
CAPÍTULO 8. OPTIMIZACIÓN PARA EL ENTRENAMIENTO DE MODELOS PROFUNDOS

ser cabezas. Ver [Delfínet al.\(2014\)](#) para una revisión del trabajo teórico relevante.

Una propiedad sorprendente de muchas funciones aleatorias es que los valores propios de la hessiana se vuelven más positivos a medida que alcanzamos regiones de menor costo. En nuestra analogía de lanzar una moneda, esto significa que es más probable que nuestra moneda salga cara. *norte*veces si estamos en un punto crítico con bajo coste. Esto significa que es mucho más probable que los mínimos locales tengan un costo bajo que un costo alto. Es mucho más probable que los puntos críticos con un alto costo sean puntos de silla. Es más probable que los puntos críticos con un costo extremadamente alto sean máximos locales.

Esto sucede para muchas clases de funciones aleatorias. ¿Sucede para las redes neuronales? [Baldi y Hornik\(1989\)](#) mostró teóricamente que los autocodificadores superficiales (redes de retroalimentación entrenadas para copiar su entrada a su salida, descritas en el capítulo 14) sin no linealidades tienen mínimos globales y puntos de silla, pero no hay mínimos locales con un costo más alto que el mínimo global. Observaron sin pruebas que estos resultados se extienden a redes más profundas sin no linealidades. La salida de tales redes es una función lineal de su entrada, pero son útiles para estudiarlas como modelo de redes neuronales no lineales porque su función de pérdida es una función no convexa de sus parámetros. Tales redes son esencialmente matrices múltiples compuestas juntas. [Sajonia et al.\(2013\)](#) proporcionó soluciones exactas a la dinámica de aprendizaje completa en tales redes y mostró que el aprendizaje en estos modelos captura muchas de las características cualitativas observadas en el entrenamiento de modelos profundos con funciones de activación no lineales. [Delfínet al.\(2014\)](#) mostró experimentalmente que las redes neuronales reales también tienen funciones de pérdida que contienen muchos puntos de silla de alto costo. [Choromanska et al.\(2014\)](#) proporcionó argumentos teóricos adicionales, mostrando que otra clase de funciones aleatorias de alta dimensión relacionadas con las redes neuronales también lo hacen.

¿Cuáles son las implicaciones de la proliferación de puntos de silla para los algoritmos de entrenamiento? Para los algoritmos de optimización de primer orden que usan solo información de gradiente, la situación no está clara. El gradiente a menudo puede volverse muy pequeño cerca de un punto de silla. Por otro lado, el descenso de gradiente empíricamente parece ser capaz de escapar de los puntos de silla en muchos casos. [Buen compañero et al.\(2015\)](#) proporcionaron visualizaciones de varias trayectorias de aprendizaje de redes neuronales de última generación, con un ejemplo dado en la figura 8.2. Estas visualizaciones muestran un aplanamiento de la función de costo cerca de un punto de silla prominente donde los pesos son todos cero, pero también muestran la trayectoria de descenso del gradiente que escapa rápidamente de esta región. [Buen compañero et al.\(2015\)](#) también argumentan que se puede demostrar analíticamente que el descenso de gradiente en tiempo continuo es repelido, en lugar de atraído, por un punto de silla cercano, pero la situación puede ser diferente para usos más realistas del descenso de gradiente.

Para el método de Newton, está claro que los puntos de silla constituyen un problema.

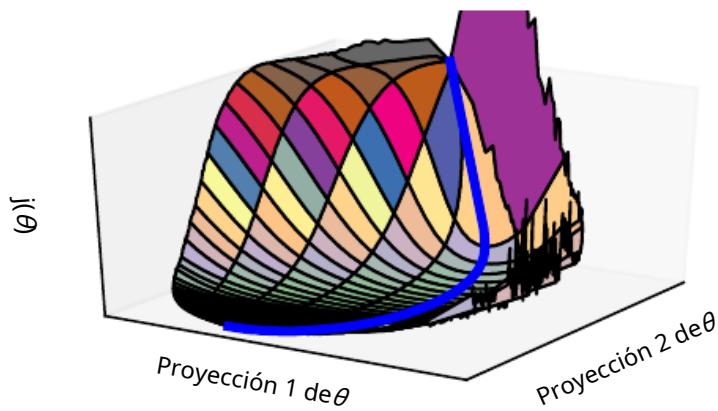


Figura 8.2: Una visualización de la función de costo de una red neuronal. Imagen adaptada con permiso de [Buen compañero et al.](#)(2015). Estas visualizaciones parecen similares para las redes neuronales feedforward, redes convolucionales y redes recurrentes aplicadas al reconocimiento de objetos reales y tareas de procesamiento de lenguaje natural. Sorprendentemente, estas visualizaciones no suelen mostrar muchos obstáculos evidentes. Antes del éxito del descenso de gradiente estocástico para entrenar modelos muy grandes a partir de aproximadamente 2012, generalmente se creía que las superficies de función de costo de red neuronal tenían una estructura mucho más no convexa de lo que revelan estas proyecciones. El principal obstáculo revelado por esta proyección es un punto de silla de alto costo cerca de donde se inicializan los parámetros, pero, como lo indica el camino azul, la trayectoria de entrenamiento SGD escapa fácilmente de este punto de silla. La mayor parte del tiempo de entrenamiento se dedica a atravesar el valle relativamente plano de la función de costo,

El descenso de gradiente está diseñado para moverse "cuesta abajo" y no está diseñado explícitamente para buscar un punto crítico. El método de Newton, sin embargo, está diseñado para resolver un punto donde el gradiente es cero. Sin la modificación apropiada, puede saltar a un punto de silla. La proliferación de puntos de silla en espacios de alta dimensión probablemente explica por qué los métodos de segundo orden no han logrado reemplazar el descenso de gradiente para el entrenamiento de redes neuronales. [Delfín et al. \(2014\)](#) introdujo un **método de Newton sin silla de montar** para la optimización de segundo orden y mostró que mejora significativamente con respecto a la versión tradicional. Los métodos de segundo orden siguen siendo difíciles de escalar a grandes redes neuronales, pero este enfoque sin montura es prometedor si pudiera escalarse.

Hay otros tipos de puntos con gradiente cero además de mínimos y puntos de silla. También hay máximos, que son muy parecidos a los puntos de silla desde la perspectiva de la optimización: muchos algoritmos no se sienten atraídos por ellos, pero el método de Newton sin modificar sí. Los máximos de muchas clases de funciones aleatorias se vuelven exponencialmente raros en el espacio de alta dimensión, al igual que los mínimos.

También puede haber regiones anchas y planas de valor constante. En estos lugares, el gradiente y también la arpillería son todos cero. Estas ubicaciones degeneradas plantean grandes problemas para todos los algoritmos de optimización numérica. En un problema convexo, una región amplia y plana debe consistir enteramente en mínimos globales, pero en un problema de optimización general, dicha región podría corresponder a un valor alto de la función objetivo.

8.2.4 Acantilados y Explosión de Gradientes

Las redes neuronales con muchas capas a menudo tienen regiones extremadamente empinadas que se asemejan a acantilados, como se ilustra en la figura [8.3](#). Estos resultan de la multiplicación de varios pesos grandes juntos. En la cara de una estructura de acantilado extremadamente empinada, el paso de actualización de gradiente puede mover los parámetros extremadamente lejos, por lo general saltando fuera de la estructura del acantilado por completo.

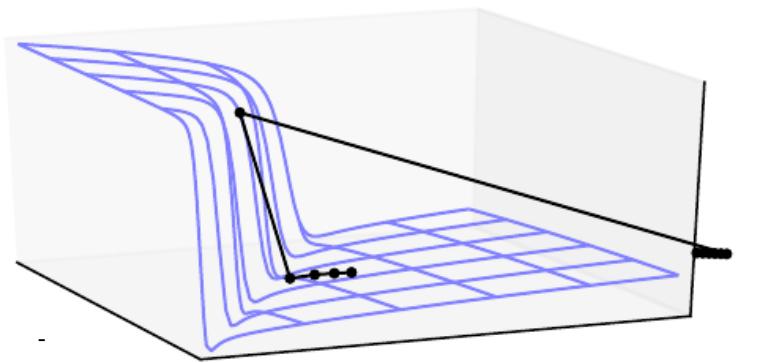


Figura 8.3: La función objetivo para redes neuronales profundas altamente no lineales o para redes neuronales recurrentes a menudo contiene marcadas no linealidades en el espacio de parámetros que resultan de la multiplicación de varios parámetros. Estas no linealidades dan lugar a derivadas muy altas en algunos lugares. Cuando los parámetros se acercan a tal región de acantilado, una actualización de descenso de gradiente puede catapultar los parámetros muy lejos, posiblemente perdiendo la mayor parte del trabajo de optimización que se había realizado. Figura adaptada con permiso de [Pascanu et al.](#)(2013).

El acantilado puede ser peligroso tanto si nos acercamos a él desde arriba como desde abajo, pero afortunadamente sus consecuencias más graves se pueden evitar utilizando el **recorte de degradado** heurística descrita en la sección 10.11.1. La idea básica es recordar que el gradiente no especifica el tamaño de paso óptimo, sino solo la dirección óptima dentro de una región infinitesimal. Cuando el algoritmo de descenso de gradiente tradicional propone dar un paso muy grande, la heurística de recorte de gradiente interviene para reducir el tamaño del paso para que sea lo suficientemente pequeño como para que sea menos probable que salga de la región donde el gradiente indica la dirección del descenso aproximadamente más pronunciado. Las estructuras de acantilados son más comunes en las funciones de costo de las redes neuronales recurrentes, porque dichos modelos involucran una multiplicación de muchos factores, con un factor para cada paso de tiempo. Las secuencias temporales largas incurren así en una cantidad extrema de multiplicación.

8.2.5 Dependencias a largo plazo

Otra dificultad que deben superar los algoritmos de optimización de redes neuronales surge cuando el gráfico computacional se vuelve extremadamente profundo. Las redes feedforward con muchas capas tienen gráficos computacionales tan profundos. También lo hacen las redes recurrentes, descritas en el capítulo 10, que construye gráficos computacionales muy profundos

aplicando repetidamente la misma operación en cada paso de tiempo de una secuencia temporal larga. La aplicación repetida de los mismos parámetros da lugar a dificultades especialmente pronunciadas.

Por ejemplo, supongamos que un gráfico computacional contiene un camino que consiste en multiplicar repetidamente por una matriz W . Despuéspasos, esto es equivalente a multiplicar por W_t . Suponer que W tiene una descomposición propia $W=V\text{diag}(\lambda)V^{-1}$. En este caso simple, es fácil ver que

$$W_t = V\text{diag}(\lambda)V^{-1}t = V\text{diag}(\lambda)_tV^{-1}. \quad (8.11)$$

Cualquier valor propio λ que no están cerca de un valor absoluto de 1 explotarán si son mayores que 1 en magnitud o desaparecen si son menores que 1 en magnitud. El **problema de gradiente de desaparición y explosión** se refiere al hecho de que los gradientes a través de dicho gráfico también se escalan de acuerdo con $\text{diag}(\lambda)_t$. Los gradientes que se desvanecen dificultan saber en qué dirección deben moverse los parámetros para mejorar la función de costo, mientras que los gradientes explosivos pueden hacer que el aprendizaje sea inestable. Las estructuras de los acantilados descritas anteriormente que motivan el recorte del gradiente son un ejemplo del fenómeno del gradiente explosivo.

La multiplicación repetida por W en cada paso de tiempo descrito aquí es muy similar al **método de poder**algoritmo utilizado para encontrar el valor propio más grande de una matriz W y el vector propio correspondiente. Desde este punto de vista, no es de extrañar que X - W eventualmente desechará todos los componentes de X que son ortogonales al vector propio principal de W .

Las redes recurrentes usan la misma matriz W en cada paso de tiempo, pero las redes feedforward no lo hacen, por lo que incluso las redes feedforward muy profundas pueden evitar en gran medida el problema del gradiente que desaparece y explota ([sussillo, 2014](#)).

Posponemos una discusión adicional sobre los desafíos de entrenar redes recurrentes hasta la sección [10.7](#), después de que las redes recurrentes se hayan descrito con más detalle.

8.2.6 Gradientes inexactos

La mayoría de los algoritmos de optimización están diseñados asumiendo que tenemos acceso al gradiente exacto o matriz hessiana. En la práctica, generalmente solo tenemos una estimación ruidosa o incluso sesgada de estas cantidades. Casi todos los algoritmos de aprendizaje profundo se basan en estimaciones basadas en muestreo, al menos en la medida en que utilizan un minilote de ejemplos de entrenamiento para calcular el gradiente.

En otros casos, la función objetivo que queremos minimizar es en realidad intratable. Cuando la función objetivo es intratable, normalmente su gradiente también lo es. En tales casos, solo podemos aproximar el gradiente. Estos problemas surgen principalmente

con los modelos más avanzados en parte **tercero**. Por ejemplo, la divergencia contrastiva proporciona una técnica para aproximar el gradiente de la log-verosimilitud intratable de una máquina de Boltzmann.

Varios algoritmos de optimización de redes neuronales están diseñados para tener en cuenta las imperfecciones en la estimación del gradiente. También se puede evitar el problema eligiendo una función de pérdida sustituta que sea más fácil de aproximar que la pérdida real.

8.2.7 Mala correspondencia entre la estructura local y global

Muchos de los problemas que hemos discutido hasta ahora corresponden a propiedades de la función de pérdida en un solo punto; puede ser difícil dar un solo paso si $\nabla(\theta)$ está mal acondicionado en el punto actual θ , o si θ se encuentra en un acantilado, o si θ es un punto de silla que oculta la oportunidad de progresar cuesta abajo desde la pendiente.

Es posible superar todos estos problemas en un solo punto y aun así tener un desempeño deficiente si la dirección que resulta en la mayor mejora a nivel local no apunta hacia regiones distantes de mucho menor costo.

Buen compañero *et al.* (2015) argumentan que gran parte del tiempo de ejecución del entrenamiento se debe a la longitud de la trayectoria necesaria para llegar a la solución. Cifra 8.2 muestra que la trayectoria de aprendizaje pasa la mayor parte de su tiempo trazando un amplio arco alrededor de una estructura en forma de montaña.

Gran parte de la investigación sobre las dificultades de la optimización se ha centrado en si el entrenamiento llega a un mínimo global, un mínimo local o un punto de silla, pero en la práctica las redes neuronales no llegan a un punto crítico de ningún tipo. Cifra 8.1 muestra que las redes neuronales a menudo no llegan a una región de pequeño gradiente. De hecho, tales puntos críticos ni siquiera existen necesariamente. Por ejemplo, la función de pérdida $-\text{registro}_{\text{pag}}(y/X;\theta)$ puede carecer de un punto mínimo global y, en cambio, acercarse asintóticamente a algún valor a medida que el modelo se vuelve más seguro. Para un clasificador con discreto y $\text{y}_{\text{pag}}(y/X)$ proporcionado por un softmax, la probabilidad logarítmica negativa puede volverse arbitrariamente cercana a cero si el modelo puede clasificar correctamente cada ejemplo en el conjunto de entrenamiento, pero es imposible alcanzar el valor cero. Asimismo, un modelo de valores reales $\text{pag}(y/X) = \text{norte}(y, F(\theta), \beta^{-1})$ puede tener una log-verosimilitud negativa que da asintotas al infinito negativo, si $F(\theta)$ es capaz de predecir correctamente el valor de todos los conjuntos de entrenamiento y objetivos, el algoritmo de aprendizaje aumentará β sin límite. Ver figura 8.4 para ver un ejemplo de una falla de optimización local para encontrar un buen valor de función de costo incluso en ausencia de mínimos locales o puntos de silla.

La investigación futura deberá desarrollar una mayor comprensión de los factores que influyen en la duración de la trayectoria de aprendizaje y caracterizar mejor el resultado.

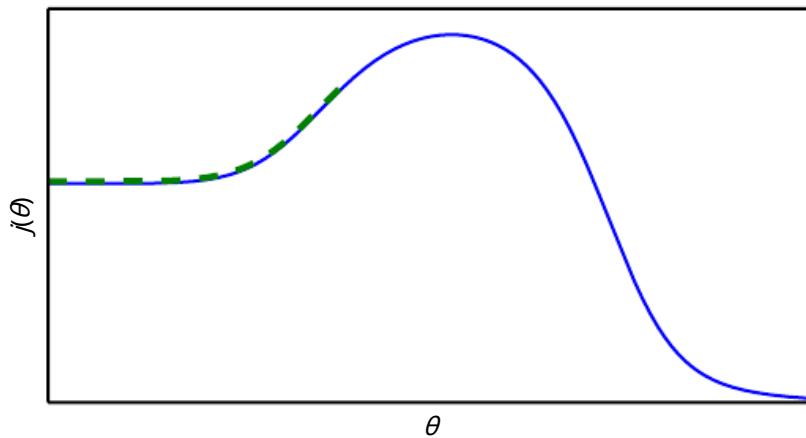


Figura 8.4: La optimización basada en movimientos cuesta abajo locales puede fallar si la superficie local no apunta hacia la solución global. Aquí proporcionamos un ejemplo de cómo puede ocurrir esto, incluso si no hay puntos de silla ni mínimos locales. Esta función de costo de ejemplo contiene solo asíntotas hacia valores bajos, no mínimos. La principal causa de dificultad en este caso es estar inicializado en el lado equivocado de la "montaña" y no poder atravesarla. En el espacio dimensional superior, los algoritmos de aprendizaje a menudo pueden circumnavegar tales montañas, pero la trayectoria asociada con hacerlo puede ser larga y resultar en un tiempo de entrenamiento excesivo, como se ilustra en la figura.8.2.

del proceso

Muchas direcciones de investigación existentes tienen como objetivo encontrar buenos puntos iniciales para problemas que tienen una estructura global difícil, en lugar de desarrollar algoritmos que utilicen movimientos no locales.

El descenso de gradiente y, esencialmente, todos los algoritmos de aprendizaje que son efectivos para entrenar redes neuronales se basan en realizar pequeños movimientos locales. Las secciones anteriores se han centrado principalmente en cómo la dirección correcta de estos movimientos locales puede ser difícil de calcular. Es posible que podamos calcular algunas propiedades de la función objetivo, como su gradiente, solo aproximadamente, con sesgo o varianza en nuestra estimación de la dirección correcta. En estos casos, la descendencia local puede o no definir un camino razonablemente corto hacia una solución válida, pero en realidad no podemos seguir el camino de descendencia local. La función objetivo puede tener problemas como un condicionamiento deficiente o gradientes discontinuos, lo que hace que la región donde el gradiente proporciona un buen modelo de la función objetivo sea muy pequeña. En estos casos, descendencia local con escalones de tamaño-puede definir un camino razonablemente corto a la solución, pero solo podemos calcular la dirección de descenso local con pasos de tamaño $d - \cdot$. En estos casos, la ascendencia local puede definir o no un camino hacia la solución, pero el camino contiene muchos pasos, por lo que seguir el camino incurre en un problema.

alto costo computacional. A veces, la información local no nos proporciona una guía, cuando la función tiene una región plana amplia, o si logramos aterrizar exactamente en un punto crítico (por lo general, este último escenario solo ocurre con los métodos que resuelven explícitamente los puntos críticos, como el método de Newton). En estos casos, la ascendencia local no define en absoluto un camino hacia una solución. En otros casos, los movimientos locales pueden ser demasiado codiciosos y llevarnos por un camino cuesta abajo pero alejado de cualquier solución, como en la figura 8.4, o a lo largo de una trayectoria innecesariamente larga hacia la solución, como en la figura 8.2. Actualmente, no entendemos cuáles de estos problemas son más relevantes para dificultar la optimización de redes neuronales, y esta es un área activa de investigación.

Independientemente de cuál de estos problemas sea más significativo, todos ellos podrían evitarse si existe una región del espacio conectada razonablemente directamente a una solución por un camino que puede seguir la descendencia local, y si somos capaces de iniciar el aprendizaje dentro de ese pozo. región comportada. Este último punto de vista sugiere investigar la elección de buenos puntos iniciales para que los utilicen los algoritmos de optimización tradicionales.

8.2.8 Límites teóricos de optimización

Varios resultados teóricos muestran que existen límites en el rendimiento de cualquier algoritmo de optimización que podamos diseñar para redes neuronales (Blum y Rivest, 1992; Judd, 1989; Wolpert y MacReady, 1997). Por lo general, estos resultados tienen poca relación con el uso de redes neuronales en la práctica.

Algunos resultados teóricos se aplican solo al caso en el que las unidades de una red neuronal generan valores discretos. Sin embargo, la mayoría de las unidades de redes neuronales emiten valores que aumentan gradualmente, lo que hace factible la optimización a través de la búsqueda local. Algunos resultados teóricos muestran que existen clases de problemas que son intratables, pero puede ser difícil saber si un problema particular cae dentro de esa clase. Otros resultados muestran que encontrar una solución para una red de un tamaño dado es intratable, pero en la práctica podemos encontrar una solución fácilmente usando una red más grande para la cual muchas más configuraciones de parámetros corresponden a una solución aceptable. Además, en el contexto del entrenamiento de redes neuronales, generalmente no nos preocupamos por encontrar el mínimo exacto de una función, sino que solo buscamos reducir su valor lo suficiente como para obtener un buen error de generalización. El análisis teórico de si un algoritmo de optimización puede lograr este objetivo es extremadamente difícil. Por lo tanto, desarrollar límites más realistas en el rendimiento de los algoritmos de optimización sigue siendo un objetivo importante para la investigación del aprendizaje automático.

8.3 Algoritmos básicos

Anteriormente hemos introducido el descenso de gradiente (sección 4.3) algoritmo que sigue el gradiente de un conjunto de entrenamiento completo cuesta abajo. Esto se puede acelerar considerablemente mediante el uso de descenso de gradiente estocástico para seguir el gradiente de minilotes seleccionados al azar cuesta abajo, como se analiza en la sección 5.9 y sección 8.1.3.

8.3.1 Descenso de gradiente estocástico

El descenso de gradiente estocástico (SGD) y sus variantes son probablemente los algoritmos de optimización más utilizados para el aprendizaje automático en general y para el aprendizaje profundo en particular. Como se discutió en la sección 8.1.3, es posible obtener una estimación no sesgada del gradiente tomando el gradiente promedio en un minilote de m ejemplos extraídos iid de la distribución de generación de datos.

Algoritmo 8.1 muestra cómo seguir esta estimación de la pendiente cuesta abajo.

Algoritmo 8.1 Actualización de descenso de gradiente estocástico (SGD) en la iteración de entrenamiento k

Requerir:Tasa de aprendizaje $-k$.

Requerir:Parámetro inicial θ

mientras no se cumple el criterio de parada **hacer**

Muestra un mini lote de m ejemplos del conjunto de entrenamiento $\{X(1), \dots, X(m)\}$ con los objetivos correspondientes $y(i)$. Calcular estimación de gradiente: $\hat{g} \leftarrow \frac{1}{m} \sum_{i=1}^m L(F(X(i); \theta), y(i))$

Aplicar actualización: $\theta \leftarrow \theta - \eta \hat{g}$

terminar **mientras**

Un parámetro crucial para el algoritmo SGD es la tasa de aprendizaje. Anteriormente, describimos SGD como usando una tasa de aprendizaje fija-. En la práctica, es necesario disminuir gradualmente la tasa de aprendizaje con el tiempo, por lo que ahora denotamos la tasa de aprendizaje en la iteración k como $-k$.

Esto se debe a que el estimador de gradiente SGD introduce una fuente de ruido (el muestreo aleatorio de m ejemplos de entrenamiento) que no se desvanece ni siquiera cuando llegamos a un mínimo. En comparación, el verdadero gradiente de la función de costo total se vuelve pequeño y luego 0 cuando nos acercamos y alcanzamos un mínimo utilizando el descenso de gradiente por lotes, por lo que el descenso de gradiente por lotes puede usar una tasa de aprendizaje fija. Una condición suficiente para garantizar la convergencia de SGD es que

$$\lim_{k \rightarrow \infty} -k = 0, \quad (8.12)$$

$$\sum_{k=1}^{\infty} -\alpha_k < \infty. \quad (8.13)$$

En la práctica, es común disminuir la tasa de aprendizaje linealmente hasta que la iteración τ :

$$-\alpha_k = (1 - \alpha_0) - \alpha(\tau - \tau_0) \quad (8.14)$$

con $\alpha = k^{-\frac{1}{\tau}}$. Después de la iteración τ , es común salir constante.

La tasa de aprendizaje puede elegirse por prueba y error, pero generalmente es mejor elegirla monitoreando las curvas de aprendizaje que trazan la función objetivo como una función del tiempo. Esto es más un arte que una ciencia, y la mayoría de las guías sobre este tema deben considerarse con cierto escepticismo. Cuando se utiliza el programa lineal, los parámetros a elegir son α_0, α, τ . Generalmente τ puede establecerse en el número de iteraciones necesarias para realizar unos cientos de pasadas a través del conjunto de entrenamiento. Generalmente α debe establecerse aproximadamente 1% del valor de α_0 . La cuestión principal es cómo configurar α . Si es demasiado grande, la curva de aprendizaje mostrará oscilaciones violentas, y la función de costo a menudo aumentará significativamente. Las oscilaciones suaves están bien, especialmente si se entrena con una función de costo estocástica como la función de costo que surge del uso de la deserción. Si la tasa de aprendizaje es demasiado baja, el aprendizaje avanza lentamente, y si la tasa de aprendizaje inicial es demasiado baja, el aprendizaje puede quedarse estancado con un valor de costo alto. Por lo general, la tasa de aprendizaje inicial óptima, en términos de tiempo total de capacitación y el valor del costo final, es más alta que la tasa de aprendizaje que produce el mejor rendimiento después de las primeras 100 iteraciones más o menos. Por lo tanto, generalmente es mejor monitorear las primeras iteraciones y usar una tasa de aprendizaje que sea más alta que la tasa de aprendizaje de mejor desempeño en este momento, pero no tan alta como para causar una inestabilidad severa.

La propiedad más importante de SGD y la optimización basada en gradientes en línea o en mini-lotes relacionados es que el tiempo de cálculo por actualización no crece con la cantidad de ejemplos de entrenamiento. Esto permite la convergencia incluso cuando el número de ejemplos de entrenamiento se vuelve muy grande. Para un conjunto de datos lo suficientemente grande, SGD puede converger dentro de una tolerancia fija de su error de conjunto de prueba final antes de que haya procesado todo el conjunto de entrenamiento.

Para estudiar la tasa de convergencia de un algoritmo de optimización es común medir la **exceso de error** $J(\theta) - \min_{\theta} J(\theta)$, que es la cantidad en que la función de costo actual excede el costo mínimo posible. Cuando SGD se aplica a un problema convexo, el exceso de error es $O(\sqrt{\epsilon})$ después de $\sqrt{\epsilon}$ iteraciones, mientras que en el fuertemente convexo caso es $O(1/\sqrt{\epsilon})$. Estos límites no se pueden mejorar a menos que se den condiciones adicionales. El descenso de gradiente por lotes disfruta de mejores tasas de convergencia que el descenso de gradiente estocástico en teoría. Sin embargo, el límite de Cramér-Rao (Cramér, 1946; Rao, 1945) establece que el error de generalización no puede disminuir más rápido que $O(1/\sqrt{k})$.

y ramo(2008) argumentan que, por lo tanto, puede no valer la pena seguir un algoritmo de optimización que converge más rápido que $O(1/\sqrt{k})$ para el aprendizaje automático. Tareas: una convergencia más rápida presumiblemente corresponde a un sobreajuste. Además, el análisis asintótico oscurece muchas ventajas que tiene el descenso de gradiente estocástico después de un pequeño número de pasos. Con grandes conjuntos de datos, la capacidad de SGD para hacer un rápido progreso inicial mientras evalúa el gradiente solo para unos pocos ejemplos supera su lenta convergencia asintótica. La mayoría de los algoritmos descritos en el resto de este capítulo logran beneficios que importan en la práctica pero que se pierden en los factores constantes oscurecidos por la $O(1/\sqrt{k})$ análisis asintótico. Uno también puede compensar los beneficios del descenso de gradiente estocástico y por lotes aumentando gradualmente el tamaño del minibatch durante el curso del aprendizaje.

Para obtener más información sobre SGD, consulte fondo(1998).

8.3.2 Momento

Si bien el descenso de gradiente estocástico sigue siendo una estrategia de optimización muy popular, aprender con ella a veces puede ser lento. El método del impulso (polica, 1964) está diseñado para acelerar el aprendizaje, especialmente frente a curvaturas altas, gradientes pequeños pero consistentes o gradientes ruidosos. El algoritmo de impulso acumula un promedio móvil exponencialmente decreciente de gradientes pasados y continúa moviéndose en su dirección. El efecto del impulso se ilustra en la figura 8.5.

Formalmente, el algoritmo de cantidad de movimiento introduce una variable v que juega el papel de la velocidad: es la dirección y la velocidad a la que los parámetros se mueven a través del espacio de parámetros. La velocidad se establece en un promedio exponencialmente decreciente del gradiente negativo. El nombre **impulso** deriva de una analogía física, en la que el gradiente negativo es una fuerza que mueve una partícula a través del espacio de parámetros, de acuerdo con las leyes de movimiento de Newton. El momento en física es masa por velocidad. En el algoritmo de aprendizaje de cantidad de movimiento, asumimos una unidad de masa, por lo que el vector de velocidad v también puede considerarse como el momento de la partícula. Un hiperparámetro $\alpha \in [0, 1)$ determina qué tan rápido decaen exponencialmente las contribuciones de los gradientes anteriores. La regla de actualización viene dada por:

$$v \leftarrow \alpha v - \nabla_{\theta} \text{ metro} - \frac{1}{\beta+1} \sum_{i=1}^{\beta} L(F(X^{(i)}; \theta), y^{(i)}), \quad (8.15)$$

$$\theta \leftarrow \theta + v. \quad (8.16)$$

La velocidad v acumula los elementos de gradiente $\nabla_{\theta} \text{ metro}$ $\sum_{i=1}^{\beta} L(F(X^{(i)}; \theta), y^{(i)})$. El factor α es relativo a β , cuantos más gradientes anteriores afecten a la dirección actual. El algoritmo SGD con impulso se da en el algoritmo 8.2.

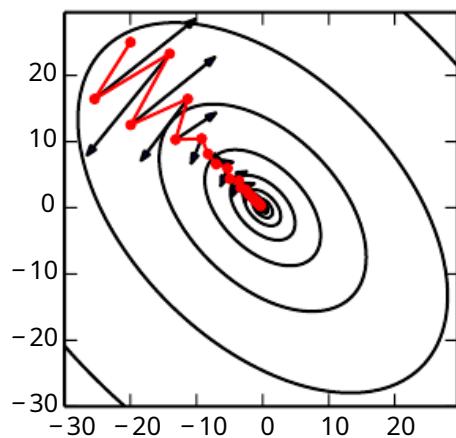


Figura 8.5: Momentum apunta principalmente a resolver dos problemas: mal condicionamiento de la matriz Hessiana y varianza en el gradiente estocástico. Aquí, ilustramos cómo el impulso supera el primero de estos dos problemas. Las líneas de contorno representan una función de pérdida cuadrática con una matriz hessiana mal condicionada. El camino rojo que atraviesa los contornos indica el camino seguido por la regla de aprendizaje de impulso, ya que minimiza esta función. En cada paso del camino, dibujamos una flecha que indica el paso que daría el descenso de gradiente en ese punto. Podemos ver que un objetivo cuadrático mal acondicionado parece un valle o cañón largo y angosto con lados empinados. El impulso atraviesa correctamente el cañón a lo largo, mientras que los pasos de gradiente pierden tiempo moviéndose hacia adelante y hacia atrás a través del eje angosto del cañón. Comparar también figura 4.6, que muestra el comportamiento del descenso de gradiente sin cantidad de movimiento.

Anteriormente, el tamaño del paso era simplemente la norma del gradiente multiplicado por la tasa de aprendizaje. Ahora, el tamaño del paso depende de qué tan grande y qué tan alineado esté un *secuencia* de gradientes son. El tamaño del paso es mayor cuando muchos gradientes sucesivos apuntan exactamente en la misma dirección. Si el algoritmo de momento siempre observa gradiente *gramo*, entonces acelerará en la dirección de *gramo*, hasta alcanzar una velocidad terminal donde el tamaño de cada paso es

$$\frac{-\|g\|}{1-\alpha}. \quad (8.17)$$

Por lo tanto, es útil pensar en el hiperparámetro de cantidad de movimiento en términos de $\frac{1}{1-\alpha}$. Para ejemplo, $\alpha=0.9$ corresponde a multiplicar la velocidad máxima por 10 en relación con el algoritmo de descenso de gradiente.

Valores comunes de α utilizados en la práctica incluyen 0.5, 0.9 y 0.99 como la tasa de aprendizaje, α también puede adaptarse con el tiempo. Por lo general, comienza con un valor pequeño y luego se eleva. Es menos importante adaptarse a α con el tiempo que encogerse α con el tiempo.

Algoritmo 8.2 Descenso de gradiente estocástico (SGD) con impulso

Requerir: Tasa de aprendizaje α , parámetro de impulso α .

Requerir: Parámetro inicial θ , velocidad inicial v .

mientras no se cumple el criterio de parada **hacer**

Muestra un mini lote de 1_{metro} ejemplos del conjunto de entrenamiento $\{X(1), \dots, X(\text{metro})\}$ con objetivos correspondientes $y(1)$.

Calcular estimación de gradiente: $g = -\nabla_{\theta} L(F(X(1); \theta), y(1))$

Actualización de la velocidad de cálculo: $v \leftarrow \alpha v - g$

Aplicar actualización: $\theta \leftarrow \theta + v$ **terminar mientras**

Podemos ver el algoritmo de impulso como si simulara una partícula sujeta a la dinámica newtoniana de tiempo continuo. La analogía física puede ayudar a generar intuición sobre cómo se comportan los algoritmos de impulso y descenso de gradiente.

La posición de la partícula en cualquier punto en el tiempo está dada por $\theta(t)$. La partícula experimenta una fuerza neta $F(t)$. Esta fuerza hace que la partícula se acelere:

$$F(t) = \frac{\partial^2}{\partial t^2} \theta(t). \quad (8.18)$$

En lugar de ver esto como una ecuación diferencial de segundo orden de la posición, podemos introducir la variable $v(t)$ que representa la velocidad de la partícula en el tiempo t y reescribir la dinámica newtoniana como una ecuación diferencial de primer orden:

$$v(t) = \frac{\partial}{\partial t} \theta(t), \quad (8.19)$$

$$F(t) = -\frac{\partial}{\partial t} V(t). \quad (8.20)$$

El algoritmo de impulso consiste entonces en resolver las ecuaciones diferenciales mediante simulación numérica. Un método numérico simple para resolver ecuaciones diferenciales es el método de Euler, que consiste simplemente en simular la dinámica definida por la ecuación dando pasos pequeños y finitos en la dirección de cada gradiente.

