

CAPÍTULO 6. REDES PROFUNDAS DE FEEDFORWARD

la fracción de conteos de cada resultado observado en el conjunto de entrenamiento:

$$\text{softmax}(z(X; \theta))_i \approx \frac{\sum_{j=1}^m 1_{Y(j)=i, X(j)=X}}{\sum_{j=1}^m 1_{X(j)=X}}. \quad (6.31)$$

Debido a que la máxima verosimilitud es un estimador consistente, esto está garantizado siempre que la familia del modelo sea capaz de representar la distribución de entrenamiento. En la práctica, la capacidad limitada del modelo y la optimización imperfecta significarán que el modelo solo puede aproximar estas fracciones.

Muchas funciones objetivas distintas de log-verosimilitud no funcionan tan bien con la función softmax. Específicamente, las funciones objetivo que no usan un registro para deshacer el Exp del softmax fallan en aprender cuando el argumento del Exp se vuelve muy negativo, haciendo que el gradiente desaparezca. En particular, el error cuadrático es una función de pérdida deficiente para las unidades softmax y puede fallar al entrenar el modelo para cambiar su salida, incluso cuando el modelo hace predicciones incorrectas altamente confiables (Brida, 1990). Para comprender por qué estas otras funciones de pérdida pueden fallar, debemos examinar la propia función softmax.

Al igual que el sigmoide, la activación de softmax puede saturarse. La función sigmoidea tiene una sola salida que se satura cuando su entrada es extremadamente negativa o extremadamente positiva. En el caso del softmax, hay múltiples valores de salida. Estos valores de salida pueden saturarse cuando las diferencias entre los valores de entrada se vuelven extremas. Cuando el softmax se satura, muchas funciones de costo basadas en el softmax también se saturan, a menos que puedan invertir la función de activación de saturación.

Para ver que la función softmax responde a la diferencia entre sus entradas, observe que la salida softmax es invariable al agregar el mismo escalar a todas sus entradas:

$$\text{softmax}(z) = \text{softmax}(z + C). \quad (6.32)$$

Usando esta propiedad, podemos derivar una variante numéricamente estable del softmax:

$$\text{softmax}(z) = \text{softmax}(z - \max_i z_i). \quad (6.33)$$

La versión reformulada nos permite evaluar softmax con solo pequeños errores numéricos incluso cuando z contiene números extremadamente grandes o extremadamente negativos. Al examinar la variante numéricamente estable, vemos que la función softmax está impulsada por la cantidad de desviación de sus argumentos. $\max_i z_i$.

una salida softmax(z) se satura a 1 cuando la entrada correspondiente es máxima ($z = \max_i z_i$) y es mucho mayor que todas las otras entradas. La salida softmax(z) también puede saturar a 0 cuando z no es máximo y el máximo es mucho mayor. Esta es una generalización de la forma en que se saturan las unidades sigmoideas, y

puede causar dificultades similares para el aprendizaje si la función de pérdida no está diseñada para compensarlo.

El argumentoza la función softmax se puede producir de dos maneras diferentes. El más común es simplemente tener una capa anterior de la red neuronal que genere cada elemento dez, como se describe arriba usando la capa lineal $z = W \cdot h + b$. Si bien es sencillo, este enfoque en realidad sobreparametriza la distribución. La restricción de que el norte las salidas deben sumar 1 significa que solo $n-1$ los parámetros son necesarios; la probabilidad de la n -ésimo valor puede obtenerse restando el primer $n-1$ probabilidades de 1. Por lo tanto, podemos imponer un requisito de que un elemento de z sea arreglado. Por ejemplo, podemos exigir que $z_{n-1} = 0$. De hecho, esto es exactamente lo que hace la unidad sigmoidea. Definición $PAG(y=1/X) = \sigma(z)$ es equivalente a definir $PAG(y=1/X) = \text{softmax}(z)$ con dos dimensiones $z_{n-1} = 0$. Ambos norte -1 argumento y el norte Los enfoques de argumento del softmax pueden describir el mismo conjunto de distribuciones de probabilidad, pero tienen diferentes dinámicas de aprendizaje. En la práctica, rara vez hay mucha diferencia entre usar la versión sobreparametrizada o la versión restringida, y es más sencillo implementar la versión sobreparametrizada.

Desde un punto de vista neurocientífico, es interesante pensar en el softmax como una forma de crear una forma de competencia entre las unidades que participan en él: las salidas del softmax siempre suman 1, por lo que corresponde necesariamente un aumento en el valor de una unidad a una disminución en el valor de los demás. Esto es análogo a la inhibición lateral que se cree que existe entre las neuronas cercanas en la corteza. En el extremo (cuando la diferencia entre el máximo y los otros es grande en magnitud) se convierte en una forma de **El ganador lo toma todo** (una de las salidas es casi 1 y las otras son casi 0).

El nombre "softmax" puede ser algo confuso. La función está más estrechamente relacionada con la función máxima que la función max. El término "suave" se deriva del hecho de que la función softmax es continua y diferenciable. El argumento máximo La función, con su resultado representado como un vector caliente, no es continua ni diferenciable. La función softmax proporciona así una versión "suavizada" de la función máxima. La versión suave correspondiente de la función máxima es $\text{softmax}(z) = z$. Quizás sería mejor llamar a la función softmax "softargmax", pero el nombre actual es una convención arraigada.

6.2.2.4 Otros tipos de salida

Las unidades de salida lineal, sigmoidea y softmax descritas anteriormente son las más comunes. Las redes neuronales pueden generalizarse a casi cualquier tipo de capa de salida que deseemos. El principio de máxima verosimilitud proporciona una guía sobre cómo diseñar

una buena función de costo para casi cualquier tipo de capa de salida.

En general, si definimos una distribución condicional $p(y | X; \theta)$, el principio de máxima verosimilitud sugiere que utilicemos $-\text{registro} p(y | X; \theta)$ como nuestra función de costo.

En general, podemos pensar que la red neuronal representa una función $R(X; \theta)$. Las salidas de esta función no son predicciones directas del valor y . En cambio, $R(X; \theta) = \omega$ proporciona los parámetros para una distribución sobre y . Nuestra función de pérdida se puede interpretar como $-\text{registro} p(y; \omega(X))$.

Por ejemplo, podemos desear conocer la varianza de una Gaussiana condicional para y , dado X . En el caso simple, donde la varianza σ^2 es una constante, hay una expresión de forma cerrada porque el estimador de varianza de máxima verosimilitud es simplemente la media empírica de la diferencia al cuadrado entre las observaciones y su valor esperado. Un enfoque computacionalmente más costoso que no requiere escribir código de casos especiales es simplemente incluir la varianza como una de las propiedades de la distribución $p(y | X)$ que es controlada por $\omega = R(X; \theta)$. La log-verosimilitud negativa $-\text{registro} p(y; \omega(X))$ luego proporcionará una función de costo con los términos apropiados necesarios para que nuestro procedimiento de optimización aprenda la varianza de manera incremental. En el caso simple donde la desviación estándar no depende de la entrada, podemos crear un nuevo parámetro en la red que se copia directamente en ω . Este nuevo parámetro podría ser σ^2 mismo o podría ser un parámetro ν representando σ^2 o podría ser un parámetro β representando ν^{-1} , dependiendo de cómo elijamos parametrizar la distribución. Podemos desear que nuestro modelo prediga una cantidad diferente de varianza en y para diferentes valores de X . Esto se llama un **heteroscedástico** modelo. En el caso heteroscedástico, simplemente hacemos que la especificación de la varianza sea uno de los valores de salida por $R(X; \theta)$. Una forma típica de hacer esto es formular la distribución gaussiana usando precisión, en lugar de varianza, como se describe en la ecuación 3.22. En el caso multivariado es más común usar una matriz de precisión diagonal

$$\text{diag}(\beta). \quad (6.34)$$

Esta formulación funciona bien con gradiente descendente porque la fórmula para el logaritmo de la probabilidad de la distribución gaussiana parametrizada por β implica sólo la multiplicación por β y adición del registro β . El gradiente de las operaciones de multiplicación, suma y logaritmo se comporta bien. En comparación, si parametrizáramos la salida en términos de varianza, necesitaríamos usar la división. La función de división se vuelve arbitrariamente empinada cerca de cero. Si bien los gradientes grandes pueden ayudar al aprendizaje, los gradientes arbitrariamente grandes generalmente provocan inestabilidad. Si parametrizáramos la salida en términos de desviación estándar, el logaritmo de verosimilitud aún implicaría división y también elevaría al cuadrado. El gradiente a través de la operación de elevación al cuadrado puede desvanecerse cerca de cero, lo que dificulta el aprendizaje de parámetros elevados al cuadrado.

Independientemente de si usamos la desviación estándar, la varianza o la precisión, debemos asegurarnos de que la matriz de covarianza de la Gaussiana sea definida positiva. Dado que los valores propios de la matriz de precisión son los recíprocos de los valores propios de la matriz de covarianza, esto equivale a garantizar que la matriz de precisión sea definida positiva. Si usamos una matriz diagonal, o un escalar multiplicado por la matriz diagonal, entonces la única condición que debemos aplicar en la salida del modelo es la positividad. Si suponemos que σ es la activación bruta del modelo utilizado para determinar la precisión diagonal, podemos usar la función softplus para obtener un vector de precisión positivo: $\beta = \zeta(\sigma)$. Esta misma estrategia se aplica por igual si se usa la varianza o la desviación estándar en lugar de la precisión o si se usa una identidad de tiempos escalares en lugar de una matriz diagonal.

Es raro aprender una matriz de covarianza o precisión con una estructura más rica que la diagonal. Si la covarianza es total y condicional, se debe elegir una parametrización que garantice la definición positiva de la matriz de covarianza predicha. Esto se puede lograr escribiendo $\Sigma(X) = B(X)B(X)$, donde B es una matriz cuadrada sin restricciones. Un problema práctico si la matriz es de rango completo es que calcular la probabilidad es costoso, con una $d \times d$ matriz que requiere $O(d^3)$ cálculo para el determinante y el inverso de $\Sigma(X)$ (o de manera equivalente, y más comúnmente hecho, su autodescomposición o la de $B(X)$).

A menudo queremos realizar una regresión multimodal, es decir, predecir valores reales que provienen de una distribución condicional $pag(y | X)$ que puede tener varios picos diferentes en el espacio por el mismo valor de X . En este caso, una mezcla gaussiana es una representación natural de la salida ([Jacobset al., 1991; obispo, 1994](#)). Las redes neuronales con mezclas gaussianas como su salida a menudo se denominan **redes de densidad de mezcla**. Una salida de mezcla gaussiana con n componentes se define por la distribución de probabilidad condicional

$$pag(y | X) = \sum_{i=1}^{n} pag(c = yo | X) norte(y; \mu(i)(X), \Sigma(i)(X)). \quad (6.35)$$

La red neuronal debe tener tres salidas: un vector que define $pag(c = yo | X)$, una matriz que proporciona $\mu(i)(X)$ para todos i , y un tensor que proporciona $\Sigma(i)(X)$ para todos i . Estas salidas deben satisfacer diferentes restricciones:

1. Componentes de la mezcla $pag(c = yo | X)$: estos forman una distribución multinomial sobre el n diferentes componentes asociados con la variable latente ¹ c , y puede

¹Consideramos que c está latente porque no lo observamos en los datos: entrada dada X y objetivo y , no es posible saber con certeza qué componente gaussiano fue el responsable de y , pero podemos imaginar que se generó eligiendo uno de ellos, y convertir esa elección no observada en una variable aleatoria.

típicamente ser obtenido por un softmax sobre un n -vector dimensional, para garantizar que estas salidas sean positivas y sumen 1.

2. Medios $\mu_i(\mathcal{X})$: indican el centro o medio asociado a la i -th componente gaussiana, y no tienen restricciones (típicamente sin ninguna no linealidad para estas unidades de salida). Si es un d -vector, entonces la red debe generar un $n \times d$ matriz que contiene todos los d -vectores dimensionales. Aprender estas medias con la máxima verosimilitud es un poco más complicado que aprender las medias de una distribución con un solo modo de salida. Solo queremos actualizar la media del componente que realmente produjo la observación. En la práctica, no sabemos qué componente produjo cada observación. La expresión para la probabilidad logarítmica negativa pondera naturalmente la contribución de cada ejemplo a la pérdida de cada componente por la probabilidad de que el componente haya producido el ejemplo.
3. Covarianzas $\Sigma_i(\mathcal{X})$: estos especifican la matriz de covarianza para cada componente i . Al igual que cuando aprendemos un solo componente gaussiano, normalmente usamos una matriz diagonal para evitar la necesidad de calcular los determinantes. Al igual que con el aprendizaje de los medios de la mezcla, la máxima verosimilitud se complica por la necesidad de asignar responsabilidad parcial para cada punto a cada componente de la mezcla. El descenso de gradiente seguirá automáticamente el proceso correcto si se le da la especificación correcta del logaritmo de verosimilitud negativo en el modelo de mezcla.

Se ha informado que la optimización basada en gradientes de mezclas gaussianas condicionales (en la salida de redes neuronales) puede no ser confiable, en parte porque se obtienen divisiones (por la varianza) que pueden ser numéricamente inestables (cuando alguna varianza llega a ser pequeña para un ejemplo particular, produciendo gradientes muy grandes). Una solución es **degradados de clip** (mira la sección 10.11.1) mientras que otro es escalar los gradientes heurísticamente ([Murray y Larochelle, 2014](#)).

