Instituto Tecnológico Autónomo de México

Uso de redes neuronales convolucionales para problemas generales

Tesis

que para obtener el título de

licenciado en matemáticas aplicadas

presenta

Gerardo Antonio López Ruiz

Asesor: Dr. Carlos Fernando Esponda Darlington

México, D.F. 2018

"Con fundamento en los artículos 21 y 27 de la Ley Federal del Derecho de Autor y como titular de los derechos moral y patrimonial de la obra titulada "**Uso de redes neuronales convolucionales para problemas generales**", otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la Biblioteca Raúl Bailléres Jr., la autorización para que fijen la obra en cualquier medio, incluido el electrónico, y la divulguen entre sus usuarios, profesores, estudiantes o terceras personas, sin que pueda percibir por tal divulgación una contraprestación".

AUTOR

Fecha

Firma

Agradecimientos

A mi madre, Elisa, sin ella nada de esto sería posible. Ella es la razón principal por la cual soy la persona que soy hoy y por ello, este logro también es suyo.

A mi hermano, Sebastian, por su reto continuo a superarme y quién al creer siempre en mi, me ha inspirado a llegar cada vez mas lejos.

A mi tía Esther y a mis abuelos Esther y Alfonso, por apoyarme en cada paso y estar ahí incondicionalmente.

A Clarissa, impulsándome siempre a ser mejor persona.

Al Dr. Carlos Fernando Esponda Darlington, quién ha sido un gran asesor y un apoyo a lo largo de esta tesis; no solo ha permitido que esta tesis exista, sino que su apoyo, tiempo y paciencia han permitido que logre continuar con mis estudios de maestría.

Al Dr. Juan Carlos Ovando Martinez y el Mto. Juan Salvador Marmol Yahya, los dos mejores profesores con los que he tenido el honor de llevar clase y quienes no solo me asistieron en la tesis, sino también fueron parte fundamental de mi admisión a la maestría.

Al Dr. Marco Antonio Morales Aguirre, por su asistencia en esta tesis, quién a pesar de no conocerme, dedicó tiempo y esfuerzo en lograr que esta tesis sea lo que es.

A mis amigos, sobre todo aquellos con los que me he desvelado estudiando, aprendiendo. Hicieron de mi carrera un placer.

Prefacio

El aprendizaje profundo es una de las herramientas de aprendizaje supervisado más utilizadas hoy en día, sobre todo para el reconocimiento de imágenes. El término se utiliza por primera vez en 1986 por Rina Dechter. Sin embargo, remonta a 1958 con Rosenblatt y el perceptrón, un algoritmo de reconocimiento de patrones.

El impacto del aprendizaje profundo en la industria empieza en los años 2000, donde las redes neuronales convolucionales procesaban del 10 al 20 por ciento de todos los cheques escritos en los Estados Unidos. Las aplicaciones de reconocimiento de voz a larga escala empezaron alrededor del año 2010.

Un algoritmo de aprendizaje profundo muy empleado hoy en día son las redes convolucionales. Este algoritmo es un tipo de redes neuronales especializadas para procesar datos que tienen una topología tipo cuadrícula. Algunos ejemplos pueden ser las series de tiempo, que pueden pensarse como cuadrículas de una dimensión tomando muestras en intervalos regulares de tiempo (o lattices). Así mismo, las imágenes pueden verse como cuadrículas de píxeles en dos dimensiones. Para este tipo de problemas, las redes neuronales convolucionales han sido muy exitosas en aplicaciones prácticas.

La hipótesis de esta tesis es que estas redes pueden resolver problemas supervisados que no tienen las formas previamente mencionadas. Haciendo un mapeo topológico de las bases de datos y convirtiéndolas en una especie de "pseudo imagen", la máquina pueda detectar patrones y pre-

decir con base en ellas. En esta tesis compararemos las redes neuronales convolucionales con otros métodos de aprendizaje supervisado, midiendo la predicción con respecto a distintas bases de datos y determinando si estas redes efectivamente tienen utilidad.

Cabe hacer notar que no es mi objetivo demostrar que modelo es superior que cualquier método de aprendizaje supervisado para cada base de datos; sino probar que es un método válido e iniciar un diálogo entre la comunidad para que quizá se tome en cuenta en la implementación de aprendizaje supervisado en diversas bases de datos como un algoritmo adicional.

Índice general

Capítulo 1

Introducción.

El trabajo presentado en esta tesis se resume en tres partes:

- Explicación de los modelos utilizados, los métodos para manipular la base de datos y para medir la precisión de los modelos.
- Explicación de las redes neuronales convolucionales y la creación del modelo utilizado para predecir, específicamente la transformación de datos creada para esto.
- Prueba de la hipótesis y comparación de los resultados a través de diversas bases de datos comúnmente utilizadas como ejemplos en el aprendizaje de máquina.

El objetivo del proyecto es probar que es un método válido e iniciar un diálogo para que se tome en cuenta en la implementación de aprendizaje supervisado como un algoritmo adicional. Los resultados fueron bastante prometedores a pesar de no demostrar que nuestro modelo es superior a cualquier otro (el cual no fue el objetivo desde un inicio) pudimos mostrar que es efectivo y que podría emplearse para otro tipo de problemas no previamente contemplados en su uso habitual.

Aprendizaje de máquina

El aprendizaje profundo es un tipo específico de algoritmo de aprendizaje de máquina supervisado. Cualquier método de aprendizaje de máquina es un algoritmo capaz de aprender de los datos. Tom Mitchell nos define este aprendizaje de la siguiente manera: "un programa de computadora se dice que es capaz de aprender de una experiencia **E** con respecto a un conjunto de tareas **T** y con un desempeño **P**; donde su desempeño (**P**) en sus tareas (**T**) mejoran con más experiencia (**E**)."

Los algoritmos de aprendizaje supervisado tienen una base de datos con atributos y adicionalmente, cada ejemplo esta asociado a una etiqueta. El término aprendizaje supervisado origina de poder ver una etiqueta que esta dada por un instructor o maestro, enseñándole a la máquina que hacer.

El reto principal del aprendizaje de máquina es que debemos predecir datos que el algoritmo nunca ha visto, no solo con los que entrenamos nuestro modelo. La habilidad de un algoritmo para desempeñarse bien en datos no observados se conoce como generalización.

La teoría del aprendizaje de máquina nos indica que un algoritmo puede generalizar bien desde un conjunto finito de entrenamiento. Sin embargo, esto puede parecer que contradice la lógica. Para evitar este problema, el aprendizaje de máquina busca encontrar reglas que son *probablemente* correctas para la *mayoría* de los elementos en el conjunto.

Al entrenar un modelo de aprendizaje de máquina, debemos tener acceso a un conjunto de *entrenamiento*, un *conjunto de prueba* y algún método para poder medir el error en dicho conjunto de prueba. Esto puede parecer un problema de optimización, sin embargo, la diferencia entre los dos radica en que en el aprendizaje de máquina, queremos que el error de la generalización también sea bajo.

Por ejemplo, en la regresión lineal entrenamos el modelo minimizando el error de entrenamiento,

$$\frac{1}{m_{(entrenamiento)}} \| X_{(entrenamiento)} w \neq y_{(entrenamiento)} \|,$$

Sin embargo, lo que nos importa es el error del conjunto de prueba,

$$\frac{1}{m_{(prueba)}} \| X_{(prueba)} w \neq y_{(prueba)} \|$$

Sobre-ajuste y sobre-generalización

¿Cómo podemos afectar el desempeño de un conjunto de prueba cuando solo podemos utilizar el conjunto de entrenamiento? Esto se debe a que en los problemas de aprendizaje de máquina asumimos que los datos de entrenamiento y de prueba están generados por una misma distribución probabilística y que los datos son independientes e idénticamente distribuidos. Con estas asunciones podemos decir que el error esperado del conjunto de entrenamiento y el error esperado del conjunto de prueba son el mismo.

Para los problemas de aprendizaje supervisado tomamos una muestra de los datos como conjunto de entrenamiento y lo utilizamos para elegir los parámetros que reducen el error de este conjunto. Posteriormente, el resto de los datos los usamos para crear el conjunto de prueba y aplicamos el algoritmo a este.

Para determinar que tan bien se desempeña el algoritmo, medimos su habilidad para:

- Hacer el error de entrenamiento pequeño.
- Hacer la diferencia entre el error de entrenamiento y el de prueba pequeña.

Estos dos factores corresponden a los dos retos principales en el aprendizaje de máquina: sobre-generalización (*underfitting*) y sobre-ajuste (*overfitting*). La sobre-generalización ocurre cuando el modelo no es capaz de encontrar un error suficientemente bajo en el conjunto de entrenamiento. El sobre-ajuste ocurre cuando la diferencia entre el error de entrenamiento y el de prueba es muy grande.

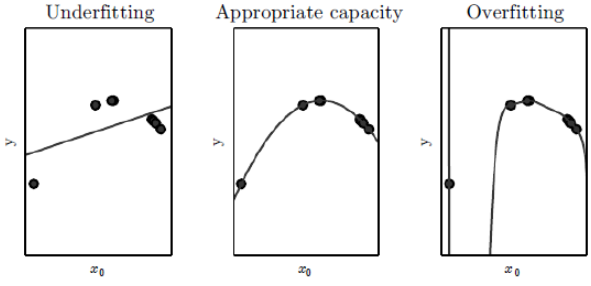

Figura 1.1: Ejemplo de sobre-ajuste y sobre-generalización, en la primera imagen vemos claramente que nuestra función lineal no esta acercándose mucho a los valores y que hay mucho espacio entre ellos, la tercera imagen muestra que crea una función que es muy específica para estos puntos y difícilmente si obtenemos mas datos van a estar cerca de esta función. (imagen obtenida de [11]).

Validación cruzada

Dividir la base de datos en un conjunto fijo de entrenamiento y un conjunto fijo de prueba puede ser problemático para bases de datos pequeñas y podrían llevar al sobre-ajuste. Un método muy común para evitar el sobre-ajuste es la validación cruzada (*cross validation*).

La validación cruzada consiste en dividir los datos en k subconjuntos y entrenar el algoritmo con uno de estos k subconjuntos, dejando los restantes $k-1$ conjuntos como conjuntos de pruebas, haciendo esto k veces al iterar sobre los conjuntos de datos. La ventaja de este método es que evita el problema de la división de datos ya que todos los datos se han uti-

lizado tanto para probar como para entrenar; evitando el sobre-ajuste de los datos, ya que el desempeño se medirá sobre el promedio de los modelos creados en cada iteración.

Algorithm 1 Algoritmo de validación cruzada. Obtenido de [9].

Definimos: Cross_validation(D,A,L,k)
Requerimos: D, la base de datos.
Requerimos: A, el algoritmo de aprendizaje, visto como una función que toma datos como entradas y una función de aprendizaje como salida.
Requerimos: L, la función de pérdida.
Requerimos: k, el número de dobleces (iteraciones).

Dividimos D en k subconjuntos exclusivos D_i, donde su unión es D.

for i de 1 a k **do**
    $f_i = A(D \setminus D_i)$
    for z^j in D_i **do**
        $e_j = L(f_i, z^j)$

Regresamos e

Matrices de confusión

En muchas ocasiones el tipo de error que tiene el modelo es muy importante. Por ejemplo, en la detección de enfermedades (cuyo costo de tratarlas es muy elevado), podría ser más importante que los que sean detectados tengan la enfermedad aunque no se encuentren algunos que sí estén enfermos. O al contrario, si es una enfermedad donde la detección es imperativa, encontrar la mayor cantidad de enfermos sin importar tanto que alguien no lo esté.

Para esto, se utiliza la matriz de confusión. Una matriz de confusión contiene información sobre el tipo de error que tiene un algoritmo. Cada entrada de la matriz significa un tipo de predicción dado. Por ejemplo, si

las etiquetas que tienes son binarias, una matriz de confusión quedaría de la siguiente manera:

- VN es el número de predicciones correctas cuando la etiqueta es 0.
- FP es el número de predicciones incorrectas cuando la etiqueta es 1.
- FN es el número de predicciones incorrectas cuando la etiqueta es 0.
- VP es el número de predicciones correctas cuando la etiqueta es 1.

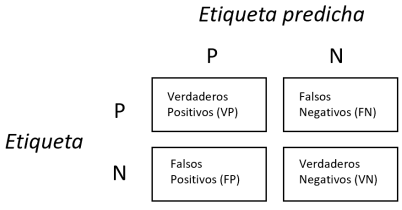

Figura 1.2: Ejemplo de una matriz de confusión(imagen obtenida de [12]).

Computación numérica

El aprendizaje de máquina normalmente tiene un alto costo computacional. Esto se debe a que los algoritmos empleados normalmente obtienen su resultado mediante un proceso iterativo, en vez de contar con una fórmula específica para predecir. Adicionalmente a esto, una de las dificultades más grandes es resolver problemas continuos con un número finito de dígitos. Esto significa que para casi todos los números reales, habrá algún error de aproximación al representar el número en la computadora. Esta aproximación se hace redondeando el número.

Overflow y underflow

Una forma de redondeo que puede llegar a causar problemas muy fuertes es el *underflow*. Esto ocurre cuando un número cercano a cero se redondea

como cero, ya que muchas funciones tendrán un comportamiento muy distinto en el cero que en números muy cercanos al cero. Por ejemplo, dividir entre cero o el logaritmo de cero.

Otra forma de redondeo que puede causar problemas es el *overflow*. El overflow surge cuando números con una magnitud muy grande son aproximados a ∞ o $-\infty$. Hacer operaciones cuando esto sucede causará que los valores se convirtiesen en *NAN* (término que significa *Not a number)*.

Un ejemplo muy claro de una función que debe ser estabilizada es la función *Softmax*. Esta función normalmente se utiliza para predecir las probabilidades asociadas con una distribución multinomial. Explicaremos la función Softmax a mayor detalle en capítulos posteriores, sin embargo, está definida como:

$$Softmax(x)_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}$$

Vea pues, qué es lo que pasa cuando todas las x_i son iguales a una constante c. Analíticamente, es evidente que el resultado debería ser $\frac{1}{n}$. No obstante, esto puede no suceder cuando c es muy grande. Si c es muy negativo, entonces tendremos un overflow en $exp(c)$. Esto significa que el denominador de la función se volverá cero y entonces el resultado final no estará definido. Cuando c es muy grande y positivo, $exp(c)$ tenderá a infinito y la función nuevamente no estará definida (overflow).

