

CAPÍTULO 10. MODELADO DE SECUENCIAS: REDES RECURRENTES Y RECURSIVAS

Una vez que se obtienen los gradientes en los nodos internos del gráfico computacional, podemos obtener los gradientes en los nodos de los parámetros. Debido a que los parámetros se comparten a través de muchos pasos de tiempo, debemos tener cuidado al denotar operaciones de cálculo que involucren estas variables. Las ecuaciones que deseamos implementar usan el *backwardprop* método de sección 6.5.6, que calcula la contribución de un solo borde en el gráfico computacional al gradiente. Sin embargo, el ∇w operador utilizado en el cálculo tiene en cuenta la contribución de W al valor de L debido a todos los pasos en el gráfico computacional. Para resolver esta ambigüedad, introducimos variables ficticias W_t que se definen como copias de W pero con cada W_t usado solo en el paso de tiempo t . Entonces podemos usar ∇W_t para denotar la contribución de los pesos en el paso de tiempo t al gradiente.

Usando esta notación, el gradiente en los parámetros restantes viene dado por:

$$\nabla_C L = \frac{\partial \alpha(t)}{\partial C} \quad \nabla_{\alpha(t)} L = \frac{\nabla_{\alpha(t)} L}{t} \quad (10.22)$$

$$\nabla_b L = \frac{\partial h(t)}{\partial b(t)} \quad \nabla_{h(t)} L = \frac{1 - h(t)^2}{\text{diagnóstico}} \quad \nabla_{h(t)} L \quad (10.23)$$

$$\nabla_v L = \frac{\partial L}{\partial \alpha_i(t)} \quad \nabla_{v\alpha_i(t)} = \frac{(\nabla_{\alpha(t)} L)h_i(t)}{t} \quad (10.24)$$

$$\nabla_w L = \frac{\partial L}{\partial h_i(t)} \quad \nabla_{W_i(t)} h_i(t) \quad (10.25)$$

$$= \frac{1 - h_i(t)^2}{\text{diagnóstico}} (\nabla_{h(t)} L) h_{(t-1)} \quad (10.26)$$

$$\nabla_{tu} L = \frac{\partial L}{\partial h_i(t)} \quad \nabla_{tu(t)} h_i(t) \quad (10.27)$$

$$= \frac{1 - h_i(t)^2}{\text{diagnóstico}} (\nabla_{h(t)} L) X_i(t) \quad (10.28)$$

No necesitamos calcular el gradiente con respecto a X_i para el entrenamiento porque no tiene ningún parámetro como ancestros en el gráfico computacional que define la pérdida.

10.2.3 Redes recurrentes como modelos gráficos dirigidos

En el ejemplo de red recurrente que hemos desarrollado hasta ahora, las pérdidas $L_{(t)}$ fueron entropías cruzadas entre objetivos de entrenamiento $y_{(t)}$ y salidas $a_{(t)}$. Al igual que con una red feedforward, en principio es posible utilizar casi cualquier pérdida con una red recurrente. La pérdida debe elegirse en función de la tarea. Al igual que con una red feedforward, generalmente deseamos interpretar la salida de la RNN como una distribución de probabilidad y generalmente usamos la entropía cruzada asociada con esa distribución para definir la pérdida. El error cuadrático medio es la pérdida de entropía cruzada asociada con una distribución de salida que es una unidad gaussiana, por ejemplo, al igual que con una red de avance.

Cuando usamos un objetivo de entrenamiento predictivo de probabilidad logarítmica, como la ecuación 10.12, entrenamos la RNN para estimar la distribución condicional del siguiente elemento de secuencia $y_{(t)}$ dadas las entradas pasadas. Esto puede significar que maximizamos la probabilidad logarítmica

$$\text{registro} \text{pag}(y_{(t)} / X(1), \dots, X(t)), \quad (10.29)$$

o, si el modelo incluye conexiones desde la salida en un paso de tiempo al siguiente paso de tiempo,

$$\text{registro} \text{pag}(y_{(t)} / X(1), \dots, X(t), y(1), \dots, y(t-1)). \quad (10.30)$$

Descomponiendo la probabilidad conjunta sobre la secuencia de valores como una serie de predicciones probabilísticas de un paso es una forma de capturar la distribución conjunta completa en toda la secuencia. Cuando no nos alimentamos del pasado y valores como entradas que condicionan la predicción del siguiente paso, el modelo gráfico dirigido no contiene bordes de ningún $y_{(t)}$ en el pasado al presente $y_{(t)}$. En este caso, las salidas son condicionalmente independientes dada la secuencia de X valores. Cuando alimentamos al verdadero y valores (no su predicción, sino los valores reales observados o generados) de vuelta a la red, el modelo gráfico dirigido contiene bordes de todos $y_{(t)}$ valores del pasado al presente $y_{(t)}$ valor.

Como ejemplo simple, consideremos el caso en el que la RNN modela solo una secuencia de variables aleatorias escalares $Y = \{y(1), \dots, y(t)\}$, sin entradas adicionales x . La entrada en el paso de tiempo t es simplemente la salida en el paso de tiempo $t-1$. La RNN luego define un modelo gráfico dirigido sobre las variables y . Parametrizamos la distribución conjunta de estas observaciones usando la regla de la cadena (ecuación 3.6) para probabilidades condicionales:

$$PAG(Y) = PAG(y(1), \dots, y(t)) = \prod_{t=1}^{-\tau} PAG(y_{(t)} / y_{(t-1)}, y_{(t-2)}, \dots, y(1)) \quad (10.31)$$

donde el lado derecho de la barra está vacío para $t=1$, por supuesto. Por lo tanto, la log-verosimilitud negativa de un conjunto de valores $\{y(1), \dots, y(t)\}$ según tal modelo

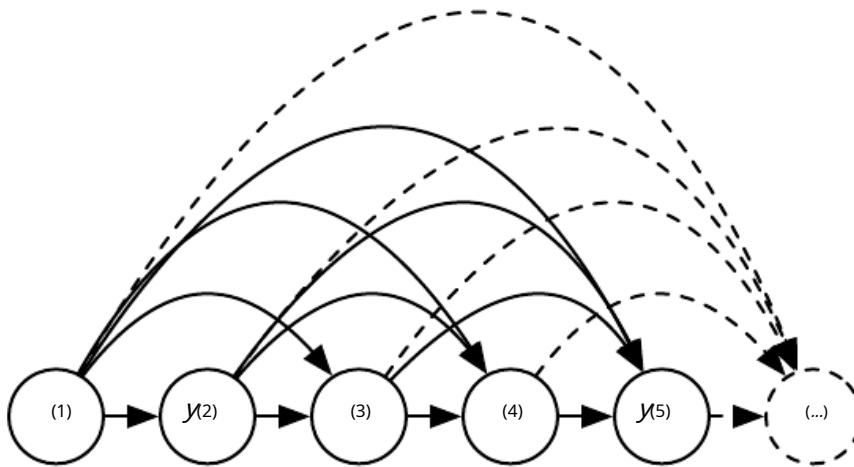


Figura 10.7: Modelo gráfico completamente conectado para una secuencia $y(1), y(2), \dots, y(t), \dots$: cada observación pasada $y(t)$ puede influir en la distribución condicional de algunos $y(t')$ (para $t' > t$), dados los valores anteriores. Parametrizando el modelo gráfico directamente de acuerdo con este gráfico (como en la ecuación 10.6) podría ser muy ineficiente, con un número cada vez mayor de entradas y parámetros para cada elemento de la secuencia. Los RNN obtienen la misma conectividad completa pero una parametrización eficiente, como se ilustra en la figura 10.8.

es

$$L = - \frac{1}{t} L(t) \quad (10.32)$$

dónde

$$L(t) = -\text{registroPAG}(y(t) | y(t-1), y(t-2), \dots, y(1)). \quad (10.33)$$

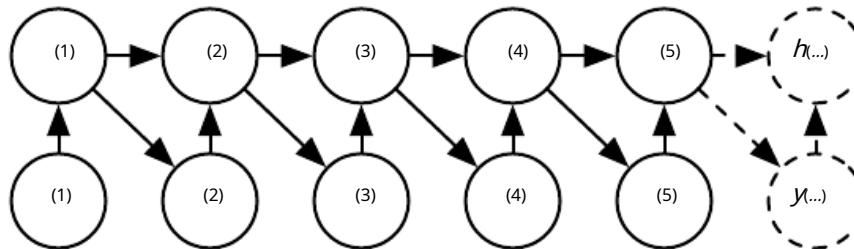


Figura 10.8: Introducir la variable de estado en el modelo gráfico de la RNN, aunque es una función determinista de sus entradas, ayuda a ver cómo podemos obtener una parametrización muy eficiente, basada en la ecuación 10.5. Cada etapa en la secuencia (por $h(t)$ y $y(t)$) involucra la misma estructura (el mismo número de entradas para cada nodo) y puede compartir los mismos parámetros con las otras etapas.

Los bordes en un modelo gráfico indican qué variables dependen directamente de otras variables. Muchos modelos gráficos tienen como objetivo lograr la eficiencia estadística y computacional al omitir los bordes que no corresponden a interacciones fuertes. Para

ejemplo, es común hacer la suposición de Markov de que el modelo gráfico solo debe contener bordes de $\{y_{(t-k)}, \dots, y_{(t-1)}\}$ en lugar de contener bordes de toda la historia pasada. Sin embargo, en algunos casos, creemos que todas las entradas pasadas deberían influir en el siguiente elemento de la secuencia. Los RNN son útiles cuando creemos que la distribución sobre $y_{(t)}$ puede depender de un valor de $y_{(t)}$ del pasado distante de una manera que no es capturada por el efecto de $y_{(t)}$ en $y_{(t-1)}$.

Una forma de interpretar una RNN como un modelo gráfico es considerar que la RNN define un modelo gráfico cuya estructura es el gráfico completo, capaz de representar dependencias directas entre cualquier par de valores de y . El modelo gráfico sobre los valores de y con la estructura gráfica completa se muestra en la figura 10.7. La interpretación gráfica completa del RNN se basa en ignorar las unidades ocultas $h_{(t)}$ al marginarlos del modelo.

Es más interesante considerar la estructura del modelo gráfico de RNN que resulta de considerar las unidades ocultas $h_{(t)}$ como variables aleatorias.¹ La inclusión de las unidades ocultas en el modelo gráfico revela que la RNN proporciona una parametrización muy eficiente de la distribución conjunta sobre las observaciones. Supongamos que representamos una distribución conjunta arbitraria sobre valores discretos con una representación tabular: una matriz que contiene una entrada separada para cada posible asignación de valores, con el valor de esa entrada dando la probabilidad de que ocurra esa asignación. Si y puede asumir k diferentes valores, la representación tabular tendría $O(k^T)$ parámetros. En comparación, debido al uso compartido de parámetros, el número de parámetros en el RNN es $O(1)$ en función de la longitud de la secuencia. El número de parámetros en la RNN puede ajustarse para controlar la capacidad del modelo, pero no está obligado a escalar con la longitud de la secuencia. Ecuación 10.5 muestra que la RNN parametriza relaciones de largo plazo entre variables de manera eficiente, utilizando aplicaciones recurrentes de la misma función f y los mismos parámetros θ en cada paso de tiempo. Cifra 10.8 ilustra la interpretación del modelo gráfico. Incorporando el $h_{(t)}$ nodos en el modelo gráfico desacopla el pasado y el futuro, actuando como una cantidad intermedia entre ellos. Una variable $y_{(t)}$ en el pasado lejano puede influir en una variable $y_{(t)}$ a través de su efecto sobre h . La estructura de este gráfico muestra que el modelo se puede parametrizar de manera eficiente utilizando las mismas distribuciones de probabilidad condicional en cada paso de tiempo, y que cuando se observan todas las variables, la probabilidad de la asignación conjunta de todas las variables se puede evaluar de manera eficiente.

Incluso con la parametrización eficiente del modelo gráfico, algunas operaciones siguen siendo un desafío computacional. Por ejemplo, es difícil predecir la falta

¹La distribución condicional sobre estas variables dados sus padres es determinista. Esto es perfectamente legítimo, aunque es algo raro diseñar un modelo gráfico con unidades ocultas tan deterministas.

valores en el medio de la secuencia.

El precio que pagan las redes recurrentes por su reducido número de parámetros es que *optimizandolos* parámetros pueden ser difíciles.

El uso compartido de parámetros que se utiliza en las redes recurrentes se basa en la suposición de que los mismos parámetros se pueden utilizar para diferentes períodos de tiempo. De manera equivalente, la suposición es que la distribución de probabilidad condicional sobre las variables en el tiempo $t+1$ dadas las variables en el tiempo t es **estacionario**, lo que significa que la relación entre el paso de tiempo anterior y el paso de tiempo siguiente no depende de t . En principio, sería posible utilizar t como una entrada adicional en cada paso de tiempo y permita que el alumno descubra cualquier dependencia del tiempo mientras comparte todo lo que pueda entre diferentes pasos de tiempo. Esto ya sería mucho mejor que usar una distribución de probabilidad condicional diferente para cada t , pero la red tendría entonces que extrapolar frente a nuevos valores de t .

Para completar nuestra visión de una RNN como modelo gráfico, debemos describir cómo extraer muestras del modelo. La operación principal que debemos realizar es simplemente tomar muestras de la distribución condicional en cada paso de tiempo. Sin embargo, hay una complicación adicional. La RNN debe tener algún mecanismo para determinar la longitud de la secuencia. Esto se puede lograr de varias maneras.

En el caso de que la salida sea un símbolo tomado de un vocabulario, se puede agregar un símbolo especial correspondiente al final de una secuencia ([Schmidhuber, 2012](#)). Cuando se genera ese símbolo, el proceso de muestreo se detiene. En el conjunto de entrenamiento, insertamos este símbolo como un miembro adicional de la secuencia, inmediatamente después de $X(t)$ en cada ejemplo de entrenamiento.

Otra opción es introducir una salida adicional de Bernoulli en el modelo que represente la decisión de continuar o detener la generación en cada paso de tiempo. Este enfoque es más general que el enfoque de agregar un símbolo adicional al vocabulario, porque puede aplicarse a cualquier RNN, en lugar de solo a los RNN que generan una secuencia de símbolos. Por ejemplo, se puede aplicar a un RNN que emite una secuencia de números reales. La nueva unidad de salida suele ser una unidad sigmoidea entrenada con la pérdida de entropía cruzada. En este enfoque, el sigmoide se entrena para maximizar la probabilidad logarítmica de la predicción correcta en cuanto a si la secuencia termina o continúa en cada paso de tiempo.

Otra forma de determinar la longitud de la secuencia τ es agregar una salida adicional al modelo que predice el número entero τ sí mismo. El modelo puede muestrear un valor de τ y luego muestra τ pasos por valor de datos. Este enfoque requiere agregar una entrada adicional a la actualización recurrente en cada paso de tiempo para que la actualización recurrente sepa si está cerca del final de la secuencia generada. Esta entrada adicional puede consistir en el valor de τ o puede consistir en $\tau - t$, el número de restantes

pasos de tiempo. Sin esta entrada adicional, la RNN podría generar secuencias que terminan abruptamente, como una oración que termina antes de completarse. Este enfoque se basa en la descomposición

$$PAG(X(1), \dots, X(t)) = PAG(t)PAG(X(1), \dots, X(t)/t). \quad (10.34)$$

La estrategia de predecir t directamente es utilizado por ejemplo por [Buen compañero et al.](#) (2014d).

10.2.4 Modelado de secuencias condicionadas por contexto con RNN

En la sección anterior describimos cómo un RNN podría corresponder a un modelo gráfico dirigido sobre una secuencia de variables aleatorias y_t sin entradas X . Por supuesto, nuestro desarrollo de RNN como en la ecuación 10.8 incluye una secuencia de entradas $X(1), X(2), \dots, X(t)$. En general, los RNN permiten la extensión de la vista del modelo gráfico para representar no solo una distribución conjunta sobre las variables sino también una distribución condicional sobre y dado X . Como se discutió en el contexto de las redes feedforward en la sección 6.2.1.1, cualquier modelo que represente una variable $PAG(y; \theta)$ se puede reinterpretar como un modelo que representa una distribución condicional $PAG(y/\omega)$ con $\omega = \theta$. Podemos extender dicho modelo para representar una distribución $PAG(y/X)$ usando el mismo $PAG(y/\omega)$ como antes, pero haciendo ω una función de X . En el caso de una RNN, esto se puede lograr de diferentes maneras. Revisamos aquí las opciones más comunes y obvias.

Previamente, hemos discutido RNN que toman una secuencia de vectores X_t para $t=1, \dots, T$ como entrada. Otra opción es tomar un solo vector X como entrada. Cuando X es un vector de tamaño fijo, simplemente podemos convertirlo en una entrada adicional del RNN que genera el y secuencia. Algunas formas comunes de proporcionar una entrada adicional a un RNN son:

1. como una entrada adicional en cada paso de tiempo, o
2. como el estado inicial $h(0)$, o
3. ambos.

El primer y más común enfoque se ilustra en la figura 10.9. La interacción entre la entrada X y cada vector unitario oculto h_t está parametrizado por una matriz de peso recientemente introducida R que estaba ausente del modelo de sólo la secuencia de y valores. El mismo producto $X \cdot R$ se agrega como entrada adicional a las unidades ocultas en cada paso de tiempo. Podemos pensar en la elección de X como determinar el valor

de X - R que es efectivamente un nuevo parámetro de sesgo utilizado para cada una de las unidades ocultas. Los pesos permanecen independientes de la entrada. Podemos pensar en este modelo tomando los parámetros θ del modelo no condicional y convertirlos en ω , donde los parámetros de sesgo dentro de ω son ahora una función de la entrada.

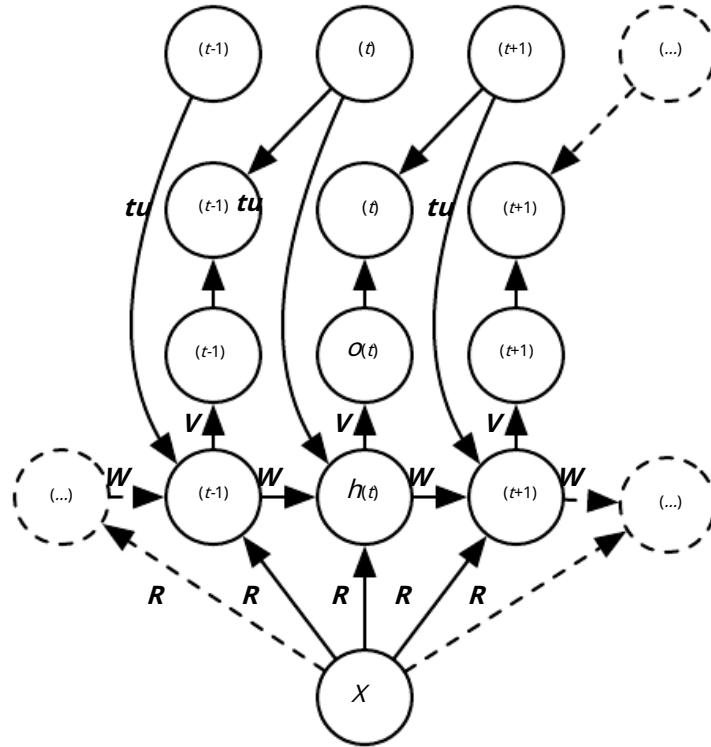


Figura 10.9: Una RNN que mapea un vector de longitud fija X en una distribución sobre secuencias. Este RNN es apropiado para tareas tales como subtítulos de imágenes, donde una sola imagen se usa como entrada para un modelo que luego produce una secuencia de palabras que describen la imagen. cada elemento y_t de la secuencia de salida observada sirve como entrada (para el paso de tiempo actual) y, durante el entrenamiento, como objetivo (para el paso de tiempo anterior).

En lugar de recibir un solo vector X como entrada, la RNN puede recibir una secuencia de vectores X_t como entrada. El RNN descrito en la ecuación 10.8 corresponde a una distribución condicional $PAG(y_1, \dots, y_t | X_1, \dots, X_t)$ que hace una suposición de independencia condicional de que esta distribución se factoriza como

$$\underset{t}{PAG}(y_t | X_1, \dots, X_t). \quad (10.35)$$

Para eliminar la suposición de independencia condicional, podemos agregar conexiones de la salida en el momento t a la unidad oculta en el momento $t+1$, como se muestra en la figura 10.10. El modelo puede entonces representar distribuciones de probabilidad arbitrarias sobre el y secuencia. Este tipo de modelo que representa una distribución sobre una secuencia dada otra

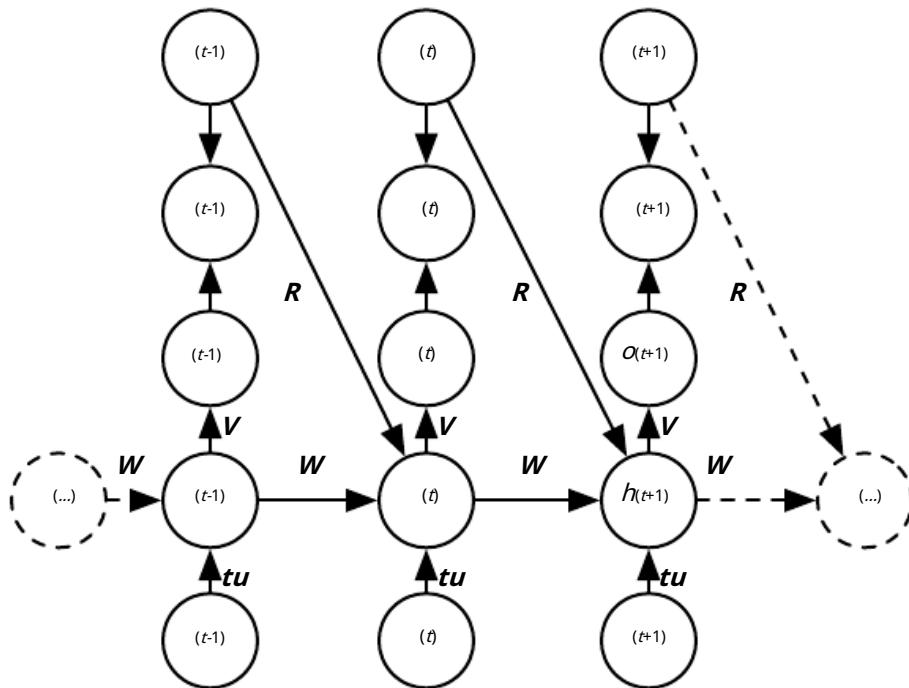


Figura 10.10: Una red neuronal recurrente condicional mapeando una secuencia de longitud variable de x valores en una distribución sobre secuencias de y valores de la misma longitud. Comparado con la figura 10.3, este RNN contiene conexiones de la salida anterior al estado actual. Estas conexiones permiten que esta RNN modele una distribución arbitraria sobre secuencias de y secuencias dadas de x de la misma longitud. El RNN de la figura 10.3 sólo es capaz de representar distribuciones en las que los y valores son condicionalmente independientes entre sí dada la x valores.

secuencia todavía tiene una restricción, que es que la longitud de ambas secuencias debe ser la misma. Describimos cómo eliminar esta restricción en la sección 10.4.

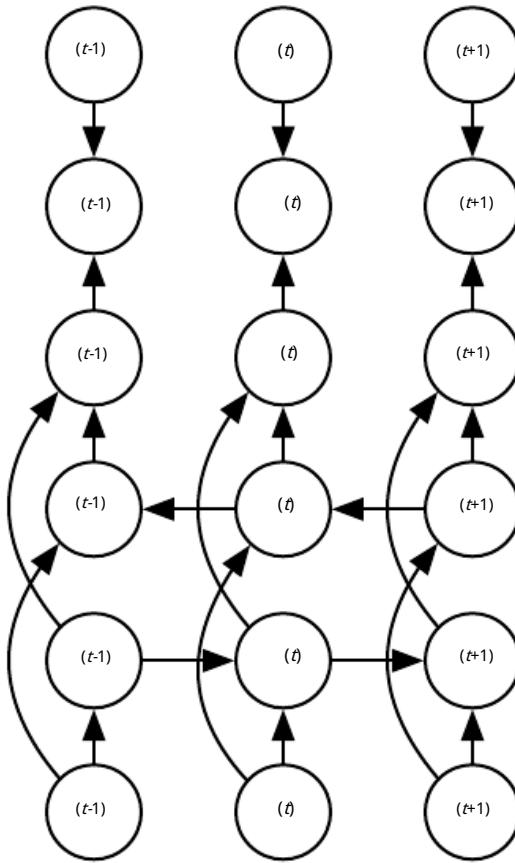


Figura 10.11: Cálculo de una red neuronal recurrente bidireccional típica, destinada a aprender a mapear secuencias de entrada x_t a las secuencias objetivo y_t , con pérdida L_t en cada paso t . El *h*la recurrencia propaga la información hacia adelante en el tiempo (hacia la derecha) mientras que la *gramola* recurrencia propaga la información hacia atrás en el tiempo (hacia la izquierda). Así en cada punto t , las unidades de salida α_t puede beneficiarse de un resumen relevante del pasado en su h_t entrada y de un resumen relevante del futuro en su $gramola_t$ aporte.

10.3 RNN bidireccionales

Todas las redes recurrentes que hemos considerado hasta ahora tienen una estructura "causal", lo que significa que el estado en el momento t solo captura información del pasado, x_1, \dots, x_{t-1} , y la entrada actual x_t . Algunos de los modelos que hemos discutido también permiten información del pasado y valores para afectar el estado actual cuando el y los valores están disponibles.

Sin embargo, en muchas aplicaciones queremos generar una predicción de y_t cuál podría

depender de *toda la secuencia de entrada*. Por ejemplo, en el reconocimiento de voz, la interpretación correcta del sonido actual como fonema puede depender de los siguientes fonemas debido a la coarticulación y, potencialmente, incluso puede depender de las siguientes palabras debido a las dependencias lingüísticas entre las palabras cercanas: si hay dos interpretaciones de la palabra actual que son acústicamente plausibles, es posible que tengamos que mirar hacia el futuro (y el pasado) para eliminar la ambigüedad. Esto también se aplica al reconocimiento de escritura a mano y muchas otras tareas de aprendizaje de secuencia a secuencia, que se describen en la siguiente sección.

Las redes neuronales recurrentes bidireccionales (o RNN bidireccionales) se inventaron para abordar esa necesidad ([Schuster y Paliwal, 1997](#)). Han tenido mucho éxito ([Tumbas, 2012](#)) en aplicaciones donde surge esa necesidad, como el reconocimiento de escritura a mano ([Tumbaset et al., 2008; Graves y Schmidhuber, 2009](#)), reconocimiento de voz ([Graves y Schmidhuber, 2005; Tumbaset et al., 2013](#)) y bioinformática ([Baldí et al., 1999](#)).

Como sugiere el nombre, las RNN bidireccionales combinan una RNN que avanza en el tiempo comenzando desde el inicio de la secuencia con otra RNN que retrocede en el tiempo comenzando desde el final de la secuencia. Cifra [10.11](#) ilustra el típico RNN bidireccional, con h_t representando el estado de la sub-RNN que avanza a través del tiempo $y_{grama(t)}$ representando el estado de la sub-RNN que retrocede en el tiempo. Esto permite que las unidades de salida o_t para calcular una representación que depende de tanto el pasado como el futuro pero es más sensible a los valores de entrada alrededor del tiempo t , sin tener que especificar una ventana de tamaño fijo alrededor t (como se tendría que hacer con una red feedforward, una red convolucional o una RNN regular con un búfer de anticipación de tamaño fijo).

Esta idea se puede extender naturalmente a la entrada bidimensional, como imágenes, al tener cuatro RNNs, cada uno yendo en una de las cuatro direcciones: arriba, abajo, izquierda, derecha. En cada punto (y_i, j) de una cuadrícula 2-D, una salida $O_{y_i, j}$ luego podría calcular una representación que capturaría principalmente información local pero también podría depender de entradas de largo alcance, si la RNN puede aprender a transportar esa información. En comparación con una red convolucional, las RNN aplicadas a imágenes suelen ser más costosas, pero permiten interacciones laterales de largo alcance entre entidades en el mismo mapa de características ([Vista et al., 2015; Kalchbrenner et al., 2015](#)). De hecho, las ecuaciones de propagación hacia adelante para tales RNN se pueden escribir en una forma que muestre que usan una convolución que calcula la entrada de abajo hacia arriba para cada capa, antes de la propagación recurrente a través del mapa de características que incorpora las interacciones laterales.

10.4 Arquitecturas de codificador-decodificador de secuencia a secuencia

hemos visto en la figura 10.5 cómo un RNN puede asignar una secuencia de entrada a un vector de tamaño fijo. hemos visto en la figura 10.9 cómo un RNN puede mapear un vector de tamaño fijo a una secuencia. hemos visto en cifras 10.3, 10.4, 10.10 y 10.11 cómo un RNN puede mapear una secuencia de entrada a una secuencia de salida de la misma longitud.

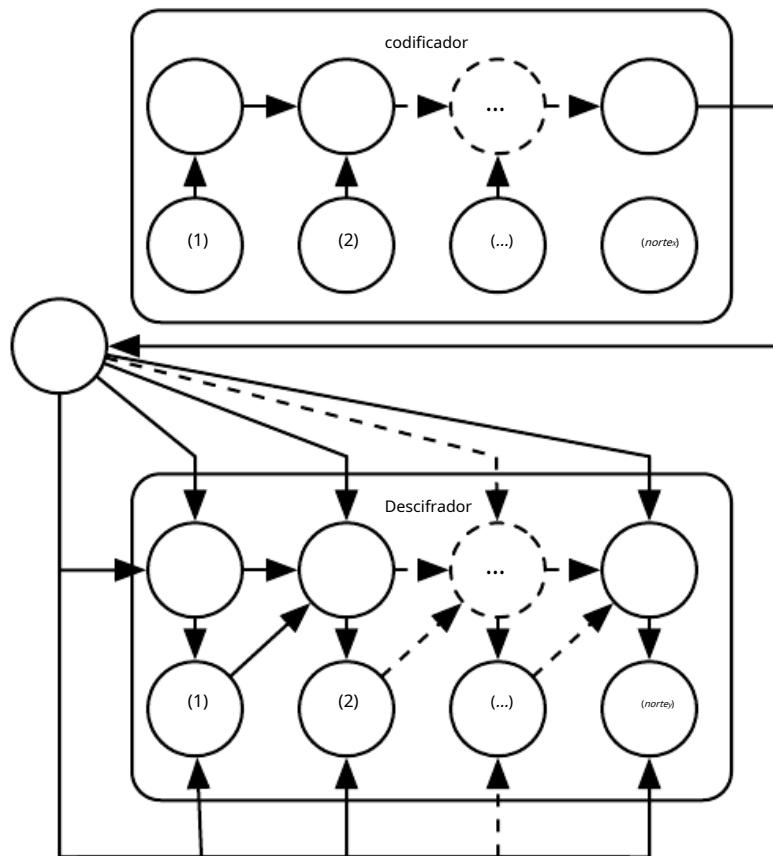


Figura 10.12: Ejemplo de una arquitectura RNN codificador-decodificador o secuencia a secuencia, para aprender a generar una secuencia de salida ($y_{(1)}, \dots, y_{(norte)}$) dada una secuencia de entrada ($X_{(1)}, X_{(2)}, \dots, X_{(norte)}$). Se compone de un codificador RNN que lee la secuencia de entrada y un decodificador RNN que genera la secuencia de salida (o calcula la probabilidad de una secuencia de salida dada). El estado oculto final del codificador RNN se usa para calcular una variable de contexto generalmente de tamaño fijo C que representa un resumen semántico de la secuencia de entrada y se da como entrada al decodificador RNN.

Aquí discutimos cómo se puede entrenar un RNN para mapear una secuencia de entrada a una secuencia de salida que no es necesariamente de la misma longitud. Esto surge en muchas aplicaciones, como reconocimiento de voz, traducción automática o preguntas.

respondiendo, donde las secuencias de entrada y salida en el conjunto de entrenamiento generalmente no tienen la misma longitud (aunque sus longitudes pueden estar relacionadas).

A menudo llamamos a la entrada de la RNN el "contexto". Queremos producir una representación de este contexto, C . El contexto C podría ser un vector o secuencia de vectores que resumen la secuencia de entrada $X = (X(1), \dots, X(n_{\text{tex}}))$.

La arquitectura RNN más simple para mapear una secuencia de longitud variable a otra secuencia de longitud variable fue propuesta por primera vez por Cho et al. (2014a) y poco después por Sutskever et al. (2014), quienes desarrollaron de forma independiente esa arquitectura y fueron los primeros en obtener una traducción de vanguardia utilizando este enfoque. El primer sistema se basa en la calificación de propuestas generadas por otro sistema de traducción automática, mientras que el segundo utiliza una red recurrente independiente para generar las traducciones. Estos autores denominaron respectivamente a esta arquitectura, ilustrada en la figura 10.12, la arquitectura codificador-decodificador o secuencia a secuencia. La idea es muy simple: (1) una **codificador o lector o aporte** RNN procesa la secuencia de entrada. El codificador emite el contexto C , generalmente como una función simple de su estado oculto final. (2) un **descifrador o escritor o producción** RNN está condicionado a ese vector de longitud fija (al igual que en la figura 10.9) para generar la secuencia de salida $Y = (y(1), \dots, y(n_{\text{tore}}))$. La innovación de este tipo de arquitectura sobre las presentadas en secciones anteriores de este capítulo es que las longitudes n_{tex} y n_{tore} pueden variar entre sí, mientras que las arquitecturas anteriores restringieron $n_{\text{tex}} = n_{\text{tore}} = \tau$. En una arquitectura de secuencia a secuencia, los dos RNN se entran conjuntamente para maximizar el promedio de registro $PAG(y(1), \dots, y(n_{\text{tore}})) / X(1), \dots, X(n_{\text{tex}})$ sobre todos los pares de X y y secuencias en el conjunto de entrenamiento. El último estado $h_{n_{\text{tex}}}$ del codificador RNN se usa típicamente como una representación C de la secuencia de entrada que se proporciona como entrada al decodificador RNN.

Si el contexto C es un vector, entonces el decodificador RNN es simplemente un RNN de vector a secuencia como se describe en la sección 10.2.4. Como hemos visto, hay al menos dos formas para que un RNN de vector a secuencia reciba entrada. La entrada se puede proporcionar como el estado inicial de la RNN, o la entrada se puede conectar a las unidades ocultas en cada paso de tiempo. Estas dos formas también se pueden combinar.

No existe la restricción de que el codificador deba tener el mismo tamaño de capa oculta que el decodificador.

Una clara limitación de esta arquitectura es cuando el contexto C la salida del codificador RNN tiene una dimensión que es demasiado pequeña para resumir correctamente una secuencia larga. Este fenómeno fue observado por Bahdanau et al. (2015) en el contexto de la traducción automática. Propusieron hacer C una secuencia de longitud variable en lugar de un vector de tamaño fijo. Además, introdujeron un **mecanismo de atención** que aprende a asociar elementos de la secuencia C a los elementos de la salida.

secuencia. Mira la sección 12.4.5.1 para más detalles.

10.5 Redes recurrentes profundas

El cálculo en la mayoría de las RNN se puede descomponer en tres bloques de parámetros y transformaciones asociadas:

1. de la entrada al estado oculto,
2. del estado oculto anterior al siguiente estado oculto, y
3. del estado oculto a la salida.