Las salidas de mezcla gaussiana son particularmente efectivas en modelos generativos de voz ([Schuster, 1999](#)) o movimientos de objetos físicos ([Tumbas, 2013](#)). La estrategia de densidad de mezcla proporciona una forma para que la red represente múltiples modos de salida y controle la varianza de su salida, lo cual es crucial para obtener un alto grado de calidad en estos dominios de valor real. En la figura se muestra un ejemplo de una red de densidad mixta.[6.4](#).

En general, es posible que deseemos continuar modelando vectores más grandes y que contiene más variables, e imponer estructuras cada vez más ricas en estas variables de salida. Por ejemplo, podemos desear que nuestra red neuronal genere una secuencia de caracteres que forme una oración. En estos casos, podemos seguir utilizando el principio de máxima verosimilitud aplicado a nuestro modelo. $p_{\text{ag}}(y; \omega(\mathcal{X}))$, pero el modelo que usamos

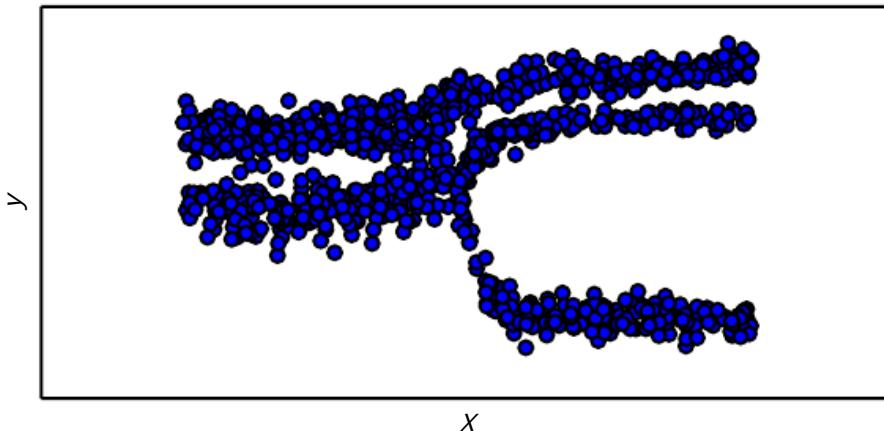


Figura 6.4: Muestras extraídas de una red neuronal con una capa de salida de densidad mixta. la entrada X se muestrea de una distribución uniforme y la salida y se muestrea de *pag* modelo (y / X). La red neuronal puede aprender mapeos no lineales desde la entrada hasta los parámetros de la distribución de salida. Estos parámetros incluyen las probabilidades que rigen cuál de los tres componentes de la mezcla generará la salida, así como los parámetros para cada componente de la mezcla. Cada componente de la mezcla es gaussiano con media y varianza predichas. Todos estos aspectos de la distribución de salida pueden variar con respecto a la entrada X , y hacerlo de manera no lineal.

para describir y se vuelve lo suficientemente complejo como para estar más allá del alcance de este capítulo. Capítulo 10 describe cómo usar redes neuronales recurrentes para definir tales modelos sobre secuencias, y parte tercero describe técnicas avanzadas para modelar distribuciones de probabilidad arbitrarias.

6.3 Unidades Ocultas

Hasta ahora, hemos centrado nuestra discusión en las opciones de diseño para redes neuronales que son comunes a la mayoría de los modelos de aprendizaje automático paramétrico entrenados con optimización basada en gradientes. Ahora pasamos a un problema que es exclusivo de las redes neuronales feedforward: cómo elegir el tipo de unidad oculta para usar en las capas ocultas del modelo.

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

Las unidades lineales rectificadas son una excelente opción por defecto de unidad oculta. Hay muchos otros tipos de unidades ocultas disponibles. Puede ser difícil determinar cuándo usar qué tipo (aunque las unidades lineales rectificadas suelen ser una opción aceptable). Nosotros

describa aquí algunas de las intuiciones básicas que motivan cada tipo de unidades ocultas. Estas intuiciones pueden ayudar a decidir cuándo probar cada una de estas unidades. Por lo general, es imposible predecir de antemano cuál funcionará mejor. El proceso de diseño consiste en prueba y error, intuir que un tipo de unidad oculta puede funcionar bien y luego entrenar una red con ese tipo de unidad oculta y evaluar su desempeño en un conjunto de validación.

Algunas de las unidades ocultas incluidas en esta lista no son realmente diferenciables en todos los puntos de entrada. Por ejemplo, la función lineal rectificada $g_{\text{ramo}}(z) = \max\{0, z\}$ no es diferenciable en $z=0$. Esto puede parecer que invalida g_{ramo} para usar con un algoritmo de aprendizaje basado en gradientes. En la práctica, el descenso de gradiente aún funciona lo suficientemente bien como para que estos modelos se utilicen para tareas de aprendizaje automático. Esto se debe en parte a que los algoritmos de entrenamiento de redes neuronales generalmente no llegan a un mínimo local de la función de costo, sino que simplemente reducen su valor de manera significativa, como se muestra en la figura 4.3. Estas ideas se describirán más adelante en el capítulo 8. Porque no esperamos que el entrenamiento llegue realmente a un punto donde el gradiente sea 0, es aceptable que los mínimos de la función de costo correspondan a puntos con pendiente indefinida. Las unidades ocultas que no son diferenciables generalmente no son diferenciables en solo una pequeña cantidad de puntos. En general, una función $g_{\text{ramo}}(z)$ tiene una derivada izquierda definida por la pendiente de la función inmediatamente a la izquierda de z y una derivada por la derecha definida por la pendiente de la función inmediatamente a la derecha de z . Una función es derivable en z solo si tanto la derivada izquierda como la derivada derecha están definidas y son iguales entre sí. Las funciones utilizadas en el contexto de las redes neuronales suelen tener derivadas izquierdas definidas y derivadas derechas definidas. En el caso de $g_{\text{ramo}}(z) = \max\{0, z\}$, la derivada por la izquierda en $z=0$ es 0 y la derivada derecha es 1. Las implementaciones de software de entrenamiento de redes neuronales generalmente devuelven uno de los derivados unilaterales en lugar de informar que el derivado no está definido o genera un error. Esto puede justificarse heurísticamente al observar que la optimización basada en gradientes en una computadora digital está sujeta a errores numéricos de todos modos. Cuando se le pide a una función que evalúe $g_{\text{ramo}}(0)$, es muy poco probable que el valor subyacente realmente fuera 0. En cambio, era probable que fuera algún pequeño valor - que fue redondeado a 0. En algunos contextos, se encuentran disponibles justificaciones más agradables teóricamente, pero generalmente no se aplican al entrenamiento de redes neuronales. El punto importante es que en la práctica uno puede ignorar con seguridad la no diferenciabilidad de las funciones de activación de unidades ocultas que se describen a continuación.

A menos que se indique lo contrario, la mayoría de las unidades ocultas pueden describirse como aceptando un vector de entradas X , calculando una transformación afín $z = W \cdot X + b$, y luego aplicando una función no lineal por elementos $g_{\text{ramo}}(z)$. La mayoría de las unidades ocultas se distinguen entre sí solo por la elección de la forma de la función de activación $g_{\text{ramo}}(z)$.

6.3.1 Unidades lineales rectificadas y sus generalizaciones

Las unidades lineales rectificadas utilizan la función de activación $gramo(z) = \max\{0, z\}$.

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

Las unidades lineales rectificadas se utilizan normalmente sobre una transformación afín:

$$h = gromo(W \cdot X + b). \quad (6.36)$$

Al inicializar los parámetros de la transformación afín, puede ser una buena práctica establecer todos los elementos de b a un valor pequeño y positivo, como 0.1. Esto hace que sea muy probable que las unidades lineales rectificadas estén inicialmente activas para la mayoría de las entradas en el conjunto de entrenamiento y permitan el paso de las derivadas.

Existen varias generalizaciones de unidades lineales rectificadas. La mayoría de estas generalizaciones funcionan de manera comparable a las unidades lineales rectificadas y, en ocasiones, funcionan mejor.

Un inconveniente de las unidades lineales rectificadas es que no pueden aprender a través de métodos basados en gradientes en ejemplos para los que su activación es cero. Una variedad de generalizaciones de unidades lineales rectificadas garantizan que reciben gradiente en todas partes.

Tres generalizaciones de unidades lineales rectificadas se basan en el uso de una pendiente distinta de cero a $z < 0$: $h = gromo(z, a) = \max(0, z) + a \min(0, z)$. **Rectificación de valor absoluto** fixa $a = -1$ para obtener $gramo(z) = |z|$. Se utiliza para el reconocimiento de objetos a partir de imágenes (Jarrett et al., 2009), donde tiene sentido buscar características que sean invariantes bajo una inversión de polaridad de la iluminación de entrada. Otras generalizaciones de unidades lineales rectificadas son de aplicación más amplia. **AReLU con fugas** (Mosa et al., 2013) corrige a a un valor pequeño como 0.01 mientras que un **ReLU paramétrico** o **PReLU** golosinas a como un parámetro aprendible (El et al., 2015).

Unidades máximas (Buen compaño et al., 2013a) generalizan aún más las unidades lineales rectificadas. En lugar de aplicar una función por elementos $gramo(z)$, maxout unidades dividirán en grupos de k valores. Cada unidad maxout genera el elemento máximo de

uno de estos grupos:

$$\text{gramo}(z) = \max_{j \in \text{GRAMO}(i)} z_j \quad (6.37)$$

dónde $\text{GRAMO}(i)$ es el conjunto de índices en las entradas para el grupo $i, \{(y_0 - 1)k + 1, \dots, ik\}$. Esto proporciona una forma de aprender una función lineal por partes que responde a múltiples direcciones en la entrada x espacio.

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

Cada unidad maxout ahora está parametrizada por k vectores de peso en lugar de uno solo, por lo que las unidades maxout normalmente necesitan más regularización que las unidades lineales rectificadas. Pueden funcionar bien sin regularización si el conjunto de entrenamiento es grande y el número de piezas por unidad se mantiene bajo ([Cai et al., 2013](#)).

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

Debido a que cada unidad está impulsada por múltiples filtros, las unidades maxout tienen cierta redundancia que las ayuda a resistir un fenómeno llamado **olvido catastrófico** en el que las redes neuronales olvidan cómo realizar tareas en las que fueron entrenadas en el pasado ([Buen compañero et al., 2014a](#)).

Las unidades lineales rectificadas y todas estas generalizaciones de las mismas se basan en el principio de que los modelos son más fáciles de optimizar si su comportamiento es más cercano al lineal. Este mismo principio general de utilizar el comportamiento lineal para obtener una optimización más sencilla también se aplica en otros contextos además de las redes lineales profundas. Las redes recurrentes pueden aprender de secuencias y producir una secuencia de estados y salidas. Al entrenarlos, uno necesita propagar la información a través de varios pasos de tiempo, lo cual es mucho más fácil cuando se involucran algunos cálculos lineales (con algunas derivadas direccionales de magnitud cercana a 1). Una de las redes recurrentes de mejor rendimiento

arquitecturas, el LSTM, propaga información a través del tiempo a través de la suma, un tipo particular y directo de tal activación lineal. Esto se discute más adelante en la sección 10.10.

6.3.2 Tangente sigmoidea e hiperbólica logística

Antes de la introducción de las unidades lineales rectificadas, la mayoría de las redes neuronales utilizaban la función logística de activación sigmoidea.

$$g_{\text{sigmo}}(z) = \sigma(z) \quad (6.38)$$

o la función de activación de la tangente hiperbólica

$$g_{\text{tanh}}(z) = \tanh(z). \quad (6.39)$$

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

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

Cuando se debe usar una función de activación sigmoidea, la función de activación de tangente hiperbólica generalmente se desempeña mejor que la sigmoidea logística. Se parece más a la función identidad, en el sentido de que $\tanh(0) = 0$ mientras $\sigma(0) = 1$.² Porque $\tanh(z)$ es similar a la función identidad cercana a 0, entrenar una red neuronal profunda $\hat{y} = w \cdot \tanh(tu \cdot \tanh(V \cdot X))$ se asemeja a la formación de un modelo lineal $\hat{y} = w \cdot tu \cdot V \cdot X$ siempre que las activaciones de la red puedan mantenerse pequeñas. Esto hace que el entrenamiento sea más fácil.

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

6.3.3 Otras unidades ocultas

Son posibles muchos otros tipos de unidades ocultas, pero se usan con menos frecuencia.

En general, una amplia variedad de funciones diferenciables funcionan perfectamente bien. Muchas funciones de activación no publicadas funcionan tan bien como las populares. Para proporcionar un ejemplo concreto, los autores probaron una red feedforward utilizando $h = \text{porque Ancho } x + b$ en el conjunto de datos MNIST y obtuvo una tasa de error de menos del 1%, que es competitiva con los resultados obtenidos utilizando funciones de activación más convencionales. Durante la investigación y el desarrollo de nuevas técnicas, es común probar muchas funciones de activación diferentes y encontrar que varias variaciones en la práctica estándar funcionan de manera comparable. Esto significa que, por lo general, los nuevos tipos de unidades ocultas se publican solo si se demuestra claramente que proporcionan una mejora significativa. Los nuevos tipos de unidades ocultas que funcionan de manera más o menos comparable a los tipos conocidos son tan comunes que no son interesantes.

No sería práctico enumerar todos los tipos de unidades ocultas que han aparecido en la literatura. Destacamos algunos especialmente útiles y distintivos.

