

REDES NEURONALES Y *DEEP LEARNING*



Adrián Colomer Granero y Gabriel Enrique Muñoz Ríos



Universidad
Internacional
de Valencia

Este material es de uso exclusivo para los alumnos de la Universidad Internacional de Valencia. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la Universidad Internacional de Valencia, sin autorización expresa de la misma.

Edita
Universidad Internacional de Valencia

Redes Neuronales y *Deep Learning*

6 ECTS

Adrián Colomer Granero y Gabriel Enrique Muñoz Ríos

Leyendas



Enlace de interés



Ejemplo



Importante



abc Los términos resaltados a lo largo del contenido en color naranja se recogen en el apartado GLOSARIO.

Índice

CAPÍTULO 1. APRENDIZAJE AUTOMÁTICO	7
1.1. El proceso de aprendizaje	8
1.1.1. La tarea, T	8
1.1.2. La medida del rendimiento, P	10
1.1.3. La experiencia, E	11
1.2. Capacidad, sobreajuste y subajuste	14
1.2.1. Regularización	17
1.3. Hiperparámetros y división del conjunto de datos	19
1.3.1. Validación cruzada	20
1.4. Sesgo y varianza en el aprendizaje automático	21
1.4.1. Estimación puntual	21
1.4.2. Sesgo	22
1.4.3. Varianza y error estándar	23
1.4.4. Compromiso entre sesgo y varianza para minimizar el error cuadrático medio	24
1.4.5. Consistencia	25
1.5. Descenso por gradiente	25
1.6. Diseñar un algoritmo de aprendizaje automático	27
 CAPÍTULO 2. REDES NEURONALES ARTIFICIALES	29
2.1. Aprender el conjunto de datos XOR con un MLP	32
2.2. Aprendizaje basado en gradientes	35
2.2.1. Funciones de coste	36
2.2.2. Unidades de salida	38
2.2.3. Unidades ocultas	42
2.2.4. Diseño de arquitectura	47
2.2.5. Propiedades y profundidad de aproximación universal	47
 CAPÍTULO 3. REGULARIZACIÓN EN EL APRENDIZAJE PROFUNDO	49
3.1. Penalización paramétrica de la norma	50
3.1.1. Regularización de parámetros L2	51
3.1.2. Regularización L1	52
3.2. Aumento sintético del conjunto de datos o <i>data augmentation</i>	53
3.3. Parada temprana	55
3.4. <i>Dropout</i>	57

CAPÍTULO 4. REDES NEURONALES CONVOLUCIONALES	61
4.1. La operación de convolución	62
4.2. Motivación	67
4.3. <i>Pooling</i>	69
4.4. Tipos de datos	71
4.5. Redes convolucionales y la historia del aprendizaje profundo	72
CAPÍTULO 5. MODELADO DE SECUENCIAS: REDES RECURRENTES Y RECURSIVAS	74
5.1. Redes neuronales recurrentes	76
5.2. Memoria a largo plazo y otras redes neuronales recurrentes de compuerta	78
5.2.1. Unidades de memoria a largo plazo (LSTM)	79
5.2.2. Otras redes neuronales recurrentes de compuerta	82
CAPÍTULO 6. APRENDIZAJE POR REFUERZO	83
6.1. Conceptos y definiciones	85
6.1.1. Conceptos básicos	85
6.1.2. Conceptos avanzados	87
6.2. Algoritmos de aprendizaje por refuerzo	87
6.2.1. Clasificación de algoritmos de aprendizaje por refuerzo	88
6.2.2. En detalle: Deep Q-Networks y Policy Gradients	90
GLOSARIO	98
ENLACES DE INTERÉS	103
BIBLIOGRAFÍA	104



Capítulo 1

Aprendizaje automático

El **aprendizaje profundo** es un tipo específico de aprendizaje automático. Para comprender bien el aprendizaje profundo, se debe tener una comprensión sólida de los principios básicos del aprendizaje automático. Este capítulo tiene como objetivo exponer los principios generales más importantes que atañen al aprendizaje automático. Se recomienda a los estudiantes que deseen una perspectiva más amplia que consideren libros de texto sobre los fundamentos del aprendizaje automático, como Bishop (2006) y Murphy (2012).

En el presente capítulo, comenzamos con una definición de lo que es un algoritmo de aprendizaje y presentamos un ejemplo: el algoritmo de regresión lineal. Luego procedemos a describir cómo el desafío de ajustar los datos de entrenamiento difiere del desafío de encontrar patrones que se puedan generalizar a nuevos datos. La mayoría de los algoritmos de aprendizaje automático tienen configuraciones llamadas *hiperparámetros*, que deben determinarse fuera del algoritmo de aprendizaje. Discutiremos cómo configurarlos utilizando datos adicionales.

La mayoría de los algoritmos de aprendizaje automático se pueden dividir en las categorías de **aprendizaje supervisado** y **aprendizaje no supervisado**. En el presente capítulo, describiremos estas categorías y daremos algunos ejemplos de algoritmos de aprendizaje simples de cada categoría. La mayoría de los algoritmos de aprendizaje profundo se basan en un algoritmo de optimización llamado **descenso de gradiente estocástico**, es por ello que justo antes de dar paso al estudio de la técnica de aprendizaje profundo, se detallarán las nociones básicas de este.

Adicionalmente, en este capítulo, describimos cómo combinar varios componentes del algoritmo, como un algoritmo de optimización, una función de coste, un modelo y un conjunto de datos, para construir un algoritmo de aprendizaje automático. Finalmente, en la sección “1.6. Diseñar un algoritmo de aprendizaje automático”, describimos algunos de los factores que han limitado la capacidad del aprendizaje automático tradicional para poder generalizar.

Estos desafíos han motivado el desarrollo de algoritmos de aprendizaje profundo que superan estos obstáculos.

1.1. El proceso de aprendizaje

Un algoritmo de aprendizaje automático es un algoritmo que puede aprender a partir de datos. Pero ¿qué entendemos por aprendizaje automático o aprendizaje máquina? Mitchell (1997) proporciona una definición sucinta: “Se dice que un programa de computadora aprende de la experiencia E con respecto a alguna clase de **tareas T** y la medida de evaluación o desempeño P si su desempeño en las **tareas T** , medido por P , mejora con la **experiencia E** ”. A continuación, se detallan los conceptos de **experiencias E** , **tareas T** y **medidas de desempeño P** proporcionando descripciones intuitivas y ejemplos de los diferentes tipos de tareas, medidas de rendimiento y experiencias que pueden usarse para construir algoritmos de aprendizaje automático.

1.1.1. La tarea, T

El aprendizaje automático nos permite abordar tareas que son demasiado difíciles de resolver con programas fijos escritos y diseñados por seres humanos basados en reglas. Desde un punto de vista científico y filosófico, el aprendizaje automático es interesante, porque desarrollar nuestra comprensión implica hacerlo desde los principios que subyacen en la inteligencia.

En esta definición relativamente formal de la palabra **tarea**, el proceso de aprendizaje en sí no es la tarea. El aprendizaje es nuestro medio para lograr la capacidad de realizar la tarea. Por ejemplo, si queremos que un robot pueda caminar, entonces caminar es la tarea.

Podríamos programar el robot para que aprenda a caminar o podríamos intentar escribir directamente un programa que especifique cómo caminar manualmente.



Las tareas de aprendizaje automático generalmente se describen en términos de cómo el sistema de aprendizaje automático debería procesar un ejemplo. Un ejemplo es una colección de características que se han medido cuantitativamente a partir de algún objeto o evento que queremos que procese el sistema de aprendizaje automático. Normalmente representamos un ejemplo como un vector $x \in \mathbb{R}^n$, donde cada entrada x_i del vector es una característica. Por ejemplo, las características de una imagen suelen ser los valores de los píxeles en la imagen.

Se pueden resolver muchos tipos de tareas con el aprendizaje automático. Algunas de las más comunes son las siguientes:

- **Clasificación.** En este tipo de tarea, se pide al modelo de predicción que especifique a qué k categorías pertenece alguna entrada. Para resolver esta tarea, generalmente se pide al algoritmo de aprendizaje que produzca una función $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$. Cuando $y = f(x)$, el modelo asigna una entrada descrita por el vector x a una categoría identificada por el código numérico y . Existen otras variantes de la tarea de clasificación, por ejemplo, donde f genera una distribución de probabilidad sobre las clases. Un ejemplo de una tarea de clasificación es el reconocimiento de objetos, donde la entrada es una imagen (generalmente descrita como un conjunto de valores de brillo de píxeles) y la salida es un código numérico que identifica el objeto en la imagen.
- **Regresión.** En este tipo de tarea, el modelo de aprendizaje debe predecir un valor numérico dada una entrada. Para resolver esta tarea, se pide al algoritmo de aprendizaje que genere una función $f: \mathbb{R}_n \rightarrow \mathbb{R}$. Este tipo de tarea es similar a la clasificación, excepto por el hecho de que el formato de salida es diferente. Un ejemplo de una tarea de regresión es la predicción del uso que hará de su póliza una persona asegurada (de mucha utilidad para establecer las primas de seguro) o la predicción de los precios futuros de ciertos índices de valores.
- **Transcripción.** En este tipo de tarea, el sistema de aprendizaje automático observa una representación relativamente no estructurada de algún tipo de datos y debe ser capaz de transcribir dicha información en forma de texto discreto. Por ejemplo, en el reconocimiento óptico de caracteres, se muestra una imagen que contiene cierto texto al modelo de transcripción y se le pide que devuelva este texto en forma de secuencia de caracteres (por ejemplo, en formato ASCII o Unicode). Google Street View, por ejemplo, utiliza el aprendizaje profundo para procesar los números de dirección de esta manera (Goodfellow, Bulatov, Ibarz, Arnoud y Shet, 2013). Otro ejemplo es el reconocimiento de voz, en el que al modelo de transcripción le entra una forma de onda de audio, y este emite una secuencia de caracteres o códigos de identificación de palabras que describen las palabras que se pronunciaron en la grabación de audio. El aprendizaje profundo es un componente crucial de los sistemas modernos de reconocimiento de voz utilizados en las principales empresas, incluidas Microsoft, IBM y Google (Hinton et al., 2012).
- **Traducción automática.** En una tarea de traducción automática, la entrada consiste en una secuencia de símbolos en cierto idioma, y el modelo de aprendizaje automático debe convertir esta secuencia de entrada en una secuencia de símbolos en otro idioma. Esto se aplica comúnmente a la traducción de idiomas, como, por ejemplo, del inglés al francés. El aprendizaje profundo ha comenzado recientemente a tener un impacto importante en este tipo de tarea (Bahdanau, Cho y Bengio, 2015; Sutskever, Vinyals y Le, 2014).
- **Salida estructurada.** Las tareas de salida estructurada implican cualquier tarea en la que la salida sea un vector (u otra estructura de datos que contenga múltiples valores) con relaciones importantes entre los diferentes elementos. Esta es una categoría amplia que incluye las tareas de transcripción y traducción descritas anteriormente, así como muchas otras. Un ejemplo es el análisis de sentencias: mapear una oración de lenguaje natural en un árbol que describa su estructura gramatical etiquetando los nodos de los árboles como verbos, sustantivos, adverbios, etc. (Collobert, 2011).

- **Detección de anomalías.** En este tipo de tarea, el modelo de predicción examina un conjunto de eventos u objetos y marca algunos de ellos como inusuales o atípicos. Un ejemplo de una tarea de detección de anomalías es la detección de fraude con tarjeta de crédito. Al modelar sus hábitos de compra, una compañía de tarjetas de crédito puede detectar el mal uso de sus tarjetas. Si un ladrón roba la información de su tarjeta de crédito, las compras del ladrón provendrán de una distribución de probabilidad diferente a la de sus compras.
- **Síntesis y muestreo.** En este tipo de tarea, se pide al algoritmo de aprendizaje automático que genere nuevos ejemplos que sean similares a los de los datos de la fase de entrenamiento. La síntesis y el muestreo a través del aprendizaje automático pueden ser útiles para aplicaciones en campos específicos en los que generar grandes volúmenes de contenido a mano sería costoso, aburrido o requeriría demasiado tiempo. Por ejemplo, en el campo de los videojuegos, es posible generar automáticamente texturas para objetos grandes o paisajes, en lugar de requerir que un artista etique manualmente cada píxel (Luo, Carrier, Courville y Bengio, 2013).
- **Reconstrucción de valores perdidos (*inpainting* en inglés).** En este tipo de tarea, el algoritmo de aprendizaje automático recibe un nuevo ejemplo $x \in \mathbb{R}_n$, pero faltan algunas entradas x_i de x . El algoritmo debe proporcionar una predicción de los valores de las entradas que faltan.
- **Eliminación de ruido.** En este tipo de tarea, el algoritmo de aprendizaje automático recibe como entrada un ejemplo ruidoso $\tilde{x} \in \mathbb{R}_n$, obtenido por un proceso de corrupción desconocido a partir de un ejemplo limpio $x \in \mathbb{R}_n$. El modelo debe predecir el ejemplo limpio x a partir de su versión corrupta \tilde{x} o, de manera más general, predecir la distribución de probabilidad condicional $p(x | \tilde{x})$.

Por supuesto, hay muchas otras tareas posibles que se pueden resolver empleando técnicas de aprendizaje automático. Las tareas anteriormente enumeradas están destinadas únicamente a proporcionar ejemplos de lo que puede hacer el aprendizaje automático, no a definir una taxonomía rígida de tareas.

1.1.2. La medida del rendimiento, P

Para evaluar las habilidades de un algoritmo de aprendizaje automático, es necesario llevar a cabo una medida cuantitativa de su rendimiento. Por lo general, esta medida de **rendimiento P** es específica de la **tarea T** que lleva a cabo el sistema.

Para tareas como la clasificación o la transcripción, a menudo medimos la **precisión del modelo**. La precisión es solo la proporción de ejemplos para los cuales el modelo produce la salida correcta.

También podemos obtener información equivalente midiendo la **tasa de error**, es decir, la proporción de ejemplos para los cuales el modelo produce una salida incorrecta.

A menudo nos referimos a la tasa de error como la pérdida esperada de 0-1. La pérdida 0-1 en un ejemplo particular es 0 si está clasificada correctamente y 1 si no lo está.



Por lo general, estamos interesados en que el algoritmo de aprendizaje automático funcione correctamente cuando se le introducen datos que no ha visto antes (fase de **prueba**), ya que esto determinará su capacidad de generalización o, lo que es lo mismo, cómo de bien funcionará cuando se implemente dicho modelo en el mundo real. Por lo tanto, estas medidas de rendimiento se computan utilizando un conjunto de datos de prueba que está separado de los datos empleados en la fase de entrenamiento del sistema de aprendizaje automático.

La elección de la medida de rendimiento puede parecer sencilla y objetiva, pero, a menudo, es difícil elegir una medida de rendimiento que se corresponda bien con el comportamiento deseado del sistema. Dicha medida de rendimiento también vendrá definida por el tipo de tarea T que estemos tratando de resolver.

En algunos casos, esto se debe a que es difícil decidir qué se debe medir. Por ejemplo, al realizar una tarea de transcripción, ¿deberíamos medir la precisión del sistema al transcribir secuencias enteras o deberíamos usar una medida de rendimiento más precisa que cuantifique la bondad del modelo sobre algunos elementos parciales de la secuencia?

Y al realizar una tarea de regresión, ¿deberíamos penalizar más al sistema si con frecuencia comete errores de tamaño mediano o si rara vez comete errores muy grandes? Este tipo de opciones de diseño dependen de la aplicación.

1.1.3. La experiencia, E

Los algoritmos de aprendizaje automático se pueden clasificar en términos generales como **no supervisados** o **supervisados** según el tipo de **experiencia E** que se les proporciona durante el proceso de aprendizaje.

A la mayoría de los algoritmos de aprendizaje se les proporciona un **conjunto de datos** completo con el que experimentar. Un conjunto de datos es una colección de muchos ejemplos, como se define en la sección “1.1. La tarea, T ”. Cada uno de los ejemplos del conjunto de datos se denomina **dato, ejemplo, muestra, instancia**, etc.

Uno de los conjuntos de datos más antiguos estudiados por los estadísticos y los investigadores del aprendizaje automático es el conjunto de datos Iris (Fisher, 1936). Es una colección de medidas de diferentes partes de 150 plantas de iris.

En este conjunto de datos, cada planta individual corresponde a un ejemplo. Las características dentro de cada ejemplo son las medidas de cada parte de la planta: la longitud del sépalo, el ancho del sépalo, la longitud del pétalo y el ancho del pétalo.

El conjunto de datos también registra a qué especie pertenecía cada planta. En el conjunto de datos hay representadas tres especies diferentes.



Los **algoritmos de aprendizaje no supervisados** experimentan con un conjunto de datos que contiene muchas características con las que aprender propiedades útiles de la estructura de este conjunto de datos. En el contexto del aprendizaje profundo, generalmente queremos aprender toda la distribución de probabilidad característica de un conjunto de datos, ya sea explícitamente, como en la estimación de densidad, o implícitamente, en tareas como la síntesis de imagen o la eliminación de ruido. Otros algoritmos de aprendizaje no supervisados realizan otras funciones, como la agrupación, que consiste en dividir el conjunto de datos en grupos de ejemplos similares.

Los **algoritmos de aprendizaje supervisados** experimentan con un conjunto de datos que también contiene características, pero cada ejemplo también está asociado a una **etiqueta u objetivo**. Por ejemplo, el conjunto de datos Iris se anota con las especies de cada planta de iris. Un algoritmo de aprendizaje supervisado puede estudiar el conjunto de datos Iris y aprender a clasificar las plantas de iris en tres especies diferentes en función de sus mediciones, gracias, en parte, a las etiquetas proporcionadas.

En términos generales, el aprendizaje no supervisado implica observar varios ejemplos de un vector aleatorio x e intentar aprender implícita o explícitamente la distribución de probabilidad $p(x)$, o algunas propiedades interesantes de esa distribución. En cambio, el aprendizaje supervisado implica observar varios ejemplos de un vector aleatorio x y un valor asociado o vector y , y luego aprender a predecir y a partir de x , generalmente estimando $p(y|x)$. El término *aprendizaje supervisado* se origina a partir de la vista del objetivo que proporciona un instructor o maestro que muestra al sistema de aprendizaje automático qué hacer. En el aprendizaje no supervisado, no hay instructor o maestro, y el algoritmo debe aprender a dar sentido a los datos sin esta guía.

El aprendizaje no supervisado y el aprendizaje supervisado no son términos formalmente definidos. Las líneas entre ellos, a menudo, son borrosas. Se pueden usar muchas tecnologías de aprendizaje automático para realizar ambas tareas. Por ejemplo, la regla de la cadena de probabilidad establece que, para un vector $x \in \mathbb{R}^n$, la distribución conjunta puede descomponerse de la siguiente manera:

$$p(x) = \prod_{i=1}^n p(x_i | x_1, \dots, x_{i-1})$$

Esta descomposición significa que podemos resolver el problema aparentemente no supervisado modelando $p(x)$ dividiéndolo en n problemas de aprendizaje supervisado. Alternativamente, podemos resolver el problema de aprendizaje supervisado de aprender $p(y|x)$ mediante el uso de tecnologías de aprendizaje no supervisadas tradicionales para aprender la distribución conjunta $p(x,y)$, y luego inferir la siguiente fórmula:

$$p(y|x) = \frac{p(x,y)}{\sum_y p(x,y')}$$

Aunque el aprendizaje no supervisado y el supervisado no son conceptos completamente distintos y formalmente definidos, ayudan a clasificar de manera general algunas de las cosas que hacemos mediante algoritmos de aprendizaje automático. Las personas se refieren a problemas de regresión, clasificación y salida estructurada como aprendizaje supervisado. La estimación de la densidad para el apoyo de otras tareas, generalmente, se considera aprendizaje no supervisado.



Son posibles otras variantes del paradigma de aprendizaje. Por ejemplo, en el **aprendizaje semi-supervisado**, algunos ejemplos incluyen un objetivo de supervisión, como etiqueta o *target*, pero otros no. En el aprendizaje de varias instancias (*multi-instance learning* en inglés), una colección completa de ejemplos se etiqueta según si contiene o no contiene un ejemplo de una clase, pero los miembros individuales de la colección no están etiquetados. Para un ejemplo reciente de aprendizaje de varias instancias con modelos profundos, vea Kotzias, Denil, De Freitas y Smyth (2015).

Algunos algoritmos de aprendizaje automático no solo experimentan con un conjunto de datos fijo. Por ejemplo, los algoritmos de **aprendizaje por refuerzo** interactúan con un entorno, por lo que hay un ciclo de retroalimentación entre el sistema de aprendizaje y sus experiencias. Dichos algoritmos se cubren en el Capítulo 6, sobre aprendizaje por refuerzo, de este manual.

Como hemos visto, la mayoría de los algoritmos de aprendizaje automático necesitan un conjunto de datos en su entrada. Un conjunto de datos se puede describir de muchas maneras, pero una muy común es mediante la matriz de características. Dicha matriz contiene un ejemplo diferente en cada fila, y a su vez, cada columna de la matriz corresponde a una **característica** o **rasgo** diferente.

Por ejemplo, el conjunto de datos Iris contiene 150 ejemplos con cuatro características para cada ejemplo. Esto significa que podemos representar el conjunto de datos con una matriz de diseño $x \in \mathbb{R}^{150 \times 4}$, donde $x_{i,1}$ es la longitud del sépalo de la planta i , $x_{i,2}$ es el ancho del sépalo de la planta i , etc.

Por supuesto, para describir un conjunto de datos como una matriz de características, debe ser posible describir cada ejemplo como un vector, y cada uno de estos vectores debe ser del mismo tamaño. Esto no siempre es posible. Por ejemplo, si tiene una colección de fotografías con diferentes anchuras y alturas, las diferentes fotografías contendrán un número diferente de píxeles, por lo que no todas las fotografías se pueden describir con la misma longitud del vector.

En casos como estos, en lugar de describir el conjunto de datos como una matriz con m filas, lo describimos como un conjunto que contiene m elementos: $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$. Esta notación no implica que dos instancias $x^{(i)}$ y $x^{(j)}$ tengan el mismo tamaño.

Como hemos comentado anteriormente, en el caso del aprendizaje supervisado, el ejemplo contiene una etiqueta, así como una colección de características. Por ejemplo, si queremos usar un algoritmo de aprendizaje para realizar reconocimiento de objetos a partir de fotografías, debemos especificar qué objeto aparece en cada una de las fotos. Podríamos hacer esto con un código numérico: 0 para una persona, 1 para un automóvil, 2 para un gato, y así sucesivamente. A menudo, cuando trabajamos con un conjunto de datos que contiene una matriz de características x , también proporcionamos un vector de etiquetas y , e y_i proporciona la etiqueta, por ejemplo, i .

Por supuesto, a veces la etiqueta puede ser más que un solo número. Por ejemplo, si queremos entrenar un sistema de reconocimiento de voz para transcribir oraciones completas, entonces la etiqueta para cada oración de ejemplo será una secuencia de palabras.

1.2. Capacidad, sobreajuste y subajuste

El desafío central en el aprendizaje automático es que nuestro algoritmo debe funcionar bien con instancias totalmente nuevas, nunca antes vistas, no solo aquellas para las que nuestro modelo fue entrenado. La capacidad de desempeñarse bien en entradas previamente no observadas se llama **habilidad de generalización**.

Por lo general, al entrenar un modelo de aprendizaje automático, tenemos acceso a un **conjunto de entrenamiento**. Lo habitual es calcular cierta medida de error en el conjunto de entrenamiento, por ejemplo, error de entrenamiento, e ir reduciendo dicho error durante el proceso de aprendizaje. Hasta ahora, lo que hemos descrito es simplemente un problema de optimización. Lo que separa el aprendizaje automático de la optimización es que queremos que el error de generalización, también llamado *error en la fase de prueba*, también sea bajo. Por lo general, estimamos el error de generalización de un modelo de aprendizaje automático midiendo su rendimiento en un **conjunto de ejemplos de prueba** que se recopilaron por separado del conjunto de entrenamiento.

¿Cómo podemos inferir en el rendimiento del conjunto de prueba cuando solo podemos observar el conjunto de entrenamiento?

Los datos de entrenamiento y de prueba son generados por una distribución de probabilidad sobre conjuntos de datos llamada **proceso de generación de datos**. Durante dicha fase de generación de datos, se deben asumir ciertas suposiciones: que los ejemplos en cada conjunto de datos son independientes entre sí, y que el conjunto de entrenamiento y el conjunto de prueba están distribuidos de manera idéntica y se han extraído de la misma distribución de probabilidad que el otro.

Este supuesto nos permite describir el proceso de generación de datos con una distribución de probabilidad sobre un solo ejemplo. La misma distribución se utiliza para generar cada ejemplo de entrenamiento y de prueba. Dicha distribución subyacente generadora de datos se denomina p_{data} .

Una conexión inmediata que podemos observar entre el error de entrenamiento y el error de prueba es que el error de entrenamiento esperado de un modelo seleccionado al azar es igual al error de prueba esperado de ese modelo. Supongamos que tenemos una distribución de probabilidad $p(x, y)$ y tomamos muestras de ella repetidamente para generar el conjunto de entrenamiento y el conjunto de prueba. Para algunos valores fijos w (pesos del algoritmo de aprendizaje), el error esperado del conjunto de entrenamiento es exactamente el mismo que el error esperado del conjunto de prueba, porque ambos conjuntos de datos se forman utilizando el mismo proceso de muestreo del conjunto de datos.



Cuando entrenamos un modelo empleando técnicas de aprendizaje automático, en primer lugar, muestreamos el conjunto de entrenamiento, el cual usamos para optimizar los **parámetros w** , y reducimos el error del conjunto de entrenamiento. Posteriormente, tomamos muestras del conjunto de prueba y llevamos a cabo el proceso de inferencia. El error de prueba esperado es mayor o igual al valor esperado del error de entrenamiento.

>>>

>>>

Los factores que determinan como de bien funcionará un algoritmo de aprendizaje automático son los siguientes:

1. La capacidad de hacer que el error de entrenamiento sea pequeño.
2. La capacidad de reducir la brecha entre el error de entrenamiento y el error de prueba.

Estos dos factores corresponden a los dos desafíos centrales en el aprendizaje automático: el **subajuste** (*underfitting* en inglés) y el **sobreajuste** (*overfitting* en inglés). El subajuste ocurre cuando el modelo no puede obtener un valor de error suficientemente bajo en el conjunto de entrenamiento. El sobreajuste ocurre cuando la brecha entre el error de entrenamiento y el error de prueba es demasiado grande.

Podemos controlar si es más probable que un modelo tienda al sobreajuste o al subajuste alterando su **capacidad**. Informalmente, la capacidad de un modelo es su habilidad para adaptarse a una amplia variedad de funciones. Los modelos con baja capacidad pueden tener dificultades para adaptarse al conjunto de entrenamiento. Los modelos con alta capacidad se pueden sobreajustar memorizando las propiedades del conjunto de entrenamiento que posteriormente no servirán en el conjunto de prueba.

Una forma de controlar la capacidad de un algoritmo de aprendizaje es eligiendo su **espacio de hipótesis**, es decir, el conjunto de funciones que el algoritmo de aprendizaje puede seleccionar como solución. Por ejemplo, el algoritmo de regresión lineal tiene el conjunto de todas las funciones lineales de su entrada como su espacio de hipótesis. Podemos generalizar la regresión lineal para incluir polinomios, en lugar de solo funciones lineales, en su espacio de hipótesis. Hacerlo aumenta la capacidad del modelo.

Los algoritmos de aprendizaje automático generalmente funcionan mejor cuando su **capacidad** es **apropiada** para la verdadera complejidad de la tarea que necesitan realizar y para la cantidad de datos de entrenamiento que se les proporciona. Los modelos con capacidad insuficiente no pueden resolver tareas complejas. Los modelos con alta capacidad pueden resolver tareas complejas, pero cuando su capacidad es mayor que la necesaria para resolver la tarea actual, se puede producir un sobreajuste. La Figura 1 muestra este principio en acción. Comparamos un predictor lineal, cuadrático y de grado 9 que intenta ajustar un problema donde la función verdadera es cuadrática.

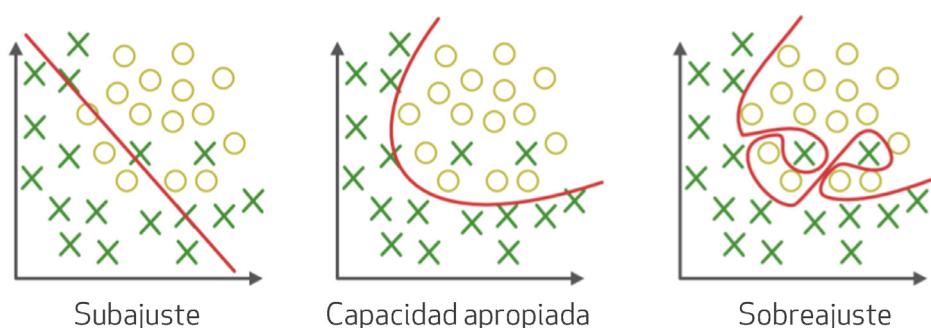


Figura 1. Ejemplo gráfico de los conceptos de *underfitting*, capacidad apropiada y *overfitting*. Adaptado de “Underfitting and Overfitting in Machine Learning”, por Dewang Nautiyal, GeeksforGeeks. Recuperado de <https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning>

La función lineal es incapaz de capturar la curvatura en el verdadero problema subyacente, por lo que no se ajusta. El predictor de grado 9 es capaz de representar la función correcta, pero también es capaz de representar infinitas funciones que pasan exactamente por los puntos de entrenamiento, porque tenemos más parámetros que ejemplos de entrenamiento. Tenemos pocas posibilidades de elegir una solución que generalice bien cuando existan tantas soluciones diferentes. En este ejemplo, el modelo cuadrático se adapta perfectamente a la estructura real de la tarea, por lo que se generaliza bien a datos nuevos.

Hasta ahora hemos descrito solo una forma de cambiar la capacidad de un modelo: cambiando la cantidad de características de entrada que tiene y agregando simultáneamente nuevos parámetros asociados con esas características. Existen otras muchas maneras de cambiar la capacidad de un modelo. La capacidad no está determinada solo por la elección del modelo. El modelo especifica qué familia de funciones puede elegir el algoritmo de aprendizaje al variar los parámetros para reducir un objetivo de entrenamiento. Esto se llama *capacidad de representación del modelo*. En muchos casos, encontrar la mejor función dentro de esta familia es un problema de optimización difícil. En la práctica, el algoritmo de aprendizaje no encuentra realmente la mejor función, sino simplemente una que reduce significativamente el error de entrenamiento. Estas limitaciones adicionales, como la imperfección del algoritmo de optimización, significan que la **capacidad efectiva** del algoritmo de aprendizaje puede ser menor que la capacidad de representación de la familia modelo.

Nuestras ideas modernas sobre la mejora de la generalización de los modelos de aprendizaje automático son refinamientos del pensamiento que se remontan a los tiempos del filósofo Ptolomeo. Este principio establece que, entre las hipótesis en competencia que explican igualmente bien las observaciones conocidas, deberíamos elegir la más simple (la navaja de Occam). Esta idea fue formalizada y acotada en el siglo xx por los fundadores de la teoría del aprendizaje estadístico (Blumer, Ehrenfeucht Haussler, Warmuth et al., 1989; Vapnik, 1982; Vapnik y Chervonenkis, 1971; Vapnik, 1995).

La teoría del aprendizaje estadístico proporciona varios medios para cuantificar la capacidad del modelo. Entre estos, el más conocido es la dimensión Vapnik-Chervonenkis, o dimensión VC. La dimensión VC mide la capacidad de un clasificador binario. La dimensión VC se define como el mayor valor posible de m para el cual existe un conjunto de entrenamiento de m puntos x diferentes que el clasificador puede etiquetar arbitrariamente.

La cuantificación de la capacidad del modelo permite que la teoría del aprendizaje estadístico haga predicciones cuantitativas. Los resultados más importantes en la teoría del aprendizaje estadístico muestran que la discrepancia entre el error de entrenamiento y el error de generalización está limitada por arriba por una cantidad que crece a medida que incrementa la capacidad del modelo, pero se reduce a medida que aumenta el número de ejemplos de entrenamiento (Blumer et al., 1989; Vapnik y Kotz, 1982; Vapnik y Chervonenkis, 1971; Vapnik, 1995). Estos límites proporcionan una justificación intelectual que dice que los algoritmos de aprendizaje automático pueden funcionar, pero rara vez se usan en la práctica cuando se trabaja con algoritmos de aprendizaje profundo. Esto se debe, en parte, a que los límites a menudo son bastante flojos y, en parte, al hecho de que puede ser bastante difícil determinar la capacidad de los algoritmos de aprendizaje profundo. El problema de determinar la capacidad de un modelo de aprendizaje profundo es especialmente difícil, porque la capacidad efectiva está limitada por las capacidades del algoritmo de optimización, y tenemos poca comprensión teórica de los problemas generales de optimización no convexos involucrados en el aprendizaje profundo.

Debemos recordar que, si bien es más probable que las funciones más simples se generalicen (para tener una brecha muy pequeña entre el error en entrenamiento y el error en prueba), aún debemos elegir una hipótesis suficientemente compleja para lograr un bajo error de entrenamiento. Típicamente, el error de entrenamiento disminuye conforme la capacidad del modelo aumenta. Típicamente, el error de generalización tiene una curva en forma de “U” en función de la capacidad del modelo. Esto se ilustra en la primera gráfica de la Figura 1.

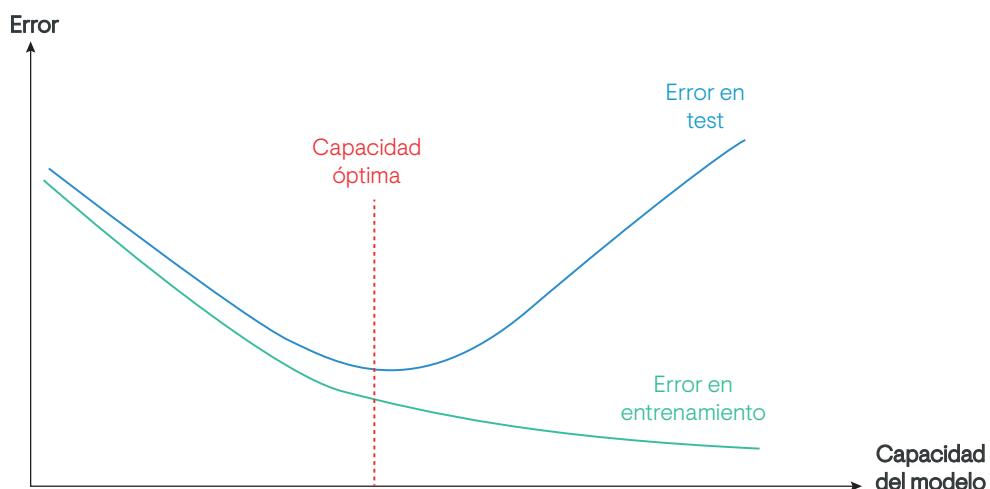


Figura 2. Relación entre capacidad y error. El incremento del error en el conjunto de test aumenta con la capacidad del modelo. Existe un punto de capacidad óptima que divide el gráfico en la zona *underfitting* y la zona *overfitting*. Adaptado de “Memorizing is not learning! – 6 tricks to prevent overfitting in machine learning”, por J. Despois, 2018, *Hackernoon*. Recuperado de <https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42>

1.2.1. Regularización

El objetivo de la investigación del aprendizaje automático no es buscar un algoritmo de aprendizaje universal o el mejor algoritmo de aprendizaje absoluto. Nuestro objetivo es comprender qué tipos de distribuciones son relevantes para el dominio de la materia, así como para la aplicación en soluciones basadas en técnicas de inteligencia artificial. Se debe analizar el dominio y estudiar qué tipos de algoritmos de aprendizaje automático funcionan bien para los datos extraídos de las distribuciones generadoras de datos que nos interesan.

El **teorema del “no existe almuerzo gratis”** (*No Free Lunch* en inglés) sugiere que debemos diseñar nuestros algoritmos de aprendizaje automático para que funcionen bien en una tarea específica. Esto se hace construyendo un conjunto de preferencias para el algoritmo de aprendizaje.

Hasta ahora, el único método para modificar un algoritmo de aprendizaje que hemos discutido concretamente es aumentar o disminuir la capacidad de representación del modelo agregando o eliminando funciones del espacio de hipótesis de soluciones que el algoritmo de aprendizaje puede elegir.

El comportamiento de nuestro algoritmo se ve fuertemente afectado no solo por cuán grande hacemos el conjunto de funciones permitidas en su espacio de hipótesis, sino también por la identidad específica de esas funciones. Por ejemplo, el algoritmo de aprendizaje más sencillo de todos, la regresión lineal, tiene un espacio de hipótesis que consiste en un conjunto de funciones lineales en su entrada.