Con la arquitectura RNN de figura 10.3, cada uno de estos tres bloques está asociado a una única matriz de pesos. En otras palabras, cuando se despliega la red, cada uno de estos corresponde a una transformación superficial. Por una transformación superficial, nos referimos a una transformación que estaría representada por una sola capa dentro de un MLP profundo. Por lo general, esta es una transformación representada por una transformación afín aprendida seguida de una no linealidad fija.

¿Sería ventajoso introducir profundidad en cada una de estas operaciones? Evidencia experimental (Tumbaset *et al.*, 2013; Pascanuet *et al.*, 2014a) lo sugiere fuertemente. La evidencia experimental está de acuerdo con la idea de que necesitamos suficiente profundidad para realizar los mapeos requeridos. Ver también Schmidhuber (1992), El Hihi y Bengio (1996), o Jaeger (2007a) para trabajos anteriores sobre RNN profundos.

Tumbaset *et al.* (2013) fueron los primeros en mostrar un beneficio significativo de descomponer el estado de un RNN en múltiples capas como en la figura 10.13 (izquierda). Podemos pensar en las capas inferiores de la jerarquía representada en la figura 10.13a como jugando un papel en la transformación de la entrada en bruto en una representación que es más apropiada, en los niveles más altos del estado oculto. Pascanuet *et al.* (2014a) van un paso más allá y proponen tener un MLP separado (posiblemente profundo) para cada uno de los tres bloques enumerados anteriormente, como se ilustra en la figura 10.13b. Las consideraciones de la capacidad de representación sugieren asignar suficiente capacidad en cada uno de estos tres pasos, pero hacerlo agregando profundidad puede perjudicar el aprendizaje al dificultar la optimización. En general, es más fácil optimizar arquitecturas menos profundas y agregar la profundidad adicional de la figura 10.13b hace el camino más corto desde una variable en el paso de tiempo t a una variable en el paso de tiempo $t+1$ volverse más largo. Por ejemplo, si se usa un MLP con una sola capa oculta para la transición de estado a estado, hemos duplicado la longitud del camino más corto entre variables en dos pasos de tiempo diferentes, en comparación con el RNN ordinario de la figura 10.3. Sin embargo, como sostiene Pascanuet *et al.* (2014a), este

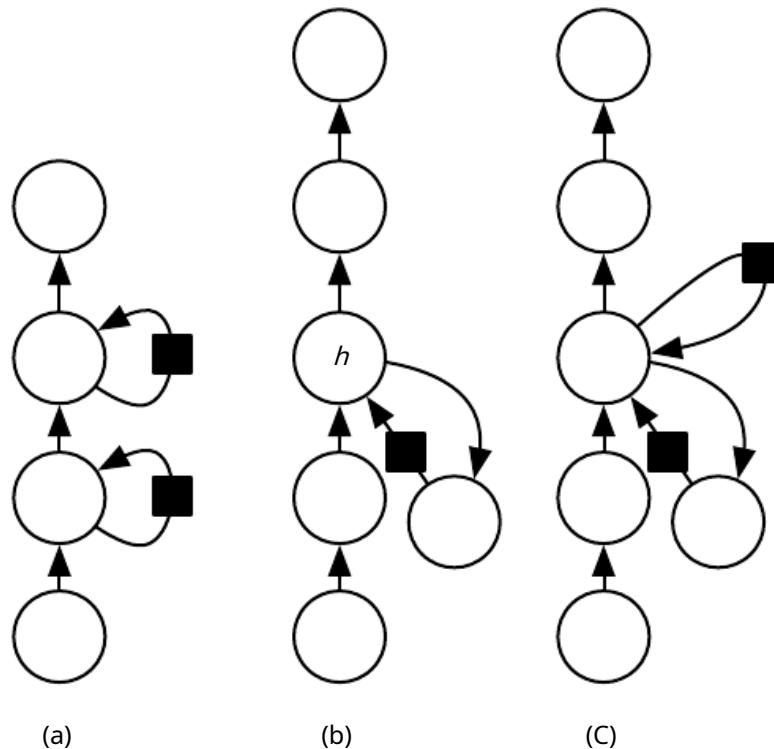


Figura 10.13: Una red neuronal recurrente se puede profundizar de muchas maneras (Pascanu *et al.*, 2014a). (a) El estado recurrente oculto se puede dividir en grupos organizados jerárquicamente. (b) Se puede introducir un cálculo más profundo (p. ej., un MLP) en las partes de entrada a oculto, de oculto a oculto y de oculto a salida. Esto puede alargar el camino más corto que une diferentes pasos de tiempo. (C) El efecto de alargamiento de la ruta se puede mitigar mediante la introducción de conexiones de salto.

se puede mitigar mediante la introducción de conexiones de salto en la ruta de oculto a oculto, como se ilustra en la figura 10.13C.

10.6 Redes neuronales recursivas

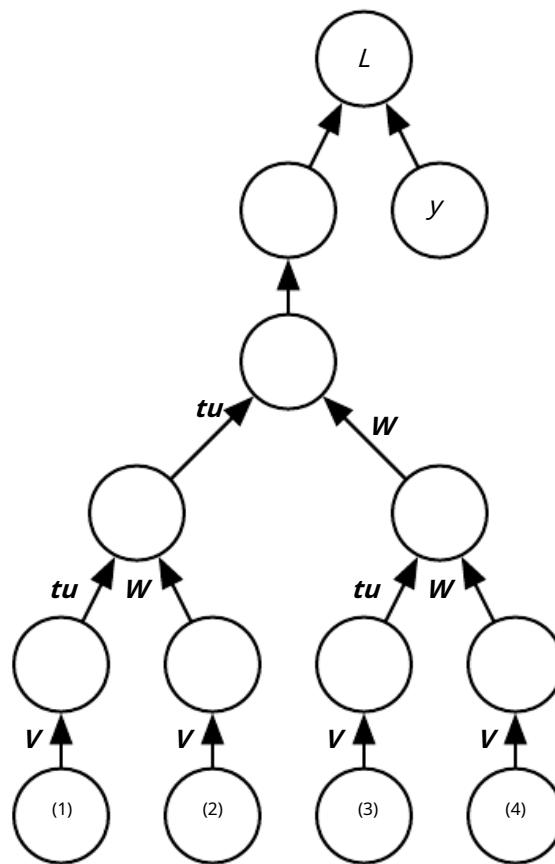


Figura 10.14: Una red recursiva tiene un gráfico computacional que generaliza el de la red recurrente de una cadena a un árbol. Una secuencia de tamaño variable $X_{(1)}, X_{(2)}, \dots, X_{(t)}$ se puede asignar a una representación de tamaño fijo (la salida o), con un conjunto fijo de parámetros (las matrices de peso t_u, V, W). La figura ilustra un caso de aprendizaje supervisado en el que algunos objetivos y se proporciona que está asociado con toda la secuencia.

Redes neuronales recursivas² representan otra generalización más de las redes recurrentes, con un tipo diferente de gráfico computacional, que está estructurado como un árbol profundo, en lugar de la estructura en forma de cadena de las RNN. El gráfico computacional típico para una red recursiva se ilustra en la figura 10.14. Neural recursivo

²Sugerimos no abreviar "red neuronal recursiva" como "RNN" para evitar confusiones con "red neuronal recurrente".

Las redes fueron introducidas por [abadejo\(1990\)](#) y su uso potencial para aprender a razonar fue descrito por [fondo\(2011\)](#). Las redes recursivas se han aplicado con éxito al procesamiento *estructuras de datos* como entrada a las redes neuronales ([Frasconi et al., 1997, 1998](#)), en procesamiento de lenguaje natural ([Socher et al., 2011a,C,2013a](#)) así como en visión artificial ([Socher et al., 2011b](#)).

Una clara ventaja de las redes recursivas sobre las redes recurrentes es que para una secuencia de la misma longitud t , la profundidad (medida como el número de composiciones de operaciones no lineales) se puede reducir drásticamente de $\tau O(\text{registro } t)$, que podría ayudar a lidiar con las dependencias a largo plazo. Una pregunta abierta es cómo estructurar mejor el árbol. Una opción es tener una estructura de árbol que no dependa de los datos, como un árbol binario equilibrado. En algunos dominios de aplicación, los métodos externos pueden sugerir la estructura de árbol adecuada. Por ejemplo, cuando se procesan oraciones en lenguaje natural, la estructura de árbol para la red recursiva se puede fijar a la estructura del árbol de análisis de la oración proporcionada por un analizador de lenguaje natural ([Socher et al., 2011a,2013a](#)). Idealmente, a uno le gustaría que el alumno mismo descubra e infiera la estructura de árbol que es apropiada para cualquier entrada dada, como lo sugiere [fondo\(2011\)](#).

Son posibles muchas variantes de la idea de red recursiva. Por ejemplo, [Frasconi et al.\(1997\)](#) y [Frasconi et al.\(1998\)](#) asocian los datos con una estructura de árbol y asocian las entradas y los objetivos con nodos individuales del árbol. El cálculo realizado por cada nodo no tiene que ser el cálculo tradicional de neuronas artificiales (transformación afín de todas las entradas seguida de una no linealidad monótona). Por ejemplo, [Socher et al.\(2013a\)](#) proponen el uso de operaciones tensoriales y formas bilineales, que previamente han resultado útiles para modelar relaciones entre conceptos ([Weston et al., 2010;Bordes et al., 2012](#)) cuando los conceptos están representados por vectores continuos (incrustaciones).

10.7 El desafío de las dependencias a largo plazo

El desafío matemático de aprender dependencias a largo plazo en redes recurrentes se introdujo en la sección [8.2.5](#). El problema básico es que los gradientes que se propagan en muchas etapas tienden a desaparecer (la mayoría de las veces) o explotar (rara vez, pero con mucho daño a la optimización). Incluso si asumimos que los parámetros son tales que la red recurrente es estable (puede almacenar recuerdos, con gradientes que no explotan), la dificultad con las dependencias a largo plazo surge de los pesos exponencialmente más pequeños que se dan a las interacciones a largo plazo (que implican la multiplicación de muchos jacobianos) en comparación con los de corto plazo. Muchas otras fuentes proporcionan un tratamiento más profundo ([Hochreiter, 1991;Doya, 1993;bengio et al.](#),

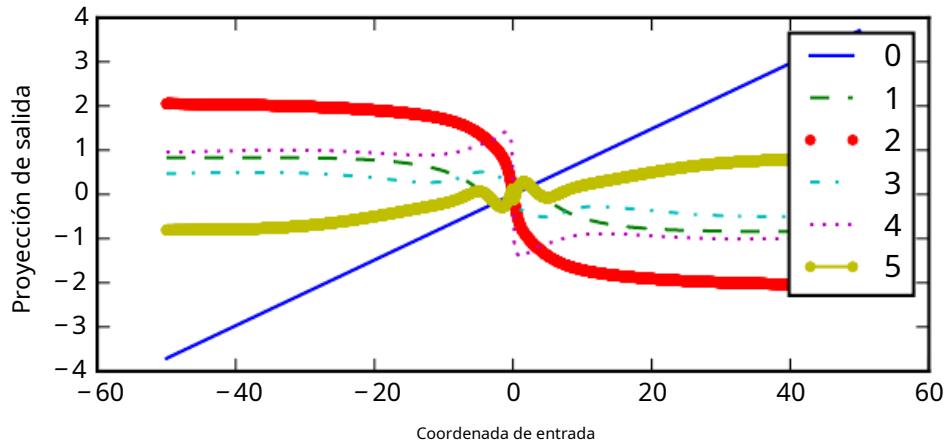


Figura 10.15: Al componer muchas funciones no lineales (como la lineal-bronceadocapa que se muestra aquí), el resultado es altamente no lineal, típicamente con la mayoría de los valores asociados con una pequeña derivada, algunos valores con una gran derivada y muchas alternancias entre crecientes y decrecientes. En este gráfico, trazamos una proyección lineal de un estado oculto de 100 dimensiones hasta una sola dimensión, trazada en el eje. El eje X es la coordenada del estado inicial a lo largo de una dirección aleatoria en el espacio de 100 dimensiones. Por lo tanto, podemos ver esta gráfica como una sección transversal lineal de una función de alta dimensión. Los gráficos muestran la función después de cada paso de tiempo, o de manera equivalente, después de cada número de veces que se ha compuesto la función de transición.

1994; Pascanu et al., 2013). En esta sección, describimos el problema con más detalle. Las secciones restantes describen enfoques para superar el problema.

Las redes recurrentes implican la composición de la misma función varias veces, una vez por paso de tiempo. Estas composiciones pueden resultar en un comportamiento extremadamente no lineal, como se ilustra en la figura 10.15.

En particular, la composición de funciones empleada por las redes neuronales recurrentes se parece un poco a la multiplicación de matrices. Podemos pensar en la relación de recurrencia

$$h(t) = W \cdot h(t-1) \quad (10.36)$$

como una red neuronal recurrente muy simple que carece de una función de activación no lineal y que carece de entradas X . Como se describe en la sección 8.2.5, esta relación de recurrencia describe esencialmente el método de potencia. Puede simplificarse a

$$h(t) = W_t^{-1} \cdot -h(0), \quad (10.37)$$

y si W admite una descomposición propia de la forma

$$W = q \Lambda q^-, \quad (10.38)$$

con ortogonal q , la recurrencia puede simplificarse aún más para

$$h(t) = q \Lambda t q h(0). \quad (10.39)$$

Los valores propios se elevan a la potencia de t haciendo que los valores propios con una magnitud inferior a uno decaigan a cero y los valores propios con una magnitud superior a uno exploten. Cualquier componente de $h(0)$ que no esté alineado con el vector propio más grande eventualmente será descartado.

Este problema es particular de las redes recurrentes. En el caso escalar, imagina multiplicar un peso w por sí mismo muchas veces. El producto w^t desaparecerá o explotará dependiendo de la magnitud de w . Sin embargo, si hacemos una red no recurrente que tiene un peso diferente w_t en cada paso de tiempo, la situación es diferente. Si el estado inicial está dado por v_0 , entonces el estado en el momento t es dado por $v_t = w_0 v_0 + w_1 v_1 + \dots + w_t v_t$. Supongamos que los w_t los valores se generan aleatoriamente, independientemente unos de otros, con media cero y varianza σ_w^2 . La varianza del producto es $O(\sqrt{t} \sigma_w^2)$. Para obtener \sqrt{t} varianza deseada σ_v^2 podemos elegir los pesos individuales con varianza $\sigma_w^2 = \sigma_v^2 / \sqrt{t}$. Las redes feedforward muy profundas con una escala cuidadosamente elegida pueden evitar el problema del gradiente que desaparece y explota, como argumenta [Sussillo \(2014\)](#).

El problema del gradiente de desaparición y explosión de las RNN fue descubierto de forma independiente por investigadores separados ([Hochreiter, 1991](#); [bengio et al., 1993, 1994](#)). Uno puede esperar que el problema pueda evitarse simplemente permaneciendo en una región del espacio de parámetros donde los gradientes no desaparezcan ni exploten. Desafortunadamente, para almacenar memorias de manera que sea resistente a pequeñas perturbaciones, la RNN debe ingresar a una región del espacio de parámetros donde los gradientes desaparecen ([bengio et al., 1993, 1994](#)). Específicamente, siempre que el modelo sea capaz de representar dependencias a largo plazo, el gradiente de una interacción a largo plazo tiene una magnitud exponencialmente menor que el gradiente de una interacción a corto plazo. No significa que sea imposible aprender, sino que puede llevar mucho tiempo aprender las dependencias a largo plazo, porque la señal sobre estas dependencias tenderá a quedar oculta por las fluctuaciones más pequeñas que surjan de las dependencias a corto plazo. En la práctica, los experimentos de [bengio et al. \(1994\)](#) muestran que a medida que aumentamos el alcance de las dependencias que deben capturarse, la optimización basada en gradientes se vuelve cada vez más difícil, y la probabilidad de entrenamiento exitoso de un RNN tradicional a través de SGD alcanza rápidamente 0 para secuencias de solo 10 o 20 de longitud.

Para un tratamiento más profundo de las redes recurrentes como sistemas dinámicos, consulte [Doya \(1993\)](#), [bengio et al. \(1994\)](#) y [Siegelmann y Sontag \(1995\)](#), con una revisión en [Pascanu et al. \(2013\)](#). Las secciones restantes de este capítulo analizan varios enfoques que se han propuesto para reducir la dificultad de aprender dependencias a largo plazo (en algunos casos, permitir que una RNN aprenda dependencias a través de

cientos de pasos), pero el problema de aprender dependencias a largo plazo sigue siendo uno de los principales desafíos en el aprendizaje profundo.

10.8 Redes de estado de eco

El mapeo de pesos recurrentes de $h_{(t-1)}$ a h_t y el mapeo de pesos de entrada de X_t a h_t son algunos de los parámetros más difíciles de aprender en una red recurrente. Uno propuesto (Jaeger, 2003; Masa et al., 2002; Jaeger y Haas, 2004; Jaeger, 2007b) el enfoque para evitar esta dificultad es establecer los pesos recurrentes de modo que las unidades ocultas recurrentes hagan un buen trabajo al capturar el historial de entradas pasadas, *y aprende solo los pesos de salida*. Esta es la idea que se propuso de forma independiente para **redes estatales de eco** o ESN (Jaeger y Haas, 2004; Jaeger, 2007b) y **máquinas de estado líquido** (Masa et al., 2002). Este último es similar, excepto que usa neuronas de pico (con salidas binarias) en lugar de las unidades ocultas de valor continuo que se usan para los ESN. Tanto las ESN como las máquinas de estado líquido se denominan **computación de yacimientos** (Lukoševičius y Jaeger, 2009) para denotar el hecho de que las unidades ocultas forman un reservorio de características temporales que pueden capturar diferentes aspectos de la historia de las entradas.

Una forma de pensar en estas redes recurrentes de computación de reservorios es que son similares a las máquinas kernel: mapean una secuencia de longitud arbitraria (la historia de las entradas hasta el momento), t , en un vector de longitud fija (el estado recurrente h_t), en el que se puede aplicar un predictor lineal (típicamente una regresión lineal) para resolver el problema de interés. Entonces, el criterio de entrenamiento puede diseñarse fácilmente para que sea convexo en función de los pesos de salida. Por ejemplo, si la salida consiste en una regresión lineal desde las unidades ocultas hasta los objetivos de salida, y el criterio de entrenamiento es el error cuadrático medio, entonces es convexo y puede resolverse de manera confiable con algoritmos de aprendizaje simple (Jaeger, 2003).

Por lo tanto, la pregunta importante es: ¿cómo establecemos los pesos recurrentes y de entrada para que se pueda representar un conjunto rico de historias en el estado de la red neuronal recurrente? La respuesta propuesta en la literatura de computación de yacimientos es ver la red recurrente como un sistema dinámico y establecer los pesos de entrada y recurrentes de manera que el sistema dinámico esté cerca del borde de la estabilidad.

La idea original era hacer que los valores propios del jacobiano de la función de transición de estado a estado estuvieran cerca de 1. Como se explica en la sección 8.2.5, una característica importante de una red recurrente es el espectro de valores propios de los jacobianos $J_t = \frac{\partial s_t}{\partial s_{(t-1)}}$. O particular importancia es el **radio espectral** de J_t , definido como el máximo de los valores absolutos de sus valores propios.

Para comprender el efecto del radio espectral, considere el caso simple de retropropagación con una matriz jacobiana \mathbf{J} que no cambia con t . Este caso ocurre, por ejemplo, cuando la red es puramente lineal. Suponer que \mathbf{J} tiene un vector propio v con valor propio correspondiente λ . Considere lo que sucede cuando propagamos un vector gradiente hacia atrás a través del tiempo. Si comenzamos con un vector gradiente g , luego de un paso de retropropagación, tendremos dg , y después n pasos que tendremos $d^{n+1}g$. Ahora considere lo que sucede si, en cambio, propagamos hacia atrás una versión perturbada d . Si empezamos con $g + dv$, luego de un paso, tendremos $(g + dv) + d\lambda v$. Después n pasos, tendremos $(g + dv) + d^n v$. De esto podemos ver que la retropropagación a partir de g y retropropagación a partir de $g + dv$ divergir por d^n después n pasos de retropropagación. Si v se elige como un vector propio unitario de \mathbf{J} con valor propio λ , luego la multiplicación por el jacobiano simplemente escala la diferencia en cada paso. Las dos ejecuciones de retropropagación están separadas por una distancia de δ/λ . Cuando v corresponde al mayor valor de $|\lambda|$, esta perturbación logra la separación más amplia posible de una perturbación inicial de tamaño d .

Cuando $|\lambda| > 1$, el tamaño de la desviación δ/λ crece exponencialmente grande. Cuando $|\lambda| < 1$, el tamaño de la desviación se vuelve exponencialmente pequeño.

Por supuesto, este ejemplo supuso que el jacobiano era el mismo en cada paso de tiempo, lo que corresponde a una red recurrente sin no linealidad. Cuando está presente una no linealidad, la derivada de la no linealidad se aproximará a cero en muchos pasos de tiempo y ayudará a prevenir la explosión resultante de un radio espectral grande. De hecho, el trabajo más reciente sobre redes de estado de eco aboga por utilizar un radio espectral mucho mayor que la unidad ([Yildiz et al., 2012](#); [Jaeger, 2012](#)).

Todo lo que hemos dicho sobre la propagación hacia atrás a través de la multiplicación repetida de matrices se aplica igualmente a la propagación hacia adelante en una red sin no linealidad, donde el estado $h_{(t+1)} = h_{(t)} \cdot W$.

Cuando un mapa lineal W siempre se encoge medida por la norma L_2 , entonces decimos que el mapa es **contractivo**. Cuando el radio espectral es menor que uno, el mapeo de $h_{(t)} \rightarrow h_{(t+1)}$ es contractivo, por lo que un pequeño cambio se vuelve más pequeño después de cada paso de tiempo. Esto necesariamente hace que la red olvide información sobre el pasado cuando usamos un nivel finito de precisión (como números enteros de 32 bits) para almacenar el vector de estado.

La matriz jacobiana nos dice cómo un pequeño cambio de $h_{(t)}$ propaga un paso hacia adelante, o de manera equivalente, cómo el gradiente en $h_{(t+1)}$ se propaga un paso hacia atrás, durante la propagación hacia atrás. Tenga en cuenta que tampoco W debe ser simétricos (aunque sean cuadrados y reales), por lo que pueden tener valores propios y vectores propios de valores complejos, con componentes imaginarios correspondientes a potencialmente oscilatorios

comportamiento (si se aplicara iterativamente el mismo jacobiano). A pesar de que una pequeña variación de $h(t)$ de interés en la retropropagación tienen un valor real, pueden expresarse en una base de valor tan complejo. Lo que importa es lo que sucede con la magnitud (valor absoluto complejo) de estos coeficientes de base posiblemente de valor complejo, cuando multiplicamos la matriz por el vector. Un valor propio con una magnitud mayor que uno corresponde a la ampliación (crecimiento exponencial, si se aplica de forma iterativa) o reducción (decaimiento exponencial, si se aplica de forma iterativa).

Con un mapa no lineal, el jacobiano es libre de cambiar en cada paso. Por lo tanto, la dinámica se vuelve más complicada. Sin embargo, sigue siendo cierto que una pequeña variación inicial puede convertirse en una gran variación después de varios pasos. Una diferencia entre el caso puramente lineal y el caso no lineal es que el uso de una no linealidad aplastante como bronceado puede hacer que la dinámica recurrente se vuelva limitada. Tenga en cuenta que es posible que la propagación hacia atrás retenga una dinámica ilimitada incluso cuando la propagación hacia adelante tiene una dinámica limitada, por ejemplo, cuando una secuencia de bronceado unidades están todas en el medio de su régimen lineal y están conectadas por matrices de peso con radio espectral mayor que 1. Sin embargo, es raro que todos los bronceado unidades para estar simultáneamente en su punto de activación lineal.

La estrategia de las redes de estado de eco es simplemente fijar los pesos para que tengan un radio espectral como 3, donde la información se transporta a través del tiempo pero no explota debido al efecto estabilizador de la saturación de no linealidades como Tanh.

Más recientemente, se ha demostrado que las técnicas utilizadas para establecer los pesos en los ESN podrían utilizarse para *inicializar* los pesos en una red recurrente totalmente entrenable (con los pesos recurrentes de oculto a oculto entrenados usando propagación hacia atrás a través del tiempo), ayudando a aprender dependencias a largo plazo ([Sutskever, 2012](#); [Sutskever et al., 2013](#)). En esta configuración, un radio espectral inicial de 1,2 funciona bien, combinado con el esquema de inicialización dispersa descrito en la sección [8.4](#).

10.9 Unidades con fugas y otras estrategias para múltiples escalas de tiempo

Una forma de lidiar con las dependencias a largo plazo es diseñar un modelo que opere en múltiples escalas de tiempo, de modo que algunas partes del modelo operen en escalas de tiempo detalladas y puedan manejar pequeños detalles, mientras que otras partes operen en escalas de tiempo gruesas y transferir información del pasado distante al presente de manera más eficiente. Son posibles varias estrategias para construir escalas de tiempo finas y gruesas. Estos incluyen la adición de conexiones de salto a lo largo del tiempo, "unidades con fugas" que integran señales con diferentes constantes de tiempo y la eliminación de algunas de las conexiones.

Se utiliza para modelar escalas de tiempo de grano fino.

10.9.1 Adición de conexiones de salto a través del tiempo

Una forma de obtener escalas de tiempo gruesas es agregar conexiones directas de variables en el pasado lejano a variables en el presente. La idea de usar este tipo de conexiones de salto se remonta a Linet al.(1996) y se deriva de la idea de incorporar retrasos en las redes neuronales feedforward (Lang y Hinton,1988). En una red recurrente ordinaria, una conexión recurrente va de una unidad a la vez a una unidad a la vez $t+1$. Es posible construir redes recurrentes con mayores retrasos (bengio,1991).

Como hemos visto en la sección 8.2.5, los gradientes pueden desaparecer o explotar exponencialmente *con respecto al número de pasos de tiempo*. Linet al.(1996) introdujo conexiones recurrentes con un retardo de tiempo de d para mitigar este problema. Gradientes ahora disminuyen exponencialmente en función de $\tau-d$ en vez de τ . Ya que hay ambos tipos de conexiones, los gradientes aún pueden explotar exponencialmente en τ . Esto permite que el algoritmo de aprendizaje capture dependencias más largas, aunque no todas las dependencias a largo plazo pueden representarse bien de esta manera.

10.9.2 Unidades con fugas y un espectro de diferentes escalas de tiempo

Otra forma de obtener caminos en los que el producto de derivadas sea cercano a uno es tener unidades con *lineal* autoconexiones y un peso cercano a uno en estas conexiones.

Cuando acumulamos un promedio móvil $\mu(t)$ de algún valor $v(t)$ aplicando la actualización $\mu(t) \leftarrow a\mu(t-1) + (1-a)v(t)$, el parámetro a es un ejemplo de una autoconexión lineal de $\mu(t-1) a \mu(t)$. Cuando a está cerca de uno, el promedio móvil recuerda información sobre el pasado durante mucho tiempo, y cuando a es casi cero, la información sobre el pasado se descarta rápidamente. Las unidades ocultas con autoconexiones lineales pueden comportarse de manera similar a tales promedios móviles. Tales unidades ocultas se llaman **unidades con fugas**.

Omitir conexiones a través de los pasos de tiempo son una forma de asegurar que una unidad siempre pueda aprender a ser influenciada por un valor de d pasos de tiempo antes. El uso de una autoconexión lineal con un peso cercano a uno es una forma diferente de garantizar que la unidad pueda acceder a valores del pasado. El enfoque de autoconexión lineal permite que este efecto se adapte de manera más suave y flexible ajustando el valor real a en lugar de ajustar la longitud de salto de valor entero.

Estas ideas fueron propuestas por Mozer(1992) y por El Hihi y Bengio(1996). También se encontró que las unidades con fugas son útiles en el contexto de las redes de estado de eco (Jaeger et al.,2007).

Existen dos estrategias básicas para establecer las constantes de tiempo utilizadas por las unidades con fugas. Una estrategia es fijarlos manualmente en valores que permanezcan constantes, por ejemplo, muestreando sus valores de alguna distribución una vez en el momento de la inicialización. Otra estrategia es hacer que las constantes de tiempo sean parámetros libres y aprenderlos. Tener tales unidades con fugas en diferentes escalas de tiempo parece ayudar con las dependencias a largo plazo ([Mozer, 1992; Pascanu et al., 2013](#)).

10.9.3 Eliminación de conexiones

Otro enfoque para manejar las dependencias a largo plazo es la idea de organizar el estado de la RNN en múltiples escalas de tiempo ([El Hihi y Bengio, 1996](#)), con información que fluye más fácilmente a través de largas distancias en las escalas de tiempo más lentas.

Esta idea difiere de las conexiones de salto a través del tiempo discutidas anteriormente porque involucra activamente *quitando* conexiones de longitud uno y reemplazándolas con conexiones más largas. Las unidades modificadas de esa manera se ven obligadas a operar en una escala de tiempo prolongada. Saltar conexiones a través del tiempo *agregar* bordes Las unidades que reciben estas nuevas conexiones pueden aprender a operar en una escala de tiempo prolongada, pero también pueden optar por concentrarse en sus otras conexiones a corto plazo.

Hay diferentes formas en las que un grupo de unidades recurrentes puede ser forzado a operar en diferentes escalas de tiempo. Una opción es hacer que las unidades recurrentes tengan fugas, pero tener diferentes grupos de unidades asociadas con diferentes escalas de tiempo fijas. Esta fue la propuesta en [Mozer\(1992\)](#) y se ha utilizado con éxito en [Pascanu et al. \(2013\)](#). Otra opción es tener actualizaciones explícitas y discretas en diferentes momentos, con una frecuencia diferente para diferentes grupos de unidades. Este es el enfoque de [El Hihi y Bengio\(1996\)](#) y [Koutnik et al.\(2014\)](#). Funcionó bien en varios conjuntos de datos de referencia.

10.10 La memoria a largo plazo y otras RNN cerradas

Al momento de escribir este artículo, los modelos de secuencia más efectivos utilizados en aplicaciones prácticas se denominan **RNN controlados**. Estos incluyen el **memoria a corto plazo** y redes basadas en la **unidad recurrente cerrada**.

Al igual que las unidades con fugas, las RNN cerradas se basan en la idea de crear caminos a través del tiempo que tienen derivados que ni desaparecen ni explotan. Las unidades con fugas hicieron esto con pesos de conexión que eran constantes elegidas manualmente o eran parámetros. Los RNN cerrados generalizan esto a los pesos de conexión que pueden cambiar

en cada paso de tiempo.

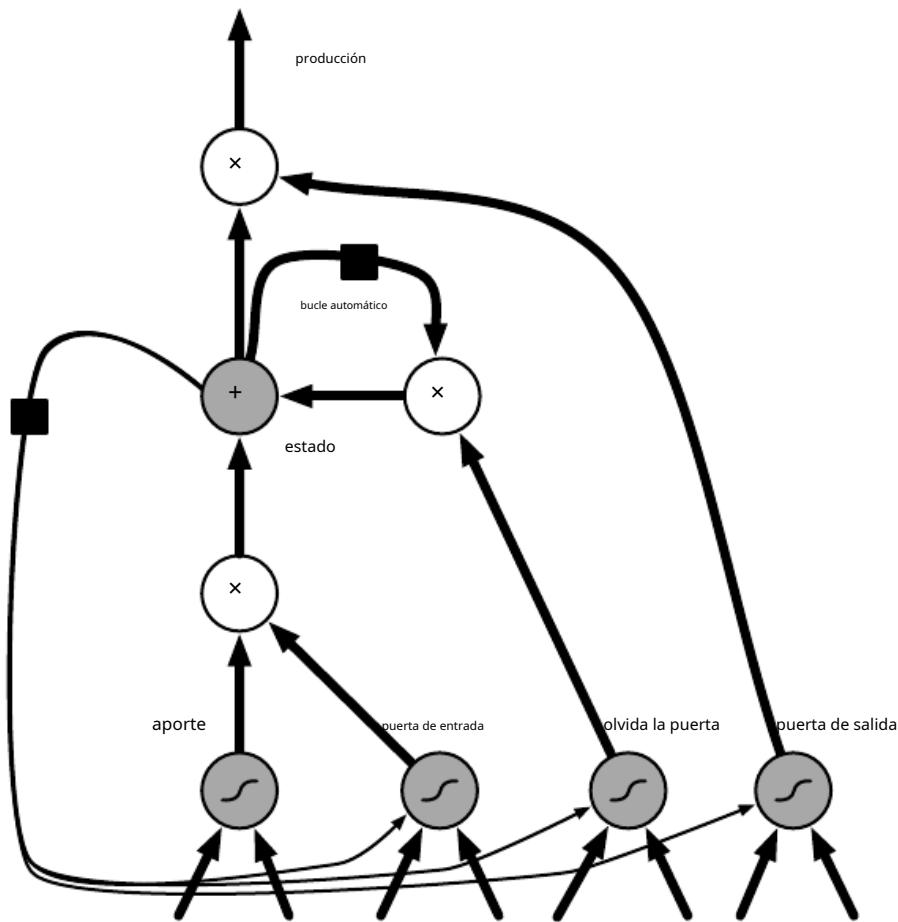


Figura 10.16: Diagrama de bloques de la “celda” de la red recurrente LSTM. Las celdas se conectan de forma recurrente entre sí, reemplazando las unidades ocultas habituales de las redes recurrentes ordinarias. Una característica de entrada se calcula con una unidad de neurona artificial regular. Su valor se puede acumular en el estado si la puerta de entrada sigmoidal lo permite. La unidad de estado tiene un autobucle lineal cuyo peso está controlado por la puerta de olvido. La puerta de salida puede cerrar la salida de la celda. Todas las unidades de activación tienen una no linealidad sigmoidea, mientras que la unidad de entrada puede tener cualquier no linealidad aplastante. La unidad de estado también se puede utilizar como entrada adicional para las unidades de activación. El cuadrado negro indica un retraso de un solo paso de tiempo.

Las unidades con fugas 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 haya utilizado esa información, podría ser útil para la red neuronal *olvidar* el viejo estado. Por ejemplo, si una secuencia está hecha de subsecuencias y queremos que una unidad con fugas acumule evidencia dentro de cada subsecuencia, necesitamos un mecanismo para olvidar el estado anterior al establecerlo en cero. En lugar de decidir manualmente cuándo borrar el estado, queremos que la red neuronal aprenda a decidir cuándo hacerlo. Este

es lo que hacen los RNN cerrados.

10.10.1 LSTM

La ingeniosa idea de introducir bucles automáticos para producir caminos en los que el gradiente puede fluir durante períodos prolongados es una contribución fundamental del diseño inicial. **memoria a largo plazo (LSTM)** modelo (Hochreiter y Schmidhuber, 1997). Una adición crucial ha sido hacer que el peso de este bucle automático esté condicionado por el contexto, en lugar de fijo (Gers et al., 2000). Al hacer que el peso de este autobucle sea controlado (controlado por otra unidad oculta), la escala de tiempo de integración se puede cambiar dinámicamente. En este caso, queremos decir que incluso para un LSTM con parámetros fijos, la escala de tiempo de integración puede cambiar según la secuencia de entrada, porque las constantes de tiempo las genera el propio modelo. El LSTM se ha encontrado extremadamente exitoso en muchas aplicaciones, como el reconocimiento de escritura a mano sin restricciones (Tumbas et al., 2009), reconocimiento de voz (Tumba et al., 2013; Graves y Jaitly, 2014), generación de escritura a mano (Tumbas, 2013), máquina traductora (Sutskever et al., 2014), subtítulos de imagen (Kiróset al., 2014b; Vinilo set al., 2014b; Xue et al., 2015) y análisis (Vinilo set al., 2014a).