Una posibilidad es no tener una activación $\text{gramo}(z)$ en absoluto. También se puede pensar en esto como el uso de la función de identidad como función de activación. Ya hemos visto que una unidad lineal puede ser útil como salida de una red neuronal. También se puede utilizar como unidad oculta. Si cada capa de la red neuronal consta solo de transformaciones lineales, entonces la red en su conjunto será lineal. Sin embargo, es aceptable que algunas capas de la red neuronal sean puramente lineales. Considere una capa de red neuronal con n entradas y p salidas, $h = \text{gramo}(W \cdot X + b)$. Podemos reemplazar esto con dos capas, con una capa usando matriz de peso t y el otro usando matriz de peso V . Si la primera capa no tiene función de activación, entonces esencialmente hemos factorizado la matriz de peso de la capa original basada en W . El enfoque factorizado es calcular $h = \text{gramo}(V \cdot t \cdot u \cdot X + b)$. Si t produce q salidas, entonces t y V juntos contienen solo $(norte + pag)q$ parámetros, mientras que W contiene *notario público* parámetros. Para pequeños q , esto puede suponer un ahorro considerable en parámetros. Tiene el costo de restringir la transformación lineal para que sea de bajo rango, pero estas relaciones de bajo rango suelen ser suficientes. Las unidades ocultas lineales ofrecen así una forma eficaz de reducir el número de parámetros en una red.

Las unidades Softmax son otro tipo de unidad que generalmente se usa como salida (como se describe en la sección 6.2.2.3) pero a veces se puede usar como una unidad oculta. Las unidades Softmax representan naturalmente una distribución de probabilidad sobre una variable discreta con k valores posibles, por lo que pueden ser utilizados como una especie de interruptor. Este tipo de unidades ocultas generalmente solo se usan en arquitecturas más avanzadas que aprenden explícitamente a manipular la memoria, como se describe en la sección 10.12.

Algunos otros tipos de unidades ocultas razonablemente comunes incluyen:

- **Función de base radial** unidad RBF: $h = \text{Exp}^{-\frac{1}{2} \sum_i (x_i - \bar{x}_i)^2 / W_{i,i}}$. Este función se vuelve más activa a medida que x se acerca a una plantilla \bar{x}_i , porque se satura para la mayoría x , puede ser difícil de optimizar.
- **softplus**: $\text{gramo}(a) = \ln(1 + e^a)$ = registro $(1 + e^a)$. Esta es una versión suave del rectificador, introducido por [Dugas et al. \(2001\)](#) para la aproximación de funciones y por [Nair y Hinton \(2010\)](#) para las distribuciones condicionales de modelos probabilísticos no dirigidos. [gloria et al. \(2011a\)](#) compararon el softplus y el rectificador y encontraron mejores resultados con este último. Generalmente se desaconseja el uso del softplus. El softplus demuestra que el rendimiento de los tipos de unidades ocultas puede ser muy contrario a la intuición: uno podría esperar que tenga una ventaja sobre el rectificador debido a que es diferenciable en todas partes o debido a que se satura menos por completo, pero empíricamente no es así.
- **DuroTanh**: esto tiene una forma similar a la bronceada y el rectificador, pero a diferencia de este último, está acotado, $\text{gramo}(a) = \max(-1, \min(1, a))$. Fue introducido por [coloberto \(2004\)](#).

El diseño de unidades ocultas sigue siendo un área activa de investigación y quedan muchos tipos de unidades ocultas útiles por descubrir.

6.4 Diseño de arquitectura

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

La mayoría de las redes neuronales están organizadas en grupos de unidades llamadas capas. La mayoría de las arquitecturas de redes neuronales organizan estas capas en una estructura de cadena, siendo cada capa una función de la capa que la precedió. En esta estructura, la primera capa está dada por

$$h^{(1)} = \text{gramo}^{(1)} W^{(1)} \cdot X + b^{(1)}, \quad (6.40)$$

la segunda capa está dada por

$$h^{(2)} = \text{gramo}^{(2)} W^{(2)} \cdot h^{(1)} + b^{(2)}, \quad (6.41)$$

etcétera.

En estas arquitecturas basadas en cadenas, las principales consideraciones arquitectónicas son elegir la profundidad de la red y el ancho de cada capa. Como veremos, una red con incluso una capa oculta es suficiente para ajustarse al conjunto de entrenamiento. Las redes más profundas a menudo pueden usar muchas menos unidades por capa y muchos menos parámetros y, a menudo, generalizar al conjunto de prueba, pero también suelen ser más difíciles de optimizar. La arquitectura de red ideal para una tarea se debe encontrar a través de la experimentación guiada por el control del error del conjunto de validación.

6.4.1 Propiedades y profundidad de la aproximación universal

Un modelo lineal, que mapea desde las características hasta las salidas a través de la multiplicación de matrices, puede, por definición, representar solo funciones lineales. Tiene la ventaja de ser fácil de entrenar porque muchas funciones de pérdida dan como resultado problemas de optimización convexos cuando se aplican a modelos lineales. Desafortunadamente, a menudo queremos aprender funciones no lineales.

A primera vista, podríamos suponer que aprender una función no lineal requiere diseñar una familia de modelos especializada para el tipo de no linealidad que queremos aprender. Afortunadamente, las redes feedforward con capas ocultas proporcionan un marco de aproximación universal. Específicamente, el **teorema de aproximación universal** (Hornik *et al.*, 1989; Cybenko, 1989) establece que una red feedforward con una capa de salida lineal y al menos una capa oculta con cualquier función de activación de "aplastamiento" (como la función de activación sigmoidea logística) puede aproximar cualquier función medible de Borel de un espacio de dimensión finita a otro con cualquier función deseada. cantidad de error distinta de cero, siempre que a la red se le proporcionen suficientes unidades ocultas. Las derivadas de la red feedforward también pueden aproximarse arbitrariamente bien a las derivadas de la función (Hornik *et al.*, 1990). El concepto de mensurabilidad de Borel está más allá del alcance de este libro; para nuestros propósitos es suficiente decir que cualquier función continua en un subconjunto cerrado y acotado de \mathbb{R}^n es Borel medible y por lo tanto puede ser逼近ado por una red neuronal. Una red neuronal también puede aproximar cualquier función de mapeo de cualquier espacio discreto de dimensión finita a otro. Si bien los teoremas originales se establecieron por primera vez en términos de unidades con funciones de activación que saturan tanto para argumentos muy negativos como muy positivos, los teoremas de aproximación universal también se han demostrado para una clase más amplia de funciones de activación, que incluye la unidad lineal rectificada que ahora se usa comúnmente. (Leshno *et al.*, 1993).

El teorema de aproximación universal significa que independientemente de la función que estemos tratando de aprender, sabemos que un MLP grande podrá *representar* esta función. Sin embargo, no tenemos la garantía de que el algoritmo de entrenamiento pueda *aprender* esa función. Incluso si el MLP es capaz de representar la función, el aprendizaje puede fallar por dos razones diferentes. En primer lugar, el algoritmo de optimización utilizado para el entrenamiento.

Es posible que no pueda encontrar el valor de los parámetros que corresponde a la función deseada. En segundo lugar, el algoritmo de entrenamiento podría elegir la función incorrecta debido al sobreajuste. Recuperar de la sección 5.2.1 que el teorema de "no hay almuerzo gratis" muestra que no existe un algoritmo de aprendizaje automático universalmente superior. Las redes feedforward proporcionan un sistema universal para representar funciones, en el sentido de que, dada una función, existe una red feedforward que aproxima la función. No existe un procedimiento universal para examinar un conjunto de entrenamiento de ejemplos específicos y elegir una función que se generalice a puntos que no están en el conjunto de entrenamiento.

El teorema de aproximación universal dice que existe una red lo suficientemente grande como para lograr cualquier grado de precisión que deseemos, pero el teorema no dice qué tan grande será esta red. [barrón\(1993\)](#) proporciona algunos límites sobre el tamaño de una red de una sola capa necesaria para aproximarse a una amplia clase de funciones. Desafortunadamente, en el peor de los casos, puede ser necesario un número exponencial de unidades ocultas (posiblemente con una unidad oculta correspondiente a cada configuración de entrada que debe distinguirse). Esto es más fácil de ver en el caso binario: el número de funciones binarias posibles en vectores $v \in \{0,1\}^n$ es 2^n y seleccionar una de esas funciones requiere 2^n bits, que en general requerirán $O(2^n)$ grados de libertad.

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

Existen familias de funciones que pueden ser aproximadas eficientemente por una arquitectura con profundidad mayor que algún valor d , pero que requieren un modelo mucho más grande si la profundidad está restringida a ser menor o igual a d . En muchos casos, el número de unidades ocultas requeridas por el modelo superficial es exponencial en n . Dichos resultados se probaron por primera vez para modelos que no se parecen a las redes neuronales continuas y diferenciables utilizadas para el aprendizaje automático, pero desde entonces se han extendido a estos modelos. Los primeros resultados fueron para circuitos de puertas lógicas ([Håstad, 1986](#)). El trabajo posterior amplió estos resultados a unidades de umbral lineales con pesos no negativos ([Hastad y Goldmann, 1991; Hajnal et al., 1993](#)), y luego a redes con activaciones de valor continuo ([Masa, 1992; Masa et al., 1994](#)). Muchas redes neuronales modernas utilizan unidades lineales rectificadas. [Leshno et al. \(1993\)](#) demostraron que las redes poco profundas con una amplia familia de funciones de activación no polinómicas, incluidas las unidades lineales rectificadas, tienen propiedades de aproximación universal, pero estos resultados no abordan las cuestiones de profundidad o eficiencia; solo especifican que una red rectificadora suficientemente amplia podría representar cualquier función. [Montúfar et al.](#)

(2014) mostró que las funciones representables con una red rectificadora profunda pueden requerir un número exponencial de unidades ocultas con una red superficial (una capa oculta). Más precisamente, demostraron que las redes lineales por partes (que se pueden obtener de las no linealidades del rectificador o de las unidades maxout) pueden representar funciones con un número de regiones que es exponencial en la profundidad de la red. Cifra 6.5 ilustra cómo una red con rectificación de valor absoluto crea imágenes especulares de la función calculada sobre alguna unidad oculta, con respecto a la entrada de esa unidad oculta. Cada unidad oculta especifica dónde doblar el espacio de entrada para crear respuestas especulares (en ambos lados de la no linealidad del valor absoluto). Al componer estas operaciones de plegado, obtenemos un número exponencialmente grande de regiones lineales por partes que pueden capturar todo tipo de patrones regulares (por ejemplo, repetitivos).

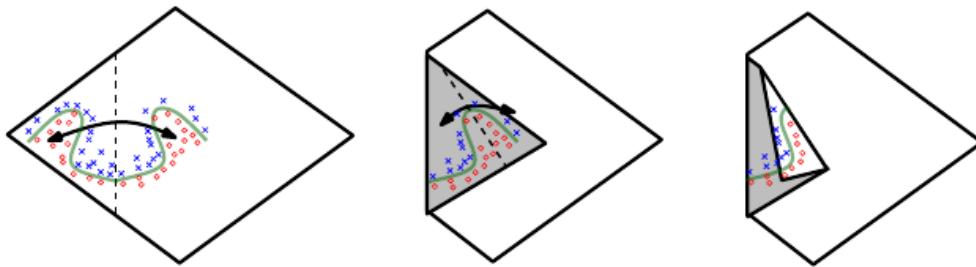


Figura 6.5: Una explicación geométrica intuitiva de la ventaja exponencial de las redes rectificadoras más profundas formalmente por Montúfaret al. (2014). (Izquierda) Una unidad de rectificación de valor absoluto tiene la misma salida para cada par de puntos espejo en su entrada. El eje de simetría del espejo está dado por el hiperplano definido por los pesos y el sesgo de la unidad. Una función calculada sobre esa unidad (la superficie de decisión verde) será una imagen especular de un patrón más simple a lo largo de ese eje de simetría. (Centro) La función se puede obtener doblando el espacio alrededor del eje de simetría. (Derecha) Se puede doblar otro patrón repetitivo sobre el primero (mediante otra unidad aguas abajo) para obtener otra simetría (que ahora se repite cuatro veces, con dos capas ocultas). Figura reproducida con permiso de Montúfaret al. (2014).

Más precisamente, el teorema principal en Montúfaret al. (2014) establece que el número de regiones lineales excavadas por una red rectificadora profunda con d entradas, profundidad o , y n unidades por capa oculta, es

$$O = \frac{n}{d}^{n \cdot d^{(y_o-1)}} \quad (6.42)$$

es decir, exponencial en la profundidad y_o . En el caso de redes maxout con k filtros por unidad, el número de regiones lineales es

$$De acuerdo a \frac{O}{k} = \frac{n}{d}^{n \cdot d^{(y_o-1)+d}}. \quad (6.43)$$

Por supuesto, no hay garantía de que los tipos de funciones que queremos aprender en aplicaciones de aprendizaje automático (y en particular para IA) compartan tal propiedad.