Estas funciones lineales pueden ser útiles para problemas donde la relación entre entradas y salidas es realmente lineal, pero son menos útiles para problemas que se comportan de manera no lineal. Por ejemplo, la regresión lineal no funcionaría bien si intentáramos usarla para predecir sin x a partir de x . De este modo, podemos controlar el rendimiento de nuestros algoritmos eligiendo de qué tipo de funciones les permitimos extraer soluciones, así como controlando la cantidad de estas funciones.

También podemos dar preferencia a un algoritmo de aprendizaje sobre otro según la solución que aporten ambos dentro de su espacio de hipótesis. Esto significa que ambas funciones son elegibles, pero se prefiere una. La solución no preferida se elegirá solo si se ajusta a los datos de entrenamiento significativamente mejor que la solución preferida.



Ejemplo

Por ejemplo, podemos modificar el criterio de entrenamiento para la regresión lineal para incluir ***weight decay*** (*disminución de peso* en español). Para realizar una regresión lineal con ***weight decay***, minimizamos una suma $J(w)$ que comprende tanto el **error cuadrático medio** en el entrenamiento como un criterio que expresa una preferencia por los pesos para tener una norma L2 al cuadrado más pequeña:

$$J(w) = \text{MSE}_{\text{train}} + \lambda w^T w$$

λ es un valor elegido de antemano que controla la fuerza de nuestra preferencia por pesos más pequeños. Cuando $\lambda = 0$, no imponemos preferencia, en cambio, cuando λ es mayor, obliga a los pesos a hacerse más pequeños. Minimizar $J(w)$ da como resultado una elección de pesos que cumplen una relación de compromiso entre su ajuste a los datos de entrenamiento y la restricción del valor de estos, garantizando que sean pequeños. Esto nos da soluciones que tienen una pendiente más pequeña o que dan menos peso a las características. Como ejemplo de cómo podemos controlar la tendencia de un modelo a sobreajustarse o subadaptarse a través de la disminución de peso, podemos entrenar un modelo de regresión polinómica de alto grado con diferentes valores de λ . Vea la Figura 3 para observar los resultados:

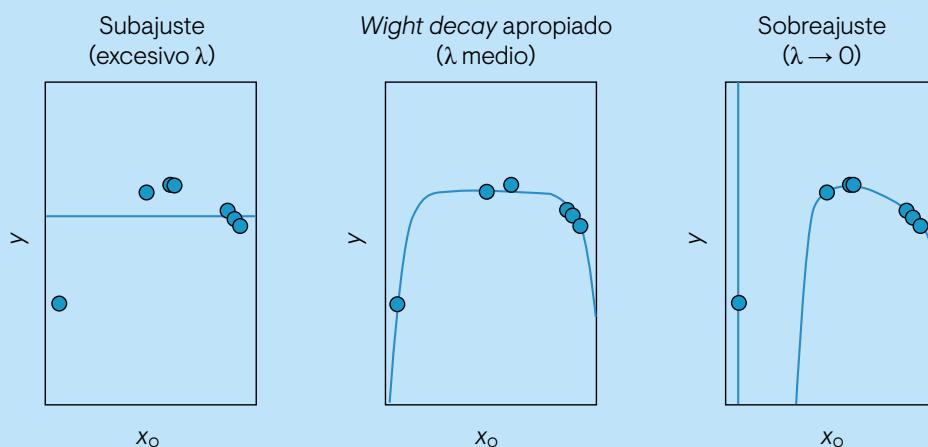


Figura 3. Impacto del parámetro λ de regularización sobre el proceso de aprendizaje. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

De manera más general, podemos regularizar un modelo que aprende una función $f(x; \theta)$ agregando una penalización llamada *regularizador* a la función de coste. En el caso de *weight decay*, el regularizador es $\Omega(w) = w^T w$.

Expresar preferencias para una función sobre otra es una forma más general de controlar la capacidad de un modelo que incluir o excluir miembros del espacio de hipótesis. Podemos pensar que excluir una función de un espacio de hipótesis expresa una preferencia infinitamente fuerte contra esa función.



En nuestro ejemplo de *weight decay*, expresamos nuestra preferencia por las funciones lineales definidas explícitamente con pesos más pequeños, a través de un término adicional en el criterio que minimizamos. Hay muchas otras formas de expresar preferencias por diferentes soluciones, tanto implícita como explícitamente. Juntos, estos diferentes enfoques se conocen como **regularización**. La regularización es cualquier modificación que hacemos a un algoritmo de aprendizaje destinado a reducir su error de generalización, pero no su error de entrenamiento. La regularización es una de las preocupaciones centrales del campo del aprendizaje de máquinas, que solo rivaliza en importancia con la **optimización**.

El teorema del “no existe almuerzo gratis” deja claro que no hay un algoritmo de aprendizaje universal y no existe una forma de regularización óptima. En su lugar, debemos elegir una forma de regularización que se adapte bien a la tarea particular que queremos resolver. La filosofía del aprendizaje profundo, en general, es que una amplia gama de tareas (como todas las tareas intelectuales que podemos realizar los seres humanos) puedan resolverse de manera efectiva utilizando formas de regularización de propósito muy general.

1.3. Hiperparámetros y división del conjunto de datos

La mayoría de los algoritmos de aprendizaje automático se caracterizan por ciertos **hiperparámetros**, configuraciones que podemos usar para controlar el comportamiento del algoritmo. Los valores de los hiperparámetros no son adaptados por el algoritmo de aprendizaje en sí durante la etapa de entrenamiento, aunque podemos diseñar un procedimiento de aprendizaje anidado en el que un algoritmo de aprendizaje aprenda los mejores hiperparámetros para otro algoritmo de aprendizaje.

El ejemplo de regresión polinómica en la Figura 1 tiene un solo hiperparámetro: el grado del polinomio, que actúa como un hiperparámetro de capacidad. El valor λ utilizado para controlar la fuerza del *weight decay* es otro ejemplo de hiperparámetro.



Cabe destacar que no es conveniente llevar a cabo la optimización de un hiperparámetro en el conjunto de entrenamiento. Esta premisa se aplica a todos los hiperparámetros que controlan la capacidad del modelo.

>>>

>>>

Si se aprenden en el conjunto de entrenamiento, dichos hiperparámetros siempre elegirían la capacidad máxima posible del modelo, lo que da como resultado un sobreajuste (véase la Figura 2). Por ejemplo, siempre podemos ajustar mejor el conjunto de entrenamiento con un polinomio de mayor grado y un *weight decay* de $\lambda = 0$ que con un polinomio de menor grado y *weight decay* positivo. Para resolver este problema, necesitamos un **conjunto de ejemplos de validación** que el algoritmo de entrenamiento no observe.

En el apartado anterior, discutimos cómo un conjunto de prueba, compuesto de ejemplos provenientes de la misma distribución que el conjunto de entrenamiento, puede emplearse para estimar el error de generalización de un modelo, después de que el proceso de aprendizaje se haya completado. Es importante que los ejemplos de prueba no se usen de ninguna manera para tomar decisiones sobre el modelo, incluidos sus hiperparámetros. Por este motivo, no se puede utilizar ningún ejemplo del conjunto de prueba en el conjunto de validación. Por lo tanto, siempre construimos el conjunto de validación a partir de datos de entrenamiento. Concretamente, dividimos los datos de entrenamiento en dos subconjuntos disjuntos. Uno de estos subconjuntos se usa para aprender los parámetros w del modelo. El otro subconjunto es nuestro conjunto de validación, utilizado para estimar el error de generalización durante o después del entrenamiento, lo que permite que los hiperparámetros se actualicen en consecuencia. El subconjunto disjunto de datos utilizado para aprender los parámetros w se sigue llamando *conjunto de entrenamiento*. El subconjunto de datos utilizado para optimizar los hiperparámetros se denomina *conjunto de validación*. Por lo general, el primero usa alrededor del 80 por ciento de los datos del conjunto de entrenamiento, y el segundo, el 20 por ciento, para la validación. Dado que el conjunto de validación se utiliza para entrenar los hiperparámetros, el error del conjunto de validación subestimará el error de generalización, aunque generalmente en una cantidad menor que el error de entrenamiento. Una vez completada la optimización de hiperparámetros, el error de generalización real y definitivo se estima utilizando el conjunto de prueba.

1.3.1. Validación cruzada

Dividir el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba fijos puede ser problemático si resulta que el conjunto de prueba es pequeño. Un pequeño conjunto de prueba implica incertidumbre estadística en torno al error de generalización promedio estimado, lo que hace difícil afirmar que el algoritmo A funciona mejor que el algoritmo B para una tarea dada.

Cuando el conjunto de datos tiene cientos de miles de ejemplos o más, esto no es un problema grave. Cuando el conjunto de datos es demasiado pequeño, existen procedimientos alternativos que permiten utilizar todos los ejemplos en la estimación del error medio de entrenamiento, a cambio de un mayor coste computacional. Estos procedimientos se basan en la idea de repetir el entrenamiento y el cálculo de las pruebas en diferentes subconjuntos o divisiones elegidos al azar del conjunto de datos original.

El más común de estos es el procedimiento de **validación cruzada k -fold**, que se muestra en la Figura 4, en el que se forma una partición del conjunto de datos dividiéndolo en k subconjuntos no superpuestos. El error de prueba puede estimarse calculando el error de prueba promedio entre las k pruebas.

En la prueba i , el subconjunto i -ésimo de los datos se usa como conjunto de prueba, y el resto de los datos se usa como conjunto de entrenamiento. Un problema es que no existen estimadores no sesgados de la varianza de dichos estimadores de error promedio (Bengio y Grandvalet, 2004), pero generalmente se usan aproximaciones.

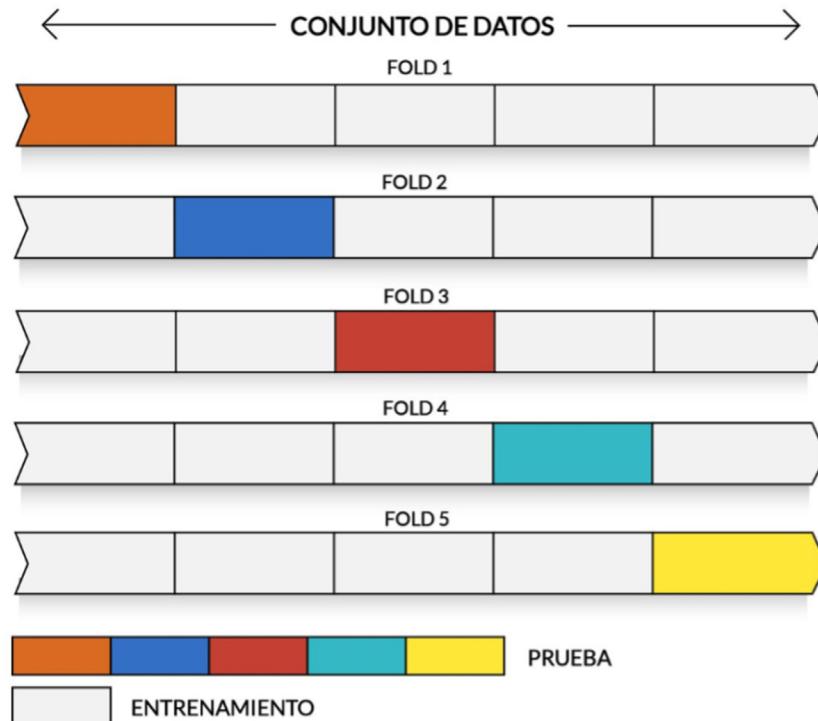


Figura 4. Ejemplo de la técnica de validación cruzada para $k = 5$. Recuperado de “Validación cruzada”, por @dataEvo, 12 de setiembre de 2018, Twitter.

1.4. Sesgo y varianza en el aprendizaje automático

El campo de la estadística nos brinda muchos de los fundamentos básicos en los que se sustenta el aprendizaje automático para resolver una tarea con éxito y garantizando la capacidad de generalización del modelo entrenado. Los conceptos fundamentales, como la estimación de parámetros, el sesgo y la varianza, son útiles para caracterizar formalmente las nociones de generalización, subajuste y sobreajuste.

1.4.1. Estimación puntual

La estimación puntual es el intento de proporcionar la mejor predicción de cierta cantidad de interés. En general, la cantidad de interés puede ser un parámetro único o un vector de parámetros en algún modelo paramétrico, como los pesos a aprender en un problema de regresión lineal, pero también puede tratarse de una función completa.

Para distinguir las estimaciones de los parámetros de su verdadero valor, nuestra convención será denotar una estimación puntual de un parámetro θ por $\hat{\theta}$.

Sea $\{x^{(1)}, \dots, x^{(m)}\}$ un conjunto de m puntos de datos independientes e idénticamente distribuidos. Un estimador puntual o estadístico es cualquier función de los datos que cumple la siguiente ecuación:

$$\hat{\theta}_m = g(x^{(1)}, \dots, x^{(m)})$$

La definición no requiere que g devuelva un valor cercano al verdadero o incluso que el rango de g sea el mismo que el conjunto de valores permitidos de θ . Esta definición de un estimador puntual es muy general y proporcionaría al diseñador de un estimador una gran flexibilidad. Mientras que casi cualquier función puede actuar como estimador, un buen estimador es una función cuya salida está cerca del verdadero g que generó los datos de entrenamiento.

Por ahora, tomamos la perspectiva frecuentista de las estadísticas. Es decir, suponemos que el verdadero valor del parámetro g es fijo pero desconocido, mientras que la estimación puntual $\hat{\theta}$ es una función de los datos. Dado que los datos se extraen de un proceso aleatorio, cualquier función de los datos es aleatoria. Por lo tanto, $\hat{\theta}$ es una variable aleatoria.

La estimación puntual también puede referirse a la estimación de la relación entre las variables de entrada y el objetivo. Nos referimos a estos tipos de estimaciones puntuales como *estimadores de funciones*. A continuación, se revisan las propiedades más comúnmente estudiadas de los estimadores puntuales.

1.4.2. Sesgo

El sesgo de un estimador se define de la siguiente forma:

$$\text{bias}(\hat{\theta}_m) = E(\hat{\theta}_m) - \theta$$

El valor esperado se calcula sobre los datos (vistos como muestras de una variable aleatoria) y θ es el verdadero valor subyacente de θ utilizado para definir la distribución generadora de datos. Se dice que un estimador $\hat{\theta}_m$ es **imparcial** si $\text{bias}(\hat{\theta}_m) = 0$, lo que implica que $E(\hat{\theta}_m) = \theta$. Se dice que un estimador $\hat{\theta}_m$ es **asintóticamente imparcial** si $\lim_{m \rightarrow \infty} \text{bias}(\hat{\theta}_m) = 0$, lo que implica $\lim_{m \rightarrow \infty} E(\hat{\theta}_m) = \theta$.



Ejemplo

Distribución de Bernoulli: considere un conjunto de muestras $\{x(1), \dots, x(m)\}$ que se distribuyen de forma independiente e idéntica de acuerdo con una distribución de Bernoulli con una media θ .

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1-\theta)^{(1-x^{(i)})}$$

Un estimador común para el parámetro θ de esta distribución es la media de muestras de entrenamiento:

$$\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

Para determinar si el estimador está sesgado, podemos sustituir la ecuación anterior en la definición de sesgo:

$$\begin{aligned} \text{bias}(\hat{\theta}_m) &= E(\hat{\theta}_m) - \theta = E\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \theta = \frac{1}{m} \sum_{i=1}^m E[x^{(i)}] - \theta = \\ &= \frac{1}{m} \sum_{i=1}^m \sum_{x^{(i)}=0}^1 (x^{(i)} \theta^{x^{(i)}} (1-\theta)^{(1-x^{(i)})}) - \theta = \frac{1}{m} \sum_{i=1}^m (\theta) - \theta = \theta - \theta = 0 \end{aligned}$$

1.4.3. Varianza y error estándar

Otra propiedad del estimador que podríamos considerar es cuánto esperamos que varíe en función de la muestra de datos. Así como calculamos la expectativa del estimador para determinar su sesgo, podemos calcular su varianza. La varianza de un estimador es simplemente la varianza:

$$\text{var}(\theta)$$

La variable aleatoria es el conjunto de entrenamiento. Alternativamente, la raíz cuadrada de la varianza se denomina *error estándar*, denotado como $SE(\theta)$.

La varianza, o el error estándar, de un estimador proporciona una medida de cómo esperaríamos que la estimación que calculamos a partir de los datos varíe a medida que remuestreemos independientemente el conjunto de datos del proceso de generación de datos subyacente. Del mismo modo que nos gustaría que un estimador exhibiera un sesgo bajo, también nos gustaría que tuviera una varianza relativamente baja.

Cuando calculamos cualquier estadístico usando un número finito de muestras, nuestra estimación del parámetro subyacente verdadero es incierta, en el sentido de que podríamos haber obtenido otras muestras de la misma distribución y sus estadísticos serían diferentes. El grado de variación esperado en cualquier estimador es una fuente de error que queremos cuantificar.

El error estándar de la media viene dado por la siguiente ecuación:

$$SE(\hat{\mu}_m) = \sqrt{\text{var}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}$$

En la fórmula anterior, σ^2 es la verdadera varianza de las muestras x^i . El error estándar a menudo se estima utilizando una estimación de σ . Desafortunadamente, ni la raíz cuadrada de la varianza de la muestra ni la raíz cuadrada del estimador imparcial de la varianza proporcionan una estimación imparcial de la desviación estándar. Ambos enfoques tienden a subestimar la verdadera desviación estándar, pero todavía se usan en la práctica. La raíz cuadrada del estimador imparcial de la varianza es menos que una subestimación. Para grandes m , la aproximación es bastante razonable.

El error estándar de la media es muy útil en los experimentos de aprendizaje automático. A menudo estimamos el error de generalización calculando la media muestral del error en el conjunto de prueba. El número de ejemplos en el conjunto de prueba determina la precisión de esta estimación. Aprovechando el teorema del límite central, que nos dice que la media se distribuirá aproximadamente con una distribución normal, podemos usar el error estándar para calcular la probabilidad de que la expectativa verdadera caiga en cualquier intervalo elegido. Por ejemplo, el intervalo de confianza del 95 por ciento centrado en la media $\hat{\mu}_m$ es el siguiente, bajo la distribución normal con media $\hat{\mu}_m$ y varianza $SE(\hat{\mu}_m)^2$:

$$(\hat{\mu}_m - 1,96 SE(\hat{\mu}_m), \hat{\mu}_m + 1,96 SE(\hat{\mu}_m))$$

En los experimentos de aprendizaje automático, es común decir que el algoritmo A es mejor que el algoritmo B si el límite superior del intervalo de confianza del 95 por ciento para el error del algoritmo A es menor que el límite inferior del intervalo de confianza del 95 por ciento para el error del algoritmo B.



Ejemplo

Distribución de Bernoulli: una vez más, consideramos un conjunto de muestras $\{x(1), \dots, x(m)\}$ extraído de forma independiente e idéntica de una distribución de Bernoulli (recuerde $P(x; \theta) = \theta(1 - \theta)$). Esta vez estamos interesados en calcular la varianza del estimador.

$$\text{var}(\hat{\theta}_m) = \text{var}\left(\frac{1}{m} \sum_{i=1}^m x^{(i)}\right) = \frac{1}{m^2} \sum_{i=1}^m \text{var}(x^{(i)}) = \frac{1}{m^2} \sum_{i=1}^m \theta(1 - \theta) = \frac{1}{m^2} m\theta(1 - \theta) = \frac{1}{m} \theta(1 - \theta)$$

La varianza del estimador disminuye en función de m el número de ejemplos en el conjunto de datos. Esta es una propiedad común de los estimadores populares a la que volveremos cuando discutamos la consistencia (véase la sección “1.4.5. Consistencia”).

1.4.4. Compromiso entre sesgo y varianza para minimizar el error cuadrático medio

El sesgo y la varianza miden dos fuentes diferentes de error en un estimador. El sesgo mide la desviación esperada del valor verdadero de la función o parámetro a estimar. La varianza, por otro lado, proporciona una medida de la desviación del valor estimado del estimador que cualquier muestreo particular de los datos es probable que cause.

¿Qué sucede cuando se nos da la posibilidad de elegir entre dos estimadores, uno con más sesgo y otro con más varianza? ¿Cómo elegimos entre ellos? Por ejemplo, imagine que estamos interesados en aproximar la función que se muestra en la Figura 1 y solo se nos ofrece la posibilidad de elegir entre un modelo con gran sesgo y uno que sufre de una gran varianza. ¿Cómo elegimos entre ellos?

La forma más común de negociar este compromiso es utilizar la validación cruzada. Empíricamente, la validación cruzada es altamente exitosa en muchas tareas del mundo real. Alternativamente, también podemos comparar el error cuadrático medio (MSE) de las estimaciones:

$$\text{MSE} = E[(\hat{\theta}_m - \theta)^2] = \text{bias}(\hat{\theta}_m)^2 + \text{var}(\hat{\theta}_m)$$

El error cuadrático medio mide la desviación general esperada, en un sentido de error al cuadrado, entre el estimador y el valor verdadero del parámetro θ .

La evaluación del error cuadrático medio incorpora tanto el sesgo como la varianza. Los estimadores deseables son aquellos con un error cuadrático medio pequeño, puesto que logran mantener tanto su sesgo como su varianza bajo control.

La relación entre sesgo y varianza está estrechamente vinculada a los conceptos de aprendizaje automático de capacidad, subajuste y sobreajuste. Cuando el error de generalización es medido por el error cuadrático medio (donde el sesgo y la varianza son componentes significativos del error de generalización), un aumento de la capacidad tiende a aumentar la varianza y disminuir el sesgo. Esto se ilustra en la Figura 5, donde vemos nuevamente la curva en forma de “U” del error de generalización en función de la capacidad.

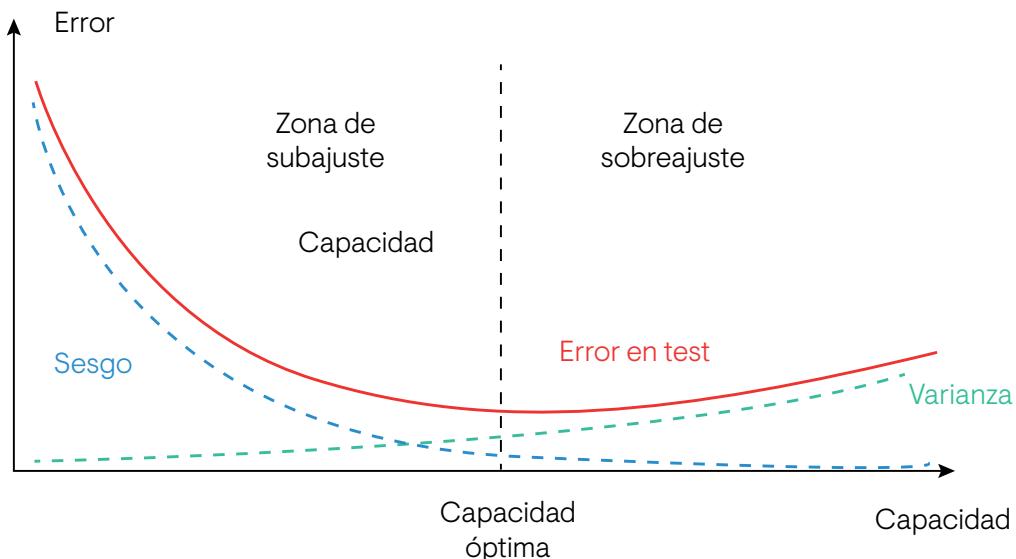


Figura 5. Efecto que tienen el sesgo y la varianza de un modelo relacionado con el error y la capacidad óptima. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

1.4.5. Consistencia

Hasta ahora hemos discutido las propiedades de varios estimadores para un conjunto de entrenamiento de tamaño fijo. Por lo general, también nos preocupa el comportamiento de un estimador a medida que crece la cantidad de datos de entrenamiento. Generalmente deseamos que, a medida que aumenta el número de puntos de datos m en nuestro conjunto de datos, nuestras estimaciones puntuales converjan al valor verdadero de los parámetros correspondientes. Más formalmente, lo representamos de la siguiente forma:

$$p \lim_{m \rightarrow \infty} \hat{\theta}_m = \theta$$

El símbolo $p\lim$ indica convergencia en la probabilidad, lo que significa que para cualquier $\epsilon > 0$, $P(|\hat{\theta}_m - \theta| > \epsilon) \rightarrow 0$ como $m \rightarrow \infty$. La condición descrita por la ecuación anterior se conoce como **consistencia**. A veces se denomina *consistencia débil*, y se pasa a denominar *consistencia fuerte* cuando la convergencia de $\hat{\theta}$ a θ es casi segura. La **convergencia segura** de una secuencia de variables aleatorias $x^{(1)}, x^{(2)} \dots$ a un valor x ocurre cuando $p(\lim_{m \rightarrow \infty} x^{(m)} = x) = 1$.

La consistencia asegura que el sesgo inducido por el estimador disminuye a medida que crece el número de ejemplos de datos. Sin embargo, lo contrario no es cierto: la imparcialidad asintótica no implica consistencia.

1.5. Descenso por gradiente

Casi todo el aprendizaje profundo está impulsado por un algoritmo muy importante: **el descenso de gradiente estocástico** (*stochastic gradient descent*, SGD, en inglés), que es una extensión del algoritmo de descenso por gradiente.

Un problema recurrente en el aprendizaje automático es la necesidad de conjuntos de entrenamiento grandes para conseguir una buena generalización, pero estos también son más costosos computacionalmente.

La función de coste utilizada por un algoritmo de aprendizaje automático a menudo se descompone como una suma sobre ejemplos de entrenamiento de alguna **función de pérdida** por muestra. Por ejemplo, la probabilidad de registro condicional negativa de los datos de entrenamiento se puede escribir de la siguiente manera:

$$J(\theta) = E_{x,y \sim p_{\text{data}}} L(x, y, \theta) = \frac{1}{m} \sum_{i=1}^m L(x^{(i)}, y^{(i)}, \theta)$$

En la ecuación anterior, L es la pérdida por muestra $L(x, y, \theta) = -\log(p(y|x; \theta))$.

Para estas funciones de coste aditivo, el descenso de gradiente requiere computar:

$$\nabla_{\theta} J(\theta) = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(x^{(i)}, y^{(i)}, \theta)$$

El coste computacional de esta operación es $O(m)$. A medida que el tamaño del conjunto de entrenamiento crece a miles de millones de ejemplos, el tiempo para tomar un solo paso de gradiente se vuelve prohibitivamente largo.



La idea de descenso de gradiente estocástico es que el gradiente es una predicción. Dicha predicción puede estimarse aproximadamente utilizando **un pequeño conjunto de muestras**. Específicamente, en cada paso del algoritmo, podemos muestrear un **batch** de ejemplos $B = \{x^{(1)}, \dots, x^{(m')}\}$ extraído uniformemente del conjunto de entrenamiento. El tamaño del *minibatch* m' generalmente se escoge por ser un número relativamente pequeño de ejemplos, que van desde uno hasta unos pocos cientos. De manera crucial, m normalmente se mantiene fijo a medida que crece el tamaño del conjunto de entrenamiento m . Podemos ajustar un conjunto de entrenamiento con miles de millones de ejemplos utilizando actualizaciones calculadas en solo un centenar de ejemplos.

La estimación del gradiente se calcula usando ejemplos del *minibatch* B :

$$g = \frac{1}{m'} \nabla_{\theta} \sum_{i=1}^{m'} L(x^{(i)}, y^{(i)}, \theta)$$

El algoritmo de descenso de gradiente estocástico sigue el gradiente estimado cuesta abajo:

$$\theta \leftarrow \theta - \epsilon g$$

En la expresión anterior, ϵ es la **tasa de aprendizaje**.

El algoritmo de descenso por gradiente básico se ha considerado a menudo lento o poco confiable. Hoy en día, sabemos que los modelos de aprendizaje automático funcionan muy bien cuando se entranan con descenso por gradiente. Es posible que no se garantice que el algoritmo de optimización llegue a un mínimo local en un período de tiempo razonable, pero a menudo encuentra un valor muy bajo de la función de coste lo suficientemente rápido como para ser útil.

El descenso de gradiente estocástico tiene muchos usos importantes fuera del contexto del aprendizaje profundo. Es la forma principal de entrenar grandes modelos lineales en conjuntos de datos muy grandes. Para un tamaño de modelo fijo, el coste por actualización de descenso de gradiente estocástico no depende del tamaño del conjunto de entrenamiento m .

En la práctica, a menudo utilizamos un modelo más grande a medida que aumenta el tamaño del conjunto de entrenamiento, pero no estamos obligados a hacerlo. El número de actualizaciones necesarias para alcanzar la convergencia generalmente aumenta con el tamaño del conjunto de entrenamiento. Sin embargo, a medida que m se aproxima al infinito, el modelo eventualmente convergerá a su mejor error de prueba posible antes de que el descenso de gradiente estocástico haya muestreado cada ejemplo en el conjunto de entrenamiento. Incrementar más m no extenderá la cantidad de tiempo de entrenamiento necesario para alcanzar el mejor error de prueba posible del modelo. Desde este punto de vista, se puede argumentar que el coste asintótico de entrenar un modelo con descenso de gradiente estocástico es $O(1)$ en función de m .

Antes de la llegada del aprendizaje profundo, la forma principal de aprender modelos no lineales era empleando el truco del *kernel* en combinación con un modelo lineal. La mayoría de los algoritmos que emplean este *kernel* requieren de la construcción de una matriz $m \times m$ $G_{ij} = k(x^{(i)}, x^{(j)})$. La construcción de esta matriz tiene un coste computacional $O(m^2)$, que es claramente indeseable para conjuntos de datos con miles de millones de ejemplos. A partir de 2006, el aprendizaje profundo suscitó mucho interés científico, porque fue capaz de generalizar a nuevos ejemplos mejor que los algoritmos de clasificación tradicionales cuando se entrenaba con conjuntos de datos de tamaño mediano, por ejemplo, decenas de miles de ejemplos. Poco después, el aprendizaje profundo generó un interés adicional en la industria, porque proporcionó una forma escalable de entrenar modelos no lineales sobre largos conjuntos de datos.

1.6. Diseñar un algoritmo de aprendizaje automático

Casi todos los algoritmos de aprendizaje automático pueden describirse como instancias particulares de una receta bastante simple: combinar una especificación de un conjunto de datos, una función de coste, un procedimiento de optimización y un modelo.

Por ejemplo, el algoritmo de regresión lineal combina un conjunto de datos que consiste en ciertas muestras x con su respectiva etiqueta y , y cuya función de coste viene definida por la siguiente ecuación:

$$J(w, b) = -E_{x,y-\hat{p}_{\text{data}}} \log p_{\text{model}}(y | x)$$

La especificación del modelo $p_{\text{model}}(y | x) = N(y; x^T w + b, 1)$ y, en la mayoría de los casos, el algoritmo de optimización definido resolviendo aquellos puntos en los que el gradiente del coste es cero mediante el uso de las ecuaciones normales.

Al darnos cuenta de que podemos reemplazar cualquiera de estos componentes en su mayoría, independientemente de los demás, podemos obtener una amplia gama de algoritmos.

La función de coste generalmente incluye al menos un término que hace que el proceso de aprendizaje realice una estimación estadística. La función de coste más común es la conocida como *verosimilitud logarítmica negativa* (*negative log likelihood* en inglés), de modo que minimizar la función de coste implica maximizar la estimación de probabilidad.

La función de coste también puede incluir términos adicionales, como términos de regularización. Por ejemplo, podemos agregar *weight decay* a la función de coste en una regresión lineal para obtener la siguiente ecuación:

$$J(w, b) = \lambda w_2^2 - E_{x,y-\hat{p}_{\text{data}}} \log p_{\text{model}}(y | x)$$

Esto todavía permite la optimización de forma cerrada.

Si cambiamos el modelo para que no sea lineal, la mayoría de las funciones de coste ya no pueden optimizarse de forma cerrada. Esto requiere que elijamos un procedimiento iterativo de optimización numérica, como el descenso de gradiente.

La receta para construir un algoritmo de aprendizaje mediante la combinación de modelos, costes y algoritmos de optimización admite el aprendizaje supervisado y no supervisado. El ejemplo de regresión lineal muestra cómo apoyar el aprendizaje supervisado. El aprendizaje no supervisado se puede respaldar definiendo un conjunto de datos que solo contenga y proporcione un coste y modelo no supervisados apropiados. Por ejemplo, podemos obtener el primer vector PCA especificando que nuestra función de pérdida es la siguiente:

$$J(w) = -E_{x \sim p_{\text{data}}} x - r(x; w)_2^2$$

Mientras que nuestro modelo se define por tener w con norma uno y la función de reconstrucción $r(x) = w^T x w$.

En algunos casos, la función de coste puede ser una función que no podemos evaluar realmente, por razones computacionales. En estos casos, todavía podemos minimizarla de manera aproximada mediante la optimización numérica iterativa, siempre que tengamos alguna forma de aproximar sus gradientes.

La mayoría de los algoritmos de aprendizaje automático hacen uso de esta receta, aunque puede no ser obvia de inmediato. Si un algoritmo de aprendizaje automático parece especialmente único o está diseñado a mano, generalmente puede entenderse que utiliza un optimizador de casos especiales. Algunos modelos, como los árboles de decisión y k -medias (*k-means* en inglés), requieren optimizadores de casos especiales porque sus funciones de coste tienen regiones planas que las hacen inapropiadas para la minimización mediante optimizadores basados en gradientes. Reconocer que la mayoría de los algoritmos de aprendizaje automático se pueden describir con esta receta ayuda a ver los diferentes algoritmos como parte de una taxonomía de métodos para realizar tareas relacionadas que funcionan por razones similares, en lugar de una larga lista de algoritmos que tienen justificaciones separadas. Las redes neuronales profundas no son una excepción a esta receta, tal y como quedará patente en el siguiente capítulo.



Capítulo 2

Redes neuronales artificiales

Esta parte del manual resume el estado del aprendizaje profundo moderno, que se utiliza para resolver aplicaciones prácticas.

El aprendizaje profundo moderno proporciona un marco poderoso para el aprendizaje supervisado. Al agregar más capas y más unidades dentro de una capa, una red profunda puede representar funciones de alto nivel de abstracción caracterizadas por una alta complejidad.

La mayoría de las tareas que consisten en mapear un vector de entrada a un vector de salida, y que son fáciles de realizar para una persona, se pueden realizar a través del aprendizaje profundo, dados conjuntos de datos suficientemente grandes de ejemplos de entrenamiento etiquetados.

Otras tareas, que no pueden describirse como asociar un vector a otro, o que son lo suficientemente difíciles como para que una persona requiera tiempo para pensar y reflexionar para realizar la tarea, permanecen más allá del alcance del aprendizaje profundo por ahora.

Esta parte del manual describe la tecnología de aproximación de funciones paramétricas centrales que está detrás de casi todas las aplicaciones prácticas modernas de aprendizaje profundo.

En este capítulo se describirá el modelo de red neuronal artificial que se utiliza para representar estas funciones.

Las **redes neuronales artificiales**, también llamadas **perceptrones multicapa (MLP)**, son los modelos de aprendizaje profundo por excelencia. El objetivo de una red prealimentada (*feedforward* en inglés) es aproximar alguna función f^* .

Por ejemplo, para un clasificador, $y = f^*(x)$ asigna una entrada x a una categoría y . Una red prealimentada define un mapeo $y = f(x; \theta)$ y aprende el valor de los parámetros θ que resultan en la mejor aproximación de la función.

Estos modelos se denominan *prealimentados* porque la información fluye a través de la función que se evalúa desde x , a través de los cálculos intermedios utilizados para definir f , hasta la salida y . No hay conexiones de retroalimentación en las que las salidas del modelo se realimenten.

Cuando las redes neuronales prealimentadas se extienden para incluir conexiones de retroalimentación, se denominan **redes neuronales recurrentes**, como se presenta en el Capítulo 5, sobre modelado de secuencias: redes recurrentes y recursivas.

Las redes prealimentadas son de extrema importancia para el aprendizaje automático. Forman la base de muchas aplicaciones comerciales importantes. Por ejemplo, las **redes convolucionales** (véase el Capítulo 4, sobre redes neuronales convolucionales) utilizadas para el reconocimiento de objetos a partir de imágenes son un tipo especializado de red de avance. Las redes prealimentadas son un paso conceptual en el camino hacia las redes recurrentes, que impulsan muchas aplicaciones de lenguaje natural.

Las redes neuronales prealimentadas se denominan **redes** porque normalmente se representan componiendo muchas funciones diferentes. El modelo se asocia con un gráfico acíclico dirigido que describe cómo se componen las funciones juntas.

Por ejemplo, podríamos tener tres funciones $f(1)$, $f(2)$ y $f(3)$ conectadas en cadena, para formar $f(x) = f(3)(f(2)(f(1)(x)))$. Estas estructuras de cadena son las estructuras más utilizadas de las redes neuronales. En este caso, llamamos $f(1)$ la primera capa de la red; $f(2)$, la segunda capa, y así sucesivamente.