El diagrama de bloques LSTM se ilustra en la figura 10.16. Las ecuaciones de propagación directa correspondientes se dan a continuación, en el caso de una arquitectura de red recurrente poco profunda. También se han utilizado con éxito arquitecturas más profundas (Tumbas et al., 2013; Pascanu et al., 2014a). En lugar de una unidad que simplemente aplica una no linealidad de elementos a la transformación afín de entradas y unidades recurrentes, las redes recurrentes LSTM tienen "celdas LSTM" que tienen una recurrencia interna (un autobucle), además de la recurrencia externa de la RNN. Cada celda tiene las mismas entradas y salidas que una red recurrente ordinaria, pero tiene más parámetros y un sistema de unidades de activación que controla el flujo de información. El más importante componente es la unidad de estados $x(t)$ que tiene un auto-bucle lineal similar a la fuga unidades descritas en el apartado anterior. Sin embargo, aquí, el peso del bucle automático (o el constante de tiempo asociada) es controlado por un **olvida la puerta** unidad $f_i(t)$ (para paso de tiempo t y celular i), que establece este peso en un valor entre 0 y 1 a través de una unidad sigmoidea:

$$f_i(t) = \sigma \cdot b_f + \sum_j t u_{y_o, j}(t) + \sum_j W_{y_o, j}^{(t-1)}, \quad (10.40)$$

dónde $x(t)$ es el vector de entrada actual y $h(t)$ es el vector de capa oculta actual, que contiene las salidas de todas las celdas LSTM, y_b , t_u , W_F son, respectivamente, sesgos, pesos de entrada y pesos recurrentes para las puertas de olvido. La celda LSTM

Por lo tanto, el estado interno se actualiza de la siguiente manera, pero con un peso de bucle automático condicional $R_i(t)$

$$S_i(t) = R_i(t) \cdot S_i^{(t-1)} + \sigma(b_{gramo} - t u_{yo,j}^{(t)} + W_{yo,j} h_j^{(t-1)}) \quad (10.41)$$

dónde b , $t u$ y W denotan respectivamente los sesgos, los pesos de entrada y los pesos recurrentes en la celda LSTM. El **puerta de entrada externa** $a_{gramo}(t)$ se calcula de manera similar a la puerta de olvido (con una unidad sigmoidea para obtener un valor de puerta entre 0 y 1), pero con sus propios parámetros:

$$a_{gramo} = \sigma(b_{gramo} + t u_{yo,j}^{(t)} + W_{yo,j} h_j^{(t-1)}) \quad (10.42)$$

La salida $h_i(t)$ de la celda LSTM también se puede apagar, a través del **puerta de salida** $q_i(t)$, que también utiliza una unidad sigmoidea para activar:

$$h_i(t) = \tanh S_i(t) \cdot q_i(t) \quad (10.43)$$

$$q_i(t) = \sigma(b_o - t u_{yo,j}^{(t)} + W_{yo,j} h_j^{(t-1)}) \quad (10.44)$$

que tiene parámetros b_o , $t u_o$, W_o por sus sesgos, pesos de entrada y recurrencia pesos, respectivamente. Entre las variantes, se puede optar por utilizar el estado de la celda $s_i(t)$ como una entrada extra (con su peso) en las tres puertas de la i -ésima unidad, como se muestra en la figura 10.16. Esto requeriría tres parámetros adicionales.

Se ha demostrado que las redes LSTM 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 et al., 1994](#); [Hochreiter y Schmidhuber, 1997](#); [Hochreiter et al., 2001](#)), luego en tareas desafiantes de procesamiento de secuencias donde se obtuvo un rendimiento de última generación ([Tumbas, 2012](#); [Tumba et al., 2013](#); [Sutskever et al., 2014](#)). Se han estudiado y utilizado variantes y alternativas al LSTM y se analizan a continuación.

10.10.2 Otros RNN controlados

¿Qué piezas de la arquitectura LSTM son realmente necesarias? ¿Qué otras arquitecturas exitosas podrían diseñarse que permitan a la red controlar dinámicamente la escala de tiempo y el comportamiento de olvido de diferentes unidades?

Algunas respuestas a estas preguntas se dan con el trabajo reciente sobre las RNN cerradas, cuyas unidades también se conocen como unidades recurrentes cerradas o GRU ([Cho et al., 2014b](#); [Chung et al., 2014, 2015a](#); [jozefowicz et al., 2015](#); [Chrupala et al., 2015](#)). La principal diferencia con el LSTM es que una sola unidad de activación controla simultáneamente el factor de olvido y la decisión de actualizar la unidad de estado. Las ecuaciones de actualización son las siguientes:

$$\begin{aligned} h_i^{(t)} = & tu_{i(t-1)} \cdot h_i^{(t-1)} + (1 - tu_i^{(t-1)}) \sigma \cdot b_r + \\ & \sum_j t u_{yo,j}^{(t-1)} + \sum_j W_{yo,j} h_j^{(t-1)}, \end{aligned} \quad (10.45)$$

dónde $t u$ significa puerta de "actualización" y r para la puerta de "reinicio". Su valor se define como de costumbre:

$$tu_i^{(t)} = \sigma \cdot b_u + \sum_j t u_{yo,j}^{(t)} + \sum_j W_{j,i} h_j^{(t)} \quad (10.46)$$

y

$$r_i^{(t)} = \sigma \cdot b_r + \sum_j t u_{yo,j}^{(t)} + \sum_j W_{yo,j} h_j^{(t)}. \quad (10.47)$$

Las puertas de reinicio y actualizaciones pueden "ignorar" individualmente partes del vector de estado. Las compuertas de actualización actúan como integradores con fugas condicionales que pueden compuertas lineales de cualquier dimensión, eligiendo así copiarla (en un extremo del sigmoide) o ignorarla por completo (en el otro extremo) reemplazándola por el nuevo valor de "estado objetivo" (hacia el que quiere converger el integrador con fugas). 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 (u puerta de olvido) podría compartirse entre varias 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 arquitectónicas de LSTM y GRU no encontraron ninguna variante que superara claramente a ambos en una amplia gama de tareas ([Greff et al., 2015](#); [jozefowicz et al., 2015](#)). Greff et al. (2015) encontró que un ingrediente crucial es la puerta de olvido, mientras que [jozefowicz et al. \(2015\)](#) encontró que agregar un sesgo de 1 a la puerta de olvido de LSTM, una práctica defendida por [Gers et al. \(2000\)](#), hace que el LSTM sea tan fuerte como la mejor de las variantes arquitectónicas exploradas.

10.11 Optimización para dependencias a largo plazo

Sección 8.2.5 y sección 10.7 han descrito los problemas de gradiente de fuga y explosión que ocurren cuando se optimizan RNN en muchos pasos de tiempo.

Una idea interesante propuesta por Martens y Sutskever (2011) es que las segundas derivadas pueden desaparecer al mismo tiempo que desaparecen las primeras derivadas. Los algoritmos de optimización de segundo orden pueden entenderse aproximadamente como dividir la primera derivada por la segunda derivada (en una dimensión superior, multiplicar el gradiente por la hessiana inversa). Si la segunda derivada se contrae a un ritmo similar al de la primera derivada, entonces la relación entre la primera y la segunda derivada puede permanecer relativamente constante. Desafortunadamente, los métodos de segundo orden tienen muchos inconvenientes, incluido el alto costo computacional, la necesidad de un minilote grande y la tendencia a sentirse atraído por los puntos de silla. Martens y Sutskever (2011) encontraron resultados prometedores utilizando métodos de segundo orden. Más tarde, Sutskever et al. (2013) encontró que métodos más simples como el impulso de Nesterov con una inicialización cuidadosa podrían lograr resultados similares. Ver Sutskever (2012) para más detalles. Ambos enfoques han sido reemplazados en gran medida por el simple uso de SGD (incluso sin impulso) aplicado a los LSTM. Esto es parte de un tema continuo en el aprendizaje automático de que a menudo es mucho más fácil diseñar un modelo que sea fácil de optimizar que diseñar un algoritmo de optimización más poderoso.

10.11.1 Gradientes de recorte

Como se discutió en la sección 8.2.4, las funciones fuertemente no lineales, como las calculadas por una red recurrente durante muchos pasos de tiempo, tienden a tener derivadas que pueden ser de magnitud muy grande o muy pequeña. Esto se ilustra en la figura 8.3 y figura 10.17, en el que vemos que la función objetivo (en función de los parámetros) tiene un “paisaje” en el que se encuentran “acantilados”: regiones anchas y más bien planas separadas por diminutas regiones donde la función objetivo cambia rápidamente, formando una especie de acantilado.

La dificultad que surge es que cuando el gradiente del parámetro es muy grande, una actualización del parámetro de descenso del gradiente podría arrojar los parámetros muy lejos, en una región donde la función objetivo es más grande, deshaciendo gran parte del trabajo que se había hecho para llegar a la solución actual. . El gradiente nos dice la dirección que corresponde al descenso más pronunciado dentro de una región infinitesimal que rodea los parámetros actuales. Fuera de esta región infinitesimal, la función de costo puede comenzar a curvarse hacia arriba. La actualización debe elegirse para que sea lo suficientemente pequeña para evitar atravesar demasiada curvatura hacia arriba. Usualmente usamos tasas de aprendizaje que

decaen lo suficientemente lento como para que los pasos consecutivos tengan aproximadamente la misma tasa de aprendizaje. Un tamaño de paso que es apropiado para una parte relativamente lineal del paisaje a menudo es inapropiado y provoca un movimiento cuesta arriba si ingresamos a una parte más curva del paisaje en el siguiente paso.

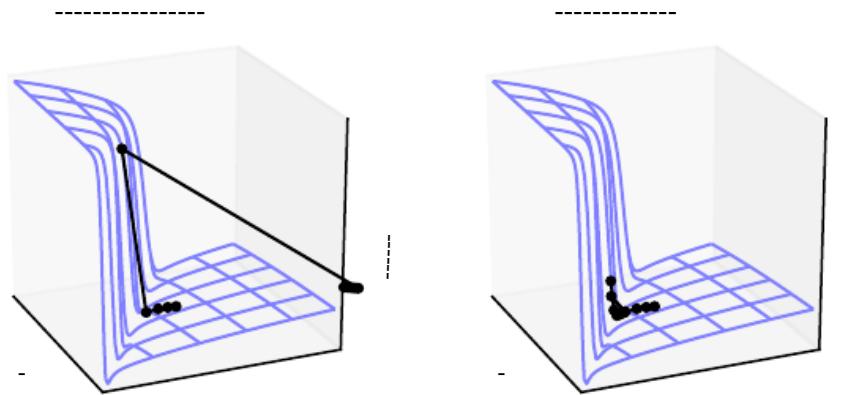


Figura 10.17: Ejemplo del efecto de recorte de gradiente en una red recurrente con dos parámetros wyb . El recorte de gradiente puede hacer que el descenso de gradiente funcione de manera más razonable en las cercanías de acantilados extremadamente empinados. Estos acantilados empinados comúnmente ocurren en redes recurrentes cerca de donde una red recurrente se comporta de manera aproximadamente lineal. El precipicio es exponencialmente empinado en el número de pasos de tiempo porque la matriz de pesos se multiplica por sí misma una vez por cada paso de tiempo.(Izquierda)El descenso de pendiente sin recorte de pendiente rebasa el fondo de este pequeño barranco y luego recibe una pendiente muy grande desde la cara del acantilado. El gran gradiente impulsa catastróficamente los parámetros fuera de los ejes de la trama.(Bien)El descenso de gradiente con recorte de gradiente tiene una reacción más moderada al acantilado. Si bien asciende por la cara del acantilado, el tamaño del paso está restringido para que no pueda alejarse de la región empinada cerca de la solución. Figura adaptada con permiso de [Pascanu et al.](#)(2013).

Los profesionales han utilizado un tipo simple de solución durante muchos años:
recortar el gradiente. Hay diferentes instancias de esta idea ([mikolov,2012; Pascanu et al., 2013](#)). Una opción es recortar el gradiente de parámetros de un minilote *elemento sabio* ([mikolov,2012](#)) justo antes de la actualización de parámetros. otra es *pararecortar la norma* $\|g\|$ del *gradientegramo* ([Pascanu et al.,2013](#)) justo antes de la actualización del parámetro:

$$\text{si } \|g\| > v \quad (10.48)$$

$$g_{\text{gramo}} \leftarrow \frac{g_{\text{gramo}}}{\|g\|} \quad (10.49)$$

dónde ϵ es el umbral de la norma y γ se utiliza para actualizar los parámetros. Debido a que el gradiente de todos los parámetros (incluidos los diferentes grupos de parámetros, como pesos y sesgos) se vuelve a normalizar junto con un solo factor de escala, el último método tiene la ventaja de que garantiza que cada paso sigue en la dirección del gradiente, pero experimenta sugieren que ambas formas funcionan de manera similar. Aunque la actualización de parámetros tiene la misma dirección que el gradiente real, con el recorte de la norma de gradiente, la norma del vector de actualización de parámetros ahora está limitada. Este gradiente acotado evita realizar un paso perjudicial cuando el gradiente explota. De hecho, incluso simplemente tomando un *paso aleatorio* cuando la magnitud del gradiente está por encima de un umbral tiende a funcionar casi igual de bien. Si la explosión es tan severa que el gradiente es numéricamente información nana (considerado infinito o no-un-número), entonces un paso aleatorio de tamaño ϵ se puede tomar y normalmente se alejará de la configuración numéricamente inestable. Recortar la norma de gradiente por minilote no cambiará la dirección del gradiente para un minilote individual. Sin embargo, tomar el promedio del gradiente recortado por la norma de muchos minibatches no es equivalente a recortar la norma del gradiente verdadero (el gradiente formado a partir del uso de todos los ejemplos). Los ejemplos que tienen una norma de gradiente grande, así como los ejemplos que aparecen en el mismo minibatch que dichos ejemplos, verán disminuida su contribución a la dirección final. Esto contrasta con el descenso de gradiente de minibatch tradicional, donde la verdadera dirección del gradiente es igual al promedio de todos los gradientes de minibatch. Dicho de otra manera, el descenso de gradiente estocástico tradicional utiliza una estimación imparcial del gradiente, mientras que el descenso de gradiente con recorte de normas introduce un sesgo heurístico que sabemos empíricamente que es útil. Con el recorte por elementos, la dirección de la actualización no está alineada con el gradiente real o el gradiente del minibatch, pero sigue siendo una dirección descendente. También se ha propuesto ([Tumbas, 2013](#)) para recortar el degradado propagado hacia atrás (con respecto a las unidades ocultas), pero no se ha publicado ninguna comparación entre estas variantes; conjeturamos que todos estos métodos se comportan de manera similar.

10.11.2 Regularización para incentivar el flujo de información

El recorte de degradado ayuda a lidiar con los degradados explosivos, pero no ayuda con los degradados que se desvanecen. Para abordar los gradientes que desaparecen y capturar mejor las dependencias a largo plazo, discutimos la idea de crear caminos en el gráfico computacional de la arquitectura recurrente desplegada a lo largo de los cuales el producto de los gradientes asociados con los arcos está cerca de 1. Un enfoque para lograr esto es con LSTM y otros selfloops y mecanismos de activación, descritos anteriormente en la sección [10.10](#). Otra idea es regularizar o restringir los parámetros para fomentar el “flujo de información”. En particular, nos gustaría que el vector gradiente $\nabla_{\theta_0} L$ esté propagado hacia atrás a

mantener su magnitud, incluso si la función de pérdida solo penaliza la salida al final de la secuencia. Formalmente, queremos

$$(\nabla_{h(t)} L) \frac{\partial h(t)}{\partial h(t-1)} \quad (10.50)$$

ser tan grande como

$$\|\nabla_{h(t)} L\| \quad (10.51)$$

Con este objetivo, [Pascanu et al.\(2013\)](#) proponen el siguiente regularizador:

$$\Omega = \frac{1}{t} \left(\frac{\|(\nabla_{h(t)} L) \frac{\partial h(t)}{\partial h(t-1)}\|^2}{\|\nabla_{h(t)} L\|^2} - 1 \right)^{-2} \quad (10.52)$$

Calcular el gradiente de este regularizador puede parecer difícil, pero [Pascanu et al.\(2013\)](#) proponen una aproximación en la que consideramos los vectores retropropagados $\nabla_{h(t)} L$ como si fueran constantes (a los efectos de este regularizador, de modo que no haya necesidad de retropropagarse a través de ellos). Los experimentos con este regularizador sugieren que, si se combina con la heurística de recorte de normas (que maneja la explosión de gradiente), el regularizador puede aumentar considerablemente el alcance de las dependencias que un RNN puede aprender. Debido a que mantiene la dinámica RNN al borde de gradientes explosivos, el recorte de gradiente es particularmente importante. Sin recorte de degradado, la explosión de degradado impide que el aprendizaje tenga éxito.

Una debilidad clave de este enfoque es que no es tan efectivo como el LSTM para tareas donde abundan los datos, como el modelado de lenguaje.

10.12 Memoria explícita

La inteligencia requiere conocimiento y la adquisición de conocimiento se puede hacer a través del aprendizaje, lo que ha motivado el desarrollo de arquitecturas profundas a gran escala. Sin embargo, existen diferentes tipos de conocimiento. Algunos conocimientos pueden ser implícitos, subconscientes y difíciles de verbalizar, por ejemplo, cómo caminar o cómo un perro se ve diferente de un gato. Otros conocimientos pueden ser explícitos, declarativos y relativamente sencillos de poner en palabras: conocimientos cotidianos de sentido común, como "un gato es un tipo de animal", o hechos muy específicos que necesita saber para lograr sus objetivos actuales, como "el la reunión con el equipo de ventas es a las 3:00 p. m. en la sala 141".

Las redes neuronales sobresalen en el almacenamiento de conocimiento implícito. Sin embargo, les cuesta memorizar hechos. El descenso de gradiente estocástico requiere muchas presentaciones del

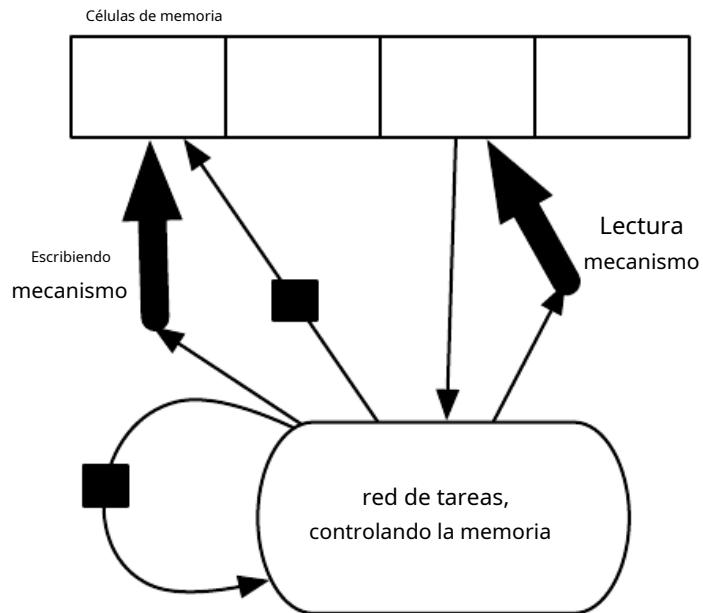


Figura 10.18: Esquema de un ejemplo de una red con memoria explícita, que captura algunos de los elementos clave del diseño de la máquina neuronal de Turing. En este diagrama distinguimos la parte de "representación" del modelo (la "red de tareas", aquí una red recurrente en la parte inferior) de la parte de "memoria" del modelo (el conjunto de celdas), que puede almacenar hechos. La red de tareas aprende a "controlar" la memoria, decidiendo desde dónde leer y dónde escribir dentro de la memoria (a través de los mecanismos de lectura y escritura, indicados por flechas en negrita que apuntan a las direcciones de lectura y escritura).

misma entrada antes de que pueda almacenarse en los parámetros de una red neuronal, e incluso entonces, esa entrada no se almacenará con especial precisión. Tumbaset al.(2014b) planteó la hipótesis de que esto se debe a que las redes neuronales carecen del equivalente del**memoria de trabajo** sistema que permite a los seres humanos retener y manipular explícitamente piezas de información que son relevantes para lograr algún objetivo. Dichos componentes de memoria explícita permitirían que nuestros sistemas no solo almacenen y recuperen rápida e "intencionalmente" hechos específicos, sino también razonen secuencialmente con ellos. La necesidad de redes neuronales que puedan procesar información en una secuencia de pasos, cambiando la forma en que la entrada se introduce en la red en cada paso, se ha reconocido durante mucho tiempo como importante para la capacidad de razonar en lugar de dar respuestas automáticas e intuitivas a la aporte (Hinton, 1990).

Para resolver esta dificultad, Weston et al.(2014) introducidoredes de memoria que incluyen un conjunto de celdas de memoria a las que se puede acceder a través de un mecanismo de direccionamiento. Las redes de memoria originalmente requerían una señal de supervisión que les indicara cómo usar sus celdas de memoria. Tumbaset al.(2014b) introdujo el**maquina neural de turing**, que es capaz de aprender a leer y escribir contenido arbitrario en celdas de memoria sin supervisión explícita sobre qué acciones emprender, y permitió el entrenamiento de extremo a extremo sin esta señal de supervisión, mediante el uso de un mecanismo de atención suave basado en contenido (ver Bahdanau et al.(2015) y sección 12.4.5.1). Este mecanismo de direccionamiento suave se ha convertido en estándar con otras arquitecturas relacionadas que emulan mecanismos algorítmicos de una manera que aún permite la optimización basada en gradientes (Sukhbaatar et al., 2015; Joulin y Mikolov, 2015; Kumar et al., 2015;vinilo set al., 2015a; Grefenstette et al., 2015).

Cada celda de memoria se puede considerar como una extensión de las celdas de memoria en LSTM y GRU. La diferencia es que la red genera un estado interno que elige en qué celda leer o escribir, al igual que los accesos a la memoria en una computadora digital leen o escriben en una dirección específica.

Es difícil optimizar funciones que producen direcciones enteras exactas. Para aliviar este problema, los NTM realmente leen o escriben desde muchas celdas de memoria simultáneamente. Para leer, toman un promedio ponderado de muchas celdas. Para escribir, modifican múltiples celdas en diferentes cantidades. Los coeficientes para estas operaciones se eligen para enfocarse en una pequeña cantidad de celdas, por ejemplo, al producirlos a través de una función softmax. El uso de estos pesos con derivadas distintas de cero permite optimizar las funciones que controlan el acceso a la memoria mediante el descenso de gradiente. El gradiente de estos coeficientes indica si cada uno de ellos debe aumentar o disminuir, pero el gradiente normalmente será grande solo para aquellas direcciones de memoria que reciben un coeficiente grande.

Estas celdas de memoria generalmente se aumentan para contener un vector, en lugar de

el único escalar almacenado por una celda de memoria LSTM o GRU. Hay dos razones para aumentar el tamaño de la celda de memoria. Una de las razones es que hemos aumentado el costo de acceder a una celda de memoria. Pagamos el costo computacional de producir un coeficiente para muchas celdas, pero esperamos que estos coeficientes se agrupen alrededor de un pequeño número de celdas. Al leer un valor vectorial, en lugar de un valor escalar, podemos compensar parte de este costo. Otra razón para usar celdas de memoria con valores vectoriales es que permiten **direcciónamiento basado en contenido**, donde el peso utilizado para leer o escribir desde una celda es una función de esa celda. Las celdas con valores vectoriales nos permiten recuperar una memoria completa con valores vectoriales si somos capaces de producir un patrón que coincida con algunos pero no con todos sus elementos. Esto es análogo a la forma en que las personas pueden recordar la letra de una canción basándose en unas pocas palabras. Podemos pensar en una instrucción de lectura basada en el contenido como si dijera: "Recupere la letra de la canción que tiene el estribillo 'Todos vivimos en un submarino amarillo'". El direcciónamiento basado en el contenido es más útil cuando hacemos que los objetos que se van a recuperar sean grandes; si cada letra de la canción estuviera almacenada en una celda de memoria separada, no podríamos encontrarlas de esta manera. En comparación, **direcciónamiento basado en la ubicación** no se permite hacer referencia al contenido de la memoria. Podemos pensar en una instrucción de lectura basada en la ubicación que dice "Recupera la letra de la canción en el espacio 347". El direcciónamiento basado en la ubicación a menudo puede ser un mecanismo perfectamente sensato incluso cuando las celdas de memoria son pequeñas.

Si el contenido de una celda de memoria se copia (no se olvida) en la mayoría de los pasos de tiempo, entonces la información que contiene se puede propagar hacia adelante en el tiempo y los gradientes se pueden propagar hacia atrás en el tiempo sin desaparecer ni explotar.

El enfoque de memoria explícita se ilustra en la figura 10.18, donde vemos que se acopla una "red neuronal de tareas" con una memoria. Aunque esa red neuronal de tareas podría ser realimentada o recurrente, el sistema general es una red recurrente. La red de tareas puede elegir leer o escribir en direcciones de memoria específicas. La memoria explícita parece permitir que los modelos aprendan tareas que los RNN ordinarios o los RNN LSTM no pueden aprender. Una de las razones de esta ventaja puede ser que la información y los gradientes se pueden propagar (hacia delante o hacia atrás en el tiempo, respectivamente) durante períodos muy prolongados.

Como alternativa a la retropropagación a través de promedios ponderados de celdas de memoria, podemos interpretar los coeficientes de direcciónamiento de memoria como probabilidades y leer estocásticamente solo una celda ([Zaremba y Sutskever, 2015](#)). La optimización de modelos que toman decisiones discretas requiere algoritmos de optimización especializados, descritos en la sección 20.9.1. Hasta ahora, entrenar estas arquitecturas estocásticas que toman decisiones discretas sigue siendo más difícil que entrenar algoritmos deterministas que toman decisiones blandas.

Ya sea suave (permitiendo retropropagación) o estocástico y duro, el

mecanismo para elegir una dirección es en su forma idéntica a la **mecanismo de atención** que se había introducido previamente en el contexto de la traducción automática ([Bahdanau et al., 2015](#)) y discutido en la sección [12.4.5.1](#). La idea de mecanismos de atención para redes neuronales se introdujo incluso antes, en el contexto de la generación de escritura a mano ([Tumbas, 2013](#)), con un mecanismo de atención que estaba obligado a avanzar solo en el tiempo a través de la secuencia. En el caso de la traducción automática y las redes de memoria, en cada paso, el foco de atención puede moverse a un lugar completamente diferente, en comparación con el paso anterior.

Las redes neuronales recurrentes proporcionan una forma de extender el aprendizaje profundo a los datos secuenciales. Son la última herramienta importante en nuestra caja de herramientas de aprendizaje profundo. Nuestra discusión ahora pasa a cómo elegir y usar estas herramientas y cómo aplicarlas a tareas del mundo real.

Capítulo 11

Metodología Práctica

La aplicación exitosa de técnicas de aprendizaje profundo requiere más que un buen conocimiento de qué algoritmos existen y los principios que explican cómo funcionan. Un buen practicante de aprendizaje automático también necesita saber cómo elegir un algoritmo para una aplicación en particular y cómo monitorear y responder a los comentarios obtenidos de los experimentos para mejorar un sistema de aprendizaje automático. Durante el desarrollo diario de los sistemas de aprendizaje automático, los profesionales deben decidir si recopilan más datos, aumentan o reducen la capacidad del modelo, agregan o eliminan funciones de regularización, mejoran la optimización de un modelo, mejoran la inferencia aproximada en un modelo o depuran el software. implementación del modelo. Todas estas operaciones requieren al menos tiempo para probar,

La mayor parte de este libro trata sobre diferentes modelos de aprendizaje automático, algoritmos de entrenamiento y funciones objetivas. Esto puede dar la impresión de que el ingrediente más importante para ser un experto en aprendizaje automático es conocer una amplia variedad de técnicas de aprendizaje automático y ser bueno en diferentes tipos de matemáticas. En la práctica, generalmente se puede hacer mucho mejor con una aplicación correcta de un algoritmo común que aplicando descuidadamente un algoritmo oscuro. La aplicación correcta de un algoritmo depende del dominio de una metodología bastante simple. Muchas de las recomendaciones de este capítulo están adaptadas de [ng\(2015\)](#).

Recomendamos el siguiente proceso de diseño práctico:

- Determine sus objetivos: qué métrica de error usar y su valor objetivo para esta métrica de error. Estos objetivos y métricas de error deben estar impulsados por el problema que la aplicación pretende resolver.
- Establezca un canal de trabajo extremo a extremo tan pronto como sea posible, incluida la

estimación de las métricas de rendimiento apropiadas.

- Instrumente bien el sistema para determinar cuellos de botella en el rendimiento. Diagnóstique qué componentes están funcionando peor de lo esperado y si se debe a un ajuste excesivo, inadecuado o a un defecto en los datos o el software.
- Realice cambios incrementales repetidamente, como la recopilación de nuevos datos, el ajuste de hiperparámetros o el cambio de algoritmos, en función de los hallazgos específicos de su instrumentación.

Como ejemplo, utilizaremos el sistema de transcripción de números de direcciones de Street View ([Buen compañero et al., 2014d](#)). El propósito de esta aplicación es agregar edificios a Google Maps. Los coches de Street View fotografían los edificios y registran las coordenadas GPS asociadas a cada fotografía. Una red convolucional reconoce el número de dirección en cada fotografía, lo que permite que la base de datos de Google Maps agregue esa dirección en la ubicación correcta. La historia de cómo se desarrolló esta aplicación comercial da un ejemplo de cómo seguir la metodología de diseño que defendemos.

A continuación describimos cada uno de los pasos de este proceso.

11.1 Métricas de rendimiento

Determinar sus objetivos, en términos de qué métrica de error usar, es un primer paso necesario porque su métrica de error guiará todas sus acciones futuras. También debe tener una idea del nivel de rendimiento que desea.

Tenga en cuenta que para la mayoría de las aplicaciones, es imposible lograr un error de cero absoluto. El error de Bayes define la tasa de error mínima que puede esperar lograr, incluso si tiene datos de entrenamiento infinitos y puede recuperar la verdadera distribución de probabilidad. Esto se debe a que sus características de entrada pueden no contener información completa sobre la variable de salida o porque el sistema puede ser intrínsecamente estocástico. También estará limitado por tener una cantidad finita de datos de entrenamiento.

La cantidad de datos de entrenamiento puede estar limitada por una variedad de razones. Cuando su objetivo es crear el mejor producto o servicio posible del mundo real, normalmente puede recopilar más datos, pero debe determinar el valor de reducir aún más el error y compararlo con el costo de recopilar más datos. La recopilación de datos puede requerir tiempo, dinero o sufrimiento humano (por ejemplo, si su proceso de recopilación de datos implica la realización de pruebas médicas invasivas). Cuando su objetivo es responder una pregunta científica sobre qué algoritmo funciona mejor en un punto de referencia fijo, el punto de referencia

la especificación generalmente determina el conjunto de entrenamiento y no se le permite recopilar más datos.

¿Cómo se puede determinar un nivel razonable de rendimiento a esperar? Por lo general, en el entorno académico, tenemos una estimación de la tasa de error que se puede lograr en función de los resultados de referencia publicados anteriormente. En la configuración de palabras reales, tenemos una idea de la tasa de error necesaria para que una aplicación sea segura, rentable o atractiva para los consumidores. Una vez que haya determinado su tasa de error deseada realista, sus decisiones de diseño se guiarán por alcanzar esta tasa de error.

Otra consideración importante además del valor objetivo de la métrica de rendimiento es la elección de qué métrica usar. Se pueden usar varias métricas de rendimiento diferentes para medir la eficacia de una aplicación completa que incluye componentes de aprendizaje automático. Estas métricas de rendimiento suelen ser diferentes de la función de costo utilizada para entrenar el modelo. Como se describe en la sección 5.1.2, es común medir la precisión, o de manera equivalente, la tasa de error de un sistema.

Sin embargo, muchas aplicaciones requieren métricas más avanzadas.

A veces es mucho más costoso cometer un tipo de error que otro. Por ejemplo, un sistema de detección de spam de correo electrónico puede cometer dos tipos de errores: clasificar incorrectamente un mensaje legítimo como spam y permitir incorrectamente que un mensaje de spam aparezca en la bandeja de entrada. Es mucho peor bloquear un mensaje legítimo que permitir que pase un mensaje cuestionable. En lugar de medir la tasa de error de un clasificador de spam, es posible que deseemos medir alguna forma de costo total, donde el costo de bloquear mensajes legítimos es más alto que el costo de permitir mensajes de spam.

A veces deseamos entrenar un clasificador binario destinado a detectar algún evento raro. Por ejemplo, podríamos diseñar una prueba médica para una enfermedad rara. Supongamos que solo una de cada millón de personas tiene esta enfermedad. Podemos lograr fácilmente una precisión del 99,9999 % en la tarea de detección, simplemente codificando el clasificador para informar siempre que la enfermedad está ausente. Claramente, la precisión es una forma pobre de caracterizar el desempeño de tal sistema. Una forma de resolver este problema es medir **precisión y recuperación**. La precisión es la fracción de detecciones informadas por el modelo que fueron correctas, mientras que la recuperación es la fracción de eventos reales que se detectaron. Un detector que diga que nadie tiene la enfermedad lograría una precisión perfecta, pero cero recuperación. Un detector que diga que todos tienen la enfermedad lograría un recuerdo perfecto, pero con una precisión igual al porcentaje de personas que tienen la enfermedad (0.0001% en nuestro ejemplo de una enfermedad que solo tiene una persona en un millón). Cuando se usa precisión y recuperación, es común trazar una **curva PR**, con precisión en el eje y y recuperación en el eje X. El clasificador genera una puntuación que es mayor si ocurrió el evento a detectar. Por ejemplo, un avance

red diseñada para detectar salidas de una enfermedad $\hat{y} = PAG(y=1/X)$, estimando la probabilidad de que una persona cuyos resultados médicos se describan por características X tiene la enfermedad. Elegimos informar una detección cada vez que esta puntuación supera algún umbral. Al variar el umbral, podemos intercambiar precisión por recuperación. En muchos casos, deseamos resumir el desempeño del clasificador con un solo número en lugar de una curva. Para hacerlo, podemos convertir la precisión pag recordar en una **Puntuación F**dada por

$$F = \frac{2\text{relaciones públicas}}{pag+r}. \quad (11.1)$$

Otra opción es reportar el área total que se encuentra debajo de la curva PR.

En algunas aplicaciones, es posible que el sistema de aprendizaje automático se niegue a tomar una decisión. Esto es útil cuando el algoritmo de aprendizaje automático puede estimar la confianza que debe tener sobre una decisión, especialmente si una decisión equivocada puede ser dañina y si un operador humano puede tomar el control ocasionalmente. El sistema de transcripción de Street View proporciona un ejemplo de esta situación. La tarea consiste en transcribir el número de dirección de una fotografía para asociar el lugar donde se tomó la foto con la dirección correcta en un mapa. Debido a que el valor del mapa se degrada considerablemente si el mapa es inexacto, es importante agregar una dirección solo si la transcripción es correcta. Si el sistema de aprendizaje automático piensa que es menos probable que un ser humano obtenga la transcripción correcta, entonces el mejor curso de acción es permitir que un humano transcriba la foto. Por supuesto, el sistema de aprendizaje automático solo es útil si puede reducir drásticamente la cantidad de fotos que los operadores humanos deben procesar. Una métrica de rendimiento natural para usar en esta situación **escobertura**. La cobertura es la fracción de ejemplos para los que el sistema de aprendizaje automático puede producir una respuesta. Es posible intercambiar cobertura por precisión. Siempre se puede obtener una precisión del 100 % negándose a procesar cualquier ejemplo, pero esto reduce la cobertura al 0 %. Para la tarea de Street View, el objetivo del proyecto era alcanzar una precisión de transcripción a nivel humano manteniendo una cobertura del 95 %. El rendimiento a nivel humano en esta tarea tiene una precisión del 98 %.