También podemos querer elegir un modelo profundo por razones estadísticas. Cada vez que elegimos un algoritmo de aprendizaje automático específico, declaramos implícitamente un conjunto de creencias previas que tenemos sobre qué tipo de función debe aprender el algoritmo. Elegir un modelo profundo codifica una creencia muy general de que la función que queremos aprender debe involucrar la composición de varias funciones más simples. Esto puede interpretarse desde el punto de vista del aprendizaje de la representación como diciendo que creemos que el problema de aprendizaje consiste en descubrir un conjunto de factores subyacentes de variación que, a su vez, pueden describirse en términos de otros factores subyacentes de variación más simples. Alternativamente, podemos interpretar el uso de una arquitectura profunda como expresión de la creencia de que la función que queremos aprender es un programa de computadora que consta de múltiples pasos, donde cada paso hace uso de la salida del paso anterior. Estos resultados intermedios no son necesariamente factores de variación, sino que pueden ser análogos a los contadores o punteros que utiliza la red para organizar su procesamiento interno. Empíricamente, una mayor profundidad parece resultar en una mejor generalización para una amplia variedad de tareas ([bengio et al., 2007; Erhan et al., 2009; bengio, 2009; Mesnil et al., 2011; Ciresán et al., 2012; Krizhevskiet al., 2012; Sermanet et al., 2013; Farabet et al., 2013; cupé et al., 2013; Kahou et al., 2013; Buen compañero et al., 2014d; Szegedy et al., 2014a](#)). Ver figura 6.6 y figura 6.7 para ejemplos de algunos de estos resultados empíricos. Esto sugiere que el uso de arquitecturas profundas expresa de hecho un previo útil sobre el espacio de funciones que aprende el modelo.

6.4.2 Otras consideraciones arquitectónicas

Hasta ahora hemos descrito las redes neuronales como simples cadenas de capas, siendo las principales consideraciones la profundidad de la red y el ancho de cada capa. En la práctica, las redes neuronales muestran una diversidad considerablemente mayor.

Se han desarrollado muchas arquitecturas de redes neuronales para tareas específicas. Las arquitecturas especializadas para la visión artificial denominadas redes convolucionales se describen en el capítulo 9. Las redes feedforward también se pueden generalizar a las redes neuronales recurrentes para el procesamiento de secuencias, descritas en el capítulo 10, que tienen sus propias consideraciones arquitectónicas.

En general, las capas no necesitan estar conectadas en cadena, aunque esta es la práctica más común. Muchas arquitecturas construyen una cadena principal pero luego le agregan características arquitectónicas adicionales, como omitir conexiones que van desde la capa i a la capa $i+2$ o más alto. Estas conexiones de salto facilitan que el degradado fluya desde las capas de salida a las capas más cercanas a la entrada.

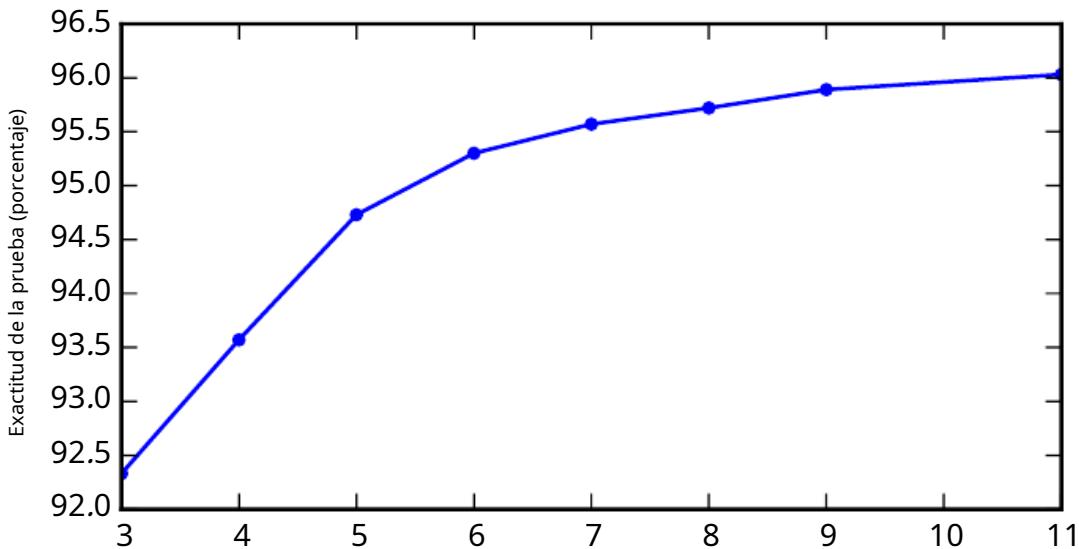


Figura 6.6: Resultados empíricos que muestran que las redes más profundas se generalizan mejor cuando se utilizan para transcribir números de varios dígitos a partir de fotografías de direcciones. Datos de [Buen compaño et al.\(2014d\)](#). La precisión del conjunto de prueba aumenta constantemente con el aumento de la profundidad. Ver figura 6.7 para un experimento de control que demuestre que otros aumentos en el tamaño del modelo no producen el mismo efecto.

Otra consideración clave del diseño de la arquitectura es exactamente cómo conectar un par de capas entre sí. En la capa de red neuronal predeterminada descrita por una transformación lineal a través de una matriz W , cada unidad de entrada está conectada a cada unidad de salida. Muchas redes especializadas en los capítulos siguientes tienen menos conexiones, de modo que cada unidad en la capa de entrada está conectada solo a un pequeño subconjunto de unidades en la capa de salida. Estas estrategias para reducir la cantidad de conexiones reducen la cantidad de parámetros y la cantidad de cómputo requerido para evaluar la red, pero a menudo dependen en gran medida del problema. Por ejemplo, las redes convolucionales, descritas en el capítulo 9, usa patrones especializados de conexiones dispersas que son muy efectivos para problemas de visión por computadora. En este capítulo, es difícil dar consejos mucho más específicos sobre la arquitectura de una red neuronal genérica. Los capítulos posteriores desarrollan las estrategias arquitectónicas particulares que se ha encontrado que funcionan bien para diferentes dominios de aplicación.

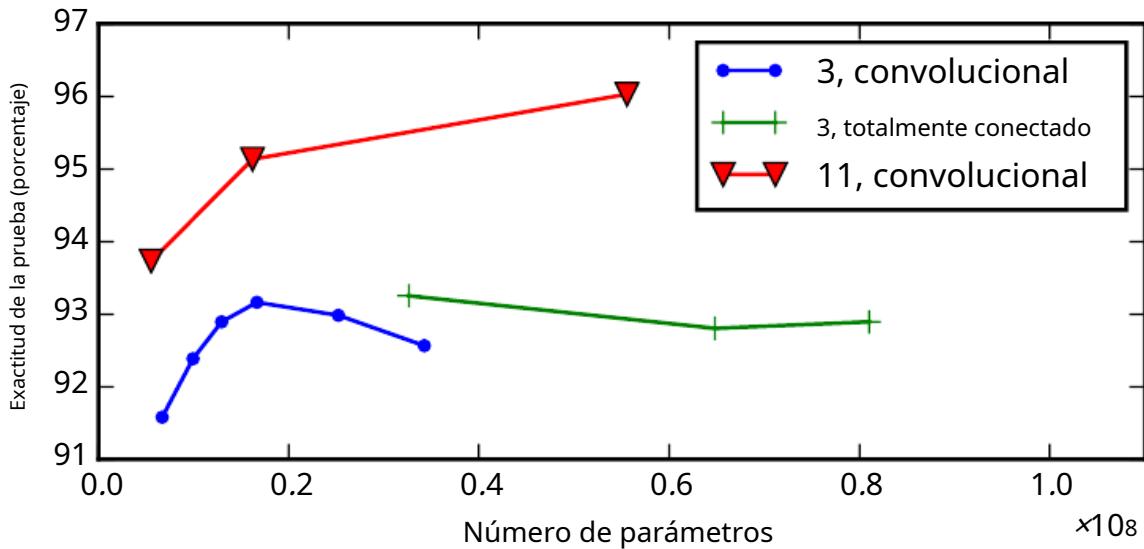


Figura 6.7: Los modelos más profundos tienden a funcionar mejor. Esto no se debe simplemente a que el modelo sea más grande. Este experimento de [Buen compañero et al. \(2014d\)](#) muestra que aumentar el número de parámetros en capas de redes convolucionales sin aumentar su profundidad no es tan efectivo para aumentar el rendimiento del conjunto de prueba. La leyenda indica la profundidad de la red utilizada para hacer cada curva y si la curva representa una variación en el tamaño de las capas convolucionales o totalmente conectadas. Observamos que los modelos superficiales en este contexto se sobreajustan en alrededor de 20 millones de parámetros, mientras que los profundos pueden beneficiarse de tener más de 60 millones. Esto sugiere que el uso de un modelo profundo expresa una preferencia útil sobre el espacio de funciones que el modelo puede aprender. Específicamente, expresa la creencia de que la función debe consistir en muchas funciones más simples compuestas juntas. Esto podría resultar en el aprendizaje de una representación que se compone a su vez de representaciones más simples (por ejemplo,

6.5 Propagación hacia atrás y otros algoritmos de diferenciación

Cuando usamos una red neuronal feedforward para aceptar una entrada x y producir una salida \hat{y} , la información fluye hacia adelante a través de la red. Las entradas x proporcionan la información inicial que luego se propaga hasta las unidades ocultas en cada capa y finalmente produce \hat{y} . Se llama **propagación hacia adelante**. Durante el entrenamiento, la propagación directa puede continuar hasta que produzca un costo escalar $J(\theta)$. El **retropropagación** algoritmo ([Rumelhart et al., 1986a](#)), a menudo llamado simplemente **backprop**, permite que la información del costo fluya hacia atrás a través de la red para calcular el gradiente.

Calcular una expresión analítica para el gradiente es sencillo, pero evaluar numéricamente dicha expresión puede ser computacionalmente costoso. El algoritmo de retropropagación lo hace utilizando un procedimiento simple y económico.

El término retropropagación a menudo se malinterpreta como el algoritmo de aprendizaje completo para redes neuronales multicapa. En realidad, la propagación hacia atrás se refiere solo al método para calcular el gradiente, mientras que otro algoritmo, como el descenso de gradiente estocástico, se usa para realizar el aprendizaje usando este gradiente. Además, la propagación hacia atrás a menudo se malinterpreta como algo específico de las redes neuronales multicapa, pero en principio puede calcular derivadas de cualquier función (para algunas funciones, la respuesta correcta es informar que la derivada de la función no está definida). Específicamente, describiremos cómo calcular el gradiente $\nabla_{\theta} F(x, y)$ para una función arbitraria F , donde x es un conjunto de variables cuyas derivadas se desean, y es un conjunto adicional de variables que son entradas de la función pero cuyas derivadas no son necesarias. En los algoritmos de aprendizaje, el gradiente que requerimos con más frecuencia es el gradiente de la función de costo con respecto a los parámetros, $\nabla_{\theta} J$. Muchas tareas de aprendizaje automático implican calcular otras derivadas, ya sea como parte del proceso de aprendizaje o para analizar el modelo aprendido. El algoritmo de retropropagación también se puede aplicar a estas tareas y no se limita a calcular el gradiente de la función de costo con respecto a los parámetros. La idea de calcular derivadas propagando información a través de una red es muy general y puede usarse para calcular valores como el jacobiano de una función. F con múltiples salidas. Restringimos nuestra descripción aquí al caso más comúnmente usado donde F tiene una sola salida.

6.5.1 Gráficos computacionales

Hasta ahora hemos discutido las redes neuronales con un lenguaje gráfico relativamente informal. Para describir el algoritmo de propagación hacia atrás con mayor precisión, es útil tener un **gráfico computacional** idioma.

Son posibles muchas formas de formalizar el cálculo como gráficos.

Aquí, usamos cada nodo en el gráfico para indicar una variable. La variable puede ser un escalar, un vector, una matriz, un tensor o incluso una variable de otro tipo.

Para formalizar nuestras gráficas, también necesitamos introducir la idea de un **operación**. Una operación es una función simple de una o más variables. Nuestro lenguaje gráfico va acompañado de un conjunto de operaciones permitidas. Las funciones más complicadas que las operaciones de este conjunto se pueden describir componiendo muchas operaciones juntas.

Sin pérdida de generalidad, definimos una operación para devolver solo una única variable de salida. Esto no pierde generalidad porque la variable de salida puede tener múltiples entradas, como un vector. Las implementaciones de software de propagación hacia atrás generalmente admiten operaciones con múltiples salidas, pero evitamos este caso en nuestra descripción porque introduce muchos detalles adicionales que no son importantes para la comprensión conceptual.

Si una variable y se calcula aplicando una operación a una variable X , luego dibujamos un borde dirigido desde X a y . A veces anotamos el nodo de salida con el nombre de la operación aplicada, y otras veces omitimos esta etiqueta cuando la operación está clara en el contexto.

En la figura se muestran ejemplos de gráficos computacionales.[6.8](#).

6.5.2 Regla de la cadena del cálculo

La regla de la cadena del cálculo (que no debe confundirse con la regla de la cadena de la probabilidad) se utiliza para calcular las derivadas de funciones formadas al componer otras funciones cuyas derivadas se conocen. La retropropagación es un algoritmo que calcula la regla de la cadena, con un orden específico de operaciones que es altamente eficiente.