Para arreglar este tipo de problemas, podemos evaluar la función Softmax como $Softmax(z)$ donde $z = x - max_i x_i$. Este cambio hace que el argumento más grande de la exponencial sea 0, que evita la posibilidad de un overflow. Adicionalmente, al menos un término del denominador debe ser igual a 1, lo cual elimina la posibilidad de underflow.

Estos conceptos son muy importantes de conocer, sin embargo, hay muchas bibliotecas que ya hacen esto por nosotros, incluyendo Keras. Biblioteca en la cual nosotros crearemos nuestras redes neuronales convolucionales.

Mal condicionamiento

El condicionamiento computacional es qué tan rápido cambia una función con respecto a pequeños cambios en los datos de entrada. Es importante saber el condicionamiento de la función, ya que podría llevar a errores de sobre-ajuste, donde un pequeño cambio en la base de datos puede dar errores muchos más altos. Entonces, aunque se tenga una base de entrenamiento con buena precisión, tal vez su base de prueba no tendría buenos resultados.

Un ejemplo sencillo puede ser una función como la siguiente: $f(x) = A^{-1}x$. Cuando $A \in R^{n \times n}$ tiene una descomposición en valores propios, su condicionamiento es:

$$\max_{(i,j)} \frac{\lambda_i}{\lambda j}$$

Ese es el radio de magnitud entre el eigenvalor más grande y el más pequeño. Cuando este número es grande, la inversión matricial es muy sensible a pequeños cambios en los datos de entrada.

Descenso por gradiente

La gran mayoría de los algoritmos de aprendizaje profundo utilizan algún tipo de optimización. Con optimización nos referimos a el objetivo de minimizar o maximizar una función $f(x)$ alterando x. La función que queremos minimizar o maximizar es llamada la función objetivo o el criterio.

Ahora que entendemos los problemas de los cuales debemos alejarnos, veremos un método de optimización muy importante y que estaremos utilizando en nuestras redes neuronales convolucionales: el descenso por gradiente.

Asumamos que tenemos una función $y = f(x)$, donde tanto x como y son números reales. La derivada de esta función $f'(x)$ da la pendiente de $f(x)$ en un punto x. Esto denota cuánto cambia la salida de una función

ante pequeños cambios. Por ejemplo, $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

La derivada es entonces utilizada para cambiar x en pequeñas cantidades y por consiguiente, para cambiar y. Por ejemplo, sabemos que $f(x - \epsilon signo(f'(x)))$ es menor que $f(x)$ para una ϵ suficientemente chica. Podemos entonces reducir $f(x)$ moviendo x en pequeñas cantidades con el signo opuesto de la derivada.

Cuando $f'(x) = 0$, la derivada no aporta mayor información sobre hacia donde moverse. Estos puntos se conocen como puntos estacionarios o puntos críticos. Un mínimo local es un punto donde $f(x)$ es menor que todos los puntos vecinos, por lo tanto no es posible disminuir $f(x)$ con valores pequeños. Así mismo, los puntos que no son mínimos o máximos locales pero donde $f(x) = 0$ se denominan puntos de silla de montar.

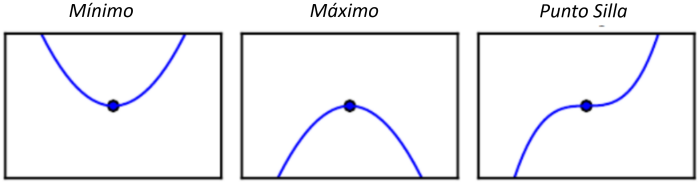

Figura 1.3: Ejemplos de puntos mínimos, máximos y sillas (imagen obtenida de [12]).

En el aprendizaje profundo, buscamos optimizar funciones que tienen muchos mínimos locales que no son óptimos, rodeados de puntos sillas en regiones muy aplanadas. Esto hace que la optimización sea muy complicada, sobre todo cuando hablamos de funciones que tienen muchas dimensiones. Por ello, normalmente nos conformamos con obtener valores donde $f'(x)$ es muy bajo, pero que no necesariamente sean mínimos locales.

El descenso por gradiente generaliza la noción de derivada al caso en el cual derivamos con respecto a un vector. Esto significa que el gradiente de f es un vector que contiene todas las derivadas parciales, denotado $\Delta_i f(x)$.

En múltiples dimensiones, los puntos críticos son aquellos puntos don-

de cada elemento del gradiente es igual a cero. La derivada direccional con dirección u (una unidad vectorial) es la pendiente de la función f con dirección u. Por esto, la derivada direccional es la derivada de la función $f(x+\alpha u)$ con respecto a α cuando $\alpha = 0$. Usando la regla de la cadena podemos ver que la derivada parcial $\frac{\delta}{\delta\alpha} f(x+\alpha u)$ es $u^\top \Delta_x f(x)$ cuando $\alpha = 0$.

Para minimizar f, nos gustaría encontrar la dirección en la cual f disminuye con mayor rapidez. Podemos hacer esto usando la derivada direccional:

$$\min_{u, u^\top u=1} u^\top \Delta_x f(x) = \min_{u, u^\top u=1} \|u\|_2 \|\Delta_x f(x)\|_2 \cos\theta$$

donde θ es el ángulo entre u y el gradiente. Sustituyendo $\|u\|_2 = 1$ e ignorando el hecho que no dependen de u, simplificamos la ecuación a $\min_u \cos\theta$. Esta ecuación se minimiza cuando la dirección u apunta a la dirección contraria del gradiente. Moviendo f en la dirección opuesta al gradiente es conocido como el método de descenso por gradiente.

Ejemplo: Mínimos cuadrados

Para ejemplificar este método, supongamos que queremos encontrar x que minimice

$$f(x) = \frac{1}{2}\|Ax - b\|_2^2$$

Primero, obtenemos el gradiente:

$$\Delta_x f(x) = A^\top (Ax - b) = A^\top Ax - A^\top b$$

El algoritmo quedaría como:

Algorithm 2 Algoritmo para minimizar $f(x) = \frac{1}{2}\|Ax - b\|_2^2$ con respecto a x usando descenso por gradiente, empezando por un valor aleatorio de x.

Establecemos el tamaño ϵ del paso y la tolerancia δ.

while $\|A^\top Ax - A^\top b\|_2 > \delta$ **do**
    x = x - $\epsilon(A^\top Ax - A^\top b)$
Regresamos x

Usando el método de Newton, dado que la función es cuadrática, su aproximación mediante el método es exacta y el algoritmo converge al mínimo global en un solo paso.

Capítulo 2

Modelos de aprendizaje utilizados

En el capítulo previo, buscamos explicar lo que es el aprendizaje de máquina; junto con algunos métodos utilizados para evitar errores comunes. Este capítulo se enfoca en mostrar algunos algoritmos comúnmente empleados para predecir y que utilizaremos para comparar nuestros resultados obtenidos con las redes neuronales convolucionales.

Máquinas de soporte vectorial

La máquina de soporte vectorial es un método de aprendizaje supervisado que engloba la siguiente idea: mapea los vectores de entrada a un espacio de atributos con alta dimensionalidad Z a través de un mapeo no lineal elegido *a priori*.

Por ejemplo, para obtener un espacio de decisión que corresponda a un polinomio de dos grados, uno puede crear un espacio de atributos, Z, con $N = \dfrac{n(n+3)}{2}$ coordenadas con la siguiente estructura:

$$z_1 = x_1, \ldots, z_n = x_n \qquad n \quad \text{coordenadas},$$
$$z_{n+1} = x_1^2, \ldots, z_{2n} = x_n^2 \qquad n \quad \text{coordenadas},$$

$$z_{2n+1} = x_1 x_2, \ldots, z_N = x_n x_{n+1} \qquad \frac{n(n-1)}{2} \quad \text{coordenadas,}$$

donde $x = (x_1, \ldots, x_n)$. El hiperplano es entonces construido en este espacio.

Dos problemas surgen con el método mencionado, uno conceptual y uno técnico. El conceptual es que debemos encontrar un hiperplano que generalice bien; dado que tenemos un espacio de atributos muy grande, no todos los hiperplanos que separan los datos lo harán de manera óptima. El problema técnico consiste en como tratar los datos si a un polinomio de grado 4 o 5, lo estamos viendo en un espacio de 200 dimensiones, un polinomio de mayor tamaño podría llegar a necesitar millones de dimensiones.

La parte conceptual fue resuelta por Vapnik en 1965 (ver [21]) donde nos explica que un hiperplano óptimo esta definido como una función de decisión lineal que separa los vectores de dos clases distintas con un margen máximo entre ellos (ver imagen). No obstante, para construir estos hiperplanos, uno solo debe tomar un pequeño extracto de la base de entrenamiento, los vectores de soporte, los cuales determinarán el margen.

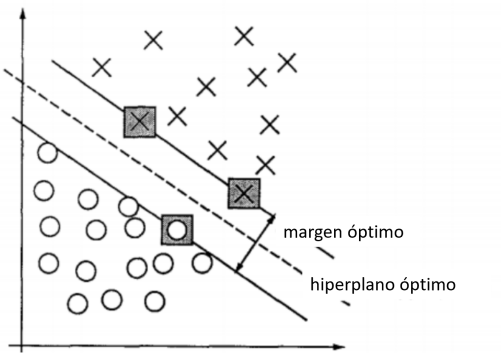

Figura 2.1: Ejemplos de un problema separable en un espacio de dos dimensiones. Los vectores de soporte, marcados con cuadrados grises, definen el margen máximo entre dos clases. (imagen obtenida de [21]).

Algoritmo para encontrar el hiperplano óptimo

Sea el conjunto de datos de entrenamiento, con sus respectivas etiquetas (modelo tomado de [21]):

$$(y_1, x_1), \ldots, (y_l, x_l), \qquad y_i \in \{-1, 1\}$$

se dice que este conjunto es linealmente separable si existe un vector w y un escalar b tal que las desigualdades

$$w \cdot x_i + b \geqslant 1 \qquad \text{si} \quad y_i = 1$$
$$w \cdot x_i + b \leq -1 \qquad \text{si} \quad y_i = -1$$

son válidas para todos los elementos del conjunto de entrenamiento. Podemos unir estas ecuaciones de la siguiente manera:

$$y_i(w \cdot x_i + b) \geqslant 1, \qquad i = 1, \ldots, l$$

El hiperplano

$$w_0 \cdot x + b_0 = 0$$

es el único que separa el conjunto de entrenamiento maximizando el margen, esto es porque determina la dirección $w/\|w\|$ donde la distancia entre las proyecciones de los vectores de entrenamiento son los máximos. La proyección $\rho(w, b)$ se obtiene como sigue:

$$\rho(w, b) = \min_{x;y=1} \frac{x \cdot w}{\|w\|} - \max_{x;y=-1} \frac{x \cdot w}{\|w\|}$$

Sustituyendo con el hiperplano dado, tenemos:

$$\rho(w_0, b_0) = \frac{1}{\|w_0\|} - \frac{-1}{\|w_0\|} = \frac{2}{\|w_0\|}$$

Esto significa que el hiperplano óptimo es el único que minimiza $w \cdot w$ bajo las restricciones; construyendo un hiperplano óptimo es entonces, un problema de programación cuadrática.

Los vectores x_i para los cuales $y_i(w \cdot x_i + b) = 1$ se denominarán como vectores de soporte. El vector w_0 que determina el hiperplano óptimo puede escribirse como una combinación de vectores de entrenamiento:

$$w_0 = \sum_{i=1}^{l} y_i \alpha_i^0 x_i$$

donde $\alpha_i^0 \geqslant 0$. Dado que α>0 únicamente para los vectores de soporte, la expresión previa representa una manera compacta de escribir w_0. Adicionalmente, para encontrar el vector de parámetros α_i:

$$\Lambda_0^\top = (\alpha_1^0, \ldots, \alpha_t^0)$$

uno debe resolver el problema cuadrático:

$$W(\Lambda) = \Lambda^\top 1 - \frac{1}{2}\Lambda_0^\top D\Lambda$$

con respecto a $\Lambda^\top = (\alpha_1^0, \ldots, \alpha_t^0)$ sujeto a las restricciones:

$$\Lambda \geqslant 0,$$
$$\Lambda^\top Y = 0,$$

donde $1^\top = (1, \ldots, 1)$ es un vector unitario de l dimensiones, $Y^\top = (y_1, \ldots, y_l$ es el vector de etiquetas y D es una matriz simétrica $l \times l$ con elementos:

$$D_{i,j} = y_i y_j x_i \cdot x_j, \qquad i, j = 1, \ldots l$$

La desigualdad $\Lambda \geqslant 0$ describe el cuadrante no negativo. Por esto, debemos maximizar $W(\Lambda) = \Lambda^\top 1 - \frac{1}{2}\Lambda_0^\top D\Lambda$ en el cuadrante no negativo, sujeto a la restricción $\Lambda^\top Y = 0$.

Este problema se puede resolver dividiendo el conjunto de entrenamiento en subconjuntos pequeños. Empezamos resolviendo el problema de programación cuadrática para el primer subconjunto. Tenemos entonces dos

posibles resultados: que los datos no tienen un hiperplano óptimo que los separe, lo cual significa que todo el conjunto de datos no puede separarse, o encontramos un hiperplano óptimo que los separen.

Sea entonces el vector que maximiza el problema en el primer subconjunto Λ_1. Entre las coordenadas de Λ_1 algunas serán igual a cero; aquellas corresponden a los vectores que no son de soporte del subconjunto. Hacemos un nuevo subconjunto de entrenamiento que contiene los vectores de soporte del primer subconjunto y vectores del segundo subconjunto que no satisfacen $y_i(w \cdot x_i + b) \geqslant 1, \quad i = 1, \ldots, l$, donde w esta determinada por Λ_1. Para este conjunto, una nueva función $W_2(\Lambda)$ es construida y maximizada por Λ_2. Continuando con este proceso se llega a que los datos no pueden separarse sin error o a un vector solución Λ_* que construye el hiperplano óptimo.

Modelo de árboles de decisión

Los árboles de decisión son modelos predictivos de aprendizaje supervisado y actualmente son de los mejores métodos para hacer predicciones. Hay muchos métodos para crear árboles, sin embargo, nos enfocaremos en el sistema C4.5 (explicado en [14]). Este es el árbol de decisión con mejores resultados para análisis supervisado que se encuentra disponible al público. Actualmente ya existe el algoritmo C5, que es mejor, pero no se encuentra disponible.