Muchas otras métricas son posibles. Podemos, por ejemplo, medir las tasas de clics, recopilar encuestas de satisfacción del usuario, etc. Muchas áreas de aplicación especializadas también tienen criterios específicos de aplicación.

Lo importante es determinar qué métrica de rendimiento mejorar con anticipación y luego concentrarse en mejorar esta métrica. Sin objetivos claramente definidos, puede ser difícil saber si los cambios en un sistema de aprendizaje automático progresan o no.

11.2 Modelos de referencia predeterminados

Después de elegir las métricas y los objetivos de rendimiento, el siguiente paso en cualquier aplicación práctica es establecer un sistema integral razonable lo antes posible. En esta sección, brindamos recomendaciones sobre qué algoritmos usar como el primer enfoque de referencia en diversas situaciones. Tenga en cuenta que la investigación de aprendizaje profundo avanza rápidamente, por lo que es probable que haya mejores algoritmos predeterminados disponibles poco después de escribir este artículo.

Dependiendo de la complejidad de su problema, es posible que desee comenzar sin utilizar el aprendizaje profundo. Si su problema tiene la posibilidad de resolverse simplemente eligiendo correctamente algunos pesos lineales, es posible que desee comenzar con un modelo estadístico simple como la regresión logística.

Si sabe que su problema entra en una categoría de "IA completa", como el reconocimiento de objetos, el reconocimiento de voz, la traducción automática, etc., es probable que le vaya bien si comienza con un modelo de aprendizaje profundo adecuado.

Primero, elija la categoría general de modelo según la estructura de sus datos. Si desea realizar un aprendizaje supervisado con vectores de tamaño fijo como entrada, utilice una red de avance con capas completamente conectadas. Si la entrada tiene una estructura topológica conocida (por ejemplo, si la entrada es una imagen), utilice una red convolucional. En estos casos, debe comenzar utilizando algún tipo de unidad lineal por partes (ReLU o sus generalizaciones como Leaky ReLU, PreLus y maxout). Si su entrada o salida es una secuencia, use una red recurrente cerrada (LSTM o GRU).

Una opción razonable de algoritmo de optimización es SGD con impulso con una tasa de aprendizaje decreciente (los esquemas de decaimiento populares que funcionan mejor o peor en diferentes problemas incluyen decaer linealmente hasta alcanzar una tasa de aprendizaje mínima fija, decaer exponencialmente o disminuir la tasa de aprendizaje por un factor de 2-10 cada vez que se estanca el error de validación). Otra alternativa muy razonable es Adam. La normalización por lotes puede tener un efecto dramático en el rendimiento de la optimización, especialmente para redes convolucionales y redes con no linealidades sigmoidales. Si bien es razonable omitir la normalización por lotes desde la primera línea de base, debe introducirse rápidamente si la optimización parece ser problemática.

A menos que su conjunto de entrenamiento contenga decenas de millones de ejemplos o más, debe incluir algunas formas leves de regularización desde el principio. La interrupción temprana debe usarse casi universalmente. Dropout es un excelente regularizador que es fácil de implementar y compatible con muchos modelos y algoritmos de entrenamiento. La normalización por lotes también reduce a veces el error de generalización y permite omitir la deserción, debido al ruido en la estimación de las estadísticas utilizadas para normalizar cada variable.

Si su tarea es similar a otra tarea que se ha estudiado extensamente, probablemente le irá bien copiando primero el modelo y el algoritmo que ya se sabe que funcionan mejor en la tarea previamente estudiada. Incluso es posible que desee copiar un modelo entrenado de esa tarea. Por ejemplo, es común usar las funciones de una red convolucional entrenada en ImageNet para resolver otras tareas de visión artificial ([Girshick et al., 2015](#)).

Una pregunta común es si comenzar usando el aprendizaje no supervisado, descrito más adelante en parte [tercero](#). Esto es algo específico del dominio. Se sabe que algunos dominios, como el procesamiento del lenguaje natural, se benefician enormemente de las técnicas de aprendizaje sin supervisión, como el aprendizaje de incrustaciones de palabras sin supervisión. En otros dominios, como la visión por computadora, las técnicas actuales de aprendizaje no supervisado no brindan ningún beneficio, excepto en el entorno semisupervisado, cuando el número de ejemplos etiquetados es muy pequeño ([Reyma et al., 2014; Rasmussen et al., 2015](#)). Si su aplicación se encuentra en un contexto en el que se sabe que el aprendizaje no supervisado es importante, inclúyalo en su primera línea de base integral. De lo contrario, solo use el aprendizaje no supervisado en su primer intento si la tarea que desea resolver no está supervisada. Siempre puede intentar agregar aprendizaje no supervisado más adelante si observa que su línea de base inicial se sobreajusta.

11.3 Determinar si recopilar más datos

Una vez establecido el primer sistema de extremo a extremo, es hora de medir el rendimiento del algoritmo y determinar cómo mejorarlo. Muchos novatos en aprendizaje automático se ven tentados a realizar mejoras probando muchos algoritmos diferentes. Sin embargo, a menudo es mucho mejor recopilar más datos que mejorar el algoritmo de aprendizaje.

¿Cómo se decide si recopilar más datos? Primero, determine si el desempeño en el conjunto de entrenamiento es aceptable. Si el rendimiento en el conjunto de entrenamiento es deficiente, el algoritmo de aprendizaje no está utilizando los datos de entrenamiento que ya están disponibles, por lo que no hay motivo para recopilar más datos. En su lugar, intente aumentar el tamaño del modelo agregando más capas o agregando más unidades ocultas a cada capa. Además, intente mejorar el algoritmo de aprendizaje, por ejemplo, ajustando el hiperparámetro de tasa de aprendizaje. Si los modelos grandes y los algoritmos de optimización cuidadosamente ajustados no funcionan bien, entonces el problema podría ser el *calidad* de los datos de entrenamiento. Los datos pueden tener demasiado ruido o pueden no incluir las entradas correctas necesarias para predecir las salidas deseadas. Esto sugiere comenzar de nuevo, recopilar datos más limpios o recopilar un conjunto más rico de características.

Si el rendimiento en el conjunto de entrenamiento es aceptable, mida el rendimiento

rendimiento en un conjunto de prueba. Si el rendimiento en el equipo de prueba también es aceptable, entonces no queda nada por hacer. Si el rendimiento del conjunto de prueba es mucho peor que el rendimiento del conjunto de entrenamiento, recopilar más datos es una de las soluciones más efectivas. Las consideraciones clave son el costo y la viabilidad de recopilar más datos, el costo y la viabilidad de reducir el error de prueba por otros medios y la cantidad de datos que se espera que sean necesarios para mejorar significativamente el rendimiento del conjunto de prueba. En las grandes empresas de Internet con millones o miles de millones de usuarios, es factible recopilar grandes conjuntos de datos y el costo de hacerlo puede ser considerablemente menor que el de otras alternativas, por lo que la respuesta casi siempre es recopilar más datos de capacitación. Por ejemplo, el desarrollo de grandes conjuntos de datos etiquetados fue uno de los factores más importantes para resolver el reconocimiento de objetos. En otros contextos, como las aplicaciones médicas, puede ser costoso o inviable recopilar más datos. Una alternativa simple a recopilar más datos es reducir el tamaño del modelo o mejorar la regularización, ajustando hiperparámetros como los coeficientes de disminución de peso o agregando estrategias de regularización como el abandono. Si encuentra que la brecha entre el entrenamiento y el rendimiento de la prueba sigue siendo inaceptable, incluso después de ajustar los hiperparámetros de regularización, es recomendable recopilar más datos. ajustando hiperparámetros, como los coeficientes de disminución de peso, o agregando estrategias de regularización, como la deserción. Si encuentra que la brecha entre el entrenamiento y el rendimiento de la prueba sigue siendo inaceptable, incluso después de ajustar los hiperparámetros de regularización, es recomendable recopilar más datos. ajustando hiperparámetros, como los coeficientes de disminución de peso, o agregando estrategias de regularización, como la deserción. Si encuentra que la brecha entre el entrenamiento y el rendimiento de la prueba sigue siendo inaceptable, incluso después de ajustar los hiperparámetros de regularización, es recomendable recopilar más datos.

Al decidir si recopilar más datos, también es necesario decidir cuánto recopilar. Es útil trazar curvas que muestren la relación entre el tamaño del conjunto de entrenamiento y el error de generalización, como en la figura 5.4. Al extrapolalar tales curvas, se puede predecir cuántos datos de entrenamiento adicionales se necesitarían para lograr un cierto nivel de rendimiento. Por lo general, agregar una pequeña fracción del número total de ejemplos no tendrá un impacto notable en el error de generalización. Por lo tanto, se recomienda experimentar con tamaños de conjuntos de entrenamiento en una escala logarítmica, por ejemplo, duplicando el número de ejemplos entre experimentos consecutivos.

Si no es factible recopilar muchos más datos, la única otra forma de mejorar el error de generalización es mejorar el propio algoritmo de aprendizaje. Esto se convierte en el dominio de la investigación y no el dominio del asesoramiento para los profesionales aplicados.

11.4 Selección de hiperparámetros

La mayoría de los algoritmos de aprendizaje profundo vienen con muchos hiperparámetros que controlan muchos aspectos del comportamiento del algoritmo. Algunos de estos hiperparámetros afectan el tiempo y el costo de memoria para ejecutar el algoritmo. Algunos de estos hiperparámetros afectan la calidad del modelo recuperado por el proceso de entrenamiento y su capacidad para inferir resultados correctos cuando se implementa en nuevas entradas.

Hay dos enfoques básicos para elegir estos hiperparámetros: elegirlos manualmente y elegirlos automáticamente. Elegir los hiperparámetros

manualmente requiere comprender qué hacen los hiperparámetros y cómo los modelos de aprendizaje automático logran una buena generalización. Los algoritmos automáticos de selección de hiperparámetros reducen en gran medida la necesidad de comprender estas ideas, pero a menudo son mucho más costosos desde el punto de vista computacional.

11.4.1 Ajuste manual de hiperparámetros

Para configurar los hiperparámetros manualmente, se debe comprender la relación entre los hiperparámetros, el error de entrenamiento, el error de generalización y los recursos computacionales (memoria y tiempo de ejecución). Esto significa establecer una base sólida sobre las ideas fundamentales sobre la capacidad efectiva de un algoritmo de aprendizaje del capítulo 5.

El objetivo de la búsqueda manual de hiperparámetros suele ser encontrar el error de generalización más bajo sujeto a algún tiempo de ejecución y presupuesto de memoria. No discutimos cómo determinar el tiempo de ejecución y el impacto en la memoria de varios hiperparámetros aquí porque esto depende en gran medida de la plataforma.

El objetivo principal de la búsqueda manual de hiperparámetros es ajustar la capacidad efectiva del modelo para que coincida con la complejidad de la tarea. La capacidad efectiva está restringida por tres factores: la capacidad de representación del modelo, la capacidad del algoritmo de aprendizaje para minimizar con éxito la función de costo utilizada para entrenar el modelo y el grado en que la función de costo y el procedimiento de entrenamiento regularizan el modelo. Un modelo con más capas y más unidades ocultas por capa tiene mayor capacidad de representación, es capaz de representar funciones más complicadas. Sin embargo, no necesariamente puede aprender todas estas funciones si el algoritmo de entrenamiento no puede descubrir que ciertas funciones hacen un buen trabajo al minimizar el costo de entrenamiento, o si los términos de regularización como la disminución del peso prohíben algunas de estas funciones.

El error de generalización normalmente sigue una curva en forma de U cuando se grafica como una función de uno de los hiperparámetros, como en la figura 5.3. En un extremo, el valor del hiperparámetro corresponde a baja capacidad y el error de generalización es alto porque el error de entrenamiento es alto. Este es el régimen de infravaloración. En el otro extremo, el valor del hiperparámetro corresponde a una capacidad alta y el error de generalización es alto porque la brecha entre el error de entrenamiento y el de prueba es alta. En algún punto intermedio se encuentra la capacidad óptima del modelo, que logra el error de generalización más bajo posible, al agregar una brecha de generalización media a una cantidad media de error de entrenamiento.

Para algunos hiperparámetros, el sobreajuste ocurre cuando el valor del hiperparámetro es grande. El número de unidades ocultas en una capa es un ejemplo de ello,

porque al aumentar el número de unidades ocultas aumenta la capacidad del modelo. Para algunos hiperparámetros, el sobreajuste ocurre cuando el valor del hiperparámetro es pequeño. Por ejemplo, el coeficiente de disminución de peso de cero permisible más pequeño corresponde a la mayor capacidad efectiva del algoritmo de aprendizaje.

No todos los hiperparámetros podrán explorar toda la curva en forma de U. Muchos hiperparámetros son discretos, como el número de unidades en una capa o el número de piezas lineales en una unidad máxima, por lo que solo es posible visitar algunos puntos a lo largo de la curva. Algunos hiperparámetros son binarios. Por lo general, estos hiperparámetros son interruptores que especifican si usar o no algún componente opcional del algoritmo de aprendizaje, como un paso de preprocessamiento que normaliza las características de entrada restando su media y dividiéndola por su desviación estándar. Estos hiperparámetros solo pueden explorar dos puntos de la curva. Otros hiperparámetros tienen algún valor mínimo o máximo que les impide explorar alguna parte de la curva. Por ejemplo, el coeficiente de caída de peso mínimo es cero. Esto significa que si el modelo no se ajusta cuando el decaimiento de peso es cero, no podemos ingresar a la región de sobreajuste modificando el coeficiente de decaimiento de peso. En otras palabras, algunos hiperparámetros solo pueden restar capacidad.

La tasa de aprendizaje es quizás el hiperparámetro más importante. tiene tiempo para ajustarse solo un hiperparámetro, ajustar la tasa de aprendizaje. controla la capacidad efectiva del modelo de una manera más complicada que otros hiperparámetros: la capacidad efectiva del modelo es más alta cuando la tasa de aprendizaje es correcta para el problema de optimización, no cuando la tasa de aprendizaje es especialmente grande o especialmente pequeña. La tasa de aprendizaje tiene una curva en forma de U para la ~~capacitación~~ error, ilustrado en la figura 11.1. Cuando la tasa de aprendizaje es demasiado grande, el descenso del gradiente puede aumentar inadvertidamente en lugar de disminuir el error de entrenamiento. En el caso cuadrático idealizado, esto ocurre si la tasa de aprendizaje es al menos dos veces mayor que su valor óptimo ([lecunet et al., 1998a](#)). Cuando la tasa de aprendizaje es demasiado pequeña, el entrenamiento no solo es más lento, sino que puede quedar atrapado permanentemente con un alto error de entrenamiento. Este efecto no se comprende bien (no sucedería con una función de pérdida convexa).

Ajustar los parámetros que no sean la tasa de aprendizaje requiere monitorear tanto el entrenamiento como el error de prueba para diagnosticar si su modelo se está ajustando demasiado o no, y luego ajustar su capacidad de manera adecuada.

Si su error en el conjunto de entrenamiento es mayor que su tasa de error objetivo, no tiene más remedio que aumentar la capacidad. Si no está utilizando la regularización y está seguro de que su algoritmo de optimización está funcionando correctamente, debe agregar más capas a su red o agregar más unidades ocultas. Desafortunadamente, esto aumenta los costos computacionales asociados con el modelo.

Si su error en el conjunto de prueba es mayor que su tasa de error objetivo, puede

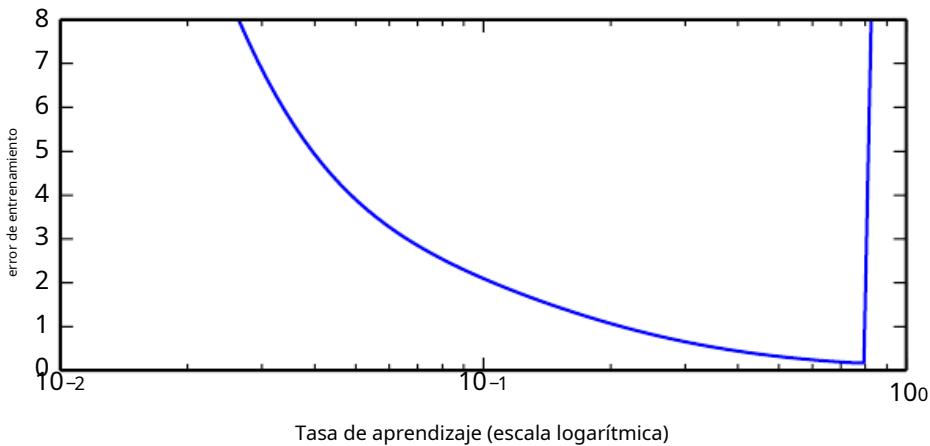


Figura 11.1: Relación típica entre la tasa de aprendizaje y el error de entrenamiento. Observe el fuerte aumento del error cuando el aprendizaje está por encima de un valor óptimo. Esto es para un tiempo de entrenamiento fijo, ya que una tasa de aprendizaje más pequeña a veces solo puede ralentizar el entrenamiento en un factor proporcional a la reducción de la tasa de aprendizaje. El error de generalización puede seguir esta curva o complicarse por los efectos de regularización que surgen de tener tasas de aprendizaje demasiado grandes o demasiado pequeñas, ya que una mala optimización puede, hasta cierto punto, reducir o evitar el sobreajuste, e incluso los puntos con un error de entrenamiento equivalente pueden tener una generalización diferente. error.

Ahora tome dos tipos de acciones. El error de prueba es la suma del error de entrenamiento y la brecha entre el error de entrenamiento y el de prueba. El error de prueba óptimo se encuentra intercambiando estas cantidades. Las redes neuronales suelen funcionar mejor cuando el error de entrenamiento es muy bajo (y, por lo tanto, cuando la capacidad es alta) y el error de prueba se debe principalmente a la brecha entre el error de entrenamiento y el de prueba. Su objetivo es reducir esta brecha sin aumentar el error de entrenamiento más rápido de lo que disminuye la brecha. Para reducir la brecha, cambie los hiperparámetros de regularización para reducir la capacidad efectiva del modelo, por ejemplo, agregando deserción o disminución de peso. Por lo general, el mejor rendimiento proviene de un modelo grande que está bien regularizado, por ejemplo, mediante el uso de la deserción.

La mayoría de los hiperparámetros se pueden establecer razonando si aumentan o disminuyen la capacidad del modelo. Algunos ejemplos se incluyen en la Tabla 11.1.

Mientras ajusta manualmente los hiperparámetros, no pierda de vista su objetivo final: buen rendimiento en el conjunto de prueba. Agregar la regularización es solo una forma de lograr este objetivo. Siempre que tenga un error de entrenamiento bajo, siempre puede reducir el error de generalización recopilando más datos de entrenamiento. La forma de fuerza bruta para garantizar prácticamente el éxito es aumentar continuamente la capacidad del modelo y el tamaño del conjunto de entrenamiento hasta que se resuelva la tarea. Este enfoque, por supuesto, aumenta el costo computacional del entrenamiento y la inferencia, por lo que solo es factible con los recursos apropiados. En

hiperparámetro	aumenta capacidad cuando...	Razón	Advertencias
Número de unidades de estudio aumentadas ocultas		Aumentar el número de unidades ocultas aumenta capacidad representativa del modelo.	Aumentar el número de unidades ocultas aumenta tanto el tiempo como el costo de la memoria de prácticamente todas las operaciones en el modelo.
Tasa de aprendizaje	sintonizado oportunamente	Una tasa de aprendizaje incorrecta, ya sea demasiado alta o demasiado baja, da como resultado un modelo con una capacidad efectiva baja debido a una falla en la	
Ker- de convolución ancho de canal	aumentó	optimización. Al aumentar el ancho del kernel, aumenta la cantidad de parámetros en el modelo.	Un kernel más amplio da como resultado una dimensión de salida más estrecha, lo que reduce la capacidad del modelo, a menos que utilice un relleno de ceros implícito para reducir este efecto. Los núcleos más amplios requieren más memoria para el almacenamiento de parámetros y aumentan el tiempo de ejecución, pero una salida más estrecha reduce el costo de la memoria.
Implícito cero relleno	aumentó	Agregar ceros implícitos antes de la convolución mantiene el tamaño de la representación	Mayor tiempo y costo de memoria de la mayoría de las operaciones.
Decaimiento de peso coeficiente	disminuido	grande Disminuir el coeficiente de caída de peso libera los parámetros del modelo para que sean más grandes	
Tasa de deserción escolar	disminuido	Dejar caer unidades con menos frecuencia les da a las unidades más oportunidades de "conspirar" entre sí para adaptarse al conjunto de entrenamiento	

Tabla 11.1: El efecto de varios hiperparámetros en la capacidad del modelo.

En principio, este enfoque podría fallar debido a las dificultades de optimización, pero para muchos problemas, la optimización no parece ser una barrera importante, siempre que el modelo se elija adecuadamente.

11.4.2 Algoritmos de optimización automática de hiperparámetros

El algoritmo de aprendizaje ideal solo toma un conjunto de datos y genera una función, sin necesidad de ajustar manualmente los hiperparámetros. La popularidad de varios algoritmos de aprendizaje, como la regresión logística y las SVM, se debe en parte a su capacidad para funcionar bien con solo uno o dos hiperparámetros sintonizados. Las redes neuronales a veces pueden funcionar bien con solo una pequeña cantidad de hiperparámetros ajustados, pero a menudo se benefician significativamente del ajuste de cuarenta o más hiperparámetros. El ajuste manual de hiperparámetros puede funcionar muy bien cuando el usuario tiene un buen punto de partida, como uno determinado por otros que han trabajado en el mismo tipo de aplicación y arquitectura, o cuando el usuario tiene meses o años de experiencia en la exploración de valores de hiperparámetros para redes neuronales. aplicado a tareas similares. Sin embargo, para muchas aplicaciones, estos puntos de partida no están disponibles. En estos casos, los algoritmos automatizados pueden encontrar valores útiles de los hiperparámetros.

Si pensamos en la forma en que el usuario de un algoritmo de aprendizaje busca buenos valores de los hiperparámetros, nos damos cuenta de que se está produciendo una optimización: estamos tratando de encontrar un valor de los hiperparámetros que optimice una función objetivo, como la validación. error, a veces bajo restricciones (como un presupuesto para tiempo de entrenamiento, memoria o tiempo de reconocimiento). Por lo tanto, es posible, en principio, desarrollar **optimización de hiperparámetros** algoritmos que envuelven un algoritmo de aprendizaje y eligen sus hiperparámetros, ocultando así los hiperparámetros del algoritmo de aprendizaje del usuario. Desafortunadamente, los algoritmos de optimización de hiperparámetros a menudo tienen sus propios hiperparámetros, como el rango de valores que deben explorarse para cada uno de los hiperparámetros del algoritmo de aprendizaje. Sin embargo, estos hiperparámetros secundarios suelen ser más fáciles de elegir, en el sentido de que se puede lograr un rendimiento aceptable en una amplia gama de tareas utilizando los mismos hiperparámetros secundarios para todas las tareas.

11.4.3 Búsqueda en cuadrícula

Cuando hay tres o menos hiperparámetros, la práctica común es realizar **búsqueda de cuadrícula**. Para cada hiperparámetro, el usuario selecciona un pequeño conjunto finito de valores para explorar. Luego, el algoritmo de búsqueda de cuadrícula entrena un modelo para cada especificación conjunta de valores de hiperparámetro en el producto cartesiano del conjunto de valores para cada hiperparámetro individual. El experimento que arroja la mejor validación

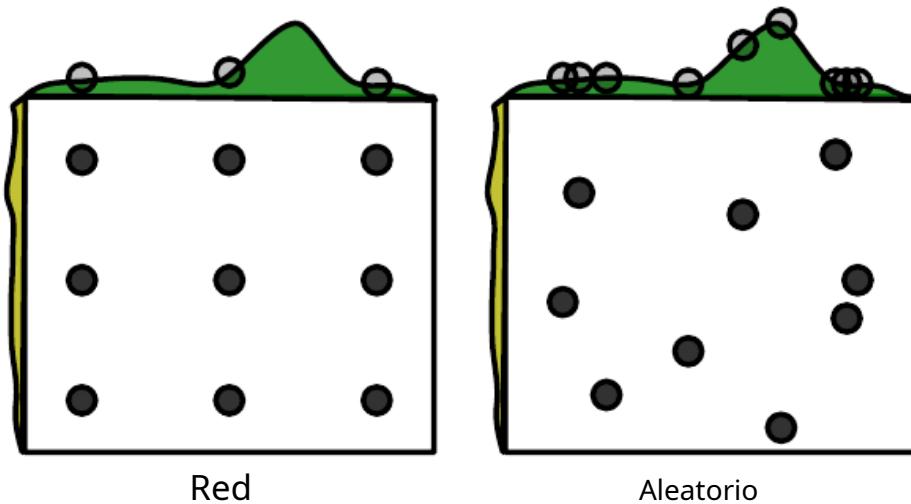


Figura 11.2: Comparación de búsqueda en cuadrícula y búsqueda aleatoria. Con fines ilustrativos, mostramos dos hiperparámetros, pero por lo general estamos interesados en tener muchos más./
*Izquierda)*Para realizar una búsqueda en cuadrícula, proporcionamos un conjunto de valores para cada hiperparámetro. El algoritmo de búsqueda ejecuta el entrenamiento para cada ajuste de hiperparámetro conjunto en el producto cruzado de estos conjuntos.(*Bien*)Para realizar una búsqueda aleatoria, proporcionamos una distribución de probabilidad sobre configuraciones conjuntas de hiperparámetros. Por lo general, la mayoría de estos hiperparámetros son independientes entre sí. Las opciones comunes para la distribución sobre un solo hiperparámetro incluyen uniforme y log-uniforme (para muestrear una distribución log-uniforme, tome la `Exp` de una muestra de una distribución uniforme). Luego, el algoritmo de búsqueda muestrea aleatoriamente configuraciones de hiperparámetros conjuntos y ejecuta el entrenamiento con cada uno de ellos. Tanto la búsqueda en cuadrícula como la búsqueda aleatoria evalúan el error del conjunto de validación y devuelven la mejor configuración. La figura ilustra el caso típico donde solo algunos hiperparámetros tienen una influencia significativa en el resultado. En esta ilustración, solo el hiperparámetro en el eje horizontal tiene un efecto significativo. La búsqueda en cuadrícula desperdicia una cantidad de cálculo que es exponencial en la cantidad de hiperparámetros no influyentes, mientras que la búsqueda aleatoria prueba un valor único de cada hiperparámetro influyente en casi todas las pruebas.
 Figura reproducida con permiso de [Bergstra y Bengio\(2012\)](#).

set error se elige entonces por haber encontrado los mejores hiperparámetros. Ver a la izquierda de la figura 11.2 para ver una ilustración de una cuadrícula de valores de hiperparámetros.

¿Cómo se deben elegir las listas de valores para buscar? En el caso de los hiperparámetros numéricos (ordenados), el elemento más pequeño y más grande de cada lista se elige de forma conservadora, en función de la experiencia previa con experimentos similares, para asegurarse de que es muy probable que el valor óptimo se encuentre en el rango seleccionado. Por lo general, una búsqueda en cuadrícula implica seleccionar valores aproximadamente en una escala logarítmica, por ejemplo, una tasa de aprendizaje tomada dentro del conjunto $\{1, 0.01, 10^{-3}, 10^{-4}, 10^{-5}\}$, o un número de unidades ocultas tomadas con el conjunto $\{50, 100, 200, 500, 1000, 2000\}$.

La búsqueda en cuadrícula generalmente funciona mejor cuando se realiza repetidamente. Por ejemplo, supongamos que ejecutamos una búsqueda de cuadrícula sobre un hiperparámetro utilizando valores de $\{-1, 0, 1\}$. Si el mejor valor encontrado es 1 , entonces subestimamos el rango en el que los mejores *a* mentiras y deberíamos cambiar la cuadrícula y ejecutar otra búsqueda con *a*, por ejemplo, $\{1, 2, 3\}$. Si encontramos que el mejor valor de *a* es 0 , entonces es posible que deseemos refinar nuestra estimación acercando y ejecutando una búsqueda de cuadrícula sobre $\{-1, 0, 1\}$.

El problema obvio con la búsqueda en cuadrícula es que su costo computacional crece exponencialmente con la cantidad de hiperparámetros. Si hay *m* hiperparámetros, cada uno tomando como máximo *n* valores, entonces el número de pruebas de entrenamiento y evaluación requeridas crece a medida que $\mathcal{O}(n^m)$. Las pruebas pueden ejecutarse en paralelo y explotar el paralelismo suelto (casi sin necesidad de comunicación entre las diferentes máquinas que realizan la búsqueda). Desafortunadamente, debido al costo exponencial de la búsqueda en cuadrícula, incluso la paralelización puede no proporcionar un tamaño de búsqueda satisfactorio.

11.4.4 Búsqueda aleatoria

Afortunadamente, existe una alternativa a la búsqueda en cuadrícula que es tan simple de programar, más conveniente de usar y converge mucho más rápido a buenos valores de los hiperparámetros: búsqueda aleatoria (Bergstra y Bengio, 2012).

Una búsqueda aleatoria procede de la siguiente manera. Primero definimos una distribución marginal para cada hiperparámetro, por ejemplo, un Bernoulli o multinomial para hiperparámetros binarios o discretos, o una distribución uniforme en una escala logarítmica para hiperparámetros de valor real positivo. Por ejemplo,

$$\text{log_learning_rate} \sim \text{tu}(-1, -5) \quad (11.2)$$

$$\text{tasa_de_aprendizaje} = 10 \text{log_learning_rate}. \quad (11.3)$$

dónde $\text{tu}(a, b)$ indica una muestra de la distribución uniforme en el intervalo (a, b) . Del mismo modo el `log_number_of_hidden_units` puede ser muestreado de $\text{tu}(\text{registro}(50), \text{registro}(2000))$.

A diferencia del caso de una búsqueda en cuadrícula, uno *no debe discretizar* los valores de los hiperparámetros. Esto permite explorar un conjunto más amplio de valores y no genera costos computacionales adicionales. De hecho, como se ilustra en la figura 11.2, una búsqueda aleatoria puede ser exponencialmente más eficiente que una búsqueda en cuadrícula, cuando hay varios hiperparámetros que no afectan fuertemente la medida de rendimiento. Esto se estudia extensamente en [Bergstra y Bengio\(2012\)](#), quienes encontraron que la búsqueda aleatoria reduce el error del conjunto de validación mucho más rápido que la búsqueda en cuadrícula, en términos del número de ensayos realizados por cada método.

Al igual que con la búsqueda en cuadrícula, es posible que a menudo desee ejecutar versiones repetidas de búsqueda aleatoria, para refinar la búsqueda en función de los resultados de la primera ejecución.

La razón principal por la que la búsqueda aleatoria encuentra buenas soluciones más rápido que la búsqueda en cuadrícula es que no hay corridas experimentales desperdiciadas, a diferencia del caso de la búsqueda en cuadrícula, cuando dos valores de un hiperparámetro (valores dados de los otros hiperparámetros) darían el mismo resultado. En el caso de la búsqueda en cuadrícula, los otros hiperparámetros tendrían los mismos valores para estas dos ejecuciones, mientras que con la búsqueda aleatoria, normalmente tendrían valores diferentes. Por lo tanto, si el cambio entre estos dos valores no hace mucha diferencia marginalmente en términos de error de conjunto de validación, la búsqueda en cuadrícula repetirá innecesariamente dos experimentos equivalentes, mientras que la búsqueda aleatoria aún brindará dos exploraciones independientes de los otros hiperparámetros.

11.4.5 Optimización de hiperparámetros basada en modelos

La búsqueda de buenos hiperparámetros se puede presentar como un problema de optimización. Las variables de decisión son los hiperparámetros. El costo a optimizar es el error del conjunto de validación que resulta del entrenamiento usando estos hiperparámetros. En configuraciones simplificadas donde es factible calcular el gradiente de alguna medida de error diferenciable en el conjunto de validación con respecto a los hiperparámetros, simplemente podemos seguir este gradiente ([bengio et al., 1999; bengio, 2000; Maclaurin et al., 2015](#)). Desafortunadamente, en la mayoría de los entornos prácticos, este gradiente no está disponible, ya sea debido a su alto costo de cómputo y memoria, o debido a que los hiperparámetros tienen interacciones intrínsecamente no diferenciables con el error del conjunto de validación, como en el caso de los hiperparámetros de valor discreto.

Para compensar esta falta de gradiente, podemos construir un modelo del error del conjunto de validación y luego proponer nuevas conjeturas de hiperparámetros realizando la optimización dentro de este modelo. La mayoría de los algoritmos basados en modelos para la búsqueda de hiperparámetros utilizan un modelo de regresión bayesiano para estimar tanto el valor esperado del error del conjunto de validación para cada hiperparámetro como la incertidumbre en torno a esta expectativa. Por lo tanto, la optimización implica un equilibrio entre la exploración (proponer hiperparámetros

para los cuales existe una gran incertidumbre, lo que puede conducir a una gran mejora pero también puede tener un desempeño deficiente) y explotación (proponer hiperparámetros que el modelo confía en que funcionarán tan bien como cualquier hiperparámetro que haya visto hasta ahora, generalmente hiperparámetros que son muy similares a que ha visto antes). Los enfoques contemporáneos para la optimización de hiperparámetros incluyen Spearmint ([snoek et al., 2012](#)), TPE ([Bergstra et al., 2011](#)) y SMAC ([Cazadore et al., 2011](#)).

Actualmente, no podemos recomendar sin ambigüedades la optimización de hiperparámetros bayesianos como una herramienta establecida para lograr mejores resultados de aprendizaje profundo o para obtener esos resultados con menos esfuerzo. La optimización de hiperparámetros bayesianos a veces funciona de manera comparable a los expertos humanos, a veces mejor, pero falla catastróficamente en otros problemas. Puede valer la pena intentar ver si funciona en un problema en particular, pero aún no es lo suficientemente maduro o confiable. Dicho esto, la optimización de hiperparámetros es un campo de investigación importante que, aunque a menudo está impulsado principalmente por las necesidades del aprendizaje profundo, tiene el potencial de beneficiar no solo a todo el campo del aprendizaje automático, sino también a la disciplina de la ingeniería en general.

Un inconveniente común a la mayoría de los algoritmos de optimización de hiperparámetros con más sofisticación que la búsqueda aleatoria es que requieren que se complete un experimento de entrenamiento antes de poder extraer cualquier información del experimento. Esto es mucho menos eficiente, en el sentido de cuánta información se puede obtener al principio de un experimento, que la búsqueda manual por parte de un médico humano, ya que normalmente se puede saber desde el principio si algún conjunto de hiperparámetros es completamente patológico. [swersky et al.\(2014\)](#) han introducido una versión temprana de un algoritmo que mantiene un conjunto de múltiples experimentos. En varios puntos de tiempo, el algoritmo de optimización de hiperparámetros puede optar por comenzar un nuevo experimento, "congelar" un experimento en ejecución que no es prometedor, o "descongelar" y reanudar un experimento que se congeló anteriormente pero ahora parece prometedor dada más información.

11.5 Estrategias de depuración

Cuando un sistema de aprendizaje automático funciona mal, generalmente es difícil saber si el bajo rendimiento es intrínseco al algoritmo en sí o si hay un error en la implementación del algoritmo. Los sistemas de aprendizaje automático son difíciles de depurar por una variedad de razones.

