1105, USA, 2012. Curran Associates Inc. http://dl.acm.org/citation.cfm?id=2999134.2999257.

[26] Larsson, Gustav, Michael Maire y Gregory Shakhnarovich: *Learning Representations for Automatic Colorization*. CoRR, abs/1603.06668, 2016. http://arxiv.org/abs/1603.06668.

[27] LeCun, Y., L. Bottou, Y. Bengio y P. Haffner: *Gradient-Based Learning Applied to Document Recognition*. En *Intelligent Signal Processing*, páginas 306–351. IEEE Press, 2001.

[28] Lin, Tsung-Yi, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár y C. Lawrence Zitnick: *Microsoft COCO: Common Objects in Context*. CoRR, abs/1405.0312, 2014. http://arxiv.org/abs/1405.0312.

[29] Lowe, David G.: *Object Recognition from Local Scale-Invariant Features*. En *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, páginas 1150–, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0164-8. http://dl.acm.org/citation.cfm?id=850924.851523.

[30] Manjunath, B. S. y W. Y. Ma: *Texture Features for Browsing and Retrieval of Image Data*. IEEE Trans. Pattern Anal. Mach. Intell., 18(8):837–842, Agosto 1996, ISSN 0162-8828. http://dx.doi.org/10.1109/34.531803.

[31] Misra, Ishan, C. Lawrence Zitnick y Martial Hebert: *Unsupervised Learning using Sequential Verification for Action Recognition*. CoRR, abs/1603.08561, 2016. http://arxiv.org/abs/1603.08561.

[32] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra y Martin A. Riedmiller: *Playing Atari with Deep Reinforcement Learning*. CoRR, abs/1312.5602, 2013. http://arxiv.org/abs/1312.5602.

[33] *The MNIST database of handwritten digits*. http://yann.lecun.com/exdb/mnist/, Visitada 2016-05-25.

[34] Mycielski, Jan: *Review: Marvin Minsky and Seymour Papert, Perceptrons, An Introduction to Computational Geometry*. Bull. Amer. Math. Soc., 78(1):12–15, Enero 1972. https://projecteuclid.org/euclid.bams/1183533389.

[35] Noroozi, Mehdi y Paolo Favaro: *Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles*. CoRR, abs/1603.09246, 2016. http://arxiv.org/abs/1603.09246.

[36] Ortiz, Raphael: *FREAK: Fast Retina Keypoint*. En *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, páginas 510–517, Washington, DC, USA, 2012. IEEE Computer Society, ISBN 978-1-4673-1226-4. http://dl.acm.org/citation.cfm?id=2354409.2354903.

[37] Osuna, Edgar, Robert Freund y Federico Girosi: *Support Vector Machines: Training and Applications*. Informe técnico, Massachusetts Institute of Technology, Cambridge, MA, USA, 1997.

[38] Owens, Andrew, Jiajun Wu, Josh H. McDermott, William T. Freeman y Antonio Torralba: *Ambient Sound Provides Supervision for Visual Learning*. CoRR, abs/1608.07017, 2016. http://arxiv.org/abs/1608.07017.

[39] Park, YoungJae, KeeHong Park y GyeYoung Kim: *Content-based Image Retrieval Using Colour and Shape Features*. Int. J. Comput. Appl. Technol., 48(2):155–161, Agosto 2013, ISSN 0952-8091. http://dx.doi.org/10.1504/IJCAT.2013.056023.

[40] Pass, G. y R. Zabih: *Histogram Refinement for Content-based Image Retrieval*. En *Proceedings of the 3rd IEEE Workshop on Applications of Computer Vision (WACV '96)*, WACV '96, páginas 96–, Washington, DC, USA, 1996. IEEE Computer Society, ISBN 0-8186-7620-5. http://dl.acm.org/citation.cfm?id=524178.836717.

[41] Rosenblatt, F.: *The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain*. Psychological Review, páginas 65–386, 1958.

[42] Rublee, Ethan, Vincent Rabaud, Kurt Konolige y Gary Bradski: *ORB: An Efficient Alternative to SIFT or SURF*. En *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, páginas 2564–2571, Washington, DC, USA, 2011. IEEE Computer Society, ISBN 978-1-4577-1101-5. http://dx.doi.org/10.1109/ICCV.2011.6126544.

[43] Shalev-Shwartz, Shai, Shaked Shammah y Amnon Shashua: *Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving*. CoRR, abs/1610.03295, 2016. http://arxiv.org/abs/1610.03295.

[44] Simonyan, Karen y Andrew Zisserman: *Very Deep Convolutional Networks for Large-Scale Image Recognition*. CoRR, abs/1409.1556, 2014. http://arxiv.org/abs/1409.1556.

[45] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever y Ruslan Salakhutdinov: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research, 15:1929–1958, 2014. http://jmlr.org/papers/v15/srivastava14a.html.

[46] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke y Andrew Rabinovich: *Going Deeper with Convolutions*. CoRR, abs/1409.4842, 2014. http://arxiv.org/abs/1409.4842.

[47] Szeliski, Richard: *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edición, 2010, ISBN 1848829345, 9781848829343.

[48] Taigman, Yaniv, Ming Yang, Marc'Aurelio Ranzato y Lior Wolf: *DeepFace: Closing the Gap to Human-Level Performance in Face Verification*. En *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, páginas 1701–1708, Washington, DC, USA, 2014. IEEE Computer Society, ISBN 978-1-4799-5118-5. http://dx.doi.org/10.1109/CVPR.2014.220.

[49] Torti López, Rubén Ezequiel: *Entrenamiento de modelos de aprendizaje profundo mediante autosupervisión*. https://github.com/ezetl/deep-learning-techniques-thesis, Visitada 2017-08.

[50] Vincent, Pascal, Hugo Larochelle, Yoshua Bengio y Pierre Antoine Manzagol: *Extracting and Composing Robust Features with Denoising Autoencoders*. En *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, páginas 1096–1103, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-205-4. http://doi.acm.org/10.1145/1390156.1390294.

[51] Xiao, Jianxiong, Krista A. Ehinger, James Hays, Antonio Torralba y Aude Oliva: *SUN Database: Exploring a Large Collection of Scene Categories*. Int. J. Comput. Vision, 119(1):3–22, Agosto 2016, ISSN 0920-5691. http://dx.doi.org/10.1007/s11263-014-0748-y.

[52] Yosinski, Jason, Jeff Clune, Yoshua Bengio y Hod Lipson: *How transferable are features in deep neural networks?* CoRR, abs/1411.1792, 2014. http://arxiv.org/abs/1411.1792.

[53] *Youtube Statistics.* https://www.youtube.com/yt/press/statistics.html, Visitada en 2016-01-19.

[54] Zeiler, Matthew D. y Rob Fergus: *Visualizing and Understanding Convolutional Networks.* CoRR, abs/1311.2901, 2013. http://arxiv.org/abs/1311.2901.