Esto explica la forma básica de la actualización del impulso, pero ¿qué son específicamente las fuerzas? Una fuerza es proporcional al gradiente negativo de la función de costo:

$-\nabla V(\theta)$. Esta fuerza empuja la partícula cuesta abajo a lo largo de la superficie de la función de costo. El algoritmo de descenso de gradiente simplemente daría un solo paso en función de cada gradiente, pero el escenario newtoniano utilizado por el algoritmo de impulso utiliza esta fuerza para alterar la velocidad de la partícula. Podemos pensar en la partícula como si fuera un disco de hockey deslizándose por una superficie helada. Siempre que desciende por una parte empinada de la superficie, toma velocidad y continúa deslizándose en esa dirección hasta que comienza a subir nuevamente.

Se necesita otra fuerza. Si la única fuerza es el gradiente de la función de costo, es posible que la partícula nunca se detenga. Imagine un disco de hockey deslizándose por un lado de un valle y hacia arriba por el otro lado, oscilando de un lado a otro para siempre, asumiendo que el hielo no tiene fricción. Para resolver este problema, agregamos otra fuerza, proporcional a $-V(t)$. En la terminología de la física, esta fuerza corresponde al arrastre viscoso, como si la partícula tuviera que empujar a través de un medio resistente como el jarabe. Esto hace que la partícula pierda energía gradualmente con el tiempo y eventualmente converja a un mínimo local.

¿Por qué usamos $-V(t)$ y arrastre viscoso en particular? Parte de la razón para usar $-V(t)$ es conveniencia matemática: es fácil trabajar con una potencia entera de la velocidad. Sin embargo, otros sistemas físicos tienen otros tipos de arrastre basados en otras potencias enteras de la velocidad. Por ejemplo, una partícula que viaja por el aire experimenta un arrastre turbulento, con una fuerza proporcional al cuadrado de la velocidad, mientras que una partícula que se mueve por el suelo experimenta fricción seca, con una fuerza de magnitud constante. Podemos rechazar cada una de estas opciones. La resistencia turbulenta, proporcional al cuadrado de la velocidad, se vuelve muy débil cuando la velocidad es pequeña. No es lo suficientemente potente como para obligar a la partícula a detenerse. Una partícula con una velocidad inicial distinta de cero que experimenta solo la fuerza de arrastre turbulento se alejará de su posición inicial para siempre, y la distancia desde el punto de partida crecerá como $O(\text{registro } t)$. Por lo tanto, debemos usar una potencia más baja de la velocidad. Si usamos una potencia de cero, que representa la fricción seca, entonces la fuerza es demasiado fuerte. Cuando la fuerza debida al gradiente de la función de costo es pequeña pero distinta de cero, la fuerza constante debida a la fricción puede hacer que la partícula se detenga antes de alcanzar un mínimo local. El arrastre viscoso evita estos dos problemas: es lo suficientemente débil

que el gradiente puede continuar causando movimiento hasta que se alcance un mínimo, pero lo suficientemente fuerte como para evitar el movimiento si el gradiente no justifica el movimiento.

8.3.3 Momento de Nésterov

Sutskever et al.(2013) introdujo una variante del algoritmo de impulso que se inspiró en el método de gradiente acelerado de Nesterov (Nésterov,1983,2004). Las reglas de actualización en este caso vienen dadas por:

$$v \leftarrow av - \nabla_{\theta} \text{ metro} - \frac{1}{L} \sum_{j=1}^{\text{metro}} L(F(X^{(j)}; \theta + av), y^{(j)}), \quad (8.21)$$

$$\theta \leftarrow \theta + v, \quad (8.22)$$

donde los parametros a y-juegan un papel similar al del método estándar del impulso. La diferencia entre el momento de Nesterov y el momento estándar es donde se evalúa el gradiente. Con Nesterovmomentum, el gradiente se evalúa después de aplicar la velocidad actual. Por lo tanto, uno puede interpretar el impulso de Nesterov como un intento de agregar unfactor de correcciónal método estándar del impulso. El algoritmo completo de impulso de Nesterov se presenta en algoritmo8.3.

En el caso del gradiente por lotes convexo, el impulso de Nesterov trae la tasa de convergencia del exceso de error de $O(1/k)$ (después k pasos para $O(1/k_2)$)como se muestra Nésterov(1983). Desafortunadamente, en el caso del gradiente estocástico, el impulso de Nesterov no mejora la tasa de convergencia.

Algoritmo 8.3 Descenso de gradiente estocástico (SGD) con impulso de Nesterov

Requerir:Tasa de aprendizaje-, parámetro de impulso a .

Requerir:Parámetro inicial θ , velocidad inicial v .

mientrasno se cumple el criterio de parada**hacer**

Muestra un mini lote de metro ejemplos del conjunto de entrenamiento $\{X^{(1)}, \dots, X^{(\text{metro})}\}$ con las etiquetas correspondientes $y^{(i)}$. Aplicar actualización provisional: $\theta \leftarrow \theta + av$ Calcule el gradiente (en el punto intermedio): $gramo \leftarrow 1$

$$\frac{1}{\text{metro}} \nabla_{\theta} \sum_{i=1}^{\text{metro}} L(F(X^{(i)}; \theta), y^{(i)})$$

Actualización de la velocidad de cálculo: $v \leftarrow av - g$

Aplicar actualización: $\theta \leftarrow \theta + v$ **terminar mientras**

8.4 Estrategias de inicialización de parámetros

Algunos algoritmos de optimización no son iterativos por naturaleza y simplemente resuelven un punto de solución. Otros algoritmos de optimización son iterativos por naturaleza pero, cuando se aplican a la clase correcta de problemas de optimización, convergen en soluciones aceptables en una cantidad de tiempo aceptable, independientemente de la inicialización. Los algoritmos de entrenamiento de aprendizaje profundo generalmente no tienen ninguno de estos lujos. Los algoritmos de entrenamiento para modelos de aprendizaje profundo suelen ser de naturaleza iterativa y, por lo tanto, requieren que el usuario especifique algún punto inicial desde el cual comenzar las iteraciones. Además, entrenar modelos profundos es una tarea lo suficientemente difícil como para que la mayoría de los algoritmos se vean fuertemente afectados por la elección de la inicialización. El punto inicial puede determinar si el algoritmo converge en absoluto, siendo algunos puntos iniciales tan inestables que el algoritmo encuentra dificultades numéricas y falla por completo. Cuando el aprendizaje converge, el punto inicial puede determinar qué tan rápido converge el aprendizaje y si converge a un punto con alto o bajo costo. Además, los puntos de costo comparable pueden tener un error de generalización muy variable, y el punto inicial también puede afectar la generalización.

Las estrategias de inicialización modernas son simples y heurísticas. Diseñar estrategias de inicialización mejoradas es una tarea difícil porque la optimización de redes neuronales aún no se comprende bien. La mayoría de las estrategias de inicialización se basan en lograr algunas buenas propiedades cuando se inicializa la red. Sin embargo, no tenemos una buena comprensión de cuáles de estas propiedades se conservan bajo qué circunstancias después de que comienza el aprendizaje. Otra dificultad es que algunos puntos iniciales pueden ser beneficiosos desde el punto de vista de la optimización pero perjudiciales desde el punto de vista de la generalización. Nuestra comprensión de cómo el punto inicial afecta la generalización es especialmente primitiva y ofrece poca o ninguna guía sobre cómo seleccionar el punto inicial.

Quizás la única propiedad que se conoce con total certeza es que los parámetros iniciales necesitan “romper la simetría” entre diferentes unidades. Si dos unidades ocultas con la misma función de activación están conectadas a las mismas entradas, estas unidades deben tener diferentes parámetros iniciales. Si tienen los mismos parámetros iniciales, entonces un algoritmo de aprendizaje determinista aplicado a un costo y modelo deterministas actualizará constantemente ambas unidades de la misma manera. Incluso si el modelo o el algoritmo de entrenamiento es capaz de usar la estocasticidad para calcular diferentes actualizaciones para diferentes unidades (por ejemplo, si uno entrena con abandono), generalmente es mejor inicializar cada unidad para calcular una función diferente de todas las demás unidades. Esto puede ayudar a garantizar que no se pierdan patrones de entrada en el espacio nulo de la propagación hacia adelante y que no se pierdan patrones de gradiente en el espacio nulo de la propagación hacia atrás. El objetivo de que cada unidad calcule una función diferente

motiva la inicialización aleatoria de los parámetros. Podríamos buscar explícitamente un gran conjunto de funciones base que sean todas diferentes entre sí, pero esto a menudo implica un costo computacional notable. Por ejemplo, si tenemos como máximo tantas salidas como entradas, podríamos usar la ortogonalización de Gram-Schmidt en una matriz de peso inicial y estar seguros de que cada unidad calcula una función muy diferente de cada otra unidad. La inicialización aleatoria de una distribución de alta entropía sobre un espacio de alta dimensión es computacionalmente más barata y es poco probable que asigne unidades para calcular la misma función entre sí.

Por lo general, establecemos los sesgos para cada unidad en constantes elegidas heurísticamente e inicializamos solo los pesos al azar. Los parámetros adicionales, por ejemplo, los parámetros que codifican la varianza condicional de una predicción, generalmente se establecen en constantes elegidas heurísticamente al igual que los sesgos.

Casi siempre inicializamos todos los pesos en el modelo a valores extraídos aleatoriamente de una distribución gaussiana o uniforme. La elección de la distribución gaussiana o uniforme no parece importar mucho, pero no ha sido estudiada exhaustivamente. Sin embargo, la escala de la distribución inicial tiene un gran efecto tanto en el resultado del procedimiento de optimización como en la capacidad de generalización de la red.

Los pesos iniciales más grandes producirán un efecto de ruptura de simetría más fuerte, lo que ayudará a evitar unidades redundantes. También ayudan a evitar la pérdida de señal durante la propagación hacia adelante o hacia atrás a través del componente lineal de cada capa: los valores más grandes en la matriz dan como resultado mayores salidas de multiplicación de matriz. Sin embargo, los pesos iniciales que son demasiado grandes pueden dar como resultado valores explosivos durante la propagación hacia adelante o hacia atrás. En redes recurrentes, los pesos grandes también pueden resultar en **encaos** (sensibilidad tan extrema a las pequeñas perturbaciones de la entrada que el comportamiento del procedimiento de propagación directa determinista parece aleatorio). Hasta cierto punto, el problema del gradiente explosivo se puede mitigar mediante el recorte de gradientes (limitando los valores de los gradientes antes de realizar un paso de descenso de gradiente). Los pesos grandes también pueden dar como resultado valores extremos que provocan la saturación de la función de activación, provocando la pérdida completa del gradiente a través de las unidades saturadas. Estos factores competitivos determinan la escala inicial ideal de los pesos.

Las perspectivas de regularización y optimización pueden brindar perspectivas muy diferentes sobre cómo debemos inicializar una red. La perspectiva de optimización sugiere que los pesos deberían ser lo suficientemente grandes para propagar la información con éxito, pero algunas preocupaciones de regularización alientan a hacerlos más pequeños. El uso de un algoritmo de optimización, como el descenso de gradiente estocástico, que realiza pequeños cambios incrementales en los pesos y tiende a detenerse en áreas que están más cerca de los parámetros iniciales (ya sea porque se atasca en una región de bajo gradiente o

debido a la activación de algún criterio de parada anticipada basado en el sobreajuste) expresa un previo de que los parámetros finales deben estar cerca de los parámetros iniciales. Recuperar de la sección 7.8 ese descenso de gradiente con parada temprana es equivalente a la caída de peso para algunos modelos. En el caso general, el descenso de gradiente con parada temprana no es lo mismo que el decaimiento del peso, pero proporciona una analogía imprecisa para pensar en el efecto de la inicialización. Podemos pensar en inicializar los parámetros θ como algo similar a imponer una prioridad gaussiana $p(\theta)$ con media θ_0 . Desde este punto de vista, tiene sentido elegir θ estar cerca de 0. Este prior dice que es más probable que las unidades no interactúen entre sí a que interactúen. Las unidades interactúan solo si el término de probabilidad de la función objetivo expresa una fuerte preferencia por que interactúen. Por otro lado, si inicializamos θ a valores grandes, entonces nuestro anterior especifica qué unidades deben interactuar entre sí y cómo deben interactuar.

Algunas heurísticas están disponibles para elegir la escala inicial de los pesos. Una heurística es inicializar los pesos de una capa completamente conectada con *metro entradas* y *norte salidas* muestreando cada peso de $t_u \sim \mathcal{N}(0, \frac{1}{\text{metro+norte}})$, mientras Glorot y Bengio (2010) sugieren usar el **inicialización normalizada**

$$W_{yo,j} \sim t_u \sim \frac{6}{\sqrt{\text{metro+norte}}}, \quad (8.23)$$

Esta última heurística está diseñada para llegar a un compromiso entre el objetivo de inicializar todas las capas para que tengan la misma varianza de activación y el objetivo de inicializar todas las capas para que tengan la misma varianza de gradiente. La fórmula se obtiene asumiendo que la red consta solo de una cadena de multiplicaciones de matrices, sin no linealidades. Las redes neuronales reales obviamente violan esta suposición, pero muchas estrategias diseñadas para el modelo lineal funcionan razonablemente bien en sus contrapartes no lineales.

Sajonja et al. (2013) recomiendan inicializar a matrices ortogonales aleatorias, con una escala cuidadosamente elegida **ganar factor gram** que da cuenta de la no linealidad aplicada en cada capa. Derivan valores específicos del factor de escala para diferentes tipos de funciones de activación no lineales. Este esquema de inicialización también está motivado por un modelo de una red profunda como una secuencia de multiplicaciones de matrices sin no linealidades. Bajo tal modelo, este esquema de inicialización garantiza que el número total de iteraciones de entrenamiento requeridas para alcanzar la convergencia sea independiente de la profundidad.

Aumentar el factor de escala **gramo** empuja a la red hacia el régimen donde las activaciones aumentan en norma a medida que se propagan hacia adelante a través de la red y los gradientes aumentan en norma a medida que se propagan hacia atrás. Sussillo (2014) mostró que establecer el factor de ganancia correctamente es suficiente para entrenar redes tan profundas como

1.000 capas, sin necesidad de utilizar inicializaciones ortogonales. Una idea clave de este enfoque es que en las redes de avance, las activaciones y los gradientes pueden crecer o reducirse en cada paso de propagación hacia adelante o hacia atrás, siguiendo un comportamiento de caminata aleatoria. Esto se debe a que las redes feedforward usan una matriz de peso diferente en cada capa. Si esta caminata aleatoria se ajusta para preservar las normas, entonces las redes feedforward pueden evitar en su mayoría el problema de gradientes que se desvanecen y explotan que surge cuando se usa la misma matriz de peso en cada paso, descrito en la sección 8.2.5.

Desafortunadamente, estos criterios óptimos para los pesos iniciales a menudo no conducen a un rendimiento óptimo. Esto puede ser por tres razones diferentes. En primer lugar, es posible que estemos utilizando los criterios incorrectos; en realidad, puede que no sea beneficioso preservar la norma de una señal en toda la red. En segundo lugar, las propiedades impuestas en la inicialización pueden no persistir después de que el aprendizaje haya comenzado. En tercer lugar, los criterios pueden tener éxito en mejorar la velocidad de optimización pero, sin darse cuenta, aumentan el error de generalización. En la práctica, generalmente necesitamos tratar la escala de los pesos como un hiperparámetro cuyo valor óptimo se encuentra en algún lugar aproximadamente cerca pero no exactamente igual a las predicciones teóricas.

Un inconveniente de las reglas de escala que establecen que todos los pesos iniciales tienen la misma desviación estándar, como $\frac{1}{\sqrt{d}}$, es que cada peso individual se vuelve extremadamente pequeño cuando las capas se vuelven grandes. [martas\(2010\)](#) introdujo un esquema de inicialización alternativo llamado **inicialización escasa** en el que cada unidad se inicializa para tener exactamente k pesos distintos de cero. La idea es mantener la cantidad total de entrada a la unidad independiente del número de entradas m sin hacer que la magnitud de los elementos de peso individuales se reduzca con m . La inicialización dispersa ayuda a lograr una mayor diversidad entre las unidades en el momento de la inicialización. Sin embargo, también impone una prioridad muy fuerte sobre los pesos que se eligen para tener valores gaussianos grandes. Debido a que el descenso de gradiente tarda mucho tiempo en reducir los valores grandes "incorrectos", este esquema de inicialización puede causar problemas para unidades como las unidades maxout que tienen varios filtros que deben coordinarse cuidadosamente entre sí.

Cuando los recursos computacionales lo permiten, generalmente es una buena idea tratar la escala inicial de los pesos para cada capa como un hiperparámetro y elegir estas escalas usando un algoritmo de búsqueda de hiperparámetros descrito en la sección 11.4.2, como la búsqueda aleatoria. La elección de utilizar una inicialización densa o dispersa también se puede convertir en un hiperparámetro. Alternativamente, uno puede buscar manualmente las mejores escalas iniciales. Una buena regla general para elegir las escalas iniciales es observar el rango o la desviación estándar de activaciones o gradientes en un solo minilote de datos. Si los pesos son demasiado pequeños, el rango de activaciones en el minilote se reducirá a medida que las activaciones se propaguen a través de la red. Al identificar repetidamente la primera capa con activaciones inaceptablemente pequeñas y

aumentando sus pesos, es posible obtener eventualmente una red con activaciones iniciales razonables en todo momento. Si el aprendizaje sigue siendo demasiado lento en este punto, puede ser útil observar el rango o la desviación estándar de los gradientes, así como las activaciones. En principio, este procedimiento se puede automatizar y generalmente es menos costoso desde el punto de vista computacional que la optimización de hiperparámetros basada en el error del conjunto de validación porque se basa en la retroalimentación del comportamiento del modelo inicial en un solo lote de datos, en lugar de la retroalimentación de un modelo entrenado en el conjunto de validación. Si bien se usó durante mucho tiempo de manera heurística, este protocolo se ha especificado recientemente de manera más formal y estudiado por [Mishkin y Matas\(2015\)](#).

Hasta ahora nos hemos centrado en la inicialización de los pesos. Afortunadamente, la inicialización de otros parámetros suele ser más sencilla.

El enfoque para establecer los sesgos debe coordinarse con el enfoque para establecer los pesos. Establecer los sesgos en cero es compatible con la mayoría de los esquemas de inicialización de peso. Hay algunas situaciones en las que podemos establecer algunos sesgos en valores distintos de cero:

- Si un sesgo es para una unidad de salida, a menudo es beneficioso inicializar el sesgo para obtener las estadísticas marginales correctas de la salida. Para hacer esto, asumimos que los pesos iniciales son lo suficientemente pequeños como para que la salida de la unidad esté determinada solo por el sesgo. Esto justifica establecer el sesgo a la inversa de la función de activación aplicada a las estadísticas marginales de la salida en el conjunto de entrenamiento. Por ejemplo, si la salida es una distribución entre clases y esta distribución es una distribución altamente sesgada con la probabilidad marginal de clase/dado por elemento C de algún vector C , entonces podemos establecer el vector de sesgo b resolviendo la ecuación $\text{softmax}(b) = C$. Esto se aplica no solo a los clasificadores sino también a los modelos que encontraremos en la Parte [tercero](#), como autocodificadores y máquinas Boltzmann. Estos modelos tienen capas cuya salida debe parecerse a los datos de entrada. X , y puede ser muy útil inicializar los sesgos de dichas capas para que coincidan con la distribución marginal sobre X .
- A veces podemos querer elegir el sesgo para evitar causar demasiada saturación en la inicialización. Por ejemplo, podemos establecer el sesgo de una unidad oculta ReLU en 0,1 en lugar de 0 para evitar saturar ReLU en la inicialización. Sin embargo, este enfoque no es compatible con los esquemas de inicialización de peso que no esperan una fuerte entrada de los sesgos. Por ejemplo, no se recomienda su uso con la inicialización de recorrido aleatorio ([sussillo,2014](#)).
- A veces, una unidad controla si otras unidades pueden participar en una función. En tales situaciones, tenemos una unidad con salida t y otra unidad $h \in [0,1]$, y se multiplican para producir una salida oh . Nosotros

puedo ver h como una puerta que determina si $oh \approx tu$ o $oh \approx 0$. En estas situaciones, queremos establecer el sesgo para h de modo que $h \approx 1$ la mayor parte del tiempo en la inicialización. De lo contrario t no tiene la oportunidad de aprender. Por ejemplo, [jozefowicz et al. \(2015\)](#) abogan por establecer el sesgo 1 para la puerta de olvido del modelo LSTM, descrita en la sección [10.10](#).

Otro tipo común de parámetro es un parámetro de varianza o de precisión. Por ejemplo, podemos realizar una regresión lineal con una estimación de varianza condicional usando el modelo

$$p_{\text{ag}}(y | X) = \text{norte}(y | w^T X + b, 1/\beta) \quad (8.24)$$

dónde β es un parámetro de precisión. Por lo general, podemos inicializar los parámetros de varianza o precisión a 1 de forma segura. Otro enfoque es suponer que las ponderaciones iniciales están lo suficientemente cerca de cero para que se puedan establecer los sesgos ignorando el efecto de las ponderaciones, luego establecer los sesgos para producir la media marginal correcta de la salida y establecer los parámetros de varianza a la varianza marginal de la salida en el conjunto de entrenamiento.

Además de estos métodos constantes o aleatorios simples para inicializar los parámetros del modelo, es posible inicializar los parámetros del modelo mediante el aprendizaje automático. Una estrategia común discutida en parte [tercero](#) de este libro es inicializar un modelo supervisado con los parámetros aprendidos por un modelo no supervisado entrenado con las mismas entradas. También se puede realizar un entrenamiento supervisado en una tarea relacionada. Incluso realizar un entrenamiento supervisado en una tarea no relacionada a veces puede generar una inicialización que ofrece una convergencia más rápida que una inicialización aleatoria. Algunas de estas estrategias de inicialización pueden generar una convergencia más rápida y una mejor generalización porque codifican información sobre la distribución en los parámetros iniciales del modelo. Aparentemente, otros funcionan bien principalmente porque establecen los parámetros para tener la escala correcta o establecen diferentes unidades para calcular funciones diferentes entre sí.

8.5 Algoritmos con tasas de aprendizaje adaptables

Hace tiempo que los investigadores de redes neuronales se dieron cuenta de que la tasa de aprendizaje era uno de los hiperparámetros más difíciles de establecer porque tiene un impacto significativo en el rendimiento del modelo. Como hemos discutido en las secciones [4.3](#) y [8.2](#), el costo suele ser muy sensible a algunas direcciones en el espacio de parámetros e insensible a otras. El algoritmo de impulso puede mitigar un poco estos problemas, pero lo hace a expensas de introducir otro hiperparámetro. Ante esto, es natural preguntarse si hay otra forma. Si creemos que las direcciones de la sensibilidad están algo alineadas con el eje, puede tener sentido usar un aprendizaje separado.

tasa para cada parámetro, y adaptar automáticamente estas tasas de aprendizaje a lo largo del curso de aprendizaje.

El **delta-barra-delta** algoritmo ([Jacobs, 1988](#)) es un enfoque heurístico temprano para adaptar las tasas de aprendizaje individuales para los parámetros del modelo durante el entrenamiento. El enfoque se basa en una idea simple: si la derivada parcial de la pérdida, con respecto a un parámetro de modelo dado, permanece del mismo signo, entonces la tasa de aprendizaje debería aumentar. Si la derivada parcial con respecto a ese parámetro cambia de signo, entonces la tasa de aprendizaje debería disminuir. Por supuesto, este tipo de regla solo se puede aplicar a la optimización de lotes completos.

Más recientemente, se han introducido una serie de métodos incrementales (o basados en mini lotes) que adaptan las tasas de aprendizaje de los parámetros del modelo. Esta sección revisará brevemente algunos de estos algoritmos.

8.5.1 AdaGrado

El **adagrad** algoritmo, mostrado en algoritmo [8.4](#), adapta individualmente las tasas de aprendizaje de todos los parámetros del modelo escalándolos inversamente proporcionales a la raíz cuadrada de la suma de todos sus valores cuadrados históricos ([Duchi et al., 2011](#)). Los parámetros con la mayor derivada parcial de la pérdida tienen una disminución correspondientemente rápida en su tasa de aprendizaje, mientras que los parámetros con pequeñas derivadas parciales tienen una disminución relativamente pequeña en su tasa de aprendizaje. El efecto neto es un mayor progreso en las direcciones de pendiente más suave del espacio de parámetros.

En el contexto de la optimización convexa, el algoritmo AdaGrad disfruta de algunas propiedades teóricas deseables. Sin embargo, se ha encontrado empíricamente que, para entrenar modelos de redes neuronales profundas, la acumulación de gradientes cuadrados *desde el comienzo del entrenamiento* puede resultar en una disminución prematura y excesiva en la tasa de aprendizaje efectivo. AdaGrad funciona bien para algunos modelos de aprendizaje profundo, pero no para todos.

8.5.2 RMS Prop

El **RMSProp** algoritmo ([Hinton, 2012](#)) modifica AdaGrad para que funcione mejor en la configuración no convexa al cambiar la acumulación de gradiente en un promedio móvil exponencialmente ponderado. AdaGrad está diseñado para converger rápidamente cuando se aplica a una función convexa. Cuando se aplica a una función no convexa para entrenar una red neuronal, la trayectoria de aprendizaje puede pasar por muchas estructuras diferentes y finalmente llegar a una región que es un cuenco localmente convexo. AdaGrad reduce la tasa de aprendizaje de acuerdo con el historial completo del gradiente al cuadrado y puede

Algoritmo 8.4 El algoritmo AdaGrad

Requerir: Tasa de aprendizaje global-

Requerir: Parámetro inicial θ

Requerir: Pequeña constante d , tal vez 10^{-7} , para la estabilidad numérica

Iniciar variable de acumulación de gradiente $r=0$

mientras no se cumple el criterio de parada **hacer**

Muestra un mini lote de $metro$ ejemplos del conjunto de entrenamiento $\{X(1), \dots, X(metro)\}$ con los objetivos correspondientes $y(i)$. Gradiente de cálculo: $gramo \leftarrow$

$$\frac{1}{metro} \sum_{i=1}^{metro} iL(F(X(i); \theta), y(i))$$

Gradiente cuadrado acumulado: $r \leftarrow r + g \cdot g$

Actualizar cálculo: $\Delta\theta \leftarrow -\sqrt{\frac{r}{d}} \cdot gramo$. (División y raíz cuadrada aplicada elemento sabio)

Aplicar actualización: $\theta \leftarrow \theta + \Delta\theta$

terminar **mientras**

han hecho que la tasa de aprendizaje sea demasiado pequeña antes de llegar a una estructura tan convexa. RMSProp utiliza un promedio exponencialmente decreciente para descartar la historia del pasado extremo para que pueda converger rápidamente después de encontrar un cuenco convexo, como si fuera una instancia del algoritmo AdaGrad inicializado dentro de ese cuenco.

RMSProp se muestra en su forma estándar en el algoritmo 8.5 y combinado con el impulso de Nesterov en el algoritmo 8.6. En comparación con AdaGrad, el uso de la media móvil introduce un nuevo hiperparámetro, ρ , que controla la escala de longitud de la media móvil.

Empíricamente, RMSProp ha demostrado ser un algoritmo de optimización efectivo y práctico para redes neuronales profundas. Actualmente es uno de los métodos de optimización de referencia que emplean de forma rutinaria los profesionales del aprendizaje profundo.

8.5.3 Adán

Adán (Kingma y Ba, 2014) es otro algoritmo de optimización de la tasa de aprendizaje adaptativo y se presenta en el algoritmo 8.7. El nombre “Adán” deriva de la frase “momentos adaptativos”. En el contexto de los algoritmos anteriores, quizás se vea mejor como una variante de la combinación de RMSProp e impulso con algunas distinciones importantes. Primero, en Adam, el momento se incorpora directamente como una estimación del momento de primer orden (con ponderación exponencial) del gradiente. La forma más sencilla de agregar impulso a RMSProp es aplicar impulso a los gradientes reescalados. El uso del impulso en combinación con el reescalamiento no tiene una motivación teórica clara. En segundo lugar, Adán incluye

Algoritmo 8.5 El algoritmo RMSProp

Requerir:Tasa de aprendizaje global-, tasa de descomposición ρ . **Requerir:**Parámetro inicial θ

Requerir:Pequeña constante d , generalmente 10^{-6} , utilizado para estabilizar la división por pequeños números.

Inicializar variables de acumulación $r=0$ **mientras**

no se cumple el criterio de parada**hacer**

Muestra un mini lote de $metro$ ejemplos del conjunto de entrenamiento $\{X(1), \dots, X(metro)\}$ con los objetivos correspondientes $y(i)$. Gradiente de cálculo: $gramo \leftarrow$

$$\frac{1}{metro} \nabla_{\theta} L(F(X(i); \theta), y(i))$$

Gradiente cuadrado acumulado: $r \leftarrow \rho r + (1 - \rho)g^2$ Calcular

actualización de parámetros: $\Delta\theta = -\sqrt{\frac{1}{d+r}} - gramo$. ($\sqrt{\frac{1}{d+r}}$ elemento aplicado)

Aplicar actualización: $\theta \leftarrow \theta + \Delta\theta$

terminar mientras

correcciones de sesgo a las estimaciones de los momentos de primer orden (el término de momento) y los momentos de segundo orden (no centrados) para tener en cuenta su inicialización en el origen (ver algoritmo 8.7). RMSProp también incorpora una estimación del momento de segundo orden (no centrado), sin embargo, carece del factor de corrección. Por lo tanto, a diferencia de Adam, la estimación del momento de segundo orden de RMSProp puede tener un alto sesgo al principio del entrenamiento. En general, se considera que Adam es bastante robusto en la elección de hiperparámetros, aunque a veces es necesario cambiar la tasa de aprendizaje del valor predeterminado sugerido.

8.5.4 Elegir el algoritmo de optimización adecuado

En esta sección, discutimos una serie de algoritmos relacionados que buscan abordar el desafío de optimizar modelos profundos adaptando la tasa de aprendizaje para cada parámetro del modelo. En este punto, una pregunta natural es: ¿qué algoritmo debería elegir?

Desafortunadamente, actualmente no hay consenso sobre este punto. [schaulet al.\(2014\)](#) presentó una valiosa comparación de una gran cantidad de algoritmos de optimización en una amplia gama de tareas de aprendizaje. Si bien los resultados sugieren que la familia de algoritmos con tasas de aprendizaje adaptables (representadas por RMSProp y AdaDelta) se desempeñó de manera bastante sólida, no ha surgido ningún algoritmo mejor.

Actualmente, los algoritmos de optimización más populares en uso activo incluyen SGD, SGD con impulso, RMSProp, RMSProp con impulso, AdaDelta y Adam. La elección de qué algoritmo usar, en este punto, parece depender

Algoritmo 8.6 Algoritmo RMSProp con impulso de Nesterov

Requerir: Tasa de aprendizaje global η , tasa de descomposición ρ , coeficiente de impulso α . **Requerir:** Parámetro inicial θ , velocidad inicial v .

Iniciar variable de acumulación $r=0$ **mientras**

no se cumple el criterio de parada **hacer**

Muestra un mini lote de m_{metro} ejemplos del conjunto de entrenamiento $\{(X(1), \dots, X(m_{\text{metro}}))\}$ con los objetivos correspondientes $\{y(i)\}$. Calcule la actualización provisional: $\theta' \leftarrow \theta + \eta v$ Gradiente de cálculo: $g \leftarrow \nabla L(\theta, y)$ Gradiante acumulado: $r \leftarrow \rho r + (1 - \rho)g^2$ Actualización de la velocidad de cálculo: $v \leftarrow \eta v - \sqrt{\rho} g$. Aplicar actualización: $\theta \leftarrow \theta + \frac{\eta}{\sqrt{r}} v$ **mientras**

$$\overline{r} = \frac{1}{m_{\text{metro}}} \sum_{i=1}^{m_{\text{metro}}} v_i^2 \quad (\text{elemento aplicado})$$

en gran medida en la familiaridad del usuario con el algoritmo (para facilitar el ajuste de hiperparámetros).

8.6 Métodos aproximados de segundo orden

En esta sección discutimos la aplicación de métodos de segundo orden al entrenamiento de redes profundas. Ver [lecunet et al. \(1998a\)](#) para un tratamiento anterior de este tema. Para simplificar la exposición, la única función objetivo que examinamos es el riesgo empírico:

$$J(\theta) = \min_{x,y \sim p_{\text{datos}}(x,y)} L(f((X;\theta), y)) = \frac{1}{m_{\text{metro}}} \sum_{i=1}^{m_{\text{metro}}} L(f(X(i);\theta), y(i)). \quad (8.25)$$

Sin embargo, los métodos que discutimos aquí se extienden fácilmente a funciones objetivo más generales que, por ejemplo, incluyen términos de regularización de parámetros como los discutidos en el capítulo [7](#).

8.6.1 Método de Newton

En la sección [4.3](#), introdujimos métodos de gradiente de segundo orden. A diferencia de los métodos de primer orden, los métodos de segundo orden utilizan derivadas segundas para mejorar la optimización. El método de segundo orden más utilizado es el método de Newton. Ahora describimos el método de Newton con más detalle, con énfasis en su aplicación al entrenamiento de redes neuronales.

Algoritmo 8.7 El algoritmo de Adán

Requerir: Número de pie-(Predeterminado sugerido:0.001)**Requerir:** Tasas de decaimiento exponencial para estimaciones de momentos, ρ_1 y ρ_2 en [0,1].