La longitud total de la cadena da la **profundidad** del modelo. El nombre **aprendizaje profundo** surgió de esta terminología. La capa final de una red prealimentada se llama **capa de salida**. Durante el entrenamiento de la red neuronal, manejamos $f(x)$ para que coincida con $f^*(x)$.

Los datos de entrenamiento nos proporcionan ejemplos ruidosos y aproximados de $f^*(x)$ evaluados en diferentes puntos de entrenamiento. Cada ejemplo x va acompañado de una **etiqueta** $y \approx f^*(x)$.

Los ejemplos de entrenamiento especifican directamente lo que debe hacer la capa de salida en cada punto x ; debe producir un valor cercano a y . El comportamiento de las otras capas no está directamente especificado por los datos de entrenamiento.

El algoritmo de aprendizaje debe decidir cómo usar esas capas para producir la salida deseada, pero los datos de entrenamiento no dicen qué debe hacer cada capa individual. En cambio, el algoritmo de aprendizaje debe decidir cómo usar estas capas para implementar mejor una aproximación de $f^*(x)$. Como los datos de entrenamiento no muestran el resultado deseado para cada una de estas capas, se denominan **capas ocultas**.



Finalmente, estas redes se denominan *neurales* porque están ligeramente inspiradas en la neurociencia. Cada capa oculta de la red suele tener un valor vectorial. La dimensionalidad de estas capas ocultas determina el **ancho** del modelo. Cada elemento del vector puede interpretarse como un papel análogo al de una neurona. En lugar de pensar que la capa representa una sola función de vector a vector, también podemos pensar en la capa como una composición de muchas **unidades** que actúan en paralelo, y que representan, cada una, una función de vector a escalar. Cada unidad se asemeja a una neurona en el sentido que recibe información de muchas otras unidades y calcula su propio valor de activación. La idea de usar muchas capas de representaciones con valores vectoriales se extrae de la neurociencia. La elección de las funciones $f^{(l)}(x)$ utilizadas para calcular estas representaciones también se guía libremente por observaciones neurocientíficas sobre las funciones que calculan las neuronas biológicas. Sin embargo, la investigación moderna de redes neuronales se guía por muchas disciplinas matemáticas y de ingeniería, y el objetivo de las redes neuronales no es modelar perfectamente el cerebro. Es mejor pensar en las redes prealimentadas como máquinas de aproximación de funciones que están diseñadas para lograr una generalización estadística, en ocasiones, extrayendo algunas ideas de lo que sabemos sobre el cerebro, en lugar de modelos de función cerebral.

Una forma de comprender las redes prealimentadas es comenzar con modelos lineales y considerar cómo superar sus limitaciones. Los modelos lineales, como la regresión logística y la regresión lineal, son atractivos porque pueden ajustarse de manera eficiente y precisa, ya sea en forma cerrada o con optimización convexa. Los modelos lineales, sin embargo, tienen el defecto obvio de que la capacidad del modelo está limitada a funciones lineales, por lo que el modelo no puede entender la interacción entre dos variables de entrada.

Para extender modelos lineales para representar funciones no lineales de x , podemos aplicar el modelo lineal a una entrada transformada $\varphi(x)$, donde φ es una transformación no lineal. De manera equivalente, podemos aplicar el truco del *kernel*, para obtener un algoritmo de aprendizaje no lineal basado en la aplicación implícita del mapeo φ . Podemos pensar en φ como un conjunto de características que describen x , o como una nueva representación para x .

La pregunta es cómo elegir el mapeo φ :

1. Una opción es utilizar un φ muy genérico, como el φ de dimensión infinita que es implícitamente empleado por las máquinas de núcleo basadas en el *kernel RBF*. Si $\varphi(x)$ tiene una dimensión lo suficientemente alta, siempre podemos tener la capacidad suficiente para ajustarlo al conjunto de entrenamiento, pero la generalización del conjunto de prueba a menudo sigue siendo deficiente. Las asignaciones de características muy genéricas generalmente se basan solo en el principio de suavidad local y no codifican suficiente información previa para resolver problemas avanzados.
2. Otra opción consiste en diseñar manualmente φ . Hasta la explosión del aprendizaje profundo, este era el enfoque dominante. Requirió décadas de esfuerzo humano para cada tarea por separado, con profesionales especializados en diferentes dominios, como reconocimiento de voz o visión por computadora, y con poca transferencia entre dominios.

3. La estrategia del aprendizaje profundo consiste en aprender φ . En este enfoque, tenemos un modelo $y = f(x; \theta, w) = \varphi(x; \theta)^T w$. En este caso tenemos parámetros θ que se emplean para aprender φ de una amplia clase de funciones, y parámetros w que se asignan desde $\varphi(x)$ a la salida deseada. Este es un ejemplo de una red prealimentada, en la que φ define una capa oculta. Este enfoque es el único de los tres que renuncia a la convexidad del problema. En este enfoque, parametrizamos la representación como $\varphi(x; \theta)$ y utilizamos el algoritmo de optimización para encontrar el θ que corresponde a una buena representación. Si lo deseamos, este enfoque puede capturar el beneficio del primer enfoque al ser altamente genérico; lo hacemos utilizando una familia muy amplia: $\varphi(x; \theta)$. El aprendizaje profundo también puede capturar el beneficio del segundo enfoque. Los investigadores en este campo pueden codificar sus conocimientos para ayudar a la generalización diseñando familias $\varphi(x; \theta)$ que esperan que funcionen bien. La ventaja es que el diseñador humano solo necesita encontrar la familia de funciones generales correcta, en lugar de encontrar precisamente la función correcta.

Este principio general de mejorar los modelos mediante el aprendizaje de características se extiende más allá de las redes prealimentadas en este capítulo. Es un tema recurrente de aprendizaje profundo que se aplica a todos los tipos de modelos descritos a lo largo de este manual.

Comenzamos este capítulo con un ejemplo simple de una red prealimentada. A continuación, abordamos cada una de las decisiones de diseño necesarias para implementar un perceptrón multicapa. Entrenar este tipo de redes requiere tomar muchas de las decisiones necesarias para entrenar un modelo lineal: elegir el optimizador, la función de coste y la forma de las unidades de salida. En este capítulo revisaremos estos conceptos básicos del aprendizaje basado en gradientes, luego procederemos a confrontar algunas de las decisiones de diseño que son exclusivas de las redes prealimentadas. Este tipo de redes han introducido el concepto de capa oculta, y esto nos obliga a elegir las funciones de activación que se utilizarán para calcular los valores de la capa oculta. También debemos diseñar la arquitectura de la red, incluyendo cuántas capas debe contener la red, cómo deben conectarse estas capas entre sí y cuántas unidades debe haber en cada capa. Aprender sobre redes neuronales profundas requiere calcular los gradientes de funciones complicadas. Presentamos el algoritmo de **retropropagación** (*backpropagation* en inglés) y sus generalizaciones modernas, que se pueden usar para calcular eficientemente estos gradientes.

2.1. Aprender el conjunto de datos XOR con un MLP



Ejemplo

Para hacer más concreta la idea de una red prealimentada, comenzamos con un ejemplo de un perceptrón multicapa totalmente funcional en una tarea muy simple: aprender la función XOR.

La función XOR (OR exclusivo) es una operación en dos valores binarios, x_1 y x_2 . Cuando exactamente uno de estos valores binarios es igual a 1, la función XOR devuelve 1. De lo contrario, devuelve 0. La función XOR proporciona la función objetivo $y = f^*(x)$ que queremos aprender. Nuestro modelo proporciona una función $y = f(x; \theta)$, y nuestro algoritmo de aprendizaje adaptará los parámetros θ para hacer que f sea lo más similar posible a f^* .

>>>

>>>

En este simple ejemplo, no nos ocuparemos de la generalización estadística. Queremos que nuestra red funcione correctamente en los cuatro puntos $X = \{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$. Entrenaremos la red en los cuatro puntos. El único desafío es adaptarse al conjunto de entrenamiento. Podemos tratar este problema como un problema de regresión y usar una función de pérdida de error cuadrático medio.

Hemos elegido esta función de pérdida para simplificar las matemáticas para este ejemplo tanto como sea posible. En aplicaciones prácticas, el error cuadrático medio generalmente no es una función de coste apropiada para modelar datos binarios.

Evaluada en todo nuestro conjunto de entrenamiento, la función de pérdida de error cuadrático medio es la siguiente:

$$J(\theta) = \frac{1}{4} \sum_{x \in X} (f^*(x) - f(x; \theta))^2$$

Ahora debemos elegir la forma de nuestro modelo, $f(x; \theta)$. Supongamos que elegimos un modelo lineal, con θ que consiste en w y b . Nuestro modelo está definido para ser de la siguiente forma:

$$f(x; w, b) = x^T w + b$$

Podemos minimizar $J(\theta)$ en forma cerrada con respecto a w y b usando las ecuaciones normales.

Después de resolver las ecuaciones normales, obtenemos $w = 0$ y $b = 1/2$. El modelo lineal simplemente genera 0,5 en todas partes. ¿Por qué pasa esto? La Figura 6 muestra cómo un modelo lineal no puede representar la función XOR.

Una forma de resolver este problema es utilizar un modelo que aprenda un espacio de características diferente en el que un modelo lineal pueda representar la solución. Específicamente, presentaremos una red prealimentada simple con una capa oculta que contiene dos unidades ocultas.

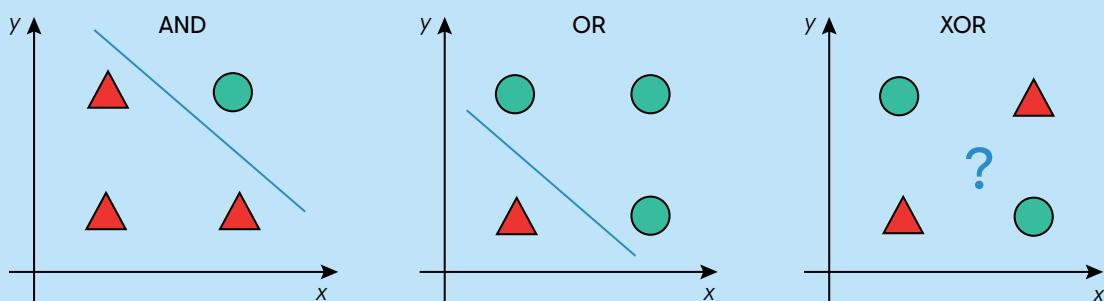


Figura 6. Distribución de los datasets binarios AND, OR y XOR.

>>>

>>>

Consulte la Figura 7 para ver una ilustración de este modelo. Esta red de avance tiene un vector de unidades ocultas h que son calculadas por una función $f^{(1)}(x; W, c)$. Los valores de estas unidades ocultas se utilizan como entrada para una segunda capa. La segunda capa es la capa de salida de la red. La capa de salida sigue siendo solo un modelo de regresión lineal, pero ahora se aplica a h , en lugar de a x . La red ahora contiene dos funciones encadenadas, $h = f^{(1)}(x; W, c)$ e $y = f^{(2)}(h; w, b)$, siendo el modelo completo $f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x))$. ¿Qué función debe calcular $f^{(1)}$? Los modelos lineales nos han servido bien hasta ahora, y puede ser tentador hacer que $f^{(1)}$ también sea lineal. Desafortunadamente, si $f^{(1)}$ fuera lineal, entonces la red prealimentada en su conjunto seguiría siendo una función lineal de su entrada. Ignorando los términos de intercepción por el momento, suponga que $f^{(1)}(x) = W^T x$ y $f(x) = x^T Ww$. Entonces $f(x) = x^T Ww$. Podríamos representar esta función como $f(x) = x^T w'$, donde $w' = Ww$.

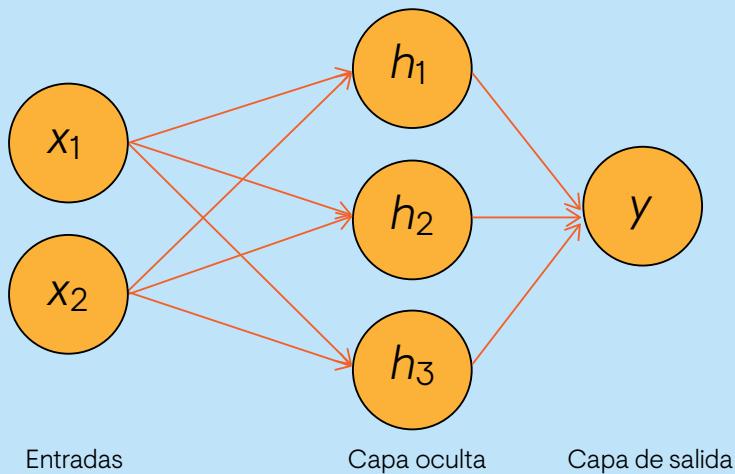


Figura 7. Representación de un perceptrón multicapa con una única capa oculta compuesta por dos unidades o neuronas.

Claramente, debemos usar una función no lineal para describir las características. La mayoría de las redes neuronales lo hacen utilizando una transformación afín controlada por parámetros aprendidos, seguida de una función fija no lineal llamada **función de activación**. Utilizamos esa estrategia aquí, definiendo $h = g(W^T x + c)$, donde W proporciona los pesos de una transformación lineal y c , los sesgos. Anteriormente, para describir un modelo de regresión lineal, utilizamos un vector de pesos y un parámetro de sesgo escalar para describir una transformación afín de un vector de entrada a un escalar de salida. Ahora, describimos una transformación afín de un vector x a un vector h , por lo que se necesita un vector completo de parámetros de sesgo. La función de activación g generalmente se elige para que sea una función que se aplica por elementos, con $h_i = g(x^T W_{:,i} + c_i)$.

>>>

En las redes neuronales modernas, la recomendación predeterminada es usar la unidad lineal rectificada, o ReLU (Glorot, Bordes y Bengio, 2011; Jarrett, Kavukcuoglu, Ranzato y LeCun, 2009; Nair y Hinton, 2010), definida por la función de activación $g(z) = \max\{0, z\}$, representada en la Figura 8. Ahora podemos especificar nuestra red completa de la forma siguiente:

$$f(x; W, c, w, b) = w^T \max\{0, W^T x + c\} + b$$

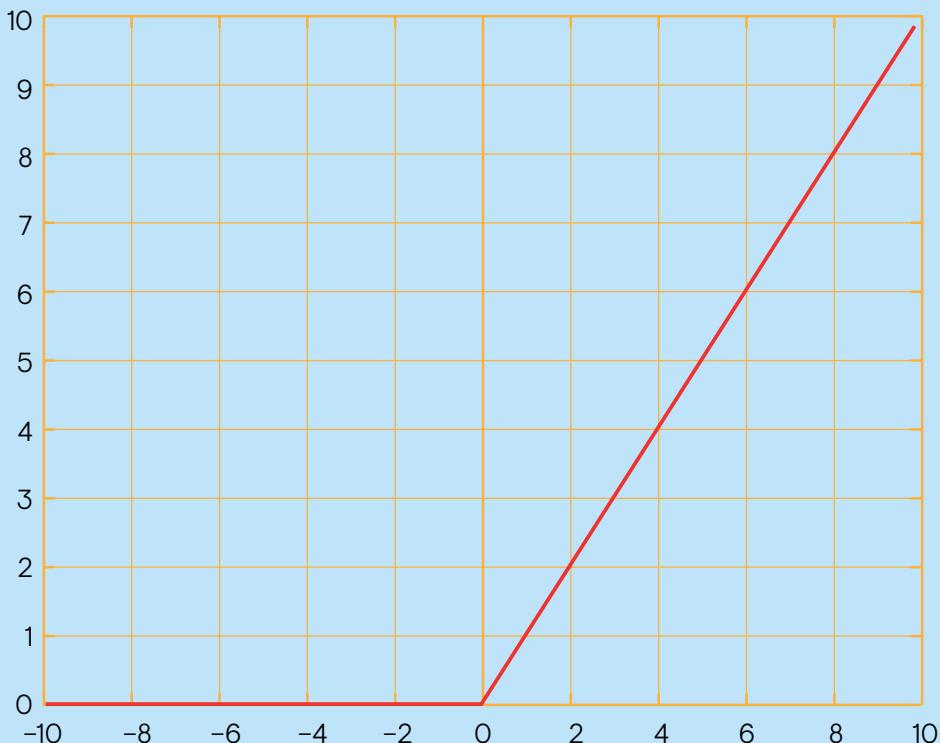


Figura 8. Función de activación de unidad lineal rectificada o ReLU.

2.2. Aprendizaje basado en gradientes

Diseñar y entrenar una red neuronal no es muy diferente de entrenar cualquier otro modelo de aprendizaje automático con descenso de gradiente. En la sección “1.6. Diseñar un algoritmo de aprendizaje automático”, describimos cómo construir un algoritmo de aprendizaje automático especificando un procedimiento de optimización, una función de coste y una familia de modelos.

La mayor diferencia entre los modelos lineales que hemos visto hasta ahora y las redes neuronales es que la no linealidad de una red neuronal hace que las funciones de pérdida más interesantes se vuelvan no convexas. Esto significa que las redes neuronales generalmente se entrenan mediante el uso de optimizadores iterativos basados en gradientes que simplemente llevan la función de coste a un valor muy bajo, en lugar de en la resolución de ecuaciones lineales que se emplean para entrenar modelos de regresión lineal o algoritmos de optimización convexa con convergencia global empleados para entrenar modelos de regresión logística o SVM.

La optimización convexa converge a partir de cualquier parámetro inicial en teoría, porque en la práctica es robusto, pero puede encontrar problemas numéricos. El **descenso de gradiente estocástico** aplicado a las funciones de pérdida no convexa no tiene tal garantía de convergencia y es sensible a la inicialización de los parámetros. Para las redes neuronales prealimentadas, es importante inicializar todos los pesos a valores aleatorios pequeños. Los términos de sesgo pueden inicializarse a cero o a valores positivos pequeños.

Por supuesto, también es posible entrenar modelos basados en la regresión lineal y las máquinas de soporte vectorial mediante el descenso por gradiente, de hecho, esto es común cuando el conjunto de entrenamiento es extremadamente grande. Desde este punto de vista, entrenar una red neuronal no es muy diferente de entrenar cualquier otro modelo. Calcular el gradiente es un poco más complicado para una red neuronal, pero se puede hacer de manera eficiente y exacta.

Al igual que con otros modelos de aprendizaje automático, para aplicar el aprendizaje basado en gradientes debemos elegir una función de coste y debemos elegir cómo representar la salida del modelo. A continuación, revisamos estas consideraciones de diseño con especial énfasis en el escenario de redes neuronales.

2.2.1. Funciones de coste

Un aspecto importante del diseño de una red neuronal profunda es la elección de la **función de coste**. Afortunadamente, las funciones de coste para las redes neuronales son muy similares a otros modelos paramétricos, como los modelos lineales.

En la mayoría de los casos, nuestro modelo paramétrico define una distribución $p(y|x;\theta)$ y simplemente utilizamos el principio de máxima verosimilitud. Esto significa que usamos la **entropía cruzada** (cross-entropy en inglés) entre los datos de entrenamiento y las predicciones del modelo como función de coste.

A veces, adoptamos un enfoque más simple, donde en lugar de predecir una distribución de probabilidad completa sobre y , simplemente predecimos algunos estadísticos de y condicionados por x . Las funciones de pérdida especializadas nos permiten entrenar un predictor de estas estimaciones.

La función de coste total utilizada para entrenar una red neuronal, a menudo, combinará una de las funciones de coste primario descritas aquí con un término de regularización. Ya hemos visto algunos ejemplos simples de regularización aplicados a modelos lineales en la sección “1.2.1. Regularización”. El enfoque de *weight decay* utilizado para modelos lineales también es directamente aplicable a redes neuronales profundas y se encuentra entre las estrategias de regularización más populares. En el Capítulo 3, sobre regularización en el aprendizaje profundo, se describen estrategias de regularización más avanzadas para redes neuronales.

Aprendizaje de distribuciones condicionales de máxima verosimilitud



La mayoría de las redes neuronales modernas se entranan con la máxima verosimilitud. Esto significa que la función de coste es simplemente la **verosimilitud logarítmica negativa** o **entropía cruzada** entre los datos de entrenamiento y la distribución del modelo. Esta función de coste viene dada por la siguiente función:

$$J(\theta) = -E_{x,y \sim p_{\text{data}}} \log p_{\text{model}}(y|x).$$

La forma específica de la función de coste cambia de modelo a modelo, dependiendo de la forma específica de $\log p_{\text{model}}$. La expansión de la ecuación anterior generalmente produce algunos términos que no dependen de los parámetros del modelo y se pueden descartar.

Por ejemplo, si $p_{\text{model}}(y|x) = (y; f(x;\theta), I)$, recuperamos el coste de error cuadrático medio, que es el siguiente:

$$J(\theta) = \frac{1}{2} E_{x,y \sim p_{\text{data}}} y - f(x;\theta)^2 + \text{const}$$

Se recupera el coste de error cuadrático medio hasta un factor de escala de 0,5 y un término que no depende de θ . La constante descartada se basa en la varianza de la distribución gaussiana, que, en este caso, elegimos no parametrizar.

Una ventaja de este enfoque, es decir, de derivar la función de coste de la máxima probabilidad es que elimina la carga de diseñar funciones de coste para cada modelo. La especificación de un modelo $p(y|x)$ determina automáticamente un registro de función de coste $p(y|x)$.

Un tema recurrente a lo largo del diseño de la red neuronal es que el gradiente de la función de coste debe ser lo suficientemente grande y predecible para servir como una buena guía para el algoritmo de aprendizaje. Las funciones que saturan (se vuelven muy planas) hacen que el gradiente se vuelva muy pequeño y no se cumpla la premisa anterior. En muchos casos, esto sucede porque las funciones de activación utilizadas para producir la salida de las unidades ocultas o las unidades de salida se saturan. La entropía cruzada ayuda a evitar este problema para muchos modelos. Varias unidades de salida implican una función \exp , que puede saturarse cuando su argumento es muy negativo. La función de registro en la función de coste de entropía cruzada deshace la \exp de algunas unidades de salida. Discutiremos la interacción entre la función de coste y la elección de la unidad de salida en la sección “2.2.2. Unidades de salida”.

Una propiedad inusual del coste de entropía cruzada para realizar la estimación de máxima verosimilitud es que generalmente no tiene un valor mínimo cuando se aplica a los modelos comúnmente utilizados en la práctica. Para variables de salida discretas, la mayoría de los modelos están parametrizados de tal manera que no pueden representar una probabilidad de cero o uno, pero pueden acercarse arbitrariamente a hacerlo. La regresión logística es un ejemplo de tal modelo. Para variables de salida de valor real, si el modelo puede controlar la densidad de la distribución de salida (por ejemplo, aprendiendo el parámetro de varianza de una distribución de salida gaussiana), entonces es posible asignar una densidad extremadamente alta a las salidas correctas del conjunto de entrenamiento, lo que resulta en una entropía cruzada que se aproxima al infinito negativo. Las técnicas de regularización que se describirán en el Capítulo 3, sobre regularización en el aprendizaje profundo, proporcionan varias formas diferentes de modificar el problema de aprendizaje para que el modelo no pueda obtener una recompensa ilimitada en este tipo de problemas.

Aprendizaje de estadísticas condicionales

En lugar de aprender una distribución de probabilidad completa $p(y|x;\theta)$, a menudo queremos aprender solo una estadística condicional de y dado x . Por ejemplo, podemos tener un predictor $f(x;\theta)$ que deseamos emplear para predecir la media de y .

Desde este punto de vista, podemos ver la función de coste como un **funcional** más que como una función. Un funcional es un mapeo de funciones a números reales. Por lo tanto, podemos pensar en aprender cómo elegir una función, en lugar de simplemente elegir un conjunto de parámetros. Podemos diseñar nuestro coste funcional para que su mínimo ocurra en alguna función específica que deseamos. Por ejemplo, podemos diseñar el coste funcional para que su mínimo recaiga en la función que asigna x al valor esperado de y dado x . Resolver un problema de optimización con respecto a una función requiere una herramienta matemática llamada **cálculo de variaciones**.

Nuestro primer resultado derivado del cálculo de variaciones es que resolver el problema de optimización siguiente:

$$f^* = \underset{f}{\operatorname{argmin}} E_{x,y \sim p_{\text{data}}} y - f(x)^2$$

Ello lleva a la siguiente fórmula, siempre y cuando esta función se encuentre dentro de la clase sobre la que optimizamos:

$$f^* = \underset{f}{\operatorname{argmin}} E_{x,y \sim p_{\text{data}}} [y]$$

En otras palabras, si pudiéramos entrenar infinitas muestras de la verdadera distribución de generación de datos, minimizar la función de coste de error cuadrático medio daría una función que predice la media de y para cada valor de x .

Diferentes funciones de coste dan diferentes estadísticas. Un segundo resultado derivado del cálculo de variaciones es que la fórmula siguiente produce una función que predice el valor medio de y para cada x , siempre que dicha función pueda ser descrita por la familia de funciones que optimizamos:

$$f^* = \underset{f}{\operatorname{argmin}} E_{x,y \sim p_{\text{data}}} |y - f(x)|$$

Esta función de coste se denomina comúnmente **error medio absoluto**.

Desafortunadamente, el error cuadrático medio y el error absoluto medio, a menudo, conducen a malos resultados cuando se usan junto a una optimización basada en gradiente. Algunas unidades de salida que saturan producen gradientes muy pequeños cuando se combinan con estas funciones de coste. Esta es una razón por la cual la función de coste de entropía cruzada es más popular que el error cuadrático medio o el error absoluto medio, incluso cuando no es necesario estimar una distribución completa $p(y|x)$.

2.2.2. Unidades de salida



La elección de la función de coste está estrechamente relacionada con la elección de la **unidad de salida**. La mayoría de las veces simplemente usamos la entropía cruzada entre la distribución de datos y la distribución del modelo. La elección de cómo representar la salida determina la forma de la función de entropía cruzada.

>>>

>>>

Cualquier tipo de unidad de red neuronal que pueda usarse como salida también puede usarse como una **unidad oculta**. Aquí, nos enfocamos en el uso de estas unidades como salidas del modelo, pero, en principio, también pueden usarse internamente. Volvemos a hablar sobre estas unidades desde el punto de vista de su uso como unidades ocultas en la sección “2.2.3. Unidades ocultas”.

A lo largo de esta sección, suponemos que la red prealimentada proporciona un conjunto de características ocultas definidas por $h = f(x; \theta)$. El papel de la capa de salida es proporcionar una transformación adicional de las características para completar la tarea que debe realizar la red (por ejemplo, clasificación binaria, clasificación multiclas, regresión, etc.).

Unidades lineales para distribuciones de salida gaussianas

Un tipo simple de unidad de salida se basa en una transformación afín sin no linealidad. A menudo se llaman simplemente **unidades lineales**.

Dadas las características h , una capa de salida lineal produce un vector $\hat{y} = W^T h + b$.

Las capas de salida lineal, a menudo, se usan para producir la media de una distribución Gaussiana condicional como la siguiente:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

Maximizar la probabilidad de registro es equivalente a minimizar el error cuadrático medio.

La estadística de máxima verosimilitud hace que sea sencillo aprender también la covarianza de la gaussiana, o hacer que esta sea función de la entrada. Sin embargo, la covarianza debe limitarse a una matriz positiva definida para todas las entradas. Es difícil satisfacer tales restricciones con una capa de salida lineal, por lo que normalmente se utilizan otras unidades de salida para parametrizar la covarianza.

Debido a que las unidades lineales no saturan, presentan poca dificultad para los algoritmos de optimización basados en gradientes y pueden usarse en una amplia variedad de algoritmos de optimización.

Unidades sigmoides para distribuciones de salida binarias

Muchas tareas requieren predecir el valor de una variable binaria y . Los problemas de clasificación en dos clases se pueden definir de esta manera.

El enfoque de máxima verosimilitud es definir una distribución de Bernoulli sobre y condicionada por x .

Una distribución de Bernoulli se define por un solo número. La red neuronal necesita predecir solo $P(y=1|x)$. Para que este número sea una probabilidad válida, debe estar en el intervalo $[0, 1]$.

Satisfacer esta restricción requiere un esfuerzo de diseño. Supongamos que usaremos una unidad lineal y umbralizaremos su valor para obtener una probabilidad válida con la siguiente fórmula:

$$P(y=1|x) = \max\{0, \min\{1, w^T h + b\}\}$$

De hecho, esto definiría una distribución condicional válida, pero no podríamos entrenar la red de manera efectiva por descenso en gradiente. Cada vez que $w^T h + b$ excediera del intervalo de la unidad, el gradiente de la salida del modelo con respecto a sus parámetros sería 0.

Un gradiente de 0 suele ser problemático, porque el algoritmo de aprendizaje ya no tiene una guía sobre cómo mejorar los parámetros correspondientes (conocido como *problema de desvanecimiento de gradiente, vanishing gradient* en inglés).

Por tanto, es mejor emplear un enfoque diferente que garantice que siempre haya un gradiente prominente cuando el modelo proporcione la respuesta incorrecta. Este enfoque se basa en el uso de unidades de salida sigmoides combinadas con la máxima verosimilitud.



Una unidad de salida **sigmoide** se define de la siguiente manera:

$$\hat{y} = \sigma(w^T h + b)$$

En la anterior fórmula, σ es la función sigmoide logística descrita por la siguiente expresión, representada en la Figura 9:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

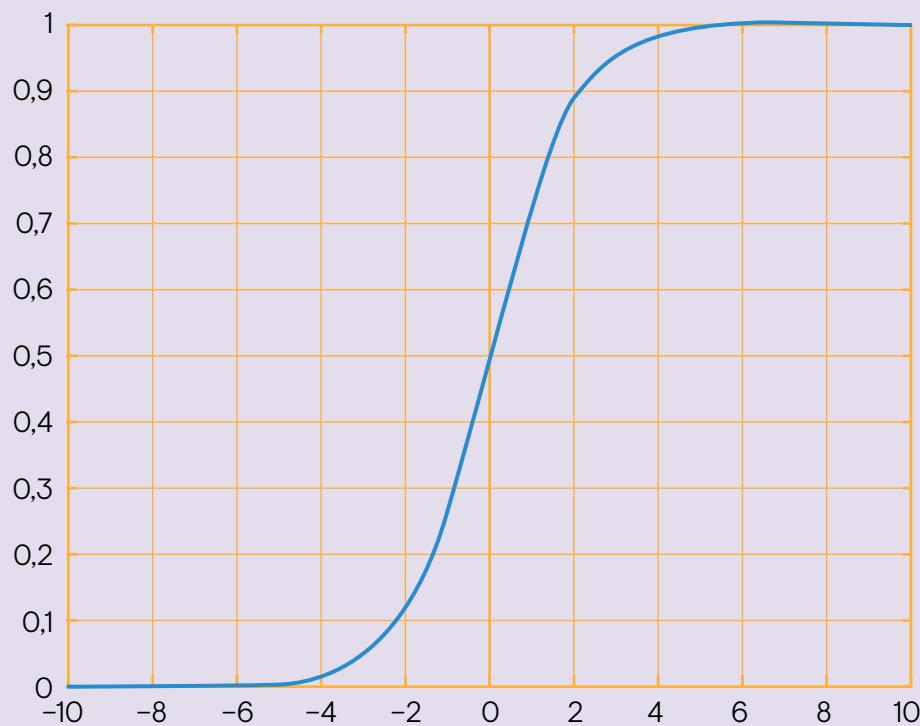


Figura 9. Función de activación sigmoide.

Cuando utilizamos la activación sigmoide en la capa de salida junto al error cuadrático medio como función de pérdidas, la pérdida puede saturar en cualquier momento que $\sigma(w^T h + b)$ sature. La función de activación sigmoide satura a cero cuando $\sigma(w^T h + b)$ se vuelve muy negativo, y satura a uno cuando $\sigma(w^T h + b)$ se vuelve muy positivo. Cuando esto sucede, el gradiente puede reducirse demasiado para ser útil para aprender, tanto si el modelo obtiene la respuesta correcta como la respuesta incorrecta. Por esta razón, la máxima verosimilitud o entropía cruzada binaria es casi siempre el enfoque preferido para entrenar con unidades sigmoides en la capa de salida.

Analíticamente, el logaritmo de la sigmoide siempre está definido y es finito, porque la sigmoide devuelve valores restringidos al intervalo abierto $(0, 1)$, en lugar de utilizar todo el intervalo cerrado de probabilidades válidas $[0, 1]$.

Unidades softmax para distribuciones de salida multiclas

Cada vez que deseamos representar una distribución de probabilidad sobre una variable discreta con n valores posibles, podemos usar la función **softmax**. Esto puede verse como una generalización de la función sigmoide, que se emplea para representar una distribución de probabilidad sobre una variable binaria.

Las funciones softmax se usan con mayor frecuencia como la salida de un clasificador, para representar la distribución de probabilidad sobre n clases diferentes.

En el caso de las variables binarias, deseamos producir un solo número, como se muestra a continuación:

$$P(y = 1 | x)$$

Debido a que este número necesita estar comprendido entre 0 y 1, y como queremos que el logaritmo del número se comporte bien para una optimización basada en gradiente, se opta por predecir un número $z = \log \tilde{P}(y = 1 | x)$.



Para generalizar al caso de una variable discreta con n valores, ahora necesitamos producir un vector \hat{y} , con $\hat{y} = P(y = i | x)$. Requerimos no solo que cada elemento de \hat{y} esté entre 0 y 1, sino que además necesitamos que todo el vector sume 1, de modo que represente una distribución de probabilidad válida. Primero, una capa lineal predice probabilidades de registro no normalizadas, como se muestra a continuación:

$$z = W^T h + b$$

En la fórmula anterior, $z_i = \log \tilde{P}(y = i | x)$. La función softmax emplea la función exponencial sobre dicho término a la vez que normaliza el término z para obtener \hat{y} . Formalmente, la función softmax se define de la siguiente forma:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

>>>

>>>

Al igual que la sigmoide, el uso de la función $\exp()$ funciona bien cuando se entrena con softmax para generar un valor objetivo utilizando la máxima verosimilitud o entropía cruzada categórica como función y pérdidas. En este caso, deseamos maximizar $\log P(y = i; z) = \log \text{softmax}(z)_i$. Definir la función softmax en términos de la función $\exp()$ es muy natural porque el logaritmo en la verosimilitud logarítmica deshace el $\exp()$ de la ecuación softmax de la siguiente manera:

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

El primer término de la ecuación anterior muestra que la entrada z_i siempre tiene una contribución directa a la función de coste. Como este término no puede saturar, sabemos que el aprendizaje puede continuar, incluso si la contribución de z_i al segundo término de la ecuación anterior se vuelve muy pequeña. Al maximizar la probabilidad logarítmica, el primer término alienta z_i a ser empujado hacia arriba, mientras que el segundo término codifica todo z para que sea empujado hacia abajo. Por lo que respecta al segundo término, $\log \sum_j \exp(z_j)$, se puede observar que este término puede ser aproximado por $\max_j z_j$. Esta aproximación se basa en la idea de que $\exp(z_k)$ es insignificante para cualquier z_k que sea notablemente menor que $\max_j z_j$. La intuición que podemos obtener de esta aproximación es que la función de coste siempre penaliza con mayor fuerza la predicción incorrecta con más nivel de activación. Si una respuesta correcta ya proporciona la mayor entrada a la capa softmax, entonces el término $-z_i$ y los términos $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ se cancelarán de manera aproximada. Este ejemplo contribuirá poco al coste total durante el proceso de entrenamiento, que estará dominado por otros ejemplos que aún no están clasificados correctamente.

Desde un punto de vista neurocientífico, es interesante pensar en la capa softmax como una competición entre las neuronas que participan en ella: las salidas de softmax siempre suman 1, por lo que un aumento en el valor de una unidad necesariamente corresponde a una disminución en el valor de las demás neuronas. Esto es análogo a la inhibición lateral que se cree que existe entre las neuronas cercanas en la corteza cerebral.

El nombre softmax puede ser algo confuso. La función está más relacionada con la función $\text{argmax}()$ que la función $\text{max}()$. El término *soft* proviene del hecho que la función softmax es continua y diferenciable. La función $\text{argmax}()$, con su resultado representado como un vector único, no es continua ni diferenciable. La función softmax proporciona una versión suavizada del $\text{argmax}()$.

2.2.3. Unidades ocultas

Hasta ahora, hemos centrado nuestra discusión en las opciones de diseño para redes neuronales que son comunes a la mayoría de los modelos de aprendizaje automático paramétricos entrenados con optimización basada en gradiente. Ahora pasamos a un problema que es exclusivo de la aproximación de redes neuronales: cómo elegir el tipo de unidad oculta o función de activación en capas intermedias de la red.

El diseño de unidades ocultas es un área de investigación extremadamente activa y todavía no tiene muchos principios teóricos definitivos.