La entrada de datos de los árboles C4.5 consiste en una colección de datos de entrenamiento, cada uno con una tupla de valores representando un número finito de parámetros y una variable dependiente. Esto podemos verlo como $A = A_1, A_{2,k}$. La clase C es lo que queremos predecir y tiene valores $C_1, C_2, ..z.., C_x$. El objetivo puede verse como una función:

$$DOM(A_1) \times DOM(A_2) \times \cdots \times DOM(A_k)DOM(C)$$

que mapea los parámetros a una clase predicha.

La estructura del árbol consiste en:

- Nodos denominados *hoja*, que contienen las decisiones a tomar (las clases).
- Nodos denominados *pruebas*, contienen las condiciones determinadas por los parámetros.

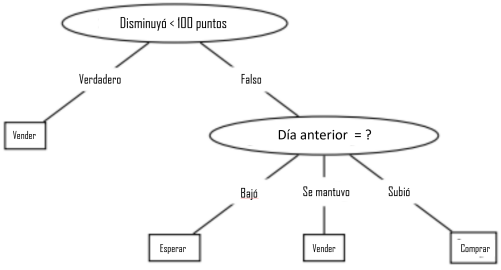

Figura 2.2: Ejemplo de un árbol de decisión, donde las hojas son rectángulos y las pruebas son óvalos, las decisiones se toman de arriba hacia abajo. (imagen obtenida de [14]).

Método de dividir y conquistar

Los árboles de decisión utilizan un método denominado dividir y conquistar para construir árboles de un conjunto S de entrenamiento. Este método consiste en:

- Si todos los casos en S pertenecen a la misma clase (por ejemplo, C_j) el árbol de decisión es una hoja etiquetada con C_j.
- En caso contrario, sea B una prueba con resultados $b_1, b_2, \ldots, b_t$ que produce una partición no trivial de S, y denotemos S_i el conjunto de casos en S en los cuales el resultado sea $b_i \in B$.

El árbol quedaría de la siguiente manera:

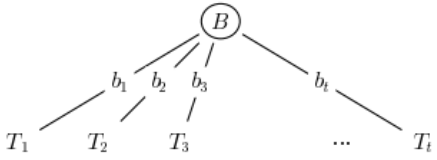

Figura 2.3: Ejemplo de dividir y conquistar donde T_i es el resultado de crecer el árbol para los casos en S_i.(imagen obtenida de [14]).

Eligiendo las pruebas

En el algoritmo de dividir y conquistar, cualquier prueba B que particiona S de manera no trivial, creará un árbol de decisión, pero diferentes B darán árboles distintos. C4.5 busca que los árboles sean los más pequeños posibles, esto es porque permiten ser entendidos mejor y dan un mejor nivel de predicción. No obstante, dado que no es posible garantizar que se encontrará el árbol más pequeño, C4.5 utiliza el método de búsqueda voraz, el cual selecciona los candidatos de tal manera que se maximice el criterio de partición.

Hay dos criterios de partición que se emplean, el criterio de ganancia de información y el criterio de radio de ganancia. Sea $RF(C_j, S)$ la función que denote la frecuencia relativa de que los casos en que S pertenece a la clase C_j. Al partir S en t subconjuntos para una prueba B, la ganancia de información estaría dada por:

$$G(S, B) = I(S) - \sum_{i}^{t} \frac{|S_i|}{|S|} I(S_i)$$

donde

$$I(S) = -\sum_{j=1}^{x} RF(C_j, S) log(RF(C_j, S))$$

entonces se busca B para S que maximice la ganancia de información

$(G(S, B))$.

Un problema con este criterio es que favorece múltiples resultados, por ejemplo, $G(S, B)$ se maximiza con una prueba en la cual cada S_i contiene un solo caso. El criterio de radio de ganancia evita este problema al también tomar en cuenta la información de la partición. Lo que hace entonces es buscar maximizar $G(S, B)/P(S, B)$ donde:

$$P(S, B) = -\sum_{i}^{t} \frac{|S_i|}{|S|} log(\frac{|S_i|}{|S|})$$

Podando el árbol

Una vez que el árbol fue creado, se busca podar el árbol para que sea más pequeño y tenga mejores niveles de predicción. El método de podado se emplea para evitar el sobre ajuste, eliminando hojas que son muy específicas para el conjunto de entrenamiento y que por tanto, no generalizan de manera óptima. El método empleado por C4.5 consiste en:

Considera un clasificador Z formado por un subconjunto de S y asumamos que Z clasifica erróneamente M casos en S. El error real de Z es su precisión con respecto a todos los elementos sobre los cuales fue tomada la muestra. C4.5 calcula el error real de Z utilizando solamente los valores de M y $|S|$ como sigue: Si un evento sucede M veces en N pruebas, M/N es la probabilidad estimada p de que ocurra dicho evento.

No obstante, C4.5 va más allá y obtiene un nivel de confianza CF para un límite superior p_r, tal que $p \leq p_r$ con probabilidad $1 - CF$. Las ecuaciones están dadas como sigue:

$$\text{CF} = \begin{cases} (1 - p_r)^N, \text{ para } M = 0 \\ \sum_{i=0}^{M} \binom{N}{i} p_r^i (1 - p_r)^{N-i}, \text{ para } M > 0 \end{cases}$$

Utilizamos U_{CF} para denotar el error superior a p_r. C4.5 usa 0.25 como el valor por defecto para podar.

Ahora, sea T un árbol de decisión que no consiste en una sola hoja, producido por un conjunto de entrenamiento S, donde T_i^* es un subárbol que ya ha sido podado. Adicionalmente, sea T_f^* el subárbol que tiene mayor cantidad de resultados B y sea L la hoja con mayor cantidad de etiquetas en S. Finalmente, el número de casos en S mal clasificados por T, T_f^* y L que sean $E_T, E_{T_f^*}$ y E_L respectivamente. El algoritmo de podado de C4.5 considera tres errores:

- $U_{CF}(E_T, |S|)$
- $U_{CF}(E_L, |S|)$; y
- $U_{CF}(E_{T_f^*}, |S|)$

Según cual sea menor, C4.5

- Deja T sin cambios.
- Reemplaza T por la hoja L; ó
- Reemplaza T por su subárbol T_f^*

Bosques aleatorios

Un bosque aleatorio es un conjunto de árboles de decisión, cada uno de estos árboles de decisión es creado con una selección aleatoria de atributos y un subconjunto de los datos del modelo. Así que es una modificación al método de bagging. Este método es muy popular debido a su gran desempeño cuando se posee gran cantidad de información ya que es muy fácil de implementar y de entrenar. Veremos más a fondo este método en capítulos posteriores.

Capítulo 3

Redes neuronales convolucionales

La hipótesis de esta tesis es que las redes neuronales convolucionales pueden resolver problemas supervisados generales haciendo un mapeo topológico de las bases de datos y mediremos su eficiencia con respecto a otros algoritmos. Dado que ya conocemos los algoritmos que utilizaremos para la comparativa, es importante que se comprenda a fondo como funcionan las redes neuronales (utilizaremos RNNs como abreviación) y con esto, las redes neuronales convolucionales.

El objetivo de este capítulo es describir el funcionamiento de este algoritmo a detalle; así mismo, explicar en qué consisten un par de capas de las redes que ayudan a mejorar la eficiencia: pooling y dropout (ver 3.3.2 y 3.2.3 respectivamente). Adicionalmente, vamos a utilizar redes neuronales que no son convolucionales para esta comparativa y explicarlas también es un objetivo; esto es porque además de ser un buen método, lo que queremos demostrar es que la convolución es válida para este tipo de problemas, no solamente las redes.

Aprendizaje profundo

Las redes neuronales son un conjunto de nodos (neuronas) que están inspiradas en el funcionamiento neuronal del cerebro. Las neuronas están

conectadas e interactúan entre ellas; cada una toma datos de entrada y realiza operaciones sencillas con dichos datos. La forma más sencilla de entender una red neuronal es a través del concepto de perceptrón, fue creado en 1958 por Frank Rosenblatt; aunque hoy la función de pesos tiene otras formas (como la regresión logística), el concepto de perceptrón sirve para entender fácilmente el funcionamiento de cada nodo en una red. Éste, consiste en tomar entradas binarias y producir salidas binarias.

Si x_j es el valor de entrada, w_j un *peso* y b el *umbral*, la salida de la neurona es determinada de la siguiente forma:

$$\text{Salida} = \begin{cases} 1, \text{ si } \sum_j w_j x_j \geqslant b \\ 0, \text{ si } \sum_j w_j x_j < b \end{cases}$$

Los pesos representan la intensidad de la conexión entre unidades. Si el peso entre la unidad A y la unidad B es de mayor magnitud (mientras todos los demás quedan igual), significa que A tiene mayor influencia sobre B. También se puede interpretar como una forma de medir lo que a cada nodo "le interesa"; esto quiere decir que cada decisión tomada por un nodo, está directamente influenciada por las decisiones de los otros nodos en un estado previo.

En el contexto del problema de reconocimiento de imágenes, definiremos los pesos de estos nodos a partir de la naturaleza de una imagen. Esta puede ser vista como una matriz de píxeles, donde cada entrada de la matriz representa un píxel y estas adquieren un valor positivo (entre más alto es el valor, más claro es el tono de la imagen).

Entendiendo el principio básico de una neurona, veamos el funcionamiento de las redes neuronales. Existen tres *capas* o *divisiones* dentro de cada red; para entender mejor cada una usaremos el ejemplo de una red para procesar imágenes (ver [16]):

1. **Capa de Entrada** Es la inicialización de la red. Se reciben las entradas iniciales; en nuestro ejemplo serían píxeles. Estas son tomadas como una suma ponderada de los *valores de intensidad del píxel* y

pasan a una función de activación (como por ejemplo, la regresión logística). Gracias a que dicha función es monótona, la activación de una unidad será más alta cuando los valores de entrada de los píxeles sean similares a los pesos de dicha unidad (en el sentido de tener un producto interno grande); por tanto, podemos entender a los pesos como un filtro de coeficientes, definiendo una cualidad de la imagen.

2. **Capa Oculta** Son ocultas porque en ellas no se reciben las entradas crudas y tampoco son aquellas que dan el resultado final del proceso. En ellas, se realiza el proceso central de otorgar nuevos pesos que se pueden interpretar como *entradas de patrones preferidas.*

3. **Capa de Salida** Otorga los resultados del proceso.

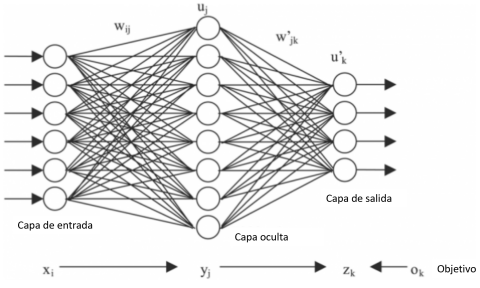

Figura 3.1: Ejemplo red neuronal (imagen obtenida de [12]).

En cada capa tenemos múltiples neuronas que van tomando los datos de salida de la neurona previa y arrojan nuevos datos. La manera en que la información se procesa y la manera en que se conectan las neuronas es lo que hace que una red neuronal se diferencie de otra. El tipo de redes neuronales descritas se denominan *redes neuronales feedforward.*

Redes neuronales feedforward.

Las redes neuronales *feedforward* son el modelo de aprendizaje profundo por excelencia. El objetivo de estas redes es el aproximar a una función

f^*, para un clasificador $y = f^*(x)$, donde la entrada x es mapeada a una categoría y. Una red neuronal feedforward define una función de mapeo $y = f(x;\theta)$ y aprende el valor del parámetro θ que mejor aproxima la función.

Estos modelos son conocidos como *feedforward* porque la información fluye a través de la función a ser evaluada en x a las computaciones necesarias para definir f y finalmente a la respuesta y; no obstante, no hay conexiones de retroalimentación en estas redes. Cuando las redes neuronales cuentan con retroalimentación son denominadas redes neuronales recurrentes.

Las redes neuronales feedforward son llamadas redes porque normalmente están representadas al contener múltiples funciones compuestas. Por ejemplo, podemos tener tres funciones $f^{(1)}(x)$,$f^{(2)}(x)$,$f^{(3)}(x)$ conectadas en cadena para formar $f^{(3)}(f^{(2)}(f^{(1)}(x))$. Estas estructuras son las más comunes entre las redes neuronales. Previamente vimos los tres tipos de capas que tienen las redes neuronales, en este caso, $f^{(1)}$ es la primera capa, $f^{(2)}$ es la segunda capa y así sucesivamente. El tamaño de la cadena nos da la profundidad del modelo.

Softmax

Uno de los algoritmos que vamos a utilizar para comparar nuestras redes convolucionales son las redes neuronales feedforward con regresión Softmax. Por ello, es importante explicar qué son y como funcionan (ver [5]). Dado que ya conocemos las redes neuronales feedforward, solo falta explicar como funciona la regresión Softmax.

La regresión Softmax consiste en dos pasos: primero añadimos la evidencia de que nuestra entrada esté dentro de ciertas clases y la convertiremos en probabilidades. Para sumar la evidencia hacemos una suma de pesos (W) de la intensidad de píxeles; así mismo, añadimos el sesgo; con esto la evidencia de una clase i dada una entrada x es:

$$\text{Evidencia}_i = \sum_{j=1} w_{i,j} * x_j + b_i,$$

donde w_i son los pasos y b_i es el sesgo para la clase i, j; j es un índice para sumar los píxeles en nuestra imagen x.

Convertimos entonces el valor de nuestra evidencia en probabilidad y, usando la función Softmax, obtenemos

$$y = \text{Softmax(evidence)}.$$

Lo que hace es básicamente convertir la evidencia en una distribución de probabilidad para los caso. Está definida como:

$$\text{Softmax}(x) = \text{normalizar}(e^x).$$

Es importante notar que el proceso exponencía los datos para que a mayor evidencia se de mayor peso, y a menor evidencia, menor peso. Posteriormente normaliza los pesos para que sumen uno.

El procedimiento queda como sigue:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{Softmax}\left[\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \right]$$

i.e.

$$y = \text{Softmax}(Wx + b)$$

ReLU (Rectified Linear Units)

La función ReLU es de las funciones de activación más utilizadas en modelos de aprendizaje profundo (ver [22]). Lo que hace esta función es regresar un 0 si recibe un dato negativo, pero en caso de que el valor sea positivo, regresa ese mismo valor. Entonces, podemos ver la función como:

$$f(x) = max(0, x)$$

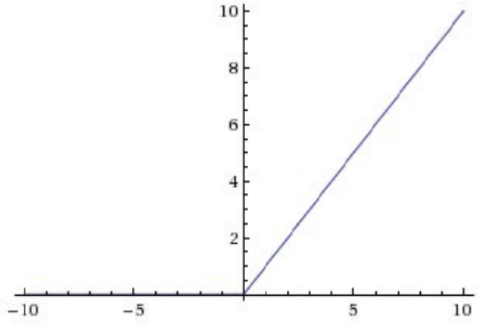

Figura 3.2: Función ReLU (imagen obtenida de [22]).