(Valores predeterminados sugeridos:0.9y0.999respectivamente)

Requerir: Pequeña constante d utilizado para la estabilización numérica. (Predeterminado sugerido:
10⁻⁸)**Requerir:** Parámetros iniciales θ Iniciar variables de primer y segundo momentos $s=0,r=0$ Iniciar paso de tiempo $t=0$ **mientras**no se cumple el criterio de
parada**hacer**Muestra un mini lote de *metro*ejemplos del conjunto de entrenamiento $\{X(1), \dots, X(metro)\}$ con
los objetivos correspondientes $y(1)$. Gradiente de cálculo: $gramo \leftarrow \nabla t \leftarrow t+1$

$$\bar{m} \theta \quad iL(F(X(i); \theta), y(i))$$

Actualice la estimación sesgada del primer momento: $s \leftarrow \rho_1 s + (1 - \rho_1) gramo$ Actualice la estimación sesgada del segundo momento: $r \leftarrow \rho_2 r + (1 - \rho_2) g - g$ Sesgocorrecto en primer momento: $s \leftarrow \frac{s}{1-\rho_1} r$ Sesgo correcto en segundo momento: $r \leftarrow \frac{r}{1-\rho_2}$ Actualizar cálculo: $\Delta\theta = -\sqrt{s} \frac{\nabla L}{\nabla d}$ (operaciones aplicadas por elementos)Aplicar actualización: $\theta \leftarrow \theta + \Delta\theta$ **terminar mientras**

El método de Newton es un esquema de optimización basado en el uso de una expansión de la serie de Taylor de segundo orden para aproximar $J(\theta)$ cerca de algún punto θ_0 , ignorando las derivadas de orden superior:

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0) \nabla_{\theta} J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H(\theta - \theta_0) \quad (8.26)$$

dónde H es la arpilla de J con respecto a θ evaluado en θ_0 . Si luego resolvemos para el punto crítico de esta función, obtenemos la regla de actualización de parámetros de Newton:

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0) \quad (8.27)$$

Así, para una función localmente cuadrática (con definida positiva H), cambiando la escala del gradiente por H^{-1} , el método de Newton salta directamente al mínimo. Si la función objetivo es convexa pero no cuadrática (hay términos de orden superior), esta actualización se puede iterar, lo que genera el algoritmo de entrenamiento asociado con el método de Newton, dado en algoritmo 8.8.

Algoritmo 8.8 Método de Newton con objetivo. $J(\theta)$

 $\frac{1}{\text{metro}} \sum_{i=1}^{\text{metro}} L(F(X(i); \theta), y(i)).$

Requerir: Parámetro inicial θ_0 **Requerir:** Conjunto de entrenamiento de metro ejemplos**mientras** criterio de parada no cumplido **hacer**Gradiente de cálculo: $g(\theta) \leftarrow \nabla_{\theta} \sum_{i=1}^{\text{metro}} L(F(X(i); \theta), y(i))$ Calcular arpilla: $H \leftarrow \nabla_{\theta} \nabla_{\theta} \sum_{i=1}^{\text{metro}} L(F(X(i); \theta), y(i))$ Calcule el inverso de Hessian: H^{-1} Actualizar cálculo: $\Delta\theta = -H^{-1} g(\theta)$ Aplicaractualización: $\theta = \theta + \Delta\theta$ **terminar** **mientras**

Para superficies que no son cuadráticas, siempre que la arpilla permanezca definida positiva, el método de Newton se puede aplicar iterativamente. Esto implica un procedimiento iterativo de dos pasos. Primero, actualice o calcule el hessiano inverso (es decir, actualizando la aproximación cuadrática). En segundo lugar, actualice los parámetros según la ecuación 8.27.

En la sección 8.2.3, discutimos cómo el método de Newton es apropiado solo cuando la arpilla es definida positiva. En el aprendizaje profundo, la superficie de la función objetivo suele ser no convexa con muchas características, como puntos de silla, que son problemáticas para el método de Newton. Si los valores propios de la arpilla no son todos positivos, por ejemplo, cerca de un punto de silla, entonces el método de Newton puede hacer que las actualizaciones se muevan en la dirección incorrecta. Esta situación se puede evitar regularizando la arpilla. Las estrategias comunes de regularización incluyen agregar una constante, a , a lo largo de la diagonal de la arpilla. La actualización regularizada pasa a ser

$$\theta = \theta_0 - [H(F(\theta_0)) + aI]^{-1} \nabla F(\theta_0). \quad (8.28)$$

Esta estrategia de regularización se utiliza en aproximaciones al método de Newton, como el algoritmo de Levenberg-Marquardt ([Levenberg, 1944](#); [Marquardt, 1963](#)), y funciona bastante bien siempre que los valores propios negativos de Hessian sigan siendo relativamente cercanos a cero. En los casos en que existan direcciones de curvatura más extremas, el valor de a tendría que ser lo suficientemente grande para compensar los valores propios negativos. Sin embargo, como a aumenta de tamaño, la arpilla se vuelve dominada por la diagonal y la dirección elegida por el método de Newton converge al gradiente estándar dividido por a . Cuando hay una fuerte curvatura negativa, a puede necesitar ser tan grande que el método de Newton daría pasos más pequeños que el descenso de gradiente con una tasa de aprendizaje elegida correctamente.

Más allá de los desafíos creados por ciertas características de la función objetivo,

como puntos de silla, la aplicación del método de Newton para entrenar grandes redes neuronales está limitada por la carga computacional significativa que impone. El número de elementos en el Hessian se eleva al cuadrado en el número de parámetros, por lo que con k parámetros (e incluso para redes neuronales muy pequeñas, el número de parámetros k puede ser de millones), el método de Newton requeriría la inversión de un $k \times k$ matriz, con una complejidad computacional de $\mathcal{O}(k^3)$. Además, dado que los parámetros cambiarán con cada actualización, se debe calcular el Hessian inverso *en cada iteración de entrenamiento*. Como consecuencia, solo las redes con un número muy pequeño de parámetros pueden entrenarse prácticamente mediante el método de Newton. En el resto de esta sección, discutiremos las alternativas que intentan obtener algunas de las ventajas del método de Newton mientras evitamos los obstáculos computacionales.

8.6.2 Gradientes conjugados

Los gradientes conjugados son un método para evitar de manera eficiente el cálculo del hessiano inverso al descender iterativamente **direcciones conjugadas**. La inspiración para este enfoque surge de un estudio cuidadoso de la debilidad del método de descenso más empinado (ver sección 4.3 para más detalles), donde las búsquedas de línea se aplican de forma iterativa en la dirección asociada con el degradado. Cifra 8.6 ilustra cómo el método de descenso más empinado, cuando se aplica en un cuenco cuadrático, progresó en un patrón de zig-zag de ida y vuelta bastante ineficaz. Esto sucede porque se garantiza que cada dirección de búsqueda de línea, cuando está dada por el gradiente, sea ortogonal a la dirección de búsqueda de línea anterior.

Sea la dirección de búsqueda anterior d_{t-1} . Como mínimo, donde termina la búsqueda de línea, la derivada direccional es cero en la dirección $d_{t-1}: \nabla_{\theta} J(\theta) \cdot d_{t-1} = 0$. Dado que el gradiente en este punto define la dirección de búsqueda actual, $d_t = \nabla_{\theta} J(\theta)$ no tendrá ninguna contribución en la dirección d_{t-1} . De este modo d_t es ortogonal a d_{t-1} . Esta relación entre d_{t-1} y d_t se ilustra en la figura 8.6 para múltiples iteraciones de descenso más empinado. Como se muestra en la figura, la elección de direcciones de descenso ortogonales no preserva el mínimo a lo largo de las direcciones de búsqueda anteriores. Esto da lugar al patrón de progreso en zig-zag, donde al descender al mínimo en la dirección del gradiente actual, debemos volver a minimizar el objetivo en la dirección del gradiente anterior. Por lo tanto, al seguir el gradiente al final de cada búsqueda de línea estamos, en cierto sentido, deshaciendo el progreso que ya hemos hecho en la dirección de la búsqueda de línea anterior. El método de gradientes conjugados busca abordar este problema.

En el método de gradientes conjugados, buscamos encontrar una dirección de búsqueda que sea **conjugada** a la dirección de búsqueda de línea anterior, es decir, no deshará el progreso realizado en esa dirección. En la iteración de entrenamiento t , la siguiente dirección de búsqueda d_t acepta

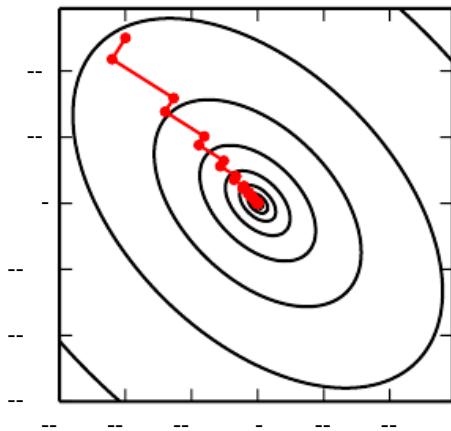


Figura 8.6: El método del descenso más pronunciado aplicado a una superficie de coste cuadrática. El método de descenso más empinado implica saltar al punto de menor costo a lo largo de la línea definida por la pendiente en el punto inicial de cada paso. Esto resuelve algunos de los problemas observados con el uso de una tasa de aprendizaje fija en la figura 4.6, pero incluso con el tamaño de paso óptimo, el algoritmo sigue avanzando de un lado a otro hacia el óptimo. Por definición, en el mínimo del objetivo a lo largo de una dirección dada, el gradiente en el punto final es ortogonal a esa dirección.

la forma:

$$d_t = \nabla_{\theta} J(\theta) + \beta_t d_{t-1} \quad (8.29)$$

dónde β_t es un coeficiente cuya magnitud controla cuánto de la dirección, d_{t-1} , debemos volver a agregar a la dirección de búsqueda actual.

dos direcciones, d_t y d_{t-1} , se definen como conjugados si d_{t-1} es tal que $d_t^T d_{t-1} = 0$, donde H es la matriz hessiana.

La forma sencilla de imponer la conjugación implicaría el cálculo de los vectores propios de H e elegir β_t , lo que no satisfaría nuestro objetivo de desarrollar un método que sea más viable computacionalmente que el método de Newton para problemas grandes. ¿Podemos calcular las direcciones conjugadas sin recurrir a estos cálculos? Afortunadamente la respuesta a eso es sí.

Dos métodos populares para calcular la β_t son:

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\theta} J(\theta_t)^T \nabla_{\theta} J(\theta_t) - \nabla_{\theta} J(\theta_{t-1})^T \nabla_{\theta} J(\theta_{t-1})}{\nabla_{\theta} J(\theta_t)^T \nabla_{\theta} J(\theta_t)} \quad (8.30)$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\theta} J(\theta_t) - \nabla_{\theta} J(\theta_{t-1})) \cdot \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1}) \cdot \nabla_{\theta} J(\theta_{t-1})} \quad (8.31)$$

Para una superficie cuadrática, las direcciones conjugadas aseguran que el gradiente a lo largo de la dirección anterior no aumente en magnitud. Por lo tanto, nos mantenemos en el mínimo a lo largo de las direcciones anteriores. Como consecuencia, en un k -espacio de parámetros dimensional, el método del gradiente conjugado requiere como máximo k búsquedas de línea para lograr el mínimo. El algoritmo de gradiente conjugado se da en algoritmo 8.9.

Algoritmo 8.9 El método del gradiente conjugado

Requerir: Parámetros iniciales θ

Requerir: Conjunto de entrenamiento de m ejemplos

Inicializar $\rho_0=0$

Inicializar $gramo_0$

=0 Inicializar $t=1$

mientras no se cumple el criterio de parada **hacer**

 Iniciar el degradado $gramo_t=0$

 Gradiente de cálculo: $gramo_t \leftarrow \nabla_{\theta} L(F(X(i); \theta), y(i))$

 Calcular $\beta_t = \frac{(gramo_t - g_{t-1}) \cdot gramo_t}{g_{t-1}^T gramo_{t-1}}$ (Polak-Ribière)

 (Gradiente conjugado no lineal: opcionalmente restablecer β_t a cero, por ejemplo si es múltiplo de alguna constante k , como $k=5$) Calcular la dirección de búsqueda: $\rho_t = -gramo_t + \beta_t \rho_{t-1}$

 Realice una búsqueda de línea para encontrar: $\rho_t = \arg \min_{\rho} L(F(X; \theta_t + \rho \rho_t), y)$

 (En una función de costo verdaderamente cuadrática, resuelva analíticamente para ρ en lugar de buscarlo explícitamente)

 Aplicar actualización: $\theta_{t+1} = \theta_t + \rho_t$

$t \leftarrow t+1$

terminar mientras

Gradientes conjugados no lineales: Hasta ahora hemos discutido el método de gradientes conjugados aplicado a funciones objetivo cuadráticas. Por supuesto, nuestro principal interés en este capítulo es explorar métodos de optimización para entrenar redes neuronales y otros modelos de aprendizaje profundo relacionados donde la función objetivo correspondiente está lejos de ser cuadrática. Quizás sorprendentemente, el método de gradientes conjugados todavía es aplicable en este entorno, aunque con algunas modificaciones. Sin ninguna seguridad de que el objetivo sea cuadrático, las direcciones conjugadas

ya no tienen la seguridad de permanecer en el mínimo del objetivo de las direcciones anteriores. Como resultado, el**gradientes conjugados no lineales**El algoritmo incluye reinicios ocasionales en los que el método de gradientes conjugados se reinicia con la búsqueda de líneas a lo largo del gradiente inalterado.

Los profesionales informan resultados razonables en las aplicaciones del algoritmo de gradientes conjugados no lineales para entrenar redes neuronales, aunque a menudo es beneficioso inicializar la optimización con algunas iteraciones de descenso de gradiente estocástico antes de comenzar los gradientes conjugados no lineales. Además, mientras que el algoritmo de gradientes conjugados (no lineales) se ha utilizado tradicionalmente como un método por lotes, las versiones de minilotes se han utilizado con éxito para el entrenamiento de redes neuronales ([Le et al., 2011](#)). Anteriormente se han propuesto adaptaciones de gradientes conjugados específicamente para redes neuronales, como el algoritmo de gradientes conjugados escalados ([Moller, 1993](#)).

8.6.3 BFGS

El**Algoritmo de Broyden-Fletcher-Goldfarb-Shanno (BFGS)**intenta traer algunas de las ventajas del método de Newton sin la carga computacional. En ese sentido, BFGS es similar al método de gradiente conjugado. Sin embargo, BFGS adopta un enfoque más directo para la aproximación de la actualización de Newton. Recuerde que la actualización de Newton está dada por

$$\theta^* = \theta_0 - H^{-1} \nabla \mathcal{J}(\theta_0), \quad (8.32)$$

dónde H es la arpillera de/con respecto a θ evaluado en θ_0 . La principal dificultad computacional al aplicar la actualización de Newton es el cálculo de la hessiana inversa H^{-1} . El enfoque adoptado por los métodos quasi-Newton (de los cuales el algoritmo BFGS es el más destacado) es aproximar la inversa con una matriz $METRO$ que se refina iterativamente mediante actualizaciones de rango bajo para convertirse en una mejor aproximación de H^{-1} .

La especificación y la derivación de la aproximación BFGS se dan en muchos libros de texto sobre optimización, incluyendo [Luenberger\(1984\)](#).

Una vez que la aproximación hessiana inversa $METRO$ se actualiza, la dirección de descenso p_t Esta determinado por $p_t = METRO_t gramot$. Se realiza una búsqueda de línea en esta dirección para determinar el tamaño del paso, $-*$, tomado en esta dirección. La actualización final de los parámetros viene dada por:

$$\theta_{t+1} = \theta_t + -*p_t. \quad (8.33)$$

Al igual que el método de gradientes conjugados, el algoritmo BFGS itera una serie de búsquedas de línea con la dirección que incorpora información de segundo orden. Sin embargo

a diferencia de los gradientes conjugados, el éxito del enfoque no depende en gran medida de que la búsqueda de la línea encuentre un punto muy cercano al mínimo verdadero a lo largo de la línea. Por lo tanto, en relación con los gradientes conjugados, BFGS tiene la ventaja de que puede dedicar menos tiempo a refinar cada línea de búsqueda. Por otro lado, el algoritmo BFGS debe almacenar la matriz hessiana inversa, $METRO$, eso requiere $O(norte^2)$ memoria, lo que hace que BFGS no sea práctico para la mayoría de los modelos modernos de aprendizaje profundo que normalmente tienen millones de parámetros.

BFGS de memoria limitada (o L-BFGS) Los costos de memoria del algoritmo BFGS se pueden reducir significativamente al evitar almacenar la aproximación hessiana inversa completa $METRO$. El algoritmo L-BFGS calcula la aproximación $METRO$ usando el mismo método que el algoritmo BFGS, pero comenzando con la suposición de que $METRO_{(t-1)}$ es la matriz identidad, en lugar de almacenar la aproximación de un paso al siguiente. Si se utiliza con búsquedas de línea exacta, las direcciones definidas por L-BFGS se conjugan mutuamente. Sin embargo, a diferencia del método de gradientes conjugados, este procedimiento se comporta bien cuando el mínimo de la línea de búsqueda se alcanza solo aproximadamente. La estrategia L-BFGS sin almacenamiento descrita aquí se puede generalizar para incluir más información sobre el Hessian almacenando algunos de los vectores utilizados para actualizar $METRO$ en cada paso de tiempo, lo que cuesta sólo $O(norte)$ por paso

8.7 Estrategias de optimización y meta-algoritmos

Muchas técnicas de optimización no son exactamente algoritmos, sino plantillas generales que se pueden especializar para producir algoritmos o subrutinas que se pueden incorporar en muchos algoritmos diferentes.

8.7.1 Normalización de lotes

Normalización por lotes ([Ioffe y Szegedy, 2015](#)) es una de las innovaciones recientes más emocionantes en la optimización de redes neuronales profundas y en realidad no es un algoritmo de optimización en absoluto. En cambio, es un método de reparametrización adaptativa, motivado por la dificultad de entrenar modelos muy profundos.

Los modelos muy profundos implican la composición de varias funciones o capas. El gradiente indica cómo actualizar cada parámetro, bajo el supuesto de que las otras capas no cambian. En la práctica, actualizamos todas las capas simultáneamente. Cuando hacemos la actualización, pueden ocurrir resultados inesperados porque muchas funciones compuestas juntas se cambian simultáneamente, utilizando actualizaciones que se calcularon bajo el supuesto de que las otras funciones permanecen constantes. como un simple

Por ejemplo, supongamos que tenemos una red neuronal profunda que tiene solo una unidad por capa y no usa una función de activación en cada capa oculta: $\hat{y} = w_1 w_2 w_3 \dots w_o$. Aquí, w_i proporciona el peso utilizado por capa i . La salida de la capa j es $h = h_{j-1} w_i$. La salida \hat{y} es una función lineal de la entrada X , sino una función no lineal de los pesos w_i . Supongamos que nuestra función de costo ha puesto un gradiente de 1 en \hat{y} , por lo que deseamos disminuir \hat{y} rápidamente. El algoritmo de retropropagación puede entonces calcular un gradiente $gramo = \nabla_{w_i} \hat{y}$. Considere lo que sucede cuando hacemos una actualización $w \leftarrow w - g$. La aproximación de la serie de Taylor de primer orden de \hat{y} predice que el valor de \hat{y} disminuirá por $-gramo$. Si quisieramos disminuir \hat{y} por 1, esta información de primer orden disponible en el gradiente sugiere que podríamos establecer la tasa de aprendizaje a 1. Sin embargo, el real la actualización incluirá efectos de segundo y tercer orden, hasta efectos de orden yo . El nuevo valor de \hat{y} es dado por

$$\hat{y}(w_1 - -gramo_1)(w_2 - -gramo_2) \dots (w_o - -gramo_o). \quad (8.34)$$

Un ejemplo de un término de segundo orden que surge de esta actualización es $-2gramo_1 gramo_2 + \dots + gramo_o$. Este término puede ser despreciable si $gramo_i$ es pequeño, o puede ser exponencialmente grande si los pesos en capas 3 a través de yo son mayores que 1. Esto hace que sea muy difícil elegir una tasa de aprendizaje adecuada, porque los efectos de una actualización de los parámetros de una capa dependen en gran medida de todas las demás capas. Los algoritmos de optimización de segundo orden abordan este problema al calcular una actualización que tiene en cuenta estas interacciones de segundo orden, pero podemos ver que en redes muy profundas, incluso las interacciones de orden superior pueden ser significativas. Incluso los algoritmos de optimización de segundo orden son costosos y, por lo general, requieren numerosas aproximaciones que les impiden tener en cuenta realmente todas las interacciones significativas de segundo orden. Construyendo un *norte* Algoritmo de optimización de $-ésimo$ orden para $norte > 2$ por lo tanto parece desesperado. ¿Qué podemos hacer en su lugar?

La normalización por lotes proporciona una forma elegante de reparametrizar casi cualquier red profunda. La reparametrización reduce significativamente el problema de coordinar actualizaciones en muchas capas. La normalización por lotes se puede aplicar a cualquier entrada o capa oculta en una red. Dejar H ser un minilote de activaciones de la capa a normalizar, organizado como una matriz de diseño, con las activaciones para cada ejemplo apareciendo en una fila de la matriz, para normalizar H , lo reemplazamos con

$$H = \frac{H - \mu}{\sigma}, \quad (8.35)$$

dónde μ es un vector que contiene la media de cada unidad y σ es un vector que contiene la desviación estándar de cada unidad. La aritmética aquí se basa en transmitir el vector μ y el vector σ para ser aplicado a cada fila de la matriz H . Dentro de cada fila, la aritmética es por elementos, por lo que $H_{yo, j}$ se normaliza restando μ_j

y dividiendo por σ . El resto de la red entonces opera en H exactamente de la misma forma en que operaba la red original H .

En el momento del entrenamiento,

$$\mu = \frac{1}{\text{metro}} \sum_i H_i; \quad (8.36)$$

y

$$\sigma = d + \frac{1}{\text{metro}} \sum_i (H - \mu)^2; \quad (8.37)$$

dónde d es un pequeño valor positivo como 10^{-8} impuesto para evitar encontrarse el gradiente indefinido de $z=0$. Crucialmente, nos retropropagamos a través de estas operaciones para calcular la media y la desviación estándar, y para aplicarlas para normalizar H . Esto significa que el gradiente nunca propondrá una operación que actúe simplemente para aumentar la desviación estándar o la media de h ; las operaciones de normalización eliminan el efecto de tal acción y ponen a cero su componente en el gradiente. Esta fue una gran innovación del enfoque de normalización por lotes. Los enfoques anteriores implicaron agregar penalizaciones a la función de costo para alentar a las unidades a tener estadísticas de activación normalizadas o involucraron intervenir para volver a normalizar las estadísticas de la unidad después de cada paso de descenso de gradiente. El primer enfoque generalmente resultó en una normalización imperfecta y el segundo generalmente resultó en una pérdida de tiempo significativa, ya que el algoritmo de aprendizaje proponía repetidamente cambiar la media y la varianza y el paso de normalización deshacía repetidamente este cambio. La normalización por lotes reparametriza el modelo para hacer que algunas unidades siempre estén estandarizadas por definición, eludiendo hábilmente ambos problemas.

En el momento de la prueba, μ y σ pueden ser reemplazados por promedios móviles que se recopilaron durante el tiempo de entrenamiento. Esto permite evaluar el modelo en un solo ejemplo, sin necesidad de utilizar definiciones de μ y σ que dependen de un minilote completo.

Revisando el $\hat{y} = xw_1 w_2 \dots w_o$ ejemplo, vemos que podemos resolver principalmente las dificultades en el aprendizaje de este modelo normalizando h_{yo-1} . Suponer que X se extrae de una unidad gaussiana. Entonces h_{yo-1} también vendrá de un gaussiano, porque la transformación de X a h_{yo-1} es lineal. Sin embargo, h_{yo-1} ya no tendrá media cero ni varianza unitaria. Después de aplicar la normalización por lotes, obtenemos el normalizado \hat{h}_{yo-1} que restablece las propiedades de media cero y varianza unitaria. Para casi cualquier actualización de las capas inferiores, \hat{h}_{yo-1} seguirá siendo una unidad gaussiana. La salida \hat{y} puede luego ser aprendida como una función lineal simple $\hat{y} = \hat{w}_{yo} \hat{h}_{yo-1}$. El aprendizaje en este modelo es ahora muy simple porque los parámetros en las capas inferiores simplemente no tienen efecto en la mayoría de los casos; su salida siempre se vuelve a normalizar a una unidad gaussiana. En algunos casos de esquina, las capas inferiores pueden tener un efecto. Cambiar uno de los pesos de la capa inferior a 0 puede hacer que la salida se vuelva degenerada, y cambiar el signo

de uno de los pesos más bajos puede cambiar la relación entre \hat{y}_{yo-1} y y . Estas situaciones son muy raras. Sin la normalización, casi todas las actualizaciones tendrían un efecto extremo en las estadísticas de h_{yo-1} . La normalización por lotes ha hecho que este modelo sea significativamente más fácil de aprender. En este ejemplo, la facilidad de aprendizaje, por supuesto, tuvo el costo de hacer que las capas inferiores fueran inútiles. En nuestro ejemplo lineal, las capas inferiores ya no tienen ningún efecto dañino, pero tampoco tienen ningún efecto beneficioso. Esto se debe a que hemos normalizado las estadísticas de primer y segundo orden, que es todo lo que puede influir una red lineal. En una red neuronal profunda con funciones de activación no lineales, las capas inferiores pueden realizar transformaciones no lineales de los datos, por lo que siguen siendo útiles. La normalización por lotes actúa para estandarizar solo la media y la varianza de cada unidad para estabilizar el aprendizaje, pero permite que cambien las relaciones entre las unidades y las estadísticas no lineales de una sola unidad.

Debido a que la capa final de la red puede aprender una transformación lineal, es posible que deseemos eliminar todas las relaciones lineales entre unidades dentro de una capa. De hecho, este es el enfoque adoptado por [Desjardin et al. \(2015\)](#), quien sirvió de inspiración para la normalización por lotes. Desafortunadamente, eliminar todas las interacciones lineales es mucho más costoso que estandarizar la media y la desviación estándar de cada unidad individual y, hasta ahora, la normalización por lotes sigue siendo el enfoque más práctico.

La normalización de la media y la desviación estándar de una unidad puede reducir el poder expresivo de la red neuronal que contiene esa unidad. Para mantener el poder expresivo de la red, es común reemplazar el lote de activaciones de unidades ocultas H con $y = \mu + \beta z$ en lugar de simplemente el normalizado H . Las variables μ y β son parámetros aprendidos que permiten que la nueva variable tenga cualquier media y desviación estándar. A primera vista, esto puede parecer inútil: ¿por qué fijamos la media en 0 , y luego introduzca un parámetro que permita volver a establecerlo en cualquier valor arbitrario β ? La respuesta es que la nueva parametrización puede representar la misma familia de funciones de entrada que la parametrización anterior, pero la nueva parametrización tiene una dinámica de aprendizaje diferente. En la antigua parametrización, la media de H fue determinado por una interacción complicada entre los parámetros en las capas debajo de H . En la nueva parametrización, la media de $y = \mu + \beta z$ está determinada únicamente por β . La nueva parametrización es mucho más fácil de aprender con descenso de gradiente.

La mayoría de las capas de redes neuronales toman la forma de $\varphi(XW + b)$ donde φ es una función de activación no lineal fija, como la transformación lineal rectificada. Es natural preguntarse si deberíamos aplicar la normalización por lotes a la entrada X , o al valor transformado $XW + b$. [Ioffe y Szegedy \(2015\)](#) recomienda

este último. Más específicamente, $XW+b$ debe ser reemplazada por una versión normalizada de XW . El término de sesgo debe omitirse porque se vuelve redundante con el β parámetro aplicado por la reparametrización de normalización por lotes. La entrada a una capa suele ser la salida de una función de activación no lineal, como la función lineal rectificada en una capa anterior. Las estadísticas de la entrada son, por lo tanto, más no gaussianas y menos susceptibles de estandarización mediante operaciones lineales.

En redes convolucionales, descritas en el capítulo 9, es importante aplicar la misma normalización $\mu\sigma$ en cada ubicación espacial dentro de un mapa de características, de modo que las estadísticas del mapa de características sigan siendo las mismas independientemente de la ubicación espacial.

8.7.2 Descenso coordinado

En algunos casos, puede ser posible resolver un problema de optimización rápidamente dividiéndolo en partes separadas. Si minimizamos $F(X)$ con respecto a una sola variable X_i , luego minimizarlo con respecto a otra variable X_j y así sucesivamente, pasando repetidamente por todas las variables, tenemos la garantía de llegar a un mínimo (local). Esta práctica se conoce como **descenso coordinado**, porque optimizamos una coordenada a la vez. Más generalmente, **descenso de coordenadas de bloques** se refiere a minimizar con respecto a un subconjunto de las variables simultáneamente. El término “descenso coordinado” se usa a menudo para referirse al descenso coordinado de bloques, así como al descenso coordinado estrictamente individual.

El descenso de coordenadas tiene más sentido cuando las diferentes variables en el problema de optimización se pueden separar claramente en grupos que juegan roles relativamente aislados, o cuando la optimización con respecto a un grupo de variables es significativamente más eficiente que la optimización con respecto a todas las variables. Por ejemplo, considere la función de costo

$$j(alto, ancho) = \sum_{yo,j} |H_{yo,j}|^2 + \frac{1}{2} \sum_{yo,j} \|X \cdot W - H\|_{yo,j}^2. \quad (8.38)$$

Esta función describe un problema de aprendizaje llamado codificación dispersa, donde el objetivo es encontrar una matriz de peso W que puede decodificar linealmente una matriz de valores de activación H para reconstruir el conjunto de entrenamiento X . La mayoría de las aplicaciones de codificación dispersa también implican una disminución del peso o una restricción en las normas de las columnas de W , con el fin de prevenir la solución patológica con extremadamente pequeño H y largo W .

La función j no es convexo. Sin embargo, podemos dividir las entradas del algoritmo de entrenamiento en dos conjuntos: los parámetros del diccionario W y las representaciones del código H . Minimizar la función objetivo con respecto a cualquiera de estos conjuntos de variables es un problema convexo. El descenso de coordenadas de bloque da así

una estrategia de optimización que nos permite utilizar algoritmos de optimización convexa eficientes, alternando entre optimizar \mathcal{W} con H fijo, luego optimizando H con \mathcal{W} fijo

El descenso de coordenadas no es una muy buena estrategia cuando el valor de una variable influye fuertemente en el valor óptimo de otra variable, como en la función $F(X) = (X_1 - X_2)^2 + \alpha x_2 - 1 + X_2$ donde α es una constante positiva. El primer término anima las dos variables a tener un valor similar, mientras que el segundo término las alienta a estar cerca de cero. La solución es poner ambos a cero. El método de Newton puede resolver el problema en un solo paso porque es un problema cuadrático definido positivo. Sin embargo, para pequeños α , el descenso de coordenadas avanzará muy lentamente porque el primer término no permite cambiar una sola variable a un valor que difiera significativamente del valor actual de la otra variable.

8.7.3 Promedio Polyak

Promedio de Polyak ([Polyak y Juditsky, 1992](#)) consiste en promediar varios puntos de la trayectoria a través del espacio de parámetros visitado por una optimización algoritmo. Si t iteraciones de puntos de visita de descenso de gradiente $\theta^{(1)}, \dots, \theta^{(t)}$, entonces el la salida del algoritmo de promediación de Polyak es $\hat{\theta}^{(t)} = \frac{1}{t} \sum_{i=1}^t \theta^{(i)}$. En algún problema clases, como el descenso de gradiente aplicado a problemas convexos, este enfoque tiene fuertes garantías de convergencia. Cuando se aplica a redes neuronales, su justificación es más heurística, pero funciona bien en la práctica. La idea básica es que el algoritmo de optimización puede saltar de un lado a otro de un valle varias veces sin visitar nunca un punto cerca del fondo del valle. Sin embargo, el promedio de todas las ubicaciones a ambos lados debería estar cerca del fondo del valle.

En problemas no convexos, el camino tomado por la trayectoria de optimización puede ser muy complicado y visitar muchas regiones diferentes. Incluir puntos en el espacio de parámetros del pasado distante que pueden estar separados del punto actual por grandes barreras en la función de costo no parece un comportamiento útil. Como resultado, cuando se aplica el promedio de Polyak a problemas no convexos, es típico usar un promedio móvil que decrece exponencialmente:

$$\hat{\theta}^{(t)} = \alpha \hat{\theta}^{(t-1)} + (1 - \alpha) \theta^{(t)}. \quad (8.39)$$

El enfoque del promedio móvil se utiliza en numerosas aplicaciones. Ver [Szegedy et al. \(2015\)](#) para un ejemplo reciente.

8.7.4 Entrenamiento previo supervisado

A veces, entrenar directamente un modelo para resolver una tarea específica puede ser demasiado ambicioso si el modelo es complejo y difícil de optimizar o si la tarea es muy difícil. A veces es más efectivo entrenar un modelo más simple para resolver la tarea y luego hacer que el modelo sea más complejo. También puede ser más efectivo entrenar al modelo para que resuelva una tarea más simple y luego pasar a enfrentar la tarea final. Estas estrategias que implican entrenar modelos simples en tareas simples antes de enfrentar el desafío de entrenar el modelo deseado para realizar la tarea deseada se conocen colectivamente como **Pre-entrenamiento**.

Avaro Los algoritmos dividen un problema en muchos componentes y luego resuelven la versión óptima de cada componente de forma aislada. Desafortunadamente, no se garantiza que la combinación de los componentes óptimos individualmente produzca una solución completa óptima. Sin embargo, los algoritmos codiciosos pueden ser computacionalmente mucho más baratos que los algoritmos que resuelven la mejor solución conjunta, y la calidad de una solución codiciosa suele ser aceptable, si no óptima. Los algoritmos codiciosos también pueden ser seguidos por un **sintonía FINA** etapa en la que un algoritmo de optimización conjunta busca una solución óptima al problema completo. Inicializar el algoritmo de optimización conjunta con una solución codiciosa puede acelerarlo en gran medida y mejorar la calidad de la solución que encuentra.

Los algoritmos de preentrenamiento, y especialmente de preentrenamiento codicioso, son omnipresentes en el aprendizaje profundo. En esta sección, describimos específicamente los algoritmos de preentrenamiento que dividen los problemas de aprendizaje supervisado en otros problemas de aprendizaje supervisado más simples. Este enfoque se conoce como **preentrenamiento supervisado codicioso**.

En el original ([bengio et al., 2007](#)) del preentrenamiento supervisado codicioso, cada etapa consiste en una tarea de entrenamiento de aprendizaje supervisado que involucra solo un subconjunto de las capas en la red neuronal final. En la figura se ilustra un ejemplo de preentrenamiento supervisado codicioso.[8.7](#), en el que cada capa oculta agregada se entrena previamente como parte de un MLP superficial supervisado, tomando como entrada la salida de la capa oculta previamente entrenada. En lugar de entrenar previamente una capa a la vez, [Simonyan y Zisserman \(2015\)](#) preentrenar una red convolucional profunda (once capas de peso) y luego usar las primeras cuatro y las últimas tres capas de esta red para inicializar redes aún más profundas (con hasta diecinueve capas de peso). Las capas intermedias de la nueva red muy profunda se inicializan aleatoriamente. A continuación, la nueva red se entrena de forma conjunta. Otra opción, explorada por [Yuet al. \(2010\)](#) es usar las salidas de los MLP previamente entrenados, así como la entrada en bruto, como entradas para cada etapa añadida.