Dejar X sea un número real, y sea F y g uno o ambos ser funciones de mapeo de un número real a un número real. Suponer que $y = g(m)$ (X) y $z = F(g(m)) = F(y)$. Entonces la regla de la cadena establece que

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (6.44)$$

Podemos generalizar esto más allá del caso escalar. Suponer que $X \in \mathbb{R}_{\text{metro}}$, $y \in \mathbb{R}_{\text{norte}}$,

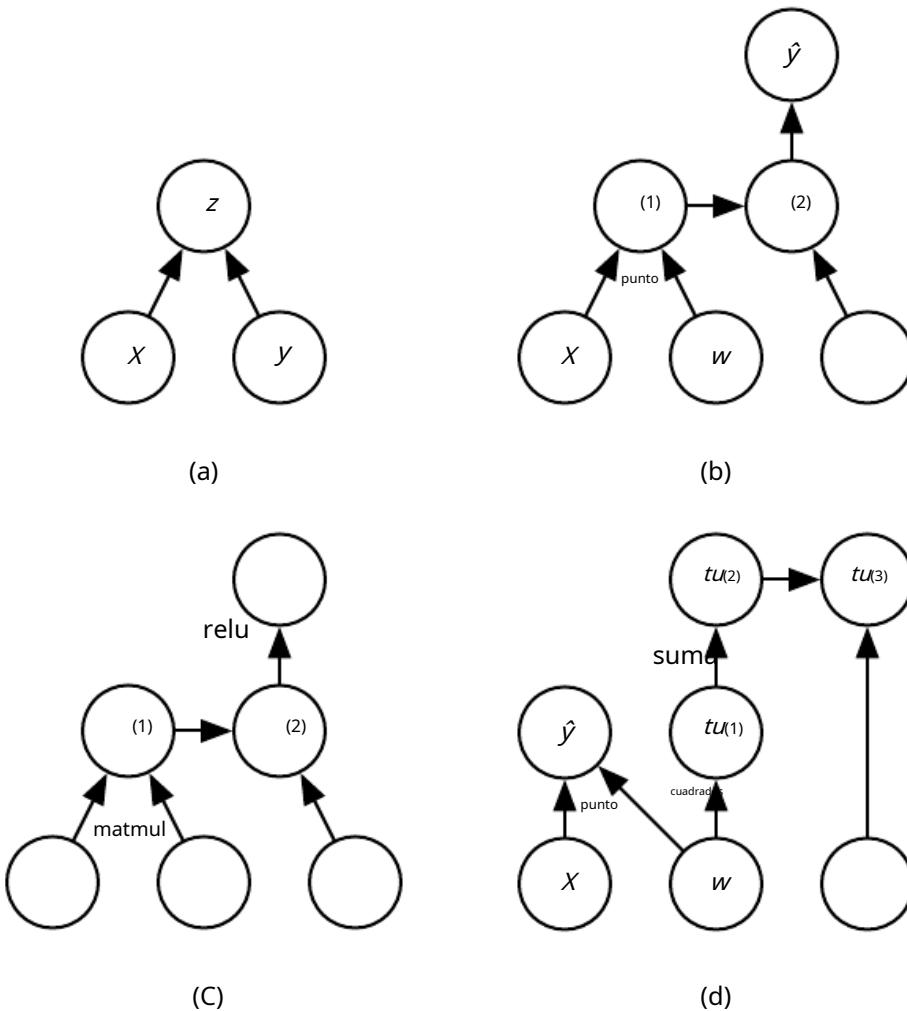


Figura 6.8: Ejemplos de gráficos computacionales.(a)El gráfico usando el \times operación para calcular $z=xy$.(b)El gráfico para la predicción de regresión logística $\hat{y}=\sigma Xw+b$. Algunas de las expresiones intermedias no tienen nombres en la expresión algebraica pero necesitan nombres en la gráfica. Simplemente nombramos el i -th tal variable $tu(i)$.(C)El gráfico computacional para la expresión $H=\max\{0,XW+b\}$, que calcula una matriz de diseño de activaciones de unidades lineales rectificadas H dada una matriz de diseño que contiene un minilote de entradas X .(d)Los ejemplos a-c aplicaron como máximo una operación a cada variable, pero es posible aplicar más de una operación. Aquí mostramos un gráfico de cálculo que aplica más de una operación a los pesos w de un modelo de regresión lineal. El los pesos se utilizan para hacer tanto la predicción \hat{y} la penalización por caída de peso $\lambda \|W\|_2$.

gramo de $\mathbf{R}_{\text{metro}}$ a $\mathbf{R}_{\text{norte}}$, y los mapas de $\mathbf{R}_{\text{norte}}$ a \mathbf{R}_{sur} . Si $\mathbf{y} = \text{gramo}(\mathbf{X})$ y $\mathbf{z} = f(\mathbf{y})$, entonces

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad (6.45)$$

En notación vectorial, esto se puede escribir de manera equivalente como

$$\nabla_{\mathbf{x}} z = \left[\frac{\partial z}{\partial x_1}, \frac{\partial z}{\partial x_2}, \dots, \frac{\partial z}{\partial x_n} \right]^T \quad (6.46)$$

dónde $\nabla_{\mathbf{x}}$ es el $n \times n$ tensor matriz jacobiana de gramo .

De esto vemos que el gradiente de una variable \mathbf{z} se puede obtener multiplicando una matriz jacobiana $\nabla_{\mathbf{x}}$ por un gradiente $\nabla_{\mathbf{y}}$. El algoritmo de retropropagación consiste de realizar tal producto de gradiente jacobiano para cada operación en el gráfico.

Por lo general, no aplicamos el algoritmo de propagación hacia atrás simplemente a vectores, sino a tensores de dimensionalidad arbitraria. Conceptualmente, esto es exactamente lo mismo que la retropropagación con vectores. La única diferencia es cómo se organizan los números en una cuadrícula para formar un tensor. Podríamos imaginar aplanar cada tensor en un vector antes de ejecutar la propagación hacia atrás, calcular un gradiente de valor vectorial y luego remodelar el gradiente nuevamente en un tensor. En esta vista reorganizada, la retropropagación sigue multiplicando jacobianos por gradientes.

Para indicar el gradiente de un valor z con respecto a un tensor \mathbf{X} , nosotros escribimos $\nabla_{\mathbf{x}} z$, como si \mathbf{X} fueran un vector. Los índices en \mathbf{X} ahora tiene múltiples coordenadas; por ejemplo, un tensor 3-D está indexado por tres coordenadas. Podemos abstraer esto usando una sola variable i para representar la tupla completa de índices. Para todos los posibles tuplas de índice i , $(\nabla_{\mathbf{x}} z)_{i,j}$ da $\frac{\partial z}{\partial x_j}$. Esto es exactamente lo mismo que para todos los posibles índices enteros i en un vector, $(\nabla_{\mathbf{x}} z)_i$ da $\frac{\partial z}{\partial x_i}$. Usando esta notación, puede escribir la regla de la cadena tal como se aplica a los tensores. Si $\mathbf{Y} = \text{gramo}(\mathbf{X})$ y $\mathbf{z} = f(\mathbf{Y})$, entonces

$$\nabla_{\mathbf{x}} z = \left[\begin{array}{c} (\nabla_{\mathbf{Y}} Y_j) \frac{\partial z}{\partial Y_j} \\ \vdots \\ (\nabla_{\mathbf{Y}} Y_j) \frac{\partial z}{\partial Y_j} \end{array} \right]. \quad (6.47)$$

6.5.3 Aplicación recursiva de la regla de la cadena para obtener backprop

Usando la regla de la cadena, es sencillo escribir una expresión algebraica para el gradiente de un escalar con respecto a cualquier nodo en el gráfico computacional que produjo ese escalar. Sin embargo, evaluar esa expresión en una computadora introduce algunas consideraciones adicionales.

Específicamente, muchas subexpresiones pueden repetirse varias veces dentro de la expresión general del gradiente. Cualquier procedimiento que calcule el gradiente.

tendrá que elegir si desea almacenar estas subexpresiones o volver a calcularlas varias veces. Un ejemplo de cómo surgen estas subexpresiones repetidas se da en la figura 6.9. En algunos casos, calcular la misma subexpresión dos veces simplemente sería un desperdicio. Para gráficos complicados, puede haber muchos de estos cálculos desperdiciados exponencialmente, lo que hace inviable una implementación ingenua de la regla de la cadena. En otros casos, calcular la misma subexpresión dos veces podría ser una forma válida de reducir el consumo de memoria a costa de un mayor tiempo de ejecución.

Primero comenzamos con una versión del algoritmo de propagación hacia atrás que especifica directamente el cálculo del gradiente real (algoritmo 6.2 junto con el algoritmo 6.1 para el cálculo directo asociado), en el orden en que realmente se realizará y de acuerdo con la aplicación recursiva de la regla de la cadena. Uno podría realizar directamente estos cálculos o ver la descripción del algoritmo como una especificación simbólica del gráfico computacional para calcular la propagación hacia atrás. Sin embargo, esta formulación no hace explícita la manipulación y la construcción del grafo simbólico que realiza el cálculo del gradiente. Tal formulación se presenta a continuación en la sección 6.5.6, con algoritmo 6.5, donde también generalizamos a nodos que contienen tensores arbitrarios.

Primero considere un gráfico computacional que describa cómo calcular un solo escalar $t_{U(norte)}$ (decir la pérdida en un ejemplo de entrenamiento). Este escalar es la cantidad cuyo gradiente queremos obtener, con respecto a los n nodos de entrada $t_{U(1)}, t_{U(2)}, \dots, t_{U(norte)}$. En otras palabras, deseamos calcular $\partial t_{U(norte)} / \partial t_{U(i)}$ para todos $i \in \{1, 2, \dots, norte\}$. En la aplicación de retropropagación para computar gradientes para descenso de gradiente sobre parámetros, $t_{U(norte)}$ será el costo asociado con un ejemplo o un mini lote, mientras que $t_{U(1)}, t_{U(2)}, \dots, t_{U(norte)}$ corresponden a los parámetros del modelo.

Supondremos que los nodos del grafo han sido ordenados de tal manera que podemos calcular su salida uno tras otro, comenzando en $t_{U(norte+1)}$ y subiendo a $t_{U(norte)}$. Como se define en el algoritmo 6.1, cada nodo $t_{U(i)}$ está asociado con una operación F_i y se calcula evaluando la función

$$t_{U(i)} = F_i(A_{(i)}) \quad (6.48)$$

dónde $A_{(i)}$ es el conjunto de todos los nodos que son padres de $t_{U(i)}$.

Ese algoritmo especifica el cálculo de la propagación hacia adelante, que podríamos poner en un gráfico *GRAMO*. Para realizar la propagación hacia atrás, podemos construir un gráfico computacional que depende de *GRAMO* y le agrega un conjunto adicional de nodos. Estos forman un subgrafo *B* con un nodo por nodo de *GRAMO*. Cálculo en *B* procede exactamente al revés del orden de cálculo en *GRAMO*, y cada nodo de *B* calcula la derivada $\partial t_{U(norte)} / \partial t_{U(i)}$ asociado con el nodo gráfico directo $t_{U(i)}$. Esto se ha hecho

Algoritmo 6.1 Un procedimiento que realiza el mapeo de cálculos. $norte$ entradas $tu(1)$ a $tu(norte)$ a una salida $tu(norte)$. Esto define un gráfico computacional donde cada nodo calcula el valor numérico $tu(i)$ aplicando una función $F(i)$ al conjunto de argumentos $A(i)$ que comprende los valores de los nodos anteriores $tu(j), j < i$, con $j \in Pensilvania(tu(i))$. La entrada al gráfico computacional es el vector X , y se establece en la primera $norte$ nodos $tu(1)$ a $tu(norte)$. La salida del gráfico computacional se lee del último nodo (salida) $tu(norte)$.

```

para  $i=1, \dots, norte$  hacer
   $tu(i) \leftarrow X_i$ 
fin para
para  $i=norte+1, \dots, norte$  hacer
   $A(i) \leftarrow \{tu(j) / j \in Pensilvania(tu(i))\}$ 
   $tu(i) \leftarrow F(i)(A(i))$ 
fin para
devolver  $tu(norte)$ 

```

usando la regla de la cadena con respecto a la salida escalar $tu(norte)$:

$$\frac{\partial u(norte)}{\partial u(j)} = \prod_{i \in Pensilvania(tu(i))} \frac{\partial u(norte)}{\partial u(i)} \frac{\partial u(i)}{\partial u(j)} \quad (6.49)$$

según lo especificado por el algoritmo 6.2. el subgrafo B contiene exactamente un borde para cada borde del nodo $tu(j)$ al nodo $tu(i)$ de $GRAMO$. el borde de $tu(j)$ a $tu(i)$ está asociado con el cómputo de $\frac{\partial u(i)}{\partial u(j)}$. Además, se realiza un producto escalar para cada nodo, entre el gradiente ya calculado con respecto a los nodos $tu(i)$ que son niños de $tu(j)$ y el vector que contiene las derivadas parciales $\frac{\partial u(i)}{\partial u(j)}$ por los mismos niños nodos $tu(i)$. Para resumir, la cantidad de cómputo requerida para realizar la propagación hacia atrás escala linealmente con el número de aristas en $GRAMO$, donde el cálculo para cada arista corresponde a calcular una derivada parcial (de un nodo con respecto a uno de sus padres) así como realizar una multiplicación y una suma. A continuación, generalizamos este análisis a nodos con valores tensoriales, que es solo una forma de agrupar múltiples valores escalares en el mismo nodo y permitir implementaciones más eficientes.

El algoritmo de propagación hacia atrás está diseñado para reducir el número de subexpresiones comunes sin tener en cuenta la memoria. Específicamente, funciona en el orden de un producto jacobiano por nodo en el gráfico. Esto se puede ver en el hecho de que backprop (algoritmo 6.2) visita cada borde del nodo $tu(j)$ al nodo $tu(i)$ de la gráfica exactamente una vez para obtener la derivada parcial asociada $\frac{\partial u(i)}{\partial u(j)}$.