Regularización para el aprendizaje profundo

Un problema muy importante en el aprendizaje de máquina es que un algoritmo que funciona muy bien en el conjunto de entrenamiento, también funcione de manera óptima en el conjunto de prueba. Por esto, hay múltiples estrategias para reducir el error del conjunto de prueba (muchas veces aumentando el conjunto de entrenamiento). Estas estrategias en conjunto son denominadas regularización.

Ian Goodfellow define regularización como "cualquier modificación que hacemos a un algoritmo de aprendizaje con la intención de reducir su error en el conjunto de prueba, pero no en el de entrenamiento". Si se hace correctamente, estas regularizaciones mejoran la precisión del algoritmo. Adicionalmente, muchas veces se utiliza para manifestar un conocimiento previo en la base de datos.

En el aprendizaje profundo, muchas estrategias se basan en aumentar el sesgo con el fin de disminuir la varianza. A continuación, veremos algunos métodos empleados para lograr esto.

Incrementando la base de datos con datos repetidos

La mejor manera para que nuestro modelo de aprendizaje de máquina tenga menor varianza es entrenándolo con una mayor cantidad de datos. No obstante, sabemos que nuestros datos son limitados. Por ello, para solucionar este problema, un método que puede funcionar es crear datos agregarlos al conjunto de entrenamiento; por ejemplo, en el caso de las imágenes, es sencillo. Mover la imagen ligeramente a cualquier dirección, podría ayudar a que nuestra red aprenda mejor a reconocer caras. Esto es, porque el rostro de un individuo no importa donde este, sino que esté.

Con este método el algoritmo tendrá una mayor oportunidad de entender bien que representa un rostro; sin embargo, también se podría concentrar en aspectos muy particulares de este ejemplo duplicado. A esto nos referimos con aumentar el sesgo pero reducir la varianza.

Cabe notar, que hay casos en donde este tipo de métodos no funcionan y que incluso puede hacer que el modelo funcione peor. Por ejemplo, cuando tenemos un modelo exponencial y queremos estimar su parámetro, añadir datos puede hacer que el parámetro sea mas difícil de estimar. La imagen abajo es un claro ejemplo.

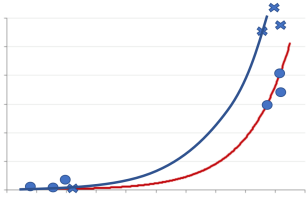

Figura 3.3: Ejemplo de una exponencial donde las cruces son datos añadidos y los círculos son los datos originales. Se puede ver que muchos de los datos añadidos están en el extremo y por ello la función cambia. Aunque puedan ser consecuencias naturales de la distribución, al irse tanto al extremo es difícil saber el parámetro a estimar y por ello, la función roja que es la original se mueve por la azul, que contiene un error mayor.

Por ello, a pesar de ser un buen método de regularización, debemos tener cuidado y ver que efectivamente nuestro modelo mejore al obtener una mayor cantidad de datos.

Bagging

Bagging (o boostrap aggregating) es una técnica para reducir el error de generalización al combinar varios modelos. La idea es entrenar varios modelos separados y después hacer que todos los modelos voten por la respuesta en los conjuntos de prueba.

La razón por la cual esto funciona es que varios modelos distintos normalmente no cometerán el mismo error en el conjunto de prueba. Aquellas técnicas que utilizan esta estrategia se denominan métodos ensambladores.

Consideremos por ejemplo un conjunto de k modelos de regresión. Asumamos que cada modelo tiene un error ϵ_i en cada ejemplo, con los errores obtenidos de una distribución normal multivariada con media cero, varianza $E[\epsilon_i^2] = v$ y covarianza $E[\epsilon_i, \epsilon_j] = c$. El error cuadrático esperado es entonces:

$$E\left[(\frac{1}{k}\sum_i \epsilon_i)^2\right] = \frac{1}{k^2}E\left[\sum_i(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j)\right] = \frac{1}{k}v + \frac{k-1}{k}c$$

Vemos pues, que en el caso donde los modelos están perfectamente correlacionados y que $c = v$, este método no ayuda. Sin embargo, cuando los errores son completamente distintos y c = 0, el error disminuye a tan solo $\frac{1}{k}v$. Esto significa que el error cuadrático medio disminuye linealmente con el tamaño del ensamblador. También significa que el ensamblador será tan bueno como el mejor de sus modelos y que si los modelos tienen errores independientes, el ensamblador hará mejor que todos los modelos.

Bagging específicamente, construye k diferentes conjuntos donde cada conjunto tiene el mismo número de elementos que el conjunto total de datos, pero estos se obtienen con reemplazo de la base de datos original. Esto

significa que muy probablemente a cada conjunto le falte algún elemento de la base de datos original y también pues, tenga elementos repetidos. Una vez hecho esto, el modelo i entrena con el conjunto i. Al haber diferencia en los conjuntos, normalmente los modelos van a predecir cosas distintas.

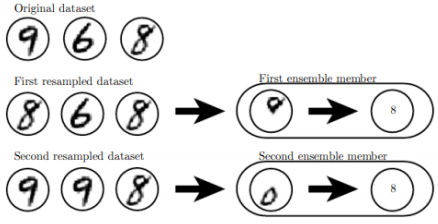

Figura 3.4: Ejemplo de bagging para la detección de números. (imagen obtenida de [13]).

Dropout

Un factor importante para utilizar un modelo de aprendizaje de máquina es el costo computacional. Cuando tienes bases de datos muy grandes o debes hacer predicciones en poco tiempo, es importante saber el tiempo que tarda tu modelo en predecir y solo utilizarlo si tienes el tiempo necesario. Dado esto, el método de bagging puede llegar a ser poco práctico, dado su alto costo computacional. Por ello, el método de *droput* es empleado como un sustituto de bagging. Dropout es un modelo de regularización que a grandes rasgos, procura hacer el trabajo de bagging con un costo computacional mucho menor.

Como vimos con bagging, definimos k modelos distintos, construimos k bases de datos obteniendo muestras de nuestro conjunto de entrenamiento con reemplazo y después entrenamos el modelo i en el conjunto i. Para entrenar con dropout, usamos un algoritmo de aprendizaje basado en el minibatch (ver [13]) que hace pequeños pasos, como por ejemplo el descenso por gradiente estocástico. En este caso, por ejemplo, minibatch significa que el gradiente es calculado a través de todo el conjunto i antes de actua-

lizar los pesos.

Cada vez que cargamos un ejemplo a un minibatch, muestreamos aleatoriamente una máscara binaria para aplicar a todos los datos de entrada y las capas ocultas en la red. La máscara para cada unidad es tomada independientemente de todas las otras. La probabilidad de que la máscara tenga el valor de 1 es un parámetro fijo que se declara previamente a que inicie el entrenamiento. Esta máscara nos indica si el valor va a ser incluido o no. Normalmente la probabilidad dada para los datos de entrada es 0.8 y para las capas ocultas es de 0.5.

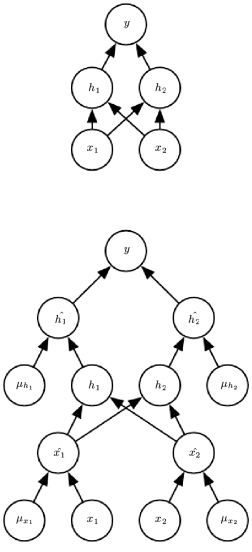

Figura 3.5: Ejemplo del algoritmo de dropout, cambiando la red de arriba por una red con los parámetros adicionales creados abajo. (imagen obtenida de [13]).

Viéndolo de una manera más formal, asumamos que tenemos un vectór de máscaras μ que nos dice qué unidades incluir y tenemos una función de costos $J(\theta, \mu)$, donde θ son los parámetros. En este caso, el dropout consiste en minimizar $E_\mu J(\theta, \mu)$.

Normalmente en el caso de bagging, los modelos son independientes. Con el dropout, los modelos comparten parámetros y cada modelo hereda un subconjunto diferente de parámetros de la red neuronal. En bagging buscamos que el modelo converja en su respectivo conjunto de entrenamiento, en dropout, muchos modelos no son entrenados explícitamente, sino que una pequeña fracción de las subredes son entrenadas para un solo paso y los parámetros que comparten causan que las subredes restantes tengan buenos ajustes de los parámetros.

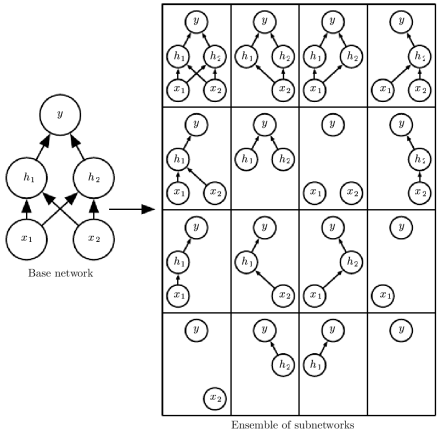

Figura 3.6: Ejemplo de las subredes generadas por el dropout. Se entrena un subconjunto de estas y los parámetros de las restantes se ajustan con base en estos resultados. (imagen obtenida de [13]).

Para hacer una predicción, el ensamblador debe acumular votos de todos los miembros, la predicción del ensamblador esta dado por la media aritmética de todas las distribuciones,

$$\frac{1}{k}\sum_{i=1}^{k} p^i(y|x)$$

En el caso del dropout, cada submodelo definido por la máscara de vec-

tores μ define una probabilidad de distribución $p(y|x, \mu)$. La media aritmética de todas las máscaras esta dada por:

$$\sum_{\mu} p(\mu)p(y|x, \mu)$$

donde $p(\mu)$ es la distribución de probabilidad que fue empleada para muestrear μ a la hora de entrenar. Dado que la suma incluye un número exponencial de términos, es muy complicada de evaluar. En vez de ello, aproximamos la inferencia con muestras, promediando la salida de varias máscaras.

Una ventaja del dropout es que es muy "barato"(hablando de costo computacional). Utilizar dropout en el entrenamiento tiene un costo computacional de $O(n)$, por ejemplo, por actualización. Adicionalmente, es un método que funciona con cualquier tipo de modelo o de conjunto entrenamiento.

Redes neuronales convolucionales

Retomando el prefacio, las redes neuronales convolucionales son un tipo especial de redes neuronales para procesar datos que tienen una topología tipo cuadrícula. Esto es por ejemplo, una imagen que se puede ver como una cuadrícula de píxeles en segunda dimensión.

El nombre *convolucional* indica que esta red utiliza una operación denominada convolución. La convolución es un tipo de operación lineal. La definición de Ian Goodfellow de las redes neuronales convolucionales es (obtenido de [13]): *"Redes neuronales que utilizan convolución en lugar de multiplicaciones de matrices generales en una de sus capas."*

Convolución

Para ejemplificar como funciona la convolución, tomaremos el ejemplo dado en el libro de Deep Learning de Ian Goodfellow (ver [13]): "Asuma-

mos que queremos seguir la ubicación de una nave espacial mediante el uso de un láser sensorial. Nuestro láser nos da un solo dato, $x(t)$, que es la ubicación de la nave espacial en el tiempo t. Tanto x como t son valores reales, entonces podemos obtener información de la ubicación de la nave de manera instantánea en cualquier momento dado."

"Ahora asumamos que nuestro sensor tiene ruido en la información que esta entregando. Para que el ruido sea menor, tomaremos un promedio de varias tomas de información. Así mismo, le daremos mayor peso a los datos más recientes. Los pesos se darán con una función $w(a)$, donde a es el momento en el cual el valor fue obtenido. Aunando esta operación a la función previa, podemos obtener una nueva función s que nos da una estimación de la ubicación de la nave espacial:

$$s(t) = \int x(a)w(t-a)da$$

Esta operación es denominada convolución."

En general, la convolución esta definida para cualquier función cuya integral esté definida y podría utilizarse para otro objetivo que no sea obtener el peso promedio. El primer argumento de la función x, es normalmente conocido como *input* (dato de entrada) y el segundo w, como el *kernel*. Así mismo, el resultado se conoce como *mapa de atributos*.

Regresando a nuestro ejemplo, al trabajar con datos en la computadora, el tiempo se vuelve discreto y los sensores darían información en intervalos regulares. Para efectos del ejercicio, se asume que el láser nos daría información cada segundo. Con esto, el tiempo t solo puede tener valores enteros. Si asumimos que x y w están definidos solo para el tiempo t, podemos definir la convolución discreta como:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Normalmente en problemas aplicados, el input es un arreglo multidimensional de datos y el kernel es un arreglo multidimensional de parámetros

que son adaptados por el algoritmo de aprendizaje. Estos arreglos multidimensionales son denominados tensores.

Si quisiésemos utilizar nuestro modelo en una imagen de dos dimensiones con tanto el input como el kernel también en dos dimensiones, nuestra operación de convolución quedaría así:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum nI(m,n)K(i-m, j-n)$$

No obstante, muchas bibliotecas de redes neuronales (incluyendo las que utilizaremos) implementan una función conocida como correlación cruzada, la cual es una modificación de la operación previa:

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

Esto permite que cuando m y n aumenten, las coordenadas del input aumentan pero el kernel quede igual, con esto tendremos menos variación en el rango de valores válidos para m, n; permitiéndonos implementarlo con mayor facilidad.

Pooling

Una capa convolucional está conformada por tres etapas:

- La capa hace varias convoluciones en paralelo para producir activaciones lineales.
- Cada activación lineal corre a través de una activación no lineal, como por ejemplo la activación RELU. Esta etapa es conocida como etapa de detección.
- Utilizamos una función de *pooling* para modificar la salida de esta capa.