Las **unidades lineales rectificadas** (ReLU, de *Rectified Linear Units*) son una excelente opción predeterminada de unidad oculta, aunque existen muchos otros tipos de unidades ocultas. Puede ser difícil determinar en qué momento usar cada tipo (aunque las ReLU suelen ser una opción aceptable). En este apartado se describen algunas de las intuiciones básicas que motivan cada tipo de unidad oculta. Estas intuiciones pueden ayudar a decidir en qué momento probar cada unidad. Predecir de antemano cuál funcionará mejor suele ser imposible. El proceso de diseño consiste en un ejercicio de prueba y error, es decir, en intuir qué tipo de unidad oculta puede funcionar bien, y luego entrenar una red con ese tipo de unidad oculta y evaluar su rendimiento en un conjunto de validación.

Algunas de las unidades ocultas incluidas en esta lista no son realmente diferenciables en todos los puntos de entrada. Por ejemplo, la función ReLU $g(z) = \max\{0, z\}$ no es diferenciable en $z = 0$. Esto puede parecer que invalida g para su uso en un algoritmo de aprendizaje basado en gradiente. En la práctica, el descenso por gradiente funciona suficientemente bien para que estos modelos se utilicen para tareas de aprendizaje automático. Esto se debe, en parte, a que los algoritmos de entrenamiento de redes neuronales generalmente no llegan al mínimo local de la función de coste, sino que simplemente reducen su valor significativamente, como se muestra en la Figura 10.



Figura 10. Representación del mínimo global y diversos mínimos locales de una función. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

Debido a que no esperamos que el entrenamiento alcance realmente un punto donde el gradiente sea 0, es aceptable que los mínimos de la función de coste correspondan a puntos con gradiente indefinido. Las unidades ocultas que no son diferenciables generalmente no son diferenciables en solo un pequeño número de puntos. En general, una función $g(z)$ tiene una derivada izquierda definida por la pendiente de la función inmediatamente a la izquierda de z y una derivada derecha definida por la pendiente de la función inmediatamente a la derecha de z .

Una función es diferenciable en z solo si tanto la derivada izquierda como la derivada derecha están definidas y son iguales entre sí. Las funciones utilizadas en el contexto de las redes neuronales generalmente tienen derivadas izquierdas definidas y derivadas derechas definidas. En el caso de $g(z) = \max\{0, z\}$, la derivada izquierda en $z = 0$ es 0, y la derivada derecha es 1.

Las implementaciones de software de entrenamiento de redes neuronales generalmente devuelven una de las derivadas unilaterales, en lugar de informar de que la derivada no está definida o genera un error. Esto puede justificarse heurísticamente al observar que la optimización basada en gradiente en una computadora está sujeta a errores numéricos de todos modos. Cuando se le pide a una función que evalúe $g(0)$, es muy poco probable que el valor subyacente sea realmente 0. En cambio, es probable que sea un pequeño valor ϵ que se haya redondeado a 0. El punto importante es que, en la práctica, se puede ignorar de forma segura la no diferenciabilidad de las funciones de activación de la unidad oculta que se describen a continuación.

A menos que se indique lo contrario, la mayoría de las unidades aceptan un vector de entradas x , calculan una transformación afín $z = W^T x + b$, y luego aplican una función no lineal $g(z)$. La mayoría de las unidades ocultas se distinguen entre sí solo por la elección de la forma de la función de activación $g(z)$.

Unidades lineales rectificadas y sus generalizaciones



Las unidades lineales rectificadas (ReLU, del inglés) utilizan la función de activación $g(z) = \max\{0, z\}$ que se representó en la Figura 8.

Estas unidades son fáciles de optimizar porque son muy similares a las unidades lineales. La única diferencia entre una unidad lineal y una unidad lineal rectificada es que esta última genera cero en la mitad de su dominio. Esto hace que las derivadas a través de una unidad lineal rectificada se caractericen por valores grandes siempre que la unidad esté activa. Los gradientes no solo son grandes, sino también consistentes. La segunda derivada de la ReLU es 0 en casi todas partes, y la primera derivada de la ReLU es 1 en todas las partes donde la unidad está activa. Esto significa que la dirección del gradiente es mucho más útil para el aprendizaje de lo que sería con funciones de activación que introducen efectos de segundo orden.

Las unidades lineales rectificadas se usan típicamente sobre una transformación afín:

$$h = g(W^T x + b)$$

Al inicializar los parámetros de la transformación afín, puede ser una buena práctica inicializar todos los elementos de b a un valor positivo pequeño, como 0,1. Al hacerlo, es muy probable que las ReLU estén inicialmente activas para la mayoría de las entradas en el conjunto de entrenamiento y permitan que pasen las derivadas.

Existen diferentes variantes de ReLU. La mayoría de estas variantes funcionan de manera comparable a la versión básica de la unidad lineal rectificada y, ocasionalmente, funcionan mejor.

Un inconveniente de las ReLU es que no pueden aprender a través de métodos basados en gradiente en ejemplos para los cuales su activación es cero. Diversas variantes de ReLU garantizan que reciban gradiente en todas partes.

Tres variantes ReLU se basan en el uso de una pendiente α_i distinta a cero cuando $z_i < 0$: $h_i = g(z_i, \alpha_i) = \max(0, z_i) + \alpha_i \min(0, z_i)$.

Esta rectificación del valor absoluto corrige $\alpha_i = -1$ para obtener $g(z) = |z|$. Se utiliza para el reconocimiento de objetos a partir de imágenes (Jarrett et al., 2009), donde tiene sentido buscar características que son invariantes bajo una inversión de polaridad de la iluminación de entrada. Otras generalizaciones de unidades lineales rectificadas son más ampliamente aplicables. Una ReLU con fugas o **LeakyReLU** (Maas, Hannun y Ng, 2013) fija α_i a un valor pequeño como 0,01, mientras que una ReLU paramétrica o **PReLU** trata α_i como un parámetro que se puede aprender (Kaiming, Xiangyu, Shaoqing y Jian, 2018).

Las unidades **maxout** (Goodfellow, Warde-Farley, Mirza, Courville y Bengio, 2013) generalizan aún más las ReLU. En lugar de aplicar una función de elemento $g(z)$, las unidades **maxout** dividen z en grupos de k valores. Cada unidad **maxout** genera el elemento máximo de uno de estos grupos como se muestra a continuación:

$$g(z)_i = \max_{j \in g^i} z_j$$

En la fórmula anterior, $g(i)$ es el conjunto de índices de las entradas para el grupo $i, \{(i-1)k+1, \dots, ik\}$. Esto proporciona una forma de aprender una función lineal por partes que responde a múltiples direcciones del espacio de entrada x .

Una unidad **maxout** puede aprender una función convexa lineal por partes de hasta k piezas. Por lo tanto, las unidades **maxout** pueden verse como el aprendizaje de la función de activación en sí, en lugar de como la relación entre las unidades. Con k lo suficientemente grande, una unidad **maxout** puede aprender a aproximar cualquier función convexa con fidelidad arbitraria. En particular, una capa **maxout** con dos piezas puede aprender a implementar la misma función de la entrada x que una capa tradicional utilizando la función ReLU básica, la función de rectificación de valor absoluto, la ReLU paramétrica, la LeakyReLU, o puede aprender a implementar una función totalmente diferente. La capa de **maxout**, por supuesto, se parametrizará de manera diferente a cualquiera de estos otros tipos de capa, por lo que la dinámica de aprendizaje será diferente incluso en los casos en que **maxout** aprenda a implementar la misma función de x que uno de los otros tipos de capa.

Cada unidad **maxout** ahora está parametrizada por k vectores de peso, en lugar de uno solamente, por lo que las unidades **maxout** generalmente necesitan más regularización que las unidades lineales rectificadas. Pueden funcionar bien sin regularización si el conjunto de entrenamiento es grande y el número de piezas por unidad se mantiene bajo (Cai, Shi y Liu, 2013).

Las unidades **maxout** tienen algunos otros beneficios. En algunos casos, uno puede obtener algunas ventajas estadísticas y computacionales al requerir menos parámetros. Específicamente, si las características capturadas por n filtros lineales diferentes se pueden resumir sin perder información al tomar el máximo sobre cada grupo de k características, entonces la siguiente capa puede pasar con k veces menos pesos.

Debido a que cada unidad es accionada por múltiples filtros, las unidades **maxout** tienen cierta redundancia que les ayuda a resistir a un fenómeno llamado **olvido catastrófico**, en el cual las redes neuronales olvidan cómo realizar tareas para las que fueron entrenadas en el pasado (Goodfellow, Mirza, Xiao, Courville y Bengio, 2013).

Las unidades lineales rectificadas y todas estas generalizaciones se basan en el principio de que los modelos son más fáciles de optimizar si su comportamiento es más cercano al lineal.

Este mismo principio general de usar el comportamiento lineal para obtener una optimización más fácil también se aplica en otros contextos, además de las redes secuenciales profundas. Las redes recurrentes pueden aprender de las secuencias y producir una secuencia de estados y salidas. Al entrenarlas, uno necesita propagar información a través de varios instantes temporales, lo que es mucho más fácil cuando se involucran algunos cálculos lineales (con algunas derivadas direccionales de magnitud cercana a 1). Una de las arquitecturas de red recurrentes de mejor rendimiento, la LSTM, propaga información a través del tiempo mediante sumas, un tipo particular de activación lineal directa. Esto se discute más a fondo en el Capítulo 5, sobre modelado de secuencias: redes recurrentes y recursivas.

Sigmoide y tangente hiperbólica



Antes de la aparición de la función ReLU, la mayoría de las redes neuronales usaban la siguiente función de activación **sigmoide**:

$$g(z) = \sigma(z)$$

O bien usaban la siguiente **tangente hiperbólica** como función de activación en sus capas intermedias:

$$g(z) = \tanh(z)$$

Estas funciones de activación están estrechamente relacionadas, porque $\tanh(z) = 2\sigma(2z) - 1$.

Ya hemos visto las unidades sigmoides como unidades de salida, usadas para predecir la probabilidad de que una variable binaria sea 1. A diferencia de las unidades lineales por partes, las unidades sigmoides saturan en la mayor parte de su dominio: saturan a un valor alto cuando z es muy positivo, saturan a un valor bajo cuando z es muy negativo, y solo son muy sensibles a su entrada cuando z está cerca de 0. La saturación generalizada de unidades sigmoides puede dificultar el aprendizaje basado en gradientes. Por esta razón, se desaconseja su uso como unidades ocultas en las redes prealimentadas. Su uso como unidades de salida es compatible con el uso del aprendizaje basado en gradientes cuando una función de coste apropiada puede deshacer la saturación de la sigmoide en la capa de salida.

Cuando se deba utilizar una función de activación sigmoide, la función de activación tangente hiperbólica generalmente funcionará mejor que la sigmoide logística. Se asemeja más a la función identidad, en el sentido de que $\tanh(0) = 0$, mientras que $\sigma(0) = 1/2$. Como \tanh es similar a la función identidad cerca de 0, entrenar una red neuronal profunda $\hat{y} = w^T \tanh(U^T \tanh(V^T x))$ es parecido a entrenar un modelo lineal $y = w^T U^T V^T x$, siempre que las activaciones de la red puedan mantenerse pequeñas. Esto facilita el entrenamiento de la red con \tanh .

Las funciones de activación sigmoide son más comunes en entornos distintos de las redes prealimentadas. Las redes recurrentes, muchos modelos probabilísticos y algunos *autoencoders*, tienen requisitos adicionales que descartan el uso de funciones de activación lineal por partes y hacen que las unidades sigmoides sean más atractivas, a pesar de los inconvenientes de la saturación.

2.2.4. Diseño de arquitectura

Otra consideración de diseño clave para las redes neuronales es determinar su arquitectura. La palabra *arquitectura* se refiere a la estructura general de la red: cuántas unidades debería tener y cómo estas unidades deberían conectarse entre sí.



La mayoría de las redes neuronales están organizadas en grupos de unidades llamadas *capas*. La mayoría de las arquitecturas de redes neuronales organizan estas capas en una estructura de cadena, en la que cada capa es una función sobre la capa que la precede. En esta estructura, la salida de la primera capa, también denominada **activación**, viene dada por la fórmula siguiente:

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

La activación de la segunda capa viene definida por esta fórmula:

$$h^{(2)} = g^{(2)}(W^{(2)T}h^{(1)} + b^{(2)})$$

Y así sucesivamente.

En estas arquitecturas basadas en cadenas, las principales consideraciones, en cuanto a arquitectura, son la elección de la profundidad de la red y el ancho de cada capa (número de neuronas/unidades por capa). Como veremos, una red con, incluso, una capa oculta es suficiente para adaptarse al conjunto de entrenamiento. Las redes más profundas a menudo pueden usar muchas menos unidades por capa y muchos menos parámetros, así como generalizar con frecuencia al conjunto de prueba, pero también tienden a ser más difíciles de optimizar. Se debe encontrar la arquitectura de red ideal para una tarea, a través de la experimentación guiada mediante la monitorización del error del conjunto de validación.



Enlace de interés

Este recurso interactivo sirve para comprender los fundamentos de las redes neuronales. Mediante una interfaz de usuario amigable, se pueden afianzar conceptos clave de la asignatura viendo cómo afectan de manera directa en el entrenamiento de una red neuronal empleando distintas distribuciones de datos de entrada.

<https://playground.tensorflow.org>

2.2.5. Propiedades y profundidad de aproximación universal

Un modelo lineal, es decir, el mapeo de características de entrada a salidas a través de la multiplicación matricial, por definición tan solo puede representar funciones lineales. Tiene la ventaja de ser fácil de entrenar porque muchas funciones de pérdida resultan en problemas de optimización convexa cuando se aplican a modelos lineales. Desafortunadamente, en la mayor parte de aplicaciones queremos que nuestros sistemas aprendan funciones no lineales.

A primera vista, podríamos suponer que aprender una función no lineal requiere diseñar una familia modelo especializada para el tipo de no linealidad que queremos aprender. Afortunadamente, las redes prealimentadas con capas ocultas proporcionan un marco de aproximación universal.

Especificamente, el **teorema de aproximación universal** (Cybenko, 1989; Hornik, Stinchcombe y White, 1989) afirma que una red prealimentada con una capa de salida lineal y, al menos, una capa oculta con cualquier función de activación (como la función de activación sigmoide logística) puede aproximar cualquier función medible de Borel de un espacio de dimensión finita a otro con cualquier cantidad de error que no sea cero, siempre que la red reciba suficientes unidades ocultas. Las derivadas de la red prealimentada también pueden逼近arse a las derivadas de la función de manera correcta (Hornik, Stinchcombe y White, 1990). El concepto de mensurabilidad de Borel va más allá del alcance de este manual; para nuestro propósito es suficiente decir que cualquier función continua en un subconjunto cerrado y acotado de \mathbb{R}^n es medible por Borel y, por lo tanto, puede ser逼近ada por una red neuronal. Una red neuronal también puede逼近ar cualquier mapeo de funciones desde cualquier espacio discreto dimensional finito a otro. Si bien los teoremas originales se expresaron por primera vez en términos de unidades con funciones de activación que saturan tanto para argumentos muy negativos como muy positivos, los teoremas de aproximación universal también se han demostrado para una clase más amplia de funciones de activación, que incluye la función ReLU explicada anteriormente (Leshno, Lin, Pinkus y Schocken, 1993).

El teorema de aproximación universal significa que, independientemente de la función que estemos tratando de aprender, sabemos que un perceptrón multicapa profundo podrá representar esta función. Sin embargo, no tenemos garantías de que el algoritmo de entrenamiento sea capaz de aprender esa función. Incluso si el perceptrón multicapa puede representar la función, el aprendizaje puede fallar por dos razones diferentes. Primero, el algoritmo de optimización utilizado para el entrenamiento puede que no sea capaz de encontrar el valor de los parámetros que corresponde a la función deseada. En segundo lugar, el algoritmo de entrenamiento podría elegir la función incorrecta como resultado de un **sobreajuste**. Recuerde de la sección “1.2.1. Regularización” que el teorema del “no existe almuerzo gratis” muestra que no hay un algoritmo de aprendizaje automático universalmente superior. Las redes prealimentadas proporcionan un sistema universal para representar funciones en el sentido de que, dada una función, existe una red prealimentada que se approxima a la función. Sin embargo, no hay un procedimiento universal para examinar un conjunto de entrenamiento de ejemplos específicos y elegir una función que generalice a puntos que no están en el conjunto de entrenamiento.

De acuerdo con el teorema de aproximación universal, existe una red lo suficientemente grande como para lograr el grado de precisión que deseamos, pero el teorema no dice cómo de grande será esta red. Barron (1993) proporciona algunos límites en el tamaño de una red de una sola capa necesaria para逼近ar una amplia clase de funciones. Desafortunadamente, en el peor de los casos, un número exponencial de unidades ocultas (posiblemente con una unidad oculta correspondiente a cada configuración de entrada que necesita ser distinguida) puede llegar a ser requerido.

En resumen, una red prealimentada con una sola capa es suficiente para representar cualquier función, pero la capa puede ser demasiado grande y puede fallar a la hora de aprender y generalizar correctamente. En muchas circunstancias, el uso de modelos más profundos puede reducir la cantidad de unidades requeridas para representar la función deseada y puede reducir la cantidad de error de generalización (error en el conjunto de prueba).



Capítulo 3

Regularización en el aprendizaje profundo

Un problema central en el aprendizaje automático es cómo hacer un algoritmo que funcione bien, no solo en los datos de entrenamiento, sino también ante nuevas muestras. Muchas estrategias utilizadas en el aprendizaje automático están diseñadas explícitamente para reducir el error de prueba, posiblemente a cambio de un mayor error durante el entrenamiento. Estas estrategias se conocen colectivamente como *regularización*. Hay muchas formas de regularización disponibles que se enmarcan dentro del paradigma del aprendizaje profundo. De hecho, desarrollar estrategias de regularización más efectivas ha sido uno de los principales esfuerzos de investigación en este campo.

En el Capítulo 1, sobre aprendizaje automático, se introdujeron los conceptos básicos de generalización, adaptación, sobreajuste, sesgo, varianza y regularización. Si aún no está familiarizado con estas nociones, consulte ese capítulo antes de continuar con este.

En este capítulo, describimos la regularización con más detalle, enfocándonos en estrategias de regularización para modelos profundos.

En la sección “1.2.1. Regularización”, definimos la regularización como cualquier modificación que hagamos en un algoritmo de aprendizaje que tenga la intención de reducir su error de generalización, a costa de un aumento en el error de entrenamiento. Existen muchas estrategias de regularización. Algunas técnicas imponen restricciones adicionales en un modelo de aprendizaje automático, como agregar restricciones a los valores de los parámetros.

Otras técnicas agregan términos adicionales en la función objetivo, que pueden considerarse como correspondientes a una restricción suave en los valores de los parámetros. Si se eligen con cuidado, estas restricciones y penalizaciones adicionales pueden mejorar el rendimiento del modelo sobre el conjunto de prueba. A veces, estas restricciones y penalizaciones están diseñadas para codificar tipos específicos de conocimiento previo. Otras veces, estos parámetros adicionales están diseñados para expresar una preferencia genérica por una clase de modelo más simple y promover así la generalización. En otras ocasiones, las penalizaciones y restricciones son necesarias para determinar un problema indeterminado. Otras formas de regularización, conocidas como *métodos de conjunto*, combinan múltiples hipótesis que explican los datos de entrenamiento.

En el contexto del aprendizaje profundo, la mayoría de las estrategias de regularización se basan en regularizar los estimadores. La regularización de un estimador funciona intercambiando un aumento en el sesgo por una reducción en la varianza. Un regularizador efectivo es aquel que hace un intercambio rentable, es decir, reduce la varianza significativamente sin aumentar excesivamente el sesgo. Cuando discutimos la generalización y el sobreajuste en el Capítulo 1, sobre aprendizaje automático, nos enfocamos en tres situaciones en las que se puede encontrar el modelo que se está entrenando: (1) se excluye el verdadero proceso de generación de datos, que se corresponde con un efecto de **subajuste**; (2) coincide con el verdadero proceso de generación de datos; o (3) se incluye el verdadero proceso de generación de datos, pero también muchos otros procesos de generación de datos posibles. En este último caso, nos encontraremos frente a un problema de **sobreajuste**, donde la varianza domina el error. El objetivo de la regularización es llevar un modelo del tercer al segundo régimen.

En la práctica, una familia de modelos demasiado compleja no necesariamente incluye la función objetivo o el verdadero proceso de generación de datos, ni siquiera una aproximación cercana de cualquiera de ellos. Casi nunca tenemos acceso al verdadero proceso de generación de datos, por lo que nunca podemos saber con certeza si la familia de modelos que se estima incluye el proceso de generación o no. Sin embargo, la mayoría de las aplicaciones en las que se involucran algoritmos de aprendizaje profundo se dirigen a dominios en los que el verdadero proceso de generación de datos queda fuera de la familia de modelos. Los algoritmos de aprendizaje profundo generalmente se aplican a dominios extremadamente complicados, como imágenes, secuencias de audio y texto, para los cuales el verdadero proceso de generación consiste esencialmente en simular todo el universo poblacional. Estamos tratando de encajar una clavija cuadrada (el proceso de generación de datos) en un agujero redondo (nuestra familia de modelos).

Controlar la complejidad del modelo no es una simple cuestión de encontrar el modelo de tamaño correcto, con el número correcto de parámetros, sino que podríamos encontrar, y de hecho en escenarios prácticos de aprendizaje profundo casi siempre encontramos, que el modelo que mejor se ajusta (en el sentido de minimizar el error de generalización o prueba) es un modelo grande que se ha regularizado adecuadamente. A continuación, se describen varias estrategias de regularización.

3.1. Penalización paramétrica de la norma

La regularización se ha utilizado durante décadas, desde mucho antes de la explosión del aprendizaje profundo. Los modelos lineales, como la regresión lineal y la regresión logística, permiten estrategias de regularización simples, directas y efectivas.

Muchos enfoques de regularización se basan en limitar la capacidad de los modelos, como las redes neuronales, la regresión lineal o la regresión logística, al agregar una penalización de la norma del parámetro $\Omega(\theta)$ a la función objetivo J . Denotamos la función objetivo regularizada por \tilde{J} :

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

En la fórmula anterior, $\alpha \in [0, \infty)$ es un hiperparámetro que pondera la contribución relativa del término de penalización de la norma, Ω , en relación con la función objetivo estándar, J . Establecer α igual a 0 no produce regularización. A mayores valores de α , mayor regularización.

Cuando nuestro algoritmo de entrenamiento minimiza la función objetivo regularizada \tilde{J} , disminuye tanto el objetivo original J en los datos de entrenamiento como alguna medida del tamaño de los parámetros θ (o algún subconjunto de los parámetros). Diferentes valores de la norma del parámetro Ω pueden dar como resultado diferentes soluciones. En esta sección, discutimos los efectos de las diversas normas cuando se usan como penalizaciones en los parámetros del modelo.

Antes de profundizar en el comportamiento de regularización de diferentes normas, observamos que, para las redes neuronales, generalmente elegimos usar una penalización de norma de parámetro Ω que penaliza solo los pesos de la transformación afín en cada capa y deja los sesgos sin regularizar. Los sesgos (*bias* en inglés) generalmente requieren menos datos que los pesos para ajustarse con precisión. Cada peso describe cómo interactúan dos variables. Ajustar bien el peso requiere observar ambas variables en una variedad de condiciones. Cada sesgo controla una sola variable. Esto significa que no inducimos demasiada varianza y dejamos los sesgos sin regularizar. Además, la regularización de los parámetros de sesgo podría inducir al subajuste. Por lo tanto, utilizamos el vector w para indicar todos los pesos que deberían verse afectados por una penalización de la norma, mientras que el vector θ denota todos los parámetros, incluyendo tanto los w como los parámetros no regularizados.

En el contexto de las redes neuronales, a veces es deseable usar una penalización separada con un coeficiente α diferente para cada capa de la red. Debido a que puede ser costoso buscar el valor correcto de múltiples hiperparámetros, es razonable usar la misma disminución de peso en todas las capas para reducir el espacio de búsqueda.

3.1.1. Regularización de parámetros L2

Como ya se introdujo en la sección “1.2.1. Regularización”, uno de los tipos más simples y más comunes de penalización es la penalización paramétrica de la norma L2, comúnmente conocida como ***weight decay***.

Esta estrategia de regularización traslada los pesos más cerca del origen al agregar un término de regularización $\Omega(\theta) = \frac{1}{2}w^2$ a la función objetivo. La regularización L2 también se conoce como **regularización de Tikhonov** o **regresión de arista** (*ridge regression* en inglés).

Podemos hacernos una idea del comportamiento de la regularización L2 al estudiar el gradiente de la función objetivo regularizada. Para simplificar la matemática, suponemos que no hay parámetro *bias*, por lo que θ es solo w . Dicho modelo tiene la siguiente función objetivo total:

$$\tilde{J}(w; X, y) = \frac{\alpha}{2}w^T w + J(w; X, y)$$

Y tiene el correspondiente parámetro gradiente:

$$\nabla_w \tilde{J}(w; X, y) = \frac{\alpha}{2} w^T w + \nabla_w J(w; X, y)$$

Para dar un solo paso de gradiente con el objetivo de actualizar los pesos, realizamos esta actualización:

$$w \leftarrow w - \epsilon (\alpha w + \nabla_w J(w; X, y))$$

También se puede escribir así:

$$w \leftarrow (1 - \epsilon \alpha) w - \epsilon \nabla_w J(w; X, y)$$

Podemos ver que añadir el término de penalización de peso ha modificado la regla de aprendizaje para reducir de forma multiplicativa el vector de peso por un factor constante en cada paso, justo antes de realizar la actualización de gradiente habitual.

3.1.2. Regularización L1

Si bien la *weight decay* L2 es la forma más común de reducir grados de libertad en el modelo, existen otras formas de penalizar el tamaño de sus parámetros. Otra opción es usar la regularización L1. Formalmente, la regularización L1 se define sobre los parámetros w del modelo de la siguiente forma:

$$\Omega(\theta) = \|w_1\| = \sum_i |w_i|$$

Es decir, como la suma de valores absolutos de los parámetros individuales. Ahora analicemos el efecto de la regularización L1 en el modelo de regresión lineal simple, sin parámetro de sesgo, que estudiamos en nuestro análisis de la regularización de L2. En particular, estamos interesados en establecer las diferencias entre las formas de regularización L1 y L2. Al igual que con la disminución de peso L2, la disminución de peso L1 controla la fuerza de la regularización escalando el parámetro de penalización Ω empleando un hiperparámetro positivo α . Por lo tanto, la función objetivo regularizada $\tilde{J}(w; X, y)$ viene dada por la siguiente fórmula:

$$\tilde{J}(w; X, y) = \alpha \|w_1\| + J(w; X, y)$$

Que tiene el gradiente correspondiente (en realidad, subgradiente):

$$\nabla_w \tilde{J}(w; X, y) = \alpha \text{sign}(w) + \nabla_w J(w; X, y)$$

En la fórmula anterior, $\text{sign}(w)$ es simplemente el signo de w aplicado por elementos.

Al estudiar la ecuación anterior, podemos observar de inmediato que el efecto de la regularización L1 es bastante diferente al de la regularización L2. Concretamente, podemos ver que la contribución de regularización al gradiente ya no se escala linealmente con cada w_i ; en cambio, es un factor constante con un signo igual al signo (w_i).



Enlace de interés

Este enlace de interés es muy útil para comprender en profundidad cómo funciona la penalización paramétrica de la norma. Contiene la implementación de dicho mecanismo.

<https://www.pyimagesearch.com/2016/09/19/understanding-regularization-for-image-classification-and-machine-learning>

3.2. Aumento sintético del conjunto de datos o *data augmentation*

La mejor manera de hacer que un modelo de aprendizaje automático generalice mejor es entrenarlo con más datos. Por supuesto, en la práctica, la cantidad de datos que tenemos es limitada. Una forma de solucionar este problema es **crear datos sintéticos** y agregarlos al conjunto de datos de entrenamiento. Para algunas tareas de aprendizaje automático, es razonablemente sencillo crear nuevos datos sintéticos.

Este enfoque es más sencillo cuando se trata de resolver una tarea de clasificación. Un clasificador necesita tomar una entrada x de múltiples dimensiones y mapearla a una identidad de categoría única y .

Esto significa que la tarea principal que enfrenta un clasificador es ser invariable para una amplia variedad de transformaciones. Podemos generar nuevos pares (x, y) fácilmente, simplemente hay que transformar las entradas x en nuestro conjunto de entrenamiento.

Este enfoque no es tan fácil de aplicar en muchas otras tareas. Por ejemplo, es difícil generar nuevos datos sintéticos para una tarea de estimación de densidad, a menos que ya hayamos resuelto dicho problema de estimación de densidad.



El aumento del conjunto de datos ha sido una técnica particularmente efectiva para un problema de clasificación específico: el reconocimiento de objetos. Las imágenes son de alta dimensionalidad y se caracterizan por un enorme grado de variabilidad, y muchos de estos efectos de variabilidad se pueden simular fácilmente. Las operaciones como realizar una translación de unos pocos píxeles en diferentes direcciones sobre las imágenes de entrenamiento a menudo pueden mejorar en gran medida la generalización, incluso si el modelo ya ha sido diseñado para ser parcialmente invariante a la translación mediante el uso de las técnicas de **convolución** y **agregación** (*pooling* en inglés) que se describen en el siguiente capítulo.

Muchas otras operaciones, como la rotación o el escalado de la imagen (véase la Figura 11), también han demostrado ser bastante eficaces.

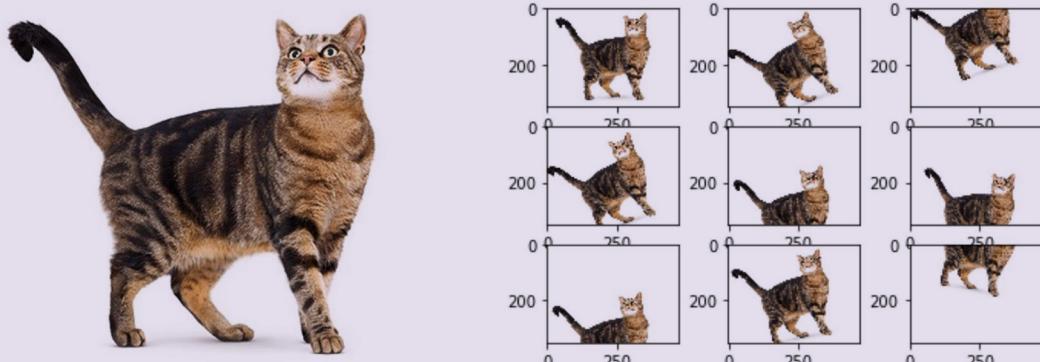


Figura 11. Representación gráfica de algunas transformaciones clave de la técnica de *data augmentation*. Recuperado de “Image Augmentation for Deep Learning”, por S. Lau, 2017, *Towards Data Science*. Disponible en <https://towardsdatascience.com/image-augmentation-for-deep-learning-histogram-equalization-a71387f609b2>

Hay que tener cuidado de no aplicar transformaciones que podrían llevar a un cambio de la clase correcta. Por ejemplo, las tareas de reconocimiento óptico de caracteres requieren reconocer la diferencia entre “b” y “d” y la diferencia entre “6” y “9”, por lo que las rotaciones horizontales y las rotaciones de 180° no son formas apropiadas de aumentar los conjuntos de datos para estas tareas.

También hay transformaciones en las que nos gustaría que nuestros clasificadores fueran invariantes, pero que no son fáciles de realizar. Por ejemplo, la rotación fuera del plano no se puede implementar como una operación geométrica simple sobre los píxeles de entrada.

El aumento del conjunto de datos también es efectivo para las tareas de reconocimiento de voz (Jaitly y Hinton, 2013).

La inyección de ruido en la entrada a una red neuronal (Sietsma y Dow, 1991) también puede verse como una forma de aumento de datos. Para muchas tareas de clasificación e incluso algunas de regresión, la tarea aún debería ser posible de resolver, incluso si se agrega un pequeño ruido aleatorio a la entrada. Sin embargo, las redes neuronales no son muy resistentes al ruido (Tang y Eliasmith, 2010). Una forma de mejorar la robustez de las redes neuronales es simplemente entrenarlas con ruido aleatorio aplicado a sus entradas.

La inyección de ruido de entrada forma parte de algunos algoritmos de aprendizaje no supervisados, como el *autoencoder* de eliminación de ruido (Vincent, Larochelle, Bengio y Manzagol, 2008). La inyección de ruido también funciona cuando el ruido se aplica sobre las unidades ocultas, lo que puede verse como un aumento del conjunto de datos en múltiples niveles de abstracción. Poole, Sohl-Dickstein y Ganguli (2014) mostraron recientemente que este enfoque puede ser altamente efectivo siempre que la magnitud del ruido se ajuste cuidadosamente. El *dropout*, una poderosa estrategia de regularización que se describirá en la sección “3.4. Dropout”, puede verse como un proceso de construcción de nuevas entradas a multiplicar por ruido.

Al comparar los resultados de referencia en la literatura del aprendizaje automático, es importante tener en cuenta el efecto del aumento del conjunto de datos.

A menudo, los esquemas de aumento de conjuntos de datos diseñados a mano pueden reducir drásticamente el error en prueba. Para comparar el rendimiento de un algoritmo de aprendizaje automático con otro, es necesario realizar experimentos controlados. Al comparar el algoritmo de aprendizaje automático A y el algoritmo de aprendizaje automático B, hay que asegurarse de que ambos algoritmos se evalúen utilizando los mismos esquemas de aumento de conjuntos de datos diseñados a mano. Supongamos que el algoritmo A funciona mal sin el uso de aumento de datos y que el algoritmo B funciona bien cuando se combina con numerosas transformaciones sintéticas de la entrada. En tal caso, las transformaciones sintéticas probablemente fueron la causa de la mejora del rendimiento, en lugar del algoritmo de aprendizaje automático B. A veces, decidir si un experimento se ha controlado adecuadamente requiere un juicio subjetivo.

Por ejemplo, los algoritmos de aprendizaje automático que inyectan ruido a la entrada están realizando una forma de aumento de conjunto de datos. Por lo general, las operaciones que generalmente son aplicables (como agregar ruido gaussiano a la entrada) se consideran parte del algoritmo de aprendizaje automático, mientras que las operaciones que son específicas de un dominio de aplicación (como recortar aleatoriamente una imagen) se consideran pasos de preprocesamiento.

3.3. Parada temprana

Al entrenar modelos grandes con capacidad de representación suficiente para sobreajustar la tarea, a menudo observamos que el error de entrenamiento disminuye constantemente con el tiempo, pero el error del conjunto de validación comienza a aumentar nuevamente. La Figura 12 muestra un ejemplo en el que se representa dicho comportamiento.

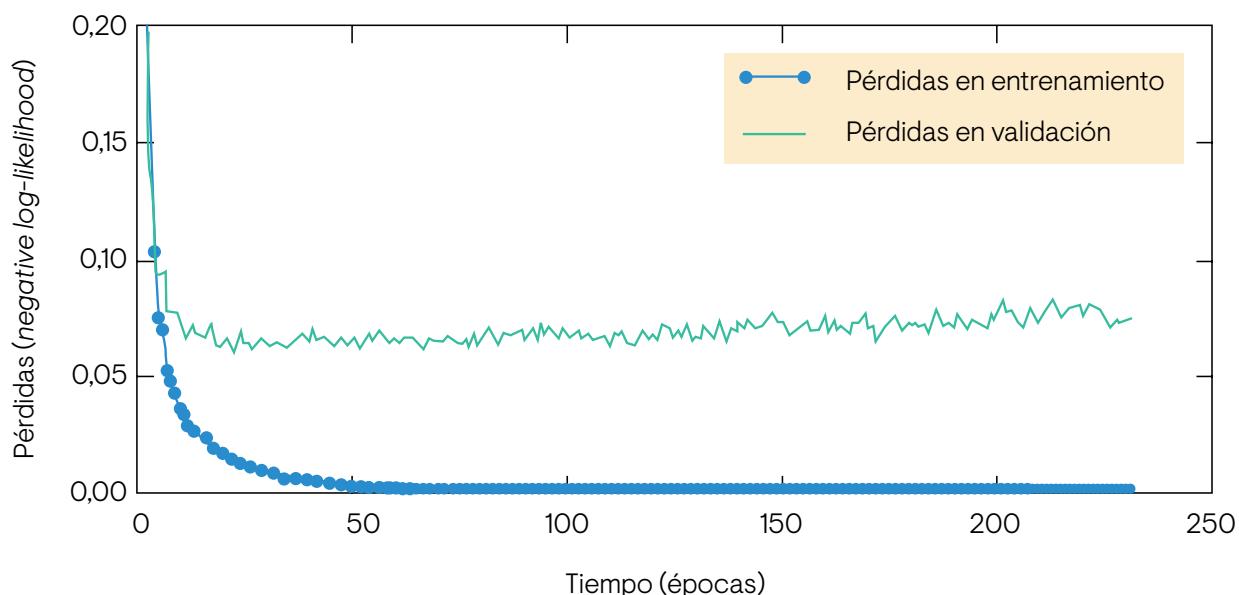


Figura 12. Representación de una curva de aprendizaje (entrenamiento y validación) en la que se evidencian síntomas de overfitting. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.