En la mayoría de los casos, no sabemos a priori cuál es el comportamiento previsto del algoritmo. De hecho, el objetivo de usar el aprendizaje automático es que descubrirá un comportamiento útil que no pudimos especificar nosotros mismos. Si entrenamos un

red neuronal en un *nuevo* tarea de clasificación y logra un error de prueba del 5%, no tenemos una forma sencilla de saber si este es el comportamiento esperado o el comportamiento subóptimo.

Otra dificultad es que la mayoría de los modelos de aprendizaje automático tienen varias partes que son adaptables. Si una parte se rompe, las otras partes se pueden adaptar y aun así lograr un rendimiento más o menos aceptable. Por ejemplo, supongamos que estamos entrenando una red neuronal con varias capas parametrizadas por pesos W y sesgos b . Supongamos además que hemos implementado manualmente la regla de descenso de gradiente para cada parámetro por separado y cometimos un error en la actualización de los sesgos:

$$b \leftarrow \text{segundo} - a \quad (11.4)$$

dónde a es la tasa de aprendizaje. Esta actualización errónea no utiliza el degradado en absoluto. Hace que los sesgos se vuelvan constantemente negativos durante el aprendizaje, lo que claramente no es una implementación correcta de ningún algoritmo de aprendizaje razonable. Sin embargo, es posible que el error no sea evidente solo al examinar la salida del modelo. Dependiendo de la distribución de la entrada, los pesos pueden adaptarse para compensar los sesgos negativos.

La mayoría de las estrategias de depuración para redes neuronales están diseñadas para sortear una o ambas de estas dos dificultades. O bien diseñamos un caso que sea tan simple que realmente se pueda predecir el comportamiento correcto, o diseñamos una prueba que ejerza una parte de la implementación de la red neuronal de forma aislada.

Algunas pruebas de depuración importantes incluyen:

Visualice el modelo en acción: Cuando entrene a un modelo para detectar objetos en imágenes, vea algunas imágenes con las detecciones propuestas por el modelo superpuestas a la imagen. Cuando entrene un modelo generativo de voz, escuche algunas de las muestras de voz que produce. Esto puede parecer obvio, pero es fácil caer en la práctica de observar únicamente medidas cuantitativas de rendimiento como la precisión o la probabilidad logarítmica. La observación directa del modelo de aprendizaje automático que realiza su tarea ayudará a determinar si los números de rendimiento cuantitativo que logra parecen razonables. Los errores de evaluación pueden ser algunos de los errores más devastadores porque pueden inducirlo a creer erróneamente que su sistema está funcionando bien cuando no es así.

Visualiza los peores errores: La mayoría de los modelos pueden generar algún tipo de medida de confianza para la tarea que realizan. Por ejemplo, los clasificadores basados en una capa de salida softmax asignan una probabilidad a cada clase. La probabilidad asignada a la clase más probable da así una estimación de la confianza que tiene el modelo en su decisión de clasificación. Por lo general, el entrenamiento de máxima verosimilitud da como resultado que estos valores sean sobreestimaciones en lugar de probabilidades precisas de predicción correcta.

pero son algo útiles en el sentido de que los ejemplos que en realidad tienen menos probabilidades de estar etiquetados correctamente reciben probabilidades más pequeñas según el modelo. Al ver los ejemplos de conjuntos de entrenamiento que son los más difíciles de modelar correctamente, a menudo se pueden descubrir problemas con la forma en que se han preprocesado o etiquetado los datos. Por ejemplo, el sistema de transcripción de Street View originalmente tenía un problema en el que el sistema de detección de número de dirección recortaba la imagen demasiado y omitía algunos de los dígitos. La red de transcripción luego asignó una probabilidad muy baja a la respuesta correcta en estas imágenes. Ordenar las imágenes para identificar los errores más confiables mostró que había un problema sistemático con el recorte. La modificación del sistema de detección para recortar imágenes mucho más anchas dio como resultado un rendimiento mucho mejor del sistema en general,

Razonamiento sobre el software que usa el tren y el error de prueba: A menudo es difícil determinar si el software subyacente está implementado correctamente. Se pueden obtener algunas pistas del error del tren y la prueba. Si el error de entrenamiento es bajo pero el error de prueba es alto, entonces es probable que el procedimiento de entrenamiento funcione correctamente y que el modelo esté sobreajustado por razones algorítmicas fundamentales. Una posibilidad alternativa es que el error de prueba se mida incorrectamente debido a un problema al guardar el modelo después del entrenamiento y luego volver a cargarlo para la evaluación del conjunto de prueba, o si los datos de prueba se prepararon de manera diferente a los datos de entrenamiento. Si tanto el error del tren como el de la prueba son altos, entonces es difícil determinar si hay un defecto de software o si el modelo no se ajusta correctamente debido a razones algorítmicas fundamentales. Este escenario requiere más pruebas, descritas a continuación.

Ajustar un pequeño conjunto de datos: Si tiene un alto error en el conjunto de entrenamiento, determine si se debe a una falta de ajuste genuina o a un defecto de software. Por lo general, incluso los modelos pequeños pueden garantizar que se ajusten a un conjunto de datos suficientemente pequeño. Por ejemplo, un conjunto de datos de clasificación con solo un ejemplo se puede ajustar simplemente configurando correctamente los sesgos de la capa de salida. Por lo general, si no puede entrenar un clasificador para etiquetar correctamente un solo ejemplo, un codificador automático para reproducir con éxito un solo ejemplo con alta fidelidad o un modelo generativo para emitir constantemente muestras que se asemejan a un solo ejemplo, hay un defecto de software que impide la optimización exitosa en el entrenamiento. colocar. Esta prueba se puede extender a un pequeño conjunto de datos con pocos ejemplos.

Comparar derivadas retropropagadas con derivadas numéricas: si está utilizando un marco de software que requiere que implemente sus propios cálculos de gradiente, o si está agregando una nueva operación a una biblioteca de diferenciación y debe definir subprop método, entonces una fuente común de error es implementar esta expresión de gradiente incorrectamente. Una forma de verificar que estas derivadas son correctas

es comparar las derivadas calculadas por su implementación de diferenciación automática con las derivadas calculadas por un**diferencias finitas**. Porque

$$F'(X) = \lim_{\Delta X \rightarrow 0} \frac{F(X + \Delta X) - F(X)}{\Delta X}, \quad (11.5)$$

podemos aproximar la derivada usando un pequeño, finito:-

$$F'(X) \approx \frac{F(X + \Delta X) - F(X)}{\Delta X}. \quad (11.6)$$

Podemos mejorar la precisión de la aproximación usando el**diferencia centrada**:

$$F'(X) \approx \frac{F(X + \Delta X) - F(X - \Delta X)}{2 \Delta X}. \quad (11.7)$$

El tamaño de la perturbación-debe elegirse para que sea lo suficientemente grande como para garantizar que la perturbación no se redondee demasiado por cálculos numéricos de precisión finita.

Por lo general, querremos probar el gradiente o el jacobiano de una función con valores vectoriales $\mathbf{f}(x)$: $\mathbf{R}_{n \times m} \rightarrow \mathbf{R}_{m \times n}$. Desafortunadamente, las diferencias finitas solo nos permiten tomar una única derivada a la vez. Podemos ejecutar diferencias finitas $\mathbf{M}_{n \times n}$ veces para evaluar todas las derivadas parciales de $\mathbf{f}(x)$, o podemos aplicar la prueba a una nueva función que usa proyecciones aleatorias tanto en la entrada como en la salida de $\mathbf{f}(x)$. Por ejemplo, podemos aplicar nuestra prueba de la implementación de las derivadas a $\mathbf{f}(x)$ donde $\mathbf{f}(x) = \mathbf{t} \circ \mathbf{g}(x)$, donde \mathbf{t} y \mathbf{g} son vectores elegidos aleatoriamente. Informática $\mathbf{f}(x)$ correctamente requiere ser capaz de retropropagarse a través de $\mathbf{g}(x)$ correctamente, pero es eficiente hacerlo con diferencias finitas porque \mathbf{f} tiene una sola entrada y una sola salida. Por lo general, es una buena idea repetir esta prueba para más de un valor de \mathbf{t} para reducir la posibilidad de que la prueba pase por alto errores que son ortogonales a la proyección aleatoria.

Si uno tiene acceso al cálculo numérico de números complejos, entonces hay una forma muy eficiente de estimar numéricamente el gradiente usando números complejos como entrada a la función ([Escudero y Trapp, 1998](#)). El método se basa en la observación de que

$$\mathbf{f}(x + i\Delta) = \mathbf{f}(x) + i\mathbf{f}'(x)\Delta + O(\Delta^2) \quad (11.8)$$

$$\text{real}(\mathbf{f}(x + i\Delta)) = \mathbf{f}(x) + O(\Delta^2), \text{Imagen}\left(\frac{\mathbf{f}(x + i\Delta)}{\sqrt{i}}\right) = \mathbf{f}'(x)\Delta + O(\Delta^2), \quad (11.9)$$

dónde $i = \sqrt{-1}$. A diferencia del caso de valor real anterior, no hay efecto de cancelación debido a tomar la diferencia entre el valor de \mathbf{f} en diferentes puntos. Esto permite el uso de pequeños valores de Δ como $= 10^{-150}$, que hacen que el error insignificante a todos los efectos prácticos.

Monitorear histogramas de activaciones y gradiente: A menudo es útil visualizar estadísticas de activaciones y gradientes de redes neuronales, recopiladas durante una gran cantidad de iteraciones de entrenamiento (quizás una época). El valor de activación previa de las unidades ocultas puede deciros si las unidades se saturan o con qué frecuencia lo hacen. Por ejemplo, para los rectificadores, ¿con qué frecuencia están apagados? ¿Hay unidades que siempre están apagadas? Para las unidades tanh, el promedio del valor absoluto de las preactivaciones nos dice qué tan saturada está la unidad. En una red profunda donde los gradientes propagados crecen o desaparecen rápidamente, la optimización puede verse obstaculizada. Finalmente, es útil comparar la magnitud de los gradientes de los parámetros con la magnitud de los propios parámetros. Como lo sugiere [fondo\(2015\)](#), nos gustaría que la magnitud de las actualizaciones de parámetros en un minilote represente algo así como el 1 % de la magnitud del parámetro, no el 50 % ni el 0,001 % (lo que haría que los parámetros se movieran con demasiada lentitud). Puede ser que algunos grupos de parámetros se muevan a buen ritmo mientras que otros están estancados. Cuando los datos son escasos (como en el lenguaje natural), algunos parámetros pueden actualizarse muy raramente, y esto debe tenerse en cuenta al monitorear su evolución.

Finalmente, muchos algoritmos de aprendizaje profundo brindan algún tipo de garantía sobre los resultados producidos en cada paso. Por ejemplo, en parte [tercero](#), veremos algunos algoritmos de inferencia aproximada que funcionan usando soluciones algebraicas a problemas de optimización. Por lo general, estos se pueden depurar probando cada una de sus garantías. Algunas garantías que ofrecen algunos algoritmos de optimización incluyen que la función objetivo nunca aumentará después de un paso del algoritmo, que el gradiente con respecto a algún subconjunto de variables será cero después de cada paso del algoritmo y que el gradiente con respecto a todos las variables serán cero en la convergencia. Por lo general, debido a un error de redondeo, estas condiciones no se mantendrán exactamente en una computadora digital, por lo que la prueba de depuración debe incluir algún parámetro de tolerancia.

11.6 Ejemplo: reconocimiento de números de varios dígitos

Para brindar una descripción completa de cómo aplicar nuestra metodología de diseño en la práctica, presentamos una breve descripción del sistema de transcripción de Street View, desde el punto de vista del diseño de los componentes de aprendizaje profundo.

Obviamente, muchos otros componentes del sistema completo, como los autos de Street View, la infraestructura de la base de datos, etc., fueron de suma importancia.

Desde el punto de vista de la tarea de aprendizaje automático, el proceso comenzó con la recopilación de datos. Los autos recopilaron los datos sin procesar y los operadores humanos proporcionaron etiquetas. La tarea de transcripción estuvo precedida por una cantidad significativa de selección de conjuntos de datos, incluido el uso de otras técnicas de aprendizaje automático para detectar la casa

números antes de transcribirlos.

El proyecto de transcripción comenzó con una selección de métricas de rendimiento y valores deseados para estas métricas. Un principio general importante es adaptar la elección de la métrica a los objetivos comerciales del proyecto. Debido a que los mapas solo son útiles si tienen una alta precisión, era importante establecer un requisito de alta precisión para este proyecto. Específicamente, el objetivo era obtener una precisión del 98 % a nivel humano. Este nivel de precisión puede no ser siempre factible de obtener. Para alcanzar este nivel de precisión, el sistema de transcripción de Street View sacrifica la cobertura. La cobertura se convirtió así en la principal métrica de rendimiento optimizada durante el proyecto, con una precisión del 98 %. A medida que mejoró la red convolucional, se hizo posible reducir el umbral de confianza por debajo del cual la red se niega a transcribir la entrada,

Después de elegir objetivos cuantitativos, el siguiente paso en nuestra metodología recomendada es establecer rápidamente un sistema de línea de base sensato. Para tareas de visión, esto significa una red convolucional con unidades lineales rectificadas. El proyecto de transcripción comenzó con un modelo de este tipo. En ese momento, no era común que una red convolucional generara una secuencia de predicciones. Para comenzar con la línea de base más simple posible, la primera implementación de la capa de salida del modelo consistió en *n* diferentes unidades softmax para predecir una secuencia de *n* caracteres. Estas unidades softmax se entrenaron exactamente igual que si la tarea fuera clasificación, con cada unidad softmax entrenada de forma independiente.

Nuestra metodología recomendada es refinar iterativamente la línea de base y probar si cada cambio supone una mejora. El primer cambio en el sistema de transcripción de Street View estuvo motivado por una comprensión teórica de la métrica de cobertura y la estructura de los datos. Específicamente, la red se niega a clasificar una entrada x siempre que la probabilidad de la secuencia de salida $p(y | x)$ por algún umbral t . Inicialmente, la definición de $p(y | x)$ fue ad-hoc, basado en simplemente multiplicar todas las salidas de softmax juntas. Esto motivó el desarrollo de una capa de salida especializada y una función de costo que en realidad calculaba una probabilidad logarítmica basada en principios. Este enfoque permitió que el mecanismo de rechazo del ejemplo funcionara de manera mucho más efectiva.

En este punto, la cobertura todavía estaba por debajo del 90%, pero no había problemas teóricos obvios con el enfoque. Por lo tanto, nuestra metodología sugiere instrumentar el rendimiento del tren y del conjunto de prueba para determinar si el problema es un ajuste insuficiente o excesivo. En este caso, el error del tren y del conjunto de prueba fue casi idéntico. De hecho, la razón principal por la que este proyecto se llevó a cabo sin problemas fue la disponibilidad de un conjunto de datos con decenas de millones de ejemplos etiquetados. Debido a que el error del tren y del conjunto de prueba eran tan similares, esto sugirió que el problema se debió a

por falta de ajuste o por un problema con los datos de entrenamiento. Una de las estrategias de depuración que recomendamos es visualizar los peores errores del modelo. En este caso, eso significó visualizar las transcripciones incorrectas del conjunto de entrenamiento que el modelo dio la mayor confianza. Estos demostraron consistir principalmente en ejemplos en los que la imagen de entrada se había recortado demasiado, con algunos de los dígitos de la dirección eliminados por la operación de recorte. Por ejemplo, una foto de una dirección "1849" puede estar demasiado recortada y solo queda visible el "849". Este problema podría haberse resuelto dedicando semanas a mejorar la precisión del sistema de detección de números de dirección responsable de determinar las regiones de cultivo. En cambio, el equipo tomó una decisión mucho más práctica, simplemente expandir el ancho de la región de recorte para que sea sistemáticamente más ancho de lo que predijo el sistema de detección de número de dirección. Este único cambio agregó diez puntos porcentuales a la cobertura del sistema de transcripción.

Finalmente, los últimos puntos porcentuales de rendimiento provinieron del ajuste de los hiperparámetros. Esto consistió principalmente en hacer el modelo más grande manteniendo algunas restricciones en su costo computacional. Debido a que el error de entrenamiento y prueba se mantuvo aproximadamente igual, siempre estuvo claro que cualquier déficit de rendimiento se debía a un ajuste insuficiente, así como a algunos problemas restantes con el propio conjunto de datos.

En general, el proyecto de transcripción fue un gran éxito y permitió que cientos de millones de direcciones se transcribieran más rápido y a un menor costo de lo que hubiera sido posible mediante el esfuerzo humano.

Esperamos que los principios de diseño descritos en este capítulo conduzcan a muchos otros éxitos similares.

Capítulo 12

Aplicaciones

En este capítulo, describimos cómo usar el aprendizaje profundo para resolver aplicaciones en visión por computadora, reconocimiento de voz, procesamiento de lenguaje natural y otras áreas de aplicación de interés comercial. Comenzamos discutiendo las implementaciones de redes neuronales a gran escala requeridas para las aplicaciones de IA más serias. A continuación, revisamos varias áreas de aplicación específicas para las que se ha utilizado el aprendizaje profundo. Si bien uno de los objetivos del aprendizaje profundo es diseñar algoritmos que sean capaces de resolver una amplia variedad de tareas, hasta ahora se necesita cierto grado de especialización. Por ejemplo, las tareas de visión requieren procesar una gran cantidad de características de entrada (píxeles) por ejemplo. Las tareas de lenguaje requieren modelar una gran cantidad de valores posibles (palabras en el vocabulario) por característica de entrada.

12.1 Aprendizaje profundo a gran escala

El aprendizaje profundo se basa en la filosofía del conexionismo: mientras que una neurona biológica individual o una característica individual en un modelo de aprendizaje automático no es inteligente, una gran población de estas neuronas o características que actúan juntas pueden exhibir un comportamiento inteligente. Realmente es importante enfatizar el hecho de que el número de neuronas debe ser *grande*. Uno de los factores clave responsables de la mejora en la precisión de las redes neuronales y la mejora de la complejidad de las tareas que pueden resolver entre la década de 1980 y la actualidad es el aumento espectacular del tamaño de las redes que utilizamos. Como vimos en la sección 1.2.3, los tamaños de las redes han crecido exponencialmente durante las últimas tres décadas, pero las redes neuronales artificiales son tan grandes como los sistemas nerviosos de los insectos.

Debido a que el tamaño de las redes neuronales es de suma importancia, el aprendizaje profundo

requiere una infraestructura de hardware y software de alto rendimiento.

12.1.1 Implementaciones de CPU rápidas

Tradicionalmente, las redes neuronales se entrenaban utilizando la CPU de una sola máquina. Hoy en día, este enfoque se considera generalmente insuficiente. Ahora usamos principalmente la computación GPU o las CPU de muchas máquinas conectadas en red. Antes de pasar a estas costosas configuraciones, los investigadores trabajaron arduamente para demostrar que las CPU no podían administrar la alta carga de trabajo computacional requerida por las redes neuronales.

Una descripción de cómo implementar un código de CPU numérico eficiente está más allá del alcance de este libro, pero enfatizamos aquí que la implementación cuidadosa para familias de CPU específicas puede generar grandes mejoras. Por ejemplo, en 2011, las mejores CPU disponibles podían ejecutar cargas de trabajo de redes neuronales más rápido cuando usaban aritmética de punto fijo en lugar de aritmética de punto flotante. Al crear una implementación de punto fijo cuidadosamente ajustada, [Vanhoucke et al. \(2011\)](#) obtuvo una aceleración triple sobre un fuerte sistema de punto flotante. Cada nuevo modelo de CPU tiene diferentes características de rendimiento, por lo que, a veces, las implementaciones de punto flotante también pueden ser más rápidas. El principio importante es que la especialización cuidadosa de las rutinas de cálculo numérico puede generar grandes beneficios. Otras estrategias, además de elegir si usar punto fijo o flotante, incluyen la optimización de estructuras de datos para evitar errores de caché y el uso de instrucciones vectoriales. Muchos investigadores de aprendizaje automático descuidan estos detalles de implementación, pero cuando el rendimiento de una implementación restringe el tamaño del modelo, la precisión del modelo se ve afectada.

12.1.2 Implementaciones de GPU

La mayoría de las implementaciones modernas de redes neuronales se basan en unidades de procesamiento de gráficos. Las unidades de procesamiento de gráficos (GPU) son componentes de hardware especializados que se desarrollaron originalmente para aplicaciones de gráficos. El mercado de consumo de sistemas de videojuegos impulsó el desarrollo de hardware de procesamiento de gráficos. Las características de rendimiento necesarias para los buenos sistemas de videojuegos también resultan beneficiosas para las redes neuronales.

La renderización de videojuegos requiere realizar muchas operaciones en paralelo rápidamente. Los modelos de personajes y entornos se especifican en términos de listas de coordenadas tridimensionales de vértices. Las tarjetas gráficas deben realizar multiplicaciones y divisiones de matrices en muchos vértices en paralelo para convertir estas coordenadas 3D en coordenadas 2D en pantalla. La tarjeta gráfica debe realizar muchos cálculos en cada píxel en paralelo para determinar el color de cada píxel. En ambos casos, el

los cálculos son bastante simples y no implican muchas ramificaciones en comparación con la carga de trabajo computacional que normalmente encuentra una CPU. Por ejemplo, cada vértice del mismo objeto rígido se multiplicará por la misma matriz; no hay necesidad de evaluar una instrucción if por vértice para determinar por qué matriz multiplicar. Los cálculos también son completamente independientes entre sí y, por lo tanto, se pueden paralelizar fácilmente. Los cálculos también implican el procesamiento de búferes masivos de memoria, que contienen mapas de bits que describen la textura (patrón de color) de cada objeto que se representará. En conjunto, esto da como resultado que las tarjetas gráficas se hayan diseñado para tener un alto grado de paralelismo y un alto ancho de banda de memoria, a costa de tener una velocidad de reloj más baja y menos capacidad de bifurcación en relación con las CPU tradicionales.

Los algoritmos de redes neuronales requieren las mismas características de rendimiento que los algoritmos de gráficos en tiempo real descritos anteriormente. Las redes neuronales generalmente involucran grandes y numerosos búferes de parámetros, valores de activación y valores de gradiente, cada uno de los cuales debe actualizarse por completo durante cada paso del entrenamiento. Estos búferes son lo suficientemente grandes como para quedar fuera del caché de una computadora de escritorio tradicional, por lo que el ancho de banda de la memoria del sistema a menudo se convierte en el factor limitante de la velocidad. Las GPU ofrecen una ventaja convincente sobre las CPU debido a su alto ancho de banda de memoria. Los algoritmos de entrenamiento de redes neuronales normalmente no implican muchas ramificaciones ni un control sofisticado, por lo que son apropiados para el hardware de GPU. Dado que las redes neuronales se pueden dividir en múltiples "neuronas" individuales que se pueden procesar independientemente de las otras neuronas en la misma capa,

El hardware de GPU originalmente era tan especializado que solo podía usarse para tareas gráficas. Con el tiempo, el hardware de la GPU se volvió más flexible, lo que permitió usar subrutinas personalizadas para transformar las coordenadas de los vértices o asignar colores a los píxeles. En principio, no había ningún requisito de que estos valores de píxel se basaran realmente en una tarea de renderizado. Estas GPU podrían usarse para computación científica al escribir la salida de una computación en un búfer de valores de píxeles. [Steinkrau et al. \(2005\)](#) implementaron una red neuronal totalmente conectada de dos capas en una GPU e informaron una aceleración tres veces superior a su línea de base basada en CPU. Poco después, [Chellapilla et al. \(2006\)](#) demostraron que la misma técnica podría usarse para acelerar redes convolucionales supervisadas.

La popularidad de las tarjetas gráficas para el entrenamiento de redes neuronales se disparó después de la llegada de **GPU de propósito general**. Estas GP-GPU podrían ejecutar código arbitrario, no solo renderizar subrutinas. El lenguaje de programación CUDA de NVIDIA proporcionó una forma de escribir este código arbitrario en un lenguaje similar a C. Con su modelo de programación relativamente conveniente, paralelismo masivo y memoria alta

ancho de banda, las GP-GPU ahora ofrecen una plataforma ideal para la programación de redes neuronales. Los investigadores de aprendizaje profundo adoptaron rápidamente esta plataforma poco después de que estuvo disponible (Iluvia *et al.*, 2009; Ciresán *et al.*, 2010).

Escribir código eficiente para GP-GPU sigue siendo una tarea difícil que es mejor dejarla en manos de especialistas. Las técnicas requeridas para obtener un buen rendimiento en la GPU son muy diferentes de las que se utilizan en la CPU. Por ejemplo, un buen código basado en CPU generalmente está diseñado para leer la mayor cantidad posible de información del caché. En GPU, la mayoría de las ubicaciones de memoria de escritura no se almacenan en caché, por lo que en realidad puede ser más rápido calcular el mismo valor dos veces, en lugar de calcularlo una vez y leerlo de la memoria. El código de la GPU también es inherentemente de subprocesos múltiples y los diferentes subprocesos deben coordinarse entre sí con cuidado. Por ejemplo, las operaciones de memoria son más rápidas si pueden ser **fusionados**. Las lecturas o escrituras combinadas ocurren cuando varios subprocesos pueden leer o escribir un valor que necesitan simultáneamente, como parte de una sola transacción de memoria. Los diferentes modelos de GPU pueden fusionar diferentes tipos de patrones de lectura o escritura. Por lo general, las operaciones de memoria son más fáciles de fusionar si *entre* *norte* hilos, hilo/byte de acceso/+de la memoria, y es un múltiplo de alguna potencia de 2. Las especificaciones exactas difieren entre los modelos de GPU. Otra consideración común para las GPU es asegurarse de que cada subproceso de un grupo ejecute la misma instrucción simultáneamente. Esto significa que la ramificación puede ser difícil en la GPU. Los hilos se dividen en pequeños grupos llamados **deformaciones**. Cada subproceso en un warp ejecuta la misma instrucción durante cada ciclo, por lo que si diferentes subprocesos dentro del mismo warp necesitan ejecutar diferentes rutas de código, estas diferentes rutas de código deben atravesarse secuencialmente en lugar de en paralelo.

Debido a la dificultad de escribir código GPU de alto rendimiento, los investigadores deben estructurar su flujo de trabajo para evitar tener que escribir código GPU nuevo para probar nuevos modelos o algoritmos. Por lo general, se puede hacer esto construyendo una biblioteca de software de operaciones de alto rendimiento como convolución y multiplicación de matrices, y luego especificando modelos en términos de llamadas a esta biblioteca de operaciones. Por ejemplo, la biblioteca de aprendizaje automático Pylearn2 (Buen compañero *et al.*, 2013c) especifica todos sus algoritmos de aprendizaje automático en términos de llamadas a Theano (Bergstra *et al.*, 2010; bastié *et al.*, 2012) y cuda-convnet (Krizhevski, 2010), que proporcionan estas operaciones de alto rendimiento. Este enfoque factorizado también puede facilitar el soporte para múltiples tipos de hardware. Por ejemplo, el mismo programa Theano puede ejecutarse en CPU o GPU, sin necesidad de cambiar ninguna de las llamadas a Theano. Otras bibliotecas como TensorFlow (Abadi *et al.*, 2015) y Antorcha (coloberto *et al.*, 2011b) proporcionan características similares.

12.1.3 Implementaciones distribuidas a gran escala

En muchos casos, los recursos computacionales disponibles en una sola máquina son insuficientes. Por lo tanto, queremos distribuir la carga de trabajo de entrenamiento e inferencia entre muchas máquinas.

Distribuir la inferencia es simple, porque cada ejemplo de entrada que queremos procesar puede ejecutarse en una máquina separada. Esto se conoce como **paralelismo de datos**.

También es posible conseguir **modelos de paralelismo**, donde varias máquinas trabajan juntas en un solo punto de datos, y cada máquina ejecuta una parte diferente del modelo. Esto es factible tanto para la inferencia como para el entrenamiento.

El paralelismo de datos durante el entrenamiento es algo más difícil. Podemos aumentar el tamaño del minilote utilizado para un solo paso SGD, pero generalmente obtenemos rendimientos menos que lineales en términos de rendimiento de optimización. Sería mejor permitir que varias máquinas calculen varios pasos de descenso de gradiente en paralelo. Desafortunadamente, la definición estándar de descenso de gradiente es como un algoritmo completamente secuencial: el gradiente en el paso t es una función de los parámetros producidos por el paso $t-1$.

Esto se puede resolver usando **descenso de gradiente estocástico asíncrono** ([bengio et al., 2001](#); [rech et al., 2011](#)). En este enfoque, varios núcleos de procesador comparten la memoria que representa los parámetros. Cada núcleo lee los parámetros sin bloqueo, luego calcula un gradiente y luego incrementa los parámetros sin bloqueo. Esto reduce la cantidad promedio de mejora que produce cada paso de descenso de gradiente, porque algunos de los núcleos sobrescriben el progreso de los demás, pero la mayor tasa de producción de pasos hace que el proceso de aprendizaje sea más rápido en general. [Decano et al. \(2012\)](#) fue pionera en la implementación de múltiples máquinas de este enfoque sin bloqueo para el descenso de gradiente, donde los parámetros son administrados por **un servidor de parámetros** en lugar de almacenarse en la memoria compartida. El descenso de gradiente asíncrono distribuido sigue siendo la estrategia principal para entrenar grandes redes profundas y es utilizado por la mayoría de los principales grupos de aprendizaje profundo en la industria ([Chilimbi et al., 2014](#); [Wu et al., 2015](#)). Los investigadores académicos de aprendizaje profundo generalmente no pueden permitirse la misma escala de sistemas de aprendizaje distribuido, pero algunas investigaciones se han centrado en cómo construir redes distribuidas con hardware de costo relativamente bajo disponible en el entorno universitario ([Coaté et al., 2013](#)).

12.1.4 Compresión del modelo

En muchas aplicaciones comerciales, es mucho más importante que el tiempo y el costo de memoria de ejecutar la inferencia en un modelo de aprendizaje automático sean bajos que el tiempo y el costo de memoria del entrenamiento. Para aplicaciones que no requieren

personalización, es posible entrenar un modelo una vez y luego implementarlo para que lo utilicen miles de millones de usuarios. En muchos casos, el usuario final tiene más recursos limitados que el desarrollador. Por ejemplo, uno podría entrenar una red de reconocimiento de voz con un poderoso grupo de computadoras y luego implementarla en teléfonos móviles.

Una estrategia clave para reducir el costo de la inferencia es **modelo de compresión**(Buciluă *et al.*,2006). La idea básica de la compresión de modelos es reemplazar el modelo original y costoso con un modelo más pequeño que requiere menos memoria y tiempo de ejecución para almacenar y evaluar.

La compresión del modelo es aplicable cuando el tamaño del modelo original se debe principalmente a la necesidad de evitar el sobreajuste. En la mayoría de los casos, el modelo con el error de generalización más bajo es un conjunto de varios modelos entrenados de forma independiente. evaluando todo *n* miembros del conjunto es caro. A veces, incluso un solo modelo generaliza mejor si es grande (por ejemplo, si se regulariza con deserción).

Estos modelos grandes aprenden alguna función $F(X)$, pero hágalo utilizando muchos más parámetros de los necesarios para la tarea. Su tamaño es necesario solo debido al número limitado de ejemplos de entrenamiento. Tan pronto como hayamos ajustado esta función $F(X)$, podemos generar un conjunto de entrenamiento que contenga infinitos ejemplos, simplemente aplicando F a puntos muestreados aleatoriamente X . Luego entrenamos el modelo nuevo, más pequeño, para que coincida $F(X)$ sobre estos puntos. Para usar la capacidad del nuevo modelo pequeño de la manera más eficiente, es mejor probar el nuevo X puntos de una distribución que se asemeja a las entradas de prueba reales que se suministrarán al modelo más adelante. Esto se puede hacer corrompiendo ejemplos de entrenamiento o dibujando puntos de un modelo generativo entrenado en el conjunto de entrenamiento original.

Alternativamente, uno puede entrenar el modelo más pequeño solo en los puntos de entrenamiento originales, pero entrenarlo para copiar otras características del modelo, como su distribución posterior sobre las clases incorrectas (Hinton *et al.*,2014,2015).

12.1.5 Estructura dinámica

Una estrategia para acelerar los sistemas de procesamiento de datos en general es construir sistemas que tengan **estructura dinámica**en el gráfico que describe el cálculo necesario para procesar una entrada. Los sistemas de procesamiento de datos pueden determinar dinámicamente qué subconjunto de muchas redes neuronales debe ejecutarse en una entrada determinada. Las redes neuronales individuales también pueden exhibir una estructura dinámica internamente al determinar qué subconjunto de características (unidades ocultas) para calcular la información dada de la entrada. Esta forma de estructura dinámica dentro de las redes neuronales a veces se denominac **computación condicional**(bengio,2013;bengio *et al.*,2013b). Dado que muchos componentes de la arquitectura pueden ser relevantes solo para una pequeña cantidad de posibles entradas, el

el sistema puede funcionar más rápido calculando estas características solo cuando son necesarias.

La estructura dinámica de los cómputos es un principio básico de las ciencias de la computación que se aplica generalmente en toda la disciplina de la ingeniería de software. Las versiones más simples de la estructura dinámica aplicada a las redes neuronales se basan en determinar qué subconjunto de algún grupo de redes neuronales (u otros modelos de aprendizaje automático) se debe aplicar a una entrada en particular.

Una estrategia venerable para acelerar la inferencia en un clasificador es usar una **cascada** de clasificadores. La estrategia en cascada se puede aplicar cuando el objetivo es detectar la presencia de un objeto (o evento) raro. Para saber con certeza que el objeto está presente, debemos usar un clasificador sofisticado con alta capacidad, que es costoso de ejecutar. Sin embargo, debido a que el objeto es raro, generalmente podemos usar muchos menos cálculos para rechazar las entradas que no contienen el objeto. En estas situaciones, podemos entrenar una secuencia de clasificadores. Los primeros clasificadores de la secuencia tienen poca capacidad y están entrenados para tener un alto recuerdo. En otras palabras, están capacitados para asegurarse de que no rechacemos incorrectamente una entrada cuando el objeto está presente. El clasificador final está entrenado para tener una alta precisión. En el momento de la prueba, ejecutamos la inferencia ejecutando los clasificadores en una secuencia, abandonando cualquier ejemplo tan pronto como un elemento en la cascada lo rechace. En general, esto nos permite verificar la presencia de objetos con alta confianza, utilizando un modelo de alta capacidad, pero no nos obliga a pagar el costo de la inferencia completa para cada ejemplo. Hay dos formas diferentes en que la cascada puede lograr una alta capacidad. Una forma es hacer que los últimos miembros de la cascada tengan individualmente una alta capacidad. En este caso, el sistema en su conjunto obviamente tiene una gran capacidad, porque algunos de sus miembros individuales la tienen. También es posible hacer una cascada en la que cada modelo individual tenga una capacidad baja pero el sistema en su conjunto tenga una capacidad alta debido a la combinación de muchos modelos pequeños. Una forma es hacer que los últimos miembros de la cascada tengan individualmente una gran capacidad. En este caso, el sistema en su conjunto obviamente tiene una gran capacidad, porque algunos de sus miembros individuales la tienen. También es posible hacer una cascada en la que cada modelo individual tenga una capacidad baja pero el sistema en su conjunto tenga una capacidad alta debido a la combinación de muchos modelos pequeños. [viola y jones\(2001\)](#) usó una cascada de árboles de decisión mejorados para implementar un detector de rostros rápido y robusto adecuado para su uso en cámaras digitales portátiles. Su clasificador localiza una cara utilizando esencialmente un enfoque de ventana deslizante en el que se examinan y rechazan muchas ventanas si no contienen caras. Otra versión de cascadas utiliza los modelos anteriores para implementar una especie de mecanismo de atención estricta: los primeros miembros de la cascada localizan un objeto y los miembros posteriores de la cascada realizan un procesamiento adicional dada la ubicación del objeto. Por ejemplo, Google transcribe los números de dirección de las imágenes de Street View mediante una cascada de dos pasos que primero localiza el número de dirección con un modelo de aprendizaje automático y luego lo transcribe con otro ([Buen compañero et al., 2014d](#)).