El pooling reemplaza la salida de la red en una vecindad con un resumen estadístico de salidas cercanas. Por ejemplo, la operación de *max pooling* (la cual emplearemos múltiples veces en nuestras redes), obtiene

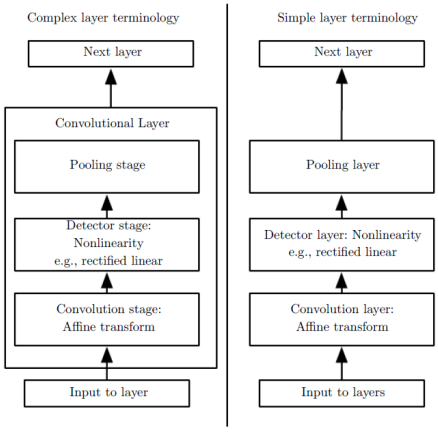

Figura 3.7: Dos terminologías para describir una capa convolucional (Imagen obtenida de [13]).

el máximo valor dada una vecindad rectangular. Otra función de pooling podría ser tomar el promedio de dicha vecindad o también el promedio ponderado basado en la distancia del píxel central.

En todos los casos, la función de pooling te permite hacer una representación con menor varianza a pequeños cambios en los datos de entrada. Esto es muy útil cuando no estamos buscando una característica de una imagen, sino que la característica esté en la imagen. Por ejemplo, podemos buscar un rostro, sin importar en que parte de la imagen se encuentre; el pooling puede verse como la agregación del conocimiento previo de que los datos deben de ser invariantes a pequeños cambios. Al ser esto cierto, nuestro nivel de predicción aumenta fuertemente.

Adicionalmente, dado que esta técnica resume las respuestas de un vecindario entero, el número de elementos en la etapa de detección será reducido y por tanto, al tener menos datos, la cantidad de operaciones posteriores es menor y por ello, el costo computacional disminuye. El reducir el número de elementos es conocido como *downsampling*.

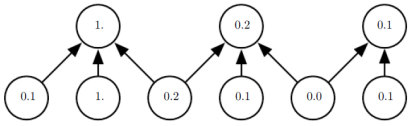

Figura 3.8: Ejemplo de max pooling con downsampling.

Otras capas

Además de las capas que conocemos, hay un par de capas adicionales que utilizaremos y que es necesario que recalquemos: la capa de aplanado (*flattening layer*) y la capa completamente conectada (explicaciones obtenidas de [23]).

- La capa de aplanado convierte la salida de la capa previa en un vector de una dimensión.
- La capa completamente conectada es una capa conectada a todas las capas anteriores que toman un volumen de entrada y su salida es un vector de N dimensiones, donde N es el numero de clases el programa debe escoger de.

Transformación de datos

En este trabajo se propone una transformación de datos de problemas en los que usualmente no se aplican redes neuronales convolucionales en una imagen que pueda ser procesada mediante una red como las mencionadas. Esto es posible mediante una función que transforme los datos en imágenes.

Para ello, vayámonos a la definición de *función*: sean dos conjuntos X y Y dados y asumamos que para cada elemento de $x \in X$ le corresponde un elemento $y \in Y$, denotado por $f(x)$. Uno entonces dice que una función f esta dada en X (y también que la variable y es una función de la variable x, o que y depende de x) y escribimos $f : X \to Y$.

Para nuestro modelo, lo que queremos crear es una función f que convierta los datos actuales D en una especia de imagen I, esto es $f : D \to I$

que permita resolver el problema con una red neuronal convolucional. Para esto, lo primero que necesitamos es que la función sea *inyectiva*, ya que queremos que cualquier dato $d \in D$ tengamos solamente un valor $i \in I$ distinto a cualquier otro dato de D. Ahora que sabemos como debe ser nuestra función, necesitamos entender como son nuestros datos y como se ve la imagen que queremos procesar.

La idea contemplada para transformar los datos consiste en convertir cada elemento de la matriz en un píxel. Esto sería de la siguiente forma: sea la matriz de datos $D^{n \times m}$, queremos que cada elemento $d_{i,j}$ se convierta en un píxel $i_{i,j}$. Para ello, tomaremos cada columna i y la estandarizaremos con valores entre 0 y 255, que representan uno de los tres colores del píxel.

Una vez que hacemos esto, tendremos una matriz con valores entre 0 y 255 que representan un color del píxel. Sin embargo, un píxel esta representado por tres colores y por lo tanto debemos de obtener dos matrices mas con estos valores. Por ello, replicamos los valores de manera idéntica en la matriz, por ejemplo si teníamos un vector $(0, 1, 0)$, se convertiría en un vector con tres vectores: $[(0, 1, 0), (0, 1, 0), (0, 1, 0)]$ donde el primer vector representa la escala de rojos, la segunda de verdes y la última de azules, combinando los colores como se hace en una imagen. Esto tiene la ventaja de que los colores exactos no se repetirán excepto para los mismos datos datos en una columna, lo cual tiene sentido dado que la columna representa un parámetro y no queremos que para un mismo parámetro con valor idéntico se den colores distintos. Adicionalmente, dado que se estandarizó por columna, los colores entre parámetros debería de cambiar fuertemente ya que una ligera modificación cambiará toda la combinación de colores.

Esta función también tiene una fuerte desventaja: la imagen creada de esta manera se verá como una serie de lineas de distintas tonalidades, claramente separadas unas de otras, pero con una separación relativamente aleatoria que no representa nada en particular. Por ejemplo, un parámetro con 7 valores tendrá 7 tonos distribuidos según los valores, el cambio a un píxel que representa a otro parámetro de digamos 15 valores, contará con cambios probablemente fuertes de color entre ellos, pero no por eso significa que los parámetros sean muy distintos. Adicionalmente, parámetros con

intervalos de valores similares tenderán a tener colores similares.

No obstante, dado que las pruebas realizadas con esta función fueron exitosas y el objetivo es encontrar un solo caso de éxito, no se vio necesidad de crear otras funciones.

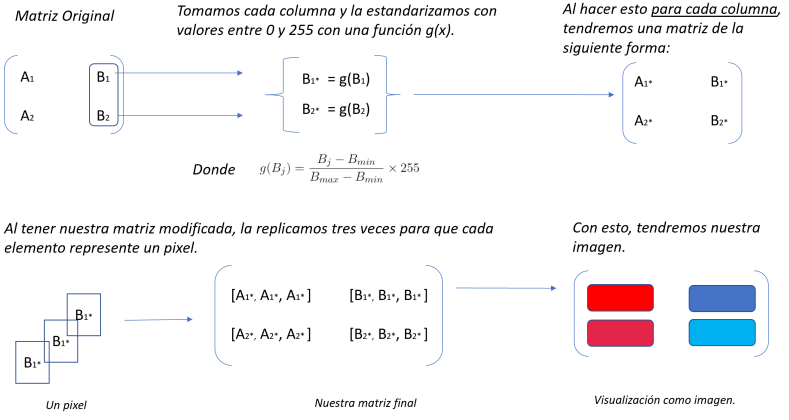

Figura 3.9: Explicación gráfica de la transformación de los datos.

Capítulo 4

Implementación y comparativa

Recordando nuestra hipótesis, creemos que las redes neuronales convolucionales pueden resolver problemas supervisados generales que no son lattices ni imágenes haciendo un mapeo topológico de las bases de datos y convirtiéndolas en una especie de "pseudo imagen", la máquina pueda detectar patrones y predecir con base en ellas.

En el capítulo previo, explicamos la contribución central: la transformación de un problema general en una imagen. Este capítulo se enfoca en las pruebas realizadas con redes neuronales convolucionales. En él, primero mostramos cómo se implementan las redes neuronales convolucionales; posteriormente, hacemos pruebas preliminares con datos generados para ver que vale la pena hacer la comparativa desde un inicio, comparando varios tipos de redes convolucionales (las capas varían entre unas y otras) y corroborando que las transformaciones de datos propuestas en el capítulo anterior sean útiles. Una vez terminadas estas secciones, procederemos a utilizar varias bases de datos disponibles al público. Las modificaremos y utilizaremos los algoritmos propuestos previamente, con el objetivo de mostrar que nuestra hipótesis es válida.

Digitos del MNIST con Tensorflow

Como ejemplo, utilizamos la biblioteca de Tensorflow para hacer una red neuronal con regresión Softmax para clasificar los dígitos del MNIST escritos a mano. El MNIST es una base de datos que contiene imágenes de dígitos escritos a mano. El código empleado se encuentra en el apéndice [1] y [3].

Tensorflow es una biblioteca que nos permite crear un conjunto de operaciones interactivas que se ejecutan completamente fuera de Python para disminuir el costo computacional.

Con esta biblioteca hicimos una prueba para predecir con una base de datos aleatoriamente generada. Creamos 55,000 diferentes funciones de distribución normal, cada una con 784 dimensiones, con media 0 y desviación estándar del 1 al 10 (y la desviación estándar es lo que queremos predecir). El código para la creación de los datos y la implementación se encuentran en el apéndice [2]. Cabe notar que se utilizó el método de validación cruzada para tanto este modelo como para los demás puestos posteriormente, con proporción 80-20.

Los resultados fueron prometedores. Al intentar predecir, vimos inicialmente que hacer una red neuronal con Softmax, no sirvió; con ella, tuvimos una predicción correcta del 17 por ciento. Sin embargo, al utilizar una red neuronal convolucional, logramos un 93 por ciento de predicción. Aunque sea una prueba sencilla, es una prueba que nos hace ver que efectivamente puede haber una buena predicción con las redes neuronales convolucionales, que no todos los algoritmos pueden lograr. Dado esto, decidimos continuar con las pruebas con bases de datos reales para poder entender qué tan bien funciona realmente.

Keras

Para las próximas bases de datos, se decidió utilizar la biblioteca denominada Keras. Esto se debe a que era más sencillo adaptar las bases de datos y correrlas con esta biblioteca.

Keras es una interfaz de programación de aplicaciones de redes neuronales escrita en Python y corre encima de TensorFlow. Fue desarrollada de tal manera que se enfocó en permitir experimentar de manera rápida. La estructura de Keras es un modelo, una manera para organizar capas de una red neuronal. El modelo más sencillo de Keras es el modelo *sequential*, el cual es una pila lineal de capas. Este es el modelo que utilizaremos para crear nuestra red.

El problema de CIFAR - 10

El problema de identificar automáticamente objetos en fotografías es difícil dado el número tan grande de permutaciones entre los objetos, posiciones, iluminación, etcétera. Para una mejor comprensión de este tipo de problemas explicaremos el problema de CIFAR-10 ya que nos permite mostrar las capacidades del aprendizaje profundo. Adicionalmente, permitirá al lector ver el funcionamiento de la biblioteca en una base de datos que tiene resultados favorables y bien documentados.

CIFAR-10 es una base de datos creada por el Canadian Institute for Advanced Research que consiste en 60,000 fotos divididas en 10 clases. Las clases consisten en objetos comunes como aviones, automóviles, pájaros, gatos, etc. En ella, 50,000 imágenes son utilizadas para el entrenamiento del modelo y las 10,000 restantes para evaluar su precisión. Las fotos están en color rojo, verde y azul; cada imagen esta dado por cuadrados de 32x32 píxeles (el código para visualizar un pequeño extracto de imágenes de CIFAR-10 se puede ver en el apéndice [3]). Imprimiremos unas cuantas imágenes para comprender mejor la base.

Cada imagen esta representada como una matriz tridimensional, con dimensiones para rojo, verde, azul, grosor y altura. Crearemos un modelo sencillo de redes ñneuronales convolucionales para solucionar el problema.

Diseñaremos ahora una red neuronal convolucional para resolver el problema. La arquitectura quedara como sigue:

Figura 4.1: Algunas imágenes del CIFAR-10 (Imágen obtenida de [6]).

- Capa convolucional con función de activación ReLu.
- Capa Dropout en 20 por ciento.
- Capa convolucional con función de activación ReLu.
- Capa Max Pool
- Capa convolucional con función de activación ReLu.
- Capa Dropout en 20 por ciento.
- Capa convolucional con función de activación ReLu.
- Capa Max Pool.
- Capa convolucional con función de activación ReLu.
- Capa Dropout en 20 por ciento.
- Capa convolucional con función de activación ReLu.
- Capa Max Pool.
- Capa de aplanado.
- Capa Dropout en 20 por ciento.
- Capa completamente conectada.
- Capa Dropout en 20 por ciento.
- Capa completamente conectada.

- Capa Dropout en 20 por ciento.
- Capa completamente conectada.

Corriendo el código que se encuentra en el apéndice [4], vemos un 80 por ciento de precisión.

Ahora que entendemos como funcionan las redes neuronales convolucionales con el CIFAR-10 en Keras, implementaremos redes neuronales convolucionales con Keras en problemas que no sea de imágenes, y veremos como funcionan.

"Recipe for disaster": Base de datos del Titanic

La primera base de datos elegida es la base de datos de Kaggle: *Titanic: Machine Learning from Disaster*. Esta base esta muy bien documentada y será un buen lugar donde empezar; contiene toda la información de los pasajeros que abordaron el Titanic y se debe predecir si sobrevivirán o no. Trataremos de adaptar la base de datos para resolver este problema con diferentes métodos de aprendizaje supervisado y comparar estos resultados con las redes neuronales convolucionales.

Cabe hacer notar que, los códigos por subsección se encontrarán en el apéndice. Recomendamos consultar el código conforme se vean los resultados.; asumiremos conocimiento de ellos e iremos directamente a la descripción, modificación e implementación de los métodos.

Descripción de los datos

La base de datos del Titanic consiste en 12 variables:

- PassengerID: Variable categórica, la cual cuenta con las claves únicas de cada pasajero. Estas claves empiezan en 1 y terminan en 891.
- Survived: Variable categórica que consta de dos elementos, 0 y 1. El 0 indica que el pasajero no sobrevivió; el 1 que sí. Esta es nuestra variable más importante, ya que es la que queremos predecir.

- PClass: Variable categórica con 3 elementos (1,2,3), es la clase en la cual se encontraba cada pasajero. Primera clase es más cara que segunda y esta más cara que tercera.
- Name: Variable categórica que nos dice el nombre de cada pasajero. Esta variable es única para cada pasajero y no nos aporta realmente más información. Tanto esta como passengerID son variables que no son útiles para la predicción pero sí para hacernos una idea de quienes eran las personas. Una buena idea sería comparar los títulos de cada pasajero para ver si de ahí podemos obtener una nueva variable.
- Sex: Variable categórica que indica el sexo del pasajero.
- Age: Variable continua que indica la edad del pasajero.
- SibSp: Variable discreta, nos dice el número de hermanos y esposas que estaban a bordo del crucero con el pasajero.
- Parch: Variable discreta, nos dice el número de padres e hijos que estaban a bordo del crucero con el pasajero. Probablemente sea buena idea unir esta y la variable previa en una para así determinar el tamaño de la familia.
- Ticket: Número de ticket del pasajero. Variable categórica. Mezcla de numérica y alfanumérica. Dado que los tickets no se repiten, no se nos ocurre como ayudaría en la predicción.
- Fare: Precio pagado por el pasajero por el viaje (¿podría ser que los que pagan más sobreviven con mayor facilidad?).
- Cabin: Cabina en donde se encontraba el pasajero. Es una variable alfanumérica. Se nos presenta la misma situación que con Ticket.
- Embarked: Variable categórica que nos dice de donde embarco cada pasajero.