Este hecho significa que podemos obtener un modelo con un mejor error en el conjunto de validación (y por lo tanto, con suerte, un mejor error sobre el conjunto de prueba) cargando la configuración de parámetros del instante en el que se registra el valor más bajo de error en validación. Cada vez que mejora el error en el conjunto de validación, almacenamos una copia de los parámetros del modelo. Cuando termina el proceso de entrenamiento, devolvemos estos parámetros, en lugar de los últimos. El algoritmo termina cuando ningún parámetro ha mejorado el mejor valor de error registrado a lo largo de un número determinado de iteraciones previamente especificado.

Esta estrategia se conoce como **parada temprana** (*early stopping* en inglés). Probablemente es la forma de regularización más utilizada en el aprendizaje profundo. Su popularidad se debe tanto a su efectividad como a su simplicidad.

Una forma de pensar en la técnica de parada temprana es como un algoritmo de selección de hiperparámetros muy eficiente, ya que el número de pasos en entrenamiento es un hiperparámetro más.

Podemos ver en la Figura 12 que este hiperparámetro tiene una curva de rendimiento sobre el conjunto de validación en forma de “U”. La mayoría de los hiperparámetros que controlan la capacidad del modelo tienen esta forma, tal y como se ilustra en la Figura 2. En el caso de una detención temprana, estamos controlando la capacidad efectiva del modelo determinando cuántos pasos puede tomar para adaptarse al conjunto de entrenamiento. La mayoría de los hiperparámetros deben ser elegidos mediante un costoso proceso de prueba y error, donde establecemos un hiperparámetro al comienzo del entrenamiento y luego ejecutamos el entrenamiento durante varios pasos para ver su efecto.

El hiperparámetro “tiempo de entrenamiento” es único porque, por definición, una sola ejecución de entrenamiento prueba muchos valores del hiperparámetro. El único coste significativo para elegir este hiperparámetro automáticamente mediante una detención temprana es ejecutar una evaluación periódica sobre la validación durante el entrenamiento. Idealmente, esto se realiza en paralelo al proceso de entrenamiento en una máquina separada, una CPU separada o una GPU separada del proceso de entrenamiento principal. Si dichos recursos no están disponibles, entonces el coste de estas evaluaciones periódicas puede reducirse utilizando un conjunto de validación que sea pequeño en comparación con el conjunto de entrenamiento, o evaluando el error del conjunto de validación con menos frecuencia y obteniendo una estimación con una menor resolución temporal.

Un coste adicional para la técnica de detección temprana es la necesidad de mantener una copia de los mejores parámetros. Este coste generalmente es insignificante, porque es aceptable almacenar estos parámetros en una memoria más lenta y más grande (por ejemplo, se puede guardar el entrenamiento en la memoria de la GPU, pero almacenar los parámetros óptimos en la memoria del anfitrión, o *host*, o en una unidad de disco). Dado que los mejores parámetros se escriben con poca frecuencia y nunca se leen durante el entrenamiento, estas escrituras lentas ocasionales tienen poco efecto en el tiempo total de entrenamiento.

La parada temprana es una forma discreta de regularización, ya que casi no requiere cambios en el procedimiento de entrenamiento subyacente, la función objetivo o el conjunto de valores de parámetros permitidos. Esto significa que es fácil usarla sin influir en la dinámica de aprendizaje. Esto contrasta con la regularización *weight decay*, donde uno debe tener cuidado de no usar demasiado *weight decay* y atrapar la red en un mínimo local correspondiente a una solución con pesos pequeños.

La parada temprana se puede usar sola o junto con otras estrategias de regularización. Incluso cuando se utilizan estrategias de regularización que modifican la función objetivo para fomentar una mejor generalización, es raro que la mejor generalización ocurra en un mínimo local del objetivo del entrenamiento.

La parada temprana requiere un conjunto de validación, lo que significa que algunos datos de entrenamiento no se envían al modelo. Para aprovechar mejor estos datos adicionales, se puede realizar un entrenamiento adicional después de que se haya completado el entrenamiento inicial con parada temprana. En el segundo paso de entrenamiento adicional, se incluyen todos los datos de entrenamiento. Hay dos estrategias básicas que se pueden usar para este segundo procedimiento de entrenamiento.

Una estrategia es inicializar el modelo nuevamente y volver a entrenar con todos los datos. En este segundo proceso de entrenamiento, entrenamos para el mismo número de **épocas**, ya que el procedimiento de parada temprana determinado fue óptimo en el primer proceso de entrenamiento. Hay algunas sutilezas asociadas con este procedimiento.

Es importante notar que, en la segunda ronda de entrenamiento, cada pasada a través del conjunto de datos requerirá más actualizaciones de parámetros, porque el conjunto de entrenamiento es más grande.

Otra estrategia para usar todos los datos es mantener los parámetros obtenidos de la primera ronda de entrenamiento y luego continuar el entrenamiento, pero ahora usando todos los datos.

En esta etapa, ahora ya no tenemos una guía sobre cuándo parar en términos de una serie de pasos. En cambio, podemos monitorizar la función de pérdidas promedio en el conjunto de validación y continuar el entrenamiento hasta que caiga por debajo del valor que se obtuvo para el conjunto de entrenamiento cuando el procedimiento de parada se detuvo en la primera fase de entrenamiento.

Esta estrategia evita el alto coste de volver a entrenar el modelo desde cero, pero no se comporta tan bien. Por ejemplo, el objetivo en el conjunto de validación puede no alcanzar el valor objetivo, por lo que ni siquiera se garantiza que esta estrategia termine.

La técnica de parada temprana también es útil porque reduce el coste computacional del entrenamiento. Además de la reducción obvia en el coste debido a la limitación del número de iteraciones de entrenamiento, también tiene el beneficio de proporcionar regularización sin requerir la adición de términos de penalización a la función de coste o el cálculo de los gradientes de dichos términos adicionales.

3.4. Dropout



La técnica de *dropout* (Srivastava, Hinton, Krizhevsky y Salakhutdinov, 2014) proporciona un método computacionalmente poco costoso, pero muy poderoso, para regularizar una amplia familia de modelos. Como primera aproximación, el *dropout* puede considerarse un método para hacer que el *bagging* (o empaquetado) sea práctico para conjuntos de muchas redes neuronales grandes.

El *bagging* implica entrenar múltiples modelos y que cada uno de ellos evalúe cada uno de los ejemplos de prueba durante la fase de inferencia. Esto parece poco práctico cuando cada modelo es una gran red neuronal, ya que entrenar y evaluar dichas redes es costoso en términos de tiempo de ejecución y memoria. Es común usar conjuntos de cinco a diez redes neuronales: Szegedy et al. (2014) utilizaron seis para ganar el ILSVRC (ImageNet Large Scale Visual Recognition Challenge), pero esto se vuelve rápidamente difícil de manejar.

Más concretamente, el *dropout* entrena el conjunto de subredes que se pueden formar eliminando unidades que no son de salida de una red base subyacente, como se ilustra en la Figura 13.

En la mayoría de las redes neuronales actuales, basadas en una serie de transformaciones afines y no linealidades, podemos eliminar una neurona de una red multiplicando su valor de salida por cero.

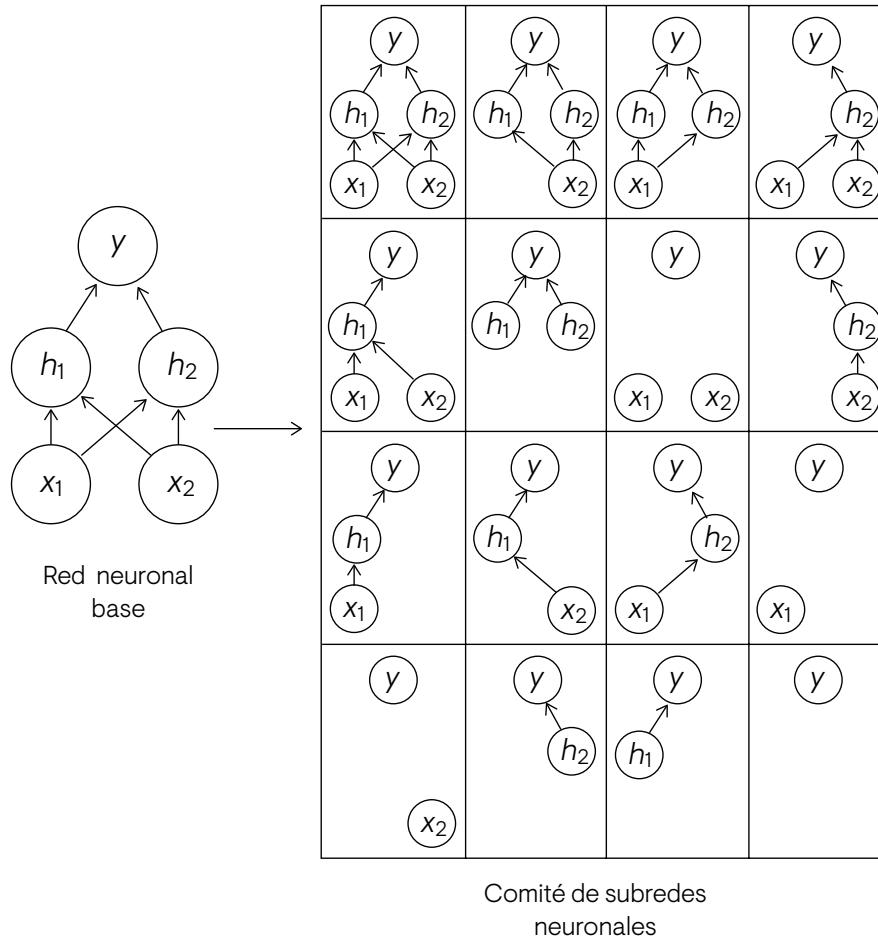


Figura 13. Representación gráfica de la técnica *dropout*. Diferentes subredes se crean para satisfacer el entrenamiento de una red neuronal base empleando dicha técnica. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

Cuando desarrollamos un algoritmo de *bagging*, definimos k diferentes modelos, construimos k diferentes conjuntos de datos mediante muestreo con reemplazo del conjunto de entrenamiento, y luego entrenamos el modelo i con el conjunto de datos i . El objetivo del *dropout* es aproximar este proceso, empleando un número exponencialmente grande de redes neuronales en nuestro *bagging*. En particular, para entrenar con *dropout*, utilizamos un algoritmo de aprendizaje basado en *minibatch* que realiza pequeños pasos, como el descenso de gradiente estocástico.

Cada vez que cargamos un ejemplo en un *minibatch*, muestreamos aleatoriamente una máscara binaria diferente a aplicar a todas las unidades de entrada y ocultas de la red. La máscara para cada unidad se muestrea independientemente de todas las demás. La probabilidad de muestrear un valor de máscara de uno (haciendo que se incluya una unidad) es un hiperparámetro fijado antes de que comience el entrenamiento. No es una función del valor actual de los parámetros del modelo o del ejemplo de entrada. Por lo general, se incluye una unidad de entrada con probabilidad 0,8 y una unidad oculta con probabilidad 0,5. Luego ejecutamos la **propagación hacia adelante**, la propagación hacia atrás y la actualización de pesos como de costumbre. La Figura 14 ilustra cómo ejecutar la propagación hacia adelante empleando el *dropout*.

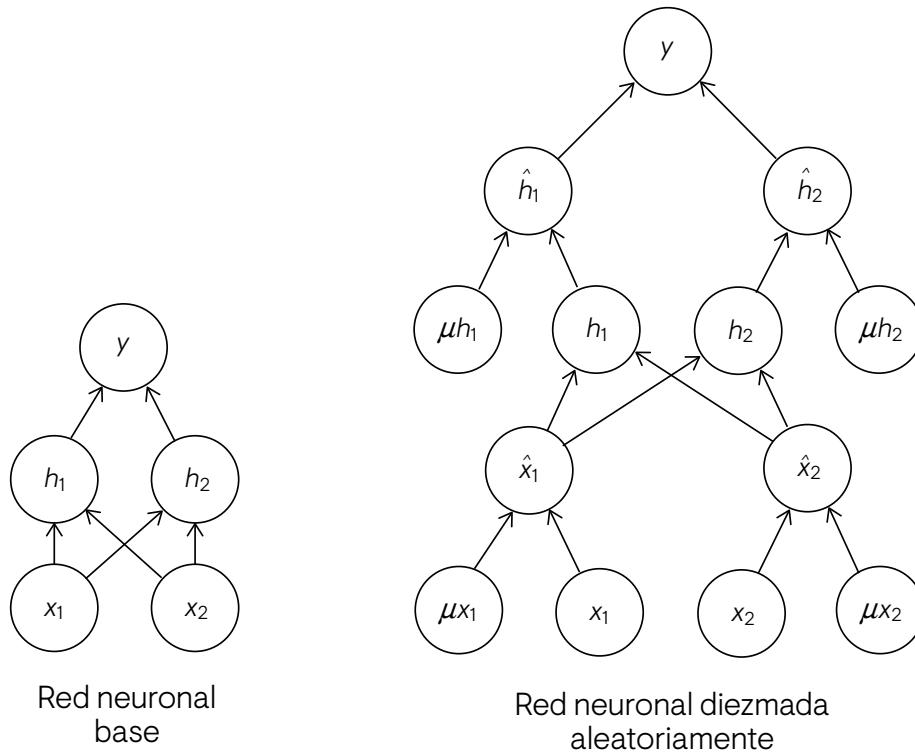


Figura 14. Efecto de diezmado aleatorio en los parámetros a optimizar de la red neuronal. Necesario para llevar a cabo la propagación hacia delante durante el proceso de entrenamiento. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

Más formalmente, supongamos que un vector de máscara μ especifica qué neuronas incluir, y $J(\theta, \mu)$ define el coste del modelo definido por los parámetros θ y máscara μ . El entrenamiento empleando el *dropout* consiste en minimizar $E_\mu J(\theta, \mu)$. El valor esperado contiene un número de términos exponencial, pero podemos obtener una estimación imparcial de su gradiente mediante el muestreo de valores de μ .

El entrenamiento con *dropout* no es lo mismo que el entrenamiento empleando *bagging*. En el caso del *bagging*, los modelos son todos independientes. En el caso del *dropout*, los modelos comparten parámetros, y cada modelo hereda un subconjunto diferente de parámetros de la red neuronal principal. Este intercambio de parámetros permite representar un número exponencial de modelos con una cantidad de memoria manejable. En el caso del *bagging*, cada modelo está entrenado para converger en su respectivo conjunto de entrenamiento. En el caso del *dropout*, generalmente la mayoría de los modelos no están entrenados explícitamente. Una pequeña fracción de las subredes se entrena para un solo paso, y el intercambio de parámetros hace que las subredes restantes lleguen a una buena configuración de los parámetros. Estas son las únicas diferencias. Más allá de esto, el *dropout* sigue el algoritmo de *bagging*. Por ejemplo, el conjunto de entrenamiento empleado por cada subred es un subconjunto del conjunto de entrenamiento original muestreado con reemplazo.

En la fase de prueba, un comité *bagging* debe acumular votos de todos sus miembros. Nos referimos a este proceso como *inferencia* en este contexto. Hasta ahora, nuestra descripción de *bagging* y *dropout* no ha requerido que el modelo sea explícitamente probabilístico. Ahora, supongamos que la función del modelo es generar una distribución de probabilidad.

En el caso de *bagging*, cada modelo i produce una distribución de probabilidad $p(i)(y|x)$. La predicción del conjunto viene dada por la media aritmética de todas estas distribuciones:

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

En el caso del *dropout*, cada submodelo definido por un vector máscara μ define una distribución de probabilidad $p(y|x,\mu)$. La media aritmética sobre todas las máscaras viene dada por la fórmula siguiente:

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

En la fórmula anterior, $p(\mu)$ es la distribución de probabilidad que fue empleada para muestrear μ durante el proceso de entrenamiento.



Capítulo 4

Redes neuronales convolucionales

Las **redes convolucionales** (Lecun, 1989), también conocidas como **redes neuronales convolucionales**, o CNN del inglés, son un tipo especializado de red neuronal para procesar datos que tienen una topología en cuadrícula o matriz. Los ejemplos incluyen datos de series temporales, que pueden considerarse como una cuadrícula 1D que toma muestras a intervalos de tiempo regulares, y datos de imágenes, que pueden considerarse como una cuadrícula de píxeles 2D. Las redes convolucionales han tenido un éxito tremendo en aplicaciones prácticas. El nombre *red neuronal convolucional* indica que la red emplea una operación matemática llamada *convolución*. La convolución es un tipo especializado de operación lineal. Las redes convolucionales son simplemente redes neuronales que utilizan la convolución, en lugar de la multiplicación matricial general, en al menos una de sus capas.

En este capítulo, en primer lugar, describimos qué es la convolución. A continuación, explicamos la motivación detrás del uso de la convolución en una red neuronal. Por lo general, la operación utilizada en una red neuronal convolucional no corresponde exactamente a la definición de convolución tal como se usa en otros campos, como la ingeniería o las matemáticas puras. Posteriormente describimos una operación llamada *pooling*, que emplean casi todas las redes convolucionales. El objetivo de este capítulo en el presente manual es describir las herramientas para diseñar redes convolucionales, mientras que en las videoconferencias de la asignatura se describen las arquitecturas de las redes neuronales convolucionales más importantes de la literatura, así como las pautas generales para diseñar y entrenar la red neuronal convolucional de manera práctica.

La investigación en arquitecturas de red convolucionales avanza tan rápido que, cada semana o mes, se da con una mejor arquitectura para resolver cierto problema. Es por ello que no tiene cabida dejar plasmada en este documento ninguna arquitectura de red como tal. No obstante, las mejores arquitecturas de red de la literatura se han compuesto mediante los bloques de construcción aquí descritos.

4.1. La operación de convolución

Supongamos que estamos rastreando la ubicación de una nave espacial mediante un sensor láser. Nuestro sensor láser proporciona una salida única $x(t)$, la posición de la nave espacial en el instante de tiempo t . Tanto x como t tienen un valor real, es decir, podemos obtener una lectura diferente del sensor láser en cualquier instante temporal.

Ahora suponga que nuestro sensor láser es algo ruidoso. Para obtener una estimación menos ruidosa de la posición de la nave espacial, nos gustaría promediar varias muestras. Por supuesto, las medidas más recientes son más relevantes, por lo que queremos que sea un promedio ponderado que otorgue más peso a las medidas recientes. Podemos hacer esto con una función de ponderación $w(a)$, donde a mide lo reciente o antigua que es una medida. Si aplicamos una operación promedio ponderada en cada momento, obtenemos una nueva función que proporciona una estimación suavizada de la posición de la nave espacial:

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

Esta operación se llama **convolución**. La operación de convolución se denota típicamente mediante un asterisco:

$$s(t) = (x * w)(t)$$

En nuestro ejemplo, w debe ser una función de densidad de probabilidad válida, o la salida no será un promedio ponderado. Además, w debe ser 0 para todos los argumentos de entrada negativos. Sin embargo, estas limitaciones son particulares de nuestro ejemplo. En general, la convolución viene dada para cualquier función para la que la integral anterior esté definida y puede usarse para otros fines, además de tomar promedios ponderados.

En la terminología de red convolucional, el primer argumento (en este ejemplo, la función x) en la convolución se denomina **entrada**, y el segundo argumento (en este ejemplo, la función w), **núcleo** o **kernel**. El resultado de la convolución de la salida a veces se denomina **mapa de características** o **activaciones** (*activation map* en inglés).

En nuestro ejemplo, la idea de un sensor láser que pueda proporcionar mediciones a cada instante no es realista. Por lo general, cuando trabajamos con datos en un ordenador, el tiempo se muestrea y nuestro sensor proporciona datos a intervalos regulares. En nuestro ejemplo, podría ser más realista suponer que el láser proporciona una medición una vez por segundo. El índice de tiempo t puede tomar solo valores enteros. Si ahora suponemos que x y w se definen solo en el entero t , podemos definir la convolución discreta de la siguiente forma:

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

En las aplicaciones de aprendizaje automático, la entrada suele ser una matriz multidimensional de datos, y el *kernel* suele ser una matriz multidimensional de parámetros a optimizar mediante el algoritmo de aprendizaje. Nos referiremos a estas matrices multidimensionales como *tensores*.

Debido a que cada elemento de la entrada y el núcleo deben almacenarse explícitamente por separado, generalmente asumimos que estas funciones son cero en todas partes, menos en el conjunto finito de puntos para los que almacenamos los valores.

Esto significa que, en la práctica, podemos implementar la suma infinita como una suma sobre un número finito de elementos de la matriz.

Finalmente, a menudo usamos convoluciones sobre más de un eje a la vez. Por ejemplo, si usamos una imagen bidimensional / como entrada, probablemente también queramos usar un núcleo bidimensional K :

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

La convolución es commutativa, lo que significa que podemos escribir de manera equivalente la siguiente ecuación:

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

Por lo general, la última fórmula es más sencilla de implementar en una biblioteca de aprendizaje automático, porque hay menos variación en el rango de valores válidos de m y n .



La propiedad commutativa de la convolución surge porque hemos volteado el *kernel* en relación con la entrada, en el sentido de que a medida que m aumenta, el índice en la entrada aumenta, pero el índice en el *kernel* disminuye. La única razón para voltear el núcleo es demostrar la propiedad commutativa. Mientras que la propiedad commutativa es útil para describir teóricamente la operación convolución, no suele ser una propiedad importante en la implementación de la red neuronal. De hecho, la mayoría de bibliotecas de redes neuronales implementan una función relacionada llamada **correlación cruzada**, que es lo mismo que convolución, pero sin voltear el *kernel*:

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

Muchas bibliotecas de aprendizaje automático implementan la correlación cruzada, pero la llaman *convolución*. En este texto seguimos esta convención de llamar a ambas operaciones *convolución* y especificamos si queremos voltear el *kernel* o no en contextos donde el volteo del *kernel* es relevante.

>>>

>>>

La Figura 15 muestra un ejemplo de convolución (sin volteo del núcleo) aplicado a un tensor 2D.

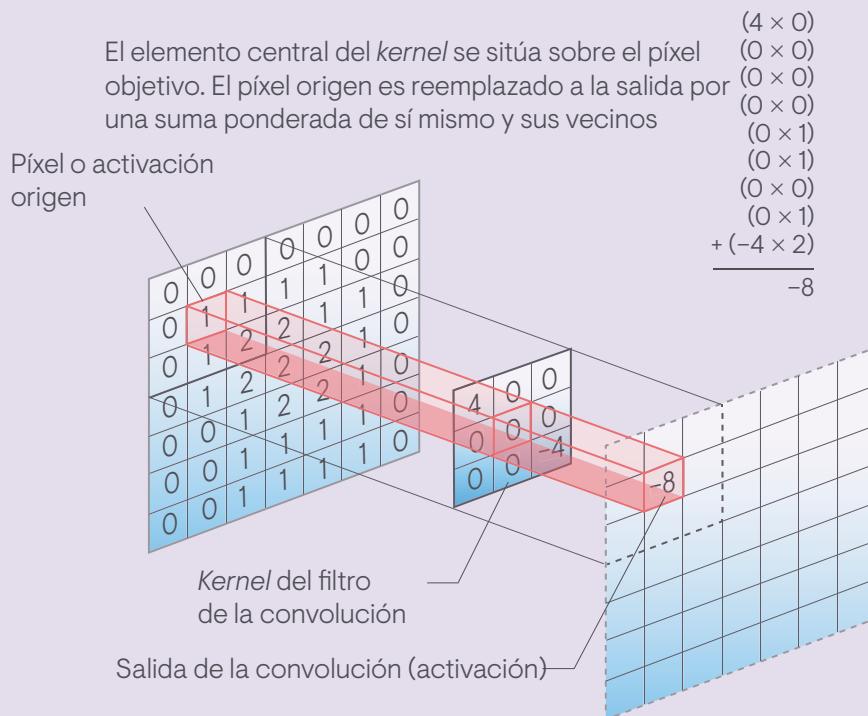


Figura 15. Representación de la operación convolución en una red neuronal convolucional. El *kernel* se desplaza a través de toda la imagen (o mapa de activaciones) para llevar a cabo dicha operación.

Es importante destacar también que, cuando nos referimos a la convolución en el contexto de las redes neuronales, generalmente nos referimos a una operación que consiste en aplicar muchas convoluciones en paralelo. Esto se debe a que la convolución con un solo *kernel* puede extraer solo un tipo de característica, aunque en muchas ubicaciones espaciales. Por lo general, queremos que cada capa convolucional de nuestra red extraiga muchos tipos de características, en múltiples ubicaciones espaciales.

Además, la entrada generalmente no es solo una cuadrícula de valores reales; más bien es una cuadrícula de observaciones con valores vectoriales. Por ejemplo, una imagen en color tiene un valor de intensidad en el canal rojo, otro en el verde y otro en el azul, en cada píxel. En una red convolucional de múltiples capas, la entrada a la segunda capa es la salida de la primera capa, que generalmente se compone como la salida de muchas convoluciones diferentes (empleando varios *kernels* o filtros) en cada posición.

Cuando trabajamos con imágenes, generalmente pensamos que la entrada y la salida de la convolución son tensores tridimensionales, con un índice referente a los diferentes canales y dos índices por lo que respecta a las coordenadas espaciales de cada canal. Pero lo cierto es que las implementaciones software generalmente funcionan procesando las imágenes o instancias de **entrenamiento por lotes (batches)**. Es por ello por lo que en realidad se emplean tensores 4D, en los que el cuarto eje indexa diferentes ejemplos por **batch**. Esta cuarta dimensión la omitiremos en nuestra descripción por simplicidad.

Debido a que las redes convolucionales normalmente usan la convolución multicanal, no se garantiza que las operaciones lineales en las que se basan sean conmutativas, incluso si se usa el volteo de *kernel*. Estas operaciones multicanal solo son conmutativas si cada operación tiene el mismo número de canales de salida que los canales de entrada.

Supongamos que tenemos un tensor de *kernel* 4D K con el elemento $K_{i,j,k,l}$ que proporciona el peso de una unidad en el canal i de salida y una unidad en el canal j de entrada, con un desplazamiento de k filas y l columnas entre la unidad de salida y la unidad de entrada. Suponga que nuestra entrada consiste en datos observados V con el elemento $V_{i,j,k}$ que da el valor de la unidad de entrada dentro del canal i en la fila j y la columna k . Suponga que nuestra salida Z tiene el mismo formato que V . Si Z se obtiene al convolucionar K con V sin voltear K , entonces tenemos la fórmula siguiente:

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

En esta suma, el sumatorio sobre l, m y n se lleva a cabo sobre todos los valores para los que las operaciones de indexación sobre los tensores son válidas. En notación de álgebra lineal, la indexación de matrices es 1 para la primera entrada; es por ello que se requiere el -1 en la fórmula anterior. En lenguajes de programación como C o Python la indexación comienza en 0, por lo que se simplifica la ecuación anterior.



Es posible que deseemos omitir algunas posiciones en las que deslizar el *kernel* para reducir el coste computacional (a expensas de no extraer nuestras características de manera tan precisa). Se puede pensar en esto como una forma de disminuir las dimensiones espaciales de la entrada a la salida mediante la capa convolucional. Si queremos muestrear solo cada z píxeles en cada dirección hacia la salida, entonces podemos definir una función de convolución diezmada c , como la siguiente:

$$Z_{i,j,k} = c(K, V, s)_{i,j,k} \sum_{l,m,n} [V_{l,(j-1)s+m, (k-1)s+n} K_{i,l,m,n}]$$

Nos referimos a s como el paso (**stride** en inglés) de esta convolución diezmada. También es posible definir un valor de *stride* (s) distinto para cada dirección de movimiento. Vea la Figura 16 para una mejor comprensión de este concepto:

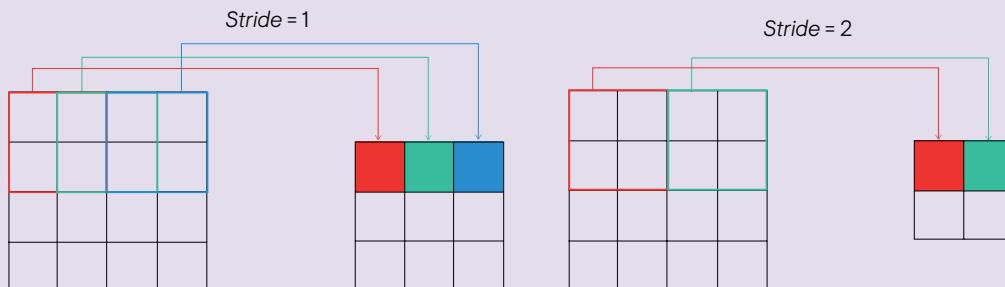


Figura 16. Efecto del parámetro *stride* durante la operación convolución. Las dimensiones del mapa de activación a la salida disminuyen conforme aumenta el *stride*. Adaptado de “Intuitive understanding of 1D, 2D, and 3D convolutions in convolutional neural networks”, por thushv89, StackOverflow. Recuperado de <https://stackoverflow.com/questions/42883547/intuitive-understanding-of-1d-2d-and-3d-convolutions-in-convolutional-neural-n>

>>>

>>>

Una característica esencial de cualquier implementación de red convolucional es la capacidad de rellenar con ceros (**zero padding** en inglés) los bordes de la entrada V para ampliarla y no perder la información que contienen. Sin esta característica, el ancho del **mapa de activación** de salida se reduce un píxel menos que el ancho del *kernel* en cada capa. El *zero padding* de la entrada nos permite controlar el ancho del *kernel* y el tamaño de la salida de forma independiente. Sin *zero padding*, nos vemos obligados a elegir entre reducir la extensión espacial de la red rápidamente o usar *kernels* pequeños: ambos escenarios limitan significativamente el poder expresivo de una red neuronal convolucional. Consulte la Figura 17 para ver un ejemplo.

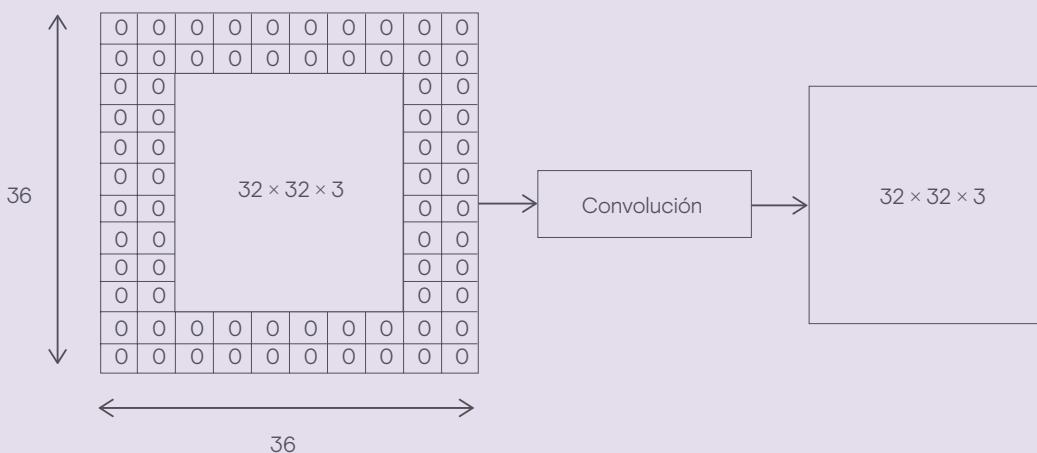


Figura 17. Representación gráfica de la técnica zero padding cuando se escoge la opción *same*.

Vale la pena mencionar tres casos especiales de la configuración *zero padding*. Uno es el caso extremo en el que no se utiliza ningún relleno de ceros y el *kernel* de convolución solo puede visitar aquellas posiciones donde todo el núcleo está contenido por completo dentro de la imagen. En la terminología práctica, esta convolución se llama **valid**. En este caso, todos los píxeles a la salida son una función del mismo número de píxeles a la entrada, por lo que el comportamiento de un píxel de salida es algo más regular. Sin embargo, el tamaño de la salida se reduce tras cada capa convolucional. Si la imagen de entrada tiene ancho m y el núcleo tiene ancho k , el ancho de salida será $m - k + 1$. La ratio de esta reducción puede ser dramática si los núcleos utilizados son muy grandes. Como la reducción es mayor que 0, el número de capas convolucionales que se pueden incluir en la red estará limitado. A medida que se agregan capas, llegará un momento en el que la dimensión espacial de la red caerá a 1. A partir de ahí, las capas adicionales no pueden considerarse convolucionales.

Otro caso especial de la configuración de *zero padding* es cuando se aplica el relleno suficiente para mantener el tamaño de la salida igual al de la entrada. En la práctica, este tipo de convolución se denomina **same**. En este caso, la red puede contener tantas capas convolucionales como el *hardware* disponible pueda soportar, ya que la operación de convolución no modifica las posibilidades de arquitectura disponible en la siguiente capa. Sin embargo, los píxeles de entrada cerca del borde influyen en menos píxeles de salida que los píxeles de entrada que se encuentran en el centro. Esto puede hacer que los píxeles del borde estén subrepresentados en el modelo. Esto motiva el otro caso extremo, al que en la práctica nos referimos como **convolución completa o full**.

En el último caso, en la convolución completa, se agregan suficientes ceros por cada píxel que se visita k veces en cada dirección, lo que resulta en una imagen de salida de ancho $m + k - 1$. En este caso, los píxeles de salida cerca del borde son una función de menos píxeles que los píxeles de salida cerca del centro. Esto puede dificultar el aprendizaje de un único *kernel* que funcione bien en todas las posiciones del mapa de características convolucional.

Por lo general, la cantidad óptima de relleno cero se encuentra entre la convolución *valid* y *same*.



Enlaces de interés

El primer enlace de interés ofrece un banco de juegos para comprender los entresijos de las redes neuronales convolucionales. Es posible visualizar los mapas de activación que produce el proceso de inferencia ante una nueva imagen de test o visualizar los pesos de los filtros más representativos. Para comprender el funcionamiento de este banco de juegos es importante leer detenidamente el segundo enlace de interés.

<https://convnetplayground.fastforwardlabs.com>

<https://towardsdatascience.com/convnetplayground-979d441ebf82>

4.2. Motivación

La convolución aprovecha tres ideas importantes que pueden ayudar a mejorar un sistema de aprendizaje automático: **interacciones dispersas, intercambio de parámetros y representaciones equivariantes**. Además, la convolución proporciona mecanismos para trabajar con entradas de tamaño variable. A continuación, describimos cada una de estas ideas.

Las capas de redes neuronales tradicionales emplean la multiplicación de matrices definiendo una matriz de parámetros que describe la interacción entre cada unidad de entrada y cada unidad de salida. Esto significa que todas y cada una de las unidades de salida interactúan con todas y cada una de las unidades de entrada. Sin embargo, las redes convolucionales suelen tener interacciones dispersas (también conocidas como **conectividad dispersa** o **pesos dispersos**). Esto se logra haciendo que el *kernel* sea más pequeño que la entrada.

Por ejemplo, al procesar una imagen, la imagen de entrada puede tener miles o millones de píxeles, pero podemos detectar características pequeñas y significativas, como bordes con núcleos que ocupan solo decenas o cientos de píxeles. Esto significa que necesitamos almacenar menos parámetros, lo que reduce los requisitos de memoria del modelo y mejora su eficiencia estadística. También significa que calcular la salida requiere menos operaciones. Estas mejoras en la eficiencia suelen ser bastante grandes. Si hay m entradas y n salidas, entonces la multiplicación de la matriz requiere $m \times n$ parámetros, y los algoritmos empleados en la práctica se caracterizan por $O(m \times n)$ tiempo de ejecución. Si limitamos el número de conexiones que cada salida puede tener en k , entonces el enfoque con conectividad dispersa requiere solo $k \times n$ parámetros y $O(k \times n)$ tiempo de ejecución. Para muchas aplicaciones prácticas, es posible obtener un buen rendimiento en la tarea de aprendizaje automático, manteniendo k varios órdenes de magnitud más pequeño que m . En una red convolucional profunda, las unidades en las capas más profundas pueden interactuar indirectamente con una porción más grande de la entrada. Esto permite que la red describa de manera eficiente interacciones de alto nivel de abstracción entre muchas variables mediante la construcción de interacciones más simples a partir de interacciones dispersas.

Cuando hablamos del intercambio de parámetros nos referimos al uso del mismo parámetro para más de una función en un modelo. En una red neuronal tradicional, cada elemento de la matriz de pesos se usa exclusivamente una vez cuando se calcula la salida de una capa. Se multiplica por un elemento de la entrada y luego nunca se vuelve a utilizar. Como sinónimo de compartir parámetros, se puede decir que una red tiene **pesos vinculados**, porque el valor del peso aplicado a una entrada está vinculado al valor de un peso aplicado en otro lugar. En una red neuronal convolucional, cada miembro del *kernel* se usa en todas las posiciones de la entrada sobre las que se va deslizando dicho *kernel* (excepto, quizás, algunos de los píxeles del borde, dependiendo de las decisiones de diseño con respecto al borde). El uso compartido de parámetros utilizado por la operación de convolución significa que, en lugar de aprender un conjunto separado de parámetros para cada ubicación, aprendemos un único conjunto.