¿Por qué ayudaría el preentrenamiento supervisado codicioso? La hipótesis discutida inicialmente por [bengio et al. \(2007\)](#) es que ayuda a orientar mejor a los

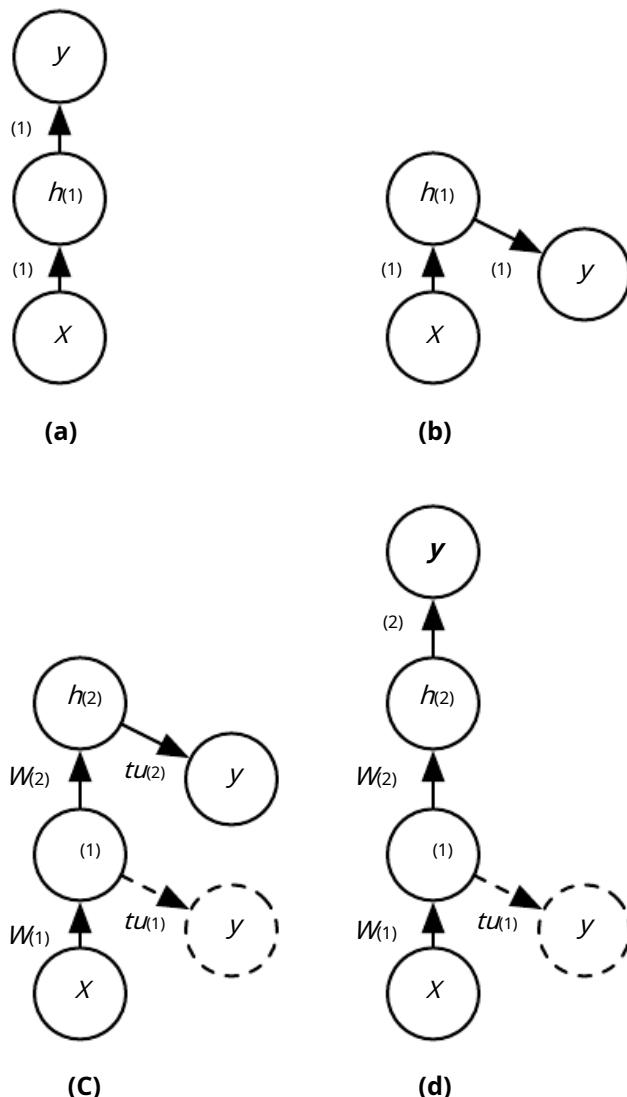


Figura 8.7: Ilustración de una forma de preentrenamiento supervisado codicioso ([bengio et al., 2007](#)).
 (a) Empezamos entrenando una arquitectura suficientemente superficial.(b) Otro dibujo de la misma arquitectura.(c) Mantenemos solo la capa de entrada a oculta de la red original y descartamos la capa de oculta a salida. Enviamos la salida de la primera capa oculta como entrada a otro MLP de una sola capa oculta supervisado que está entrenado con el mismo objetivo que la primera red, agregando así una segunda capa oculta. Esto se puede repetir para tantas capas como se desee.(d) Otro dibujo del resultado, visto como una red de avance. Para mejorar aún más la optimización, podemos ajustar conjuntamente todas las capas, ya sea solo al final o en cada etapa de este proceso.

niveles intermedios de una jerarquía profunda. En general, el entrenamiento previo puede ayudar tanto en términos de optimización como de generalización.

Un enfoque relacionado con el preentrenamiento supervisado extiende la idea al contexto del aprendizaje por transferencia: [Yosinski et al. \(2014\)](#) preentrenar una red convolucional profunda con 8 capas de pesos en un conjunto de tareas (un subconjunto de las 1000 categorías de objetos de ImageNet) y luego inicializar una red del mismo tamaño con la primera k capas de la primera red. Todas las capas de la segunda red (con las capas superiores inicializadas aleatoriamente) se entranan conjuntamente para realizar un conjunto diferente de tareas (otro subconjunto de las 1000 categorías de objetos de ImageNet), con menos ejemplos de entrenamiento que para el primer conjunto de tareas. Otros enfoques para transferir el aprendizaje con redes neuronales se analizan en la sección [15.2.](#)

8.7.5 Diseño de modelos para ayudar a la optimización

Para mejorar la optimización, la mejor estrategia no siempre es mejorar el algoritmo de optimización. En cambio, muchas mejoras en la optimización de los modelos profundos provienen del diseño de los modelos para que sean más fáciles de optimizar.

En principio, podríamos usar funciones de activación que aumentan y disminuyen en patrones irregulares no monótonos. Sin embargo, esto haría que la optimización fuera extremadamente difícil. En la práctica, *es más importante elegir una familia de modelos que sea fácil de optimizar que utilizar un potente algoritmo de optimización*. La mayoría de los avances en el aprendizaje de redes neuronales en los últimos 30 años se han obtenido cambiando la familia de modelos en lugar de cambiar el procedimiento de optimización. El descenso de gradiente estocástico con impulso, que se utilizó para entrenar redes neuronales en la década de 1980, sigue en uso en las aplicaciones modernas de redes neuronales de última generación.

Especificamente, las redes neuronales modernas reflejan una elección de diseño utilizar transformaciones lineales entre capas y funciones de activación que son diferenciables en casi todas partes y tienen pendiente significativa en gran parte de su dominio. En particular, las innovaciones de modelos como el LSTM, las unidades lineales rectificadas y las unidades maxout se han movido hacia el uso de más funciones lineales que los modelos anteriores, como redes profundas basadas en unidades sigmoidales. Estos modelos tienen buenas propiedades que facilitan la optimización. El gradiente fluye a través de muchas capas siempre que el jacobiano de la transformación lineal tenga valores singulares razonables. Además, las funciones lineales aumentan constantemente en una sola dirección, por lo que incluso si la salida del modelo está muy lejos de ser correcta, simplemente calculando el gradiente, queda claro en qué dirección debe moverse su salida para reducir la función de pérdida. En otras palabras, *local* la información de gradiente corresponde razonablemente bien a moverse hacia una solución distante.

Otras estrategias de diseño de modelos pueden ayudar a facilitar la optimización. Por ejemplo, las rutas lineales o las conexiones de salto entre capas reducen la longitud de la ruta más corta desde los parámetros de la capa inferior hasta la salida y, por lo tanto, mitigan el problema del gradiente de desaparición ([Srivastava et al., 2015](#)). Una idea relacionada para omitir conexiones es agregar copias adicionales de la salida que se adjuntan a las capas ocultas intermedias de la red, como en GoogLeNet ([Szegedy et al., 2014a](#)) y redes profundamente supervisadas ([Sotavento et al., 2014](#)). Estos "cabezales auxiliares" están capacitados para realizar la misma tarea que la salida principal en la parte superior de la red para garantizar que las capas inferiores reciban un gran gradiente. Cuando se completa el entrenamiento, los cabezales auxiliares pueden descartarse. Esta es una alternativa a las estrategias de preentrenamiento, que se introdujeron en la sección anterior. De esta forma, se pueden entrenar todas las capas conjuntamente en una sola fase pero cambiando la arquitectura, para que las capas intermedias (especialmente las inferiores) puedan obtener algunas pistas sobre lo que hacen.

debe hacer, a través de un camino más corto. Estas sugerencias proporcionan una señal de error a las capas inferiores.

8.7.6 Métodos de continuación y aprendizaje del currículo

Como se argumenta en la sección 8.2.7, muchos de los desafíos en la optimización surgen de la estructura global de la función de costo y no pueden resolverse simplemente haciendo mejores estimaciones de las direcciones de actualización locales. La estrategia predominante para superar este problema es intentar inicializar los parámetros en una región que está conectada a la solución por un camino corto a través del espacio de parámetros que puede descubrir la descendencia local.

Métodos de continuación son una familia de estrategias que pueden facilitar la optimización eligiendo puntos iniciales para garantizar que la optimización local pase la mayor parte de su tiempo en regiones del espacio que se comportan bien. La idea detrás de los métodos de continuación es construir una serie de funciones objetivo sobre los mismos parámetros. Para minimizar una función de costo $J(\theta)$, construiremos nuevas funciones de costo $\{J_0, \dots, J_{(norte)}\}$. Estas funciones de costos están diseñadas para ser cada vez más difíciles, convirtiéndose bastante fácil de minimizar, y $J_{(norte)}$, lo más difícil, siendo $J(\theta)$, la verdadera función de coste que motiva todo el proceso. cuando decimos eso J_i es más fácil que J_{i+1} , queremos decir que se comporta bien en más de θ espacio. Es más probable que una inicialización aleatoria aterrice en la región donde la descendencia local puede minimizar la función de costo con éxito porque esta región es más grande. Las series de funciones de costo están diseñadas para que la solución de una sea un buen punto inicial de la siguiente. Por lo tanto, comenzamos resolviendo un problema fácil y luego refinamos la solución para resolver problemas cada vez más difíciles hasta que llegamos a una solución para el verdadero problema subyacente.

Los métodos de continuación tradicionales (anteriores al uso de métodos de continuación para el entrenamiento de redes neuronales) generalmente se basan en suavizar la función objetivo. Ver Wu (1997) para un ejemplo de tal método y una revisión de algunos métodos relacionados. Los métodos de continuación también están estrechamente relacionados con el recocido simulado, que agrega ruido a los parámetros (Kirkpatrick et al., 1983). Los métodos de continuación han tenido un gran éxito en los últimos años. Ver mobahi y pescador (2015) para obtener una descripción general de la literatura reciente, especialmente para aplicaciones de IA.

Tradicionalmente, los métodos de continuación se diseñaron principalmente con el objetivo de superar el desafío de los mínimos locales. Específicamente, fueron diseñados para alcanzar un mínimo global a pesar de la presencia de muchos mínimos locales. Para hacerlo, estos métodos de continuación construirían funciones de costo más fáciles al “difuminar” la función de costo original. Esta operación de desenfoque se puede hacer aproximando

$$J_i(\theta) = \min_{\theta' \sim N(\theta; \theta, \sigma^2)} J(\theta') \quad (8.40)$$

mediante muestreo. La intuición de este enfoque es que algunas funciones no convexas

se vuelven aproximadamente convexos cuando están borrosos. En muchos casos, este desenfoque conserva suficiente información sobre la ubicación de un mínimo global para que podamos encontrar el mínimo global resolviendo versiones del problema progresivamente menos borrosas. Este enfoque puede descomponerse de tres maneras diferentes. En primer lugar, podría definir con éxito una serie de funciones de costo donde la primera es convexa y el óptimo rastrea de una función a la siguiente llegando al mínimo global, pero podría requerir tantas funciones de costo incremental que el costo de todo el procedimiento sigue siendo alto. . Los problemas de optimización NP-hard siguen siendo NP-hard, incluso cuando se aplican métodos de continuación. Las otras dos formas en que fallan los métodos de continuación corresponden a que el método no es aplicable. Primero, la función podría no volverse convexa, sin importar cuánto esté borrosa. $J(\theta) = -\theta \cdot \theta$. En segundo lugar, la función puede volverse convexa como resultado del desenfoque, pero el mínimo de esta función desdibujada puede rastrear un mínimo local en lugar de un mínimo global de la función de costo original.

Aunque los métodos de continuación se diseñaron principalmente para tratar el problema de los mínimos locales, ya no se cree que los mínimos locales sean el problema principal para la optimización de redes neuronales. Afortunadamente, los métodos de continuación todavía pueden ayudar. Las funciones objetivas más sencillas introducidas por el método de continuación pueden eliminar regiones planas, disminuir la varianza en las estimaciones de gradiente, mejorar el condicionamiento de la matriz hessiana o hacer cualquier otra cosa que facilite el cálculo de las actualizaciones locales o mejore la correspondencia entre las direcciones de actualización local y el progreso. hacia una solución global.

bengio et al.(2009) observó que un enfoque llamado **aprendizaje curricular o formación** puede interpretarse como un método de continuación. El aprendizaje del plan de estudios se basa en la idea de planificar un proceso de aprendizaje para comenzar aprendiendo conceptos simples y progresar hacia el aprendizaje de conceptos más complejos que dependen de estos conceptos más simples. Anteriormente se sabía que esta estrategia básica aceleraba el progreso en el entrenamiento de animales (Desollador,1958;Peterson,2004;Krueger y Dayan,2009) y aprendizaje automático (solomonoff,1989;Elman,1993;Sanger,1994).bengio et al.(2009) justificaron esta estrategia como un método de continuación, donde anteriormente se hace más fáciles aumentando la influencia de ejemplos más simples (ya sea asignando coeficientes más grandes a sus contribuciones a la función de costo, o muestreándolos con más frecuencia), y se demostró experimentalmente que se pueden obtener mejores resultados siguiendo un plan de estudios en una red neuronal a gran escala. tarea de modelado del lenguaje. El aprendizaje del plan de estudios ha tenido éxito en una amplia gama de lenguaje natural (Spitkovskiet al., 2010; colobertoet al.,2011a;mikolovet al.,2011b;Tu y Honavar,2011) y visión artificial (Kumaret al., 2010;Lee y Grauman,2011;Supancic y Ramanan,2013) tareas. También se verificó que el aprendizaje del currículo es consistente con la forma en que los humanosenseñar(Kanet al.,2011): los profesores empiezan mostrando más fácil y

ejemplos más prototípicos y luego ayudar al alumno a refinar la superficie de decisión con los casos menos obvios. Las estrategias basadas en el currículo son *más efectivas* para la enseñanza de seres humanos que las estrategias basadas en muestras uniformes de ejemplos, y también puede aumentar la eficacia de otras estrategias de enseñanza (Basu y Christensen, 2013).

Otra importante contribución a la investigación sobre el aprendizaje curricular surgió en el contexto del entrenamiento de redes neuronales recurrentes para capturar dependencias a largo plazo: Zaremba y Sutskever(2014) encontraron que se obtenían resultados mucho mejores con un *currículum estocástico*, en el que siempre se presenta al alumno una mezcla aleatoria de ejemplos fáciles y difíciles, pero donde la proporción promedio de los ejemplos más difíciles (aquí, aquellos con dependencias a largo plazo) se incrementa gradualmente. Con un plan de estudios determinista, no se observó ninguna mejora con respecto a la línea de base (entrenamiento ordinario del conjunto de entrenamiento completo).

Ahora hemos descrito la familia básica de modelos de redes neuronales y cómo regularizarlos y optimizarlos. En los capítulos siguientes, nos dirigimos a las especializaciones de la familia de redes neuronales, que permiten escalar las redes neuronales a tamaños muy grandes y procesar datos de entrada que tienen una estructura especial. Los métodos de optimización discutidos en este capítulo a menudo son directamente aplicables a estas arquitecturas especializadas con poca o ninguna modificación.

Capítulo 9

Redes Convolucionales

Redes convolucionales(lecun,1989), también conocido como**redes neuronales convolucionales** o CNN, son un tipo especializado de red neuronal para procesar datos que tienen una topología conocida similar a una cuadrícula. Los ejemplos incluyen datos de series temporales, que se pueden considerar como una cuadrícula 1D que toma muestras a intervalos de tiempo regulares, y datos de imágenes, que se pueden considerar como una cuadrícula 2D de píxeles. Las redes convolucionales han tenido un gran éxito en aplicaciones prácticas. El nombre "red neuronal convolucional" indica que la red emplea una operación matemática llamada **circunvolución**. La convolución es un tipo especializado de operación lineal.*Las redes convolucionales son simplemente redes neuronales que utilizan la convolución en lugar de la multiplicación general de matrices en al menos una de sus capas.*

En este capítulo, primero describiremos qué es la convolución. A continuación, explicaremos la motivación detrás del uso de la convolución en una red neuronal. Luego describiremos una operación llamada **puesta en común**, que emplean casi todas las redes convolucionales. Por lo general, la operación utilizada en una red neuronal convolucional no se corresponde exactamente con la definición de convolución que se utiliza en otros campos como la ingeniería o las matemáticas puras. Describiremos varias variantes de la función de convolución que se utilizan ampliamente en la práctica para redes neuronales. También mostraremos cómo se puede aplicar la convolución a muchos tipos de datos, con diferentes números de dimensiones. Luego discutimos los medios para hacer que la convolución sea más eficiente. Las redes convolucionales se destacan como un ejemplo de principios neurocientíficos que influyen en el aprendizaje profundo. Discutiremos estos principios neurocientíficos y luego concluiremos con comentarios sobre el papel que han jugado las redes convolucionales en la historia del aprendizaje profundo. Un tema que este capítulo no aborda es cómo elegir la arquitectura de su red convolucional. El objetivo de este capítulo es describir los tipos de herramientas que proporcionan las redes convolucionales, mientras que el capítulo 11

describe las pautas generales para elegir qué herramientas usar en qué circunstancias. La investigación sobre arquitecturas de redes convolucionales avanza tan rápidamente que cada pocas semanas o meses se anuncia una nueva mejor arquitectura para un punto de referencia determinado, lo que hace que sea poco práctico describir la mejor arquitectura en forma impresa. Sin embargo, las mejores arquitecturas se han compuesto consistentemente de los bloques de construcción descritos aquí.

9.1 La operación de convolución

En su forma más general, la convolución es una operación sobre dos funciones de un argumento de valor real. Para motivar la definición de convolución, comenzamos con ejemplos de dos funciones que podríamos usar.

Supongamos que estamos rastreando la ubicación de una nave espacial con un sensor láser. Nuestro sensor láser proporciona una salida única $X(t)$, la posición de la nave espacial en el momento t . Ambos X y w son de valor real, es decir, podemos obtener una lectura diferente del sensor láser en cualquier instante de tiempo.

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

$$s(t) = \int X(a)w(t - u)da \quad (9.1)$$

Esta operación se llama **circunvolución**. La operación de convolución generalmente se indica con un asterisco:

$$s(t) = (X * w)(t) \quad (9.2)$$

En nuestro ejemplo, w debe ser una función de densidad de probabilidad válida, o el resultado no es un promedio ponderado. También, w necesita ser 0 para todos los argumentos negativos, o mirará hacia el futuro, que presumiblemente está más allá de nuestras capacidades. Sin embargo, estas limitaciones son particulares de nuestro ejemplo. En general, la convolución se define para cualquier función para la cual se define la integral anterior y se puede usar para otros fines además de tomar promedios ponderados.

En la terminología de red convolucional, el primer argumento (en este ejemplo, la función X) a la convolución se refiere a menudo como **el portey** el segundo

argumento (en este ejemplo, la función w) como el **núcleo**. La salida a veces se denominan **mapa de características**.

En nuestro ejemplo, la idea de un sensor láser que pueda proporcionar mediciones en todo momento no es realista. Por lo general, cuando trabajamos con datos en una computadora, el tiempo se discretizará y nuestro sensor proporcionará datos a intervalos regulares. En nuestro ejemplo, podría ser más realista suponer que nuestro láser proporciona una medición una vez por segundo. El índice de tiempo t puede entonces tomar sólo valores enteros. Si ahora suponemos que X y w se definen solo en enteros t , podemos definir la convolución discreta:

$$S(t) = (X * w)(t) = \sum_{a=-\infty}^{+\infty} X(a)w(t - a) \quad (9.3)$$

En las aplicaciones de aprendizaje automático, la entrada suele ser una matriz multidimensional de datos y el kernel suele ser una matriz multidimensional de parámetros que se adaptan mediante el algoritmo de aprendizaje. Nos referiremos a estos arreglos multidimensionales como tensores. Debido a que cada elemento de la entrada y el núcleo deben almacenarse explícitamente por separado, generalmente asumimos que estas funciones son cero en todas partes excepto en el conjunto finito de puntos para los que almacenamos los valores. Esto significa que en la práctica podemos implementar la suma infinita como una suma sobre un número finito de elementos de matriz.

Finalmente, a menudo usamos circunvoluciones sobre más de un eje a la vez. Por ejemplo, si usamos una imagen bidimensional I como nuestra entrada, probablemente también querremos usar un núcleo bidimensional k :

$$S(yo, j) = (I * k)(yo, j) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} I(Minnesota)k(yo - m, j - norte). \quad (9.4)$$

La convolución es comutativa, lo que significa que podemos escribir de manera equivalente:

$$S(yo, j) = (k * I)(yo, j) = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} I(yo - m, j - norte)k(Minnesota). \quad (9.5)$$

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

La propiedad comutativa de la convolución surge porque tenemos **Floridapicado** el núcleo en relación con la entrada, en el sentido de que como *metro* aumenta, el índice en la entrada aumenta, pero el índice en el kernel disminuye. La única razón para invertir el núcleo es obtener la propiedad comutativa. Mientras que la propiedad comutativa

es útil para escribir pruebas, por lo general no es una propiedad importante de una implementación de red neuronal. En cambio, muchas bibliotecas de redes neuronales implementan una función relacionada llamada **correlación cruzada**, que es lo mismo que convolución pero sin invertir el núcleo:

$$S(yo, j) = (I * k)(yo, j) = \sum_{\substack{m \\ norte}} I(i+m, j+norte)k(Minnesota). \quad (9.6)$$

Muchas bibliotecas de aprendizaje automático implementan la correlación cruzada, pero la llaman convolución. En este texto seguiremos esta convención de llamar a ambas operaciones convolución, y especificaremos si nos referimos a voltear el kernel o no en contextos donde el cambio de kernel es relevante. En el contexto del aprendizaje automático, el algoritmo de aprendizaje aprenderá los valores apropiados del kernel en el lugar apropiado, por lo que un algoritmo basado en convolución con voldeo del kernel aprenderá un kernel que está volteado en relación con el kernel aprendido por un algoritmo sin el voldeo. . También es raro que la convolución se use sola en el aprendizaje automático; en cambio, la convolución se usa simultáneamente con otras funciones, y la combinación de estas funciones no commuta independientemente de si la operación de convolución invierte su núcleo o no.

Ver figura 9.1 para ver un ejemplo de convolución (sin cambio de kernel) aplicado a un tensor 2-D.

La convolución discreta puede verse como una multiplicación por una matriz. Sin embargo, la matriz tiene varias entradas restringidas para ser iguales a otras entradas. Por ejemplo, para la convolución discreta univariante, cada fila de la matriz está restringida para ser igual a la fila anterior desplazada por un elemento. Esto se conoce como **Matriz de Toeplitz**. En dos dimensiones, un **matriz circulante de doble bloque** corresponde a la convolución. Además de estas restricciones de que varios elementos sean iguales entre sí, la convolución generalmente corresponde a una matriz muy dispersa (una matriz cuyas entradas son en su mayoría iguales a cero). Esto se debe a que el kernel suele ser mucho más pequeño que la imagen de entrada. Cualquier algoritmo de red neuronal que funcione con la multiplicación de matrices y que no dependa de propiedades específicas de la estructura de la matriz debería funcionar con convolución, sin necesidad de realizar más cambios en la red neuronal. Las redes neuronales convolucionales típicas hacen uso de más especializaciones para manejar grandes entradas de manera eficiente, pero estas no son estrictamente necesarias desde una perspectiva teórica.

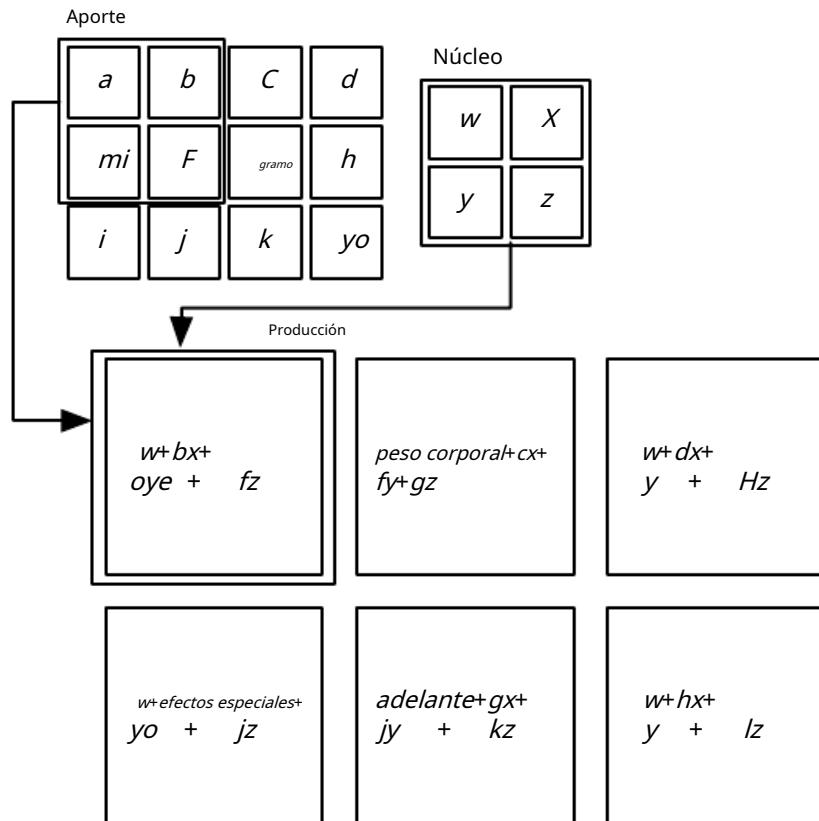


Figura 9.1: Un ejemplo de convolución 2-D sin cambio de kernel. En este caso, restringimos la salida solo a posiciones en las que el kernel se encuentra completamente dentro de la imagen, lo que se denomina convolución "válida" en algunos contextos. Dibujamos cuadros con flechas para indicar cómo se forma el elemento superior izquierdo del tensor de salida aplicando el kernel a la región superior izquierda correspondiente del tensor de entrada.

9.2 Motivación

La convolución aprovecha tres ideas importantes que pueden ayudar a mejorar un sistema de aprendizaje automático:**interacciones escasas, intercambio de parámetros y representaciones equivalentes**. Además, la convolución proporciona un medio para trabajar con entradas de tamaño variable. Ahora describimos cada una de estas ideas a su vez.

Las capas de redes neuronales tradicionales utilizan la multiplicación de matrices por una matriz de parámetros con un parámetro separado que describe la interacción entre cada unidad de entrada y cada unidad de salida. Esto significa que cada unidad de salida interactúa con cada unidad de entrada. Sin embargo, las redes convolucionales suelen tener**interacciones escasas** (también conocido como**escasa conectividad pesos escasos**). Esto se logra haciendo que el núcleo sea más pequeño que la entrada. Por ejemplo, al procesar una imagen, la imagen de entrada puede tener miles o millones de píxeles, pero podemos detectar características pequeñas y significativas, como bordes con núcleos que ocupan solo decenas o cientos de píxeles. Esto significa que necesitamos almacenar menos parámetros, lo que reduce los requisitos de memoria del modelo y mejora su eficiencia estadística. También significa que calcular la salida requiere menos operaciones. Estas mejoras en la eficiencia suelen ser bastante grandes. Si hay m entradas y n salidas, entonces la multiplicación de matrices requiere $m \times n$ parámetros y los algoritmos utilizados en la práctica han $\mathcal{O}(m \times n)$ tiempo de ejecución (por ejemplo). Si limitamos el número de conexiones que cada salida puede tener que k , entonces el enfoque escasamente conectado requiere solo $k \times n$ parámetros y $\mathcal{O}(k \times n)$ tiempo de ejecución. Para muchas aplicaciones prácticas, es posible obtener un buen rendimiento en la tarea de aprendizaje automático manteniendo k varios órdenes de magnitud menor que m . Para demostraciones gráficas de conectividad escasa, consulte la figura 9.2 y figura 9.3. En una red convolucional profunda, las unidades en las capas más profundas pueden *indirectamente* interactuar con una porción más grande de la entrada, como se muestra en la figura 9.4. Esto permite que la red describa de manera eficiente interacciones complicadas entre muchas variables mediante la construcción de tales interacciones a partir de bloques de construcción simples que describen solo interacciones escasas.

Compartir parámetros se refiere al uso del mismo parámetro para más de una función en un modelo. En una red neuronal tradicional, cada elemento de la matriz de peso se usa exactamente una vez al calcular la salida de una capa. Se multiplica por un elemento de la entrada y luego nunca se vuelve a visitar. Como sinónimo de compartir parámetros, se puede decir que una red tiene **pesos atados**, porque el valor del peso aplicado a una entrada está ligado al valor de un peso aplicado en otro lugar. En una red neuronal convolucional, cada miembro del kernel se usa en todas las posiciones de la entrada (excepto quizás algunos de los píxeles del límite, dependiendo de las decisiones de diseño con respecto al límite). El uso compartido de parámetros utilizado por la operación de convolución significa que, en lugar de aprender un conjunto separado de parámetros

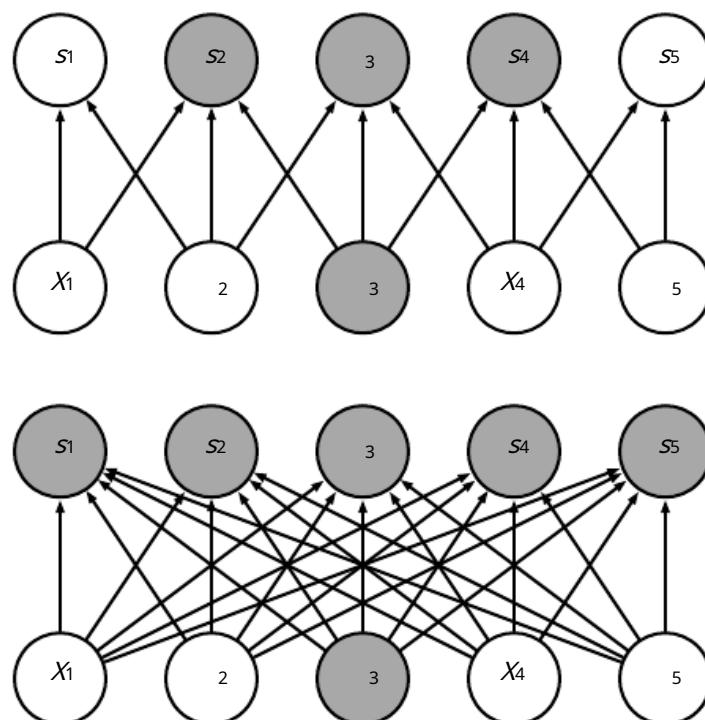


Figura 9.2: *Conejividad escasa, vista desde abajo.* Destacamos una unidad de entrada, X_3 y también resalte las unidades de salida en que se ven afectados por esta unidad. (Arriba) Cuando se está formado por convolución con un núcleo de ancho 3, sólo tres salidas se ven afectadas por X_3 . (Abajo) Cuando se forma por multiplicación de matrices, la conectividad ya no es escasa, por lo que todas las salidas se ven afectadas por X_3 .

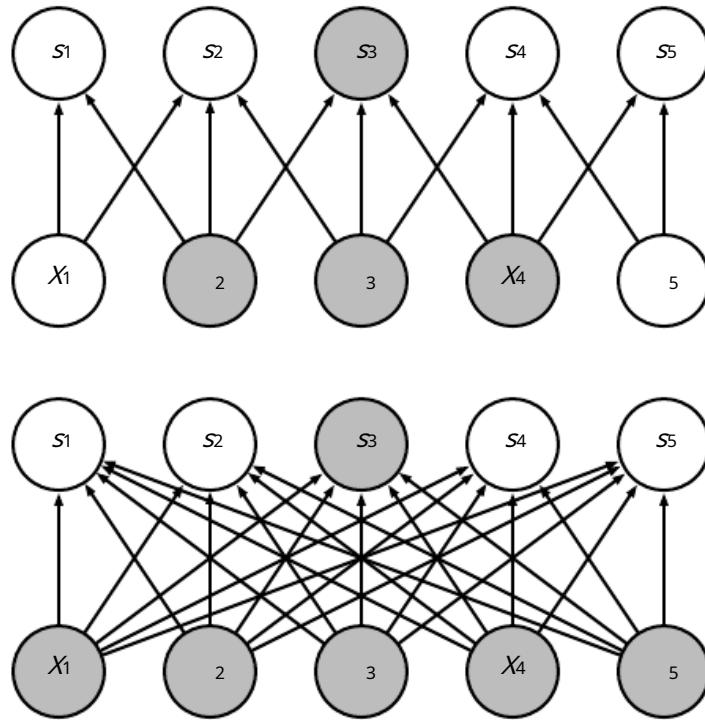


Figura 9.3: *Coneectividad escasa, vista desde arriba:* Destacamos una unidad de salida, s_3 y también resalte las unidades de entrada en X que afectan a esta unidad. Estas unidades se conocen como **campo receptivo** de s_3 . (Arriba) Cuando se está formado por convolución con un núcleo de ancho 3, solo afectan tres entradas a s_3 . (Abajo) Cuando se forma por multiplicación de matrices, la conectividad ya no es escasa, por lo que todas las entradas afectan a s_3 .

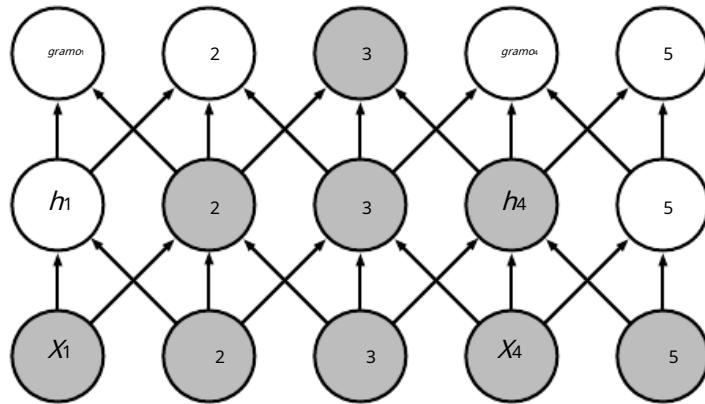


Figura 9.4: El campo receptivo de las unidades en las capas más profundas de una red convolucional es mayor que el campo receptivo de las unidades en las capas superficiales. Este efecto aumenta si la red incluye características arquitectónicas como convolución estriada (figura 9.12) o agrupación (sección 9.3). Esto significa que aunque *directo* conexiones en una red convolucional son muy escasas, las unidades en las capas más profundas pueden ser *indirectamente* conectado a toda o la mayor parte de la imagen de entrada.