Los propios árboles de decisión son un ejemplo de estructura dinámica, porque cada nodo del árbol determina cuál de sus subárboles debe evaluarse para cada entrada. Una forma sencilla de lograr la unión del aprendizaje profundo y la estructura dinámica

es entrenar un árbol de decisión en el que cada nodo utiliza una red neuronal para tomar la decisión de división (Guo y Gelfand, 1992), aunque esto normalmente no se ha hecho con el objetivo principal de acelerar los cálculos de inferencia.

Con el mismo espíritu, se puede utilizar una red neuronal, llamada **puerta** para seleccionar cuál de varios **redes de expertos** se utilizará para calcular la salida, dada la entrada actual. La primera versión de esta idea se llamó **mezcla de expertos** (Nowlan, 1990; Jacobson et al., 1991), en el que el gater genera un conjunto de probabilidades o pesos (obtenidos a través de una no linealidad softmax), uno por experto, y el resultado final se obtiene mediante la combinación ponderada de los resultados de los expertos. En ese caso, el uso del gater no ofrece una reducción en el costo computacional, pero si el gater elige un solo experto para cada ejemplo, obtenemos el **dura mezcla de expertos** (Colloberto et al., 2001, 2002), lo que puede acelerar considerablemente el tiempo de entrenamiento e inferencia. Esta estrategia funciona bien cuando el número de decisiones de activación es pequeño porque no es combinatoria. Pero cuando queremos seleccionar diferentes subconjuntos de unidades o parámetros, no es posible usar un "interruptor suave" porque requiere enumerar (y calcular salidas para) todas las configuraciones de gater. Para hacer frente a este problema, se han explorado varios enfoques para entrenar gaters combinatorios. Bengio et al. (2013b) experimentan con varios estimadores del gradiente de las probabilidades de activación, mientras que Tocino et al. (2015) y Bengio et al. (2015a) usan técnicas de aprendizaje por refuerzo (gradientes de política) para aprender una forma de abandono condicional en bloques de unidades ocultas y obtener una reducción real en el costo computacional sin impactar negativamente en la calidad de la aproximación.

Otro tipo de estructura dinámica es un interruptor, donde una unidad oculta puede recibir información de diferentes unidades según el contexto. Este enfoque de enrutamiento dinámico puede interpretarse como un mecanismo de atención (Olshausen et al., 1993). Hasta ahora, el uso de un interruptor duro no ha demostrado ser efectivo en aplicaciones a gran escala. En cambio, los enfoques contemporáneos utilizan un promedio ponderado sobre muchas entradas posibles y, por lo tanto, no logran todos los posibles beneficios computacionales de la estructura dinámica. Los mecanismos de atención contemporáneos se describen en la sección 12.4.5.1.

Un obstáculo importante para el uso de sistemas estructurados dinámicamente es la disminución del grado de paralelismo que resulta del sistema que sigue diferentes ramas de código para diferentes entradas. Esto significa que pocas operaciones en la red pueden describirse como multiplicación de matrices o convolución por lotes en un minilote de ejemplos. Podemos escribir subrutinas más especializadas que convolucionen cada ejemplo con diferentes núcleos o multipliquen cada fila de una matriz de diseño por un conjunto diferente de columnas de pesos. Desafortunadamente, estas subrutinas más especializadas son difíciles de implementar de manera eficiente. Las implementaciones de CPU serán lentas debido a la falta de caché.

Las implementaciones de coherencia y GPU serán lentas debido a la falta de transacciones de memoria fusionadas y la necesidad de serializar warps cuando los miembros de un warp toman diferentes ramas. En algunos casos, estos problemas se pueden mitigar dividiendo los ejemplos en grupos que toman la misma rama y procesando estos grupos de ejemplos simultáneamente. Esta puede ser una estrategia aceptable para minimizar el tiempo necesario para procesar una cantidad fija de ejemplos en un entorno fuera de línea. En una configuración en tiempo real donde los ejemplos deben procesarse continuamente, la partición de la carga de trabajo puede generar problemas de equilibrio de carga. Por ejemplo, si asignamos una máquina para procesar el primer paso en cascada y otra máquina para procesar el último paso en cascada, entonces la primera tenderá a estar sobrecargada y la última tenderá a estar subcargada.

12.1.6 Implementaciones de hardware especializado de redes profundas

Desde los primeros días de la investigación de redes neuronales, los diseñadores de hardware han trabajado en implementaciones de hardware especializadas que podrían acelerar el entrenamiento y/o la inferencia de algoritmos de redes neuronales. Consulte revisiones anteriores y más recientes de hardware especializado para redes profundas ([Lindsey y Lindblad, 1994](#); [Beiú et al., 2003](#); [misra y saha,2010](#)).

Diferentes formas de hardware especializado ([Graf y Jackel, 1989](#); [Mead e Ismail, 2012](#); [Kim et al., 2009](#); [Phamet et al., 2012](#); [Chenet et al., 2014a,b](#)) se han desarrollado en las últimas décadas, ya sea con ASIC (circuito integrado de aplicación específica), ya sea digital (basado en representaciones binarias de números), analógico ([Graf y Jackel, 1989](#); [Mead e Ismail, 2012](#)) (basado en implementaciones físicas de valores continuos como voltajes o corrientes) o implementaciones híbridas (combinando componentes digitales y analógicos). En los últimos años se han desarrollado implementaciones más flexibles de FPGA (matriz cerrada programable en campo) (en las que los detalles del circuito se pueden escribir en el chip una vez construido).

Aunque las implementaciones de software en unidades de procesamiento de uso general (CPU y GPU) suelen utilizar 32 o 64 bits de precisión para representar números de punto flotante, se sabe desde hace tiempo que era posible utilizar menos precisión, al menos en el momento de la inferencia ([holt y panadero,1991](#); [Holi y Hwang,1993](#); [Presley y Haggard,1994](#); [Simard y Graf,1994](#); [Wawrzyneket al.,1996](#); [Sávichet et al., 2007](#)). Este se ha convertido en un problema más apremiante en los últimos años, ya que el aprendizaje profundo ha ganado popularidad en los productos industriales y se ha demostrado el gran impacto de un hardware más rápido con las GPU. Otro factor que motiva la investigación actual sobre hardware especializado para redes profundas es que la tasa de progreso de un solo núcleo de CPU o GPU se ha ralentizado y las mejoras más recientes en

la velocidad de cómputo proviene de la paralelización entre núcleos (ya sea en CPU o GPU). Esto es muy diferente de la situación de la década de 1990 (la era anterior de las redes neuronales) donde las implementaciones de hardware de las redes neuronales (que podrían demorar dos años desde el inicio hasta la disponibilidad de un chip) no podían seguir el ritmo del rápido progreso y los bajos precios de las redes neuronales. CPU de uso general. La creación de hardware especializado es, por lo tanto, una forma de ir más allá, en un momento en que se están desarrollando nuevos diseños de hardware para dispositivos de bajo consumo, como teléfonos, con el objetivo de aplicaciones de aprendizaje profundo para el público en general (por ejemplo, con voz, visión por computadora o lenguaje natural).

Trabajo reciente sobre implementaciones de baja precisión de redes neuronales basadas en backprop ([Vanhoucke et al., 2011](#); [Courbariaux et al., 2015](#); [Gupta et al., 2015](#)) sugiere que entre 8 y 16 bits de precisión pueden ser suficientes para usar o entrenar redes neuronales profundas con retropropagación. Lo que está claro es que se requiere más precisión durante el entrenamiento que en el momento de la inferencia, y que se pueden usar algunas formas de representación dinámica de números en punto fijo para reducir la cantidad de bits necesarios por número. Los números de punto fijo tradicionales están restringidos a un rango fijo (que corresponde a un exponente dado en una representación de punto flotante). Las representaciones dinámicas de punto fijo comparten ese rango entre un conjunto de números (como todos los pesos en una capa). El uso de representaciones de punto fijo en lugar de punto flotante y el uso de menos bits por número reduce el área de superficie del hardware, los requisitos de energía y el tiempo de cálculo necesario para realizar multiplicaciones.

12.2 Visión artificial

La visión por computadora ha sido tradicionalmente una de las áreas de investigación más activas para las aplicaciones de aprendizaje profundo, porque la visión es una tarea que no requiere esfuerzo para los humanos y muchos animales, pero que es un desafío para las computadoras ([Ballard et al., 1983](#)). Muchas de las tareas de referencia estándar más populares para los algoritmos de aprendizaje profundo son formas de reconocimiento de objetos o reconocimiento óptico de caracteres.

La visión artificial es un campo muy amplio que abarca una amplia variedad de formas de procesar imágenes y una asombrosa diversidad de aplicaciones. Las aplicaciones de la visión por computadora van desde la reproducción de las habilidades visuales humanas, como el reconocimiento de rostros, hasta la creación de categorías completamente nuevas de habilidades visuales. Como ejemplo de la última categoría, una aplicación reciente de visión por computadora es reconocer las ondas de sonido a partir de las vibraciones que inducen en los objetos visibles en un video ([Davíset al., 2014](#)). La mayoría de las investigaciones de aprendizaje profundo sobre visión por computadora no se han centrado en tales

aplicaciones exóticas que amplían el ámbito de lo que es posible con imágenes, pero más bien un pequeño núcleo de objetivos de IA destinados a replicar las habilidades humanas. La mayor parte del aprendizaje profundo para la visión por computadora se usa para el reconocimiento o detección de objetos de alguna forma, ya sea que esto signifique informar qué objeto está presente en una imagen, anotar una imagen con cuadros delimitadores alrededor de cada objeto, transcribir una secuencia de símbolos de una imagen o etiquetar cada píxel de una imagen con la identidad del objeto al que pertenece. Debido a que el modelado generativo ha sido un principio rector de la investigación del aprendizaje profundo, también hay una gran cantidad de trabajo sobre la síntesis de imágenes utilizando modelos profundos. Mientras que la síntesis de imágenes *ex nihilo* generalmente no se considera un esfuerzo de visión por computadora, los modelos capaces de síntesis de imágenes suelen ser útiles para la restauración de imágenes, una tarea de visión por computadora que implica reparar defectos en imágenes o eliminar objetos de imágenes.

12.2.1 Preprocesamiento

Muchas áreas de aplicación requieren un preprocesamiento sofisticado porque la entrada original viene en una forma que es difícil de representar para muchas arquitecturas de aprendizaje profundo. La visión por computadora generalmente requiere relativamente poco de este tipo de preprocesamiento. Las imágenes deben estandarizarse para que todos sus píxeles se encuentren en el mismo rango razonable, como [0,1] o [-1, 1]. Mezclar imágenes que se encuentran en [0,1] con imágenes que se encuentran en [0, 255] generalmente resultará en fallas. Dar formato a las imágenes para que tengan la misma escala es el único tipo de preprocesamiento que es estrictamente necesario. Muchas arquitecturas de visión por computadora requieren imágenes de un tamaño estándar, por lo que las imágenes deben recortarse o escalarse para que se ajusten a ese tamaño. Incluso este cambio de escala no siempre es estrictamente necesario.[waibel et al., 1989](#)). Otros modelos convolucionales tienen una salida de tamaño variable que escala automáticamente en tamaño con la entrada, como los modelos que eliminan el ruido o etiquetan cada píxel de una imagen ([Hadsell et al., 2007](#)).

El aumento del conjunto de datos puede verse como una forma de preprocesar solo el conjunto de entrenamiento. El aumento de conjuntos de datos es una excelente manera de reducir el error de generalización de la mayoría de los modelos de visión artificial. Una idea relacionada aplicable en el momento de la prueba es mostrar al modelo muchas versiones diferentes de la misma entrada (por ejemplo, la misma imagen recortada en ubicaciones ligeramente diferentes) y hacer que las diferentes instancias del modelo voten para determinar la salida. Esta última idea puede interpretarse como un enfoque conjunto y ayuda a reducir el error de generalización.

Se aplican otros tipos de preprocesamiento tanto al tren como al conjunto de prueba con el objetivo de poner cada ejemplo en una forma más canónica para reducir la cantidad de variación que el modelo necesita tener en cuenta. Reduciendo la cantidad de

la variación en los datos puede reducir el error de generalización y reducir el tamaño del modelo necesario para ajustarse al conjunto de entrenamiento. Las tareas más simples pueden resolverse con modelos más pequeños, y es más probable que las soluciones más simples se generalicen bien. El preprocesamiento de este tipo generalmente está diseñado para eliminar algún tipo de variabilidad en los datos de entrada que es fácil de describir para un diseñador humano y que el diseñador humano confía en que no tiene relevancia para la tarea. Cuando se entrena con grandes conjuntos de datos y modelos grandes, este tipo de preprocesamiento a menudo es innecesario y es mejor dejar que el modelo aprenda a qué tipos de variabilidad debe volverse invariable. Por ejemplo, el sistema AlexNet para clasificar ImageNet solo tiene un paso de preprocesamiento: restar la media entre los ejemplos de entrenamiento de cada píxel ([Krizhevski et al., 2012](#)).

12.2.1.1 Normalización de contraste

Una de las fuentes de variación más obvias que se pueden eliminar de manera segura para muchas tareas es la cantidad de contraste en la imagen. El contraste simplemente se refiere a la magnitud de la diferencia entre los píxeles brillantes y oscuros en una imagen. Hay muchas formas de cuantificar el contraste de una imagen. En el contexto del aprendizaje profundo, el contraste generalmente se refiere a la desviación estándar de los píxeles en una imagen o región de una imagen. Supongamos que tenemos una imagen representada por un tensor $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$, con $X_{yo, j, 1}$ siendo la intensidad roja en la fila y y columna j , $X_{yo, j, 2}$ dando al verde intensidad y $X_{yo, j, 3}$ dando intensidad al azul. Entonces el contraste de toda la imagen viene dado por

$$\text{Contraste} = \sqrt{\frac{1}{3rc} \sum_{i=1, j=1, k=1}^r (X_{yo, j, k} - \bar{\mathbf{X}})^2} \quad (12.1)$$

dónde $\bar{\mathbf{X}}$ es la intensidad media de toda la imagen:

$$\bar{\mathbf{X}} = \frac{1}{3rc} \sum_{i=1, j=1, k=1}^r X_{yo, j, k}. \quad (12.2)$$

Normalización de contraste global(GCN) tiene como objetivo evitar que las imágenes tengan cantidades variables de contraste al restar la media de cada imagen y luego volver a escalarla para que la desviación estándar en sus píxeles sea igual a alguna constante. Este enfoque se complica por el hecho de que ningún factor de escala puede cambiar el contraste de una imagen de contraste cero (una cuyos píxeles tienen todos la misma intensidad). Las imágenes con un contraste muy bajo pero distinto de cero a menudo tienen poco contenido de información. Dividir por la desviación estándar verdadera generalmente no logra nada

más que amplificar el ruido del sensor o los artefactos de compresión en tales casos. Esto motiva la introducción de un pequeño parámetro de regularización positivo λ para sesgar la estimación de la desviación estándar. Alternativamente, uno puede restringir el denominador para que sea al menos -. Dada una imagen de entrada \mathbf{X} , GCN produce una imagen de salida \mathbf{X}' , definida de tal manera que

$$\mathbf{X}'_{yo, j, k=s} = \frac{\mathbf{X}_{yo, j, k} - \bar{\mathbf{X}}}{\sqrt{\frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^s (\mathbf{X}_{yo, j, k} - \bar{\mathbf{X}})^2}} \quad (12.3)$$

Es poco probable que los conjuntos de datos que consisten en imágenes grandes recortadas en objetos interesantes contengan imágenes con una intensidad casi constante. En estos casos, es seguro ignorar prácticamente el problema del pequeño denominador estableciendo λ=0 y evitar la división por 0 en casos extremadamente raros configurando a un valor extremadamente bajo como 10^-8. Este es el enfoque utilizado por [Buen compañero et al.](#)(2013a) en el conjunto de datos CIFAR-10. Es más probable que las imágenes pequeñas recortadas aleatoriamente tengan una intensidad casi constante, lo que hace que la regularización agresiva sea más útil. [Coatéset al.](#)(2011) usado λ=0 y λ=10 en pequeños parches seleccionados al azar extraídos de CIFAR-10.

El parámetro de escala por lo general se puede configurar para 1, como hecho por [Coatéset al.](#)(2011), o elegido para hacer que cada píxel individual tenga una desviación estándar en los ejemplos cercana a 1, como lo hace [Buen compañero et al.](#)(2013a).

La desviación estándar en la ecuación 12.3 es solo una reescala de la L_2 norma de la imagen (asumiendo que la media de la imagen ya ha sido eliminada). Es preferible definir GCN en términos de desviación estándar en lugar de L_2 norma porque la desviación estándar incluye la división por el número de píxeles, por lo que GCN basado en la desviación estándar permite la misma para ser utilizado independientemente del tamaño de la imagen. Sin embargo, la observación de que la L_2 norma es proporcional a la desviación estándar puede ayudar a construir una intuición útil. Uno puede entender GCN como ejemplos de mapeo a una capa esférica. Ver figura 12.1 para una ilustración. Esta puede ser una propiedad útil porque las redes neuronales a menudo responden mejor a direcciones en el espacio que a ubicaciones exactas. Responder a múltiples distancias en la misma dirección requiere unidades ocultas con vectores de peso colineales pero diferentes sesgos. Tal coordinación puede ser difícil de descubrir para el algoritmo de aprendizaje. Además, muchos modelos gráficos poco profundos tienen problemas para representar múltiples modos separados a lo largo de la misma línea. GCN evita estos problemas al reducir cada ejemplo a una dirección en lugar de una dirección y una distancia.

Contrariamente a la intuición, hay una operación de preprocessamiento conocida como **esferas** y no es la misma operación que GCN. Sphering no se refiere a hacer que los datos se encuentren en una capa esférica, sino a reescalar los componentes principales para tener

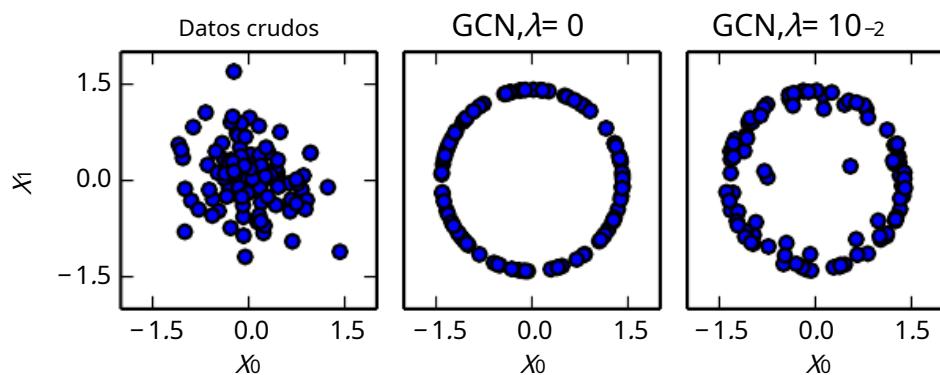


Figura 12.1: GCN mapea ejemplos en una esfera. (Izquierda) Los datos de entrada sin procesar pueden tener cualquier norma. (Centro) GCN con $\lambda=0$ asigna todos los ejemplos distintos de cero perfectamente en una esfera. Aquí usamos $s=1$ y $y=10^{-8}$. Debido a que usamos GCN basado en la normalización de la desviación estándar en lugar de la L_2 norma, la esfera resultante no es la esfera unitaria. (Derecha) GCN regularizado, con $\lambda>0$, dibuja ejemplos hacia la esfera pero no descarta por completo la variación en su norma. Salimossy lo mismo de antes.

igual varianza, de modo que la distribución normal multivariante utilizada por PCA tiene contornos esféricos. La esfera es más comúnmente conocida como **blanqueo**.

La normalización del contraste global a menudo fallará en resaltar las características de la imagen que nos gustaría destacar, como los bordes y las esquinas. Si tenemos una escena con un área oscura grande y un área brillante grande (como la plaza de una ciudad con la mitad de la imagen a la sombra de un edificio), la normalización del contraste global garantizará que haya una gran diferencia entre el brillo del área oscura y el brillo del área de luz. Sin embargo, no garantizará que se destaque los bordes dentro de la región oscura.

esto motiva **normalización del contraste local**. La normalización del contraste local garantiza que el contraste se normalice en cada ventana pequeña, en lugar de en la imagen como un todo. Ver figura 12.2 para una comparación de la normalización de contraste global y local.

Son posibles varias definiciones de normalización de contraste local. En todos los casos, se modifica cada píxel restando la media de los píxeles cercanos y dividiendo por una desviación estándar de los píxeles cercanos. En algunos casos, esto es literalmente la media y la desviación estándar de todos los píxeles en una ventana rectangular centrada en el píxel que se va a modificar ([Caballo pinto et al., 2008](#)). En otros casos, se trata de una media ponderada y una desviación estándar ponderada utilizando pesos gaussianos centrados en el píxel que se va a modificar. En el caso de las imágenes en color, algunas estrategias procesan diferentes colores.

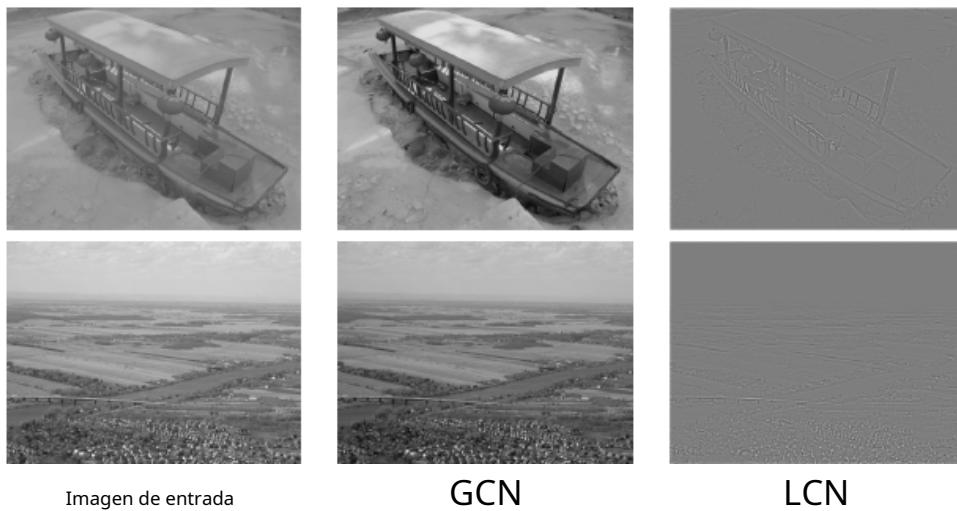


Figura 12.2: Una comparación de normalización de contraste global y local. Visualmente, los efectos de la normalización del contraste global son sutiles. Coloca todas las imágenes aproximadamente en la misma escala, lo que reduce la carga del algoritmo de aprendizaje para manejar múltiples escalas. La normalización del contraste local modifica mucho más la imagen, descartando todas las regiones de intensidad constante. Esto permite que el modelo se centre solo en los bordes. Las regiones de textura fina, como las casas de la segunda fila, pueden perder algunos detalles debido a que el ancho de banda del kernel de normalización es demasiado alto.

canales por separado, mientras que otros combinan información de diferentes canales para normalizar cada píxel ([Sermanet et al., 2012](#)).

La normalización del contraste local generalmente se puede implementar de manera eficiente mediante el uso de convolución separable (consulte la sección 9.8) para calcular mapas de características de medias locales y desviaciones estándar locales, y luego usar la resta por elementos y la división por elementos en diferentes mapas de características.

La normalización de contraste local es una operación diferenciable y también se puede utilizar como una no linealidad aplicada a las capas ocultas de una red, así como una operación de preprocessamiento aplicada a la entrada.

Al igual que con la normalización de contraste global, normalmente necesitamos regularizar la normalización de contraste local para evitar la división por cero. De hecho, debido a que la normalización del contraste local normalmente actúa en ventanas más pequeñas, es aún más importante regularizar. Las ventanas más pequeñas tienen más probabilidades de contener valores que son casi iguales entre sí y, por lo tanto, es más probable que tengan una desviación estándar cero.

12.2.1.2 Aumento de conjuntos de datos

Como se describe en la sección 7.4, es fácil mejorar la generalización de un clasificador aumentando el tamaño del conjunto de entrenamiento agregando copias adicionales de los ejemplos de entrenamiento que han sido modificados con transformaciones que no cambian la clase. El reconocimiento de objetos es una tarea de clasificación que es especialmente adecuada para esta forma de aumento de conjuntos de datos porque la clase es invariable a tantas transformaciones y la entrada se puede transformar fácilmente con muchas operaciones geométricas. Como se describió anteriormente, los clasificadores pueden beneficiarse de traducciones aleatorias, rotaciones y, en algunos casos, cambios de la entrada para aumentar el conjunto de datos. En aplicaciones especializadas de visión por computadora, las transformaciones más avanzadas se usan comúnmente para el aumento de conjuntos de datos. Estos esquemas incluyen la perturbación aleatoria de los colores en una imagen (Krizhevskiet al., 2012) y distorsiones geométricas no lineales de la entrada (lecunet al., 1998b).

12.3 Reconocimiento de voz

La tarea del reconocimiento de voz es mapear una señal acústica que contiene una expresión hablada en lenguaje natural en la secuencia correspondiente de palabras previstas por el hablante. Dejar $X = (X_1, X_2, \dots, X_n)$ indican la secuencia de vectores de entrada acústica (tradicionalmente producidos al dividir el audio en cuadros de 20 ms). La mayoría de los sistemas de reconocimiento de voz preprocesan la entrada utilizando funciones especializadas diseñadas a mano, pero algunos (Jaitly y Hinton, 2011) los sistemas de aprendizaje profundo aprenden características a partir de entradas sin procesar. Dejar $y = (y_1, y_2, \dots, y_n)$ indican la secuencia de salida de destino (generalmente una secuencia de palabras o caracteres). El **reconocimiento automático de voz** (ASR) tarea consiste en crear una función F_{ASR} que calcula el lingüístico más probable secuencia dada la secuencia acústica X :

$$F_{ASR}(X) = \underset{y}{\text{argmax}} PAG(y | X = X)$$
 (12.4)

dónde PAG es la verdadera distribución condicional que relaciona las entradas X a los objetivos y .

Desde la década de 1980 y hasta aproximadamente 2009-2012, los sistemas de reconocimiento de voz de última generación combinaron principalmente modelos ocultos de Markov (HMM) y modelos de mezcla gaussiana (GMM). Los GMM modelaron la asociación entre características acústicas y fonemas (Bahlet al., 1987), mientras que los HMM modelaron la secuencia de fonemas. La familia de modelos GMM-HMM trata las formas de onda acústicas como generadas por el siguiente proceso: primero, un HMM genera una secuencia de fonemas y estados subfonémicos discretos (como el comienzo, el medio y el final de cada

fonema), luego un GMM transforma cada símbolo discreto en un breve segmento de forma de onda de audio. Aunque los sistemas GMM-HMM dominaron ASR hasta hace poco, el reconocimiento de voz fue en realidad una de las primeras áreas en las que se aplicaron las redes neuronales, y numerosos sistemas ASR de finales de los 80 y principios de los 90 utilizaron redes neuronales (Bourlard y Wellekens, 1989; waibelet al., 1989; Robinson y Fallside, 1991; bengioet al., 1991, 1992; Königet al., 1996). En ese momento, el rendimiento de ASR basado en redes neuronales coincidía aproximadamente con el rendimiento de los sistemas GMM-HMM. Por ejemplo, Robinson y Fallside(1991) logró una tasa de error de fonema del 26 % en el TIMIT (Garófoloet al., 1993) corpus (con 39 fonemas para discriminar), que era mejor o comparable a los sistemas basados en HMM. Desde entonces, TIMIT ha sido un punto de referencia para el reconocimiento de fonemas, desempeñando un papel similar al que desempeña MNIST para el reconocimiento de objetos. Sin embargo, debido a la compleja ingeniería involucrada en los sistemas de software para el reconocimiento de voz y el esfuerzo que se había invertido en construir estos sistemas sobre la base de GMM-HMM, la industria no vio un argumento convincente para cambiar a redes neuronales. Como consecuencia, hasta finales de la década de 2000, la investigación académica e industrial sobre el uso de redes neuronales para el reconocimiento de voz se centró principalmente en el uso de redes neuronales para aprender funciones adicionales para los sistemas GMM-HMM.

Más tarde, con *modelos mucho más grandes y profundos* y conjuntos de datos mucho más grandes, la precisión del reconocimiento mejoró drásticamente mediante el uso de redes neuronales para reemplazar los GMM para la tarea de asociar características acústicas a fonemas (o estados subfonémicos). A partir de 2009, los investigadores del habla aplicaron una forma de aprendizaje profundo basada en el aprendizaje no supervisado al reconocimiento del habla. Este enfoque de aprendizaje profundo se basó en el entrenamiento de modelos probabilísticos no dirigidos llamados máquinas de Boltzmann restringidas (RBM) para modelar los datos de entrada. Los RBM se describirán en parte **tercero**. Para resolver las tareas de reconocimiento de voz, se utilizó un preentrenamiento no supervisado para construir redes de avance profundo cuyas capas se inicializaron entrenando un RBM. Estas redes toman representaciones acústicas espectrales en una ventana de entrada de tamaño fijo (alrededor de un marco central) y predicen las probabilidades condicionales de los estados HMM para ese marco central. El entrenamiento de redes tan profundas ayudó a mejorar significativamente la tasa de reconocimiento en TIMIT (mohamedet al., 2009, 2012a), lo que reduce la tasa de error de fonemas de alrededor del 26 % al 20,7 %. Ver mohamedet al.(2012b) para un análisis de las razones del éxito de estos modelos. Las extensiones a la canalización básica de reconocimiento telefónico incluyeron la adición de funciones adaptables al altavoz (mohamed et al., 2011) que redujo aún más la tasa de error. Esto fue seguido rápidamente por el trabajo para expandir la arquitectura del reconocimiento de fonemas (que es en lo que se enfoca TIMIT) al reconocimiento de voz de gran vocabulario (Dahlet al., 2012), que implica no solo reconocer fonemas sino también reconocer secuencias de palabras de un amplio vocabulario. Redes profundas para reconocimiento de voz eventualmente

pasó de estar basado en preentrenamiento y máquinas de Boltzmann a estar basado en técnicas como unidades lineales rectificadas y deserción ([Zeiler et al., 2013; Dahl et al., 2013](#)). En ese momento, varios de los principales grupos de oratoria de la industria habían comenzado a explorar el aprendizaje profundo en colaboración con investigadores académicos. [Hinton et al. \(2012a\)](#) describen los avances logrados por estos colaboradores, que ahora se implementan en productos como los teléfonos móviles.

Más tarde, a medida que estos grupos exploraron conjuntos de datos etiquetados cada vez más grandes e incorporaron algunos de los métodos para inicializar, entrenar y configurar la arquitectura de redes profundas, se dieron cuenta de que la fase de preentrenamiento sin supervisión era innecesaria o no aportaba ninguna mejora significativa.

Estos avances en el rendimiento del reconocimiento de la tasa de errores de palabras en el reconocimiento de voz no tenían precedentes (alrededor del 30 % de mejora) y se produjeron después de un largo período de unos diez años durante los cuales las tasas de errores no mejoraron mucho con la tecnología tradicional GMM-HMM, a pesar de la tamaño en continuo crecimiento de los conjuntos de entrenamiento (ver figura 2.4 de [Deng y Yu \(2014\)](#)). Esto creó un cambio rápido en la comunidad de reconocimiento de voz hacia el aprendizaje profundo. En cuestión de aproximadamente dos años, la mayoría de los productos industriales para el reconocimiento de voz incorporaron redes neuronales profundas y este éxito impulsó una nueva ola de investigación sobre arquitecturas y algoritmos de aprendizaje profundo para ASR, que aún continúa en la actualidad.

Una de estas innovaciones fue el uso de redes convolucionales ([Sainath et al., 2013](#)) que replican los pesos a lo largo del tiempo y la frecuencia, mejorando las redes neuronales de retardo de tiempo anteriores que replicaban los pesos solo a lo largo del tiempo. Los nuevos modelos convolucionales bidimensionales consideran el espectrograma de entrada no como un vector largo sino como una imagen, con un eje que corresponde al tiempo y el otro a la frecuencia de los componentes espectrales.

Otro impulso importante, aún en curso, ha sido hacia sistemas de reconocimiento de voz de aprendizaje profundo de extremo a extremo que eliminan por completo el HMM. El primer gran avance en esta dirección provino de [Tumbaset al. \(2013\)](#) que entrenó un LSTM RNN profundo (ver sección [10.10](#)), utilizando la inferencia MAP sobre la alineación de cuadro a fonema, como en [Lecun et al. \(1998b\)](#) y en el marco de CTC ([Tumbas et al., 2006; Tumbas, 2012](#)). Un RNN profundo ([Tumbaset al., 2013](#)) tiene variables de estado de varias capas en cada paso de tiempo, dando al gráfico desplegado dos tipos de profundidad: profundidad ordinaria debido a una pila de capas y profundidad debido al despliegue de tiempo. Este trabajo llevó la tasa de errores de fonemas en TIMIT a un mínimo histórico del 17,7 %. Ver [Pascanu et al. \(2014a\)](#) y [Chung et al. \(2014\)](#) para otras variantes de RNN profundas, aplicadas en otros entornos.

Otro paso contemporáneo hacia el ASR de aprendizaje profundo de extremo a extremo es permitir que el sistema aprenda a "alinear" la información de nivel acústico con el nivel fonético.

información (Chorowskiet al., 2014; Luet al., 2015).

12.4 Procesamiento del lenguaje natural

Procesamiento natural del lenguaje(PNL) es el uso de lenguajes humanos, como el inglés o el francés, por parte de una computadora. Los programas de computadora normalmente leen y emiten lenguajes especializados diseñados para permitir un análisis eficiente y sin ambigüedades por parte de programas simples. Los lenguajes más naturales a menudo son ambiguos y desafían la descripción formal. El procesamiento del lenguaje natural incluye aplicaciones como la traducción automática, en la que el alumno debe leer una oración en un idioma humano y emitir una oración equivalente en otro idioma humano. Muchas aplicaciones de PNL se basan en modelos de lenguaje que definen una distribución de probabilidad sobre secuencias de palabras, caracteres o bytes en un lenguaje natural.

Al igual que con las otras aplicaciones discutidas en este capítulo, las técnicas de redes neuronales muy genéricas se pueden aplicar con éxito al procesamiento del lenguaje natural. Sin embargo, para lograr un rendimiento excelente y escalar bien a aplicaciones grandes, algunas estrategias específicas de dominio se vuelven importantes. Para construir un modelo eficiente de lenguaje natural, generalmente debemos usar técnicas especializadas para procesar datos secuenciales. En muchos casos, optamos por considerar el lenguaje natural como una secuencia de palabras, en lugar de una secuencia de caracteres o bytes individuales. Debido a que el número total de palabras posibles es tan grande, los modelos de lenguaje basados en palabras deben operar en un espacio discreto disperso y de dimensiones extremadamente altas. Se han desarrollado varias estrategias para hacer que los modelos de dicho espacio sean eficientes, tanto en un sentido computacional como estadístico.