Esto no afecta el tiempo de ejecución de la propagación hacia adelante, que sigue siendo $O(k \times n)$, pero sí reduce aún más los requisitos de almacenamiento del modelo a k parámetros. Recuerde que k es generalmente varios órdenes de magnitud más pequeño que m . Como m y n suelen tener aproximadamente el mismo tamaño, k es prácticamente insignificante en comparación con $m \times n$. Por lo tanto, la convolución es dramáticamente más eficiente que la multiplicación de la matriz densa de un perceptrón multicapa en términos de requisitos de memoria y eficiencia estadística.

En el caso de la convolución, la forma particular de compartir parámetros hace que la capa tenga una propiedad llamada **equivarianza a la translación**. Decir que una función es equivariante significa que, si la entrada cambia, la salida cambia de la misma manera. Más concretamente, una función $f(x)$ es equivalente a una función g si $f(g(x)) = g(f(x))$. En el caso de convolución, si dejamos que g sea cualquier función que traslade la entrada, es decir, la desplace, entonces la función de convolución es equivalente a g . Por ejemplo, definamos I como una función que proporciona brillo a la imagen de entrada. Supongamos que g es una función que asigna una función de imagen a otra función de imagen, de modo que $I' = g(I)$ es la función de imagen e $I'(x, y) = I(x - 1, y)$. Esta función desplaza cada píxel de I una unidad hacia la derecha. Si aplicamos esta transformación a I , y aplicamos la convolución, el resultado será el mismo que si aplicamos la convolución a I' , y luego aplicamos la transformación g a la salida. Al procesar datos de series temporales, esto significa que la convolución produce una especie de línea de tiempo que muestra cuándo diferentes características aparecen en la entrada. Si desplazamos un evento temporal en una secuencia de entrada, la representación exacta de dicho evento aparecerá más tarde en la salida.

De manera similar sucede con las imágenes, la convolución crea un mapa 2D en el que ciertas características aparecen a la entrada. Si movemos el objeto a la entrada, su representación se moverá de la misma forma a la salida. Esto es útil cuando sabemos que alguna función sobre un pequeño número de píxeles vecinos se aplica sobre múltiples ubicaciones de entrada. Por ejemplo, al procesar imágenes, es útil para **detectar bordes en la primera capa** de una red convolucional. Los mismos bordes aparecen más o menos en todas partes en la imagen, por lo que es práctico compartir parámetros en toda la imagen. En algunos casos, es posible que no deseemos compartir parámetros en toda la imagen. Por ejemplo, si estamos procesando imágenes recortadas para centrarlas en la cara de un individuo, es probable que queramos extraer diferentes características en diferentes ubicaciones: la parte de la red que procesa la parte superior de la cara debe buscar cejas, mientras que la parte de la red que procesa la parte inferior de la cara necesita buscar una barbilla.

La convolución no es naturalmente equivalente a otras transformaciones, como los cambios de escala o la rotación de una imagen. Se necesita de otros mecanismos para manejar este tipo de transformaciones.

4.3. Pooling

El bloque típico de una red convolucional consta de tres etapas (véase la Figura 18). La primera capa de las que se compone el bloque realiza varias convoluciones en paralelo (mediante un determinado número de filtros o *kernels*) para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación de salida de la operación convolución es la entrada a una función de activación no lineal, como la función de activación ReLU. En la tercera etapa, empleamos una capa de *pooling* para modificar aún más la salida del bloque convolucional.

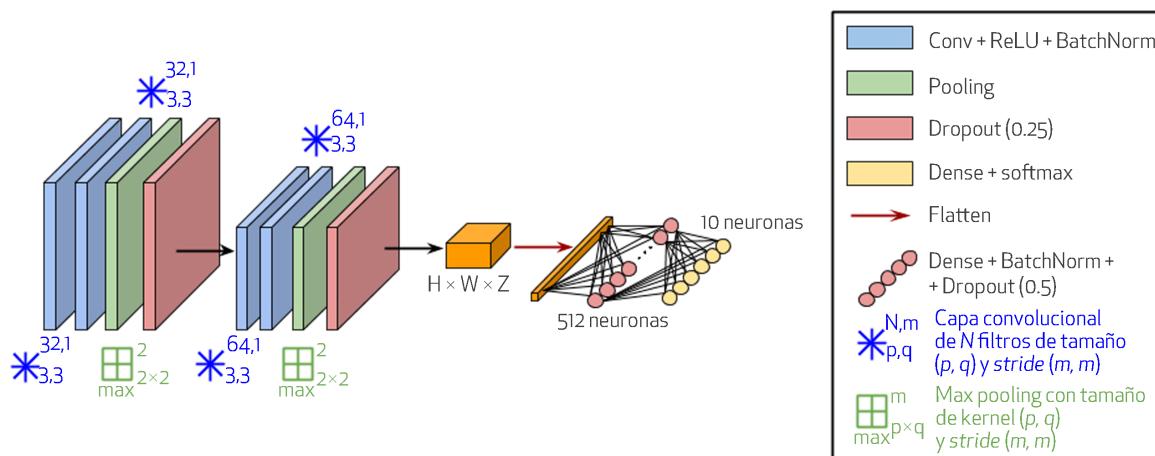


Figura 18. Arquitectura de red neuronal convolucional típica. En este caso se trata de una CNN compuesta por dos bloques convolucionales.



Una función de *pooling* disminuye las dimensiones espaciales del mapa de activación que sale de la capa convolucional con la pertinente función de activación. Por ejemplo, la operación *max pooling* (Zhou y Chellappa, 1988) computa la activación máxima dentro de un vecindario rectangular del mapa de activación (véase la Figura 19). Otras funciones de *pooling* populares calculan el promedio de un vecindario rectangular (*average pooling*), la norma L2 de un vecindario rectangular o un promedio ponderado basado en la distancia al píxel central.

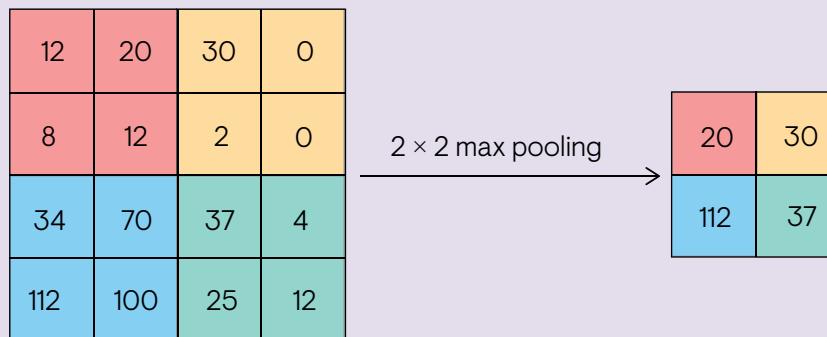


Figura 19. Representación gráfica de la operación max pooling (ejemplo con kernel 2×2). Por Firelord Phoenix bajo licencia CC BY-SA. Adaptado de <https://computersciencewiki.org/index.php/File:MaxpoolSample2.png>

En todos los casos, el *pooling* ayuda a hacer que la representación sea aproximadamente **invariable** cuando se producen pequeñas translaciones a la entrada. La invarianza a la traslación significa que, si desplazamos la entrada una pequeña cantidad, los valores de la mayoría de las salidas de la capa *pooling* no cambian. La invarianza en translaciones locales es una propiedad muy útil cuando nos interesa saber si hay presencia de características relevantes más que saber exactamente dónde están. Por ejemplo, al determinar si una imagen contiene una cara, no necesitamos saber la ubicación de los ojos con una precisión de píxeles perfectos, solo necesitamos saber que hay un ojo en el lado izquierdo de la cara y un ojo en el lado derecho de la cara. En otros contextos, es más importante preservar la ubicación de una característica. Por ejemplo, si queremos encontrar una esquina definida por dos bordes que se encuentran en una orientación específica, necesitamos preservar la ubicación de los bordes lo suficientemente bien como para probar si se encuentran. Realizar *pooling* sobre regiones espaciales produce invarianza a la traslación, pero si hacemos *pooling* sobre las salidas de las convoluciones parametrizadas por separado, las características pueden aprender a qué transformaciones se vuelven invariantes.

Debido a que el *pooling* sintetiza todas las activaciones en todo un vecindario, es posible utilizar menos unidades de *pooling* que neuronas detectoras, y obtener estadísticas del vecindario (max, media, L2, etc.) separadas por k píxeles, en lugar de emplear un *stride* de 1 píxel. La Figura 19 ejemplifica este hecho. Con esto se mejora la eficiencia computacional de la red, porque la siguiente capa tiene aproximadamente k veces menos entradas para procesar. Cuando el número de parámetros en la siguiente capa es una función del tamaño de su entrada (como cuando la siguiente capa está completamente conectada), esta reducción en el tamaño de entrada también puede mejorar la eficiencia estadística y reducir los requisitos de memoria para almacenar los parámetros.

Para muchas tareas, el *pooling* es esencial para manejar entradas de diferentes tamaños. Por ejemplo, si queremos clasificar imágenes de tamaño variable, la entrada a la capa de clasificación debe tener un tamaño fijo. Esto generalmente se logra variando el tamaño de un desplazamiento entre las regiones de *pooling* para que la capa de clasificación siempre reciba el mismo número de activaciones, independientemente del tamaño de entrada. Por ejemplo, la capa *pooling* final de la red puede definirse para generar cuatro conjuntos de estadísticas resumen, uno para cada cuadrante de una imagen, independientemente del tamaño de la imagen.

Algunos trabajos teóricos nos orientan sobre qué tipos de *pooling* se deben utilizar en diversas situaciones (Boureau, Ponce, Fr y Lecun, 2010). También es posible agrupar características dinámicamente, por ejemplo, ejecutando un algoritmo de *pooling* en las ubicaciones de características interesantes (Boureau, Le Roux, Bach, Ponce y Lecun, 2011). Este enfoque produce un conjunto diferente de regiones de agrupación para cada imagen. Otro enfoque es aprender una estructura de agrupación única que luego se aplica a todas las imágenes (Jia, Huang y Darrell, 2012).

Cabe informar de que la operación de *pooling* puede complicar algunos tipos de arquitecturas de redes neuronales que utilizan información de arriba hacia abajo, como las máquinas Boltzmann y *autoencoders*.



Enlace de interés

Se trata de un repositorio con explicaciones teóricas y ejemplos prácticos de la mayoría de los conceptos que abarca la asignatura. El módulo 2 de este enlace cubre con detalle el apartado de redes neuronales convolucionales.

<https://cs231n.github.io>

4.4. Tipos de datos

Como ya se ha comentado anteriormente, los datos que alimentan una red convolucional generalmente consisten en varios canales. Cada canal es la observación de una cantidad diferente en un instante o intervalo temporal. A continuación, se enumeran una serie de ejemplos de datos con diferentes dimensionalidades y número de canales.

- Ejemplos de entradas con un único canal
 - **1D: señal de audio.** El eje sobre el que se aplica la convolución corresponde al tiempo. Muestreamos el tiempo y medimos la amplitud de la forma de onda según un periodo temporal dado por una determinada frecuencia de muestreo.
 - **2D: datos de audio bidimensionales.** Son datos que se han procesado previamente con una transformada de Fourier. Podemos transformar la forma de onda de audio en un tensor bidimensional con diferentes filas correspondientes a diferentes frecuencias y con diferentes columnas correspondientes a diferentes puntos temporales. Usar la convolución en el tiempo hace que el modelo sea equivariante a variaciones temporales. Aplicar la convolución sobre el eje de frecuencia hace que el modelo sea equivariante frecuencialmente, de modo que la misma melodía reproducida en una octava diferente produce la misma representación, pero a una altura diferente en la salida de la red.
 - **3D: datos volumétricos tridimensionales.** Una fuente común de este tipo de datos es la tomografía computarizada, empleada en la adquisición de imágenes médicas.
- Ejemplos de entrada multicanal
 - **1D: datos de animación de esqueleto.** Las animaciones de personajes renderizados en ordenador en 3D se generan al alterar la pose de un “esqueleto” con el tiempo. En cada momento, la pose del personaje se describe mediante una descripción de los ángulos de cada una de las articulaciones en el esqueleto del personaje. Cada canal en los datos que alimentamos al modelo convolucional representa el ángulo sobre un eje de una articulación.
 - **2D: datos de imagen en color.** Un canal contiene un valor de intensidad de píxel rojo, otro verde y otro azul. El *kernel* de convolución se mueve sobre los ejes horizontal y vertical de la imagen, lo que confiere equivarianza a la translación en ambas direcciones.
 - **3D: datos de vídeo en color.** Un eje corresponde al tiempo; otro, a la altura del cuadro de vídeo, y otro, al ancho del cuadro de vídeo. Para un ejemplo de redes convolucionales aplicadas al vídeo, consulte Chen, Ting, Marlin y De Freitas (2010).

Hasta ahora hemos discutido solo el caso en el que cada ejemplo en entrenamiento y prueba tienen las mismas dimensiones espaciales. Una ventaja de las redes convolucionales es que también pueden procesar entradas con diferentes extensiones espaciales. Estos tipos de entrada no pueden procesarse mediante perceptrones multicapa. Esto proporciona una razón convincente para usar redes convolucionales, incluso cuando el coste computacional y el sobreajuste no sean un problema.

Por ejemplo, considere una colección de imágenes en las que cada imagen tiene un ancho y una altura diferentes. No está claro cómo modelar tales entradas con una matriz de pesos de tamaño fijo.

La convolución es fácil de aplicar; el *kernel* simplemente se aplica un número diferente de veces dependiendo del tamaño de la entrada, mientras que la salida de la operación convolución se escala en consecuencia. La convolución puede verse como una multiplicación matricial; el mismo núcleo de convolución incide sobre un tamaño de vecindario (**campo receptivo** o **receptive field** en inglés) diferente según cada tamaño de entrada. A veces, la salida de la red, así como la entrada, pueden tener un tamaño variable, por ejemplo, si queremos asignar una etiqueta de clase a cada píxel de la entrada. En este caso, no es necesario un trabajo adicional de diseño de la red. En otros casos, la red debe producir una salida de tamaño fijo, por ejemplo, si queremos asignar una sola etiqueta de clase a toda la imagen. En este caso, debemos realizar algunos pasos de diseño adicionales, como insertar una capa de *pooling* cuyas regiones de agrupación escalen en tamaño proporcional al tamaño de la entrada, para mantener un número fijo de salidas agrupadas.

Tenga en cuenta que el uso de convolución para procesar entradas de tamaño variable solo tiene sentido para las entradas que tienen un tamaño variable, porque contienen cantidades variables de observación del mismo tipo: diferentes longitudes de adquisición en el tiempo, diferentes anchos de observaciones en el espacio, etc. La convolución no tiene sentido si la entrada tiene un tamaño variable por incluir diferentes tipos de observaciones. Por ejemplo, si estamos procesando solicitudes de acceso a la universidad, y nuestras características consisten en calificaciones y puntajes de exámenes estandarizados, pero no todos los solicitantes hicieron el examen estandarizado.

4.5. Redes convolucionales y la historia del aprendizaje profundo

Las redes convolucionales han jugado un papel importante en la historia del aprendizaje profundo. Las redes neuronales convolucionales fueron algunos de los primeros modelos profundos en funcionar bien, mucho antes de que los modelos profundos arbitrarios se consideraran viables. Las redes convolucionales también fueron de las primeras redes neuronales que resolvieron aplicaciones comerciales importantes y permanecen a la vanguardia de las aplicaciones comerciales de aprendizaje profundo en la actualidad. Por ejemplo, en la década de 1990, el grupo de investigación de redes neuronales de AT&T desarrolló una red convolucional para leer cheques bancarios (Lecun, Bottou, Bengio y Haffner, 1998). A finales de la década de 1990, este sistema implementado por NCR leía más del diez por ciento de todos los cheques en los Estados Unidos. Más tarde, varios sistemas de reconocimiento de escritura y OCR basados en redes convolucionales fueron implementados por Microsoft. Consulte (LeCun, Kavukcuoglu y Farabet, 2010) para un análisis más profundo de la historia de las redes convolucionales hasta 2010.

Las redes convolucionales también se utilizaron para ganar muchos concursos. El interés comercial en el aprendizaje profundo comenzó cuando Krizhevsky, Sutskever y Hinton (2012) ganaron el desafío de reconocimiento de objetos **ImageNet**, pero es cierto que las redes convolucionales ya se habían utilizado para ganar otros concursos de aprendizaje automático y visión artificial con menos impacto años antes.

Las redes convolucionales fueron de las primeras redes profundas en ser entrenadas mediante retropropagación. No está del todo claro por qué las redes convolucionales tuvieron éxito cuando se consideró que las redes neuronales generales fallaron en sus entrenamientos con retropropagación. Puede ser simplemente que las redes convolucionales fueran más eficientes computacionalmente que las redes totalmente conectadas o perceptrones multicapa, por lo que fue más fácil realizar múltiples experimentos con ellas y ajustar su implementación e hiperparámetros.

Con el *hardware* actual, grandes redes totalmente conectadas parecen funcionar razonablemente en muchas tareas, incluso cuando se utilizan conjuntos de datos que estaban disponibles y funciones de activación que eran populares durante los tiempos en que se creía que estas redes no funcionaban bien. Puede ser que las principales barreras para el éxito de las redes neuronales fueran psicológicas (los profesionales no esperaban que las redes neuronales funcionaran, por lo que no hicieron un esfuerzo serio para usarlas). Sea cual fuere el caso, es una suerte que las redes convolucionales hayan funcionado bien desde hace décadas. En muchos sentidos, llevaron el peso del aprendizaje profundo y allanaron el camino para la aceptación de las redes neuronales en general.

Las redes convolucionales están especializadas para trabajar con datos que tienen una topología claramente estructurada en cuadrícula y para escalar dichos modelos a un tamaño muy profundo. Para procesar datos secuenciales unidimensionales, pasamos a otra poderosa especialización dentro del marco de redes neuronales: las redes neuronales recurrentes.



Capítulo 5

Modelado de secuencias: redes recurrentes y recursivas

Las **redes neuronales recurrentes**, o RNN (Rumelhart, Hinton y Williams, 1986), son una familia de redes neuronales para procesar datos secuenciales. Si recordamos el capítulo anterior, una red convolucional es una red neuronal especializada en procesar una cuadrícula de valores x , como, por ejemplo, una imagen. Una red neuronal recurrente es una red neuronal especializada en procesar una secuencia de valores $x(1), \dots, x(\tau)$.

Así como las redes convolucionales pueden escalar fácilmente a imágenes con gran ancho y alto, y algunas redes convolucionales pueden procesar imágenes de tamaño variable, las redes recurrentes pueden escalar a secuencias mucho más largas de lo que sería práctico para redes sin especialización en secuencias. La mayoría de las redes recurrentes también pueden procesar secuencias de longitud variable.

Para pasar de redes multicapa a redes recurrentes, debemos rescatar una de las primeras ideas del aprendizaje automático y los modelos estadísticos de la década de 1980: compartir parámetros en diferentes partes de un modelo. El uso compartido de parámetros permite extender y aplicar el modelo a ejemplos de diferentes formas (diferentes longitudes) y generalizar a través de ellos.

Si tuviéramos parámetros separados para cada valor temporal, no podríamos generalizar a longitudes de secuencia que no se vieran durante el entrenamiento, ni compartir la potencia estadística en diferentes longitudes de secuencia y en diferentes posiciones temporales.

Dicho intercambio es particularmente importante cuando una información específica puede ocurrir en múltiples posiciones dentro de la secuencia. Por ejemplo, considere las dos oraciones “Fui a Nepal en 2009” y “En 2009, fui a Nepal”. Si pedimos a un modelo de aprendizaje automático que lea cada oración y extraiga el año en que el narrador fue a Nepal, este debería reconocer el año 2009 como información relevante, ya sea cuando la palabra aparezca en sexta posición dentro de la oración o en segunda.

Supongamos que entrenamos una red prealimentada que procesa oraciones de longitud fija. Una red prealimentada tradicional totalmente conectada tendría parámetros separados para cada característica de entrada, por lo que necesitaría aprender todas las reglas del idioma por separado en cada posición de la oración. En comparación, una red neuronal recurrente comparte los mismos pesos en varios pasos de tiempo.

Una idea relacionada es el uso de la convolución a través de una secuencia temporal 1D. Este enfoque convolucional es la base de las redes neuronales de retardo temporal (Lang, Waibel y Hinton, 1990; Waibel, Hanazawa, Hinton, Shikano y Lang, 1989). La operación convolución permite que una red comparta parámetros a lo largo del tiempo, pero de manera superficial.

La salida de convolución es una secuencia en la que cada miembro de la salida es función de un pequeño número de miembros vecinos de la entrada. La idea de compartir parámetros se manifiesta en la aplicación del mismo *kernel* de convolución en cada paso de tiempo.

Las redes recurrentes comparten parámetros de una manera diferente. Cada miembro de la salida es una función de los miembros anteriores de la salida. Cada miembro de la salida se obtiene utilizando la misma regla de actualización aplicada a las salidas anteriores. Esta formulación recurrente resulta en el intercambio de parámetros a través de un grafo computacional muy profundo.

Por simplicidad, nos referimos a las redes neuronales recurrentes que operan en una secuencia que contiene vectores $x(t)$ con el índice temporal t , que varía de 1 a τ . En la práctica, las redes recurrentes generalmente operan en *minibatches* de dichas secuencias, con una longitud de secuencia diferente τ para cada miembro del *minibatch*. Hemos omitido los índices de *minibatch* para simplificar la notación. Además, el índice del paso del tiempo no necesita referirse literalmente al paso del tiempo en el mundo real. A veces se refiere solo a la posición en la secuencia.

Las redes neuronales recurrentes también se pueden aplicar en dos dimensiones a través de datos espaciales, como imágenes, e incluso cuando se aplican a datos que involucran tiempo, la red puede tener conexiones que retrocedan en el tiempo, siempre que se observe la secuencia completa antes de proporcionarla a la red.

Este capítulo se basa en la idea de un grafo computacional para incluir ciclos. Estos ciclos representan la influencia del valor presente de una variable en su propio valor en un paso de tiempo futuro. Tales grafos computacionales nos permiten definir redes neuronales recurrentes. Posteriormente, describimos diferentes formas de construir, entrenar y usar redes neuronales recurrentes. Para obtener más información sobre redes neuronales recurrentes de la disponible en este manual, remitimos al lector al libro de texto de Graves (2012).

5.1. Redes neuronales recurrentes

A continuación, presentamos algunos ejemplos de patrones de diseño importantes para redes neuronales recurrentes:

- Redes recurrentes que producen una salida en cada paso de tiempo y tienen conexiones recurrentes entre unidades ocultas, como se ilustra en la Figura 20:

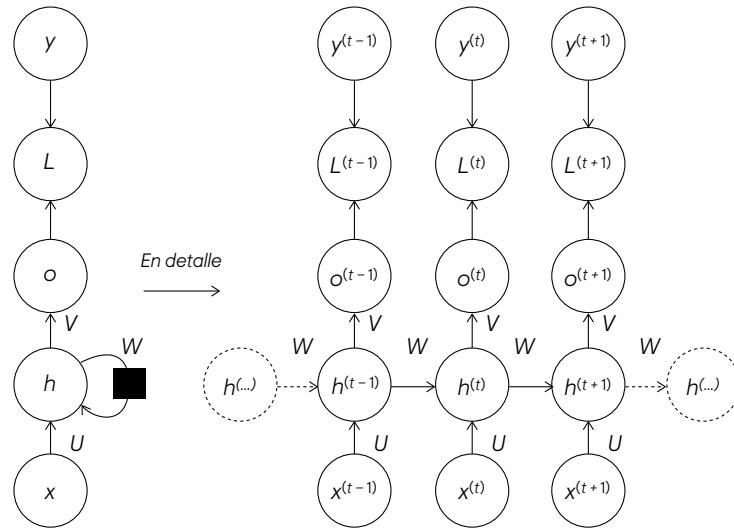


Figura 20. Red recurrente que produce una salida en cada paso de tiempo y tiene conexiones recurrentes entre unidades ocultas. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

- Redes recurrentes que producen una salida en cada paso de tiempo y tienen conexiones recurrentes solo desde la salida en un instante temporal a las unidades ocultas en el siguiente instante de tiempo, como se ilustra en la Figura 21:

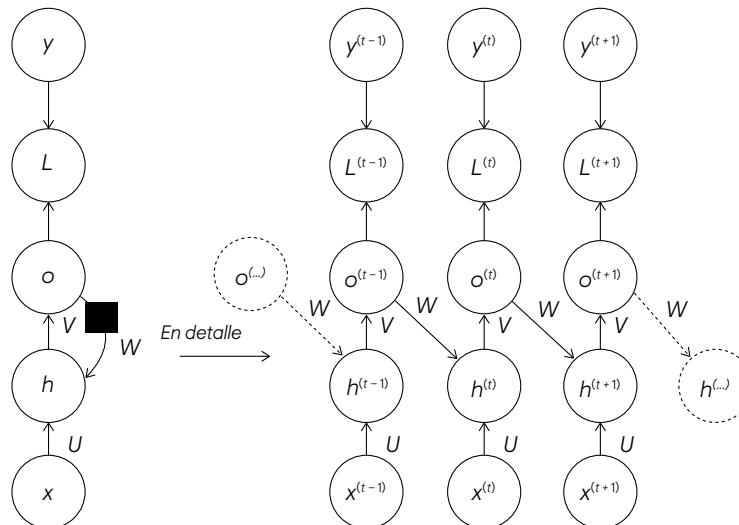


Figura 21. Red recurrente que produce una salida en cada paso de tiempo y tiene conexiones recurrentes solo desde la salida en un instante temporal a las unidades ocultas en el siguiente instante de tiempo. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, *Deep learning*, Massachusetts: MIT Press.

- Redes recurrentes con conexiones recurrentes entre unidades ocultas que leen una secuencia completa y luego producen una única salida, como se ilustra en la Figura 22:

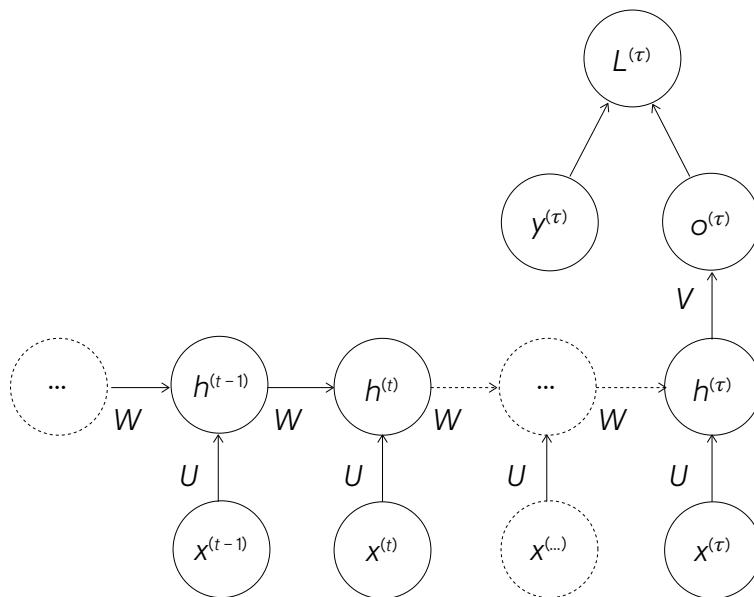


Figura 22. Red recurrente con conexiones recurrentes entre unidades ocultas, que leen una secuencia completa y producen una única salida. Adaptado de I. Goodfellow, Y. Bengio y A. Courville, 2016, Deep learning, Massachusetts: MIT Press.

La red neuronal recurrente de la Figura 20 es universal en el sentido de que cualquier función que pueda ser calculada por una máquina de Turing puede ser calculada por una red recurrente de tamaño finito. La salida se puede leer desde la red neuronal recurrente después de varios instantes temporales, algo que es asintóticamente lineal al número de instantes temporales utilizados por la máquina de Turing y asintóticamente lineal a la longitud de la entrada (Hyy, 1996; Siegelmann, 1995; Siegelmann y Sontag, 1991 y 1995). Las funciones computables por una máquina de Turing son discretas, por lo que estos resultados se refieren a la implementación exacta de la función, no a aproximaciones. La red neuronal recurrente, cuando se usa como una máquina de Turing, toma una secuencia binaria como entrada, y sus salidas deben ser discretizadas para proporcionar una salida binaria. Es posible calcular todas las funciones en esta configuración utilizando una única red neuronal recurrente de tamaño finito (Siegelmann y Sontag, 1995). La entrada de la máquina de Turing es una especificación de la función a calcular, por lo que la misma red que simula esta máquina de Turing es suficiente para todos los problemas. La red neuronal recurrente teórica utilizada en la demostración puede simular una pila ilimitada al representar sus activaciones y pesos con números racionales de precisión ilimitada.

A continuación, se desarrollan las ecuaciones de propagación hacia delante para la red neuronal recurrente representada en la Figura 20. La figura no especifica la elección de la función de activación para las unidades ocultas. Aquí asumimos la función de activación tangente hiperbólica. Además, la figura no especifica exactamente qué valores toman la salida y la función de pérdidas. Supondremos que la salida es discreta, como si la red neuronal recurrente se usara para predecir palabras o caracteres. Una forma natural de representar variables discretas es considerar la salida o como una probabilidad logarítmica no normalizada de cada valor posible de la variable discreta.

Luego podemos aplicar la operación softmax como un paso posterior al procesamiento para obtener un vector \hat{y} de probabilidades normalizadas sobre la salida. La propagación hacia delante comienza con una especificación del estado inicial $h(0)$. Luego, para cada paso de tiempo de $t = 1$ a $t = \tau$, aplicamos las siguientes ecuaciones de actualización:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

$$h^{(t)} = \tanh(a^{(t)})$$

$$o^{(t)} = c + Vh^{(t)}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

En las ecuaciones anteriores, los parámetros son los vectores de sesgo b y c , junto con las matrices de ponderación U , V y W , respectivamente, para conexiones de entrada a neurona oculta, de neurona oculta a salida y de neurona oculta a neurona oculta. Este es un ejemplo de una red recurrente que asigna una secuencia de entrada a una secuencia de salida de la misma longitud. La pérdida total para una secuencia dada de valores de x emparejada con una secuencia de valores y es la suma de las pérdidas en todos los instantes de tiempo. Por ejemplo, si $L^{(t)}$ es la verosimilitud logarítmica negativa de $y(t)$, dado $x(1), \dots, x(t)$, entonces tenemos la siguiente ecuación:

$$L(\{x^{(1)}, \dots, x^{\tau}\}, \{y^{(1)}, \dots, y^{\tau}\}) = \sum_t L^{(t)} = -\sum_t \log p_{\text{model}}(y^{(t)} | \{x^{(1)}, \dots, x^{(t)}\})$$

En la ecuación anterior, $p_{\text{model}}(y^{(t)} | \{x^{(1)}, \dots, x^{(t)}\})$ se obtiene leyendo la entrada para $y^{(t)}$ del vector de salida $\hat{y}^{(t)}$. Calcular el gradiente de esta función de pérdida respecto a los parámetros es una operación costosa. El cálculo del gradiente implica realizar un paso de propagación hacia adelante moviéndose de izquierda a derecha a través de nuestra ilustración del gráfico desenrollado de la Figura 20, seguido de un paso de propagación hacia atrás que se mueve de derecha a izquierda a través del gráfico. El tiempo de ejecución es $O(\tau)$ y no se puede reducir mediante la parallelización, porque el gráfico de propagación directa es inherentemente secuencial; un instante temporal solo puede calcularse después de obtener el anterior. Los estados calculados en el paso hacia adelante deben almacenarse hasta que se reutilicen durante el paso hacia atrás, por lo que el coste de la memoria también es $O(\tau)$. El algoritmo de retropropagación aplicado al gráfico desenrollado con coste $O(\tau)$ se denomina **retropropagación a través del tiempo** (BPTT). La red con recurrencia entre unidades ocultas es, por lo tanto, muy poderosa, pero también costosa de entrenar.

5.2. Memoria a largo plazo y otras redes neuronales recurrentes de compuerta



En el momento de escribir este manual, los modelos de secuencia más efectivos utilizados en aplicaciones prácticas se denominan **redes neuronales recurrentes de compuerta** (*gated RNN* en inglés). Estos incluyen memoria a largo plazo y redes basadas en **unidades recurrentes de compuerta** (*gated recurrent units* en inglés).

>>>

Las redes neuronales recurrentes de compuerta se basan en la idea de crear caminos a través del tiempo que tienen derivadas que no incurren ni en problemas de **explosión** ni de **desvanecimiento del gradiente** (*vanishing* y *exploding gradients* en inglés).

Las redes neuronales recurrentes de compuerta generalizan pesos de interconexión que pueden cambiar en cada instante de tiempo.

Las redes neuronales recurrentes de compuerta permiten que la red acumule información (como evidencia de una característica o categoría en particular) durante un período prolongado. Sin embargo, una vez que se ha utilizado esa información, puede ser útil que la red neuronal olvide el estado anterior.

Por ejemplo, si una secuencia está hecha de subsecuencias y queremos que una unidad acumule evidencias dentro de cada subsecuencia, necesitamos un mecanismo para olvidar el estado anterior que la ponga a cero.

En lugar de decidir manualmente cuándo borrar el estado, queremos que la red neuronal aprenda a decidir cuándo hacerlo.

5.2.1. Unidades de memoria a largo plazo (LSTM)

La idea inteligente de introducir bucles automáticos para producir caminos donde el gradiente pueda fluir durante largas duraciones temporales es la contribución central del modelo inicial de memoria a largo plazo (*long-short term memory* o LSTM) (Hochreiter y Schmidhuber, 1997).

Una contribución adicional crucial ha sido hacer que el peso de este autociclo esté condicionado al contexto, en lugar de ser fijo (Gers, Schmidhuber y Cummins, 2000).

Haciendo que el peso de este circuito cerrado este controlado por una compuerta (otra unidad oculta), la escala de tiempo de integración se puede cambiar dinámicamente.

En este caso, queremos decir que, incluso para una memoria a largo plazo con parámetros fijos, la escala de tiempo de integración puede cambiar en función de la secuencia de entrada, porque las constantes de tiempo son emitidas por el propio modelo.

Se ha encontrado que la memoria a largo plazo es extremadamente exitosa en muchas aplicaciones, como el reconocimiento de escritura a mano sin restricciones (Graves y Schmidhuber, 2009), el reconocimiento de voz (Graves y Jaitly, 2014; Graves, Mohamed y Hinton, 2013), la generación de escritura a mano (Graves, 2013), la traducción automática (Sutskever et al., 2014), los subtítulos de imágenes (Kiros, Salakhutdinov y Zemel, 2014; Vinyals, Toshev, Bengio y Erhan, 2014; Xu et al., 2015) y el analizador sintáctico (*parsing* en inglés) (Vinyals et al., 2014).

El diagrama de un bloque de memoria a largo plazo se ilustra en la Figura 23. Las ecuaciones de propagación hacia delante correspondientes se formulan a continuación para arquitectura de red recurrente poco profunda. Algunas arquitecturas de red más profundas también se han utilizado con éxito (Graves, Mohamed y Hinton, 2013; Pascanu, Gulcehre, Cho y Bengio, 2014). En lugar de una unidad que simplemente aplica una no linealidad en el elemento de la transformación afín de entradas y unidades recurrentes, las redes recurrentes de memoria a largo plazo disponen de celdas de memoria a largo plazo que tienen una recurrencia interna (un bucle automático), además de la recurrencia externa de la red neuronal recurrente.