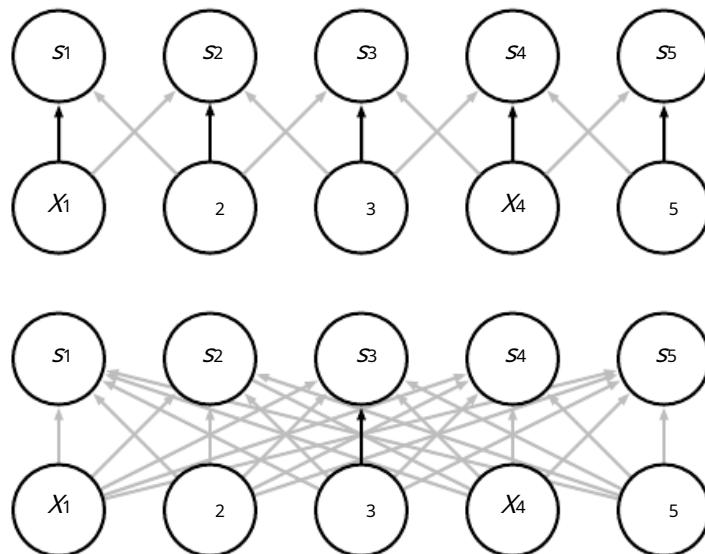


Figura 9.5: Parámetro compartido: Las flechas negras indican las conexiones que usan un parámetro en particular en dos modelos diferentes.(Arriba)Las flechas negras indican usos del elemento central de un kernel de 3 elementos en un modelo convolucional. Debido al uso compartido de parámetros, este único parámetro se utiliza en todas las ubicaciones de entrada.(Abajo)La única flecha negra indica el uso del elemento central de la matriz de pesos en un modelo completamente conectado. Este modelo no tiene parámetros compartidos, por lo que el parámetro se usa solo una vez.

para cada ubicación, aprendemos solo un conjunto. Esto no afecta el tiempo de ejecución de la propagación hacia adelante, todavía es $O(k \times norte)$, pero reduce aún más los requisitos de almacenamiento del modelo para k parámetros. Recordar que k suele ser varios órdenes de magnitud menor que $metro$. Desde $metro \times norte$ por lo general son aproximadamente del mismo tamaño, k es prácticamente insignificante en comparación con $metro \times norte$. La convolución es, por lo tanto, dramáticamente más eficiente que la multiplicación de matrices densas en términos de requisitos de memoria y eficiencia estadística. Para ver una representación gráfica de cómo funciona el uso compartido de parámetros, consulte la figura 9.5.

Como ejemplo de estos dos primeros principios en acción, figura 9.6 muestra cómo la escasa conectividad y el uso compartido de parámetros pueden mejorar drásticamente la eficiencia de una función lineal para detectar bordes en una imagen.

En el caso de la convolución, la forma particular de compartir parámetros hace que la capa tenga una propiedad llamada **equivalencia** a la traducción. Decir que una función es equivariante significa que si la entrada cambia, la salida cambia de la misma manera. En concreto, una función $F(X)$ es equivalente a una función $gramo$ si $F(gramo(X)) = gramo(F(X))$. En el caso de la convolución, si dejamos $gramo$ sea cualquier función que traduzca la entrada, es decir, la desplace, entonces la función de convolución es equivalente a $gramo$. Por ejemplo, deje I sea una función que proporcione el brillo de la imagen en coordenadas enteras. Dejar $gramo$ sea una función

mapear una función de imagen a otra función de imagen, tal que $I = \text{gramo}(I)$ es la función imagen con $I(x,y) = I(x-1,y)$. Esto cambia cada píxel de una unidad a la derecha. Si aplicamos esta transformación a I , luego aplicamos convolución, el resultado será el mismo que si aplicáramos convolución a I , luego aplicó la transformación gramo a la salida. Al procesar datos de series de tiempo, esto significa que la convolución produce una especie de línea de tiempo que muestra cuándo aparecen diferentes características en la entrada. Si movemos un evento más adelante en el tiempo en la entrada, exactamente la misma representación aparecerá en la salida, solo que más tarde en el tiempo. De manera similar con las imágenes, la convolución crea un mapa 2-D de dónde aparecen ciertas características en la entrada. Si movemos el objeto en la entrada, su representación se moverá lo mismo en la salida. Esto es útil cuando sabemos que alguna función de una pequeña cantidad de píxeles vecinos es útil cuando se aplica a múltiples ubicaciones de entrada. Por ejemplo, al procesar imágenes, es útil detectar bordes en la primera capa de una red convolucional. Los mismos bordes aparecen más o menos en todas partes de la imagen, por lo que es práctico compartir parámetros en toda la imagen. En algunos casos, es posible que no deseemos compartir parámetros en toda la imagen. Por ejemplo, si estamos procesando imágenes que se recortan para centrarlas en la cara de una persona, probablemente queramos extraer diferentes características en diferentes ubicaciones: la parte de la red que procesa la parte superior de la cara debe buscar las cejas, mientras que la parte de la red que procesa la parte inferior de la cara necesita buscar un mentón.

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

Finalmente, algunos tipos de datos no pueden ser procesados por redes neuronales definidas por multiplicación de matrices con una matriz de forma fija. La convolución permite el procesamiento de algunos de estos tipos de datos. Discutimos esto más adelante en la sección 9.7.

9.3 Agrupación

Una capa típica de una red convolucional consta de tres etapas (ver figura 9.7). En la primera etapa, la capa realiza varias convoluciones en paralelo para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación lineal pasa por una función de activación no lineal, como la función de activación lineal rectificada. Esta etapa a veces se denomina **detectores** escenario. En la tercera etapa, utilizamos un **función de agrupación** para modificar aún más la salida de la capa.

Una función de agrupación reemplaza la salida de la red en un lugar determinado con una estadística de resumen de las salidas cercanas. por ejemplo, la **agrupación máxima** (Zhou

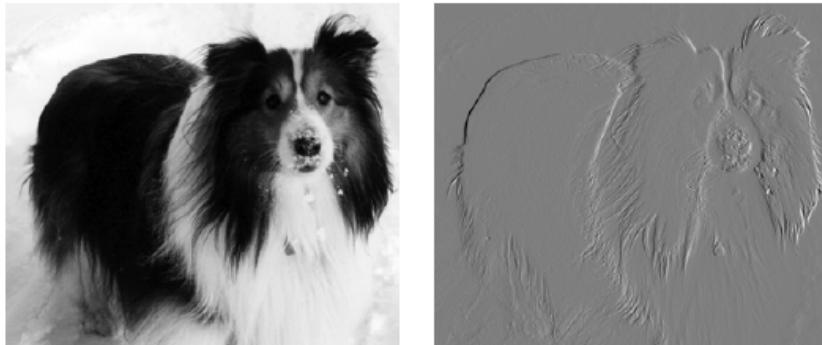


Figura 9.6:*Eficiencia de la detección de bordes*. La imagen de la derecha se formó tomando cada píxel de la imagen original y restando el valor de su píxel vecino a la izquierda. Esto muestra la fuerza de todos los bordes orientados verticalmente en la imagen de entrada, lo que puede ser una operación útil para la detección de objetos. Ambas imágenes tienen 280 píxeles de alto. La imagen de entrada tiene 320 píxeles de ancho, mientras que la imagen de salida tiene 319 píxeles de ancho. Esta transformación se puede describir mediante un núcleo de convolución que contiene dos elementos y requiere $319 \times 280 \times 3 = 267,960$ fl operaciones de punto flotante (dos multiplicaciones y una suma por píxel de salida) para calcular usando convolución. Para describir la misma transformación con una multiplicación de matrices se necesitaría $320 \times 280 \times 319 \times 280$, o más de ocho mil millones de entradas en la matriz, lo que hace que la convolución sea cuatro mil millones de veces más eficiente para representar esta transformación. El sencillo algoritmo de multiplicación de matrices realiza más de diecisésis mil millones de operaciones de punto flotante, lo que hace que la convolución sea aproximadamente 60 000 veces más eficiente desde el punto de vista computacional. Por supuesto, la mayoría de las entradas de la matriz serían cero. Si almacenamos solo las entradas distintas de cero de la matriz, tanto la multiplicación de matrices como la convolución requerirían el mismo número de operaciones de coma flotante para calcular. La matriz aún necesitaría contener $2 \times 319 \times 280 = 178,640$ entradas. La convolución es una forma extremadamente eficiente de describir transformaciones que aplican la misma transformación lineal de una pequeña región local en toda la entrada. (Crédito de la foto: Paula Goodfellow)

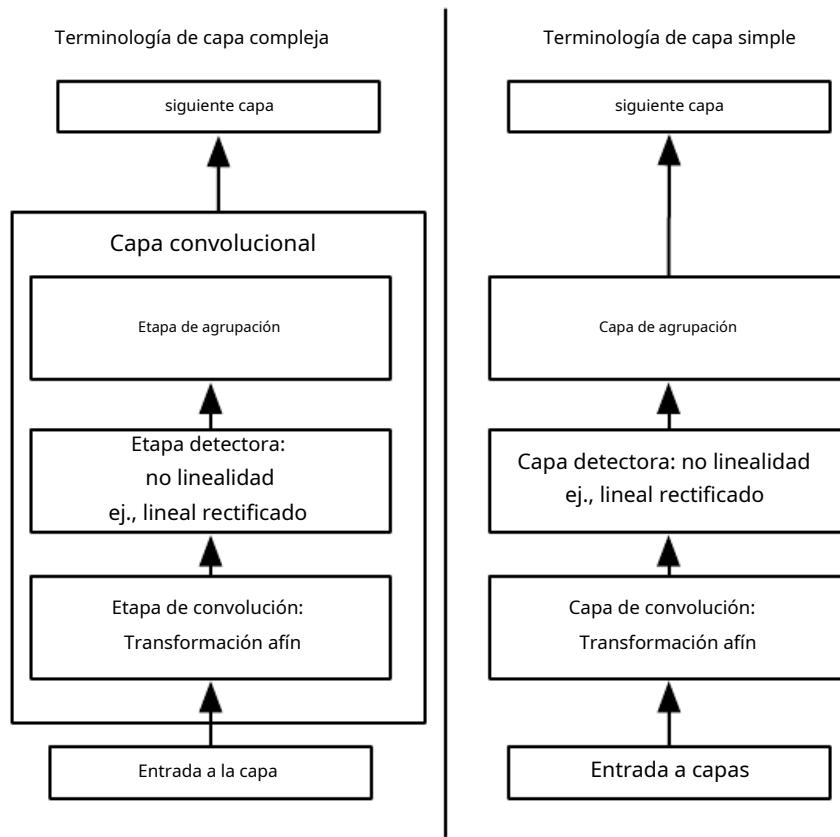


Figura 9.7: Los componentes de una capa de red neuronal convolucional típica. Hay dos conjuntos de terminología de uso común para describir estas capas. (*Izquierda*) En esta terminología, la red convolucional se ve como un pequeño número de capas relativamente complejas, cada una de las cuales tiene muchas "etapas". En esta terminología, existe un mapeo uno a uno entre los tensores del núcleo y las capas de red. En este libro generalmente usamos esta terminología. (*Derecha*) En esta terminología, la red convolucional se ve como un mayor número de capas simples; cada paso del procesamiento se considera como una capa por derecho propio. Esto significa que no todas las "capas" tienen parámetros.

y Chellappa, 1988) la operación informa la salida máxima dentro de una vecindad rectangular. Otras funciones populares de agrupación incluyen el promedio de un vecindario rectangular, la L_2 norma de un vecindario rectangular, o un promedio ponderado basado en la distancia desde el píxel central.

En todos los casos, la puesta en común ayuda a que la representación sea aproximadamente **invariante** a pequeñas traducciones de la entrada. La invariancia a la traducción significa que si traducimos la entrada en una pequeña cantidad, los valores de la mayoría de las salidas agrupadas no cambian. Ver figura 9.8 para ver un ejemplo de cómo funciona esto. *La invariancia a la traducción local puede ser una propiedad muy útil si nos importa más si alguna característica está presente que dónde está exactamente.* Por ejemplo, al determinar si una imagen contiene una cara, no necesitamos saber la ubicación de los ojos con una precisión perfecta de píxeles, solo necesitamos saber que hay un ojo en el lado izquierdo de la cara y un ojo en el lado derecho. lado de la cara. En otros contextos, es más importante preservar la ubicación de una entidad. Por ejemplo, si queremos encontrar una esquina definida por dos bordes que se encuentran en una orientación específica, debemos conservar la ubicación de los bordes lo suficientemente bien como para probar si se encuentran.

El uso de la agrupación puede verse como la adición de un previo infinitamente fuerte de que la función que aprende la capa debe ser invariable a las pequeñas traducciones. Cuando esta suposición es correcta, puede mejorar en gran medida la eficiencia estadística de la red.

La agrupación de regiones espaciales produce invariancia a la traducción, pero si agrupamos las salidas de convoluciones parametrizadas por separado, las características pueden aprender a qué transformaciones se vuelven invariantes (consulte la figura 9.9).

Debido a que la agrupación resume las respuestas de todo un vecindario, es posible usar menos unidades de agrupación que unidades detectoras, informando estadísticas de resumen para regiones de agrupación espaciadas k píxeles de distancia en lugar de 1 píxel de distancia. Ver figura 9.10 para un ejemplo. Esto mejora la eficiencia computacional de la red porque la siguiente capa tiene aproximadamente k veces menos entradas para procesar. Cuando el número de parámetros en la siguiente capa es una función de su tamaño de entrada (como cuando la siguiente capa está completamente conectada y se basa en la multiplicación de matrices), esta reducción en el tamaño de entrada también puede resultar en una eficiencia estadística mejorada y requisitos de memoria reducidos para almacenar los parámetros.

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

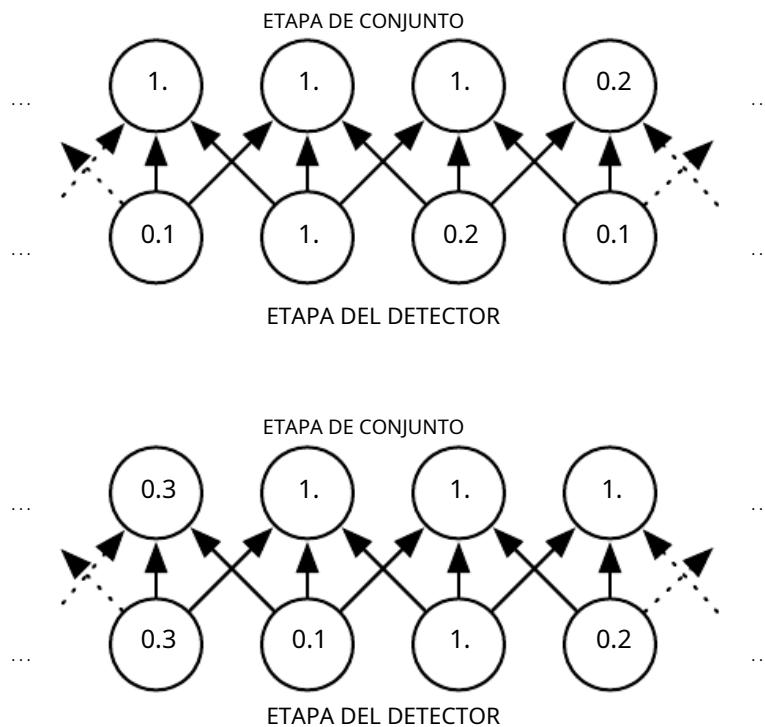


Figura 9.8: Max pooling introduce invariancia.(Arriba)Una vista de la mitad de la salida de una capa convolucional. La fila inferior muestra las salidas de la no linealidad. La fila superior muestra los resultados de la agrupación máxima, con un paso de un píxel entre las regiones de agrupación y un ancho de región de agrupación de tres píxeles.(Abajo)Una vista de la misma red, después de que la entrada se haya desplazado un píxel a la derecha. Todos los valores de la fila inferior han cambiado, pero solo la mitad de los valores de la fila superior han cambiado, porque las unidades de agrupación máximas solo son sensibles al valor máximo en la vecindad, no a su ubicación exacta.

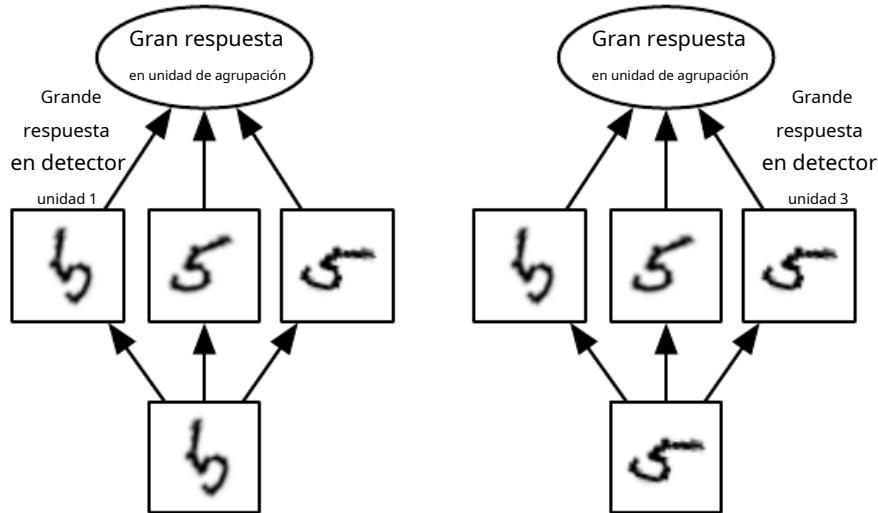


Figura 9.9: Ejemplo de invariancias aprendidas: Una unidad de agrupación que agrupa varias características que se aprenden con parámetros separados puede aprender a ser invariable a las transformaciones de la entrada. Aquí mostramos cómo un conjunto de tres filtros aprendidos y una unidad de agrupación máxima pueden aprender a volverse invariantes a la rotación. Los tres filtros están destinados a detectar un 5 escrito a mano. Cada filtro intenta hacer coincidir una orientación ligeramente diferente del 5. Cuando aparece un 5 en la entrada, el filtro correspondiente lo hará coincidir y provocará una gran activación en una unidad detectora. Entonces, la unidad de agrupación máxima tiene una gran activación, independientemente de qué unidad detectora se haya activado. Mostramos aquí cómo la red procesa dos entradas diferentes, lo que resulta en la activación de dos unidades detectoras diferentes. El efecto en la unidad de agrupación es aproximadamente el mismo en ambos sentidos. Este principio es aprovechado por las redes maxout ([Buen compañero et al., 2013a](#)) y otras redes convolucionales. La agrupación máxima sobre posiciones espaciales es naturalmente invariable a la traducción; este enfoque multicanal solo es necesario para aprender otras transformaciones.

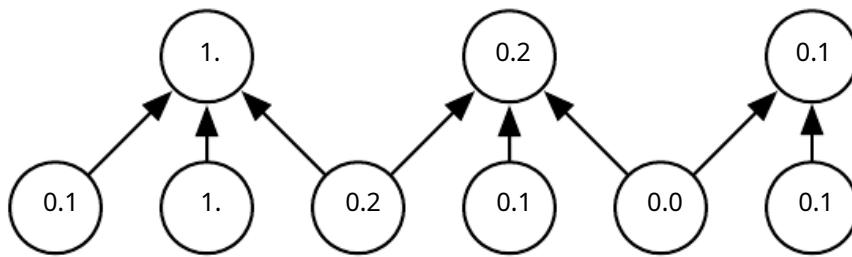


Figura 9.10: Agrupación con reducción de resolución. Aquí usamos max-pooling con un ancho de grupo de tres y una zancada entre grupos de dos. Esto reduce el tamaño de la representación por un factor de dos, lo que reduce la carga computacional y estadística en la siguiente capa. Tenga en cuenta que la región de agrupación más a la derecha tiene un tamaño más pequeño, pero debe incluirse si no queremos ignorar algunas de las unidades del detector.

Algunos trabajos teóricos brindan orientación sobre qué tipos de agrupación se deben usar en diversas situaciones (Boureau et al., 2010). También es posible agrupar características de forma dinámica, por ejemplo, mediante la ejecución de un algoritmo de agrupamiento en las ubicaciones de características interesantes (Boureau et al., 2011). Este enfoque produce un conjunto diferente de regiones de agrupación para cada imagen. Otro enfoque es aprender una única estructura de agrupación que luego se aplica a todas las imágenes (Jia et al., 2012).

La agrupación puede complicar algunos tipos de arquitecturas de redes neuronales que utilizan información de arriba hacia abajo, como las máquinas Boltzmann y los codificadores automáticos. Estos temas se discutirán más adelante cuando presentemos estos tipos de redes en parte **tercero**. La agrupación en máquinas convolucionales de Boltzmann se presenta en la sección **20.6**. Las operaciones de tipo inverso en unidades de agrupación necesarias en algunas redes diferenciables se cubrirán en la sección **20.10.6**.

En la figura se muestran algunos ejemplos de arquitecturas de red convolucionales completas para la clasificación mediante convolución y agrupación **9.11**.

9.4 Convolución y agrupación como un previo infinitamente fuerte

Recuerda el concepto de **distribución de probabilidad previa** de la sección **5.2**. Esta es una distribución de probabilidad sobre los parámetros de un modelo que codifica nuestras creencias sobre qué modelos son razonables, antes de que hayamos visto ningún dato.

Los priores pueden considerarse débiles o fuertes dependiendo de qué tan concentrada esté la densidad de probabilidad en el prior. Una distribución previa débil es una distribución previa con alta entropía, como una distribución gaussiana con alta varianza. Tal prior permite que los datos muevan los parámetros más o menos libremente. Un previo fuerte tiene una entropía muy baja, como una distribución gaussiana con poca varianza. Tal prior juega un papel más activo en la determinación de dónde terminan los parámetros.

Un prior infinitamente fuerte coloca una probabilidad cero en algunos parámetros y dice que estos valores de parámetros están completamente prohibidos, independientemente de cuánto apoyo brinden los datos a esos valores.

Podemos imaginar una red convolucional similar a una red completamente conectada, pero con un anterior infinitamente fuerte sobre sus pesos. Este prior infinitamente fuerte dice que los pesos de una unidad oculta deben ser idénticos a los pesos de su vecino, pero desplazados en el espacio. El anterior también dice que los pesos deben ser cero, excepto en el pequeño campo receptivo espacialmente contiguo asignado a esa unidad oculta. En general, podemos pensar en el uso de la convolución como la introducción de una distribución de probabilidad previa infinitamente fuerte sobre los parámetros de una capa. este previo

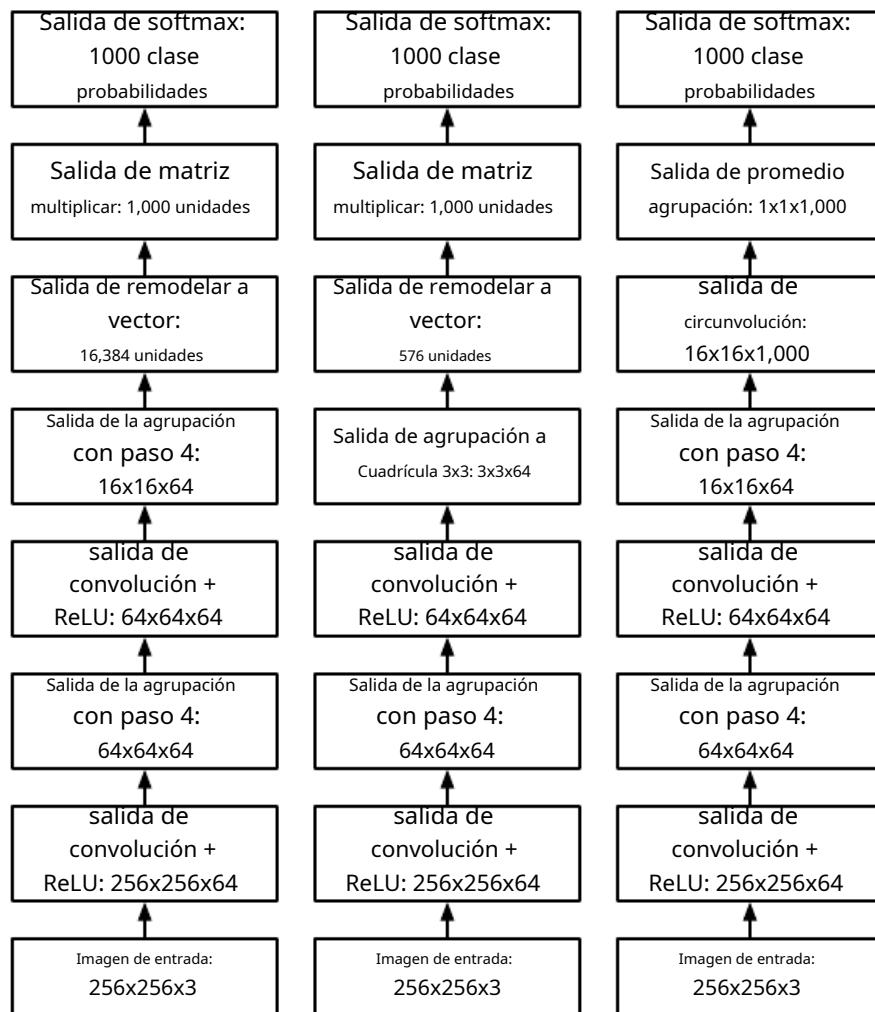


Figura 9.11: Ejemplos de arquitecturas para clasificación con redes convolucionales. Los pasos y profundidades específicos utilizados en esta figura no son recomendables para un uso real; están diseñados para ser muy superficiales a fin de caber en la página. Las redes convolucionales reales también suelen implicar cantidades significativas de ramificación, a diferencia de las estructuras de cadena utilizadas aquí por simplicidad. *(Izquierda)* Una red convolucional que procesa un tamaño de imagen fijo. Después de alternar entre convolución y agrupación en algunas capas, el tensor del mapa de características convolucionales se reforma para aplanar las dimensiones espaciales. El resto de la red es un clasificador de red feedforward ordinario, como se describe en el capítulo 6. *(Centro)* Una red convolucional que procesa una imagen de tamaño variable, pero aún mantiene una sección completamente conectada. Esta red utiliza una operación de agrupación con grupos de tamaño variable, pero con un número fijo de grupos, para proporcionar un vector de tamaño fijo de 576 unidades a la parte completamente conectada de la red. *(Derecha)* Una red convolucional que no tiene ninguna capa de peso completamente conectada. En cambio, la última capa convolucional genera un mapa de características por clase. Presumiblemente, el modelo aprende un mapa de la probabilidad de que ocurra cada clase en cada ubicación espacial. Promediar un mapa de características hasta un solo valor proporciona el argumento para el clasificador softmax en la parte superior.

dice que la función que la capa debe aprender contiene solo interacciones locales y es equivalente a la traducción. Asimismo, el uso de la puesta en común es un principio infinitamente fuerte que cada unidad debe ser invariante a pequeñas traslaciones.

Por supuesto, implementar una red convolucional como una red completamente conectada con un previo infinitamente fuerte sería un desperdicio computacional extremo. Pero pensar en una red convolucional como una red completamente conectada con un previo infinitamente fuerte puede darnos una idea de cómo funcionan las redes convolucionales.

Una idea clave es que la convolución y la agrupación pueden causar un ajuste insuficiente. Como cualquier anterior, la convolución y la agrupación solo son útiles cuando las suposiciones realizadas por el anterior son razonablemente precisas. Si una tarea se basa en conservar información espacial precisa, el uso de la agrupación en todas las entidades puede aumentar el error de entrenamiento. Algunas arquitecturas de red convolucional ([Szegedy et al., 2014a](#)) están diseñados para usar la agrupación en algunos canales pero no en otros canales, con el fin de obtener características altamente invariantes y características que no se ajusten mal cuando la invariancia de traducción previa es incorrecta. Cuando una tarea implica incorporar información de ubicaciones muy distantes en la entrada, entonces el previo impuesto por la convolución puede ser inapropiado.

Otra idea clave de este punto de vista es que solo debemos comparar modelos convolucionales con otros modelos convolucionales en los puntos de referencia del rendimiento del aprendizaje estadístico. Los modelos que no usan convolución podrían aprender incluso si permutáramos todos los píxeles de la imagen. Para muchos conjuntos de datos de imágenes, existen puntos de referencia separados para modelos que son **permutación invariantes** y deben descubrir el concepto de topología a través del aprendizaje y los modelos que tienen el conocimiento de las relaciones espaciales codificado por su diseñador.

9.5 Variantes de la Función de Convolución Básica

Cuando se analiza la convolución en el contexto de las redes neuronales, por lo general no nos referimos exactamente a la operación de convolución discreta estándar tal como se entiende generalmente en la literatura matemática. Las funciones utilizadas en la práctica difieren ligeramente. Aquí describimos estas diferencias en detalle y destacamos algunas propiedades útiles de las funciones utilizadas en las redes neuronales.

En primer lugar, cuando nos referimos a la convolución en el contexto de las redes neuronales, en realidad nos referimos a una operación que consta de muchas aplicaciones de convolución en paralelo. Esto se debe a que la convolución con un solo kernel solo puede extraer un tipo de característica, aunque en muchas ubicaciones espaciales. Por lo general, queremos que cada capa de nuestra red extraiga muchos tipos de características, en muchos lugares.

Además, la entrada no suele ser solo una cuadrícula de valores reales. Más bien, es una cuadrícula de observaciones con valores vectoriales. Por ejemplo, una imagen en color tiene una intensidad de rojo, verde y azul en cada píxel. En una red convolucional multicapa, la entrada a la segunda capa es la salida de la primera capa, que normalmente tiene la salida de muchas convoluciones diferentes en cada posición. Cuando trabajamos con imágenes, generalmente pensamos en la entrada y la salida de la convolución como tensores tridimensionales, con un índice en los diferentes canales y dos índices en las coordenadas espaciales de cada canal. Las implementaciones de software generalmente funcionan en modo por lotes, por lo que en realidad usarán tensores 4-D, con el cuarto eje indexando diferentes ejemplos en el lote, pero omitiremos el eje del lote en nuestra descripción aquí para simplificar.

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

Supongamos que tenemos un tensor kernel 4-D \mathbf{k} con elemento $k_{yo, j, k}$, dando la fuerza de conexión entre una unidad en el canal j de la salida y una unidad en el canal k de la entrada, con un desplazamiento de k filas y j columnas entre la unidad de salida y la unidad de entrada. Supongamos que nuestra entrada consiste en datos observados \mathbf{V} con elemento $V_{yo, j, k}$, dando el valor de la unidad de entrada dentro del canal j en fila y y columna k . Supongamos que nuestra salida consiste en \mathbf{Z} con el mismo formato que \mathbf{V} . Si \mathbf{Z} se produce por convolución \mathbf{k} al otro lado de \mathbf{V} sin voltear \mathbf{k} , entonces

$$Z_{yo, j, k} = \sum_{yo, m, n} V_{yo+m-1, j+n-1} k_{yo, j, k} \quad (9.7)$$

donde termina la suma $yo, metro, norte$ es sobre todos los valores para los que son válidas las operaciones de indexación del tensor dentro de la sumatoria. En notación de álgebra lineal, indexamos en arreglos usando un 1 para la primera entrada. Esto requiere la -1 en la fórmula anterior. Lenguajes de programación como C y el índice de Python a partir de 0, simplificando aún más la expresión anterior.

Es posible que deseemos omitir algunas posiciones del kernel para reducir el costo computacional (a expensas de no extraer nuestras funciones con tanta precisión). Podemos pensar en esto como una reducción de la muestra de la salida de la función de convolución completa. Si queremos muestrear sólo cada píxeles en cada dirección en la salida, entonces podemos definir una función de convolución submuestreada C tal que

$$Z_{yo, j, k} = C(\mathbf{k}, \mathbf{V}, s)_{yo, j, k} = \sum_{yo, m, n} V_{yo+(j-1)s, m+metro, (k-1)s+norte} k_{yo, j, k} \quad (9.8)$$

Nos referimos a como el **pasode** esta convolución submuestreada. También es posible

para definir un paso separado para cada dirección de movimiento. Ver figura 9.12 para una ilustración.

Una característica esencial de cualquier implementación de red convolucional es la capacidad de llenar con ceros implícitamente la entrada para hacerla más ancha. Sin esta función, el ancho de la representación se reduce en un píxel menos que el ancho del kernel en cada capa. El relleno cero de la entrada nos permite controlar el ancho del kernel y el tamaño de la salida de forma independiente. Sin relleno cero, nos vemos obligados a elegir entre reducir rápidamente la extensión espacial de la red y usar kernels pequeños, ambos escenarios que limitan significativamente el poder expresivo de la red. Ver figura 9.13 para un ejemplo.

Vale la pena mencionar tres casos especiales de la configuración de relleno cero. Uno es el caso extremo en el que no se utiliza ningún tipo de relleno con ceros y el kernel de convolución solo puede visitar posiciones en las que todo el kernel está contenido completamente dentro de la imagen. En la terminología de MATLAB, esto se llama **válida** convolución. En este caso, todos los píxeles de la salida son función del mismo número de píxeles de la entrada, por lo que el comportamiento de un píxel de salida es algo más regular. Sin embargo, el tamaño de la salida se reduce en cada capa. Si la imagen de entrada tiene ancho m y el kernel tiene ancho k , la salida será de ancho $m - k + 1$. La tasa de este encogimiento puede ser dramática si los granos utilizados son grandes. Dado que la contracción es mayor que 0, limita el número de capas convolucionales que se pueden incluir en la red. A medida que se agregan capas, la dimensión espacial de la red finalmente se reducirá a 1×1 , momento en el que las capas adicionales no pueden considerarse significativamente convolucionales. Otro caso especial de la configuración de relleno cero es cuando se agrega suficiente relleno cero para mantener el tamaño de la salida igual al tamaño de la entrada. MATLAB llama a esto **mismo** convolución. En este caso, la red puede contener tantas capas convolucionales como el hardware disponible pueda soportar, ya que la operación de convolución no modifica las posibilidades arquitectónicas disponibles para la siguiente capa. Sin embargo, los píxeles de entrada cerca del borde influyen en menos píxeles de salida que los píxeles de entrada cerca del centro. Esto puede hacer que los píxeles del borde queden poco representados en el modelo. Esto motiva el otro caso extremo, al que MATLAB se refiere como **llenado** convolución, en la que se agregan suficientes ceros para que cada píxel sea visitado k veces en cada dirección, lo que da como resultado una imagen de salida de ancho $m + k - 1$. En este caso, los píxeles de salida cerca del borde son una función de menos píxeles que los píxeles de salida cerca del centro. Esto puede dificultar el aprendizaje de un solo kernel que funcione bien en todas las posiciones en el mapa de características convolucionales. Por lo general, la cantidad óptima de relleno cero (en términos de precisión de clasificación del conjunto de prueba) se encuentra en algún lugar entre la convolución "válida" y la "mismá".