Algoritmo 6.2 Versión simplificada del algoritmo de propagación hacia atrás para calcular las derivadas de $t_{U(norte)}$ con respecto a las variables en el gráfico. Este ejemplo está destinado a una mayor comprensión al mostrar un caso simplificado donde todas las variables son escalares, y deseamos calcular las derivadas con respecto a $t_{U(1)}, \dots, t_{U(norte)}$. Esta versión simplificada calcula las derivadas de todos los nodos del gráfico. El costo computacional de este algoritmo es proporcional al número de aristas en el gráfico, asumiendo que la derivada parcial asociada con cada arista requiere un tiempo constante. Este es del mismo orden que el número de cálculos para la propagación hacia adelante. Cada $t_{U(j)}$ es una función de los padres $t_{U(i)}$ de $t_{U(j)}$, de este modo vincular los nodos del gráfico de avance con los agregados para el gráfico de propagación hacia atrás.

Ejecutar propagación hacia adelante (algoritmo 6.1 para este ejemplo) para obtener las activaciones de la red

Inicializar tabla_graduación, una estructura de datos que almacenará las derivadas que se han calculado. La entrada grad_table[$t_{U(j)}$] almacenará el valor calculado de $\frac{\partial U(norte)}{\partial U(j)}$.

grad_table[$t_{U(norte)}$] $\leftarrow 1$ **para** $j = norte - 1$ hasta 1 **hacer**

La siguiente línea calcula $\frac{\partial t_{U(norte)}}{\partial U(j)} = \sum_{i:j \in Pensilvania(t_{U(i)})} \frac{\partial U(norte)}{\partial U(i)} \frac{\partial U(i)}{\partial U(j)}$ SN interfaz de gradientes almacenados:

grad_table[$t_{U(j)}$] $\leftarrow \sum_{i:j \in Pensilvania(t_{U(i)})} \text{grad_table}[t_{U(i)}] \frac{\partial t_{U(norte)}}{\partial U(i)}$

fin para

devolver {grad_table[$t_{U(i)}$] / $i = 1, \dots, norte$ }

La retropropagación evita así la explosión exponencial en subexpresiones repetidas. Sin embargo, otros algoritmos pueden evitar más subexpresiones al realizar simplificaciones en el gráfico computacional, o pueden conservar la memoria al volver a calcular en lugar de almacenar algunas subexpresiones. Revisaremos estas ideas después de describir el propio algoritmo de propagación hacia atrás.

6.5.4 Cálculo de retropropagación en MLP totalmente conectado

Para aclarar la definición anterior del cálculo de la propagación hacia atrás, consideremos el gráfico específico asociado con un MLP multicapa totalmente conectado.

Algoritmo 6.3 primero muestra la propagación hacia adelante, que asigna parámetros a la pérdida supervisada $L(\hat{y}, y)$ asociado con un solo ejemplo de entrenamiento (entrada, objetivo) (x, y) , con \hat{y} la salida de la red neuronal cuando x se proporciona en la entrada.

Algoritmo 6.4 luego muestra el cálculo correspondiente a realizar para

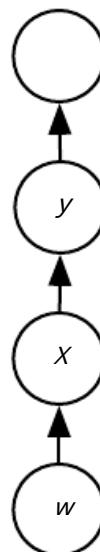


Figura 6.9: Un gráfico computacional que da como resultado subexpresiones repetidas al calcular el gradiente. Dejar $w \in \mathbb{R}$ ser la entrada al gráfico. Usamos la misma función $F: \mathbb{R} \rightarrow \mathbb{R}$ como la operación que aplicamos en cada paso de una cadena: $x = F(w)$, $y = F(x)$, $z = F(y)$.

Computar $\frac{\partial z}{\partial w}$, aplicamos la ecuación 6.44 y obtener:

$$\frac{\partial z}{\partial w} \quad (6.50)$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \quad (6.51)$$

$$= F(y)F(x)F(w) = F(F(F(w)))F \quad (6.52)$$

$$(F(w))F(w) \quad (6.53)$$

Ecuación 6.52 sugiere una implementación en la que calculamos el valor de $F(w)$ una sola vez y almacenarlo en la variable X . Este es el enfoque adoptado por el algoritmo de retropropagación. Un enfoque alternativo es sugerido por la ecuación 6.53, donde la subexpresión $F(w)$ aparece más de una vez. En el enfoque alternativo, $F(w)$ se vuelve a calcular cada vez que se necesita. Cuando la memoria requerida para almacenar el valor de estas expresiones es baja, el enfoque de propagación hacia atrás de la ecuación 6.52 es claramente preferible debido a su tiempo de ejecución reducido. Sin embargo, la ecuación 6.53 también es una implementación válida de la regla de la cadena y es útil cuando la memoria es limitada.

aplicando el algoritmo de retropropagación a este gráfico.

Algoritmos 6.3 y 6.4 son demostraciones que se eligen para que sean simples y fáciles de entender. Sin embargo, están especializados en un problema específico.

Las implementaciones modernas de software se basan en la forma generalizada de retropropagación descrita en la sección 6.5.6 a continuación, que puede acomodar cualquier gráfico computacional mediante la manipulación explícita de una estructura de datos para representar el cálculo simbólico.

Algoritmo 6.3 Propagación hacia adelante a través de una red neuronal profunda típica y el cálculo de la función de costo. La pérdida $L(\hat{y}, y)$ depende de la salida \hat{y} en el objetivo y (mira la sección 6.2.1.1 para ejemplos de funciones de pérdida). Para obtener el costo total J , la pérdida puede agregarse a un regularizador $\Omega(\theta)$, donde θ contiene todos los parámetros (pesos y sesgos). Algoritmo 6.4 muestra cómo calcular los gradientes de J con respecto a los parámetros W y b . Para simplificar, esta demostración utiliza solo un ejemplo de entrada única X . Las aplicaciones prácticas deben utilizar un minibatch. Mira la sección 6.5.7 para una demostración más realista.

Requerir: profundidad de la red, yo

Requerir: $W_i, i \in \{1, \dots, l\}$, las matrices de peso del modelo

Requerir: $b_i, i \in \{1, \dots, l\}$, los parámetros de sesgo del modelo

Requerir: X , la entrada a procesar **Requerir:** y , la salida objetivo

```
h(0)=X  
para k=1, ..., l hacer  
    a(k)=b(k)+ W(k)h(k-1)  
    h(k)=f(a(k))  
fin para  
ŷ=h(yo)  
j=L(ŷ,y) + λΩ(θ)
```

6.5.5 Derivadas de símbolo a símbolo

Las expresiones algebraicas y los gráficos computacionales operan en **simbólos**, o variables que no tienen valores específicos. Estas representaciones algebraicas y basadas en gráficos se llaman **simbólicas** representaciones. Cuando realmente usamos o entrenamos una red neuronal, debemos asignar valores específicos a estos símbolos. Sustituimos una entrada simbólica a la red X con un específico **numérico** valor, como [1.2, 3.765, -1.8].

Algoritmo 6.4 Hacia atrás computación para la red neuronal profunda del algoritmo 6.3, que utiliza además de la entrada x un objetivo y . Este cálculo produce los gradientes en las activaciones $a^{(k)}$ para cada capa k , comenzando desde la capa de salida y retrocediendo hasta la primera capa oculta. A partir de estos gradientes, que pueden interpretarse como una indicación de cómo debe cambiar la salida de cada capa para reducir el error, se puede obtener el gradiente en los parámetros de cada capa. Los gradientes de pesos y sesgos se pueden usar inmediatamente como parte de una actualización de gradiente estocástico (realizando la actualización justo después de que se hayan calculado los gradientes) o se pueden usar con otros métodos de optimización basados en gradientes.

Después del cálculo directo, calcule el gradiente en la capa de salida: $gramo \leftarrow \nabla_{\hat{y}} = \nabla_{\hat{y}} L(\hat{y}, y)$ para $k = yo, yo - 1, \dots, 1$ hacer

 Convierta el degradado en la salida de la capa en un degradado en la activación de preno linealidad (multiplicación por elementos si \neq es elemento-sabio): $gramo \leftarrow \nabla_{a^{(k)}} = g \cdot F(a^{(k)})$

 Calcule los gradientes de pesos y sesgos (incluido el término de regularización, cuando sea necesario):

$$\begin{aligned} \nabla_{b^{(k)}} &= gramo + \lambda \nabla_{b^{(k)}} \Omega(\theta) & \nabla_{W^{(k)}} &= gh^{(k-1)} \\ &+ \lambda \nabla_{W^{(k)}} \Omega(\theta) \end{aligned}$$

 Propaga los degradados con las activaciones de la siguiente capa oculta de nivel inferior:
 $gramo \leftarrow \nabla_{h^{(k-1)}} = W^{(k)} \cdot gramo$ fin para

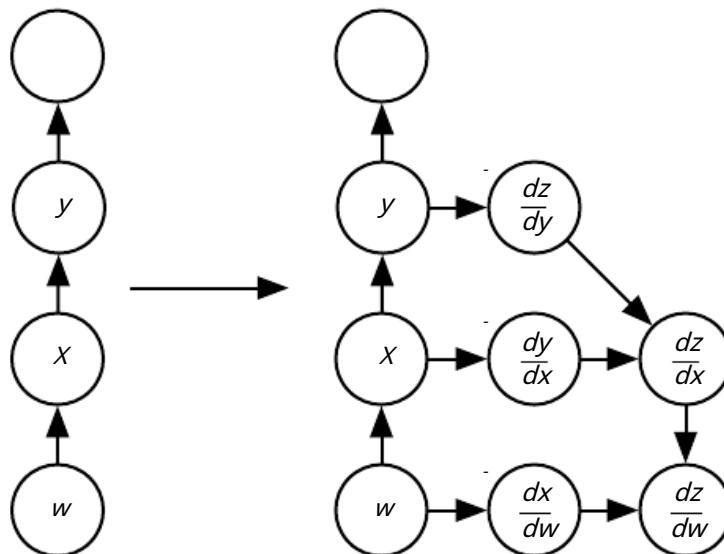


Figura 6.10: Un ejemplo del enfoque de símbolo a símbolo para calcular derivadas. En este enfoque, el algoritmo de propagación hacia atrás no necesita acceder nunca a ningún valor numérico específico real. En su lugar, agrega nodos a un gráfico computacional que describe cómo calcular estas derivadas. Un motor genérico de evaluación de gráficos puede calcular posteriormente las derivadas de cualquier valor numérico específico. (Izquierda) En este ejemplo, comenzamos con un gráfico que representa $z = f(f(f(w)))$. (Bien) Ejecutamos el algoritmo de retropropagación, indicándole que construya el gráfico para la expresión correspondiente $\frac{dz}{dw}$. En este ejemplo, no explicamos cómo funciona el algoritmo de propagación hacia atrás. El propósito es solo ilustrar cuál es el resultado deseado: un gráfico computacional con una descripción simbólica de la derivada.

Algunos enfoques de la propagación hacia atrás toman un gráfico computacional y un conjunto de valores numéricos para las entradas del gráfico, luego devuelven un conjunto de valores numéricos que describen el gradiente en esos valores de entrada. A este enfoque lo llamamos diferenciación de "símbolo a número". Este es el enfoque utilizado por bibliotecas como Torch ([coloberto et al., 2011b](#)) y Café ([jia, 2013](#)).

Otro enfoque es tomar un gráfico computacional y agregar nodos adicionales al gráfico que proporcionen una descripción simbólica de las derivadas deseadas. Este es el enfoque adoptado por Theano ([Bergstra et al., 2010; bastiénet al., 2012](#)) y TensorFlow ([Abadi et al., 2015](#)). Un ejemplo de cómo funciona este enfoque se ilustra en la figura 6.10. La principal ventaja de este enfoque es que las derivadas se describen en el mismo lenguaje que la expresión original. Debido a que las derivadas son solo otro gráfico computacional, es posible volver a ejecutar la propagación hacia atrás, diferenciando las derivadas para obtener derivadas más altas. El cálculo de derivadas de orden superior se describe en la sección 6.5.10.

Usaremos el último enfoque y describiremos el algoritmo de propagación hacia atrás en

términos de construcción de un gráfico computacional para las derivadas. Cualquier subconjunto del gráfico puede entonces evaluarse utilizando valores numéricos específicos en un momento posterior. Esto nos permite evitar especificar exactamente cuándo se debe calcular cada operación. En cambio, un motor de evaluación de gráficos genérico puede evaluar cada nodo tan pronto como los valores de sus padres estén disponibles.

La descripción del enfoque basado en símbolo a símbolo subsume el enfoque de símbolo a número. Se puede entender que el enfoque de símbolo a número realiza exactamente los mismos cálculos que se realizan en el gráfico construido por el enfoque de símbolo a símbolo. La diferencia clave es que el enfoque de símbolo a número no expone el gráfico.

6.5.6 Propagación inversa general

El algoritmo de retropropagación es muy simple. Para calcular el gradiente de algún escalar r con respecto a uno de sus antepasados X en el gráfico, comenzamos observando que el gradiente con respecto a X es dado por $\frac{\partial r}{\partial X} = 1$. Entonces podemos calcular el gradiente con respecto a cada parente de X en el gráfico multiplicando el gradiente de corriente por el jacobiano de la operación que produjo Z . Continuamos multiplicando por jacobianos viajando hacia atrás a través de la gráfica de esta manera hasta llegar a X . Para cualquier nodo al que se pueda llegar yendo hacia atrás desde Z a través de dos o más caminos, simplemente sumamos los gradientes que llegan de diferentes caminos a ese nodo.

Más formalmente, cada nodo en el gráfico $GRAMO$ corresponde a una variable. Para lograr la máxima generalidad, describimos esta variable como un tensor V . En general, el tensor puede tener cualquier número de dimensiones. Incluyen escalares, vectores y matrices.