Cada celda tiene las mismas entradas y salidas que una red recurrente ordinaria, pero también tiene más parámetros y un sistema de unidades de compuerta que controla el flujo de información. La componente más importante es la unidad de estado $s_i^{(t)}$, que tiene un bucle automático lineal cuyo peso (o la constante de tiempo asociada) es controlado por una unidad de puerta de olvido $f_i^{(t)}$ (para el instante de tiempo t y la celda i), que establece este peso en un valor entre 0 y 1 a través de una unidad sigmoide como la que mostramos a continuación:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{ij}^f x_j^{(t)} + \sum_j W_{ij}^f h_j^{(t-1)}\right)$$

$x(t)$ es el vector de entrada actual y $h(t)$ es el vector de la capa oculta actual, que contiene las salidas de todas las celdas de memoria a largo plazo, y b^f , U^f y W^f son los sesgos, los pesos de entrada y los pesos recurrentes, respectivamente, de las puertas de olvido. El estado interno de la celda de memoria a largo plazo se actualiza de la siguiente manera:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{ij} x_j^{(t)} + \sum_j W_{ij} h_j^{(t-1)}\right)$$

b , U y W indican, respectivamente, los sesgos, los pesos de entrada y los pesos recurrentes en la celda de memoria a largo plazo. La **unidad de puerta de entrada externa** $g(t)$ se calcula de manera similar a la puerta de olvido (con una unidad sigmoide para obtener un valor de activación entre 0 y 1), pero con sus propios parámetros:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{ij}^g x_j^{(t)} + \sum_j W_{ij}^g h_j^{(t-1)}\right)$$

La salida $h_i^{(t)}$ de la celda de memoria a largo plazo también se puede desconectar a través de la puerta de salida $q_i^{(t)}$, que también usa una unidad sigmoide para el *gating*:

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{ij}^o x_j^{(t)} + \sum_j W_{ij}^o h_j^{(t-1)}\right)$$

Los parámetros b^o , U^o , W^o son los sesgos, los pesos de entrada y los pesos recurrentes, respectivamente. Entre las variantes, uno puede elegir usar el estado de la celda $s(t)$ como una entrada adicional (con su peso) en las tres puertas de la unidad i -ésima, como se muestra en la Figura 23. Esto requeriría tres parámetros adicionales.

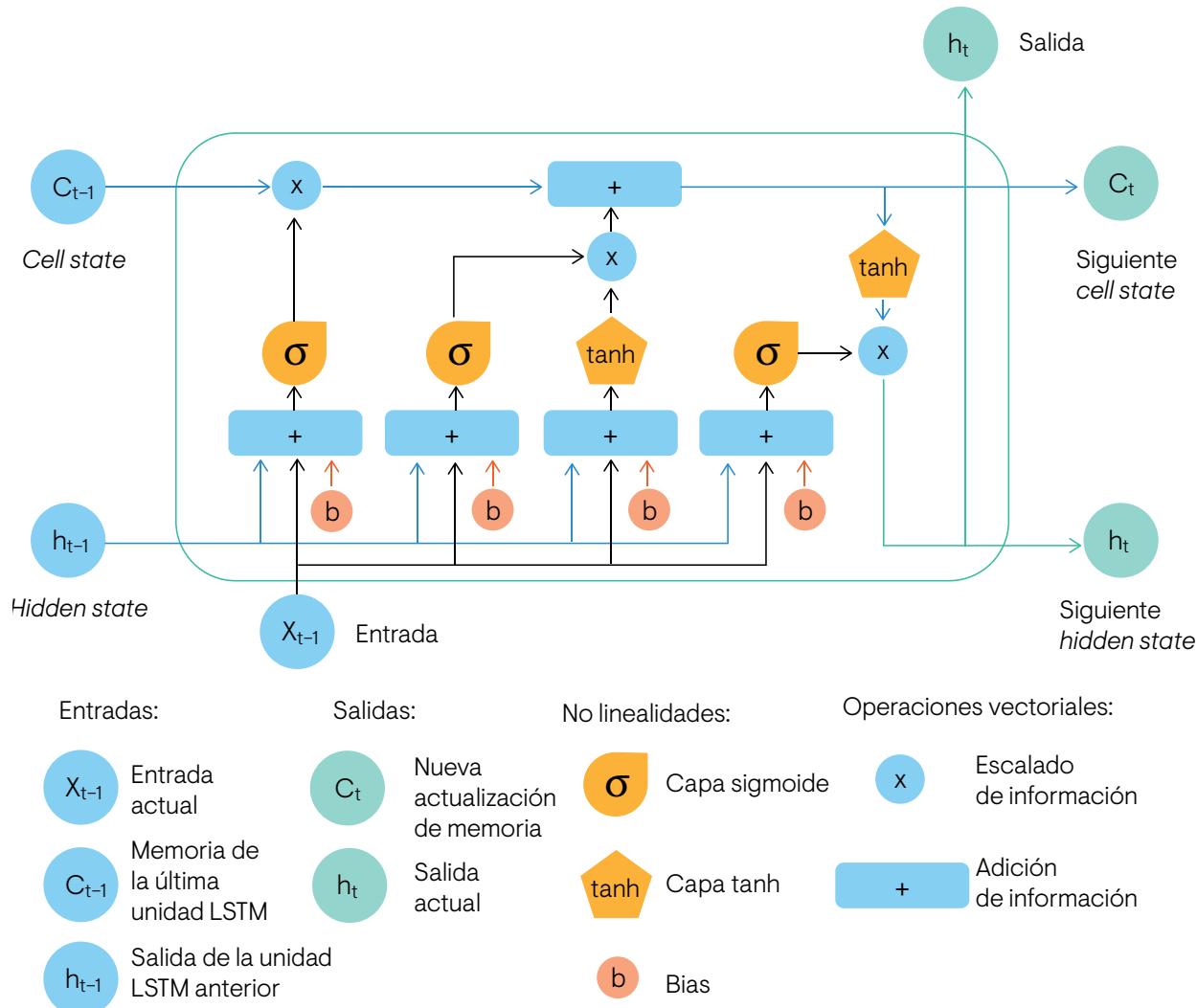


Figura 23. Representación gráfica de una unidad LSTM. Adaptado de “Application of Long Short-Term Memory (LSTM) Neural Network for Flood Forecasting”, por X.-H. Le, H. V. Ho, G. Lee y S. Jung, 2019, Water, 11(7). Recuperado de <https://www.mdpi.com/2073-4441/11/7/1387/pdf>

Se ha demostrado que las redes de memoria a largo plazo aprenden dependencias a largo plazo más fácilmente que las arquitecturas recurrentes simples, primero en conjuntos de datos artificiales diseñados para probar la capacidad de aprender dependencias a largo plazo (Bengio, Simard y Frasconi, 1994; Gers et al., 2000; Hochreiter, Bengio, Frasconi y Schmidhuber, 2001), y luego en tareas más desafiantes de procesamiento de secuencias donde se obtuvo un rendimiento muy prometedor (Graves, 2012; Graves, Mohamed y Hinton, 2013; Sutskever et al., 2014).

Las variantes y alternativas a las unidades de memoria a largo plazo que se han estudiado y utilizado se analizan a continuación.



Enlace de interés

Este enlace de interés detalla minuciosamente el funcionamiento de una unidad LSTM y la labor que ejerce cada una de las compuertas que la componen.

<http://colah.github.io/posts/2015-08-Understanding-LSTMs>

5.2.2. Otras redes neuronales recurrentes de compuerta

¿Qué piezas de la arquitectura de memoria a largo plazo son realmente necesarias? ¿Qué otras arquitecturas exitosas podrían diseñarse que permitan a la red controlar dinámicamente la escala temporal y olvidar el comportamiento de diferentes unidades?

Algunas respuestas a estas preguntas se obtienen con el trabajo reciente sobre redes neuronales recurrentes de compuerta, cuyas unidades también se conocen como *gated recurrent units*, o GRU (Cho, Merriënboer, Bahdanau y Bengio, 2014; Chrupała, Alishahi y NI, 2015; Chung, Gulcehre, Cho y Bengio, 2014 y 2015; Jozefowicz y Zaremba, 2015). La principal diferencia con la unidad de memoria a largo plazo es que una sola unidad de *gating* controla simultáneamente el factor de olvido y la decisión de actualizar la unidad de estado. Las ecuaciones de actualización son las siguientes:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{ij} x_j^{(t-1)} + \sum_j W_{ij} r_j^{(t-1)} h_j^{(t-1)} \right)$$

En la ecuación anterior, u representa la puerta de actualización y r , la puerta de restablecimiento. Sus valores se definen como se muestra a continuación:

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{ij}^u x_j^{(t)} + \sum_j W_{ij}^u h_j^{(t)} \right)$$

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{ij}^r x_j^{(t)} + \sum_j W_{ij}^r h_j^{(t)} \right)$$

Las puertas de restablecimiento y actualización pueden ignorar de manera individual partes del vector de estado. Las puertas de actualización pueden regular linealmente cualquier dimensión, eligiendo copiarlo (en un extremo de la sigmoide) o ignorarlo completamente (en el otro extremo) reemplazándolo con el nuevo valor de estado objetivo. Las puertas de reinicio controlan qué partes del estado se utilizan para calcular el siguiente estado objetivo, introduciendo un efecto no lineal adicional en la relación entre el estado pasado y el estado futuro.

Se pueden diseñar muchas más variantes en torno a este tema. Por ejemplo, la salida de la puerta de reinicio (o puerta de olvido) podría compartirse entre múltiples unidades ocultas. Alternativamente, el producto de una puerta global (que cubre un grupo completo de unidades, como una capa completa) y una puerta local (por unidad) podría usarse para combinar el control global y el control local. Sin embargo, varias investigaciones sobre variaciones de arquitectura de memoria a largo plazo y *gated recurrent units* no encontraron ninguna variante que superara claramente a ambas en una amplia gama de tareas (Greff, Srivastava, Koutník, Steunebrink y Schmidhuber, 2015; Jozefowicz y Zaremba, 2015). Greff et al. (2015) encontraron que un ingrediente crucial es la puerta del olvido, mientras que Jozefowicz y Zaremba (2015) descubrieron que agregar un sesgo de 1 a la puerta de olvido de memoria a largo plazo, una práctica defendida por Gers et al. (2000), hace que la memoria a largo plazo sea igual de potente que la mejor de las arquitecturas exploradas.



Capítulo 6

Aprendizaje por refuerzo

Como ya hemos visto a lo largo del presente manual, el aprendizaje basado en redes neuronales, o *deep learning*, ha provocado muchos avances en otras ramas de la inteligencia artificial o del aprendizaje automático, como ocurre en el caso del aprendizaje por refuerzo.

En el aprendizaje por refuerzo, continuamos hablando de algoritmos avanzados de aprendizaje y de un aprendizaje guiado por los datos disponibles. Antes de entrar en detalle, vamos a situar el aprendizaje por refuerzo en el ámbito de la inteligencia artificial, ya que uno de los puntos más interesantes de esta familia de algoritmos es la forma en la que desarrollan su aprendizaje.



Mientras que en otras ramas de la inteligencia artificial los grandes volúmenes de datos sirven como un repositorio de experiencia, en el aprendizaje por refuerzo, el dato se genera en tiempo real, a partir de las simulaciones y los entornos donde estos algoritmos se desarrollan.

En relación con el aprendizaje basado en datos, normalmente encontramos dos enfoques principales: métodos supervisados y métodos no supervisados. Brevemente, cuando hablamos de métodos supervisados nos referimos a aquellos algoritmos que relacionan los datos disponibles con una etiqueta o valor objetivo.

Esta etiqueta o valor objetivo puede estar contenido en los datos (como una variable más) o se puede definir a partir de reglas de negocio que se tienen que aplicar sobre el mismo conjunto de datos. Por otro lado, los métodos no supervisados forman la familia de algoritmos que se aplica sobre conjuntos de datos en los que no se conoce un objetivo *a priori*. Un ejemplo de este segundo tipo son los algoritmos de clusterización.

Tomando como punto de partida la comparativa entre el aprendizaje por refuerzo y los aprendizajes supervisados y no supervisados que encontramos en Sutton y Barto (1992), podemos analizar en qué nivel se sitúa el aprendizaje por refuerzo. Para muchos expertos, al ser una familia de algoritmos que aprende a partir de la experiencia y datos pasados, el aprendizaje por refuerzo se sitúa al mismo nivel que el aprendizaje supervisado y el aprendizaje no supervisado. Por otro lado, y debido a la relación del aprendizaje por refuerzo con otras ramas de la inteligencia artificial, como visión por computador, robótica o sistemas expertos, existe otra gran parte de expertos que define el aprendizaje por refuerzo como una rama en sí.

Más allá de entender en qué área podemos englobar al aprendizaje por refuerzo, es importante entender qué puede aportar teniendo en cuenta el aprendizaje basado en datos. A partir del repaso analizado en Lepenioti, Bousdekis, Apostolou y Mentzas (2020), podemos visualizar la relación que hay entre el tipo de aprendizaje y el resultado esperado:

- a. Aprendizaje no supervisado → análisis descriptivo
- b. Aprendizaje supervisado → análisis predictivo
- c. Aprendizaje por refuerzo → análisis prescriptivo

Cuando hablamos de **aprendizaje no supervisado**, nos referimos al tipo de extracción de conocimiento sin un objetivo definido *a priori*. Trabajamos directamente sobre el conjunto de variables disponibles y el objetivo es encontrar correlaciones y patrones a partir de la similitud de los elementos que componen el conjunto de datos.



El resultado de este aprendizaje es un análisis descriptivo, ya que el conocimiento que se extrae es a pasado, es decir, el conocimiento necesita ser interpretado para verdaderamente descubrir nuevas relaciones o nuevos comportamientos que antes no se conocían.

Respecto al **aprendizaje supervisado**, en este caso, sí que disponemos de un objetivo a alcanzar mediante el algoritmo de aprendizaje. A este objetivo también se lo conoce como *variable objetivo*, ya que es una variable que podemos encontrar en nuestro conjunto de datos o que podemos definir a partir de reglas de negocio conocidas.



Cuando se relaciona un conjunto de variables de nuestros datos con otra variable como objetivo para conocer sus valores futuros, el aprendizaje supervisado se basa en un análisis predictivo.

La última rama que encontramos es el **aprendizaje por refuerzo**. En los dos casos anteriores, los conjuntos de datos son datos recopilados a lo largo del tiempo, normalmente por grupos de personas o por procesos automáticos. Es por ello que estos conjuntos de datos pueden pecar de estar muy sesgados hacia el objetivo que se quiere alcanzar con ellos.

En el caso del aprendizaje por refuerzo, en el que el aprendizaje se produce desde cero a partir de la interacción de un agente con su entorno, las reglas de conocimiento que se infieren son originales y no siguen ningún tipo de sesgo.



Es por ello que el análisis que se puede aplicar en este sentido se conoce como *análisis prescriptivo*, ya que el conocimiento que surge aporta enfoques desconocidos hasta el momento, así como un nuevo paradigma en la toma de decisiones a la hora de solucionar el problema propuesto.

6.1. Conceptos y definiciones

El aprendizaje por refuerzo es un tipo de aprendizaje basado en el ensayo y error (Sutton y Barto, 1992). Los dos componentes principales del aprendizaje por refuerzo son el entorno y el agente y, mediante la interacción entre el agente y el entorno, el agente va adaptándose y aprendiendo para alcanzar un objetivo predefinido. Este objetivo estará definido en términos de recompensa, que es el valor que el entorno va devolviendo al agente en cada paso de la interacción. El objetivo final del agente es encontrar una estrategia óptima que maximice la recompensa obtenida.

Para sacar todo el partido posible a las opciones que el aprendizaje por refuerzo ofrece, tenemos que detenernos en la nomenclatura y la terminología necesaria. Lo haremos de manera breve, dando una definición concisa de los términos más importantes. Como curiosidad, en Sutton y Barto (1992) se analiza también la relación de esta rama con otras, como la psicología o la neurociencia, de ahí que muchos de los conceptos que se revisarán a lo largo del capítulo se usen en estos campos también.

6.1.1. Conceptos básicos

A continuación, presentamos algunos de los conceptos básicos:

1. **Entorno.** Es el medio donde nuestro agente se va a desarrollar. Está formado por un conjunto de elementos con los que el agente podrá interactuar para alcanzar su objetivo. Además de elementos interactivos, el entorno también se define en términos de lógica de funcionamiento que afectan directamente a la forma en la que se desarrollará el proceso de aprendizaje.
2. **Agente.** Es el elemento digital capaz de aprender y adaptarse al entorno en el que se desarrolla y que, por medio de la interacción con él, va obteniendo experiencia y va mejorándose con el fin de encontrar una estrategia óptima para alcanzar el objetivo definido.

3. **Acción.** Las acciones forman el conjunto de posibilidades que un agente puede ejecutar en un momento determinado. Dependiendo de la acción que se ejecute, el entorno proporcionará una retroalimentación específica al agente. Normalmente, el conjunto de acciones disponibles para realizar un experimento de aprendizaje por refuerzo es fijo para así poder modelar las acciones teniendo en cuenta la retroalimentación obtenida. Es importante tener en cuenta que el conjunto de acciones disponibles puede ser intratable dependiendo del problema, lo que impacta en el tipo de técnicas que se pueden usar para llegar a una solución.
4. **Observación.** Como parte de la retroalimentación que el entorno devuelve al agente cuando este toma una acción en un momento determinado, se encuentra la observación. La observación es una fotografía o resumen del punto en el que se encuentra el entorno en ese momento. Por ejemplo, si nuestro entorno fuera un videojuego, la observación, en un momento determinado, sería una imagen de la pantalla del videojuego.
5. **Estado.** La información que encontramos en las observaciones es lo que se conoce como *información raw*. En la mayoría de ocasiones, el agente no es capaz de interpretar satisfactoriamente esta información, ya que necesita algún tipo de preprocesamiento. El estado es el resultado de aplicar todas estas tareas sobre la observación y así tener los datos listos para poder usarlos en el modelo.



La diferencia entre observación y estado es muy sutil y, dependiendo del autor o recurso, hay veces que los términos se intercambian. Lo importante es tener en mente que siempre es necesario transformar la información antes de usarla como entrada del modelo.

6. **Recompensa.** Es el resultado que el entorno devuelve al agente a partir de la acción que se ha ejecutado. Desde el punto de vista del aprendizaje, la recompensa es la medida que nos indica si se está aprendiendo o no. El objetivo del agente es encontrar la estrategia óptima para maximizar la recompensa que se obtiene.
7. **Episodio y paso (step en inglés).** Normalmente, la interacción entre el agente y el entorno se mide en episodios. El episodio sirve para controlar el número de ejecuciones completas que lleva a cabo nuestro agente. Cada instante de tiempo durante la ejecución de un episodio es lo que se conoce como *paso*. Intuitivamente, podemos entender un paso como un momento de tiempo congelado durante la ejecución de la solución. El paso también se usa como medida en pruebas de rendimiento (*benchmarks* en inglés) y comparaciones entre algoritmos y soluciones.



Si hablásemos de videojuegos, un episodio sería una partida completa, desde su inicio hasta que aparece el texto “game over”. Si nos referimos al campo de la robótica, tendríamos que definir el final de la ejecución.

6.1.2. Conceptos avanzados

A continuación, exponemos algunos conceptos avanzados:

1. **Estrategia.** La estrategia, o *policy*, define el método de selección de acciones que sigue el agente para alcanzar la recompensa máxima. La estrategia es el componente que va a ir evolucionando a medida que el agente va aprendiendo. Como se ha comentado anteriormente, el objetivo final del agente es encontrar la estrategia óptima que maximice la recompensa.
2. **Exploración.** Para entender el concepto de exploración tenemos que centrarnos en la estrategia. Cuando hablamos de estrategia nos referimos a cómo el agente selecciona las acciones a ejecutar en el entorno. Al comienzo del proceso de aprendizaje, este proceso es aleatorio y, conforme van avanzando los episodios ejecutados, el proceso se vuelve menos aleatorio y da paso a que la selección de las acciones se base en el aprendizaje acumulado. El intervalo de tiempo (normalmente, medido en pasos) entre aleatoriedad total y aleatoriedad mínima a la hora de seleccionar acciones es lo que se conoce como *proceso de exploración*.
3. **Explotación.** La fase de explotación se produce cuando el agente ha aprendido lo suficiente y es capaz de decidir qué acciones ir realizando de manera satisfactoria. Esta fase ocurre a continuación de la fase de exploración. Es común usar un pequeño porcentaje de aleatoriedad para evitar que el agente se quede bloqueado en situaciones nuevas o en mínimos locales.



Dependiendo del algoritmo que se use, los conceptos de exploración y explotación se pueden intercambiar. Intuitivamente, la idea es saber que, al comienzo del aprendizaje, el agente ejecuta acciones aleatoriamente para ir adquiriendo experiencia y, seguidamente, emplear ese conocimiento acumulado para así ir perfeccionando la toma de decisiones. Al final, su conocimiento será suficiente para realizar siempre la mejor acción disponible.

4. **Experiencia.** Es la estructura de datos donde se irá almacenando la información generada durante la ejecución. El elemento mínimo que forma la experiencia es lo que se conoce como *transición*, y está formada por el estado actual, la acción tomada, la recompensa obtenida y el siguiente estado. Las transiciones se obtienen en cada paso de la ejecución y se van almacenando en la experiencia global. Serán conjuntos de estas transiciones los que se usarán en el entrenamiento del modelo.

6.2. Algoritmos de aprendizaje por refuerzo

Hasta ahora hemos hablado de aprendizaje por refuerzo a grandes rasgos, centrándonos en los conceptos desde un punto de vista más teórico que funcional. A lo largo de los puntos anteriores hemos hecho referencia varias veces al concepto de modelo, pero sin entrar demasiado en detalle. En este capítulo es el momento de centrarnos en los algoritmos y modelos que usaremos para nuestras soluciones de aprendizaje por refuerzo.

Comenzamos este capítulo diferenciando entre algoritmo y modelo. Un algoritmo es toda la lógica sobre la que una solución de aprendizaje por refuerzo se sostiene, sobre todo en relación con la forma de encarar el problema y con el uso que hace de los componentes necesarios para alcanzar un resultado satisfactorio. Resaltaremos en estos componentes el tipo de estrategia a seguir y el uso que se va a hacer de los datos disponibles.

Por otro lado, un modelo es la función que se encargará de traducir estados del entorno en acciones a ejecutar. Podemos ver el modelo como el conocimiento necesario para seguir una estrategia satisfactoria. El hecho de que la estrategia vaya cambiando durante las ejecuciones se debe a los cambios que también sufre el modelo, es decir, se debe a la evolución del modelo debido al proceso de entrenamiento.

Cuando hablamos de modelos, no necesariamente nos referimos a modelos de **aprendizaje automático** (*machine learning* en inglés) o, más concretamente para el caso del aprendizaje por refuerzo, a modelos de **aprendizaje profundo** (*deep learning*, en inglés). El modelo es la función aproximadora de estados a acciones. El hecho de que usemos modelos basados en aprendizaje profundo se debe al tipo de dato con el que se trabajará.



Como ejemplo típico en los últimos años, encontramos el trabajo de Mnih et al. (2015) en el caso de los videojuegos. Al trabajar con información visual en un píxel, el tipo de modelo que se usa son **redes convolucionales** para tratar y explotar el dato de la manera más óptima posible durante el proceso de aprendizaje.

6.2.1. Clasificación de algoritmos de aprendizaje por refuerzo

Tal y como describen Arulkumaran, Deisenroth, Brundage y Bharath (2017), podemos encontrar diversas formas de clasificar los distintos enfoques que podemos hallar en las soluciones de aprendizaje por refuerzo. En nuestro caso, vamos a centrar la clasificación en dos vías: algoritmos basados en la estrategia y algoritmos basados en el modelo.

Algoritmos basados en la estrategia

Como veíamos en el capítulo anterior, la estrategia se refiere a la forma en la que el agente selecciona una acción en un estado. Dependiendo del algoritmo que se implemente y del momento en el que se encuentre el proceso de aprendizaje, la selección de acciones es diferente, ya que se tienen en cuenta elementos como el conjunto de la experiencia que se está usando, el tipo de evaluación que se hace de cada acción o el grado de aleatoriedad que se tendrá en cuenta.

En nuestro caso, diferenciamos dos grupos dependiendo de la estrategia que se siga: *on-policy* y *off-policy*. Una estrategia *on-policy* se enfoca en hacer uso de la experiencia a corto plazo. Por experiencia a corto plazo nos referimos al conjunto de transiciones que se han ido almacenando en un número de pasos fijo. Este tipo de estrategia implica que cada vez que el modelo del agente se entrena, lo hará solo teniendo en cuenta la experiencia almacenada con la estrategia (versión del modelo) anterior. Una vez el modelo se ha actualizado o entrenado, la experiencia se desechará y se comenzará una **trayectoria** nueva.

En el otro lado, encontramos la estrategia *off-policy*. En esta estrategia, la actualización del modelo se realiza cada cierto número de pasos, pero teniendo en cuenta toda la experiencia que se ha ido almacenando hasta ese momento, usando un conjunto aleatorio extraído de la experiencia total para entrenar el modelo. Al usar la experiencia total se emplearán transiciones que se han obtenido con estrategias pasadas, incluso estrategias totalmente aleatorias heredadas del comienzo de la ejecución.



Enlace de interés

Este repositorio educativo de contenidos relacionados con el aprendizaje profundo por refuerzo es obra de la empresa de inteligencia artificial Openai. En este caso se centra en la diferencia entre estrategia *on-policy* y *off-policy*.

<https://spinningup.openai.com/en/latest/user/algorithms.html#the-on-policy-algorithms>

Algoritmos basados en el modelo

En paralelo a la clasificación relacionada con la estrategia a seguir, nos centraremos en la clasificación basada en el modelo. El primer detalle importante a tener en cuenta es que el modelo al que nos referimos en esta clasificación es el modelo del entorno, es decir, si, en cualquier momento de la ejecución, tenemos acceso a las dinámicas y a toda la información que el entorno nos puede ofrecer. Esto cambiaría el contexto de la toma de decisiones, ya que, al disponer de toda esta información adicional, la selección de acciones se ve impactada por este conocimiento. Dependiendo del conocimiento de este modelo, encontramos dos conjuntos de algoritmos: *model-based* y *model-free*.

Como indicamos en el párrafo anterior, si tenemos la información suficiente para modelar el comportamiento del entorno, estaremos trabajando en una solución *model-based*. Este tipo de soluciones se suelen apoyar en otras técnicas dentro de la inteligencia artificial, como, por ejemplo, los **algoritmos de planificación** o las búsquedas basadas en **(meta)heurísticas**. Al conocer el modelo que rige el entorno, podemos estimar las mejores acciones en el estado actual a partir de estimaciones a futuro, ya que, en todo momento, podemos aproximar esa estimación de una manera más precisa.



El obstáculo que podemos encontrarnos es que, en problemas complejos, el espacio de acciones y de estados es muy amplio y, por ello, la mayoría de las veces tendremos que combinar nuestra solución de aprendizaje por refuerzo con otros algoritmos que nos permitan aplicar una solución más tratable desde el punto de vista computacional.

En el caso de no conocer cómo se comporta nuestro entorno, trabajaremos en un dominio *model-free*. Este tipo de problema es el que nos encontraremos en la mayoría de retos actuales relacionados con aprendizaje por refuerzo, como es el caso de la aplicación de estas soluciones en videojuegos de Atari, en Mnih et al. (2015). Al no conocer un modelo del entorno, nuestra solución se centrará en aproximar lo mejor posible las mejores acciones para distintos estados del entorno.



Enlace de interés

Este repositorio educativo de contenidos relacionados con el aprendizaje profundo por refuerzo es obra de la empresa de inteligencia artificial Openai. En este caso, se diferencia entre *model-free* y *model-based*. Se muestran algunos ejemplos de algoritmos para cada grupo.

https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

6.2.2. En detalle: Deep Q-Networks y Policy Gradients

En este capítulo vamos a entrar en detalle en dos ejemplos básicos de algoritmos de aprendizaje por refuerzo como son Deep Q-Networks y Policy Gradients. Estos dos algoritmos son la base sobre la que se sustentan algoritmos más avanzados que han ido apareciendo estos últimos años como A3C (Mnih et al., 2016), PPO (Schulman, Wolski, Dhariwal, Radford y Klimov, 2017) o DDPG (Silver et al., 2014). Estas nuevas versiones han ido evolucionando y mejorando distintos aspectos de Deep Q-Networks y Policy Gradients, como los que enumeramos a continuación:

- Uso de la experiencia para el proceso de aprendizaje.
- Adaptaciones en las funciones de coste para potenciar situaciones y definir cómo se realizará la toma de decisiones.
- Combinación de distintos enfoques para quedarnos con lo más destacado de cada uno.
- Explotación de la capacidad computacional y la ejecución multiproceso.

En este manual nos centraremos en las versiones más básicas de cada algoritmo. Trataremos en detalle cómo se desarrolla el proceso de aprendizaje en cada caso y cuáles serían los componentes más destacados.

Deep Q-Networks

Deep Q-Networks (Mnih et al., 2015) es un algoritmo de aprendizaje por refuerzo basado en el algoritmo de Q-Learning cuya principal aportación es la combinación de este algoritmo con técnicas de aprendizaje profundo.



El modelo de aprendizaje profundo más usado en las soluciones Deep Q-Networks son las redes convolucionales. Esto se debe a que, en la mayoría de casos, el entorno de simulación está basado en videojuegos y el tipo de dato disponible es sobre cada píxel.

Q-Learning (Watkins y Dayan, 1992) es un algoritmo que se centra en cuantificar la calidad (la Q del nombre viene del inglés, *quality*) de seleccionar una acción en un estado determinado. La calidad, a su vez, será definida como ‘recompensa esperada en el futuro’.



El objetivo del algoritmo es encontrar la función Q óptima, esto es, la función que en cada estado ejecuta la mejor acción de entre todas las disponibles. Es importante notar que la función Q se definirá en relación al par estado-acción.

Esta función Q también se puede interpretar como la función encargada de definir la estrategia que sigue el agente. En el caso de Q-Learning, la estrategia que comúnmente se utiliza es una estrategia *greedy* apoyada en el proceso de exploración-explotación que hemos visto anteriormente.

Matemáticamente, esta estrategia se define de la forma siguiente:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

Figura 24. Estrategia en el algoritmo de DQN. Adaptado de “Human Level Control Through Deep Reinforcement Learning”, por V. Mnih et al, 2015, *Nature*, 518, p. 2.

Cada vez que el agente tiene que decidir qué acción tomar, la estrategia estimará las recompensas esperadas de cada una de las acciones y seleccionará la acción que le retorne el valor más alto. Siguiendo esta estrategia durante todos los pasos de la ejecución se llegaría a la secuencia de acciones óptima para alcanzar la mayor recompensa posible.

Cuando hablamos de recompensa esperada, nos referimos a la suma de recompensas futuras que espera el agente en un estado determinado. Un elemento clave en esta definición es el concepto de futuro. Para hacer una estimación del valor futuro, se definirá la recompensa futura en términos de expectativa.

$$Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a] \longrightarrow R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Figura 25. Recompensa esperada en DQN. Adaptado de “Asynchronous methods for Deep Reinforcement Learning”, por V. Mnih et al., 2016, *Proceedings of Machine Learning Research*, 16, p. 2.

La fórmula de la izquierda, la podemos interpretar como la estimación de la recompensa esperada a partir de la información que se va obteniendo durante la ejecución. A la derecha, tenemos el cálculo de la recompensa para el estado actual basado en la recompensa inmediata y la suma de las recompensas futuras. Es importante tener en cuenta el factor multiplicativo de cada recompensa de los estados siguientes, conocido como *factor de descuento* (*discount factor* en inglés). Este factor define la importancia que damos a las recompensas futuras en nuestra estimación, y su valor por defecto se encuentra entre 0,95 y 0,99.

Para el cálculo de la recompensa esperada utilizaremos la ecuación de Bellman, que se define de la siguiente manera:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Figura 26. Ecuación de Bellman de DQN. Adaptado de “Human Level Control Through Deep Reinforcement Learning”, por V. Mnih et al, 2015, *Nature*, 518, p. 2.

Vamos a analizar los distintos elementos que componen esta ecuación. Para empezar, está compuesta a partir del conjunto de elementos que hemos ido almacenando en nuestra experiencia: estado, acción, recompensa y siguiente estado. Esta es la razón por la que se almacenan estos campos en la experiencia, ya que serán los elementos que se necesiten para la función de coste usada durante el proceso de aprendizaje.

Otro punto importante se relaciona con la recurrencia definida en la fórmula. En cada estado se selecciona una acción, lo que nos devuelve la recompensa actual:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon} [r + \gamma \max_{a'} Q^*(s', a') | s, a]$$

Figura 27. Ecuación de Bellman de DQN. Adaptado de “Human Level Control Through Deep Reinforcement Learning”, por V. Mnih et al, 2015, *Nature*, 518, p. 2.

Ahora tenemos que incluir cómo de bueno es el estado al que nos movemos. De ese objetivo se encarga el segundo operando, que analiza todas las acciones disponibles para ver qué recompensa nos devuelve cada una y así saber cuál es la mejor recompensa en el siguiente estado:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon} [r + \gamma \max_{a'} Q^*(s', a') | s, a]$$

Figura 28. Ecuación de Bellman de DQN. Adaptado de “Human Level Control Through Deep Reinforcement Learning”, por V. Mnih et al, 2015, *Nature*, 518, p. 2.

Uniendo los dos operandos obtenemos la recompensa esperada en el estado actual para la acción seleccionada. Esta fórmula, definida en términos de recurrencia temporal, nos permite ejecutar la ecuación de Bellman durante un número finito de estados para ir construyendo la función Q de forma cada vez más precisa.

Aprendizaje profundo en Deep Q-Networks

Con las fórmulas definidas previamente somos capaces de obtener una estrategia óptima en espacios de estados y de acciones manejables computacionalmente. Si el problema tiene demasiada complejidad, no seremos capaces de modelar nuestros pares de estado-acción, debido a la cantidad de situaciones y estados distintos que nuestro agente encontrará. Es justo en estas situaciones cuando se usará un modelo de red neuronal como función aproximadora, como en Mnih et al. (2015).

El hecho de reemplazar nuestra función Q por una red neuronal hace que necesitemos tener en cuenta todos los detalles necesarios para que una solución basada en aprendizaje profundo funcione correctamente. En este manual nos centraremos en el uso que se hace de la experiencia y en la definición de la función de coste.

Experiencia

Al usar redes neuronales en nuestra solución de aprendizaje por refuerzo, haremos uso de la experiencia que se va almacenando durante la ejecución para entrenar el modelo. En el caso concreto de Deep Q-networks, la estrategia que seguiremos es una estrategia *off-policy*, ya que se usarán transiciones que se han ido almacenando con distintas versiones del modelo en distintos momentos de la ejecución.

El entrenamiento se realizará usando subconjuntos aleatorios de transiciones, a partir de la experiencia global. Es en estos subconjuntos donde se pueden encontrar transiciones de ejecuciones pasadas.

Función de coste

Una vez conocida la estrategia que utilizaremos para usar la experiencia adquirida durante el entrenamiento, pasamos a la definición de la función de coste. La función de coste para medir el error que se va cometiendo está basada en la ecuación de Bellman (Mnih et al., 2015). De hecho, lo que haremos será diferenciar la ecuación de Bellman entre el operando objetivo y el operando predictivo:

$$L_i(\theta_i) = \mathbb{E} \left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - \underbrace{Q(s, a; \theta_i)}_{\text{Predicción}} \right)$$

Objetivo

Figura 29. Recompensa esperada en DQN. Adaptado de “Asynchronous methods for Deep Reinforcement Learning”, por V. Mnih et al., 2016, *Proceedings of Machine Learning Research*, 16, p. 2.

Para ir entrenando el modelo, se usará esta función de coste cada número de pasos fijado previamente, que será el intervalo de tiempo en el que el agente irá recolectando experiencia para, luego, usarla en el momento de actualizar los pesos del modelo.

Hay que destacar un detalle importante y es que ambos operandos dependen de la misma función Q . Si se implementa la función de coste basada en Q , es decir, el modelo de red neuronal, tanto la predicción como el valor esperado van a estar cambiando constantemente durante el proceso de aprendizaje, lo que provocará que el modelo no sea capaz de aprender correctamente. Es más, esta definición de la función de coste derivará la mayoría de las veces en una situación de divergencia en la que nuestro agente no es capaz de aprender.

Para solventar esta situación, en Mnih et al. (2015) se propone definir dos modelos neuronales iguales en su arquitectura. Uno de los modelos se encargará de predecir las acciones y el otro será el responsable de devolver el resultado para comparar con la acción elegida. Este segundo modelo se conoce como *modelo target* y es un punto crítico para un correcto funcionamiento del algoritmo de Deep Q-Networks. Este modelo va a tener los mismos pesos que el modelo predictivo, pero, a diferencia de este, se actualizará en intervalos de tiempo más largos. Es decir, el resto del tiempo se quedará congelado y, por tanto, mantendrá los mismos valores durante ese intervalo de tiempo, lo que permite comparar siempre con el mismo resultado.



Enlace de interés

Aquí se muestra un ejemplo de implementación del algoritmo Deep Q-Networks en Pytorch para el entorno de CartPole. Se explican en detalle los bloques importantes de la solución global. Posdesarrollado por Adam Paszke.

https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Policy Gradients

El algoritmo de Policy Gradients (Sutton, Mcallester, Singh y Mansour, 2000) es un algoritmo de aprendizaje que, como hemos visto anteriormente, se centra en obtener la estrategia óptima relacionada con la recompensa que el agente va obteniendo.

En la versión base de Policy Gradients, a diferencia de Deep Q-Networks, no se hace una estimación de la recompensa esperada relacionando los pares estado y acción, sino que en cada estado se obtiene la probabilidad de cada acción.

$$\pi_{\theta}(a|s) = \mathbb{P}[a|s; \theta]$$

Figura 30. Función de probabilidad de acciones en Policy Gradients.
Adaptado de “Deterministic Policy Gradient Algorithms”, por D. Silver et al., *Proceedings of Machine Learning Research*, 32, p. 1.