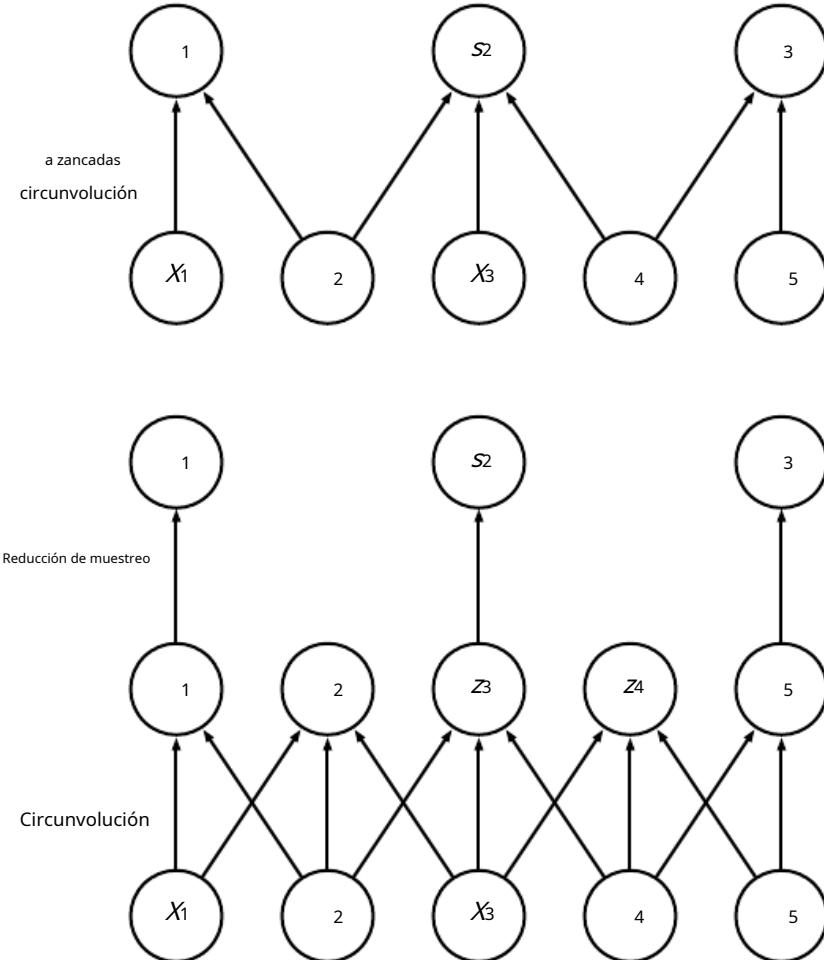


Figura 9.12: Circunvolución con zancada. En este ejemplo, usamos un paso de dos. (*Arriba*) Convolución con una longitud de zancada de dos implementada en una sola operación. (*Aabajo*) La convolución con una zancada superior a un píxel es matemáticamente equivalente a la convolución con una zancada unitaria seguida de reducción de resolución. Obviamente, el enfoque de dos pasos que involucra la reducción de muestreo es un desperdicio computacional, porque calcula muchos valores que luego se descartan.

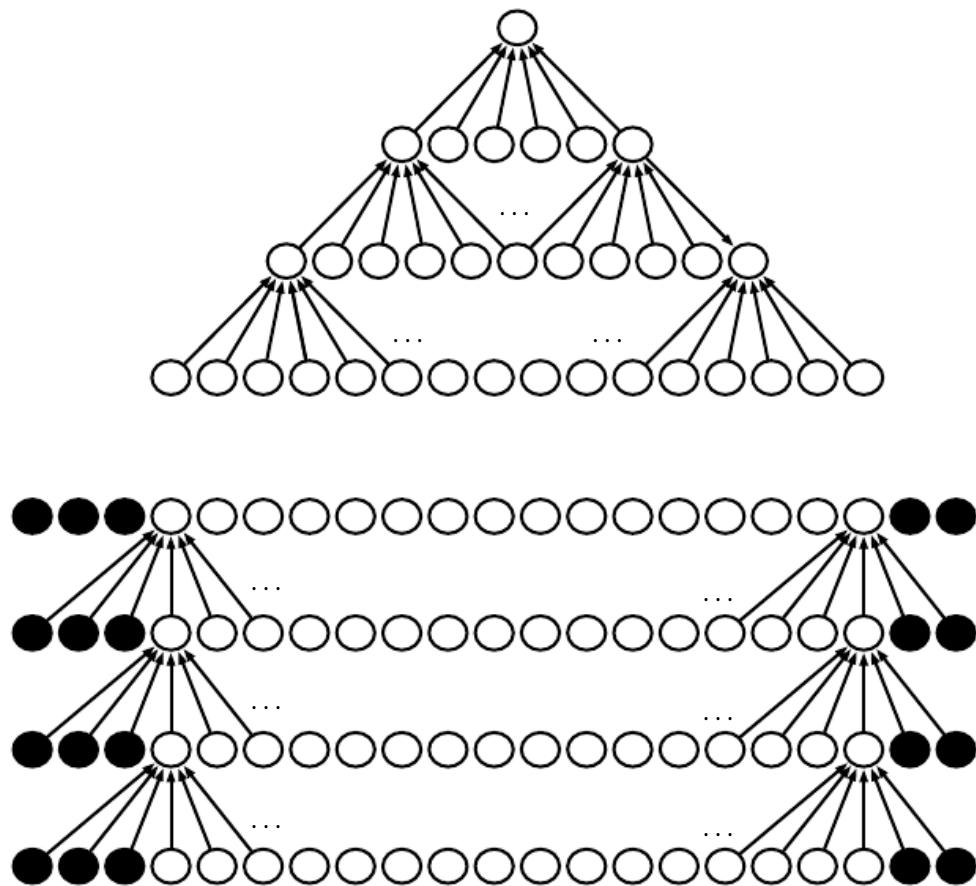


Figura 9.13:*El efecto del relleno cero en el tamaño de la red*. Considere una red convolucional con un kernel de ancho seis en cada capa. En este ejemplo, no usamos ninguna agrupación, por lo que solo la operación de convolución reduce el tamaño de la red.(Arriba)En esta red convolucional, no usamos ningún relleno de ceros implícito. Esto hace que la representación se reduzca en cinco píxeles en cada capa. A partir de una entrada de dieciséis píxeles, solo podemos tener tres capas convolucionales, y la última capa nunca mueve el núcleo, por lo que podría decirse que solo dos de las capas son realmente convolucionales. La tasa de reducción se puede mitigar mediante el uso de núcleos más pequeños, pero los núcleos más pequeños son menos expresivos y es inevitable cierta reducción en este tipo de arquitectura.(Abajo)Al agregar cinco ceros implícitos a cada capa, evitamos que la representación se reduzca con la profundidad. Esto nos permite hacer una red convolucional arbitrariamente profunda.

En algunos casos, en realidad no queremos usar convolución, sino capas conectadas localmente ([lecun, 1986, 1989](#)). En este caso, la matriz de adyacencia en el gráfico de nuestro MLP es la misma, pero cada conexión tiene su propio peso, especificado por un tensor 6-DW . Los índices en W son respectivamente: i , el canal de salida, j , la fila de salida, k , la columna de salida, yo , el canal de entrada, $metro$, el desplazamiento de fila dentro de la entrada, $ynorte$, el desplazamiento de columna dentro de la entrada. La parte lineal de una capa localmente conectada viene dada por

$$Z_{yo, j, k} = \sum_{yo, m, n} [V_{yo+m-1, k+n-1} W_{yo, j, k, l, m, n}]. \quad (9.9)$$

Esto a veces también se llama **convolución no compartida**, porque es una operación similar a la convolución discreta con un kernel pequeño, pero sin compartir parámetros entre ubicaciones. Cifra [9.14](#) compara conexiones locales, convolución y conexiones completas.

Las capas conectadas localmente son útiles cuando sabemos que cada característica debe ser una función de una pequeña parte del espacio, pero no hay motivo para pensar que la misma característica debe ocurrir en todo el espacio. Por ejemplo, si queremos saber si una imagen es la imagen de una cara, solo necesitamos buscar la boca en la mitad inferior de la imagen.

También puede ser útil hacer versiones de convolución o capas conectadas localmente en las que la conectividad se restringe aún más, por ejemplo, para restringir cada canal de salida. Ser una función de solo un subconjunto de los canales de entrada yo . Una forma común de hacer esto es hacer la primera $metro$ los canales de salida se conectan solo al primero $norte$ canales de entrada, el segundo $metro$ los canales de salida se conectan solo al segundo $norte$ canales de entrada, etc. Ver figura [9.15](#) para un ejemplo. El modelado de interacciones entre pocos canales permite que la red tenga menos parámetros para reducir el consumo de memoria y aumentar la eficiencia estadística, y también reduce la cantidad de cómputo necesaria para realizar la propagación hacia adelante y hacia atrás. Logra estos objetivos sin reducir el número de unidades ocultas.

Convolución en mosaico ([Gregor y Le Cun, 2010a; Le et al., 2010](#)) ofrece un compromiso entre una capa convolucional y una capa conectada localmente. En lugar de aprender un conjunto separado de pesos en cada ubicación espacial, aprendemos un conjunto de núcleos que rotamos a medida que nos movemos por el espacio. Esto significa que las ubicaciones inmediatamente vecinas tendrán diferentes filtros, como en una capa conectada localmente, pero los requisitos de memoria para almacenar los parámetros aumentarán solo por un factor del tamaño de este conjunto de núcleos, en lugar del tamaño de toda la función de salida. mapa. Ver figura [9.16](#) para una comparación de capas conectadas localmente, convolución en mosaico y convolución estándar.

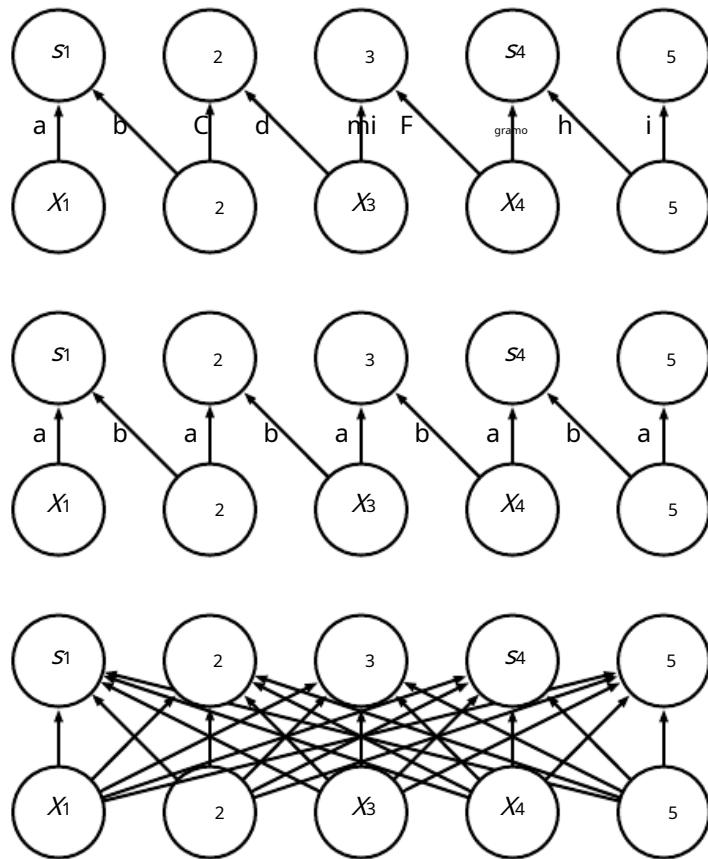


Figura 9.14: Comparación de conexiones locales, convolución y conexiones completas. (Arriba) Una capa conectada localmente con un tamaño de parche de dos píxeles. Cada borde está etiquetado con una letra única para mostrar que cada borde está asociado con su propio parámetro de peso. (Centro) Una capa convolucional con un ancho de núcleo de dos píxeles. Este modelo tiene exactamente la misma conectividad que la capa conectada localmente. La diferencia no radica en qué unidades interactúan entre sí, sino en cómo se comparten los parámetros. La capa conectada localmente no comparte parámetros. La capa convolucional usa los mismos dos pesos repetidamente en toda la entrada, como lo indica la repetición de las letras que etiquetan cada borde. (Abajo) Una capa completamente conectada se parece a una capa conectada localmente en el sentido de que cada borde tiene su propio parámetro (hay demasiados para etiquetarlos explícitamente con letras en este diagrama). Sin embargo, no tiene la conectividad restringida de la capa conectada localmente.

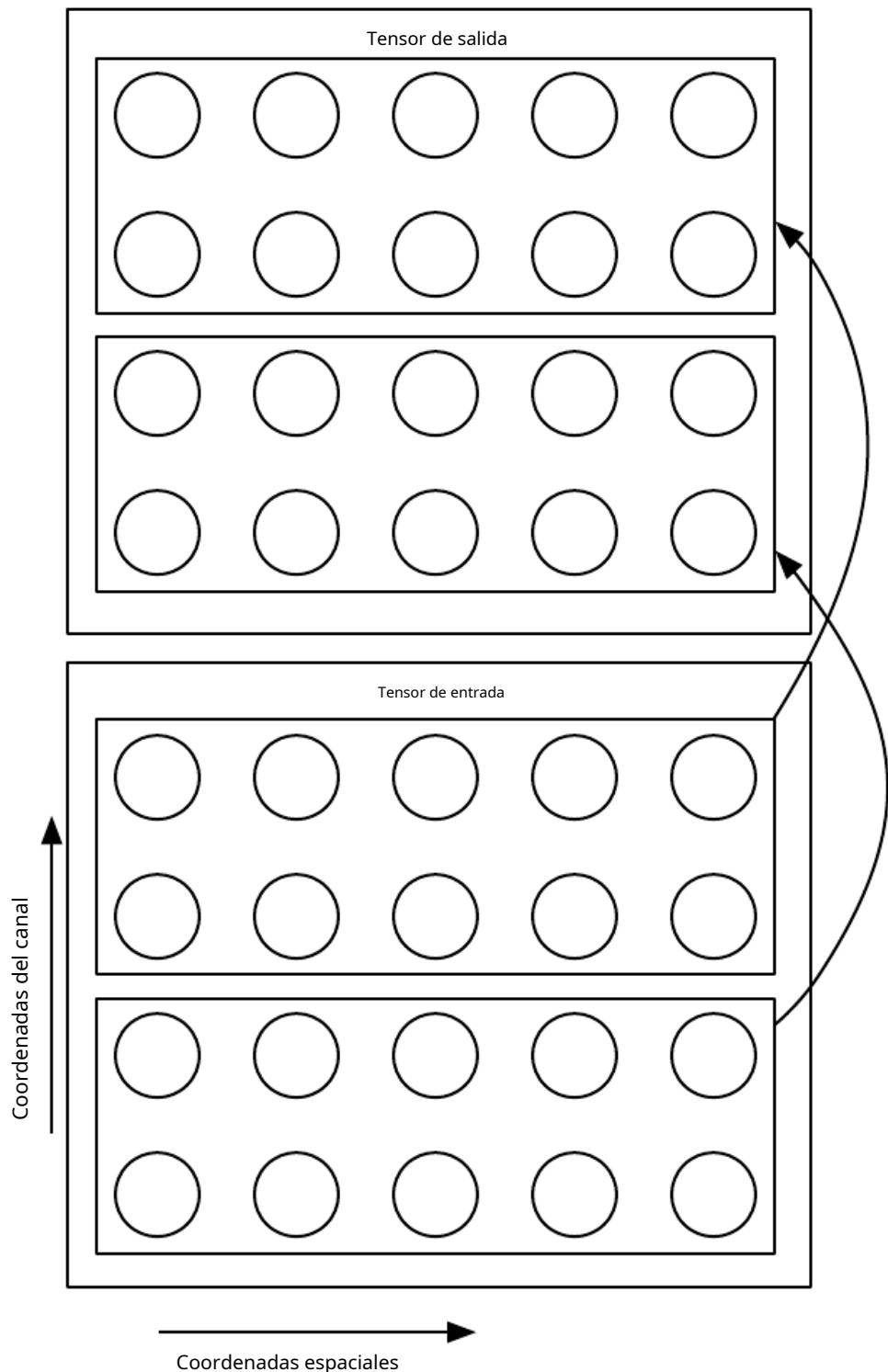


Figura 9.15: Una red convolucional con los primeros dos canales de salida conectados solo a los primeros dos canales de entrada, y los segundos dos canales de salida conectados solo a los segundos dos canales de entrada.

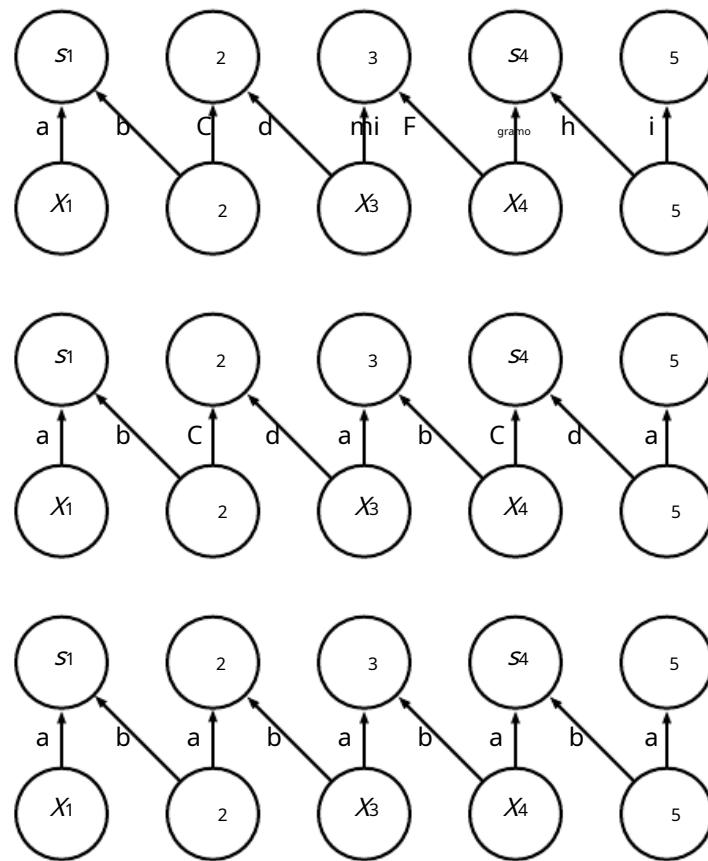


Figura 9.16: Una comparación de capas conectadas localmente, convolución en mosaico y convolución estándar. Los tres tienen los mismos conjuntos de conexiones entre unidades, cuando se usa el mismo tamaño de kernel. Este diagrama ilustra el uso de un kernel de dos píxeles de ancho. Las diferencias entre los métodos radican en cómo comparten parámetros.
 (Arriba) Una capa conectada localmente no se comparte en absoluto. Indicamos que cada conexión tiene su propio peso al etiquetar cada conexión con una letra única.
 (Centro) La convolución en mosaico tiene un conjunto de núcleos diferentes. Aquí ilustramos el caso de $t=2$. Uno de estos núcleos tiene bordes etiquetados como "a" y "b", mientras que el otro tiene bordes etiquetados como "c" y "d". Cada vez que movemos un píxel a la derecha en la salida, pasamos a usar un kernel diferente. Esto significa que, al igual que la capa conectada localmente, las unidades vecinas en la salida tienen parámetros diferentes. A diferencia de la capa conectada localmente, después de haber pasado por todos los kernels disponibles, volvemos al primer kernel. Si dos unidades de salida están separadas por un múltiplo de t pasos, luego comparten parámetros.
 (Abajo) La convolución tradicional es equivalente a la convolución en mosaico con $t=1$. Solo hay un kernel y se aplica en todas partes, como se indica en el diagrama usando el kernel con pesos etiquetados como "a" y "b" en todas partes.

Para definir la convolución en mosaico algebraicamente, sea k un tensor 6-D, donde dos de las dimensiones corresponden a diferentes ubicaciones en el mapa de salida. En lugar de tener un índice separado para cada ubicación en el mapa de salida, las ubicaciones de salida recorren un conjunto de diferentes opciones de pila de kernel en cada dirección. Si es igual al ancho de salida, esto es lo mismo que una capa conectada localmente.

$$z_{yo, j, k} = \sum_{yo, m, n} v_{yo+m-1, k+n-1, k, yo, l, m, n, \%t+1, \%t+1}, \quad (9.10)$$

donde $\%$ es la operación de módulo, con $t\%t=0, (t+1)\%t=1$, etc. Es sencillo generalizar esta ecuación para usar un rango de mosaico diferente para cada dimensión.

Tanto las capas conectadas localmente como las capas convolucionales en mosaico tienen una interacción interesante con la agrupación máxima: las unidades detectoras de estas capas están controladas por diferentes filtros. Si estos filtros aprenden a detectar diferentes versiones transformadas de las mismas características subyacentes, entonces las unidades agrupadas máximas se vuelven invariantes a la transformación aprendida (ver figura 9.9). Las capas convolucionales están codificadas para ser invariantes específicamente a la traducción.

Otras operaciones además de la convolución suelen ser necesarias para implementar una red convolucional. Para realizar el aprendizaje, uno debe poder calcular el gradiente con respecto al núcleo, dado el gradiente con respecto a las salidas. En algunos casos simples, esta operación se puede realizar mediante la operación de convolución, pero muchos casos de interés, incluido el caso de zancada mayor que 1, no tienen esta propiedad.

Recuerde que la convolución es una operación lineal y, por lo tanto, puede describirse como una multiplicación de matrices (si primero transformamos el tensor de entrada en un vector plano). La matriz involucrada es una función del kernel de convolución. La matriz es escasa y cada elemento del kernel se copia en varios elementos de la matriz. Esta vista nos ayuda a derivar algunas de las otras operaciones necesarias para implementar una red convolucional.

La multiplicación por la transpuesta de la matriz definida por convolución es una de esas operaciones. Esta es la operación necesaria para propagar hacia atrás las derivadas de errores a través de una capa convolucional, por lo que es necesaria para entrenar redes convolucionales que tengan más de una capa oculta. Esta misma operación también es necesaria si deseamos reconstruir las unidades visibles a partir de las unidades ocultas ([Simard et al., 1992](#)). La reconstrucción de las unidades visibles es una operación comúnmente utilizada en los modelos descritos en parte [tercero](#) de este libro, como codificadores automáticos, RBM y codificación dispersa. La transposición de convolución es necesaria para construir versiones convolucionales de esos modelos. Al igual que la operación de gradiente del núcleo, esta operación de gradiente de entrada se puede

implementado usando una convolución en algunos casos, pero en el caso general requiere que se implemente una tercera operación. Se debe tener cuidado para coordinar esta operación de transposición con la propagación directa. El tamaño de la salida que debe devolver la operación de transposición depende de la política de relleno cero y del paso de la operación de propagación hacia adelante, así como del tamaño del mapa de salida de la propagación hacia adelante. En algunos casos, varios tamaños de entrada para la propagación hacia adelante pueden dar como resultado el mismo tamaño del mapa de salida, por lo que la operación de transposición debe indicarse explícitamente cuál era el tamaño de la entrada original.

Estas tres operaciones (convolución, backprop de salida a pesos y backprop de salida a entradas) son suficientes para calcular todos los gradientes necesarios para entrenar cualquier profundidad de red convolucional feedforward, así como para entrenar redes convolucionales con funciones de reconstrucción basadas en el transposición de convolución. Ver [Buen compaño\(2010\)](#) para obtener una derivación completa de las ecuaciones en el caso de varios ejemplos, multidimensional y completamente general. Para dar una idea de cómo funcionan estas ecuaciones, presentamos aquí la versión bidimensional de ejemplo único.

Supongamos que queremos entrenar una red convolucional que incorpore la convolución estriada de la pila del kernel \mathbf{k} aplicado a la imagen multicanal \mathbf{V} con pasos s . Según lo definido por $C(\mathbf{k}, \mathbf{V}, s)$ como en ecuación 9.8. Supongamos que queremos minimizar alguna función de pérdida $J(\mathbf{V}, \mathbf{k})$. Durante la propagación hacia adelante, necesitaremos usar C sí mismo a la salida \mathbf{Z} , que luego se propaga por el resto de la red y se usa para calcular la función de costo J . Durante la retropropagación, recibiremos un tensor **GRAMO** tal que $GRAMO_{yo, j, k} = \frac{\partial}{\partial z_{yo, j, k}} J(\mathbf{V}, \mathbf{k})$.

Para entrenar la red, necesitamos calcular las derivadas con respecto a los pesos en el núcleo. Para ello, podemos utilizar una función

$$gramo(\mathbf{GRAMO}, \mathbf{V}, s)_{yo, j, k} = \frac{\partial}{\partial k_{yo, j, k}} J(\mathbf{V}, \mathbf{k}) = GRAMO_{yo, m, n} V_{j, (m-1) \times s + k, (n-1) \times s + yo}. \quad (9.11)$$

Minnesota

Si esta capa no es la capa inferior de la red, necesitaremos calcular el gradiente con respecto a \mathbf{V} para propagar hacia atrás el error más abajo. Para ello, podemos utilizar una función

$$h(\mathbf{k}, \mathbf{GRAMO}, s)_{yo, j, k} = \frac{\partial}{\partial V_{yo, j, k}} J(\mathbf{V}, \mathbf{k}) \quad (9.12)$$

$$= \begin{matrix} yo \\ calle \\ (yo-1) \times s + metro = j \end{matrix} \quad \begin{matrix} notario público \\ calle \\ (n-1) \times s + pag = k \end{matrix} \quad q \\ k_{q, i, m, p} GRAMO_{q, l, n}. \quad (9.13)$$

Redes de autoencoder, descritas en el capítulo 14, son redes feedforward entrenadas para copiar su entrada a su salida. Un ejemplo simple es el algoritmo PCA,

que copia su entrada x a una reconstrucción aproximada usando la función $W\text{-Ancho } x$. Es común que los codificadores automáticos más generales usen la multiplicación por la transposición de la matriz de peso tal como lo hace PCA. Para hacer tales modelos convolucionales, podemos usar la función h para realizar la transposición de la operación de convolución. Supongamos que tenemos unidades ocultas \mathbf{h} en el mismo formato que \mathbf{z} y definimos una reconstrucción

$$\mathbf{R} = h(\mathbf{k}, \mathbf{H}, s). \quad (9.14)$$

Para entrenar el autoencoder, recibiremos el gradiente con respecto a \mathbf{R} como tensor \mathbf{m}_i . Para entrenar el decodificador, necesitamos obtener el gradiente con respecto a \mathbf{k} . Esto está dado por $\text{gramo}(\mathbf{H}, \mathbf{m}_i, s)$. Para entrenar el codificador, necesitamos obtener el gradiente con respecto a \mathbf{H} . Esto está dado por $\text{C}(\mathbf{k}, \mathbf{m}_i, s)$. También es posible diferenciar a través de gramo usando \mathcal{C}_h , pero estas operaciones no son necesarias para el algoritmo de propagación hacia atrás en ninguna arquitectura de red estándar.

Generalmente, no usamos solo una operación lineal para transformar las entradas a las salidas en una capa convolucional. Por lo general, también agregamos algún término de sesgo a cada salida antes de aplicar la no linealidad. Esto plantea la cuestión de cómo compartir parámetros entre los sesgos. Para las capas conectadas localmente, es natural dar a cada unidad su propio sesgo, y para la convolución en mosaico, es natural compartir los sesgos con el mismo patrón de mosaico que los núcleos. Para las capas convolucionales, es típico tener un sesgo por canal de salida y compartirlo en todas las ubicaciones dentro de cada mapa de convolución. Sin embargo, si la entrada es de tamaño fijo conocido, también es posible aprender un sesgo separado en cada ubicación del mapa de salida. La separación de los sesgos puede reducir ligeramente la eficiencia estadística del modelo, pero también permite que el modelo corrija las diferencias en las estadísticas de la imagen en diferentes ubicaciones. Por ejemplo, cuando se utiliza un relleno de ceros implícito, las unidades detectoras en el borde de la imagen reciben menos entrada total y pueden necesitar sesgos más grandes.

9.6 Salidas estructuradas

Las redes convolucionales se pueden usar para generar un objeto estructurado de alta dimensión, en lugar de simplemente predecir una etiqueta de clase para una tarea de clasificación o un valor real para una tarea de regresión. Por lo general, este objeto es solo un tensor, emitido por una capa convolucional estándar. Por ejemplo, el modelo podría emitir un tensor \mathbf{S} , donde $S_{y, j, k}$ es la probabilidad de que el píxel (j, k) de la entrada a la red pertenece a la clase i . Esto permite que el modelo etiquete cada píxel de una imagen y dibuje máscaras precisas que sigan los contornos de los objetos individuales.

Un problema que suele surgir es que el plano de salida puede ser más pequeño que el

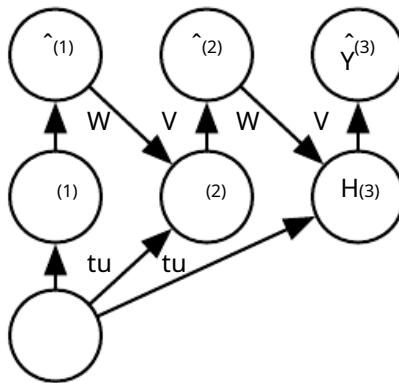


Figura 9.17: Un ejemplo de una red convolucional recurrente para el etiquetado de píxeles. La entrada es un tensor de imagen \mathbf{X} , con ejes correspondientes a filas de imágenes, columnas de imágenes y canales (rojo, verde, azul). El objetivo es generar un tensor de etiquetas $\hat{\mathbf{Y}}$, con una distribución de probabilidad sobre etiquetas para cada píxel. Este tensor tiene ejes correspondientes a filas de imágenes, columnas de imágenes y las diferentes clases. En lugar de dar salida $\hat{\mathbf{Y}}$ en un solo disparo, la red recurrente refina iterativamente su estimación $\hat{\mathbf{Y}}$ utilizando una estimación previa de $\hat{\mathbf{Y}}$ como entrada para crear una nueva estimación. Se utilizan los mismos parámetros para cada estimación actualizada, y la estimación se puede refinar tantas veces como se desee. El tensor de núcleos de convolución \mathbf{W} se utiliza en cada paso para calcular la representación oculta dada la imagen de entrada. El tensor del kernel \mathbf{V} se utiliza para producir una estimación de las etiquetas dados los valores ocultos. En todos menos en el primer paso, los núcleos \mathbf{W} están convolucionados $\hat{\mathbf{Y}}$ para proporcionar entrada a la capa oculta. En el primer paso de tiempo, este término se reemplaza por cero. Debido a que se usan los mismos parámetros en cada paso, este es un ejemplo de una red recurrente, como se describe en el capítulo 10.

plano de entrada, como se muestra en la figura 9.13. En los tipos de arquitecturas que normalmente se usan para la clasificación de un solo objeto en una imagen, la mayor reducción en las dimensiones espaciales de la red proviene del uso de capas de agrupación a gran escala. Para producir un mapa de salida de tamaño similar al de la entrada, se puede evitar la agrupación por completo ([Jainista et al., 2007](#)). Otra estrategia es simplemente emitir una cuadrícula de etiquetas de menor resolución ([Pinheiro y Colloberto, 2014, 2015](#)). Finalmente, en principio, se podría usar un operador de agrupación con zancada unitaria.

Una estrategia para el etiquetado de imágenes por píxeles es producir una suposición inicial de las etiquetas de la imagen, luego refinar esta suposición inicial utilizando las interacciones entre los píxeles vecinos. Repetir este paso de refinamiento varias veces corresponde a usar las mismas convoluciones en cada etapa, compartiendo pesos entre las últimas capas de la red profunda ([Jainista et al., 2007](#)). Esto hace que la secuencia de cálculos realizados por las sucesivas capas convolucionales con pesos compartidos entre capas sea un tipo particular de red recurrente ([Pinheiro y Colloberto, 2014, 2015](#)). Cifra 9.17 muestra la arquitectura de tal red convolucional recurrente.

Una vez que se realiza una predicción para cada píxel, se pueden usar varios métodos para procesar estas predicciones con el fin de obtener una segmentación de la imagen en regiones (Briggman *et al.*, 2009; Turaga *et al.*, 2010; Farabet *et al.*, 2013). La idea general es asumir que grandes grupos de píxeles contiguos tienden a estar asociados con la misma etiqueta. Los modelos gráficos pueden describir las relaciones probabilísticas entre píxeles vecinos. Alternativamente, la red convolucional se puede entrenar para maximizar una aproximación del objetivo de entrenamiento del modelo gráfico (Ning *et al.*, 2005; Thompson *et al.*, 2014).

9.7 Tipos de datos

Los datos utilizados con una red convolucional generalmente consisten en varios canales, siendo cada canal la observación de una cantidad diferente en algún punto del espacio o tiempo. Ver tabla 9.1 para ver ejemplos de tipos de datos con diferentes dimensionalidades y número de canales.

Para ver un ejemplo de redes convolucionales aplicadas a video, consulte Chen *et al.* (2010).

Hasta ahora hemos discutido solo el caso donde cada ejemplo en el tren y los datos de prueba tienen las mismas dimensiones espaciales. Una ventaja de las redes convolucionales es que también pueden procesar entradas con extensiones espaciales variables. Este tipo de entrada simplemente no se puede representar mediante redes neuronales tradicionales basadas en la multiplicación de matrices. Esto proporciona una razón convincente para usar redes convolucionales incluso cuando el costo computacional y el sobreajuste no son problemas significativos.

Por ejemplo, considere una colección de imágenes, donde cada imagen tiene un ancho y alto diferente. No está claro cómo modelar tales entradas con una matriz de peso de tamaño fijo. La convolución es sencilla de aplicar; el kernel simplemente se aplica un número diferente de veces según el tamaño de la entrada, y la salida de la operación de convolución se escala en consecuencia. La convolución puede verse como una multiplicación de matrices; el mismo kernel de convolución induce un tamaño diferente de matriz circulante de doble bloque para cada tamaño de entrada. A veces se permite que la salida de la red tenga un tamaño variable al igual que la entrada, por ejemplo si queremos asignar una etiqueta de clase a cada píxel de la entrada. En este caso, no es necesario ningún trabajo de diseño adicional. En otros casos, la red debe producir una salida de tamaño fijo, por ejemplo si queremos asignar una sola etiqueta de clase a toda la imagen. En este caso, debemos realizar algunos pasos de diseño adicionales, como insertar una capa de agrupación cuyas regiones de agrupación escalan en tamaño proporcional al tamaño de la entrada, para mantener un número fijo de salidas agrupadas. Algunos ejemplos de este tipo de estrategia se muestran en la figura 9.11.