Vemos pues, que la variable survived es aquella que es nuestra variable dependiente. Adicionalmente, vemos que hay 4 variables con elementos del tipo int, 2 del tipo float y 5 del tipo object. Esto es importante a la hora

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

de hacer la predicción ya que tendremos que ajustarlos para poder predecir.

Descripción cada uno de los parámetros.

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

	Name	Sex	Ticket	Cabin	Embarked
count	891	891	891	204	889
unique	891	2	681	147	3
top	Behr, Mr. Karl Howell	male	347082	G6	S
freq	1	577	7	4	644

Algunas cosas importantes e interesantes que vemos aquí son por ejemplo que el 38.38 % de los pasajeros sobrevivieron, que el costo promedio es de 32.2 pero que el más caro fue de 512 (la moneda no esta especificada pero probablemente eran dólares). Así mismo, la edad promedio es

de 29 años. También sabemos que aquí hay elementos vacíos ya que los percentiles de edad no se podían ver, esto es algo que tendremos que corregir posteriormente. Finalmente, la mitad de los pasajeros viajaron con hermano o esposa y el 38 % con hijos.

Esta es la descripción de los datos categóricos. Es interesante porque podemos obtener cierta información preliminar, como el que predominaron hombres en la embarcación (577 hombres de 891 pasajeros) y que la mayoría de las personas embarcaron en 'S'.

Análisis gráfico de variables auxiliares con la dependiente.

Trataremos de visualizar la relación de las variables auxiliares con la dependiente.

Edad

Vemos que los menores de 5 años tienen una gran probabilidad de sobrevivir. Así mismo, todas las personas de 80 años o más sobrevivieron. No obstante, la distribucion en las edades intermedias paracen ser bastante similares en ambas.

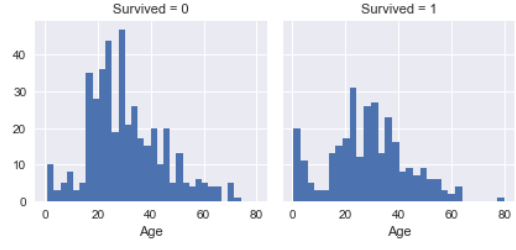

Figura 4.2: Edad de pasajeros con respecto a la cantidad de sobrevivientes.

Número de familiares

Creamos la variable nfamily que indica el número de familiares que tiene un pasajeron a bordo. Vemos que la personas que más sobrevivieron tenían uno o dos miembros de su familia a bordo.

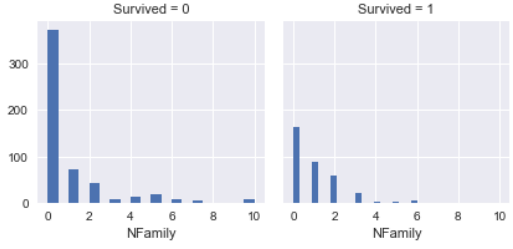

Figura 4.3: Número de familiares de un pasajero a bordo con respecto a cantidad de sobrevivencia.

Cuota pagada

Los que pagaron más tenian mayor probabilidad de salvarse.

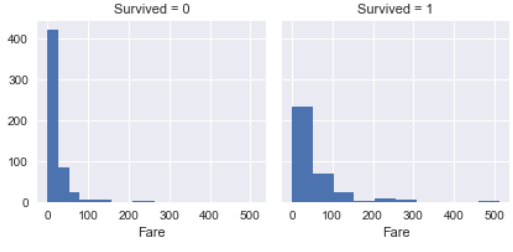

Figura 4.4: Cuota pagada de un pasajero a bordo con respecto a cantidad de sobrevivencia.

Clase

Aquí vemos claramente que las personas en primera clase sobrevivieron

con mayor probabilidad que los de tercera.

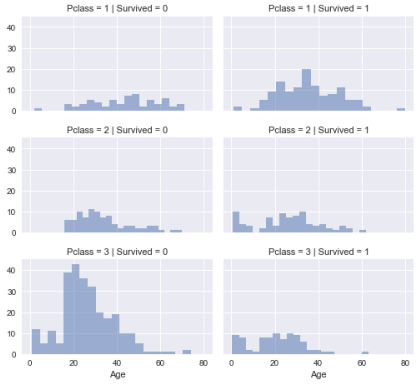

Figura 4.5: Sobrevivientes por clase en la que fueron.

Dependencias intrínsecas entre variables.

Adicionalmente, buscamos el comportamiento de la variable dependiente con algunas variables que creemos pueden tener impacto significativo. Esto es importante para ver las correlaciones que tienen los datos y decidir cuales son significativos y cuales no.

Vemos de manera evidente que los parámetros son significativos. No hay necesidad de descartarlos.

Arreglando los datos

Como pudimos observar en la visualización y análisis gráfico de los datos, estos no están en el estado óptimo que desearíamos tenerlos. Por ello, debemos de editarlos e inclusive quitar algunos para mejorar la eficiencia en la predicción (código en apéndice [6]).

Empezaremos quitando las variables *Ticket* y *Cabin*. Esto se debe a que los valores de ambas son raramente compartidas entre las observacio-

	Pclass	Survived
0	1	0.629630
1	2	0.472826
2	3	0.242363

	Embarked	Survived
0	C	0.553571
1	Q	0.389610
2	S	0.336957

	NFamily	Survived
3	3	0.724138
2	2	0.578431
1	1	0.552795
6	6	0.333333
0	0	0.303538
4	4	0.200000
5	5	0.136364
7	7	0.000000
8	10	0.000000

	Sex	Survived
0	female	0.742038
1	male	0.188908

Figura 4.6: Probabilidad de sobrevivencia con respecto al sexo, clase, embarcación y número de familiares

nes y por tanto no ayudan a predecir. A continación, queremos crear nuevas características utilizando los parámetros ya conocidos. Lo que haremos es tomar los títulos de los nombres y ver si hay una correlación entre estos y la variable *survival* antes de quitar la variable Name. Dada la cantidad de nombres distintos, creemos que es una buena idea juntar los que no se repiten mucho y nombrarlos "Otros". Intentemos nuevamente con estas modificaciones.

	Title	Survived
0	Master	0.575000
1	Miss	0.702703
2	Mr	0.156673
3	Mrs	0.793651
4	Otro	0.347826

Vemos entonces que sí parece haber una correlación con los títulos. Por lo tanto, nos quedaremos con ellos. Ahora, crearemos la variable *FamilySize* que nos indicará la cantidad de personas con las que venía el jefe de familia. Nuevamente, dada la cantidad de valores distintos, creemos que es una buena idea juntar algunos. Se dividió en tres grupos: los que vienen solos, los que vienen con menos de tres y los que vienen con más de tres.

Esto es porque vemos cierta relación entre ellos.

	Family	Survived
0	0	0.300725
1	1	0.578767
2	2	0.148936

Efectivamente vemos que funciona, por lo tanto quitaremos las variables con las que lo construimos y nos quedaremos solo con "Family".

La variable *Embarked* también debemos convertirla a numérica. Sin embargo, tenemos elementos no existentes en esta columna. Para solucionar esto, sustituiremos los valores no existentes por la embarcación más probable.

Para finalizar, debemos evitar tener valores faltantes o nulos en nuestra base. Para ello nos concentraremos en dos columnas, *Age* y *Fare* que son las que aún tienen valores faltantes.

Es común que se tenga, como en nuestro caso, una base donde hay observaciones incompletas o faltantes, se puede elegir entre eliminarlas, llenarlas con ceros o inputar datos. Esta última opción es la que desarrollaremos más, pues tiene consecuencias importantes en el análisis y predicción. Se utiliza un modelo de articulación bayesiana no paramétrico para análisis multivariante continuo y categórico, que fusiona mezclas de procesos de Dirichlet de distribuciones multinomiales para variables categóricas o mezclas de procesos de Dirichlet de distribuciones normales multivariadas para variables continuas. El modelo se encuentra en el apéndice [9].

Para incorporar la dependencia entre las variables continuas y categóricas se hace un modelado de las medias de las distribuciones normales como funciones de componentes específicos de las variables categóricas y la formación de componentes distintos de la mezcla para datos continuos con

probabilidades que se enlazan a través de un modelo jerárquico. Esta estructura permite al modelo capturar dependencias complejas entre los datos categóricos y continuos con un ajuste mínimo.

Para implementarlo, vamos a dividir los datos en intervalos con la misma cantidad de variables entre ellas. Esto nos creará intervalos más pequeños para las zonas con mucha información e intervalos mucho más grandes para las zonas con poca información. Por ello, cuando reemplazemos los valores faltantes con un valor, este será más cercano a las zonas con poca información y así evitamos fomentar sobre ajustes. Usaremos estos intervalos en vez de *Age*, ya que las edades como tal no serían tan útiles para predecir por la cantidad que son.

Veamos como queda nuestra base de datos al final:

	PassengerId	Survived	Pclass	Sex	Age	Fare	Embarked	Title	Family
0	1	0	3	0	1.0	0	0	1	1
1	2	1	1	1	2.0	3	1	3	1
2	3	1	3	1	1.0	1	0	2	0
3	4	1	1	1	2.0	3	0	3	1
4	5	0	3	0	2.0	1	0	1	0
5	6	0	3	0	0.0	1	2	1	0
6	7	0	1	0	3.0	3	0	1	0
7	8	0	3	0	0.0	2	0	4	0
8	9	1	3	1	1.0	1	0	3	1
9	10	1	2	1	0.0	2	1	3	1

Con esto podemos pasar al modelado y predicción.