Suponemos que cada variable V está asociado con las siguientes subrutinas:

- **obtener_operacion(V):** Esto devuelve la operación que calcula V , representado por los bordes que entran en V en el gráfico computacional. Por ejemplo, puede haber una clase de Python o C++ que represente la operación de multiplicación de matrices y la función **obtener_operacion**. Supongamos que tenemos una variable que se crea mediante la multiplicación de matrices, $C = AB$. Entonces **obtener_operacion(V)** devuelve un puntero a una instancia de la clase C++ correspondiente.
- **obtener_consumidores($V, GRAMO$):** Esto devuelve la lista de variables que son hijos de V en el gráfico computacional $GRAMO$.
- **obtener_entradas($V, GRAMO$):** Esto devuelve la lista de variables que son padres de V en el gráfico computacional $GRAMO$.

Cada operación opta también se asocia con una propia operación. Este bprop La operación puede calcular un producto vectorial jacobiano como se describe en la ecuación 6.47. Así es como el algoritmo de retropropagación es capaz de lograr una gran generalidad. Cada operación es responsable de saber cómo propagarse hacia atrás a través de los bordes del gráfico en el que participa. Por ejemplo, podríamos usar una operación de multiplicación de matrices para crear una variable $C = AB$. Supongamos que el gradiente de un escalar z con respecto a C es dado por GRAMO . La operación de multiplicación de matrices es responsable de definir dos reglas de propagación hacia atrás, una para cada uno de sus argumentos de entrada. Si llamamos a `abprop` método para solicitar el gradiente con respecto a A dado que el gradiente en la salida es GRAMO , entonces el `bprop` método de la operación de multiplicación de matrices debe establecer que el gradiente con respecto a A es dado por ES . Asimismo, si llamamos a `abprop` método para solicitar el gradiente con respecto a B , entonces la operación matricial es responsable de implementar el `bprop` especificando que el gradiente deseado viene dado por $A \cdot \text{GRAMO}$. El algoritmo de retropropagación en sí mismo no necesita conocer ninguna regla de diferenciación. Solo necesita llamar a cada operación `bprop` con los argumentos correctos. Formalmente, `op.bprop(entradas, X, GRAMO)` debe volver

$$\frac{\partial \text{op.f}(\text{entradas})}{\partial x_i} \text{GRAMO}_i, \quad (6.54)$$

que es solo una implementación de la regla de la cadena como se expresa en la ecuación 6.47. Aquí, `entradas` es una lista de entradas que se suministran a la operación, `op.f` es la función matemática que implementa la operación, `X` es la entrada cuyo gradiente deseamos calcular, y `GRAMO` es el gradiente en la salida de la operación.

El `op.bprop` método siempre debe pretender que todas sus entradas son distintas entre sí, incluso si no lo son. Por ejemplo, si el `Mul` operador se le pasan dos copias de `X` para computar `X2`, el `op.bprop` método aún debería regresar `X` como la derivada con respecto a ambas entradas. El algoritmo de propagación hacia atrás luego agregará ambos argumentos para obtener `2X`, que es la derivada total correcta de `X`.

Las implementaciones de software de propagación hacia atrás generalmente proporcionan tanto las operaciones como sus `bprop` métodos, de modo que los usuarios de bibliotecas de software de aprendizaje profundo puedan propagar hacia atrás a través de gráficos creados utilizando operaciones comunes como multiplicación de matrices, exponentes, logaritmos, etc. Los ingenieros de software que construyen una nueva implementación de retropropagación o los usuarios avanzados que necesitan agregar su propia operación a una biblioteca existente generalmente deben derivar la `op.bprop` método para cualquier nueva operación manualmente.

El algoritmo de retropropagación se describe formalmente en el algoritmo 6.5.

Algoritmo 6.5 El esqueleto más externo del algoritmo de propagación hacia atrás. Esta parte realiza un trabajo sencillo de configuración y limpieza. La mayor parte del trabajo importante ocurre en el `build_gradsubrutina` de algoritmo 6.6.

Requerir: T , el conjunto objetivo de variables cuyos gradientes deben calcularse. **Requerir:** $GRAMO$, el gráfico computacional **Requerir:** z , la variable a diferenciar

Dejar $GRAMO$ ser $GRAMO$ podado para contener solo nodos que son ancestros de y descendientes de nodos en T .

Inicializar `tabla_graduación`, una estructura de datos que asocia tensores a sus gradientes $\text{grad_table}[z] \leftarrow 1$ para $V \in T$ hacer

build_grad($V, G, G, \text{tabla_graduación}$)

fin para

Devolver `grad_table` prohibido para T

En la sección 6.5.2, explicamos que la propagación hacia atrás se desarrolló para evitar calcular la misma subexpresión en la regla de la cadena varias veces. El algoritmo ingenuo podría tener un tiempo de ejecución exponencial debido a estas subexpresiones repetidas. Ahora que hemos especificado el algoritmo de propagación hacia atrás, podemos entender su costo computacional. Si asumimos que la evaluación de cada operación tiene aproximadamente el mismo costo, entonces podemos analizar el costo computacional en términos del número de operaciones ejecutadas. Tenga en cuenta aquí que nos referimos a una operación como la unidad fundamental de nuestro gráfico computacional, que en realidad podría consistir en muchas operaciones aritméticas (por ejemplo, podríamos tener un gráfico que trate la multiplicación de matrices como una sola operación). Cálculo de un gradiente en un gráfico con n nodos nunca ejecutarán más de $O(n^2)$ operaciones o almacenar la salida de más de $O(n^2)$ operaciones. Aquí estamos contando operaciones en el gráfico computacional, no operaciones individuales ejecutadas por el hardware subyacente, por lo que es importante recordar que el tiempo de ejecución de cada operación puede ser muy variable. Por ejemplo, multiplicar dos matrices que contienen millones de entradas podría corresponder a una sola operación en el gráfico. Podemos ver que calcular el gradiente requiere como la mayoría $O(n^2)$ operaciones porque la etapa de propagación hacia adelante, en el peor de los casos, ejecutará todas n nodos en el gráfico original (dependiendo de los valores que queramos calcular, es posible que no necesitemos ejecutar todo el gráfico). El algoritmo de retropropagación agrega un producto vectorial jacobiano, que debe expresarse con $O(1)$ nodos, por arista en el gráfico original. Debido a que el gráfico computacional es un gráfico acíclico dirigido, tiene como máximo $O(n^2)$ bordes. Para los tipos de gráficos que se usan comúnmente en la práctica, la situación es aún mejor. La mayoría de las funciones de costo de las redes neuronales son

Algoritmo 6.6 La subrutina del bucle internobuild_grad(V, G, G , tabla_graduación) del algoritmo de propagación hacia atrás, llamado por el algoritmo de propagación hacia atrás definido en el algoritmo 6.5.

Requerir: V , la variable cuyo gradiente debe agregarse a $GRAMO$ y grad_table. **Requerir:** $GRAMO$, el gráfico a modificar.

Requerir: $GRAMO$, la restricción de $GRAMO$ a los nodos que participan en el gradiente.

Requerir: tabla_graduación, una estructura de datos que asigna nodos a sus gradientes

si Ves engrad_table entonces

Devolvergrad_table[V]

terminara si

$i \leftarrow 1$

para Cen obtener_consumidores($V, GRAMO$) hacer

op \leftarrow obtener_operacion(C)

$D \leftarrow$ build_grad(C, g, g , tabla_graduación) $GRAMO(i)$

\leftarrow op.bprop(obtener_entradas(C, $GRAMO$), V, D) $i \leftarrow i$

+1

fin para

$GRAMO \leftarrow GRAMO(i)$

grad_table[V] = $GRAMO$

Insertar $GRAMO$ y las operaciones que lo crean en $GRAMO$

Devolver $GRAMO$

más o menos estructurado en cadena, lo que hace que la propagación hacia atrás tenga $O(norte)$ costo. Esto es mucho mejor que el enfoque ingenuo, que podría necesitar ejecutar exponencialmente muchos nodos. Este costo potencialmente exponencial se puede ver expandiendo y reescribiendo la regla de la cadena recursiva (ecuación 6.49) no recursivamente:

$$\frac{\partial u_{(norte)}}{\partial u_j} = \sum_{\substack{\text{camino } (tu_{(\pi_1)}tu_{(\pi_2)}, \dots, tu_{(\pi_k)}) \\ \text{de } \pi=j \text{ a } \pi=norte}} \frac{-\partial u_{(\pi_k)}}{\partial u_{(\pi_{k-1})}}. \quad (6.55)$$

Dado que el número de caminos desde el nodo j al nodo $norte$ puede crecer exponencialmente en la longitud de estos caminos, el número de términos en la suma anterior, que es el número de dichos caminos, puede crecer exponencialmente con la profundidad del gráfico de propagación directa. Se incurriría en este gran costo porque el mismo cálculo para

$\frac{\partial u_{(j)}}{\partial u_{(norte)}}$ se reharía muchas veces. Para evitar tal recálculo, podemos pensar en la propagación hacia atrás como un algoritmo de llenado de tablas que aprovecha el almacenamiento

resultados intermedios $\frac{\partial u_{(j)}}{\partial u_{(norte)}}$. Cada nodo en el gráfico tiene una ranura correspondiente en un tabla para almacenar el gradiente para ese nodo. Al completar estas entradas de la tabla en orden,

la retropropagación evita repetir muchas subexpresiones comunes. Esta estrategia de llenar la tabla a veces se llama **programación dinámica**.

6.5.7 Ejemplo: Propagación hacia atrás para entrenamiento MLP

Como ejemplo, repasamos el algoritmo de retropropagación tal como se usa para entrenar un perceptrón multicapa.

Aquí desarrollamos una percepción multicapa muy simple con una sola capa oculta. Para entrenar este modelo, utilizaremos el descenso de gradiente estocástico de minilotes. El algoritmo de retropropagación se usa para calcular el gradiente del costo en un solo minilote. Específicamente, usamos un mini lote de ejemplos del conjunto de entrenamiento formateado como una matriz de diseño X y un vector de etiquetas de clase asociadas y . La red calcula una capa de características ocultas $H = \max\{0, XW^{(1)}\}$. Para simplificar la presentación, no utilizamos sesgos en este modelo. Suponemos que nuestro lenguaje gráfico incluye una operación que puede calcular $\max\{0, Z\}$ elemento sabio. Las predicciones de las probabilidades logarítmicas no normalizadas sobre las clases vienen dadas por $HW^{(2)}$. Suponemos que nuestro lenguaje gráfico incluye una operación de entropía cruzada que calcula la entropía cruzada entre los objetivos y y la distribución de probabilidad definida por estas probabilidades logarítmicas no normalizadas. La entropía cruzada resultante define el costo J_{MLE} . Minimizar esta entropía cruzada realiza una estimación de máxima verosimilitud del clasificador. Sin embargo, para hacer este ejemplo más realista,

también incluimos un plazo de regularización. El costo total

$$J = J_{MLE} + \lambda - \sum_{y_{o,j}} \frac{\partial^2 H^{(1)}}{\partial w_{j,y_{o,j}}}^2 + \sum_{y_{o,j}} \frac{\partial^2 H^{(2)}}{\partial w_{j,y_{o,j}}}^2 \quad (6.56)$$

consiste en la entropía cruzada y un término de decaimiento de peso con coeficiente λ . El gráfico computacional se ilustra en la figura 6.11.

El gráfico computacional para el gradiente de este ejemplo es lo suficientemente grande como para que sea tedioso dibujarlo o leerlo. Esto demuestra uno de los beneficios del algoritmo de propagación hacia atrás, que es que puede generar automáticamente gradientes que serían sencillos pero tediosos para un ingeniero de software para derivar manualmente.

Podemos rastrear aproximadamente el comportamiento del algoritmo de propagación hacia atrás mirando el gráfico de propagación hacia adelante en la figura 6.11. Para entrenar, deseamos calcular ambos $\nabla_{W^{(1)}} j$ y $\nabla_{W^{(2)}} j$. Hay dos caminos diferentes que conducen hacia atrás desde j los pesos: uno a través del costo de entropía cruzada y otro a través del costo de disminución del peso. El costo de disminución de peso es relativamente simple; siempre contribuirá $2\lambda W_j$ al gradiente en ∇_{W_j} .