Trabajar sobre el espacio de probabilidades de las acciones implicará que no se apliquen los procesos de exploración y explotación tal y como hemos visto en el manual. En vez de ello, cada vez que el agente necesite seleccionar una acción, lo hará a partir de una distribución de probabilidades aleatoria ponderada por las probabilidades de cada acción.

Esta diferencia es clave para entender el algoritmo Policy Gradients. El aprendizaje se seguirá desarrollando en términos de recompensa esperada, pero en un intervalo finito de pasos. Esta forma de ejecución provocará que el cálculo de la recompensa esperada se pueda calcular para cada conjunto de pasos sin necesidad de definir una función de recurrencia que trabaje con un límite a futuro.

Otra característica de Policy Gradients es que es un algoritmo *on-policy*. Cada vez que se ejecute el conjunto de pasos finito, la experiencia se usará para actualizar el modelo. Tras la actualización de los pesos de nuestro modelo, la experiencia se actualizará por completo. De esta forma, solo se tendrá en cuenta la experiencia obtenida con la última versión del modelo para actualizar sus pesos.

La estimación de las probabilidades de cada acción se define con una función, a partir de la recompensa obtenida. Intuitivamente, en cada estado iremos seleccionando una acción y obtendremos una recompensa. Si la recompensa es positiva, esa acción es positiva, por lo que, para ese estado, se querrá repetir en el futuro. Si la recompensa es negativa, no se querrá seleccionar esa acción.

$$\nabla_{\theta} \log \pi(a_t | s_t; \theta) R_t$$

Figura 31. Estimación de la probabilidad de una acción en base a su recompensa. Adaptado de “Asynchronous methods for Deep Reinforcement Learning”, por V. Mnih et al., 2016, *Proceedings of Machine Learning Research*, 16, p. 3.



Para facilitar las operaciones matemáticas, la distribución de probabilidades de las acciones se trasladará a una escala logarítmica.



Esta forma de actualización de las probabilidades de las acciones se ve impactada negativamente por la varianza de los datos respecto a las recompensas obtenidas. Sobre todo al comienzo de las ejecuciones, la misma acción en un estado determinado puede devolver recompensas positivas o negativas, por lo que usar solo la recompensa como factor de importancia de la acción es demasiado básico.

Aprendizaje profundo en Policy Gradients

Al igual que ocurría con Deep Q-Networks, la estimación de la estrategia se complica cuando los espacios de estados y acciones se vuelven demasiado complejos en cuanto a posibles combinaciones y decisiones.

Es por ello que el uso de redes neuronales ayuda a modelar las probabilidades del espacio de acciones a partir de un estado dado. Nuevamente, respecto al aprendizaje profundo, vamos a centrarnos en el uso que se hace de la experiencia y en la definición de la función de coste.

Experiencia

En el caso de Policy Gradients, ya hemos explicado previamente cómo se usa la experiencia. Al seguir una estrategia *on-policy*, se usará toda la experiencia almacenada durante el rango de pasos definido para actualizar los pesos del modelo.

Una vez entrenado el modelo, se eliminará toda la experiencia acumulada para comenzar una nueva trayectoria y empezar a acumular nueva experiencia con la nueva versión del modelo.

Al igual que en el caso de Deep Q-Networks, se utilizarán todos los elementos almacenados en cada transición para poder ir estimando las probabilidades de las acciones y sus respectivas recompensas.

Función de coste

En Policy Gradients, la función de coste se basará en el cálculo de la distribución de probabilidades introducido en el punto anterior:

$$\nabla_{\theta} \log \pi(a_t | s_t; \theta) R_t$$

Figura 31. Estimación de la probabilidad de una acción en base a su recompensa. Adaptado de “Asynchronous methods for Deep Reinforcement Learning”, por V. Mnih et al., 2016, *Proceedings of Machine Learning Research*, 16, p. 3.

Esta función está definida en términos de maximizar el valor resultante, por lo que se cambia su signo para transformarla en una función de minimización y así poder aplicar el algoritmo de aprendizaje de retropropagación para el modelo de red neuronal.

El elemento más importante en la función de coste es el factor multiplicativo de la probabilidad de la acción. Anteriormente, se indicó cómo usar la recompensa directamente puede provocar una gran varianza en los datos y, por tanto, guiar el proceso de aprendizaje hacia mínimos locales o hacia la divergencia de la solución.

Para solucionar esta situación, podemos encontrar en Schulman, Moritz, Levine, Jordan y Abbeel (2016) las siguientes alternativas:

$$g = \mathbb{E} \left[\sum_{t=0}^{\infty} \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right] \quad (1)$$

Ψ_t puede ser uno de los siguientes:

- | | |
|---|--|
| 1. $\sum_{t=0}^{\infty} r_t$: recompensa total de la trayectoria. | 4. $Q^{\pi}(s_t, a_t)$: función de valor basado en estado-acción. |
| 2. $\sum_{t'=t}^{\infty} r_{t'}$: recompensa siguiendo la acción a_t . | 5. $A^{\pi}(s_t, a_t)$: función de ventaja. |
| 3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: versión con un umbral de la fórmula anterior. | 6. $r_t + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$: función basada en diferencias temporales. |

Las últimas fórmulas siguen las definiciones:

$$V^{\pi}(s_t) := \mathbb{E}_{\substack{s_{t+1}, \dots \\ a_{t+1}}} \left[\sum_{l=0}^{\infty} r_{t+l} \right] \quad Q^{\pi}(s_t, a_t) := \mathbb{E}_{\substack{s_{t+1}, \dots \\ a_{t+1}}} \left[\sum_{l=0}^{\infty} r_{t+l} \right] \quad (2)$$

$$A^{\pi}(s_t, a_t) := Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \quad (\text{función de ventaja}) \quad (3)$$

Figura 32. Posibles factores para las funciones de coste en Policy Gradients. Adaptado de *High-dimensional continuous control using generalized advantage estimation*, por J. Schulman, P. Moritz, S. Levine, M. I. Jordan y P. Abbeel, 2016, p. 2.

Lo que resalta de cada versión de estas funciones para estimar la distribución de probabilidades del espacio de acciones es cómo intentan mitigar el problema de la varianza con los factores que multiplican la probabilidad de la acción. La primera y la segunda fórmula se centran en un refinamiento de la recompensa obtenida, pero siguen la misma definición presentada anteriormente.

En la tercera fórmula se utiliza un valor basal (*baseline* en inglés). Este umbral definirá el límite para considerar una acción positiva o no. El factor de importancia de la acción está definido usando la recompensa actual y un valor por defecto que, normalmente, se define a partir del análisis del entorno y de las recompensas para saber en torno a qué valores se mueve.

En las siguientes fórmulas aparece un nuevo concepto, la estimación del *value* (o valor). El *value* es un concepto muy destacado en aprendizaje por refuerzo y, a grandes rasgos, significa ‘cómo de bueno es el estado’ en el que se encuentra el agente. Por ello, se puede usar para medir la bondad del estado y, por tanto, saber si se va por buen camino con la estrategia actual.

Es común definir el factor de ponderación de las probabilidades de las acciones en función del *value*, ya que devuelve un umbral realista del valor medio de las recompensas del estado y, así, si la recompensa es mayor, la acción es positiva y viceversa.



Intuitivamente, el *value* es la media de recompensas de todas las acciones que se pueden tomar en un estado.



Al estimar las probabilidades de las acciones y el *value* del estado, el algoritmo Policy Gradients se transforma en el conocido algoritmo Actor-Critic (Konda y Tsitsiklis, 2000). En este algoritmo, el Actor correspondería a la distribución de probabilidades y el Critic, al *value* estimado, que es el que se encarga de decir cómo de buena es la decisión que se toma.

El uso del *value* con la recompensa actual como factor de ponderación nos lleva a la definición de *ventaja*, que es el parámetro que se observa en la ecuación número 5 de la Figura 32. Las tres últimas ecuaciones hacen uso de una función Q similar a la que hemos visto en Deep Q-Networks y en funciones basadas en diferencias temporales, teniendo en cuenta el *value* del siguiente estado también. La función más usada actualmente es la ecuación número 5 de la Figura 32, que define el factor de ponderación en términos de una función de ventaja.



Enlace de interés

Este es uno de los artículos más reconocidos dentro de Policy Gradients, usando el entorno de Pong. El nivel de detalle y calidad de este artículo es muy útil para entender a bajo nivel el funcionamiento de Policy Gradients. Desarrollado por Andrej Karpathy.

<http://karpathy.github.io/2016/05/31/rl>



Glosario

Algoritmo de planificación

Los algoritmos de planificación se forman parte de la rama de la inteligencia artificial de planificación automática. Una de las características principales de estos algoritmos es la definición de una secuencia de acciones reguladas por precondiciones y poscondiciones que se deben cumplir cumplir durante su ejecución. Estas condiciones están relacionadas con características del entorno y del objetivo a alcanzar.

Agregación

Capas empleadas para la reducción de dimensionalidad espacial o *pooling* en redes neuronales convolucionales. Su objetivo es reducir la cantidad de parámetros de la red sin perder las características más relevantes que se van extrayendo a lo largo de ella.

Aprendizaje no supervisado

El aprendizaje no supervisado es un método de aprendizaje automático en el que un modelo se ajusta a las observaciones. Se distingue del aprendizaje supervisado por el hecho de que no hay un conocimiento previo. En el aprendizaje no supervisado, se trata un conjunto de datos de objetos de entrada. Así, el aprendizaje no supervisado típicamente trata los objetos de entrada como un conjunto de variables aleatorias, y se construye un modelo de densidad para el conjunto de datos.

Aprendizaje por refuerzo

El aprendizaje por refuerzo, o aprendizaje reforzado, es un área del aprendizaje automático inspirada en la psicología conductista, cuyo objetivo es determinar qué acciones debe escoger un agente de software en un entorno dado con el fin de maximizar alguna noción de recompensa o premio acumulado. El problema, por su generalidad, se estudia en muchas otras disciplinas, como la teoría de juegos, la teoría de control, la investigación de operaciones, la teoría de la información, la optimización basada en la simulación, estadísticas y algoritmos genéticos.

Aprendizaje profundo

El aprendizaje profundo (en inglés, *deep learning*) es un conjunto de algoritmos de aprendizaje automático (en inglés, *machine learning*) que intenta modelar abstracciones de alto nivel en datos usando arquitecturas computacionales que admiten transformaciones no lineales múltiples e iterativas de datos expresados en forma matricial o tensorial.

Aprendizaje supervisado

En aprendizaje automático y minería de datos, el aprendizaje supervisado es una técnica para deducir una función a partir de datos de entrenamiento. Los datos de entrenamiento consisten de pares de objetos (normalmente vectores): una componente del par son los datos de entrada y la otra, los resultados deseados. La salida de la función puede ser un valor numérico (como en los problemas de regresión) o una etiqueta de clase (como en los de clasificación). El objetivo del aprendizaje supervisado es crear una función capaz de predecir el valor correspondiente a cualquier objeto de entrada válida después de haber visto una serie de ejemplos.

Batch

Conjunto de muestras o instancias que se propagan hacia delante en el caso de las redes neuronales. Si se lleva a cabo un entrenamiento por *batches*, el error al final de la red se calcula como la media del error de cada una de las muestras del *batch* (en el momento en que todas han realizado la propagación hacia delante).

Campo receptivo

Conjunto (o vecindario) de píxeles en la imagen de entrada o activaciones en un mapa de activación involucrados en la operación de convolución. Concretamente, se trata del área involucrada en la operación de suma ponderada según marcan los parámetros del núcleo o *kernel* del filtro de una capa convolucional.

Capa de entrada

Se trata de la primera capa de una red neuronal. Dicha capa recibe los datos o señales procedentes del entorno.

Capa de salida

Se trata de la última capa de una red neuronal. El número de unidades de dicha capa está relacionada tanto con la tarea que hay que resolver como con el número de clases involucradas en ella.

Capa oculta

Capa que no tiene conexión directa con el entorno. Esta capa puede ser precedida por otras capas ocultas o bien por la capa de entrada. La última capa oculta va conectada a la capa de salida.

Convolución

En redes neuronales convolucionales, la operación de convolución es el tratamiento de una matriz de entrada con otra, denominada *núcleo* o *kernel*. El filtro matriz de convolución usa una primera matriz, que es la imagen que se tratará. La imagen es una colección bidimensional de píxeles en coordenadas rectangulares. El *kernel* usado está compuesto por pesos que se deben optimizar durante el proceso de entrenamiento.

Descenso de gradiente

El método de descenso por gradiente, *gradient descent* en inglés, es uno de los algoritmos de optimización más populares en aprendizaje automático, particularmente por su uso extensivo en el campo de las redes neuronales. *Gradient descent* es un método general de minimización para cualquier función. La versión original se considera lenta pero versátil, sobre todo para casos de funciones multidimensionales.

Entropía cruzada

En teoría de la información, la entropía cruzada entre dos distribuciones de probabilidad mide la media de bits necesarios para identificar un evento de un conjunto de posibilidades si un esquema de codificación está basado en una distribución de probabilidad dada q , más que en la verdadera distribución p .

Época

En aprendizaje profundo, hablamos de época cuando se ha llevado a cabo el proceso de propagación hacia delante y retropropagación con todas las muestras de entrenamiento. Si se realiza un entrenamiento por *batches*, cuando todos estos han sido procesados, se habrá llevado a cabo una época o iteración en el proceso de entrenamiento de la red.

Error cuadrático medio

Mide el promedio de los errores al cuadrado, es decir, la diferencia entre el estimador y lo que se estima. En aprendizaje automático supervisado, el error cuadrático medio es el criterio de evaluación más usado para problemas de regresión.

Función de activación

Una función de activación se encarga de devolver una salida a partir de un valor de entrada, normalmente el conjunto de valores de salida en un rango determinado, como $(0, 1)$ o $(-1, 1)$. Se busca que las funciones introduzcan no linealidades, para poder separar este tipo de conjuntos de datos, y que las derivadas de estas funciones sean simples, para minimizar con ello el coste computacional.

Función de pérdidas

Una función de pérdida, o función objetivo, $J(x)$ mide el nivel de satisfacción con las predicciones de nuestro modelo con respecto a una respuesta correcta utilizando ciertos valores de θ . Existen varias funciones de pérdida, como el error cuadrático medio o entropía cruzada, y la selección de una de ellas depende de varios factores, como el algoritmo seleccionado o el nivel de confianza deseado, pero principalmente depende del objetivo de nuestro modelo.

ImageNet

El proyecto ImageNet es una gran base de datos de imágenes diseñada para su uso en la investigación de algoritmos de detección reconocimiento de objetos visuales. Se compone de más de 14 millones de imágenes anotadas manualmente para indicar qué objetos están representados en al menos un millón de las imágenes. También se proporcionan las *bounding boxes*. ImageNet contiene más de 20 000 categorías. Adicionalmente, ImageNet también es el nombre del concurso anual que se realiza para la investigación en algoritmos que mejoren la clasificación del conjunto de datos.

Kernel

El *kernel*, *núcleo* en español, es el conjunto de parámetros optimizables que componen un filtro convolucional. Durante la operación de convolución, el *kernel* se desliza a través de la imagen para realizar dicho cálculo.

Mapa de activación

Salida de la operación de convolución en las redes neuronales convolucionales. Se realizan operaciones de productos y sumas entre una capa y los n filtros (definidos por un *kernel*) de la capa convolucional, que generan varios mapas de características normalmente aglutinados en forma de volumen 3D.

(Meta)heurística

Una metaheurística es un método heurístico usado para resolver problemas de computabilidad, principalmente en situaciones donde no se puede implementar una solución óptima. Por ello, su objetivo es encontrar la mejor solución posible en un tiempo viable.

Padding

Método que se encarga de agregar píxeles alrededor de una imagen para evitar el efecto borde durante la operación de convolución en CNN. Gracias a esta agregación sobre los bordes, es posible capturar información relevante contenida en ellos. Se suele emplear el relleno con ceros (*zero padding*) aunque existen otros.

Propagación hacia delante

En redes neuronales artificiales, se refiere al conjunto de operaciones de multiplicación y suma de las entradas por los pesos que definen la red hasta llegar al final de esta para computar el error a la salida.

Red neuronal artificial

Una red neuronal artificial (RNA) o perceptrón multicapa está formada por múltiples capas, de tal manera que tiene capacidad para resolver problemas que no son linealmente separables, lo cual es la principal limitación del perceptrón (también llamado *perceptrón simple*). El perceptrón multicapa puede estar totalmente o localmente conectado.

Red neuronal convolucional

Una red neuronal convolucional es un tipo de red neuronal artificial en la que las neuronas corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria de un cerebro biológico. Este tipo de red es una variación de un perceptrón multicapa. Sin embargo, puesto que su aplicación se desarrolla en matrices bidimensionales, son muy efectivas para tareas de visión artificial, como en la clasificación y segmentación de imágenes, entre otras aplicaciones.

Red neuronal recurrente

Se trata de una red neuronal que integra bucles de realimentación, lo que permite que la información persista durante algunos pasos o épocas de entrenamiento a través de conexiones desde las salidas de las capas, que incrustan (*embedding*) sus resultados en los datos de entrada. Las conexiones entre nodos forman un gráfico dirigido a lo largo de una secuencia temporal. Funciona como una red con múltiples copias de sí misma, cada una con un mensaje a su sucesor. Se aplican en listas y datos temporales.

Regularización

Conjunto de técnicas empleadas para paliar el efecto indeseado del sobreajuste producido por dotar de mayor profundidad a una red neuronal, sobre todo cuando tenemos pocas muestras de entrenamiento (el número de muestras de entrada sería mucho menor que el número de parámetros usados por la red para ajustarse a esos escasos datos). Para acabar con este problema es precisamente para lo que se utilizan las técnicas de regularización.

Retropropagación

La propagación hacia atrás de errores o retropropagación (en inglés *backpropagation*) es un método de cálculo del gradiente utilizado en algoritmos de aprendizaje supervisado utilizados para entrenar redes neuronales artificiales. El método emplea un ciclo propagación-adaptación de dos fases. Una vez que se ha aplicado un patrón a la entrada de la red como estímulo, este se propaga desde la primera capa a través de las capas siguientes de la red, hasta generar una salida. La señal de salida se compara con la salida deseada y se calcula una señal de error para cada una de las salidas. Las salidas de error se propagan hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida.

Sobreajuste

En aprendizaje automático, el sobreajuste (*overfitting* en inglés) es el efecto de sobreentrenar un algoritmo de aprendizaje con unos ciertos datos para los que se conoce el resultado deseado. El algoritmo de aprendizaje debe alcanzar un estado en el que será capaz de predecir el resultado en otros casos a partir de lo aprendido con los datos de entrenamiento, generalizando para resolver situaciones distintas a las acaecidas durante este. Sin embargo, cuando un sistema se entrena demasiado (se sobreentrena) o se entrena con datos extraños, el algoritmo de aprendizaje puede quedar ajustado a unas características muy específicas de los datos de entrenamiento.

Stride

En redes neuronales convolucionales, se trata del paso que regula cada cuantos píxeles o elementos del mapa de activación se desliza el núcleo o *kernel* del filtro de una capa convolucional. Emplear un *stride* mayor a 1 supone una pérdida de información en el mapa de activación de salida.

Subajuste

El subajuste (en inglés *underfitting*) ocurre cuando un modelo estadístico no puede capturar adecuadamente la estructura subyacente de los datos. Un modelo *underfitted* es un modelo en el que faltan algunos parámetros o grados de libertad para su optimización mediante el proceso de entrenamiento. Se produciría un ajuste insuficiente, por ejemplo, al ajustar un modelo lineal a datos no lineales. Tal modelo tenderá a tener un pobre desempeño en producción.

Tasa de aprendizaje

Se emplea en los algoritmos por descenso de gradiente para cuantificar el paso en la búsqueda del mínimo global en la función de pérdidas o función objetivo. Si se elige una tasa de aprendizaje muy alta, el proceso puede divergir y no hallar los parámetros óptimos. Si la tasa es muy baja, el entrenamiento del modelo puede llevar mucho tiempo.

Trayectoria

En aprendizaje por refuerzo, la trayectoria se refiere a la secuencia de ejecución durante un número finito de *steps*, es decir, todos los estados, acciones y recompensas que han ido surgiendo a partir de la interacción agente-entorno durante este intervalo finito.



Enlaces de interés

CS231n Convolutional Neural Networks for Visual Recognition

Se trata de un repositorio con explicaciones teóricas y ejemplos prácticos de la mayoría de los conceptos que abarca la asignatura. Es muy recomendable la lectura de los recursos contenidos en los módulos 1 y 2 de este enlace.

<https://cs231n.github.io>

Playground TensorFlow

Este recurso interactivo sirve para comprender los fundamentos de las redes neuronales. Mediante una interfaz de usuario amigable, se pueden afianzar conceptos clave de la asignatura viendo cómo afectan de manera directa en el entrenamiento de una red neuronal empleando distintas distribuciones de datos de entrada.

<https://playground.tensorflow.org>

Pyimagesearch

Adrian Rosebrock, gurú del aprendizaje profundo, pone a nuestra disposición interesantes artículos semanales que versan sobre la temática principal de la asignatura. Gracias a este enlace de interés, el alumno tendrá la posibilidad de entender la puesta en marcha de algoritmos de aprendizaje profundo en problemas del mundo real.

<https://www.pyimagesearch.com/blog>

Spinning-up RL

Este repositorio educativo de contenidos relacionados con el aprendizaje profundo por refuerzo es obra de la empresa de inteligencia artificial Openai. Su contenido es muy variado e incluye artículos científicos, tutoriales y ejemplos de implementación de soluciones, entre otros.

<https://spinningup.openai.com>

Bibliografía



Arulkumaran, K., Deisenroth, M. P., Brundage, M., y Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34, 26-38. Recuperado de <https://doi.org/10.1109/MSP.2017.2743240>

Bahdanau, D., Cho, K. H., y Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. Recuperado de <https://arxiv.org/abs/1409.0473>

Barron, A. R. (1993). Universal Approximation Bounds for Superpositions of a Sigmoidal Function. *IEEE Transactions on Information Theory*, 39(3), 930-945.

Bengio, Y., y Grandvalet, Y. (2004). No Unbiased Estimator of the Variance of K-Fold Cross-Validation Yoshua Bengio Yves Grandvalet. *Journal of Machine Learning Research*, 5, 1089-1105. Recuperado de <http://www.jmlr.org/papers/volume5/grandvalet04a/grandvalet04a.pdf>

Bengio, Y., Simard, P., y Frasconi, P. (1994). Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2), 157-166. Recuperado de <https://doi.org/10.1109/72.279181>

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Nueva York: Springer-Verlag.

Blumer, A., Ehrenfeucht, A., Haussler, D., y Warmuth, M. K. (1989). Learnability and the Vapnik-Chervonenkis Dimension. *Journal of the Association for Computing Machinery* (36). Recuperado de <https://dl.acm.org/doi/10.1145/76359.76371>

Boureau, Y.-L., Le Roux, N., Bach, F., Ponce, J., y Lecun, Y. (2011). Ask the locals: multi-way local pooling for image recognition. *Proceedings of the IEEE International Conference on Computer Vision*. Recuperado de <https://nyuscholars.nyu.edu/en/publications/ask-the-locals-multi-way-local-pooling-for-image-recognition>

Boureau, Y.-L., Ponce, J., Fr, J. P., y Lecun, Y. (2010). A Theoretical Analysis of Feature Pooling in Visual Recognition. *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 111-118. Recuperado de <https://www.di.ens.fr/willow/pdfs/icml2010b.pdf>

Cai, M., Shi, Y., y Liu, J. (2013). Deep maxout neural networks for speech recognition. *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, 291-296. Recuperado de <https://doi.org/10.1109/ASRU.2013.6707745>

Chen, B., Ting, J.-A., Marlin, B., y De Freitas, N. (2010). Deep Learning of Invariant Spatio-Temporal Features from Video. Recuperado de <https://www.cs.ubc.ca/~nando/papers/nipsworkshop2010.pdf>

Cho, K., Merriënboer, B. van, Bahdanau, D., y Bengio, Y. (2014). On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *Proceedings of SSST-8*, 103-111. Recuperado de <https://doi.org/10.3115/V1/W14-4012>

- Chrupała, G., Alishahi, A., y Ni, A. (2015). *Learning language through pictures*. Recuperado de <https://arxiv.org/pdf/1506.03694.pdf>
- Chung, J., Gulcehre, C., Cho, K., y Bengio, Y. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. Recuperado de <http://arxiv.org/abs/1412.3555>
- Chung, J., Gulcehre, C., Cho, K., y Bengio, Y. (2015). Gated Feedback Recurrent Neural Networks. *32nd International Conference on Machine Learning*, 3, 2067-2075. Recuperado de <http://arxiv.org/abs/1502.02367>
- Collobert, R. (2011). Deep Learning for Efficient Discriminative Parsing. *Proceedings of Machine Learning Research*, 15, 224-232. Recuperado de <http://proceedings.mlr.press/v15/collobert11a/collobert11a.pdf>
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4), 303-314. Recuperado de <https://doi.org/10.1007/BF02551274>
- Fisher, R. A. (1936). The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7(2), 179-188. Recuperado de <https://doi.org/10.1111/j.1469-1809.1936.tb02137.x>
- Gers, F. A., Schmidhuber, J., y Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10), 2451-2471. Recuperado de <https://doi.org/10.1162/089976600300015015>
- Glorot, X., Bordes, A., y Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. *Proceedings of Machine Learning Research*, 15, 315-323. Recuperado de <http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf>
- Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., y Shet, V. (2013). *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*. Recuperado de [http://arxiv.org/abs/1312.6082](https://arxiv.org/abs/1312.6082)
- Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., y Bengio, Y. (2013). *An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks*. Recuperado de <http://arxiv.org/abs/1312.6211>
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., y Bengio, Y. (2013). Maxout Networks. *Proceedings of Machine Learning Research*, 28(3), 1319-1327. Recuperado de <http://proceedings.mlr.press/v28/goodfellow13.pdf>
- Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Heidelberg: Springer.
- Graves, A. (2013). *Generating Sequences With Recurrent Neural Networks*. Recuperado de <http://arxiv.org/abs/1308.0850>
- Graves, A., y Jaitly, N. (2014). Towards End-to-End Speech Recognition with Recurrent Neural Networks. *Proceedings of Machine Learning Research*, 32. Recuperado de <http://proceedings.mlr.press/v32/graves14.pdf>
- Graves, A., y Schmidhuber, J. (2009). Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. *Advances in Neural Information Processing Systems*, 21, 545-552. Recuperado de https://www.cs.toronto.edu/~graves/nips_2008.pdf

- Graves, A., Mohamed, A., y Hinton, G. (2013). Speech Recognition with Deep Recurrent Neural Networks. *International Conference on Acoustics, Speech and Signal Processing*. Recuperado de <http://arxiv.org/abs/1303.5778>
- Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., y Schmidhuber, J. (2015). LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10), 2222-2232. Recuperado de <https://doi.org/10.1109/TNNLS.2016.2582924>
- Hinton, G. et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6), 82-97. Recuperado de <https://doi.org/10.1109/MSP.2012.2205597>
- Hochreiter, S., Bengio, Y., Frasconi, P., y Schmidhuber, J. (2001). Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. En J. F. Kolen y S. C. Kremer (eds.), *A Field Guide to Dynamical Recurrent Networks*. Nueva York: IEEE Press.
- Hochreiter, S., y Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780. Recuperado de <https://doi.org/10.1162/neco.1997.9.8.1735>
- Hornik, K., Stinchcombe, M., y White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359-366. Recuperado de [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8)
- Hornik, K., Stinchcombe, M., y White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5), 551-560. Recuperado de [https://doi.org/10.1016/0893-6080\(90\)90005-6](https://doi.org/10.1016/0893-6080(90)90005-6)
- Hyy, H. (1996). Turing Machines are Recurrent Neural Networks. *Proceedings of STeP'96*, 13-24. Recuperado de <http://users.ics.aalto.fi/tho/stes/step96/hytyniemi1>
- Jaityl, N., y Hinton, E. S. (2013). Vocal Tract Length Perturbation (VTLN) improves speech recognition. Recuperado de <http://www.cs.toronto.edu/~ndjaityl/jaitly-icml13.pdf>
- Jarrett, K., Kavukcuoglu, K., Ranzato, A., y LeCun, Y. (2009). What is the Best Multi-Stage Architecture for Object Recognition? *Proceedings of the 12th International Conference on Computer Vision*, 2146-2153. Recuperado de <http://yann.lecun.com/exdb/publis/pdf/jarrett-iccv-09.pdf>
- Jia, Y., Huang, C., y Darrell, T. (2012). Beyond spatial pyramids: Receptive field learning for pooled image features. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 3370-3377. Recuperado de <https://doi.org/10.1109/CVPR.2012.6248076>
- Jozefowicz, R., y Zaremba, W. (2015). An Empirical Exploration of Recurrent Network Architectures. *Proceedings of Machine Learning Research*, 37. Recuperado de <http://proceedings.mlr.press/v37/jozefowicz15.pdf>
- Kaiming, H., Xiangyu, Z., Shaoqing, R., y Jian, S. (2018). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification Kaiming. *Biochemical and Biophysical Research Communications*, 498(1), 254-261. Recuperado de <https://doi.org/10.1016/j.bbrc.2018.01.076>
- Kiros, R., Salakhutdinov, R., y Zemel, R. S. (2014). Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models. Recuperado de <http://arxiv.org/abs/1411.2539>

- Konda, V. R., y Tsitsiklis, J. N. (2000). Actor-Critic Algorithms. *NIPS Conference*, 1008-1014. Recuperado de <https://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf>
- Kotzias, D., Denil, M., De Freitas, N., y Smyth, P. (2015). From group to individual labels using deep features. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 597-606. Recuperado de <https://doi.org/10.1145/2783258.2783380>
- Krizhevsky, A., Sutskever, I., y Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 25(2). Recuperado de <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- Lang, K. J., Waibel, A. H., y Hinton, G. E. (1990). A Time-Delay Neural Network Architecture for Isolated Word Recognition. *Neural Networks*, 3(1), 23-43. Recuperado de <http://www.cs.toronto.edu/~fritz/absps/langTDNN.pdf>
- Lecun, Y. (1989). Generalization and network design strategies. Recuperado de <http://citeserx.ist.psu.edu/viewdoc/download?doi=10.1.476.479&rep=rep1&type=pdf>
- Lecun, Y., Bottou, L., Bengio, Y., y Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11), 2278-2324. Recuperado de <http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>
- LeCun, Y., Kavukcuoglu, K., y Farabet, C. (2010). Convolutional networks and applications in vision. *International Symposium on Circuits and Systems*, 253-256. Recuperado de <https://doi.org/10.1109/ISCAS.2010.5537907>
- Lepenioti, K., Bousdekis, A., Apostolou, D., y Mentzas, G. (2020). Prescriptive analytics: Literature review and research challenges. *International Journal of Information Management*, 50, 57-70. Recuperado de <https://doi.org/10.1016/j.ijinfomgt.2019.04.003>
- Leshno, M., Lin, V. Y., Pinkus, A., y Schocken, S. (1993). Multilayer Feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6), 861-867. Recuperado de <http://www2.math.technion.ac.il/~pinkus/papers/neural.pdf>
- Luo, H., Carrier, P. L., Courville, A., y Bengio, Y. (2013). Texture Modeling with Convolutional Spike-and-Slab RBMs and Deep Extensions. *Proceedings of Machine Learning*, 31, 415-423. Recuperado de <http://proceedings.mlr.press/v31/luo13a.pdf>
- Maas, A. L., Hannun, A. Y., y Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. Recuperado de https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf
- Mitchell, T. M. (1997). *Machine Learning*. Nueva York: McGraw-Hill.
- Mnih, V. et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533. Recuperado de <https://doi.org/10.1038/nature14236>
- Mnih, V. et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. *Proceedings of Machine Learning Research*, 48, 2850-2869. Recuperado de <http://proceedings.mlr.press/v48/mniha16.pdf>
- Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. Ciudad: MIT Press

- Nair, V., y Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Recuperado de <https://www.cs.toronto.edu/~fritz/absps/reluICML.pdf>
- Pascanu, R., Gulcehre, C., Cho, K., y Bengio, Y. (2014). How to Construct Deep Recurrent Neural Networks. Recuperado de https://www.researchgate.net/publication/259399919_How_to_Construct_Deep_Recurrent_Neural_Networks/link/0046353178e5f2f103000000/download
- Poole, B., Sohl-Dickstein, J., y Ganguli, S. (2014). Analyzing noise in autoencoders and deep networks. Recuperado de <http://arxiv.org/abs/1406.1831>
- Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. Recuperado de <https://doi.org/10.1038/323533a0>
- Schulman, J., Moritz, P., Levine, S., Jordan, M. I., y Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. Recuperado de <https://arxiv.org/pdf/1506.02438.pdf>
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., y Klimov, O. (2017). Proximal Policy Optimization Algorithms. Recuperado de <http://arxiv.org/abs/1707.06347>
- Siegelmann, H. T. (1995). Computation beyond the turing limit. *Science*, 268(5210), 545-548. Recuperado de <https://doi.org/10.1126/science.268.5210.545>
- Siegelmann, H. T., y Sontag, E. D. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, 4(6), 77-80. Recuperado de [https://doi.org/10.1016/0893-9659\(91\)90080-F](https://doi.org/10.1016/0893-9659(91)90080-F)
- Siegelmann, H. T., y Sontag, E. D. (1995). On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1), 132-150. Recuperado de <https://doi.org/10.1006/jcss.1995.1013>
- Sietsma, J., y Dow, R. J. F. (1991). Creating artificial neural networks that generalize. *Neural Networks*, 4(1), 67-79. Recuperado de [https://doi.org/10.1016/0893-6080\(91\)90033-2](https://doi.org/10.1016/0893-6080(91)90033-2)
- Silver, D. et al. (2014). Deterministic Policy Gradient Algorithms. *Proceedings of Machine Learning Research*, 32. <http://proceedings.mlr.press/v32/silver14.pdf>
- Srivastava, N., Hinton, G., Krizhevsky, A., y Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56), 1929-1958. Recuperado de <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>
- Sutskever, I., Vinyals, O., y Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *Advances in Neural Information Processing Systems*, 4. Recuperado de <http://arxiv.org/abs/1409.3215>
- Sutton, R. S., y Barto, A. G. (1992). *Reinforcement Learning: An Introduction*. Massachusetts: MIT Press.
- Sutton, R. S., McAllester, D., Singh, S., y Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems*, 12, 1057-1063. Recuperado de <https://homes.cs.washington.edu/~todorov/courses/amath579/reading/PolicyGradient.pdf>

- Szegedy, C. et al. (2014). Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition*, 1-9. Recuperado de <https://static.googleusercontent.com/media/research.google.com/ca//pubs/archive/43022.pdf>
- Tang, Y., y Eliasmith, C. (2010). Deep networks for robust visual recognition. *Proceedings of the 27th International Conference on Machine Learning*. Recuperado de http://www.cs.toronto.edu/~tang/papers/deep_robust_rec.pdf
- Vapnik, V. (1982). *Estimation of Dependences Based on Empirical Data*. Berlín: Springer-Verlag.
- Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Nueva York: Springer.
- Vapnik, V. N., y Chervonenkis, A. Y. (1971). On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. En Vovk, V., Papadopoulos, H., y Gammerman, A. (eds.), *Theory of Probability and its Applications* (pp. 264-280). Cham: Springer. Recuperado de <https://doi.org/10.1137/1116025>
- Vincent, P., Larochelle, H., Bengio, Y., y Manzagol, P. A. (2008). Extracting and composing robust features with denoising autoencoders. *Proceedings of the 25th International Conference on Machine Learning*, 1096-1103. <https://doi.org/10.1145/1390156.1390294>
- Vinyals, O., Toshev, A., Bengio, S., y Erhan, D. (2014). Show and Tell: A Neural Image Caption Generator. *Conference on Computer Vision and Pattern Recognition*. Recuperado de <http://arxiv.org/abs/1411.4555>
- Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., y Lang, K. J. (1989). Phoneme Recognition Using Time-Delay Neural Networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3), 328-339. Recuperado de <https://doi.org/10.1109/29.21701>
- Watkins, C. J. C. H., y Dayan, P. (1992). Q-learning. *Machine Learning*, 8, 279-292. <https://doi.org/10.1007/bf00992698>
- Xu, K. et. al (2015). Show, attend and tell: Neural image caption generation with visual attention. *Proceedings of Machine Learning Research*, 37, 2048-2057.
- Zhou, Y. T., y Chellappa, R. (1988). Computation of optical flow using a neural network. *IEEE 1988 International Conference on Neural Networks*, 2, 71-78. Recuperado de <https://doi.org/10.1109/icnn.1988.23914>



Autores

Adrián Colomer Granero y Gabriel Enrique Muñoz Ríos