	Un canal solo	multicanal
	Forma de onda de audio 1-D: El eje que convolve over corresponde a 3-D. Discretizamos el tiempo y mida la amplitud de la forma de onda una vez por paso de tiempo.	Datos de animación del esqueleto: Animaciones de tiempo renderizado por computadora en los personajes se generan alterando la pose de un "esqueleto" con el tiempo. En cada momento, la pose del personaje se describe mediante una especificación de los ángulos de cada una de las articulaciones del esqueleto del personaje. Cada canal en los datos que alimentamos al modelo convolucional representa el ángulo alrededor de un eje de una
2-D	Datos de audio que han sido preprocesados con una transformada de Fourier: podemos transformar la forma de onda de audio en un tensor 2D con diferentes filas correspondientes a diferentes frecuencias y diferentes columnas correspondientes a diferentes puntos en el tiempo. El uso de la convolución en el tiempo hace que el modelo sea equivalente a los cambios en el tiempo. El uso de convolución a lo largo del eje de frecuencia hace que el modelo sea equivalente a la frecuencia, de modo que la misma melodía tocada en una octava diferente produce la misma representación pero a una altura diferente en la salida de la red. Datos	articulación. Datos de imagen en color: un canal contiene los píxeles rojos, uno los píxeles verdes y otro los píxeles azules. El núcleo de convolución se mueve sobre los ejes horizontal y vertical de la imagen, lo que confiere equivalencia de traducción en ambas direcciones.
3-D	volumétricos: una fuente común de este tipo de datos es la tecnología de imágenes médicas, como las tomografías computarizadas.	Datos de video en color: un eje corresponde al tiempo, otro a la altura del cuadro de video y otro al ancho del cuadro de video.

Tabla 9.1: Ejemplos de diferentes formatos de datos que se pueden usar con redes convolucionales.

Tenga en cuenta que el uso de la convolución para procesar entradas de tamaño variable solo tiene sentido para las entradas que tienen un tamaño variable porque contienen cantidades variables de observación del mismo tipo: diferentes longitudes de registros en el tiempo, diferentes anchos de observaciones en el espacio, etc. La convolución no tiene sentido si la entrada tiene un tamaño variable porque opcionalmente puede incluir diferentes tipos de observaciones. Por ejemplo, si estamos procesando solicitudes para la universidad y nuestras características consisten tanto en calificaciones como en puntajes de exámenes estandarizados, pero no todos los solicitantes tomaron el examen estandarizado, entonces no tiene sentido combinar los mismos pesos entre las características correspondientes a las calificaciones, y las características correspondientes a los puntajes de las pruebas.

9.8 Algoritmos de convolución eficientes

Las aplicaciones modernas de redes convolucionales a menudo involucran redes que contienen más de un millón de unidades. Implementaciones potentes que aprovechan los recursos de computación en paralelo, como se explica en la sección 12.1, son esenciales. Sin embargo, en muchos casos también es posible acelerar la convolución seleccionando un algoritmo de convolución apropiado.

La convolución es equivalente a convertir tanto la entrada como el kernel al dominio de la frecuencia mediante una transformada de Fourier, realizar la multiplicación puntual de las dos señales y volver a convertir al dominio del tiempo mediante una transformada de Fourier inversa. Para algunos tamaños de problemas, esto puede ser más rápido que la implementación ingenua de la convolución discreta.

Cuando $w \times d$ El núcleo bidimensional se puede expresar como el producto externo de d vectores, un vector por dimensión, el núcleo se llama **separable**. Cuando el kernel es separable, la convolución ingenua es ineficiente. Es equivalente a componer d convoluciones unidimensionales con cada uno de estos vectores. El enfoque compuesto es significativamente más rápido que realizar una d -convolución dimensional con su producto exterior. El kernel también toma menos parámetros para representarlos como vectores. Si el núcleo es $w \times d$ elementos de ancho en cada dimensión, entonces la convolución multidimensional ingenua requiere $O(wd)$ tiempo de ejecución y espacio de almacenamiento de parámetros, mientras que la convolución separable requiere $O(w \times d)$ Tiempo de ejecución y espacio de almacenamiento de parámetros. Por supuesto, no todas las convoluciones se pueden representar de esta manera.

Un área activa de investigación es idear formas más rápidas de realizar la convolución o la convolución aproximada sin dañar la precisión del modelo. Incluso las técnicas que mejoran la eficiencia de la propagación hacia adelante son útiles porque en el entorno comercial, es típico dedicar más recursos al despliegue de una red que a su entrenamiento.

9.9 Funciones aleatorias o no supervisadas

Por lo general, la parte más costosa del entrenamiento de redes convolucionales es aprender las funciones. La capa de salida suele ser relativamente económica debido a la pequeña cantidad de características proporcionadas como entrada a esta capa después de pasar por varias capas de agrupación. Al realizar un entrenamiento supervisado con descenso de gradiente, cada paso de gradiente requiere una ejecución completa de propagación hacia adelante y hacia atrás a través de toda la red. Una forma de reducir el costo del entrenamiento de redes convolucionales es usar funciones que no se entrena de manera supervisada.

Hay tres estrategias básicas para obtener núcleos de convolución sin entrenamiento supervisado. Una es simplemente inicializarlos aleatoriamente. Otra es diseñarlos a mano, por ejemplo, configurando cada kernel para detectar bordes en una determinada orientación o escala. Finalmente, uno puede aprender los núcleos con un criterio no supervisado. Por ejemplo, [Coaté et al.\(2011\)](#) aplicar k -significa agrupar en pequeños parches de imagen, luego usar cada centroide aprendido como un núcleo de convolución. Parte [tercero](#) describe muchos más enfoques de aprendizaje no supervisado. El aprendizaje de las características con un criterio no supervisado permite determinarlas por separado de la capa clasificadora en la parte superior de la arquitectura. Luego, se pueden extraer las características de todo el conjunto de entrenamiento solo una vez, esencialmente construyendo un nuevo conjunto de entrenamiento para la última capa. Entonces, aprender la última capa suele ser un problema de optimización convexo, suponiendo que la última capa es algo así como una regresión logística o una SVM.

Los filtros aleatorios a menudo funcionan sorprendentemente bien en redes convolucionales ([Jarrett et al., 2009; Sajonia et al., 2011; Caballo pinto et al., 2011; cox y pinto, 2011](#)). [Sajonia et al. \(2011\)](#) mostró que las capas que consisten en una convolución seguida de una agrupación se vuelven selectivas en frecuencia e invariantes a la traducción cuando se les asignan pesos aleatorios. Argumentan que esto proporciona una forma económica de elegir la arquitectura de una red convolucional: primero evalúe el rendimiento de varias arquitecturas de red convolucional entrenando solo la última capa, luego tome lo mejor de estas arquitecturas y entrene toda la arquitectura usando un enfoque más costoso. .

Un enfoque intermedio es aprender las características, pero utilizando métodos que no requieren una propagación completa hacia adelante y hacia atrás en cada paso de gradiente. Al igual que con los perceptrones multicapa, usamos un preentrenamiento codicioso por capas para entrenar la primera capa de forma aislada, luego extraemos todas las características de la primera capa solo una vez, luego entrenamos la segunda capa de forma aislada dadas esas características, y así sucesivamente. Capítulo [8](#)ha descrito cómo realizar un preentrenamiento por capas codicioso supervisado, y parte [tercero](#)extiende esto al preentrenamiento codicioso por capas utilizando un criterio no supervisado en cada capa. El ejemplo canónico de entrenamiento previo codicioso por capas de un modelo convolucional es la red de creencias profundas convolucionales ([Sotavento et al., 2009](#)). Oferta de redes convolucionales

Nos brinda la oportunidad de llevar la estrategia de preentrenamiento un paso más allá de lo que es posible con los perceptrones multicapa. En lugar de entrenar una capa convolucional completa a la vez, podemos entrenar un modelo de un parche pequeño, como [Coatéset al.\(2011\)](#) hazlo k -medio. Luego podemos usar los parámetros de este modelo basado en parches para definir los núcleos de una capa convolucional. Esto significa que es posible utilizar el aprendizaje no supervisado para entrenar una red convolucional *sin usar convolución durante el proceso de entrenamiento*. Usando este enfoque, podemos entrenar modelos muy grandes e incurrir en un alto costo computacional solo en el momento de la inferencia ([Ranzato et al., 2007b; Jarrett et al., 2009; Kavukcuoglu et al., 2010; Coatéset al., 2013](#)). Este enfoque fue popular aproximadamente entre 2007 y 2013, cuando los conjuntos de datos etiquetados eran pequeños y el poder computacional era más limitado. Hoy en día, la mayoría de las redes convolucionales se entranan de forma puramente supervisada, utilizando una propagación completa hacia adelante y hacia atrás a través de toda la red en cada iteración de entrenamiento.

Al igual que con otros enfoques del preentrenamiento no supervisado, sigue siendo difícil descifrar la causa de algunos de los beneficios observados con este enfoque. El preentrenamiento no supervisado puede ofrecer cierta regularización en relación con el entrenamiento supervisado, o simplemente puede permitirnos entrenar arquitecturas mucho más grandes debido al costo computacional reducido de la regla de aprendizaje.

9.10 La base neurocientífica de las redes convolucionales

Las redes convolucionales son quizás la mayor historia de éxito de la inteligencia artificial de inspiración biológica. Aunque las redes convolucionales se han guiado por muchos otros campos, algunos de los principios de diseño clave de las redes neuronales se extrajeron de la neurociencia.

La historia de las redes convolucionales comienza con experimentos neurocientíficos mucho antes de que se desarrollaran los modelos computacionales relevantes. Los neurofisiólogos David Hubel y Torsten Wiesel colaboraron durante varios años para determinar muchos de los hechos más básicos sobre cómo funciona el sistema de visión de los mamíferos ([Hubel y Wiesel, 1959, 1962, 1968](#)). Sus logros fueron finalmente reconocidos con un premio Nobel. Sus hallazgos, que han tenido la mayor influencia en los modelos contemporáneos de aprendizaje profundo, se basaron en el registro de la actividad de neuronas individuales en gatos. Observaron cómo las neuronas del cerebro del gato respondían a las imágenes proyectadas en ubicaciones precisas en una pantalla frente al gato. Su gran descubrimiento fue que las neuronas del sistema visual primitivo respondían con mayor fuerza a patrones de luz muy específicos, como barras orientadas con precisión, pero apenas respondían a otros patrones.

Su trabajo ayudó a caracterizar muchos aspectos de la función cerebral que están más allá del alcance de este libro. Desde el punto de vista del aprendizaje profundo, podemos centrarnos en una vista simplificada de dibujos animados de la función cerebral.

En esta vista simplificada, nos enfocamos en una parte del cerebro llamada V1, también conocida como **corteza visual primaria**. V1 es la primera área del cerebro que comienza a realizar un procesamiento significativamente avanzado de información visual. En esta vista de dibujos animados, las imágenes se forman cuando la luz llega al ojo y estimula la retina, el tejido sensible a la luz en la parte posterior del ojo. Las neuronas de la retina realizan un procesamiento previo simple de la imagen, pero no alteran sustancialmente la forma en que se representa. Luego, la imagen pasa a través del nervio óptico y una región del cerebro llamada núcleo geniculado lateral. La función principal, en lo que a nosotros respecta aquí, de estas dos regiones anatómicas es principalmente llevar la señal del ojo a V1, que se encuentra en la parte posterior de la cabeza.

Una capa de red convolucional está diseñada para capturar tres propiedades de V1:

1. V1 se organiza en un mapa espacial. En realidad, tiene una estructura bidimensional que refleja la estructura de la imagen en la retina. Por ejemplo, la luz que llega a la mitad inferior de la retina afecta solo a la mitad correspondiente de V1. Las redes convolucionales capturan esta propiedad al tener sus características definidas en términos de mapas bidimensionales.
2. V1 contiene muchos **celdas simples**. La actividad de una célula simple puede caracterizarse hasta cierto punto por una función lineal de la imagen en un pequeño campo receptivo espacialmente localizado. Las unidades detectoras de una red convolucional están diseñadas para emular estas propiedades de celdas simples.
3. V1 también contiene muchos **células complejas**. Estas celdas responden a características que son similares a las detectadas por celdas simples, pero las celdas complejas son invariantes a pequeños cambios en la posición de la característica. Esto inspira las unidades de agrupación de las redes convolucionales. Las celdas complejas también son invariantes a algunos cambios en la iluminación que no se pueden capturar simplemente agrupando las ubicaciones espaciales. Estas invariancias han inspirado algunas de las estrategias de agrupación de canales cruzados en redes convolucionales, como unidades maxout ([Buen compañero et al., 2013a](#)).

Aunque sabemos más sobre V1, generalmente se cree que los mismos principios básicos se aplican a otras áreas del sistema visual. En nuestra vista de dibujos animados del sistema visual, la estrategia básica de detección seguida de agrupación se aplica repetidamente a medida que nos adentramos más en el cerebro. A medida que atravesamos múltiples capas anatómicas del cerebro, eventualmente encontramos células que responden a algún concepto específico y son invariables a muchas transformaciones de la entrada. Estas células han sido

apodadas “células de la abuela”, la idea es que una persona pueda tener una neurona que se active al ver una imagen de su abuela, independientemente de si aparece en el lado izquierdo o derecho de la imagen, si la imagen es un primer plano de su rostro o toma ampliada de todo su cuerpo, ya sea que esté muy iluminada o en la sombra, etc.

Se ha demostrado que estas células abuelas existen realmente en el cerebro humano, en una región llamada lóbulo temporal medial ([Quiroga et al., 2005](#)). Los investigadores probaron si las neuronas individuales responderían a las fotos de personas famosas. Encontraron lo que se ha dado en llamar la “neurona de Halle Berry”: una neurona individual que se activa con el concepto de Halle Berry. Esta neurona se activa cuando una persona ve una foto de Halle Berry, un dibujo de Halle Berry o incluso un texto que contiene las palabras “Halle Berry”. Por supuesto, esto no tiene nada que ver con la propia Halle Berry; otras neuronas respondieron a la presencia de Bill Clinton, Jennifer Aniston, etc.

Estas neuronas del lóbulo temporal medial son algo más generales que las redes convolucionales modernas, que no se generalizarían automáticamente para identificar a una persona u objeto al leer su nombre. El análogo más cercano a la última capa de características de una red convolucional es un área del cerebro llamada corteza inferotemporal (IT). Al ver un objeto, la información fluye desde la retina, a través del LGN, a V1, luego a V2, luego a V4 y luego a TI. Esto sucede dentro de los primeros 100 ms de vislumbrar un objeto. Si a una persona se le permite seguir mirando el objeto durante más tiempo, la información comenzará a fluir hacia atrás a medida que el cerebro utiliza la retroalimentación de arriba hacia abajo para actualizar las activaciones en las áreas cerebrales de nivel inferior. Sin embargo, si interrumpimos la mirada de la persona y observamos solo las tasas de disparo que resultan de los primeros 100 ms de activación principalmente de retroalimentación, entonces TI demuestra ser muy similar a una red convolucional. Las redes convolucionales pueden predecir las tasas de disparo de TI y también funcionan de manera muy similar a los humanos (tiempo limitado) en tareas de reconocimiento de objetos ([dicarlo, 2013](#)).

Dicho esto, existen muchas diferencias entre las redes convolucionales y el sistema de visión de los mamíferos. Algunas de estas diferencias son bien conocidas por los neurocientíficos computacionales, pero están fuera del alcance de este libro. Algunas de estas diferencias aún no se conocen, porque muchas preguntas básicas sobre cómo funciona el sistema de visión de los mamíferos siguen sin respuesta. Como una breve lista:

- El ojo humano es en su mayoría de muy baja resolución, a excepción de un pequeño parche llamado **fóvea**. La fóvea solo observa un área del tamaño de la uña del pulgar sostenida con los brazos extendidos. Aunque sentimos que podemos ver una escena completa en alta resolución, esta es una ilusión creada por la parte subconsciente de nuestro cerebro, ya que une varias imágenes de áreas pequeñas. La mayoría de las redes convolucionales en realidad reciben grandes fotografías de resolución completa como entrada. El cerebro humano hace

varios movimientos oculares llamados **movimientos sacádicos** para vislumbrar las partes más destacadas visualmente o relevantes para la tarea de una escena. La incorporación de mecanismos de atención similares en modelos de aprendizaje profundo es una dirección de investigación activa. En el contexto del aprendizaje profundo, los mecanismos de atención han sido más exitosos para el procesamiento del lenguaje natural, como se describe en la sección 12.4.5.1. Se han desarrollado varios modelos visuales con mecanismos de foveación, pero hasta ahora no se han convertido en el enfoque dominante ([Larochelle y Hinton, 2010; Denile et al., 2012](#)).

- El sistema visual humano está integrado con muchos otros sentidos, como el oído, y factores como nuestro estado de ánimo y pensamientos. Las redes convolucionales hasta ahora son puramente visuales.
- El sistema visual humano hace mucho más que reconocer objetos. Es capaz de comprender escenas completas, incluidos muchos objetos y relaciones entre objetos, y procesa la rica información geométrica tridimensional necesaria para que nuestros cuerpos interactúen con el mundo. Las redes convolucionales se han aplicado a algunos de estos problemas, pero estas aplicaciones están en su infancia.
- Incluso áreas cerebrales simples como V1 se ven muy afectadas por la retroalimentación de los niveles superiores. La retroalimentación se ha explorado ampliamente en los modelos de redes neuronales, pero aún no se ha demostrado que ofrezca una mejora convincente.
- Si bien las tasas de disparo de TI feedforward capturan gran parte de la misma información que las características de la red convolucional, no está claro qué tan similares son los cálculos intermedios. El cerebro probablemente utiliza funciones de activación y agrupación muy diferentes. La activación de una neurona individual probablemente no esté bien caracterizada por una única respuesta de filtro lineal. Un modelo reciente de V1 involucra múltiples filtros cuadráticos para cada neurona ([Óxido et al., 2005](#)). De hecho, nuestra caricatura de "células simples" y "células complejas" podría crear una distinción inexistente; Las celdas simples y las celdas complejas pueden ser el mismo tipo de celda pero con sus "parámetros" que permiten un continuo de comportamientos que van desde lo que llamamos "simple" hasta lo que llamamos "complejo".

También vale la pena mencionar que la neurociencia nos ha dicho relativamente poco sobre cómo *tren*redes convolucionales. Las estructuras de modelos con parámetros compartidos en múltiples ubicaciones espaciales se remontan a los primeros modelos de visión conexiónistas ([Marr y Poggio, 1976](#)), pero estos modelos no utilizaron el algoritmo moderno de retropropagación ni el descenso de gradiente. Por ejemplo, el Neocognitrón ([fukushima, 1980](#)) incorporó la mayoría de los elementos de diseño de la arquitectura modelo de la red convolucional moderna, pero se basó en un algoritmo de agrupamiento no supervisado por capas.

Lang y Hinton(1988) introdujo el uso de retropropagación para entrenar **redes neuronales de retardo de tiempo**(TDNN). Para usar la terminología contemporánea, las TDNN son redes convolucionales unidimensionales aplicadas a series de tiempo. La retropropagación aplicada a estos modelos no se inspiró en ninguna observación neurocientífica y algunos la consideran biológicamente inverosímil. Tras el éxito del entrenamiento basado en la retropropagación de TDNN, (lecun et al.,1989) desarrolló la red convolucional moderna aplicando el mismo algoritmo de entrenamiento a la convolución 2-D aplicada a las imágenes.

Hasta ahora, hemos descrito cómo las células simples son más o menos lineales y selectivas para ciertas características, las células complejas son más no lineales y se vuelven invariantes a algunas transformaciones de estas características de células simples, y las pilas de capas que alternan entre selectividad e invariancia pueden producir células abuela para muy fenómenos específicos. Todavía no hemos descrito con precisión lo que detectan estas células individuales. En una red profunda no lineal, puede ser difícil comprender la función de las celdas individuales. Las celdas simples en la primera capa son más fáciles de analizar porque sus respuestas son impulsadas por una función lineal. En una red neuronal artificial, podemos mostrar una imagen del kernel de convolución para ver a qué responde el canal correspondiente de una capa convolucional. En una red neuronal biológica, no tenemos acceso a los pesos en sí. En cambio, colocamos un electrodo en la neurona misma, mostramos varias muestras de imágenes de ruido blanco frente a la retina del animal y registramos cómo cada una de estas muestras hace que la neurona se active. Entonces podemos ajustar un modelo lineal a estas respuestas para obtener una aproximación de los pesos de las neuronas. Este enfoque se conoce como **correlación inversa**(Ringach y Shapley, 2004).

La correlación inversa nos muestra que la mayoría de las celdas V1 tienen pesos que se describen mediante**funciones gabor**. La función Gabor describe el peso en un punto 2D de la imagen. Podemos pensar en una imagen como una función de coordenadas 2-D, $I(x,y)$. Del mismo modo, podemos pensar en una celda simple como un muestreo de la imagen en un conjunto de ubicaciones, definidas por un conjunto de X coordenadas x y un conjunto de y coordenadas, Y ,y aplicando pesos que también son una función de la ubicación, $w(x,y)$. Desde este punto de vista, la respuesta de una celda simple a una imagen viene dada por

$$s(I) = \sum_{x \in X, y \in Y} w(x,y)I(x,y). \quad (9.15)$$

Especificamente, $w(x,y)$ toma la forma de una función de Gabor:

$$w(x,y; a, \beta_x, \beta_y, f, \varphi, x_0, y_0, t) = a \exp -\beta^2 (x-x_0)^2 + (y-y_0)^2 \quad \text{porque efectos especiales } (\varphi), \quad (9.16)$$

dónde

$$X = (x - x_0) \operatorname{sen}(\varphi) + (y - y_0) \operatorname{pecado}(\varphi) \quad (9.17)$$

y

$$y = -(x - x_0) \operatorname{pecado}(\tau) + (y - y_0) \operatorname{porque}(\tau). \quad (9.18)$$

Aquí, $\alpha, \beta_x, \beta_y, F, \varphi, x_0, y_0$, y τ son parámetros que controlan las propiedades de la función de Gabor. Cifra 9.18 muestra algunos ejemplos de funciones de Gabor con diferentes configuraciones de estos parámetros.

Los parámetros x_0, y_0 , y τ definen un sistema de coordenadas. Traducimos y rotamos X y y para formar X_{yy} . Específicamente, la celda simple responderá a las características de la imagen centradas en el punto (x_0, y_0) , y responderá a los cambios de brillo a medida que nos desplazemos por una línea girada τ radianes de la horizontal.

Visto como una función de X_{yy} , la función ψ responde a los cambios en el brillo a medida que nos movemos a lo largo de la X -eje. Tiene dos factores importantes: uno es una función gaussiana y la otra es una función coseno.

El factor gaussiano $\alpha \operatorname{Exp}(-\beta_x X^2 - \beta_y y^2)$ puede verse como un término de activación que asegura que la celda simple solo responderá a valores cercanos a donde X_{yy} son ambos cero, en otras palabras, cerca del centro del campo receptivo de la célula. El factor de escala α ajusta la magnitud total de la respuesta de la celda simple, mientras que β_x y β_y controlan qué tan rápido cae su campo receptivo.

El factor coseno por $\operatorname{porque}(F(x - x_0) + \varphi)$ controla cómo la célula simple responde a los cambios de brillo a lo largo de la X -eje. El parámetro F controla la frecuencia del coseno y φ controla su compensación de fase.

En conjunto, esta vista de dibujos animados de celdas simples significa que una celda simple responde a una frecuencia espacial específica de brillo en una dirección específica en una ubicación específica. Las células simples están más excitadas cuando la onda de brillo en la imagen tiene la misma fase que los pesos. Esto ocurre cuando la imagen es brillante donde los pesos son positivos y oscura donde los pesos son negativos. Las celdas simples se inhiben más cuando la onda de brillo está completamente desfasada con los pesos, cuando la imagen es oscura donde los pesos son positivos y brillante donde los pesos son negativos.

La vista de dibujos animados de una celda compleja es que calcula la norma de la Vector 2-D que contiene dos respuestas de células simples: $C(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. Un caso especial importante ocurre cuando s_0 tiene todos los mismos parámetros que s_0 excepto por φ , y s_0 se establece de tal manera que s_0 está un cuarto de ciclo fuera de fase con s_0 . En este caso, s_0 forma un **par en cuadratura**. Una celda compleja definida de esta manera responde cuando la imagen Gaussiana responderá a $I(x, y) \operatorname{Exp}(-\beta_x X^2 - \beta_y y^2)$ contiene una onda sinusoidal de alta amplitud con frecuencia F en dirección τ cerca (x_0, y_0) , *independientemente del desplazamiento de fase de esta onda*. En otras palabras, la celda compleja es invariantes a pequeñas traslaciones de la imagen en la dirección τ , o negar la imagen.

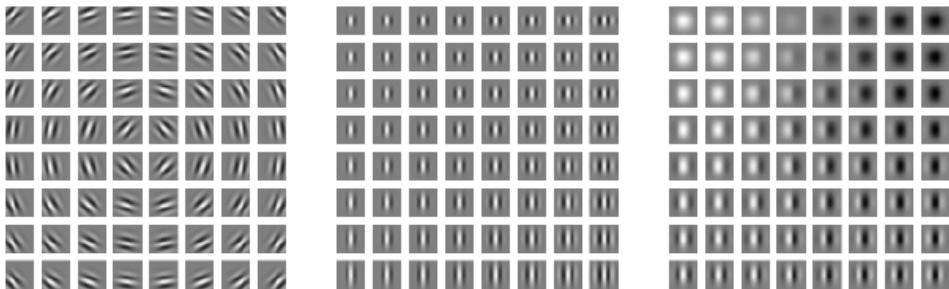


Figura 9.18: Funciones de Gabor con una variedad de configuraciones de parámetros. El blanco indica un peso positivo grande, el negro indica un peso negativo grande y el fondo gris corresponde a un peso cero. (*Izquierda*) Funciones de Gabor con diferentes valores de los parámetros que controlan el sistema de coordenadas: x_0, y_0, τ . A cada función de Gabor en esta cuadrícula se le asigna un valor de x_0, y_0 proporcional a su posición en su cuadrícula, y se elige de modo que cada filtro Gabor sea sensible a la dirección que irradia desde el centro de la cuadrícula. Para las otras dos parcelas, x_0, y_0, τ se fijan en cero. (*Centro*) Funciones de Gabor con diferentes parámetros de escala gaussiana β_x y β_y . Las funciones de Gabor están dispuestas en ancho creciente (decreciente β_x) a medida que nos movemos de izquierda a derecha a través de la cuadrícula, y aumentando la altura (disminuyendo β_y) a medida que avanzamos de arriba hacia abajo. Para las otras dos parcelas, los valores se fijan en 1,5 × el ancho de la imagen. (*Derecha*) Funciones de Gabor con diferentes parámetros sinusoidales F y φ . A medida que avanzamos de arriba hacia abajo, F aumenta, y a medida que nos movemos de izquierda a derecha, φ aumenta. Para las otras dos parcelas, φ se fija en 0 y se fija en 5 × el ancho de la imagen.

(reemplazando el negro por el blanco y viceversa).

Algunas de las correspondencias más sorprendentes entre la neurociencia y el aprendizaje automático provienen de la comparación visual de las características aprendidas por los modelos de aprendizaje automático con las empleadas por V1. Olshausen y campo (1996) mostró que un algoritmo de aprendizaje simple no supervisado, de codificación escasa, aprende características con campos receptivos similares a los de las células simples. Desde entonces, hemos encontrado que una variedad extremadamente amplia de algoritmos de aprendizaje estadístico aprenden características con funciones similares a las de Gabor cuando se aplican a imágenes naturales. Esto incluye la mayoría de los algoritmos de aprendizaje profundo, que aprenden estas características en su primera capa. Cifra 9.19 muestra algunos ejemplos. Debido a que tantos algoritmos de aprendizaje diferentes aprenden detectores de bordes, es difícil concluir que cualquier algoritmo de aprendizaje específico es el modelo "correcto" del cerebro solo en función de las características que aprende (aunque ciertamente puede ser una mala señal si un algoritmo no lo hace). No aprender algún tipo de detector de bordes cuando se aplica a imágenes naturales). Estas características son una parte importante de la estructura estadística de las imágenes naturales y pueden recuperarse mediante muchos enfoques diferentes para el modelado estadístico. Ver Hyvärinen et al. (2009) para una revisión del campo de las estadísticas de imágenes naturales.

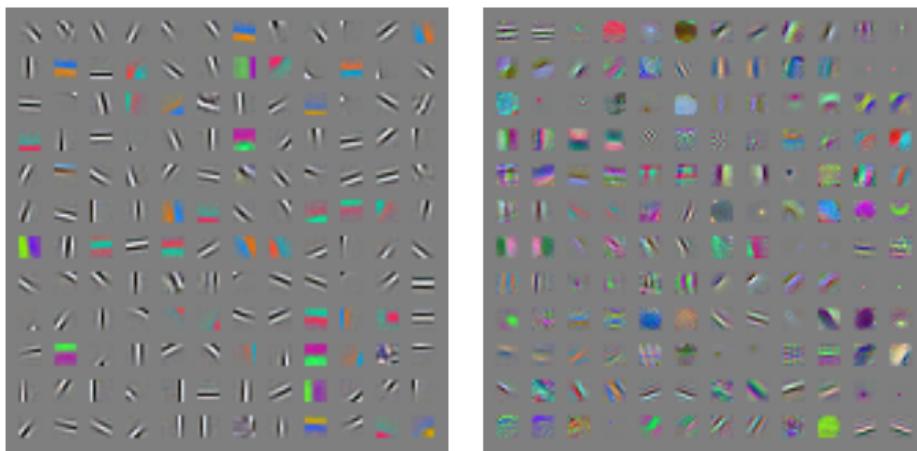


Figura 9.19: Muchos algoritmos de aprendizaje automático aprenden funciones que detectan bordes o colores específicos de los bordes cuando se aplican a imágenes naturales. Estos detectores de características recuerdan las funciones de Gabor que se sabe que están presentes en la corteza visual primaria.(Izquierda)Pesos aprendidos por un algoritmo de aprendizaje no supervisado (codificación dispersa de picos y losas) aplicados a pequeños parches de imagen.(Bien)Núcleos de convolución aprendidos por la primera capa de una red maxout convolucional completamente supervisada. Los pares de filtros vecinos impulsan la misma unidad maxout.

9.11 Redes convolucionales y la historia del aprendizaje profundo

Las redes convolucionales han jugado un papel importante en la historia del aprendizaje profundo. Son un ejemplo clave de una aplicación exitosa de los conocimientos obtenidos mediante el estudio del cerebro en aplicaciones de aprendizaje automático. También fueron algunos de los primeros modelos profundos en funcionar bien, mucho antes de que los modelos profundos arbitrarios se consideraran viables. Las redes convolucionales también fueron algunas de las primeras redes neuronales en resolver aplicaciones comerciales importantes y permanecen a la vanguardia de las aplicaciones comerciales de aprendizaje profundo en la actualidad. Por ejemplo, en la década de 1990, el grupo de investigación de redes neuronales de AT&T desarrolló una red convolucional para leer cheques ([lecunet et al., 1998b](#)). A fines de la década de 1990, este sistema implementado por NEC estaba leyendo más del 10% de todos los cheques en los EE. UU. Más tarde, Microsoft implementó varios sistemas de reconocimiento de escritura a mano y OCR basados en redes convolucionales ([Simard et al., 2003](#)). Ver capítulo 12 para obtener más detalles sobre tales aplicaciones y aplicaciones más modernas de redes convolucionales. Ver [lecunet et al. \(2010\)](#) para una historia más detallada de las redes convolucionales hasta 2010.

Las redes convolucionales también se utilizaron para ganar muchos concursos. La intensidad actual del interés comercial en el aprendizaje profundo comenzó cuando [Krizhevskiy et al. \(2012\)](#) ganó el desafío de reconocimiento de objetos de ImageNet, pero las redes convolucionales

se había utilizado para ganar otros concursos de aprendizaje automático y visión por computadora con menos impacto años antes.

Las redes convolucionales fueron algunas de las primeras redes profundas en funcionamiento entrenadas con propagación hacia atrás. No está del todo claro por qué las redes convolucionales tuvieron éxito cuando se consideró que las redes generales de retropropagación habían fallado. Puede ser simplemente que las redes convolucionales fueran más eficientes computacionalmente que las redes totalmente conectadas, por lo que fue más fácil ejecutar múltiples experimentos con ellas y ajustar su implementación e hiperparámetros. Las redes más grandes también parecen ser más fáciles de entrenar. Con el hardware moderno, las grandes redes totalmente conectadas parecen tener un rendimiento razonable en muchas tareas, incluso cuando se utilizan conjuntos de datos que estaban disponibles y funciones de activación que eran populares en la época en que se creía que las redes totalmente conectadas no funcionaban bien. Puede ser que las principales barreras para el éxito de las redes neuronales fueran psicológicas (los profesionales no esperaban que las redes neuronales funcionaran, por lo que no hicieron un esfuerzo serio para usar redes neuronales). Cualquiera que sea el caso, es una suerte que las redes convolucionales funcionaran bien hace décadas. En muchos sentidos, llevaron la antorcha del resto del aprendizaje profundo y allanaron el camino para la aceptación de las redes neuronales en general.

Las redes convolucionales brindan una forma de especializar las redes neuronales para trabajar con datos que tienen una topología estructurada en cuadrícula clara y para escalar dichos modelos a un tamaño muy grande. Este enfoque ha sido el más exitoso en una topología de imagen bidimensional. Para procesar datos secuenciales unidimensionales, pasamos a otra poderosa especialización del marco de trabajo de las redes neuronales: las redes neuronales recurrentes.

Capítulo 10

Modelado de secuencias: redes recurrentes y recursivas

Redes neuronales recurrenteso RNN ([Rumelhart et al., 1986a](#)) son una familia de redes neuronales para procesar datos secuenciales. Así como una red convolucional es una red neuronal que está especializada para procesar una cuadrícula de valores como una imagen, una red neuronal recurrente es una red neuronal que está especializada para procesar una secuencia de valores $X(1), \dots, X(t)$. Así como las redes convolucionales pueden escalar fácilmente a imágenes con gran ancho y alto, y algunas redes convolucionales pueden procesar imágenes de tamaño variable, las redes recurrentes pueden escalar a secuencias mucho más largas de lo que sería práctico para redes sin especialización basada en secuencias. La mayoría de las redes recurrentes también pueden procesar secuencias de longitud variable.

Para pasar de redes multicapa a redes recurrentes, debemos aprovechar una de las primeras ideas encontradas en el aprendizaje automático y los modelos estadísticos de la década de 1980: compartir parámetros en diferentes partes de un modelo. Compartir parámetros hace posible extender y aplicar el modelo a ejemplos de diferentes formas (diferentes longitudes, aquí) y generalizar a través de ellos. Si tuviéramos parámetros separados para cada valor del índice de tiempo, no podríamos generalizar a longitudes de secuencia no vistas durante el entrenamiento, ni compartir la fuerza estadística a través de diferentes longitudes de secuencia y en diferentes posiciones en el tiempo. Tal intercambio es particularmente importante cuando una información específica puede ocurrir en múltiples posiciones dentro de la secuencia. Por ejemplo, considere las dos oraciones “Fui a Nepal en 2009” y “En 2009, fui a Nepal.