12.4.1 *norte*-gramos

Amodelo de lenguaje define una distribución de probabilidad sobre secuencias de tokens en un lenguaje natural. Dependiendo de cómo esté diseñado el modelo, un token puede ser una palabra, un carácter o incluso un byte. Los tokens son siempre entidades discretas. Los primeros modelos de lenguaje exitosos se basaron en modelos de secuencias de tokens de longitud fija llamados *norte*-gramos. Un*norte*-grama es una secuencia de *norte*fichas

Modelos basados en*norte*-gramas definen la probabilidad condicional de la*norte*-th token dado el anterior*norte*-1fichas El modelo utiliza productos de estas distribuciones condicionales para definir la distribución de probabilidad en secuencias más largas:

$$PAG(X_1, \dots, X_t) = PAG(X_1, \dots, X_{t-1}) \prod_{i=t-n+1}^{t-1} PAG(X_i | X_{i-n+1}, \dots, X_{i-1}). \quad (12.5)$$

t=norte

Esta descomposición está justificada por la regla de probabilidad de la cadena. La distribución de probabilidad sobre la secuencia inicial $PAG(X_1, \dots, X_{n-1})$ puede ser modelado por un modelo diferente con un valor menor de *norte*.

Capacitación *norte*-gramos es sencillo porque la estimación de máxima verosimilitud se puede calcular simplemente contando cuántas veces cada posible *norte* gram ocurre en el conjunto de entrenamiento. Modelos basados en *norte*Los -gramas han sido el bloque de construcción central del modelado de lenguaje estadístico durante muchas décadas ([Jelinek y Mercer, 1980; Katz, 1987; Chen y Goodman, 1999](#)).

Para pequeños valores de *norte*, los modelos tienen nombres particulares: **unigram** para *norte*=1, **bigráma** para *norte*=2, y **trígráma** para *norte*=3. Estos nombres derivan de los prefijos latinos para los números correspondientes y del sufijo griego “-gram” que denota algo que está escrito.

Por lo general, entrenamos tanto a un *norte*-gramo modelo y un *n-1*gramo modelo simultáneamente. Esto facilita el cálculo

$$PAG(X_t | X_{t-n+1}, \dots, X_{t-1}) = \frac{PAG_{norte}(X_{t-n+1}, \dots, X_t)}{PAG_{n-1}(X_{t-n+1}, \dots, X_{t-1})} \quad (12.6)$$

simplemente buscando dos probabilidades almacenadas. Para que esto reproduzca exactamente la inferencia en PAG_{norte} , debemos omitir el carácter final de cada secuencia cuando entrenamos PAG_{n-1} .

Como ejemplo, demostramos cómo un modelo de trígráma calcula la probabilidad de la oración "EL PERRO SE ESCAPÓ". Las primeras palabras de la oración no se pueden manejar con la fórmula predeterminada basada en la probabilidad condicional porque no hay contexto al comienzo de la oración. En cambio, debemos usar la probabilidad marginal sobre las palabras al comienzo de la oración. Así evaluamos $PAG_3(\text{EL PERRO CORRIO})$. Finalmente, la última palabra se puede predecir usando el caso típico, de usar la distribución condicional $PAG(\text{LEJOS}/\text{EL PERRO CORRIÓ})$. Poniendo esto junto con la ecuación 12.6, obtenemos:

$$PAG(\text{EL PERRO SE ESCAPÓ}) = PAG_3(\text{EL PERRO CORRIO}) PAG_3(\text{EL PERRO SE ESCAPÓ}) / PAG_2(\text{EL PERRO CORRIÓ}). \quad (12.7)$$

Una limitación fundamental de la máxima verosimilitud para *norte*-gramos es que PAG_{norte} como se estima a partir de los recuentos de conjuntos de entrenamiento, es muy probable que sea cero en muchos casos, aunque la tupla (X_{t-n+1}, \dots, X_t) puede aparecer en el conjunto de prueba. Esto puede causar dos tipos diferentes de resultados catastróficos. Cuando PAG_{n-1} es cero, la relación no está definida, por lo que el modelo ni siquiera produce una salida sensible. Cuando PAG_{n-1} es distinto de cero pero PAG_{norte} es cero, la probabilidad logarítmica de la prueba es $-\infty$. Para evitar resultados tan catastróficos, la mayoría *norte* Los modelos de gramo emplean alguna forma de **alisado**. Técnicas de alisado

cambiar la masa de probabilidad de las tuplas observadas a las no observadas que son similares. Ver [Chen y Goodman\(1999\)](#) para una revisión y comparaciones empíricas. Una técnica básica consiste en agregar una masa de probabilidad distinta de cero a todos los valores de símbolo siguientes posibles. Este método puede justificarse como inferencia bayesiana con un uniforme o Dirichlet previo sobre los parámetros de conteo. Otra idea muy popular es formar un modelo mixto que contenga elementos de orden superior e inferior.*norte*-modelos de grama, donde los modelos de orden superior proporcionan más capacidad y los modelos de orden inferior tienen más probabilidades de evitar conteos de cero.**Métodos de retrocesos** buscar el orden inferior *norte*-gramos si la frecuencia del contexto $X_{t-1}, \dots, X_{t-n+1}$ es demasiado pequeño para usar el modelo de orden superior. Más formalmente, estiman la distribución sobre X_t mediante el uso de contextos $X_{t-n+k}, \dots, X_{t-1}$, por aumentar k , hasta que se encuentre una estimación suficientemente fiable.

Clásico *norte* Los modelos de grama son particularmente vulnerables a la maldición de la dimensionalidad. Hay V/n posibles *norte*-gramos y V suele ser muy grande. Incluso con un conjunto de entrenamiento masivo y modesto *norte*, mayoría *norte*-grams no ocurrirá en el conjunto de entrenamiento. Una forma de ver un clásico *norte*-gram modelo es que está realizando una búsqueda del vecino más cercano. En otras palabras, puede verse como un predictor local no paramétrico, similar a k -vecinos más cercanos. Los problemas estadísticos que enfrentan estos predictores extremadamente locales se describen en la sección [5.11.2](#). El problema para un modelo de lenguaje es aún más grave de lo habitual, porque dos palabras diferentes tienen la misma distancia entre sí en un espacio vectorial caliente. Por lo tanto, es difícil aprovechar mucha información de cualquier "vecino": solo los ejemplos de capacitación que repiten literalmente el mismo contexto son útiles para la generalización local. Para superar estos problemas, un modelo de lenguaje debe poder compartir conocimiento entre una palabra y otras palabras semánticamente similares.

Para mejorar la eficiencia estadística de *norte*-modelos de gramo, **modelos de lenguaje basados en clases** ([Marrón et al., 1992; Ney y Kneser, 1993; Niesler et al., 1998](#)) introducen la noción de categorías de palabras y luego comparten fuerza estadística entre palabras que están en la misma categoría. La idea es utilizar un algoritmo de agrupamiento para dividir el conjunto de palabras en grupos o clases, en función de sus frecuencias de concurrencia con otras palabras. Luego, el modelo puede usar ID de clase de palabra en lugar de ID de palabra individuales para representar el contexto en el lado derecho de la barra de acondicionamiento. También son posibles los modelos compuestos que combinan modelos basados en palabras y basados en clases a través de mezclas o retrocesos. Aunque las clases de palabras proporcionan una forma de generalizar entre secuencias en las que se reemplaza alguna palabra por otra de la misma clase, se pierde mucha información en esta representación.

12.4.2 Modelos de lenguaje neuronal

Modelos de lenguaje neuronal NLM son una clase de modelo de lenguaje diseñado para superar el problema de la maldición de la dimensionalidad para modelar secuencias de lenguaje natural mediante el uso de una representación distribuida de palabras ([bengio et al., 2001](#)). A diferencia de la [clase norte](#)-Modelos de gramática, los modelos de lenguaje neuronal pueden reconocer que dos palabras son similares sin perder la capacidad de codificar cada palabra como distinta de la otra. Los modelos de lenguaje neuronal comparten fuerza estadística entre una palabra (y su contexto) y otras palabras y contextos similares. La representación distribuida que el modelo aprende para cada palabra permite este intercambio al permitir que el modelo trate las palabras que tienen características en común de manera similar. Por ejemplo, si la palabra *perro* y la palabra *gato* comparten muchas representaciones que comparten muchos atributos, luego las oraciones que contienen la palabra *gato* pueden informar las predicciones que hará el modelo para las oraciones que contienen la palabra *perro*, y viceversa. Debido a que hay muchos atributos de este tipo, hay muchas formas en que puede ocurrir la generalización, transfiriendo información de cada oración de entrenamiento a un número exponencialmente grande de oraciones relacionadas semánticamente. La maldición de la dimensionalidad requiere que el modelo se generalice a un número de oraciones que es exponencial en la longitud de la oración. El modelo contrarresta esta maldición relacionando cada oración de entrenamiento con un número exponencial de oraciones similares.

A veces llamamos a estas representaciones de palabras **incrustaciones de palabras**. En esta interpretación, vemos los símbolos en bruto como puntos en un espacio de dimensión igual al tamaño del vocabulario. Las representaciones de palabras incrustan esos puntos en un espacio de características de menor dimensión. En el espacio original, cada palabra es representada por un vector caliente, por lo que cada par de palabras está a una distancia euclídea de cada uno. En el espacio de incrustación, las palabras que aparecen con frecuencia en contextos similares (o cualquier par de palabras que comparten algunas "características" aprendidas por el modelo) están cerca unas de otras. Esto a menudo da como resultado que palabras con significados similares sean vecinos. Cifra [12.3](#) hace zoom en áreas específicas de un espacio de incrustación de palabras aprendidas para mostrar cómo las palabras semánticamente similares se asignan a representaciones que están cerca unas de otras.

Las redes neuronales en otros dominios también definen incrustaciones. Por ejemplo, una capa oculta de una red convolucional proporciona una "incrustación de imágenes". Por lo general, los practicantes de PNL están mucho más interesados en esta idea de incrustaciones porque el lenguaje natural no se encuentra originalmente en un espacio vectorial de valor real. La capa oculta ha proporcionado un cambio cualitativamente más dramático en la forma en que se representan los datos.

La idea básica de usar representaciones distribuidas para mejorar los modelos para el procesamiento del lenguaje natural no se limita a las redes neuronales. También se puede usar con modelos gráficos que tienen representaciones distribuidas en forma de

múltiples variables latentes ([Mnih y Hinton, 2007](#)).

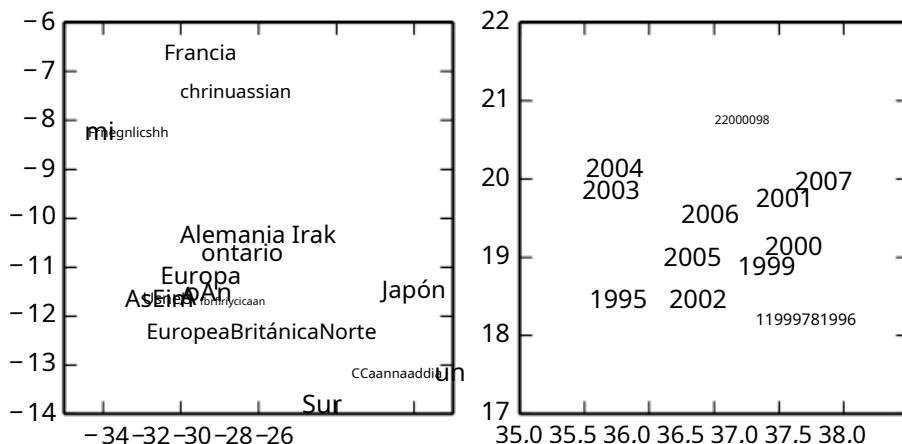


Figura 12.3: visualizaciones bidimensionales de incrustaciones de palabras obtenidas de un modelo de traducción automática neuronal ([Bahdanau et al., 2015](#)), haciendo zoom en áreas específicas donde las palabras relacionadas semánticamente tienen vectores incrustados que están cerca uno del otro. Los países aparecen a la izquierda y los números a la derecha. Tenga en cuenta que estas incrustaciones son 2-D con fines de visualización. En aplicaciones reales, las incrustaciones suelen tener una mayor dimensionalidad y pueden capturar simultáneamente muchos tipos de similitudes entre palabras.

12.4.3 Salidas de alta dimensión

En muchas aplicaciones de lenguaje natural, a menudo queremos que nuestros modelos produzcan palabras (en lugar de caracteres) como unidad fundamental del resultado. Para vocabularios grandes, puede ser muy costoso computacionalmente representar una distribución de salida sobre la elección de una palabra, porque el tamaño del vocabulario es grande. En muchas aplicaciones, V contiene cientos de miles de palabras. El enfoque ingenuo para representar tal distribución es aplicar una transformación afín desde una representación oculta al espacio de salida, luego aplicar la función softmax. Supongamos que tenemos un vocabulario V con tamaño $/V/$. La matriz de pesos que describe el componente lineal de esta transformación afín es muy grande, porque su dimensión de salida es $/V/$. Esto impone un alto costo de memoria para representar la matriz y un alto costo computacional para multiplicarla. Debido a que el softmax está normalizado en todos $/V/$ salidas, es necesario realizar la multiplicación de matriz completa en el tiempo de entrenamiento, así como en el tiempo de prueba; no podemos calcular solo el producto escalar con el vector de peso para la salida correcta. Los altos costos computacionales de la capa de salida surgen tanto en el momento del entrenamiento (para calcular la probabilidad y su gradiente) como en el momento de la prueba (para calcular las probabilidades de todas las palabras o de las seleccionadas). para especializados

funciones de pérdida, el gradiente se puede calcular de manera eficiente (Vicente et al., 2015), pero la pérdida de entropía cruzada estándar aplicada a una capa de salida softmax tradicional plantea muchas dificultades.

Suponer que h es la capa oculta superior utilizada para predecir las probabilidades de salida \hat{y} . Si parametrizamos la transformación de h con pesos aprendidos W y sesgos aprendidos b , la capa de salida affine-softmax realiza los siguientes cálculos:

$$a = b + \sum_j W_{j,i} h_j \quad \forall i \in \{1, \dots, |V|\}, \quad (12.8)$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i=1}^{|V|} e^{a_i}}. \quad (12.9)$$

Si h contiene n entradas entonces la operación anterior es $O(|V|n)$. Con n en los miles y $|V|$ en cientos de miles, esta operación domina el cálculo de la mayoría de los modelos de lenguaje neuronal.

12.4.3.1 Uso de una Lista Corta

Los primeros modelos de lenguaje neural (Bengio et al., 2001, 2003) se ocupó del alto costo de usar un softmax sobre una gran cantidad de palabras de salida al limitar el tamaño del vocabulario a 10,000 o 20,000 palabras. Schwenk y Gauvain (2002) y Schwenk (2007) construido sobre este enfoque al dividir el vocabulario en una lista corta de palabras más frecuentes (manejadas por la red neuronal) y una cola $T = |V| - L$ de palabras más raras (manejadas por un *norte*-modelo de gramo). Para poder combinar las dos predicciones, la red neuronal también tiene que predecir la probabilidad de que una palabra aparezca después del contexto. C pertenece a la cola de lista. Esto se puede lograr agregando una unidad de salida sigmoidea adicional para proporcionar una estimación de $PAG(i \in T | C)$. La salida adicional se puede usar para lograr una estimación de la distribución de probabilidad sobre todas las palabras en V como sigue:

$$\begin{aligned} PAG(y=y_o | C) &= \sum_{i \in L} PAG(y=y_o | C, y_o \in L)(1 - PAG(i \in T | C)) \\ &\quad + \sum_{i \in T} PAG(y=y_o | C, y_o \in T)PAG(i \in T | C) \end{aligned} \quad (12.10)$$

dónde $PAG(y=y_o | C, y_o \in L)$ es proporcionado por el modelo de lenguaje neuronal y $PAG(y=y_o | C, y_o \in T)$ es proporcionado por el *norte*-modelo de gramo. Con una ligera modificación, este enfoque también puede funcionar usando un valor de salida adicional en la capa softmax del modelo de lenguaje neuronal, en lugar de una unidad sigmoidea separada.

Una desventaja obvia del enfoque de lista corta es que la potencial ventaja de generalización de los modelos de lenguaje neuronal se limita a los más frecuentes.

palabras, donde, discutiblemente, es menos útil. Esta desventaja ha estimulado la exploración de métodos alternativos para tratar con salidas de alta dimensión, que se describen a continuación.

12.4.3.2 Softmax jerárquico

Un enfoque clásico ([Buen hombre,2001](#)) para reducir la carga computacional de las capas de salida de alta dimensión sobre grandes conjuntos de vocabularioVes descomponer probabilidades jerárquicamente. En lugar de necesitar un número de cálculos proporcionales a V^y (y también proporcional al número de unidades ocultas,*norte_h*), el V^y El factor puede reducirse a un valor tan bajo comoregistro V^y .[bengio\(2002\)](#) y[Morín y Bengio\(2005\)](#) introdujo este enfoque factorizado en el contexto de los modelos de lenguaje neural.

Uno puede pensar en esta jerarquía como la construcción de categorías de palabras, luego categorías de categorías de palabras, luego categorías de categorías de categorías de palabras, etc. Estas categorías anidadas forman un árbol, con palabras en las hojas. En un árbol equilibrado, el árbol tiene profundidad. $O(\text{registro } V^y)$. La probabilidad de elegir una palabra viene dada por el producto de las probabilidades de elegir la rama que conduce a esa palabra en cada nodo del camino desde la raíz del árbol hasta la hoja que contiene la palabra. Cifra [12.4](#) ilustra un ejemplo sencillo.[Mnih y Hinton\(2009\)](#) también describen cómo usar varias rutas para identificar una sola palabra a fin de modelar mejor las palabras que tienen varios significados. Calcular la probabilidad de una palabra implica la suma de todos los caminos que conducen a esa palabra.

Para predecir las probabilidades condicionales requeridas en cada nodo del árbol, generalmente usamos un modelo de regresión logística en cada nodo del árbol y proporcionamos el mismo contexto *C*omo entrada a todos estos modelos. Debido a que la salida correcta está codificada en el conjunto de entrenamiento, podemos usar el aprendizaje supervisado para entrenar los modelos de regresión logística. Esto normalmente se hace usando una pérdida de entropía cruzada estándar, correspondiente a maximizar la probabilidad logarítmica de la secuencia correcta de decisiones.

Debido a que el registro de probabilidad de salida se puede calcular de manera eficiente (tan bajo comoregistro V^y en vez de V^y), sus gradientes también se pueden calcular de manera eficiente. Esto incluye no solo el degradado con respecto a los parámetros de salida, sino también los degradados con respecto a las activaciones de la capa oculta.

Es posible, pero generalmente no práctico, optimizar la estructura de árbol para minimizar el número esperado de cálculos. Las herramientas de la teoría de la información especifican cómo elegir el código binario óptimo dadas las frecuencias relativas de las palabras. Para ello, podríamos estructurar el árbol de forma que el número de bits asociados a una palabra sea aproximadamente igual al logaritmo de la frecuencia de esa palabra. Sin embargo, en

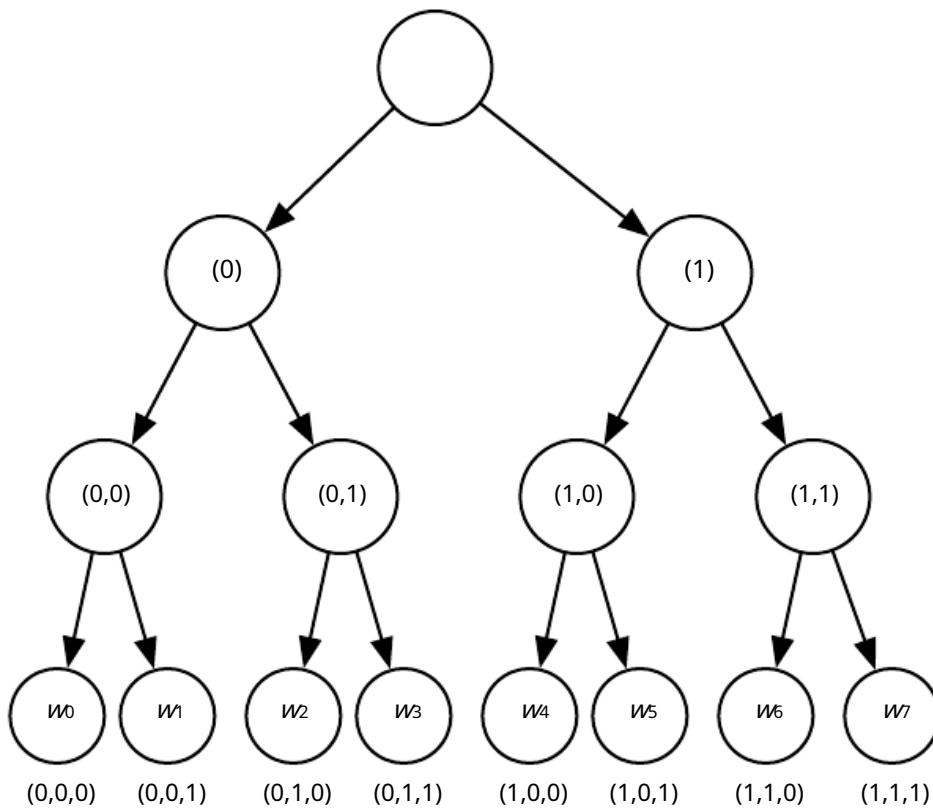


Figura 12.4: Ilustración de una jerarquía simple de categorías de palabras, con 8 palabras w_0, \dots, w_7 organizados en una jerarquía de tres niveles. Las hojas del árbol representan palabras específicas reales. Los nodos internos representan grupos de palabras. Cualquier nodo puede ser indexado por la secuencia de decisiones binarias (0=izquierda, 1=derecha) para llegar al nodo desde la raíz. Superclase (0) contiene las clases (0,0) y (0,1), que contienen respectivamente los conjuntos de palabras $\{w_0, w_1\}$ y $\{w_2, w_3\}$, y de manera similar superclase (1) contiene las clases (1,0) y (1,1), que contienen respectivamente las palabras $\{w_4, w_5\}$ y $\{w_6, w_7\}$. Si el árbol está suficientemente equilibrado, la profundidad máxima (número de decisiones binarias) es del orden del logaritmo del número de palabras V : la elección de uno de V palabras se pueden obtener haciendo $O(\log V)$ operaciones (una para cada uno de los nodos en la ruta desde la raíz). En este ejemplo, calcular la probabilidad de una palabra y se puede hacer multiplicando tres probabilidades, asociadas con las decisiones binarias de moverse hacia la izquierda o hacia la derecha en cada nodo en el camino desde la raíz hasta un nodo y . Dejar $b_i(y)$ ser la i -ésima decisión binaria al atravesar el árbol hacia el valor y . La probabilidad de muestrear una salida y se descompone en un producto de probabilidades condicionales, usando la regla de la cadena para probabilidades condicionales, con cada nodo indexado por el prefijo de estos bits. Por ejemplo, nodo (1,0) corresponde al prefijo $(b_0(w_4) = 1, b_1(w_4) = 0)$, y la probabilidad de w_4 se puede descomponer de la siguiente manera:

$$PAG(y = w_4) = PAG(b_0=1, b_1=0, b_2=0) \quad (12.11)$$

$$= PAG(b_0=1) PAG(b_1=0 / b_0=1) PAG(b_2=0 / b_0=1, b_1=0). \quad (12.12)$$

En la práctica, los ahorros computacionales generalmente no valen la pena porque el cálculo de las probabilidades de salida es solo una parte del cálculo total en el modelo de lenguaje neuronal. Por ejemplo, supongamos que hay n capas ocultas de ancho completamente conectadas n . Dejar n sea el promedio ponderado del número de bits necesarios para identificar una palabra, con la ponderación dada por la frecuencia de estas palabras. En este ejemplo, el número de operaciones necesarias para calcular el oculto activaciones crece a medida que $O(en^2)$ mientras que los cálculos de salida crecen como $O(n^2en^2)$. Mientras $n \leq e$, podemos reducir más el cálculo reduciendo n que por encogerse en n . En efecto, n suele ser pequeño. Debido a que el tamaño del vocabulario rara vez supera el millón de palabras y $\log(10^6) \approx 20$, es posible reducir n a alrededor de 20, pero n es a menudo mucho más grande, alrededor de 1000 o más. En lugar de optimizar cuidadosamente un árbol con un factor de ramificación de 2, en cambio, se puede definir un árbol con profundidad dos y un factor de ramificación de V . Tal árbol corresponde simplemente a definir un conjunto de clases de palabras mutuamente excluyentes. El enfoque simple basado en un árbol de profundidad dos captura la mayor parte del beneficio computacional de la estrategia jerárquica.

Una pregunta que permanece algo abierta es cómo definir mejor estas clases de palabras, o cómo definir la jerarquía de palabras en general. Los primeros trabajos utilizaron jerarquías existentes ([Morín y Bengio, 2005](#)) pero la jerarquía también se puede aprender, idealmente junto con el modelo de lenguaje neural. Aprender la jerarquía es difícil. Una optimización exacta de la verosimilitud logarítmica parece intratable porque la elección de una jerarquía de palabras es discreta, no susceptible de optimización basada en gradientes. Sin embargo, se podría utilizar la optimización discreta para optimizar aproximadamente la partición de palabras en clases de palabras.

Una ventaja importante del softmax jerárquico es que brinda beneficios computacionales tanto en el momento del entrenamiento como en el momento de la prueba, si en el momento de la prueba queremos calcular la probabilidad de palabras específicas.

Por supuesto, calcular la probabilidad de todos V las palabras seguirán siendo caras incluso con el softmax jerárquico. Otra operación importante es seleccionar la palabra más probable en un contexto dado. Desafortunadamente, la estructura de árbol no proporciona una solución eficiente y exacta a este problema.

Una desventaja es que, en la práctica, el softmax jerárquico tiende a dar peores resultados de prueba que los métodos basados en muestreo que describiremos a continuación. Esto puede deberse a una mala elección de las clases de palabras.

12.4.3.3 Muestreo de importancia

Una forma de acelerar el entrenamiento de los modelos de lenguaje neuronal es evitar calcular explícitamente la contribución del gradiente de todas las palabras que no aparecen.

en la siguiente posición. Cada palabra incorrecta debe tener baja probabilidad bajo el modelo. Puede ser computacionalmente costoso enumerar todas estas palabras. En cambio, es posible muestrear solo un subconjunto de las palabras. Usando la notación introducida en la ecuación 12.8, el gradiente se puede escribir de la siguiente manera:

$$\frac{\partial \text{registro } PAG(y / \mathcal{C})}{\partial \theta} = \frac{\partial \text{registro softmax}(a)}{\partial \theta} \quad (12.13)$$

$$= \frac{\partial}{\partial \theta} \text{registro} - \frac{m_i a_y}{m_i a_i} \quad (12.14)$$

$$= \frac{\partial}{\partial \theta} (a_y - \sum_i m_i a_i) \quad (12.15)$$

$$= \frac{\partial a_y}{\partial \theta} - \sum_i PAG(y=o / \mathcal{C}) \frac{\partial a_i}{\partial \theta} \quad (12.16)$$

dónde a es el vector de activaciones (o puntajes) anteriores a softmax, con un elemento por palabra. El primer término es el **fase positiva** término (empujar a hacia arriba) mientras que el segundo término es el **fase negativa** término (empujar a abajo para todos i , con peso $PAG(y=o / \mathcal{C})$). Dado que el término de fase negativa es una expectativa, podemos estimarlo con una muestra de Monte Carlo. Sin embargo, eso requeriría tomar muestras del propio modelo. El muestreo del modelo requiere computación $PAG(y=o / \mathcal{C})$ para todos i en el vocabulario, que es precisamente lo que estamos tratando de evitar.

En lugar de muestrear del modelo, se puede muestrear de otra distribución, llamada distribución propuesta (denotada q), y use los pesos apropiados para corregir el sesgo introducido por el muestreo de la distribución incorrecta ([Bengio y Sénécal, 2003](#); [Bengio y Sénécal, 2008](#)). Esta es una aplicación de una técnica más general llamada **muestreo de importancia**, que se describirá con más detalle en la sección 17.2. Desafortunadamente, incluso el muestreo de importancia exacta no es eficiente porque requiere ponderaciones computacionales p_{agi}/q_{qi} , donde $p_{agi}=PAG(y=o / \mathcal{C})$, que solo se puede calcular si todas las puntuaciones a son computados. La solución adoptada para esta aplicación se denomina **muestreo de importancia sesgada**, donde los pesos de importancia se normalizan para sumar 1. Cuando la palabra negativa $norte$ se muestrea, el gradiente asociado se pondera por

$$W_i = - \frac{p_{agnorte}/q_{norte_i}}{\sum_{j=1}^n p_{agnorte_j}/q_{norte_j}}. \quad (12.17)$$

Estos pesos se utilizan para dar la importancia adecuada a las *metromuestras* negativas de q utilizado para formar la contribución de fase negativa estimada a la

degradado:

$$\frac{1}{N} \sum_{i=1}^N \frac{\partial a_i}{\partial \theta} \approx \frac{1}{|V|} \sum_{v \in V} w_v \frac{\partial a_{v, \text{norte}}}{\partial \theta}. \quad (12.18)$$

Una distribución de unigrama o bigrama funciona bien como la distribución propuesta. Es fácil estimar los parámetros de tal distribución a partir de datos. Después de estimar los parámetros, también es posible muestrear de tal distribución de manera muy eficiente.

El muestreo de importancia no solo es útil para acelerar los modelos con grandes salidas softmax. En términos más generales, es útil para acelerar el entrenamiento con grandes capas de salida dispersas, donde la salida es un vector disperso en lugar de un 1-de-norte elección. Un ejemplo es **un bolsa de palabras**. Una bolsa de palabras es un vector escaso y dónde v indica la presencia o ausencia de la palabra / del vocabulario del documento. Alternativamente, v puede indicar el número de veces que esa palabra / aparece. Los modelos de aprendizaje automático que emiten vectores tan dispersos pueden ser costosos de entrenar por una variedad de razones. Al principio del aprendizaje, es posible que el modelo no elija hacer que la salida sea realmente escasa. Además, la función de pérdida que usamos para el entrenamiento podría describirse más naturalmente en términos de comparar cada elemento de la salida con cada elemento del objetivo. Esto significa que no siempre está claro que haya un beneficio computacional al usar salidas dispersas, porque el modelo puede optar por hacer que la mayoría de la salida sea distinta de cero y todos estos valores distintos de cero deben compararse con el entrenamiento correspondiente. incluso si el objetivo de entrenamiento es cero. [Delfínet al. \(2011\)](#) demostraron que dichos modelos pueden acelerarse mediante el muestreo por importancia. El algoritmo eficiente minimiza la reconstrucción de pérdidas para las "palabras positivas" (aquellas que no son cero en el objetivo) y un número igual de "palabras negativas". Las palabras negativas se eligen al azar, utilizando una heurística para muestrear palabras que tienen más probabilidades de estar equivocadas. El sesgo introducido por este sobremuestreo heurístico se puede corregir utilizando ponderaciones de importancia.

En todos estos casos, la complejidad computacional de la estimación del gradiente para la capa de salida se reduce para que sea proporcional al número de muestras negativas en lugar de ser proporcional al tamaño del vector de salida.

12.4.3.4 Estimación de contraste de ruido y pérdida de clasificación

Se han propuesto otros enfoques basados en el muestreo para reducir el costo computacional de entrenar modelos de lenguaje neural con vocabularios extensos. Un ejemplo temprano es la pérdida de clasificación propuesta por [Collobert y Weston \(2008a\)](#), que ve la salida del modelo de lenguaje neuronal para cada palabra como una puntuación e intenta hacer la puntuación de la palabra correcta *a* estar clasificado alto en comparación con los otros

puntuaciones). La pérdida de clasificación propuesta entonces es

$$L = \max_i(0, 1 - a_y + a_i). \quad (12.19)$$

El gradiente es cero para el i -ésimo término si la puntuación de la palabra observada, a_y , es mayor que la puntuación de la palabra negativa a por un margen de 1. Un problema con este criterio es que no proporciona probabilidades condicionales estimadas, que son útiles en algunas aplicaciones, incluido el reconocimiento de voz y la generación de texto (incluidas las tareas de generación de texto condicional, como la traducción).

Un objetivo de entrenamiento utilizado más recientemente para el modelo de lenguaje neuronal es la estimación de contraste de ruido, que se presenta en la sección 18.6. Este enfoque se ha aplicado con éxito a los modelos de lenguaje neural (Mnih y Teh, 2012; Mnih y Kavukcuoglu, 2013).

12.4.4 Combinación de modelos de lenguaje neuronal con n-gramos

Una gran ventaja de *n*-gramo modelos sobre redes neuronales es que los *n*-gramo modelos logran una alta capacidad de modelo (al almacenar las frecuencias de muchas tuplas) mientras requieren muy poco cálculo para procesar un ejemplo (buscando solo unas pocas tuplas que coinciden con el contexto actual). Si usamos tablas hash o árboles para acceder a los recuentos, el cálculo utilizado para *n*-gramos es casi independiente de la capacidad. En comparación, duplicar la cantidad de parámetros de una red neuronal generalmente también duplica aproximadamente su tiempo de cálculo. Las excepciones incluyen modelos que evitan usar todos los parámetros en cada pasada. Las capas de incrustación indexan solo una incrustación en cada paso, por lo que podemos aumentar el tamaño del vocabulario sin aumentar el tiempo de cálculo por ejemplo. Algunos otros modelos, como las redes convolucionales en mosaico, pueden agregar parámetros mientras reducen el grado de uso compartido de parámetros para mantener la misma cantidad de cómputo. Sin embargo, las capas de redes neuronales típicas basadas en la multiplicación de matrices utilizan una cantidad de cálculo proporcional al número de parámetros.

Una manera fácil de agregar capacidad es combinar ambos enfoques en un conjunto que consta de un modelo de lenguaje neuronal y un *n*-gramo modelo de lenguaje (Bengio et al., 2001, 2003). Como con cualquier conjunto, esta técnica puede reducir el error de prueba si los miembros del conjunto cometen errores independientes. El campo del aprendizaje de conjuntos proporciona muchas formas de combinar las predicciones de los miembros del conjunto, incluida la ponderación uniforme y los pesos elegidos en un conjunto de validación. Mikolov et al. (2011a) amplió el conjunto para incluir no solo dos modelos, sino una gran variedad de modelos. También es posible emparejar una red neuronal con un modelo de máxima entropía y entrenar ambos conjuntamente (Mikolov et al., 2011b). Este enfoque puede ser visto como un entrenamiento

una red neuronal con un conjunto adicional de entradas que están conectadas directamente a la salida y no conectadas a ninguna otra parte del modelo. Las entradas adicionales son indicadores de la presencia de determinadas *norte*-gramas en el contexto de entrada, por lo que estas variables son muy dimensionales y muy escasas. El aumento en la capacidad del modelo es enorme: la nueva parte de la arquitectura contiene hasta $/sv /norte$ parámetros, pero la cantidad de computación adicional necesaria para procesar una entrada es mínima porque las entradas adicionales son muy escasas.

12.4.5 Traducción automática neuronal

La traducción automática es la tarea de leer una oración en un idioma natural y emitir una oración con el significado equivalente en otro idioma. Los sistemas de traducción automática a menudo involucran muchos componentes. A un alto nivel, suele haber un componente que propone muchas traducciones candidatas. Muchas de estas traducciones no serán gramaticales debido a las diferencias entre los idiomas. Por ejemplo, muchos idiomas ponen adjetivos después de los sustantivos, por lo que cuando se traducen directamente al inglés producen frases como "manzana roja". El mecanismo de propuesta sugiere muchas variantes de la traducción sugerida, idealmente incluyendo "manzana roja". Un segundo componente del sistema de traducción, un modelo de lenguaje, evalúa las traducciones propuestas y puede calificar "manzana roja" como mejor que "manzana roja".