Implementación de RNNs convolucionales y comparativa

Una vez que la base de datos estuvo lista, hicimos predicciones con diferentes métodos para ver como las redes neuronales convolucionales funcionan en comparación. Para esto, implementamos las redes neuronales convolucionales en la base de datos del Titanic, junto con varios métodos adicionales de aprendizaje supervisado. Así, la estructura de la red convolucional pequeña queda como sigue:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_21 (Conv2D)           (None, 32, 7, 32)         896
_________________________________________________________________
dropout_21 (Dropout)         (None, 32, 7, 32)         0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 32, 7, 32)         9248
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 32, 3, 16)         0
_________________________________________________________________
flatten_11 (Flatten)         (None, 1536)              0
_________________________________________________________________
dense_21 (Dense)             (None, 512)               786944
_________________________________________________________________
dropout_22 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_22 (Dense)             (None, 2)                 1026
=================================================================
Total params: 798,114
Trainable params: 798,114
Non-trainable params: 0
```

Figura 4.7: Descripción de la red neuronal convolucional.

Adicionalmente, los resultados de nuestros modelos quedaron como sigue (el código obtenido para los resultados se encuentra en los apéndices [7] y [8].):

Algoritmo	Precisión
Máquina de soporte vectorial	78.68 %
Árboles de decisión	78.13 %
Redes neuronales con feedforward	65.25 %
Bosques aleatorios	80.49 %
Redes neuronales convolucionales simples	79.94 %
Redes neuronales convolucionales densas	67.88 %

Cabe notar que se utilizó validación cruzada para cada uno de los modelos, con el conjunto de prueba del 80 %. Las máquinas de soporte vectorial tuvieron un kernel lineal y la penalización del error C, se instanció como 0,25. Los árboles de decisión C4.5 tuvieron una profundidad máxima de 16 nodos como tope. Adicionalmente, se utilizó bagging para este método con un número máximo del 85 % de los datos como muestra y de igual manera un 85 % de los atributos como un máximo de atributos. Los demás parámetros fueron los de facto de las bibliotecas correspondientes de sklearn. Estos parámetros de mantuvieron fijos para todas las comparativas.

Vemos pues, que nuestro modelo sencillo de redes neuronales convolucionales tiene resultados muy buenos, siendo solo el bosque aleatorio quien tiene mejores predicciones y este solo es mejor por menos de 1 %. El modelo profundo probablemente tenga un sobre ajuste y por ello no tuvo tan buenos resultados, debido a que son demasiadas capas. No obstante, podemos decir que tenemos nuestra primera aplicación de redes neuronales convolucionales existosa.

Ahora, usaremos otras bases de datos para mostrar que no fue un caso aislado, sino que efectivamente los resultados son consistentes y por tanto, prometedores.

Detección del sexo de abulones

La base de datos de abulones fue creada por Warwick J Nash, Tracy L Sellers, Simon R Talbot, Andrew J Cawthorn y Wes B Ford en 1994. Esta consiste en las medidas de los abulones y su sexo. El objetivo es predecir el sexo de los abulones desde sus medidas fisícas y la edad. Esta base de datos es bastante compleja y las mejores predicciones llegan a 60 % de precisión (ver [11]). Por ello, no esperamos resultados espectaculares, pero si suficientes para competir con otros métodos.

Los datos contenidos son:

- Sexo: M,F,I (infante)
- Length: Tamaño de la concha desde los puntos más distantes a esta.
- Diameter: Diámetro, perpendicular al tamaño.
- Height: Altura del abulón.
- Whole weight: Peso de todo el abulón.
- Shucked weight: Peso de la carne del abulón.
- Viscera weight: Peso de las entrañas del abulón (sin sangre).
- Shell weight: Peso después de secarse el abulón.
- Rings: Número de anillos dentro de la concha.

	0	1	2	3	4	5	6	7	8
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
3	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
4	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Figura 4.8: Subconjunto de la base de datos de los abulones.

Dado que esta base de datos ya ha sido revisada y modificada, no es necesario hacer todo lo que se hizo en la base previa. Por ello, lo único que se modificó fue el sexo para que sean valores y utilizarlos para etiquetar. Los resultados fueron los siguientes:

Algoritmo	Precisión
Máquina de soporte vectorial	52.57 %
Árboles de decisión	52.09 %
Redes neuronales con feedforward	49.87 %
Bosques aleatorios	51.12 %
Redes neuronales convolucionales simples	50.21 %
Redes neuronales convolucionales densas	48.11 %

Nuevamente nuestras redes tienen una precisión cercana a los demás métodos, siendo las redes convolucionales simples las que obtienen el cuarto lugar, a diferencia del segundo que se obtuvo en la base anterior. A pesar de no haber tenido tan buenos resultados, la diferencia en precisión comparada con el mejor de los modelos que empleamos para predecir es tan solo del 2 %.

Estrellas pulsares

Esta base de datos, conocida como *HTRU2* es una base de datos que describe estrellas candidatas a ser estrellas pulsares y fue obtenida en el *High Time Resolution universe survey*, que fue la encuesta en alta resolución del universo (ver [19]).

Las estrellas pulsares son un tipo raro de estrellas neuronas que producen un radio de emisión detectable en la tierra. Conforme estas estrellas rotan, sus emisiones se disparan a través del espacio y cuando pasan por nuestra linea de visión, se produce un patrón detectable de emisiones radiales. Como este tipo de estrella rota rápidamente, esta emisión se repite periodicamente, permitiendo que estas estrellas se encuentren con relativa facilidad.

No obstante, cada estrella pulsar produce un patrón ligeramente diferente en sus emisiones, por lo cual la dificultad de encontrar una (aunada al ruido) aumenta drásticamente. Por esto, surge la necesidad de utilizar el aprendizaje de máquina.

La base de datos cuenta con 8 variables continuas y una categórica. La variable categórica es una variable binaria que representa si es una estrella pulsar o no. Los datos que tenemos son los siguientes:

- Mean of the profile: Promedio de los pulsos.
- Standard deviation of the profile: Desviación estándar de los pulsos.
- Excess kurtosis of the profile: Exceso de curtosis de los pulsos.
- Skewness of the profile: Asimetría de los pulsos.
- Mean of the Dispersion Measure curve: Promedio de la curva de dispersión.
- Standard deviation of the DM curve: Desviación estándar de la curva de dispersión.
- Excess kurtosis of the DM curve: Exceso de curtosis de la curva de dispersión.
- Skewness of the DM curve: Asimetría de la curva de dispersión.

En total, se tuvieron 17,898 ejemplos; de los cuales habían 1639 positivos y 16,259 negativos.

	Mean of the integrated profile	Standard deviation of the integrated profile	Excess kurtosis of the integrated profile	Skewness of the integrated profile	Mean of the DM-SNR curve	Standard deviation of the DM-SNR curve	Excess kurtosis of the DM-SNR curve	Skewness of the DM-SNR curve	target_class
0	140.562500	55.683782	-0.234571	-0.699648	3.199833	19.110426	7.975532	74.242225	0
1	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	0
2	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	0
3	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	0
4	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	0
5	93.570312	46.698114	0.531905	0.416721	1.636288	14.545074	10.621748	131.394004	0
6	119.484375	48.765059	0.031460	-0.112168	0.999164	9.279612	19.206230	479.756567	0
7	130.382812	39.844056	-0.158323	0.389540	1.220736	14.378941	13.539456	198.236457	0
8	107.250000	52.627078	0.452688	0.170347	2.331940	14.486853	9.001004	107.972506	0
9	107.257812	39.496488	0.465882	1.162877	4.079431	24.980418	7.397080	57.784738	0

Figura 4.9: Subconjunto de la base de datos de las estrellas pulsares.

Cada vez que hacemos las pruebas con un modelo, sobre todo en casos sin variables categóricas, corremos el modelo antes de iniciar con la modificación de los datos. Dado el buen resultado inicial que tuvimos, no vimos necesidad de modificar los datos en lo absoluto. Los resultados quedaron como sigue:

Algoritmo	Precisión
Máquina de soporte vectorial	97.56 %
Árboles de decisión	97.85 %
Redes neuronales con feedforward	96.61 %
Bosques aleatorios	97.71 %
Redes neuronales convolucionales simples	90.95 %
Redes neuronales convolucionales densas	90.95 %

Este ejemplo tuvo resultados sorprendentes, no solo la base de datos se dejó intacta, sino que los métodos tuvieron resultados muy positivos. Siendo nuestro método el que tuvo un nivel de predicción menor, aproximadamente el 7 %, nos dejo un poco insatisfechos. No obstante, es bueno saber que efectivamente esto no funcionará para todas las bases de datos.

Además, es muy interesante ver que ambas redes convolucionales quedaron con el mismo nivel de predicción, probablemente un mínimo local en el que ambos quedaron atorados.

Finalmente, un criterio importante es el costo computacional. Cuando los modelos utilizados para comparar tenían un tiempo aproximado de 15

segundos (excepto las máquinas de soporte vectorial, que tardaron alrededor de un minuto); las redes convolucionales simples tardaron aproximadamente 3 minutos y las densas 16 minutos. No obstante, dado que nuestro modelo no fue hecho en el rigor de los otros modelos, ni con el tiempo de estos, es comprensible que el costo computacional sea mayor.

Capítulo 5

Conclusiones e ideas posteriores

Resumiendo nuestros resultados, podemos ver que en la base del Titanic, nuestras redes quedaron en segundo lugar, con una diferencia muy pequeña entre los bosques aleatorios y estas. En la base de los abulones tuvo mejores resultados que las redes neuronales con feedforward, pero quedó detrás de los otros métodos. Finalmente, en las estrellas pulsares, tuvo resultados inferiores al resto, pero con niveles de predicción de aproximadamente el 90 %. Estos resultados fueron muy prometedores, sobre todo si tomamos en cuenta que los algoritmos que usamos para comparar también fueron cambiando de lugar. Por ejemplo, en Titanic el mejor algoritmo fue el de bosques aleatorios; en los abulones, fueron las máquinas de soporte vectorial y en las estrellas pulsares, los árboles de decisión. Esto es básicamente una manera intuitiva de mostrar que no hay un algoritmo superior, sino que cada algoritmo tiene mejor desempeño en algunas bases de datos; y que nuestro modelo puede formar parte de ellos.

Adicionalmente, cabe notar que el trabajo hecho es aún poco, y queda mucho que ver. Las transformaciones que hicimos a pesar de ser efectivas, no son óptimas. Es probable que haya maneras mas eficientes de transformar los datos. Como vimos en el capítulo 3, decidimos convertir los problemas a píxeles donde cada dato representaba un píxel; pero hay muchas mas opciones que podríamos tomar. Podríamos dejar de ver el problema

como la imagen típica y buscar resaltar aquellos parámetros mas importantes. También podríamos utilizar los datos como una especie de lápiz, siendo coordenadas que trazamos como si fuese un dibujo en lienzo con la dirección generada en cada renglón de datos; se podrían cambiar el orden y la cantidad de capas, así como aumentar los epochs para ver que resultados arrojan. Básicamente las opciones son amplias y hay mucho por hacer para encontrar la manera óptima de utilizar este método.

Posteriormente, un problema que debemos considerar al hacer redes neuronales convoluciones, es el costo computacional. En ejercicios con pocos datos como los que vimos previamente, el tiempo necesario no era demasiado. Sin embargo, conforme mas datos se tengan, se verá una distinción aún mas clara entre el tiempo en procesar con este modelo con respecto al resto. Por ello, una idea que podría ser viable, es tener una red neuronal convolucional preentrenada y ver que tan bien predice, no solamente porque el entrenamiento hará que ya no tenga un alto costo computacional, sino porque ya tiene conocimiento previo y podría servir para predecir. Mejor aún, se podrían crear redes preentrenadas con problemas generales y usar estas para predecir. En fin, las posibilidades son muchas.

En el preámbulo mencionamos que estas pruebas no fueron hechas para demostrar la superioridad del método, sino simplemente para poder mostrar que es un método viable e iniciar un diálogo; con los ejemplos anteriores vemos que puede ser incluido como un método para predecir de manera positiva y con base en esto, el diálogo puede ser iniciado; no obstante, tambien vimos que hay muchas cosas que se pueden hacer para mejorar estos modelos. Espero que con esta tesis surja un diálogo y se pueda un día, implementar como un algoritmo adicional para problemas generales.

Apéndice A

Códigos empleados

[1] Tensorflow para clasificar dígitos del MNIST:

Importamos la biblioteca.

```
import tensorflow as tf
```

Para crear operaciones interactivas manipulamos variables simbólicas:

```
x = tf.placeholder(tf.float32,[None,784])
```

donde x no es un valor específico, sino que "guarda un lugar". Es un valor que daremos posteriormente. Como queremos agregar las imágenes del MNIST en un vector de 784, lo representamos como un tensor bidimensional con forma [None,784] (None porque la dimensión puede ser de cualquier tamaño, dado que sería el número de observaciones).

Así mismo, también necesitamos los pesos y sesgos para nuestro modelo. Podemos imaginar tratarlos como entradas adicionales. Tensorflow las llama "Variable", que es un tensor modificable.

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

Para implementar el modelo basta con escribir:

```
y_ = tf.placeholder(tf.float32,[None,10])
```

Posteriormente, implementamos la función de entropía cruzada:

```
tf.nn.Softmax_cross_entropy_with_logits
```

Como tensorflow conoce todo el entorno de la computadora, puede automáticamente usar el algoritmo de back propagation para determinar como las variables afectan la pérdida que deseas minimizar. Por ello, solo basta con elegir el algoritmo de optimización, el cual para nosotros será descenso por gradiente:

```
train_step =tf.train.GradientDescentOptimizer(0.5).minimize(
    cross_entropy)
```

Ahora, inicializamos un modelo interactivo:

```
sees = tf.InteractiveSession()
```

Después, instanciamos una operación para inicializar las variables:

```
tf. global_variables_initializer .run()
```

Finalmente entrenamos 1000 veces:

```
for _ in range(1000):
    batch_xs,batch_ys = mnist.train.next_batch(1000)
    sees.run(train_step, feed_dict={x:batch_xs,y_:batch_ys})
```

Evaluemos el modelo:

Obtendremos una lista de booleanos, según si la predicción es cierta o no.

```
correct_prediction = tf.equal(tf.argmax(y,1),tf.argmax(y_,1))
```

Obtenemos entonces el promedio de esta lista.

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Finalmente imprimimos:

```
print(sess.run(accuracy,feed_dict = {x:mnist.test.images,
y_:mnist.test.labels})

[out]: 0.9214
```

[2] Prueba de normales con Tensorflow

```
# bibliotecas
import tensorflow as tf
import numpy as np
import random

# Importamos los datos.
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# Instanciamos todas en cero.
for i in range(0,55000):
    mnist.train.images[i] = np.zeros(784)
    mnist.train.labels[i] = np.zeros(10)

"" Haremos normales con media 0 y svd del 1 al 10
con estos resultados vamos a llenar los datos del mnist. ""

# Para cada una de las observaciones de entrenamiento.
for i in range(0,55000):

# Creamos un nmero aleatorio entre 10 y 100 (de 10 en 10)
    rand = random.randint(1,9)

 # Esto sera una observacion de 784 normales con svd rand.
    mnist.train.images[i] = np.random.normal(0,rand,784)


    if rand == 1:
        mnist.train.labels[i] = [0,0,0,0,0,0,0,0,0,1]
    if rand == 2:
        mnist.train.labels[i] = [0,0,0,0,0,0,0,0,1,0]
    if rand == 3:
        mnist.train.labels[i] = [0,0,0,0,0,0,0,1,0,0]
```

```
    if rand == 4:
        mnist.train.labels[i] =  [0,0,0,0,0,0,1,0,0,0]
    if rand == 5:
        mnist.train.labels[i] =  [0,0,0,0,0,1,0,0,0,0]
    if rand == 6:
        mnist.train.labels[i] =  [0,0,0,0,1,0,0,0,0,0]
    if rand == 7:
        mnist.train.labels[i] =  [0,0,0,1,0,0,0,0,0,0]
    if rand == 8:
        mnist.train.labels[i] =  [0,0,1,0,0,0,0,0,0,0]
    if rand == 9:
        mnist.train.labels[i] =  [0,1,0,0,0,0,0,0,0,0]
    if rand == 10:
        mnist.train.labels[i] =  [1,0,0,0,0,0,0,0,0,0]

# Ahora lo hacemos con las de prueba.

# Para cada una de las observaciones de entrenamiento.
for i in range(0,10000):

# Creamos un nmero aleatorio entre 1 y 10
    rand = random.randint(1,9)

# Esto sera una observacion de 784 normales con svd rand.
    mnist.test.images[i] = np.random.normal(0,rand,784)

    if rand == 1:
        mnist.test.labels[i] =  [0,0,0,0,0,0,0,0,0,1]
    if rand == 2:
        mnist.test.labels[i] =  [0,0,0,0,0,0,0,0,1,0]
    if rand == 3:
        mnist.test.labels[i] =  [0,0,0,0,0,0,0,1,0,0]
    if rand == 4:
        mnist.test.labels[i] =  [0,0,0,0,0,0,1,0,0,0]
    if rand == 5:
        mnist.test.labels[i] =  [0,0,0,0,0,1,0,0,0,0]
```

```
    if rand == 6:
        mnist.test.labels[i] =  [0,0,0,0,1,0,0,0,0,0]
    if rand == 7:
        mnist.test.labels[i] =  [0,0,0,1,0,0,0,0,0,0]
    if rand == 8:
        mnist.test.labels[i] =  [0,0,1,0,0,0,0,0,0,0]
    if rand == 9:
        mnist.test.labels[i] =  [0,1,0,0,0,0,0,0,0,0]
    if rand == 10:
        mnist.test.labels[i] =  [1,0,0,0,0,0,0,0,0,0]
```

A continuación, vamos a utilizar Tensorflow, con el modelo de Softmax para etiquetar esta base de datos.

```
[in]:

import argparse
import sys
random.seed(7)
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
y_ = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(tf.nn.Softmax_cross_entropy_with_logits(
    labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(
    cross_entropy)

sess = tf.InteractiveSession()
tf.  global_variables_initializer  ().run()

for _ in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
```

```
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})


correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                    y_: mnist.test.labels}))

[out]: 0.1736
```

Vemos que los resultados no son muy buenos, por ello, intentemos con convolucionales y comparemos.

```
[in]:

# Cdigo tomado de Google Tensorflow tutorial.

random.seed(7)
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

```
x_image = tf.reshape(x, [-1,28,28,1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

cross_entropy = tf.reduce_mean(
    tf.nn.Softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g" %(i, train_accuracy))
```

```
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g" %accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

"""Dado que el output es muy extenso por la cantidad de iteraciones,
solo dejaremos las ultimas iteraciones y la precisin del modelo."""

[out]:

step 19000, training accuracy 0.84
step 19100, training accuracy 0.94
step 19200, training accuracy 0.9
step 19300, training accuracy 0.94
step 19400, training accuracy 1
step 19500, training accuracy 0.92
step 19600, training accuracy 0.88
step 19700, training accuracy 0.94
step 19800, training accuracy 0.92
step 19900, training accuracy 0.96
test accuracy 0.937
```

Los resultados son prometedores. Dado esto, es posible que nuestra hipótesis sea verídica. Proseguiremos con pruebas más robustas.