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

Una idea relacionada es el uso de convolución a través de una secuencia temporal 1-D. Este enfoque convolucional es la base de las redes neuronales de retardo de tiempo ([Lang y Hinton, 1988;waibelet al., 1989;Idioma et al., 1990](#)). La operación de convolución permite que una red comparta parámetros a lo largo del tiempo, pero es poco profunda. La salida de la convolución es una secuencia en la que cada miembro de la salida es una función de un pequeño número de miembros vecinos de la entrada. La idea de compartir parámetros se manifiesta en la aplicación del mismo kernel de convolución en cada paso de tiempo. Las redes recurrentes comparten parámetros de una manera diferente. Cada miembro de la salida es una función de los miembros anteriores de la salida. Cada miembro de la salida se produce usando la misma regla de actualización aplicada a las salidas anteriores. Esta formulación recurrente da como resultado el intercambio de parámetros a través de un gráfico computacional muy profundo.

Para simplificar la exposición, nos referimos a las RNN como operando en una secuencia que contiene vectores X_t con el índice de paso de tiempo t que van desde 1 a T . En la práctica, las redes recurrentes suelen operar en minilotes de tales secuencias, con una longitud de secuencia diferente T para cada miembro del minilote. Hemos omitido los índices de minilotes para simplificar la notación. Además, el índice de paso de tiempo no necesita referirse literalmente al paso del tiempo en el mundo real. A veces se refiere sólo a la posición en la secuencia. Las RNN también se pueden aplicar en dos dimensiones a través de datos espaciales como imágenes, e incluso cuando se aplican a datos relacionados con el tiempo, la red puede tener conexiones que retroceden en el tiempo, siempre que se observe la secuencia completa antes de proporcionarla a la red.

Este capítulo amplía la idea de un gráfico computacional para incluir ciclos. Estos ciclos representan la influencia del valor presente de una variable sobre su propio valor en un paso de tiempo futuro. Dichos gráficos computacionales nos permiten definir redes neuronales recurrentes. Luego describimos muchas formas diferentes de construir, entrenar y usar redes neuronales recurrentes.

Para obtener más información sobre redes neuronales recurrentes de la que está disponible en este capítulo, remitimos al lector al libro de texto de [Tumbas\(2012\)](#).

10.1 Despliegue de gráficos computacionales

Un gráfico computacional es una forma de formalizar la estructura de un conjunto de cálculos, como los involucrados en el mapeo de entradas y parámetros a salidas y pérdidas. Consulte la sección 6.5.1 para una introducción general. En esta sección explicamos la idea de **despliegue** un cómputo recursivo o recurrente en un gráfico computacional que tiene una estructura repetitiva, típicamente correspondiente a una cadena de eventos. El despliegue de este gráfico da como resultado el intercambio de parámetros a través de una estructura de red profunda.

Por ejemplo, considere la forma clásica de un sistema dinámico:

$$S(t) = F(S(t-1); \theta), \quad (10.1)$$

dónde $S(t)$ se llama el estado del sistema.

Ecuación 10.1 es recurrente porque la definición de $S(t)$ en el momento t se refiere a la misma definición en el momento $t-1$.

Para un número finito de pasos de tiempo τ , el gráfico se puede desarrollar aplicando la definición $\tau-1$ veces. Por ejemplo, si desarrollamos la ecuación 10.1 para $\tau=3$ pasos de tiempo, obtenemos

$$S(3) = F(S(2); \theta) \quad (10.2)$$

$$= F(F(S(1); \theta); \theta) \quad (10.3)$$

Desplegar la ecuación aplicando repetidamente la definición de esta manera ha producido una expresión que no implica recursividad. Tal expresión ahora se puede representar mediante un gráfico computacional acíclico dirigido tradicional. El gráfico computacional de la ecuación desplegado 10.1 y ecuación 10.3 se ilustra en la figura 10.1.

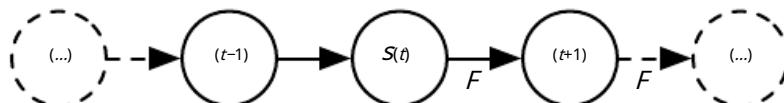


Figura 10.1: El sistema dinámico clásico descrito por la ecuación 10.1, ilustrado como un gráfico computacional desplegado. Cada nodo representa el estado en algún momento t y la función F mapea el estado en tal estado en $t+1$. Los mismos parámetros (el mismo valor de θ utilizado para parametrizar F) se utilizan para todos los pasos de tiempo.

Como otro ejemplo, consideremos un sistema dinámico impulsado por una señal externa $X(t)$,

$$S(t) = F(S(t-1), X(t); \theta), \quad (10.4)$$

donde vemos que el estado ahora contiene información sobre toda la secuencia pasada.

Las redes neuronales recurrentes se pueden construir de muchas maneras diferentes. Así como casi cualquier función puede considerarse una red neuronal de retroalimentación, esencialmente cualquier función que involucre recurrencia puede considerarse una red neuronal recurrente.

Muchas redes neuronales recurrentes utilizan la ecuación 10.5 o una ecuación similar para definir los valores de sus unidades ocultas. Para indicar que el estado son las unidades ocultas de la red, ahora reescribimos la ecuación 10.4 usando la variable h para representar al estado:

$$h_t = f(h_{t-1}, X_t; \theta), \quad (10.5)$$

ilustrado en la figura 10.2, los RNN típicos agregarán características arquitectónicas adicionales, como capas de salida que leen información fuera del estado h para hacer predicciones.

Cuando la red recurrente está entrenada para realizar una tarea que requiere predecir el futuro a partir del pasado, la red típicamente aprende a usar h_t como una especie de resumen con pérdidas de los aspectos relevantes para la tarea de la secuencia pasada de entradas hasta t . Este resumen es, en general, necesariamente con pérdidas, ya que mapea una secuencia de longitud arbitraria ($X_t, X_{t-1}, X_{t-2}, \dots, X_2, X_1$) a un vector de longitud fija h_t . Dependiendo del criterio de entrenamiento, este resumen puede mantener selectivamente algunos aspectos de la secuencia pasada con más precisión que otros aspectos. Por ejemplo, si el RNN se usa en el modelado de lenguaje estadístico, generalmente para predecir la siguiente palabra dadas las palabras anteriores, puede que no sea necesario almacenar toda la información en la secuencia de entrada hasta el momento t , sino solo la información suficiente para predecir el resto de la oración. La situación más exigente es cuando pedimos h_t ser lo suficientemente rico como para permitir que uno recupere aproximadamente la secuencia de entrada, como en los marcos de trabajo de autocodificador (capítulo 14).

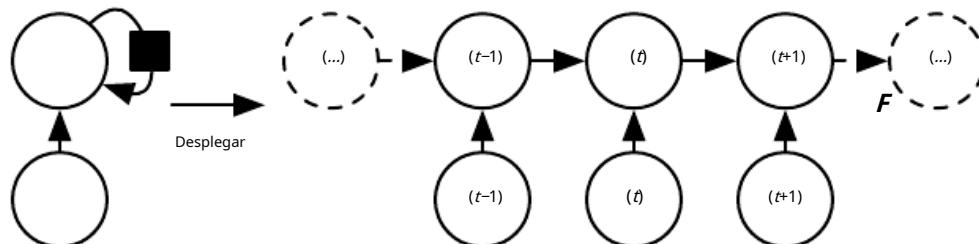


Figura 10.2: Una red recurrente sin salidas. Esta red recurrente solo procesa información de la entrada X_t al incorporarlo al estado h_t que se transmite a través del tiempo. (Izquierda) Diagrama de circuito. El cuadrado negro indica un retraso de un solo paso de tiempo. (Bien) La misma red vista como un gráfico computacional desplegado, donde cada nodo ahora está asociado con una instancia de tiempo particular.

Ecuación 10.5 se puede dibujar de dos maneras diferentes. Una forma de dibujar el RNN es con un diagrama que contenga un nodo para cada componente que pueda existir en un

implementación física del modelo, como una red neuronal biológica. En esta vista, la red define un circuito que opera en tiempo real, con partes físicas cuyo estado actual puede influir en su estado futuro, como en la figura de la izquierda.[10.2](#). A lo largo de este capítulo, usamos un cuadrado negro en un diagrama de circuito para indicar que una interacción tiene lugar con un retraso de un solo paso de tiempo, desde el estado en el momento t al estado en el momento $t+1$. La otra forma de dibujar el RNN es como un gráfico computacional desplegado, en el que cada componente está representado por muchas variables diferentes, con una variable por paso de tiempo, que representa el estado del componente en ese momento. Cada variable para cada paso de tiempo se dibuja como un nodo separado del gráfico computacional, como en la parte derecha de la figura.[10.2](#). Lo que llamamos despliegue es la operación que asigna un circuito como en el lado izquierdo de la figura a un gráfico computacional con piezas repetidas como en el lado derecho. El gráfico desplegado ahora tiene un tamaño que depende de la longitud de la secuencia.

Podemos representar la recurrencia desplegada después de t pasos con una función $gramo(t)$:

$$h(t) = gromo(t)(X(t), X(t-1), X(t-2), \dots, X(2), X(1)) \quad (10.6)$$

$$= F(h(t-1), X(t); \theta) \quad (10.7)$$

La función $gramo(t)$ toma toda la secuencia pasada ($X(t), X(t-1), X(t-2), \dots, X(2), X(1)$) como entrada y produce el estado actual, pero la estructura recurrente desplegada nos permite factorizar $gramo(t)$ en la aplicación repetida de una función F . El proceso de despliegado presenta así dos grandes ventajas:

1. Independientemente de la longitud de la secuencia, el modelo aprendido siempre tiene el mismo tamaño de entrada, porque se especifica en términos de transición de un estado a otro, en lugar de especificarse en términos de una historia de estados de longitud variable.
2. Es posible utilizar el *mismo* función de transición F con los mismos parámetros en cada paso de tiempo.

Estos dos factores hacen posible aprender un solo modelo, F , que opera en todos los pasos de tiempo y todas las longitudes de secuencia, en lugar de tener que aprender un modelo separado $gramo(t)$ para todos los pasos de tiempo posibles. El aprendizaje de un solo modelo compartido permite la generalización a longitudes de secuencia que no aparecían en el conjunto de entrenamiento y permite estimar el modelo con muchos menos ejemplos de entrenamiento de los que se necesitarían sin compartir parámetros.

Tanto el gráfico recurrente como el gráfico desenrollado tienen sus usos. El gráfico recurrente es sucinto. El gráfico desplegado proporciona una descripción explícita de qué cálculos realizar. El gráfico desplegado también ayuda a ilustrar la idea de

la información fluye hacia delante en el tiempo (calculando salidas y pérdidas) y hacia atrás en el tiempo (calculando gradientes) al mostrar explícitamente el camino a lo largo del cual fluye esta información.

10.2 Redes neuronales recurrentes

Armado con el desenrollado gráfico y el intercambio de parámetros ideas de la sección 10.1, podemos diseñar una amplia variedad de redes neuronales recurrentes.

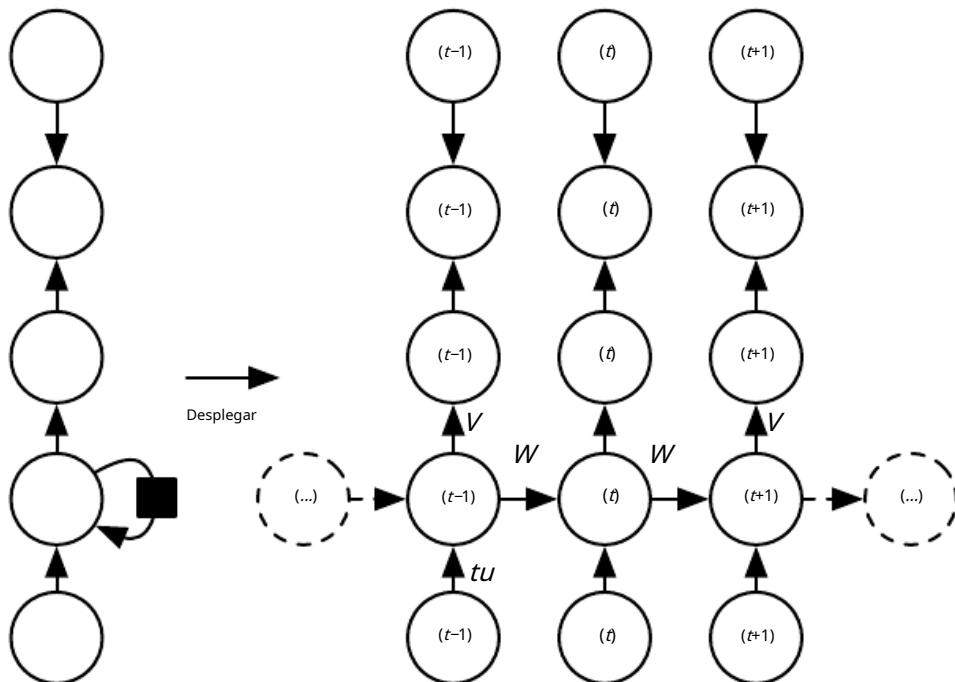


Figura 10.3: El gráfico computacional para calcular la pérdida de entrenamiento de una red recurrente que mapea una secuencia de entrada de x valores a una secuencia correspondiente de salida o valores. Una pérdida L mide qué tan lejos cada o es del objetivo de entrenamiento correspondiente y . Cuando usamos salidas softmax, asumimos o son las probabilidades logarítmicas no normalizadas. La pérdida L calcula internamente $\hat{y} = \text{softmax}(o)$ y lo compara con el objetivo y . El RNN tiene entrada a conexiones ocultas parametrizadas por una matriz de peso tu , conexiones recurrentes ocultas a ocultas parametrizadas por una matriz de peso W , y conexiones ocultas a salida parametrizadas por una matriz de peso V . Ecuación 10.8 define la propagación hacia adelante en este modelo. (Izquierda) La RNN y su pérdida dibujada con conexiones recurrentes. (Derecha) Lo mismo visto como un gráfico computacional desplegado en el tiempo, donde cada nodo ahora está asociado con una instancia de tiempo en particular.

Algunos ejemplos de patrones de diseño importantes para redes neuronales recurrentes incluyen los siguientes:

- Redes recurrentes que producen una salida en cada paso de tiempo y tienen conexiones recurrentes entre unidades ocultas, ilustradas en la figura 10.3.
- Redes recurrentes que producen una salida en cada paso de tiempo y tienen conexiones recurrentes solo desde la salida en un paso de tiempo hasta las unidades ocultas en el siguiente paso de tiempo, ilustradas en la figura 10.4
- Redes recurrentes con conexiones recurrentes entre unidades ocultas, que leen una secuencia completa y luego producen una sola salida, ilustradas en la figura 10.5.

cifra 10.3 es un ejemplo razonablemente representativo al que volveremos a lo largo de la mayor parte del capítulo.

La red neuronal recurrente de la figura 10.3 y ecuación 10.8 es universal en el sentido de que cualquier función computable por una máquina de Turing puede ser computada por tal red recurrente de tamaño finito. La salida se puede leer del RNN después de un número de pasos de tiempo que es asintóticamente lineal en el número de pasos de tiempo utilizados por la máquina de Turing y asintóticamente lineal en la longitud de la entrada (Siegelmann y Sontag, 1991; Siegelmann, 1995; Siegelmann y Sontag, 1995; Hyotyniemi, 1996). Las funciones computables por una máquina de Turing son discretas, por lo que estos resultados se refieren a la implementación exacta de la función, no a las aproximaciones. La RNN, cuando se usa como una máquina de Turing, toma una secuencia binaria como entrada y sus salidas deben discretizarse para proporcionar una salida binaria. Es posible calcular todas las funciones en esta configuración utilizando un único RNN específico de tamaño finito (Siegelmann y Sontag, 1995) utilizan 886 unidades). La “entrada” de la máquina de Turing es una especificación de la función a calcular, por lo que la misma red que simula esta máquina de Turing es suficiente para todos los problemas. El RNN teórico utilizado para la prueba puede simular una pila ilimitada al representar sus activaciones y pesos con números racionales de precisión ilimitada.

Ahora desarrollamos las ecuaciones de propagación directa para el RNN representado en la figura 10.3. La figura no especifica la elección de la función de activación para las unidades ocultas. Aquí asumimos la función de activación de la tangente hiperbólica. Además, la figura no especifica exactamente qué forma toman las funciones de salida y pérdida. Aquí asumimos que la salida es discreta, como si el RNN se usara para predecir palabras o caracteres. Una forma natural de representar variables discretas es considerar la salida o como dando las probabilidades logarítmicas no normalizadas de cada valor posible de la variable discreta. Luego podemos aplicar la operación softmax como un paso de procesamiento posterior para obtener un vector \hat{y} de probabilidades normalizadas sobre la salida. La propagación directa comienza con una especificación del estado inicial $h(0)$. Entonces, para cada paso de tiempo de

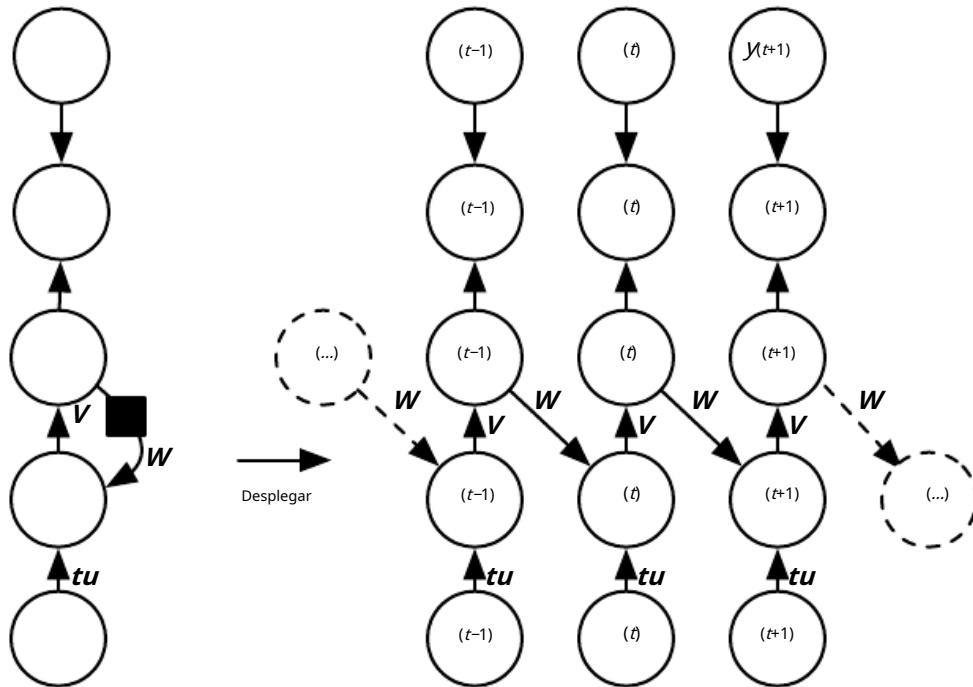


Figura 10.4: Una RNN cuya única recursión es la conexión de retroalimentación de la salida a la capa oculta. En cada paso de tiempo t , la entrada es x_t , las activaciones de la capa oculta son $h_{(t)}$, las salidas son $y_{(t)}$ y la perdida es $L_{(t)}$. (Izquierda) Diagrama de circuito. (Derecha) Gráfico computacional desplegado. Tal RNN es menos poderoso (puede expresar un conjunto más pequeño de funciones) que los de la familia representada por la figura 10.3. El RNN en figura 10.3 puede elegir poner cualquier información que quiera sobre el pasado en su representación oculta h y transmitir h al futuro. El RNN en esta figura está entrenado para poner un valor de salida específico en o , lo que es la única información que se le permite enviar al futuro. No hay conexiones directas de h avanzando. El anterior h está conectado con el presente solo indirectamente, a través de las predicciones que se utilizó para producir. A menos que o sea muy rico y de alta dimensión, por lo general carecerá de información importante del pasado. Esto hace que el RNN en esta figura sea menos poderoso, pero puede ser más fácil de entrenar porque cada paso de tiempo se puede entrenar de forma aislada de los demás, lo que permite una mayor paralelización durante el entrenamiento, como se describe en la sección 10.2.1.

$t=1$ a $t=\tau$, aplicamos las siguientes ecuaciones de actualización:

$$a(t) = b + \text{¿Qué?}_{t-1} + \text{experiencia}_t \quad (10.8)$$

$$h(t) = \tanh(a(t)) \quad (10.9)$$

$$o(t) = C + V h(t) \quad (10.10)$$

$$\hat{y}(t) = \text{softmax}(o(t)) \quad (10.11)$$

donde los parámetros son los vectores de sesgo b y C junto con las matrices de peso V y W , respectivamente para conexiones de entrada a oculto, de oculto a salida y de oculto a oculto. Este es un ejemplo de una red recurrente que asigna una secuencia de entrada a una secuencia de salida de la misma longitud. La pérdida total para una secuencia dada de X valores emparejados con una secuencia de y los valores serían entonces solo la suma de las pérdidas en todos los pasos de tiempo. Por ejemplo, si $L(t)$ es el negativo de la probabilidad logarítmica de $y(t)$ dado $X(1), \dots, X(t)$, entonces

$$L = -\sum_{t=1}^T \{X(1), \dots, X(t)\}, \{y(1), \dots, y(t)\} \quad (10.12)$$

$$= -\sum_{t=1}^T L(t) \quad (10.13)$$

$$= -\sum_{t=1}^T \underset{\text{registro } pag_{\text{modelo}}}{y(t)} \{X^{(1)}, \dots, X^{(t)}\}, \quad (10.14)$$

dónde $pag_{\text{modelo}}(t) / \{X(1), \dots, X(t)\}$ se obtiene leyendo la entrada para $y(t)$ desde el vector de salida del modelo $\hat{y}(t)$. Calcular el gradiente de esta función de pérdida con respecto a los parámetros es una operación costosa. El cálculo del gradiente implica realizar un pase de propagación hacia adelante moviéndose de izquierda a derecha a través de nuestra ilustración del gráfico desenrollado en la figura 10.3, seguido de un pase de propagación hacia atrás que se mueve de derecha a izquierda a través del gráfico. El tiempo de ejecución es $O(\tau)$ y no puede reducirse mediante paralelización porque el gráfico de propagación directa es inherentemente secuencial; cada paso de tiempo solo puede calcularse después del anterior. Los estados calculados en el paso hacia adelante deben almacenarse hasta que se reutilicen durante el paso hacia atrás, por lo que el costo de la memoria también es $O(\tau)$. El algoritmo de retropropagación aplicado al gráfico desenrollado con $O(\tau)$ el costo se llama **retropropagación a través del tiempo** o BPTT y se analiza más adelante en la sección 10.2.2. La red con recurrencia entre unidades ocultas es, por tanto, muy potente pero también costosa de entrenar. ¿Hay alguna alternativa?

10.2.1 Docentes forzados y redes con recurrencia de salida

La red con conexiones recurrentes solo desde la salida en un paso de tiempo a las unidades ocultas en el siguiente paso de tiempo (que se muestra en la figura 10.4) es estrictamente menos potente

porque carece de conexiones recurrentes ocultas a ocultas. Por ejemplo, no puede simular una máquina de Turing universal. Debido a que esta red carece de recurrencia de oculto a oculto, requiere que las unidades de salida capturen toda la información sobre el pasado que la red utilizará para predecir el futuro. Debido a que las unidades de salida se entrena explícitamente para que coincidan con los objetivos del conjunto de entrenamiento, es poco probable que capturen la información necesaria sobre el historial pasado de la entrada, a menos que el usuario sepa cómo describir el estado completo del sistema y lo proporcione como parte del proceso. Objetivos establecidos de entrenamiento. La ventaja de eliminar la recurrencia de oculto a oculto es que, para cualquier función de pérdida basada en la comparación de la predicción en el tiempo al objetivo de entrenamiento en el momento t , todos los pasos de tiempo están desacoplados. De este modo, el entrenamiento se puede paralelizar, con el gradiente para cada paso t calculado de forma aislada. No es necesario calcular primero la salida del paso de tiempo anterior, porque el conjunto de entrenamiento proporciona el valor ideal de esa salida.

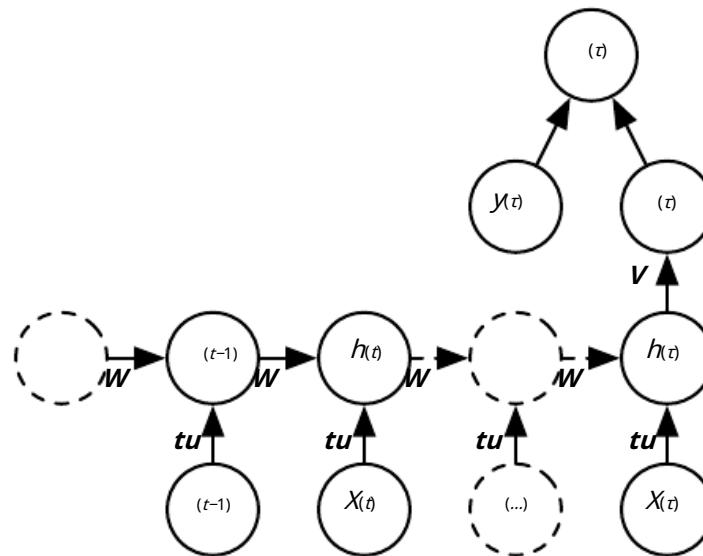


Figura 10.5: Red neuronal recurrente desplegada en el tiempo con una única salida al final de la secuencia. Dicha red se puede usar para resumir una secuencia y producir una representación de tamaño fijo que se usa como entrada para un procesamiento posterior. Puede haber un objetivo justo al final (como se muestra aquí) o el gradiente en la salida α_{t+1} se puede obtener retropropagando desde más módulos aguas abajo.

Los modelos que tienen conexiones recurrentes desde sus salidas que conducen al modelo pueden entrenarse con **profesor forzando**. El forzado del profesor es un procedimiento que surge del criterio de máxima verosimilitud, en el que durante el entrenamiento el modelo recibe la salida de verdad del terreno $y(t)$ como entrada en el momento $t+1$. Podemos ver esto al examinar una secuencia con dos pasos de tiempo. El máximo condicional

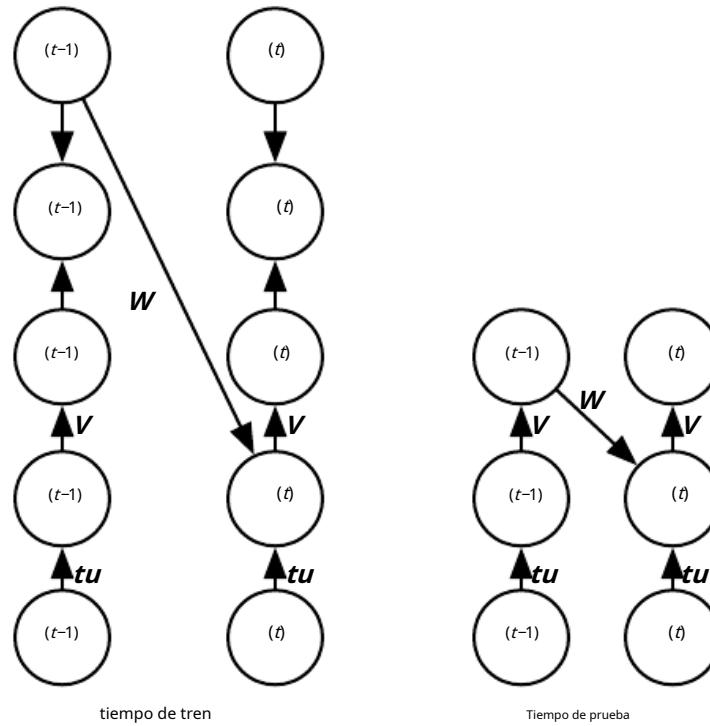


Figura 10.6: Ilustración de la fuerza del maestro. El forzado de maestros es una técnica de entrenamiento que se aplica a RNN que tienen conexiones desde su salida a sus estados ocultos en el siguiente paso de tiempo. (Izquierda) A la hora del tren, damos de comer alsalida correctay_textraído del conjunto de trenes como entrada para $h_{(t+1)}$. (Bien) Cuando se implementa el modelo, generalmente no se conoce el verdadero resultado. En este caso, aproximamos la salida correctay_tcon la salida del modelo o_t y vuelva a introducir la salida en el modelo.

criterio de probabilidad es

$$\underset{\text{registropag}}{-} \mathcal{Y}(1), \mathcal{Y}(2) / X(1), X(2) \quad (10.15)$$

$$\underset{= \text{registropag}}{-} \mathcal{Y}(2) / \mathcal{Y}(1), X(1), X(2) \quad + \underset{\text{registropag}}{-} \mathcal{Y}(1) / X(1), X(2) \quad (10.16)$$

En este ejemplo, vemos que en el momento $t=2$, el modelo está entrenado para maximizar la probabilidad condicional de $\mathcal{Y}(2)$ dado *ambos* el X secuencia hasta ahora y la anterior valor del conjunto de entrenamiento. Por lo tanto, la máxima verosimilitud específica que durante el entrenamiento, en lugar de retroalimentar la propia salida del modelo, estas conexiones deben alimentarse con los valores objetivo que especifican cuál debe ser la salida correcta. Esto se ilustra en la figura 10.6.

Originalmente, motivamos el forzamiento del profesor para que nos permitiera evitar la propagación hacia atrás a través del tiempo en modelos que carecen de conexiones de oculto a oculto. El forzado del profesor aún se puede aplicar a los modelos que tienen conexiones de oculto a oculto, siempre que tengan conexiones desde la salida en un paso de tiempo a los valores calculados en el siguiente paso de tiempo. Sin embargo, tan pronto como las unidades ocultas se vuelven una función de pasos de tiempo anteriores, el algoritmo BPTT es necesario. Por lo tanto, algunos modelos pueden ser entrenados tanto con el forzado del maestro como con el BPTT.

La desventaja de la obligatoriedad estricta del profesor surge si la red se va a utilizar posteriormente en un **lazo abierto** modo, con las salidas de la red (o muestras de la distribución de salida) retroalimentadas como entrada. En este caso, el tipo de entradas que ve la red durante el entrenamiento podría ser bastante diferente del tipo de entradas que verá en el momento de la prueba. Una forma de mitigar este problema es entrenar tanto con entradas forzadas por el profesor como con entradas de ejecución libre, por ejemplo, prediciendo el objetivo correcto una serie de pasos en el futuro a través de las rutas recurrentes de salida a entrada desplegadas. De esta manera, la red puede aprender a tener en cuenta las condiciones de entrada (como las que genera en el modo de ejecución libre) que no se ven durante el entrenamiento y cómo mapear el estado hacia uno que hará que la red genere salidas adecuadas después. unos pocos pasos Otro enfoque ([bengio et al., 2015b](#)) para mitigar la brecha entre las entradas vistas en el momento del tren y las entradas vistas en el momento de la prueba elige aleatoriamente usar valores generados o valores de datos reales como entrada. Este enfoque explota una estrategia de aprendizaje del plan de estudios para utilizar gradualmente más valores generados como entrada.

10.2.2 Cálculo del gradiente en una red neuronal recurrente

Calcular el gradiente a través de una red neuronal recurrente es sencillo. Uno simplemente aplica el algoritmo generalizado de propagación hacia atrás de la sección [6.5.6](#)

al gráfico computacional desenrollado. No se necesitan algoritmos especializados. Los gradientes obtenidos por retropropagación pueden usarse con cualquier técnica basada en gradientes de uso general para entrenar una RNN.

Para ganar algo de intuición sobre cómo se comporta el algoritmo BPTT, proporcionamos un ejemplo de cómo calcular gradientes por BPTT para las ecuaciones RNN anteriores (ecuación 10.8 y ecuación 10.12). Los nodos de nuestro gráfico computacional incluyen los parámetros t, V, W, b y α así como la secuencia de nodos indexados por t para $X(t), h(t), \alpha(t)$ y $L(t)$. Para cada nodo α tenemos que calcular el gradiente $\nabla_{\alpha(t)} L$ recursivamente, en función del gradiente calculado en los nodos que lo siguen en el gráfico. Comenzamos la recursión con los nodos inmediatamente anteriores a la pérdida final

$$\frac{\partial L}{\partial L(t)} = 1. \quad (10.17)$$

En esta derivación suponemos que las salidas $\alpha(t)$ se utilizan como argumento de la función softmax para obtener el vector \hat{y} de probabilidades sobre la salida. También asumimos que la pérdida es la probabilidad logarítmica negativa del verdadero objetivo $y(t)$ dada la entrada hasta ahora. El gradiente $\nabla_{\alpha(t)} L$ en las salidas en el paso de tiempo t , para todos ϵ , es como sigue:

$$(\nabla_{\alpha(t)} L) = \frac{\partial L}{\partial \alpha(t)} = \frac{\partial L}{\partial L(t)} \frac{\partial L(t)}{\partial \alpha(t)} = \hat{y}(t) - 1_{y_0, y(t)}. \quad (10.18)$$

Trabajamos nuestro camino hacia atrás, comenzando desde el final de la secuencia. En el paso de tiempo final t , $h(t)$ solo tiene $\alpha(t)$ como descendiente, por lo que su gradiente es simple:

$$\nabla_{h(t)} L = V \nabla_{\alpha(t)} / \quad (10.19)$$

Luego podemos iterar hacia atrás en el tiempo para propagar hacia atrás los gradientes a través del tiempo, desde $t=\tau-1$ abajo a $t=1$, señalando que $h(t)$ (para $t < \tau$) tiene como descendientes a ambos $\alpha(t)$ y $h(t+1)$. Su gradiente está dado por lo tanto por

$$\nabla_{h(t)} L = \frac{\partial h(t+1)}{\partial h(t)} (\nabla_{h(t+1)} L) + \frac{\partial \alpha(t)}{\partial h(t)} (\nabla_{\alpha(t)} L) \quad (10.20)$$

$$= W(\nabla_{h(t+1)} L) \text{ diag. } 1 - h(t+1)^{-2} + V(\nabla_{\alpha(t)} L) \quad (10.21)$$

Dónde $\text{diag. } 1 - h(t+1)^{-2}$ indica la matriz diagonal que contiene los elementos $1 - (h(t+1)_i^{-2})$. Este es el jacobiano de la tangente hiperbólica asociada con la unidad oculta i en el momento $t+1$.