El primer uso de las redes neuronales para la traducción automática fue actualizar el modelo de lenguaje de un sistema de traducción mediante el uso de un modelo de lenguaje neuronal (Schwenk *et al.*, 2006; Schwenk, 2010). Anteriormente, la mayoría de los sistemas de traducción automática habían utilizado un *norte* modelo de gramo para este componente. El *norte*-Los modelos basados en gramas utilizados para la traducción automática incluyen no solo el retroceso tradicional *norte*-modelos de gramo (Jelinek y Mercer, 1980; Katz, 1987; Chen y Goodman, 1999) pero también **modelos de lenguaje de máxima entropía** (Berger *et al.*, 1996), en el que una capa affine-softmax predice la siguiente palabra dada la presencia de frecuentes *norte*-gramas en el contexto.

Los modelos de lenguaje tradicional simplemente informan la probabilidad de una oración en lenguaje natural. Debido a que la traducción automática implica producir una oración de salida dada una oración de entrada, tiene sentido extender el modelo de lenguaje natural para que sea condicional. Como se describe en la sección 6.2.1.1, es sencillo extender un modelo que define una distribución marginal sobre alguna variable para definir una distribución condicional sobre esa variable dado un contexto C , donde C puede ser una sola variable o una lista de variables. Devlin *et al.* (2014) superó el estado del arte en algunos puntos de referencia estadísticos de traducción automática mediante el uso de un MLP para calificar una frase t_1, t_2, \dots, t_k en el idioma de destino dada una frase $s_1, s_2, \dots, s_{norte}$ en el idioma de origen. Las estimaciones de $MLPPAG(t_1, t_2, \dots, t_k / s_1, s_2, \dots, s_{norte})$. La estimación formada por este MLP reemplaza la estimación proporcionada por condicional *norte*-modelos de gramo.

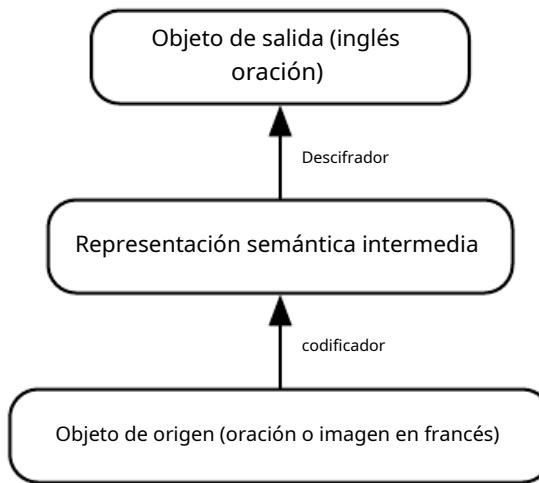


Figura 12.5: La arquitectura de codificador-decodificador para mapear hacia adelante y hacia atrás entre una representación de superficie (como una secuencia de palabras o una imagen) y una representación semántica. Al usar la salida de un codificador de datos de una modalidad (como el mapeo del codificador de oraciones en francés a representaciones ocultas que capturan el significado de las oraciones) como la entrada a un decodificador para otra modalidad (como el mapeo del decodificador de representaciones ocultas que capturan el significado de las oraciones al inglés), podemos entrenar sistemas que traducen de una modalidad a otra. Esta idea se ha aplicado con éxito no solo a la traducción automática, sino también a la generación de subtítulos a partir de imágenes.

Un inconveniente del enfoque basado en MLP es que requiere que las secuencias sean preprocesadas para que tengan una longitud fija. Para hacer que la traducción sea más flexible, nos gustaría usar un modelo que pueda acomodar entradas y salidas de longitud variable. Un RNN proporciona esta capacidad. Sección 10.2.4 describe varias formas de construir un RNN que representa una distribución condicional sobre una secuencia dada alguna entrada, y la sección 10.4 describe cómo lograr este condicionamiento cuando la entrada es una secuencia. En todos los casos, un modelo primero lee la secuencia de entrada y emite una estructura de datos que resume la secuencia de entrada. Llamamos a este resumen el “contexto” C . El contexto C puede ser una lista de vectores, o puede ser un vector o tensor. El modelo que lee la entrada para producir C puede ser un RNN (Cho et al., 2014a; Sutskever et al., 2014; Väquero et al., 2014) o una red convolucional (Kalchbrenner y Blunsom, 2013). Un segundo modelo, generalmente un RNN, luego lee el contexto C y genera una oración en el idioma de destino. Esta idea general de un marco codificador-decodificador para la traducción automática se ilustra en la figura 12.5.

Para generar una oración completa condicionada a la oración fuente, el modelo debe tener una forma de representar la oración fuente completa. Los modelos anteriores solo podían representar palabras o frases individuales. De una representación

Desde el punto de vista del aprendizaje, puede ser útil aprender una representación en la que las oraciones que tienen el mismo significado tienen representaciones similares independientemente de si fueron escritas en el idioma de origen o en el idioma de destino. Esta estrategia se exploró primero usando una combinación de convoluciones y RNN ([Kalchbrenner y Blunsom, 2013](#)). Trabajos posteriores introdujeron el uso de un RNN para calificar las traducciones propuestas ([Cho et al., 2014a](#)) y para generar oraciones traducidas ([Sutskever et al., 2014](#)). [Vaquero et al.](#)(2014) escalaron estos modelos a vocabularios más grandes.

12.4.5.1 Uso de un mecanismo de atención y alineación de piezas de datos

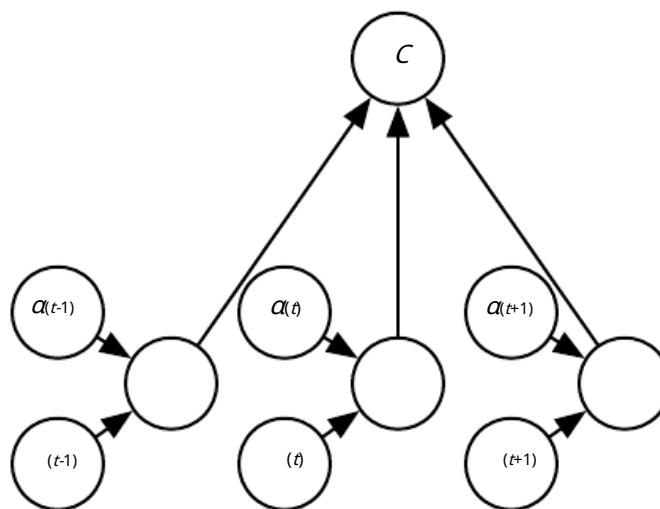


Figura 12.6: Un mecanismo de atención moderno, presentado por [Bahdanau et al.\(2015\)](#), es esencialmente un promedio ponderado. Un vector de contexto C se forma tomando un promedio ponderado de vectores de características h_t con pesos α_t . En algunas aplicaciones, los vectores de características h_t son unidades ocultas de una red neuronal, pero también pueden ser entradas sin procesar para el modelo. Los pesos α_t son producidos por el propio modelo. Suelen ser valores en el intervalo $[0,1]$ y están destinados a concentrarse en torno a una sola h_t para que el promedio ponderado se aproxime a la lectura de ese paso de tiempo específico con precisión. Los pesos α_t generalmente se producen aplicando una función softmax a las puntuaciones de relevancia emitidas por otra parte del modelo. El mecanismo de atención es más costoso computacionalmente que indexar directamente el deseado h_t , pero la indexación directa no se puede entrenar con descenso de gradiente. El mecanismo de atención basado en promedios ponderados es una aproximación suave y diferenciable que se puede entrenar con los algoritmos de optimización existentes.

Usar una representación de tamaño fijo para capturar todos los detalles semánticos de una oración muy larga de, digamos, 60 palabras es muy difícil. Se puede lograr entrenando un RNN lo suficientemente grande, lo suficientemente bien y durante el tiempo suficiente, como lo demuestra [Cho et al.\(2014a\)](#) y [Sutskever et al.\(2014\)](#). Sin embargo, un enfoque más eficiente es leer la oración o el párrafo completo (para obtener el contexto y la esencia de lo que

se está expresando), luego produzca las palabras traducidas una a la vez, enfocándose cada vez en una parte diferente de la oración de entrada para recopilar los detalles semánticos que se requieren para producir la siguiente palabra de salida. Esa es exactamente la idea que Bahdanau *et al.* (2015) introdujo por primera vez. El mecanismo de atención utilizado para centrarse en partes específicas de la secuencia de entrada en cada paso de tiempo se ilustra en la figura 12.6.

Podemos pensar que un sistema basado en la atención tiene tres componentes:

1. Un proceso que “*lee*” datos sin procesar (como palabras de origen en una oración de origen) y los convierte en representaciones distribuidas, con un vector de características asociado con cada posición de palabra.
2. Una lista de vectores de características que almacenan la salida del lector. Esto puede entenderse como un “*memoria*” que contiene una secuencia de hechos, que pueden ser recuperados posteriormente, no necesariamente en el mismo orden, sin tener que visitarlos todos.
3. Un proceso que “*explota*” el contenido de la memoria para realizar secuencialmente una tarea, teniendo en cada paso de tiempo la capacidad de poner atención en el contenido de un elemento de la memoria (o varios, con un peso diferente).

El tercer componente genera la oración traducida.

Cuando las palabras en una oración escrita en un idioma se alinean con las palabras correspondientes en una oración traducida en otro idioma, es posible relacionar las incrustaciones de palabras correspondientes. Trabajos anteriores mostraron que se podía aprender una especie de matriz de traducción que relacionaba la palabra incrustaciones en un idioma con la palabra incrustaciones en otro (Kočiský *et al.*, 2014), lo que arroja tasas de error de alineación más bajas que los enfoques tradicionales basados en los recuentos de frecuencia en la tabla de frases. Incluso hay un trabajo anterior sobre el aprendizaje de vectores de palabras en varios idiomas (Kleméntievet *et al.*, 2012). Muchas extensiones a este enfoque son posibles. Por ejemplo, una alineación translingual más eficiente (gouwset *al.*, 2014) permite entrenar en conjuntos de datos más grandes.

12.4.6 Perspectiva histórica

La idea de representaciones distribuidas de símbolos fue introducida por Rumelhart *et al.* (1986a) en una de las primeras exploraciones de retropropagación, con símbolos correspondientes a la identidad de los miembros de la familia y la red neuronal capturando las relaciones entre los miembros de la familia, con ejemplos de entrenamiento que forman trillizos como (Colin, Mother, Victoria). La primera capa de la red neuronal aprendió una representación de cada miembro de la familia. Por ejemplo, las características de Colin

podría representar en qué árbol genealógico estaba Colin, en qué rama de ese árbol estaba, de qué generación era, etc. Uno puede pensar en la red neuronal como reglas aprendidas computacionales que relacionan estos atributos para obtener las predicciones deseadas. Luego, el modelo puede hacer predicciones, como inferir quién es la madre de Colin.

La idea de formar una incrustación de un símbolo se extendió a la idea de una incrustación de una palabra por Deerwester et al. (1990). Estas incrustaciones se aprendieron usando el SVD. Más tarde, las incrustaciones serían aprendidas por redes neuronales.

La historia del procesamiento del lenguaje natural está marcada por transiciones en la popularidad de diferentes formas de representar la entrada al modelo. Siguiendo este trabajo inicial sobre símbolos o palabras, algunas de las primeras aplicaciones de las redes neuronales a la PNL (Miikkulainen y Dyer, 1991; Schmidhuber, 1996) representó la entrada como una secuencia de caracteres.

bengio et al. (2001) volvió a centrarse en el modelado de palabras e introdujo modelos de lenguaje neuronal, que producen incrustaciones de palabras interpretables. Estos modelos neuronales han pasado de definir representaciones de un pequeño conjunto de símbolos en la década de 1980 a millones de palabras (incluidos nombres propios y errores ortográficos) en aplicaciones modernas. Este esfuerzo de escalamiento computacional condujo a la invención de las técnicas descritas anteriormente en la sección 12.4.3.

Inicialmente, el uso de palabras como unidades fundamentales de los modelos lingüísticos mejoró el rendimiento del modelado lingüístico. bengio et al. (2001). Hasta el día de hoy, las nuevas técnicas impulsan continuamente ambos modelos basados en personajes (Sutskever et al., 2011) y modelos basados en palabras, con trabajos recientes (Gillick et al., 2015) incluso modelando bytes individuales de caracteres Unicode.

Las ideas detrás de los modelos de lenguaje neuronal se han extendido a varias aplicaciones de procesamiento de lenguaje natural, como el análisis sintáctico (Henderson, 2003, 2004; coloberto, 2011), etiquetado de parte del discurso, etiquetado de roles semánticos, fragmentación, etc., a veces utilizando una única arquitectura de aprendizaje multitarea (Collobert y Weston, 2008a; coloberto et al., 2011a) en el que las incrustaciones de palabras se comparten entre tareas.

Las visualizaciones bidimensionales de incrustaciones se convirtieron en una herramienta popular para analizar modelos de lenguaje luego del desarrollo del algoritmo de reducción de dimensionalidad t-SNE (van der Maaten y Hinton, 2008) y su aplicación de alto perfil para visualización de incrustaciones de palabras por Joseph Turian en 2009.

12.5 Otras aplicaciones

En esta sección, cubrimos algunos otros tipos de aplicaciones de aprendizaje profundo que son diferentes de las tareas estándar de reconocimiento de objetos, reconocimiento de voz y procesamiento de lenguaje natural discutidas anteriormente. Parte **tercero** de este libro ampliará ese alcance aún más a tareas que siguen siendo principalmente áreas de investigación.

12.5.1 Sistemas de recomendación

Una de las principales familias de aplicaciones del aprendizaje automático en el sector de las tecnologías de la información es la capacidad de hacer recomendaciones de artículos a usuarios o clientes potenciales. Se pueden distinguir dos tipos principales de aplicaciones: publicidad en línea y recomendaciones de artículos (a menudo, estas recomendaciones todavía tienen el propósito de vender un producto). Ambos se basan en la predicción de la asociación entre un usuario y un elemento, ya sea para predecir la probabilidad de alguna acción (el usuario compra el producto o algún proxy para esta acción) o la ganancia esperada (que puede depender del valor del producto) si se muestra un anuncio o se hace una recomendación sobre ese producto a ese usuario. Actualmente, Internet está financiado en gran parte por diversas formas de publicidad en línea. Hay partes importantes de la economía que dependen de las compras en línea. Empresas como Amazon y eBay utilizan el aprendizaje automático, incluido el aprendizaje profundo, para sus recomendaciones de productos. A veces, los artículos no son productos que realmente estén a la venta. Los ejemplos incluyen seleccionar publicaciones para mostrar en las noticias de las redes sociales, recomendar películas para ver, recomendar chistes, recomendar consejos de expertos, emparejar jugadores para videojuegos o emparejar personas en servicios de citas.

A menudo, este problema de asociación se maneja como un problema de aprendizaje supervisado: dada cierta información sobre el elemento y sobre el usuario, predecir el proxy de interés (el usuario hace clic en el anuncio, el usuario ingresa una calificación, el usuario hace clic en el botón "Me gusta", el usuario compra un producto, el usuario gasta cierta cantidad de dinero en el producto, el usuario dedica tiempo a visitar una página del producto, etc.). Esto a menudo termina siendo un problema de regresión (prediciendo algún valor esperado condicional) o un problema de clasificación probabilística (prediciendo la probabilidad condicional de algún evento discreto).

Los primeros trabajos sobre los sistemas de recomendación se basaron en información mínima como entradas para estas predicciones: la identificación del usuario y la identificación del artículo. En este contexto, la única forma de generalizar es confiar en la similitud entre los patrones de valores de la variable objetivo para diferentes usuarios o para diferentes elementos. Suponga que al usuario 1 y al usuario 2 les gustan los elementos A, B y C. De esto, podemos inferir que al usuario 1 y

el usuario 2 tiene gustos similares. Si al usuario 1 le gusta el elemento D, esto debería ser una señal clara de que al usuario 2 también le gustará D. Los algoritmos basados en este principio se denominan **filtración colaborativa**. Son posibles tanto los enfoques no paramétricos (como los métodos del vecino más cercano basados en la similitud estimada entre patrones de preferencias) como los métodos paramétricos. Los métodos paramétricos a menudo se basan en el aprendizaje de una representación distribuida (también llamada incrustación) para cada usuario y para cada elemento. La predicción bilineal de la variable de destino (como una calificación) es un método paramétrico simple que tiene mucho éxito y, a menudo, se encuentra como un componente de los sistemas más avanzados. La predicción se obtiene mediante el producto escalar entre la incrustación del usuario y la incrustación del elemento (posiblemente corregida por constantes que dependen solo del ID del usuario o del ID del elemento). Dejar R sea la matriz que contiene nuestras predicciones, A una matriz con incrustaciones de usuario en sus filas y B una matriz con incrustaciones de elementos en sus columnas. Dejar b y C ser vectores que contengan respectivamente un tipo de sesgo para cada usuario (que representa cuán malhumorado o positivo es ese usuario en general) y para cada elemento (que representa su popularidad general). La predicción bilineal se obtiene así de la siguiente manera:

$$\hat{R}_{tu,yo} = b_{tu} + C_i + \sum_j A_{tu,j} B_{ji}. \quad (12.20)$$

Por lo general, uno quiere minimizar el error cuadrático entre las calificaciones pronosticadas $\hat{R}_{tu,yo}$ y calificaciones reales $R_{tu,yo}$. Las incorporaciones de usuarios y elementos se pueden visualizar convenientemente cuando se reducen por primera vez a una dimensión baja (dos o tres), o se pueden usar para comparar usuarios o elementos entre sí, al igual que las incrustaciones de palabras. Una forma de obtener estas incrustaciones es realizando una descomposición en valores singulares de la matriz R de los objetivos reales (como las calificaciones). Esto corresponde a la factorización $R=UDV$ (o una variante normalizada) en el producto de dos factores, las matrices de rango inferior $A=UD$ y $B=V$. Un problema con el SVD es que trata las entradas que faltan de manera arbitraria, como si correspondieran a un valor objetivo de 0. En su lugar, nos gustaría evitar pagar ningún costo por las predicciones realizadas sobre las entradas que faltan. Afortunadamente, la suma de los errores al cuadrado en las calificaciones observadas también se puede minimizar fácilmente mediante la optimización basada en gradientes. La SVD y la predicción bilineal de la ecuación 12.20 ambos se desempeñaron muy bien en la competencia por el premio Netflix ([Bennett y Lanning, 2007](#)), con el objetivo de predecir las calificaciones de las películas, basándose únicamente en las calificaciones anteriores de un gran conjunto de usuarios anónimos. Muchos expertos en aprendizaje automático participaron en esta competencia, que tuvo lugar entre 2006 y 2009. Elevó el nivel de investigación en sistemas de recomendación que utilizan aprendizaje automático avanzado y produjo mejoras en los sistemas de recomendación. Aunque no ganó por sí sola, la predicción bilineal simple o SVD fue un componente de los modelos de conjunto.

presentado por la mayoría de los competidores, incluidos los ganadores (Tösch et al., 2009; Koren, 2009).

Más allá de estos modelos bilineales con representaciones distribuidas, uno de los primeros usos de las redes neuronales para el filtrado colaborativo se basa en el modelo probabilístico no dirigido RBM (Salakhutdinov et al., 2007). Los RBM fueron un elemento importante del conjunto de métodos que ganó la competencia de Netflix (Tösch et al., 2009; Koren, 2009). También se han explorado variantes más avanzadas sobre la idea de factorizar la matriz de calificaciones en la comunidad de redes neuronales (Salakhutdinov y Mnih, 2008).

Sin embargo, existe una limitación básica de los sistemas de filtrado colaborativo: cuando se introduce un nuevo elemento o un nuevo usuario, su falta de historial de calificación significa que no hay forma de evaluar su similitud con otros elementos o usuarios (respectivamente), o el grado de asociación entre, digamos, ese nuevo usuario y elementos existentes. Esto se llama el problema de las recomendaciones de arranque en frío. Una forma general de resolver el problema de la recomendación de arranque en frío es introducir información adicional sobre los usuarios y elementos individuales. Por ejemplo, esta información adicional podría ser información del perfil de usuario o características de cada elemento. Los sistemas que utilizan dicha información se denominan **sistemas de recomendación basados en contenido**. La asignación de un amplio conjunto de funciones de usuario o funciones de elementos a una incrustación se puede aprender a través de una arquitectura de aprendizaje profundo (Huang et al., 2013; Elkahky et al., 2015).

También se han aplicado arquitecturas de aprendizaje profundo especializadas, como redes convolucionales, para aprender a extraer características de contenido enriquecido, como pistas de audio musicales, para recomendaciones musicales (van den Oord et al., 2013). En ese trabajo, la red convolucional toma características acústicas como entrada y calcula una incrustación para la canción asociada. El producto escalar entre la incrustación de esta canción y la incrustación de un usuario se usa para predecir si un usuario escuchará la canción.

12.5.1.1 Exploración versus Explotación

Al hacer recomendaciones a los usuarios, surge un problema que va más allá del aprendizaje supervisado ordinario y entra en el ámbito del aprendizaje por refuerzo. Muchos problemas de recomendación se describen con mayor precisión teóricamente como **bandidos contextuales** (Langford y Zhang, 2008; Lue et al., 2010). El problema es que cuando utilizamos el sistema de recomendaciones para recopilar datos, obtenemos una visión sesgada e incompleta de las preferencias de los usuarios: solo vemos las respuestas de los usuarios a los elementos que se recomendaron y no a los otros elementos. Además, en algunos casos, es posible que no obtengamos información sobre los usuarios para los que no se ha hecho ninguna recomendación (por ejemplo, con las subastas de anuncios, puede ser que el precio propuesto para un

el anuncio estaba por debajo de un umbral de precio mínimo o no gana la subasta, por lo que el anuncio no se muestra en absoluto). Más importante aún, no obtenemos información sobre qué resultado habría resultado de recomendar cualquiera de los otros elementos. Esto sería como entrenar a un clasificador eligiendo una clase y para cada ejemplo de entrenamiento X típicamente la clase con la probabilidad más alta de acuerdo con el modelo) y luego solo obtener como retroalimentación si esta era la clase correcta o no. Claramente, cada ejemplo transmite menos información que en el caso supervisado donde la etiqueta verdadera es directamente accesible, por lo que se necesitan más ejemplos. Peor aún, si no tenemos cuidado, podríamos terminar con un sistema que continúa eligiendo las decisiones equivocadas incluso cuando se recopilan más y más datos, porque la decisión correcta inicialmente tenía una probabilidad muy baja: hasta que el alumno toma la decisión correcta, no aprende acerca de la decisión correcta. Esto es similar a la situación en el aprendizaje por refuerzo donde solo se observa la recompensa por la acción seleccionada. En general, el aprendizaje por refuerzo puede implicar una secuencia de muchas acciones y muchas recompensas. El escenario de los bandidos es un caso especial de aprendizaje por refuerzo, en el que el alumno realiza una única acción y recibe una única recompensa. El problema del bandido es más fácil en el sentido de que el alumno sabe qué recompensa está asociada con qué acción. En el escenario general de aprendizaje por refuerzo, una recompensa alta o baja podría haber sido causada por una acción reciente o por una acción en el pasado lejano. El término **contextual** bandidos se refiere al caso en el que la acción se realiza en el contexto de alguna variable de entrada que puede informar la decisión. Por ejemplo, al menos conocemos la identidad del usuario y queremos elegir un elemento. El mapeo del contexto a la acción también se denomina **política**. El ciclo de retroalimentación entre el alumno y la distribución de datos (que ahora depende de las acciones del alumno) es un tema de investigación central en la literatura sobre aprendizaje por refuerzo y bandidos.

El aprendizaje por refuerzo requiere elegir un equilibrio entre **exploración** y **explotación**. La explotación se refiere a tomar acciones que provienen de la mejor versión actual de la política aprendida, acciones que sabemos que lograrán una gran recompensa. La exploración se refiere a tomar acciones específicamente para obtener más datos de entrenamiento. Si sabemos que el contexto dado X , acción a nos da una recompensa de 1, no sabemos si esa es la mejor recompensa posible. Es posible que queramos explotar nuestra política actual y seguir tomando medidas para estar relativamente seguros de obtener una recompensa de 1. Sin embargo, también podemos querer explorar probando acción a . No sabemos qué pasará si intentamos la acción a . Esperamos obtener una recompensa de 2, pero corremos el riesgo de obtener una recompensa de 0. De cualquier manera, al menos ganamos algo de conocimiento.

La exploración se puede implementar de muchas maneras, desde realizar acciones aleatorias ocasionales con la intención de cubrir todo el espacio de acciones posibles, hasta enfoques basados en modelos que calculan una elección de acción en función de su recompensa esperada y la cantidad de incertidumbre del modelo sobre esa recompensa.

Muchos factores determinan hasta qué punto preferimos la exploración o la explotación. Uno de los factores más destacados es la escala de tiempo que nos interesa. Si el agente tiene poco tiempo para acumular la recompensa, preferimos una mayor explotación. Si el agente tiene mucho tiempo para acumular la recompensa, comenzamos con más exploración para que las acciones futuras se puedan planificar de manera más efectiva con más conocimiento. A medida que pasa el tiempo y mejora nuestra política aprendida, avanzamos hacia una mayor explotación.

El aprendizaje supervisado no tiene compromiso entre exploración y explotación porque la señal de supervisión siempre especifica qué salida es correcta para cada entrada. No es necesario probar diferentes salidas para determinar si una es mejor que la salida actual del modelo; siempre sabemos que la etiqueta es la mejor salida.

Otra dificultad que surge en el contexto del aprendizaje por refuerzo, además de la compensación de exploración-explotación, es la dificultad de evaluar y comparar diferentes políticas. El aprendizaje por refuerzo implica la interacción entre el alumno y el entorno. Este ciclo de retroalimentación significa que no es sencillo evaluar el desempeño del alumno utilizando un conjunto fijo de valores de entrada del conjunto de prueba. La política misma determina qué entradas se verán.[Dudik et al.\(2011\)](#) presentan técnicas para evaluar bandidos contextuales.

12.5.2 Representación del conocimiento, razonamiento y respuesta a preguntas

Los enfoques de aprendizaje profundo han tenido mucho éxito en el modelado del lenguaje, la traducción automática y el procesamiento del lenguaje natural debido al uso de incrustaciones de símbolos ([Rumelhart et al., 1986a](#)) y palabras ([Deerwester et al., 1990;bengio et al., 2001](#)). Estas incrustaciones representan conocimiento semántico sobre palabras y conceptos individuales. Una frontera de investigación es desarrollar incorporaciones para frases y relaciones entre palabras y hechos. Los motores de búsqueda ya utilizan el aprendizaje automático para este propósito, pero aún queda mucho por hacer para mejorar estas representaciones más avanzadas.

12.5.2.1 Conocimiento, relaciones y respuesta a preguntas

Una dirección de investigación interesante es determinar cómo se pueden entrenar las representaciones distribuidas para capturar la **relaciones**entre dos entidades. Estas relaciones nos permiten formalizar hechos sobre los objetos y cómo los objetos interactúan entre sí.

En matemáticas, una **relación binaria**es un conjunto de pares ordenados de objetos. Se dice que los pares que están en el conjunto tienen la relación, mientras que los que no están en el conjunto

no. Por ejemplo, podemos definir la relación “es menor que” sobre el conjunto de entidades $\{1,2,3\}$ definiendo el conjunto de pares ordenados $S = \{(1,2), (1,3), (2,3)\}$. Una vez definida esta relación, podemos usarla como un verbo. Porque $(1,2) \in S$, decimos que 1 es menor que 2. Porque $(2,1) \notin S$, no podemos decir que 2 es menor que 1. Por supuesto, las entidades que se relacionan entre sí no tienen por qué ser números. Podríamos definir una relación `es_un_tipo_de` que contiene tuplas como (perro, mamífero).

En el contexto de la IA, pensamos en una relación como una oración en un lenguaje sintácticamente simple y altamente estructurado. La relación juega el papel de un verbo, mientras que dos argumentos de la relación juegan el papel de su sujeto y objeto. Estas oraciones toman la forma de un triplete de tokens

$$(sujeto, verbo, objeto) \quad (12.21)$$

con valores

$$(entidad_i, relación_j, entidad_k). \quad (12.22)$$

También podemos definir un **atributo**, un concepto análogo a una relación, pero tomando solo un argumento:

$$(entidad_i, atributo). \quad (12.23)$$

Por ejemplo, podríamos definir el `tiene_piel` atributo, y aplicarlo a entidades como perro.

Muchas aplicaciones requieren representar relaciones y razonar sobre ellas. ¿Cuál es la mejor manera de hacer esto dentro del contexto de las redes neuronales?

Los modelos de aprendizaje automático, por supuesto, requieren datos de entrenamiento. Podemos inferir relaciones entre entidades a partir de conjuntos de datos de entrenamiento que consisten en lenguaje natural no estructurado. También existen bases de datos estructuradas que identifican relaciones explícitamente. Una estructura común para estas bases de datos es la **base de datos relacional**, que almacena este mismo tipo de información, aunque no formateada como tres oraciones simbólicas. Cuando una base de datos pretende transmitir conocimiento de sentido común sobre la vida cotidiana o conocimiento experto sobre un área de aplicación a un sistema de inteligencia artificial, llamamos a la base de datos un **base de conocimientos**. Las bases de conocimiento van desde las generales como Base libre, OpenCyc, WordNet, owlkbase,¹ etc. a bases de conocimiento más especializadas, como Ontología de genes.² Las representaciones de entidades y relaciones se pueden aprender considerando cada triplete en una base de conocimiento como un ejemplo de entrenamiento y maximizando un objetivo de entrenamiento que capture su distribución conjunta (*Bordes et al., 2013a*).

¹Disponible respectivamente en estos sitios web: freebase.com, cyc.com/opencyc, [red de palabras.princeton.edu](http://reddepalabras.princeton.edu), wikiba.se

²geneontology.org

Además de los datos de entrenamiento, también necesitamos definir una familia modelo para entrenar. Un enfoque común es extender los modelos de lenguaje neuronal para modelar entidades y relaciones. Los modelos de lenguaje neuronal aprenden un vector que proporciona una representación distribuida de cada palabra. También aprenden sobre interacciones entre palabras, como qué palabra es probable que venga después de una secuencia de palabras, aprendiendo las funciones de estos vectores. Podemos extender este enfoque a entidades y relaciones aprendiendo un vector incrustado para cada relación. De hecho, el paralelismo entre el lenguaje de modelado y el conocimiento de modelado codificado como relaciones es tan cercano que los investigadores han entrenado representaciones de tales entidades usando *ambos* bases de conocimiento y oraciones en lenguaje natural ([Bordes et al., 2011, 2012](#); [Wang et al., 2014a](#)) o combinando datos de múltiples bases de datos relacionales ([Bordes et al., 2013b](#)). Existen muchas posibilidades para la parametrización particular asociada con dicho modelo. Primeros trabajos sobre el aprendizaje de las relaciones entre entidades ([Paccanaro y Hinton, 2000](#)) postuló formas paramétricas altamente restringidas ("incrustaciones relacionales lineales"), a menudo utilizando una forma diferente de representación para la relación que para las entidades. Por ejemplo, [Paccanaro y Hinton\(2000\)](#) y [Bordes et al.\(2011\)](#) utilizaron vectores para entidades y matrices para relaciones, con la idea de que una relación actúa como un operador sobre entidades. Alternativamente, las relaciones pueden ser consideradas como cualquier otra entidad ([Bordes et al., 2012](#)), lo que nos permite hacer afirmaciones sobre las relaciones, pero se dota de mayor flexibilidad a la maquinaria que las combina para modelar su distribución conjunta.

Una aplicación práctica a corto plazo de tales modelos es **predicción de enlaces**: predicción de arcos faltantes en el gráfico de conocimiento. Esta es una forma de generalización a hechos nuevos, basada en hechos antiguos. La mayoría de las bases de conocimiento que existen actualmente se han construido a través del trabajo manual, lo que tiende a dejar muchas y probablemente la mayoría de las verdaderas relaciones ausentes de la base de conocimiento. Ver [Wang et al. \(2014b\)](#), [Linet al.\(2015\)](#) y [García-Duránet al.\(2015\)](#) para ver ejemplos de tal aplicación.

Evaluar el rendimiento de un modelo en una tarea de predicción de enlaces es difícil porque solo tenemos un conjunto de datos de ejemplos positivos (hechos que se sabe que son ciertos). Si el modelo propone un hecho que no está en el conjunto de datos, no estamos seguros de si el modelo ha cometido un error o ha descubierto un nuevo hecho previamente desconocido. Por lo tanto, las métricas son algo imprecisas y se basan en probar cómo el modelo clasifica un conjunto retenido de hechos positivos verdaderos conocidos en comparación con otros hechos que tienen menos probabilidades de ser ciertos. Una forma común de construir ejemplos interesantes que probablemente sean negativos (hechos que probablemente sean falsos) es comenzar con un hecho verdadero y crear versiones corruptas de ese hecho, por ejemplo, reemplazando una entidad en la relación con una entidad diferente seleccionada al azar.

Otra aplicación de bases de conocimiento y representaciones distribuidas para ellas es **desambiguación del sentido de las palabras**([Navigli y Velardi,2005;Bordes et al., 2012](#)), que es la tarea de decidir cuál de los sentidos de una palabra es el adecuado, en algún contexto.

Eventualmente, el conocimiento de las relaciones combinado con un proceso de razonamiento y comprensión del lenguaje natural podría permitirnos construir un sistema general de respuesta a preguntas. Un sistema general de respuesta a preguntas debe ser capaz de procesar información de entrada y recordar hechos importantes, organizados de una manera que le permita recuperarlos y razonar sobre ellos más tarde. Este sigue siendo un problema abierto difícil que solo puede resolverse en entornos de "juguete" restringidos. Actualmente, el mejor enfoque para recordar y recuperar hechos declarativos específicos es usar un mecanismo de memoria explícita, como se describe en la sección[10.12](#). Las redes de memoria se propusieron por primera vez para resolver una tarea de respuesta a preguntas de juguete ([Weston et al.,2014](#)).[Kumar et al/\(2015\)](#) han propuesto una extensión que usa redes recurrentes GRU para leer la entrada en la memoria y producir la respuesta dado el contenido de la memoria.

El aprendizaje profundo se ha aplicado a muchas otras aplicaciones además de las descritas aquí, y seguramente se aplicará a muchas más después de escribir este artículo. Sería imposible describir algo remotamente parecido a una cobertura completa de tal tema. Esta encuesta proporciona una muestra representativa de lo que es posible a partir de este escrito.

Esto concluye parte[Yo](#), que ha descrito prácticas modernas que involucran redes profundas, que comprenden todos los métodos más exitosos. En términos generales, estos métodos implican el uso del gradiente de una función de costo para encontrar los parámetros de un modelo que se aproxima a alguna función deseada. Con suficientes datos de entrenamiento, este enfoque es extremadamente poderoso. Pasamos ahora a la parte[tercero](#), en el que entramos en el territorio de la investigación, métodos que están diseñados para trabajar con menos datos de entrenamiento o para realizar una mayor variedad de tareas, donde los desafíos son más difíciles y no tan cerca de resolverse como las situaciones que hemos descrito hasta ahora. lejos.