[3] Código para visualizar extracto de CIFAR-10:

```
from keras.datasets import cifar10
from matplotlib import pyplot
from scipy.misc import toimage
(X_{train}, y_{train}), (X_{test}, y_{test}) = cifar10.load_data()
for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(toimage($X_{train[i]}$))
pyplot.show()
```

[4] Resolviendo CIFAR-10 con Keras

```
bibliotecas necesarias.

import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')

Cargamos los datos:

(Xtrain}, Ytrain), (Xtest, Ytest) = cifar10.load_data()

Normalizamos a intervalos de 0 a 1:

Xtrain = Xtrain.astype('float32')
Xtest = Xtest.astype('float32')
Xtrain = Xtrain / 255.0
Xtest = Xtest / 255.0
Ytest = np_utils.to_categorical(y_test)
Ytrain = np_utils.to_categorical(y_train)
num_classes = Ytest.shape

Creamos el modelo

model = Sequential()
```

```
model.add(Conv2D(32, (3, 3), input_shape=(3, 32, 32), activation='relu'
    , padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu', kernel_constraint =maxnorm(3)
    ))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu', kernel_constraint =maxnorm(3))
    )
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='Softmax'))
```

Compilamos el modelo

```
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=[
    'accuracy'])
```

Probamos y evaluamos el modelo:

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=
    epochs, batch_size=32)
```

```
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))

test accuracy 0.8038
```

[5] Descripción de los datos del Titanic

Edad

```
g = sns.FacetGrid(df, col='Survived')
g.map(plt.hist, 'Age', bins=30)
plt.show()
```

Creación NFamiliy

```
df['NFamily'] = df['SibSp'] + df['Parch']
```

NFamily vs Sobrevivir

```
g = sns.FacetGrid(df, col='Survived')
g.map(plt.hist, 'NFamily', bins=20)
plt.show()
```

Cuota pagada

```
g = sns.FacetGrid(df, col='Survived')
g.map(plt.hist, 'Fare', bins=10)
plt.show()
```

Clase

```
grid = sns.FacetGrid(df, col='Survived', row='Pclass', size=2.2, aspect
    =1.6)
grid.map(plt.hist, 'Age', alpha=.5, bins=20)
grid.add_legend();
plt.show()
```

Lugar donde embarcaron

```
grid = sns.FacetGrid(df, col='Survived', row='Embarked', size=2.2,
    aspect=1.6)
```

```
grid.map(plt.hist, 'Age', alpha=.5, bins=20)
grid.add_legend();
plt.show()
```

Dependencias intrínsicas entre variables.

```
df[['Pclass', 'Survived']].groupby(['Pclass'], as_index=False).mean().
    sort_values(
                                                    by='Survived',
                                                        ascending=
                                                        False)
```

```
df[["Sex", "Survived"]].groupby(['Sex'], as_index=False).mean().
    sort_values(
                                                  by='Survived',
                                                      ascending=False)
```

```
df[["NFamily", "Survived"]].groupby(['NFamily'],as_index=False).mean()
    .sort_values(
                                                    by='Survived',
                                                        ascending=
                                                        False)
```

```
df[["Embarked", "Survived"]].groupby(['Embarked'],as_index=False).
    mean().sort_values(
                                                    by='Survived',
                                                        ascending=
                                                        False)
```

[6] Arreglando los datos, Titanic

```
df = df.drop(['Ticket', 'Cabin'], axis=1);
```

```
df['Title'] = df['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)
df[['Title', 'Survived']].groupby(['Title'], as_index=False).mean()
```

```
df['Title'] = df['Title'].replace(['Lady', 'Countess','Capt', 'Col','
    Don', 'Dr',
                                    'Major', 'Rev', 'Sir', 'Jonkheer',
                                        'Dona'], 'Otro')
df['Title'] = df['Title'].replace('Mlle', 'Miss')
df['Title'] = df['Title'].replace('Ms', 'Miss')
df['Title'] = df['Title'].replace('Mme', 'Mrs')
```

```
df[['Title', 'Survived']].groupby(['Title'], as_index=False).mean()
```

```
df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
```

```
df[['FamilySize', 'Survived']].groupby(['FamilySize'], as_index=False).
    mean().sort_values(by='Survived', ascending=False)
```

```
df['Family'] = 0
df.loc[(df['FamilySize'] < 5) & (df['FamilySize'] > 1), 'Family'] = 1
df.loc[df['FamilySize'] > 5, 'Family'] = 2
df[['Family', 'Survived']].groupby(['Family'], as_index=False).mean()
```

```
df = df.drop(['Parch', 'SibSp', 'FamilySize','NFamily'], axis=1)
```

Convirtiendo las variables categóricas.

```
mapeo = {"Mr": 1, "Miss": 2, "Mrs": 3, "master": 4, "Otro": 5}
df['Title'] = df['Title'].map(mapeo)
df['Title'] = df['Title']. fillna (0)
df['Sex'] = df['Sex'].map( {'female': 1, 'male': 0} ).astype(int)
```

```
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].dropna().mode()
    [0])
```

```
df['Embarked'] = df['Embarked'].map( {'S': 0, 'C': 1, 'Q': 2} ).astype(
    int)
```

```
df['Age'].fillna(80/5,inplace=True);
df['Int_edad'] = pd.cut(df['Age'], 5)
df[['Int_edad', 'Survived']].groupby(['Int_edad'], as_index=False).
    mean().sort_values(by='Int_edad', ascending=True)
```

```
df.loc[ df['Age'] <= 16, 'Age'] = 0;
df.loc[(df['Age'] > 16) & (df['Age'] <= 32), 'Age'] = 1;
df.loc[(df['Age'] > 32) & (df['Age'] <= 48), 'Age'] = 2;
df.loc[(df['Age'] > 48) & (df['Age'] <= 64), 'Age'] = 3;
df.loc[ df['Age'] > 64, 'Age'] = 4;
# Ahora quitamos Int_edad al no ser necesaria.
df = df.drop(['Int_edad'], axis=1)
```

```
df['Fare'].fillna(512/4,inplace=True);
df['Int_fare'] = pd.qcut(df['Fare'], 4)
df[['Int_fare', 'Survived']].groupby(['Int_fare'], as_index=
                    False).mean().sort_values(by='Int_fare', ascending=
                        True)
```

```
df.loc[ df['Fare'] <= 7.91, 'Fare'] = 0;
df.loc[(df['Fare'] > 7.91) & (df['Fare'] <= 14.454), 'Fare'] = 1
df.loc[(df['Fare'] > 14.454) & (df['Fare'] <= 31), 'Fare']   = 2
df.loc[ df['Fare'] > 31, 'Fare'] = 3
df['Fare'] = df['Fare'].astype(int)
df =df.drop(['Int_fare'], axis=1)
```

[7] Métodos para comparación

```
etiqueta = df['Survived']
datos =df.drop(['Survived'], axis=1)
from sklearn import cross_validation
```

```
from sklearn.metrics import accuracy_score
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix
from sklearn.neural_network import MLPClassifier
```

Máquina de soporte vectorial

```
def SVM(datos,etiqueta,test_size):
    seed = 7
    X_train, X_test, y_train, y_test = cross_validation.
        train_test_split (datos, etiqueta, test_size =test_size,
        random_state=seed)
    clf = svm.SVC(kernel="linear",C=0.025)
    clf.fit(X_train,y_train)
    y_pred = clf.predict(X_test)
    predictions = [round(value) for value in y_pred]
    accuracy = accuracy_score(y_test, predictions)
    print("Accuracy: %.2f%%" % (accuracy * 100.0))
    return y_pred,y_test
```

Árboles de decisión

```
def DTC(datos,etiquetas,test):
    seed = 7
    X_train, X_test, y_train, y_test = cross_validation.
        train_test_split (datos, etiqueta, test_size =test, random_state
        =seed)
    rf = BaggingClassifier( DecisionTreeClassifier (max_depth = 16),
        max_samples = 0.85,max_features=0.85)
    rf.fit(X_train,y_train)
    y_pred = rf.predict(X_test)
    predictions = [round(value) for value in y_pred]
    accuracy = accuracy_score(y_test, predictions)
```

```
    print("Accuracy: %.2f%%" % (accuracy * 100.0))
    return y_pred,y_test
```

Bosques aleatorios.

```
def RandomForest(datos,etiquetas,test):
    seed = 7
    X_train, X_test, y_train, y_test = cross_validation.
        train_test_split (datos, etiqueta, test_size =test, random_state
        =seed)
    rf = RandomForestClassifier()
    rf.fit(X_train,y_train)
    y_pred = rf.predict(X_test)
    predictions = [round(value) for value in y_pred]
    accuracy = accuracy_score(y_test, predictions)
    print("Accuracy: %.2f%%" % (accuracy * 100.0))
    return y_pred,y_test
```

Redes neuronales con feedforward.

```
def RNN(datos,etiquetas,test):
    seed = 7
    X_train, X_test, y_train, y_test = cross_validation.
        train_test_split (datos, etiqueta, test_size =test, random_state
        =seed)
    rf = MLPClassifier()
    rf.fit(X_train,y_train)
    y_pred = rf.predict(X_test)
    predictions = [round(value) for value in y_pred]
    accuracy = accuracy_score(y_test, predictions)
    print("Accuracy: %.2f%%" % (accuracy * 100.0))
    return y_pred,y_test
```

[8] Implementación de RNNCs

```
import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
from keras import backend as K
K.set_image_dim_ordering('th')
datos = datos.drop(['PassengerId'], axis=1)

def shuffle(df, n=3, axis=1):
    for _ in range(n):
        df.apply(np.random.shuffle, axis=axis)
    return df

df = np.stack([np.dstack([datos.values.astype(int)] * 32)] * 3).
    transpose(1, 0, 2, 3)
seed = 8
X_train, X_test, y_train, y_test = cross_validation.train_test_split(df
    , etiqueta,
     test_size =0.8, random_state=seed)


X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]

# Creamos el modelo
```

```
model = Sequential()

# Capa de entrada convolucional, 32 maps caracteristicos de tamao 3x3
    , una funcin de activacion rectificadora y un limite de peso con
    norma maxima 3.
model.add(Conv2D(32, (3, 3), input_shape=(3, 7, 32), padding='same',
    activation='relu', kernel_constraint=maxnorm(3)))

# Dropout en 20 %.
model.add(Dropout(0.2))

# Otra capa de entrada convolucional, 32 maps caracteristicos de
    tamao 3x3, una funcin de activacion rectificadora y un limite
    de peso con norma maxima 3.
model.add(Conv2D(32, (3, 3), activation='relu', padding='same',
    kernel_constraint=maxnorm(3)))

# Capa Max Pool de tamao 2x2
model.add(MaxPooling2D(pool_size=(2, 2)))

# Capa aplanada (**)
model.add(Flatten())

# Capa completamente conectada con 512 unidades y una funcin de
    activacion relu.
model.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3))
    )

# Dropout en 50 %.
model.add(Dropout(0.5))

#Capa de salida completamente conectada con 10 unidades y una funcin
    de activacion Softmax.
model.add(Dense(num_classes, activation='Softmax'))

epochs = 50
```

```
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=[
    'accuracy'])
print(model.summary())
```

Probamos y evaluamos el modelo.

```
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=epochs, batch_size=32)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Red neuronal convolucional con más capas.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(3, 7, 32), activation='relu',
    padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(1, 1)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu', kernel_constraint=maxnorm(3)
    ))
model.add(Dropout(0.2))
```

```
model.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3))
    )
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='Softmax'))

epochs = 150
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=[
    'accuracy'])
print(model.summary())
```

```
model.fit(X_train, y_train, validation_data=(X_test, y_test),
                                        epochs=epochs, batch_size=32)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Referencias

[1] *The data science blog.*
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

[2] Graham Templeton (2015).
Artificial neural networks are changing the world. What are they?

[3] Michael Nielsen (2017).
Neural networks and deep learning.
http://neuralnetworksanddeeplearning.com/chap1.html

[4] Tan, Pang-Ning, Kumar, Vipin y Steinbach, Michael (2006).
Introduction to Data Mining.
http://www-users.cs.umn.edu/~kumar/dmbook/index.php

[5] Tensorflow tutorials.
MNIST For ML Beginners
https://www.tensorflow.org/get_started/mnist/beginners

[6] Jason Brownlee (2016).
Object Recognition with Convolutional Neural Networks in the Keras Deep Learning Library.

[7] Peter Dayan.
The MIT Encyclopedia of the Cognitive Sciences. Unsupervised learning.
http://www.gatsby.ucl.ac.uk/~dayan/papers/dun99b.pdf

[8] Trevor Hastie, Robert Tibshirani, Jerome Friedman.
The Elements of Statistical Learning, second edition. Springer Series in Statistics .

[9] Jeff Schneider.
Tutorial 5, Cross-Validation..
`https://www.cs.cmu.edu/~schneide/tut5/node42.html`.

[10] Freund, Y.; Schapire, R. E. (1999).
Large margin classification using the perceptron algorithm

[11] Abalone dataset
`https://archive.ics.uci.edu/ml/datasets/abalone`

[12] Confusion matrix
`http://www2.cs.uregina.ca/~dbd/cs831/notes/confusion_matrix/confusion_matrix.html`

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

[14] Ron Kohavi, Ross Quinlan
Decision tree discovery
`http://ai.stanford.edu/~ronnyk/treesHB.pdf`

[15] Christopher Olah
Visual Information Theory
`http://colah.github.io/posts/2015-09-Visual-Information/`

[16] Tensorflow tutorials
A Guide to TF Layers: Building a Convolutional Neural Network
`https://www.tensorflow.org/tutorials/layers`

[17] Tensorflow tutorials
Convolutional neural networks
`https://www.tensorflow.org/tutorials/deep_cnn`

[18] UCI Machine learning repository
http://mlr.cs.umass.edu/ml/datasets.html

[19] Kaggle datasets
https://www.kaggle.com/datasets

[20] Tom M. Mitchell
Machine Learning

[21] Corinna Cortes, Vladimir Vapnik
Support-Vector Networks
Machine Learning, 20, 273-297 (1995)

[22] Dan Becker
Rectified Linear Units (ReLU) in Deep Learning https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning

[23] Core Layers - Keras Documentation
https://keras.io/layers/core/