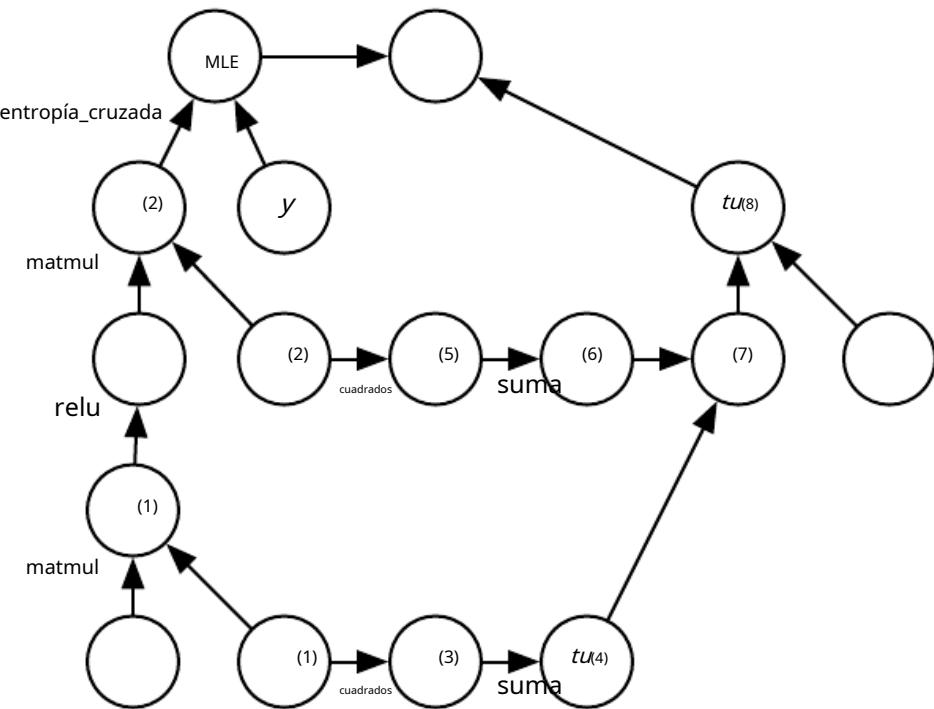


Figura 6.11: El gráfico computacional usado para calcular el costo usado para entrenar nuestro ejemplo de un MLP de una sola capa usando la pérdida de entropía cruzada y la caída de peso.

El otro camino a través del costo de la entropía cruzada es un poco más complicado. Dejar *GRAMO* sea el gradiente en las probabilidades logarítmicas no normalizadas $t_{u(2)}$ proporcionado por el *entropía_cruzada* operación. El algoritmo de propagación hacia atrás ahora necesita explorar dos ramas diferentes. En la rama más corta, agrega *H-GRAMO* al gradiente en $\mathcal{W}(2)$, utilizando la regla de propagación hacia atrás para el segundo argumento de la operación de multiplicación de matrices. La otra rama corresponde a la cadena más larga que desciende más a lo largo de la red. Primero, el algoritmo de retropropagación calcula $\nabla H = GW(2)$ -usando la regla de retropropagación para el primer argumento de la operación de multiplicación de matrices. A continuación, el *relu* La operación utiliza su regla de retropropagación para poner a cero los componentes del gradiente correspondientes a las entradas de $t_{u(1)}$ que eran menores de 0. Que el resultado se llame *GRAMO*. El último paso del algoritmo de propagación hacia atrás es usar la regla de propagación hacia atrás para el segundo argumento de la *matmul* operación para agregar *X-GRAMO* al gradiente en $\mathcal{W}(1)$.

Una vez calculados estos gradientes, es responsabilidad del algoritmo de descenso de gradientes, u otro algoritmo de optimización, utilizar estos gradientes para actualizar los parámetros.

Para el MLP, el costo computacional está dominado por el costo de la multiplicación de matrices. Durante la etapa de propagación directa, multiplicamos por cada peso

matriz, resultando en $O(w)$ sumas y multiplicaciones, donde w es el número de pesos. Durante la etapa de propagación hacia atrás, multiplicamos por la transpuesta de cada matriz de peso, que tiene el mismo costo computacional. El principal costo de memoria del algoritmo es que necesitamos almacenar la entrada en la no linealidad de la capa oculta. Este valor se almacena desde el momento en que se calcula hasta que el pase hacia atrás regresa al mismo punto. El costo de la memoria es por lo tanto $O(Minnesota \cdot h)$, donde m es el número de ejemplos en el minibatch y n es el número de unidades ocultas.

6.5.8 Complicaciones

Nuestra descripción del algoritmo de propagación hacia atrás aquí es más simple que las implementaciones realmente utilizadas en la práctica.

Como se señaló anteriormente, hemos restringido la definición de una operación para que sea una función que devuelva un solo tensor. La mayoría de las implementaciones de software deben admitir operaciones que pueden devolver más de un tensor. Por ejemplo, si deseamos calcular tanto el valor máximo en un tensor como el índice de ese valor, es mejor calcular ambos en un solo paso a través de la memoria, por lo que es más eficiente implementar este procedimiento como una sola operación con dos salidas.

No hemos descrito cómo controlar el consumo de memoria de backpropagation. La retropropagación a menudo implica la suma de muchos tensores juntos. En el enfoque ingenuo, cada uno de estos tensores se calcularía por separado y luego se sumarían todos en un segundo paso. El enfoque ingenuo tiene un cuello de botella de memoria demasiado alto que se puede evitar manteniendo un solo búfer y agregando cada valor a ese búfer a medida que se calcula.

Las implementaciones del mundo real de la propagación hacia atrás también necesitan manejar varios tipos de datos, como punto flotante de 32 bits, punto flotante de 64 bits y valores enteros. La política de manejo de cada uno de estos tipos tiene especial cuidado en el diseño.

Algunas operaciones tienen gradientes indefinidos, y es importante realizar un seguimiento de estos casos y determinar si el gradiente solicitado por el usuario no está definido.

Varios otros tecnicismos hacen que la diferenciación en el mundo real sea más complicada. Estos tecnicismos no son insuperables, y este capítulo ha descrito las herramientas intelectuales clave necesarias para calcular derivadas, pero es importante tener en cuenta que existen muchas más sutilezas.

6.5.9 Diferenciación fuera de la comunidad de aprendizaje profundo

La comunidad de aprendizaje profundo ha estado algo aislada de la comunidad informática más amplia y ha desarrollado en gran medida sus propias actitudes culturales.

acerca de cómo realizar la diferenciación. De manera más general, el campo de la **diferenciación automática** se ocupa de cómo calcular derivadas algorítmicamente. El algoritmo de propagación hacia atrás descrito aquí es solo un enfoque para la diferenciación automática. Es un caso especial de una clase más amplia de técnicas llamadas **acumulación de modo inverso**. Otros enfoques evalúan las subexpresiones de la regla de la cadena en diferentes órdenes. En general, determinar el orden de evaluación que resulte en el costo computacional más bajo es un problema difícil. Encontrar la secuencia óptima de operaciones para calcular el gradiente es NP-completo (Naumann, 2008), en el sentido de que puede requerir la simplificación de expresiones algebraicas en su forma menos costosa.

Por ejemplo, supongamos que tenemos variables $p_{\text{a}1}, p_{\text{a}2}, \dots, p_{\text{a}n}$ representación de probabilidades y variables z_1, z_2, \dots, z_n que representan probabilidades logarítmicas no normalizadas. Suponer definimos

$$q_i = -\frac{\text{Exp}(z_i)}{\sum_j \text{Exp}(z_j)}, \quad (6.57)$$

donde construimos la función softmax a partir de exponenciación, - suma y división operaciones, y construimos una pérdida de entropía cruzada $J = -\sum_i p_{\text{a}i} \log q_i$. Un humano matemático puede observar que la derivada de J con respecto a z_i toma una forma muy simple: $q_i / -p_{\text{a}i}$. El algoritmo de retropropagación no es capaz de simplificar el gradiente de esta manera y, en cambio, propagará explícitamente los gradientes a través de todas las operaciones de logaritmo y exponenciación en el gráfico original. Algunas bibliotecas de software como Theano (Bergstra et al., 2010; bastié et al., 2012) son capaces de realizar algunos tipos de sustitución algebraica para mejorar el gráfico propuesto por el algoritmo de propagación hacia atrás puro.

Cuando el gráfico de avance *GRAM* tiene un solo nodo de salida y cada derivada parcial $\frac{\partial u_i}{\partial v_j}$ puede calcularse con una cantidad constante de cómputo, la propagación hacia atrás garantiza que el número de cómputos para el cálculo de gradiente sea del mismo orden que el número de cómputos para el cálculo hacia adelante: esto

se puede ver en el algoritmo 6.2 porque cada derivada parcial local $\frac{\partial u_i}{\partial v_j}$ entonces calcularse solo una vez junto con una multiplicación y suma asociadas para la formulación recursiva de la regla de la cadena (ecuación 6.49). Por lo tanto, el cálculo general es $O(\# \text{ bordes})$. Sin embargo, potencialmente puede reducirse simplificando el gráfico computacional construido por retropropagación, y esta es una tarea NP-completa. Las implementaciones como Theano y TensorFlow utilizan heurísticas basadas en la coincidencia de patrones de simplificación conocidos para intentar simplificar el gráfico de forma iterativa. Definimos la propagación inversa solo para el cálculo de un gradiente de una salida escalar, pero la propagación inversa se puede extender para calcular un jacobiano (cualquiera de los dos). k diferentes nodos escalares en el gráfico, o de un nodo tensorial que contiene k valores). Una implementación ingenua puede entonces necesitar k veces más cálculo: para

cada nodo interno escalar en el gráfico directo original, la implementación ingenua calcula *k* gradientes en lugar de un solo gradiente. Cuando el número de salidas del gráfico es mayor que el número de entradas, a veces es preferible utilizar otra forma de diferenciación automática llamada **acumulación de modo directo**. Se ha propuesto el cálculo en modo directo para obtener el cálculo en tiempo real de gradientes en redes recurrentes, por ejemplo ([Williams y Zipser, 1989](#)). Esto también evita la necesidad de almacenar los valores y gradientes de todo el gráfico, sacrificando la eficiencia computacional por la memoria. La relación entre el modo hacia adelante y el modo hacia atrás es análoga a la relación entre la multiplicación por la izquierda y la multiplicación por la derecha de una secuencia de matrices, como

$$A B C D, \quad (6.58)$$

donde las matrices se pueden considerar como matrices jacobianas. Por ejemplo, si D es un vector columna mientras que A tiene muchas filas, esto corresponde a un gráfico con una sola salida y muchas entradas, y comenzar las multiplicaciones desde el final y retroceder solo requiere productos matriz-vector. Esto corresponde al modo hacia atrás. En cambio, comenzar a multiplicar desde la izquierda implicaría una serie de productos matriz-matriz, lo que hace que todo el cálculo sea mucho más costoso. Sin embargo, si A tiene menos filas que D tiene columnas, es más barato ejecutar las multiplicaciones de izquierda a derecha, correspondientes al modo directo.

En muchas comunidades fuera del aprendizaje automático, es más común implementar software de diferenciación que actúa directamente sobre el código del lenguaje de programación tradicional, como Python o el código C, y genera automáticamente programas que diferencian funciones escritas en estos lenguajes. En la comunidad de aprendizaje profundo, los gráficos computacionales generalmente se representan mediante estructuras de datos explícitas creadas por bibliotecas especializadas. El enfoque especializado tiene la desventaja de requerir que el desarrollador de la biblioteca defina elbpropmétodos para cada operación y limitando el usuario de la biblioteca a solo aquellas operaciones que han sido definidas. Sin embargo, el enfoque especializado también tiene la ventaja de permitir que se desarrollen reglas de propagación hacia atrás personalizadas para cada operación, lo que permite al desarrollador mejorar la velocidad o la estabilidad de maneras no obvias que, presumiblemente, un procedimiento automático no podría replicar.

Por lo tanto, la propagación hacia atrás no es la única forma ni la forma óptima de calcular el gradiente, pero es un método muy práctico que sigue siendo muy útil para la comunidad de aprendizaje profundo. En el futuro, la tecnología de diferenciación para redes profundas puede mejorar a medida que los profesionales del aprendizaje profundo sean más conscientes de los avances en el campo más amplio de la diferenciación automática.

6.5.10 Derivadas de orden superior

Algunos marcos de software admiten el uso de derivados de orden superior. Entre los marcos de software de aprendizaje profundo, esto incluye al menos Theano y TensorFlow. Estas bibliotecas usan el mismo tipo de estructura de datos para describir las expresiones de derivadas que usan para describir la función original que se está diferenciando. Esto significa que la maquinaria de diferenciación simbólica se puede aplicar a los derivados.

En el contexto del aprendizaje profundo, es raro calcular una sola segunda derivada de una función escalar. En cambio, normalmente estamos interesados en las propiedades de la matriz Hessiana. Si tenemos una función $F: \mathbb{R}^n \rightarrow \mathbb{R}$, entonces la matriz hessiana es de tamaño $n \times n$. En aplicaciones típicas de aprendizaje profundo, n será el número de parámetros en el modelo, que fácilmente podría ascender a miles de millones. Por lo tanto, la matriz hessiana completa es inviable incluso de representar.

En lugar de calcular explícitamente el hessiano, el enfoque típico de aprendizaje profundo es usar **Métodos de Krylov**. Los métodos de Krylov son un conjunto de técnicas iterativas para realizar varias operaciones, como invertir aproximadamente una matriz o encontrar aproximaciones a sus vectores propios o valores propios, sin usar ninguna otra operación que no sean los productos matriz-vector.

Para usar los métodos de Krylov en Hessian, solo necesitamos poder calcular el producto entre la matriz Hessian H y un vector arbitrario v . Una técnica sencilla ([cristianson, 1992](#)) para hacerlo es calcular

$$\text{alto} = \nabla_X (\nabla_X F(X)) \cdot v. \quad (6.59)$$

Ambos cálculos de gradiente en esta expresión pueden ser calculados automáticamente por la biblioteca de software apropiada. Tenga en cuenta que la expresión del gradiente exterior toma el gradiente de una función de la expresión del gradiente interior.

Si v es en sí mismo un vector producido por un gráfico computacional, es importante especificar que el software de diferenciación automática no debe diferenciar a través del gráfico que produjo v .

Aunque normalmente no es aconsejable calcular el hessiano, es posible hacerlo con productos vectoriales hessianos. Uno simplemente calcula \tilde{E}_k para todos $k=1, \dots, n$, donde m_i es el vector caliente con $m_i=1$ y todas las demás entradas iguales a 0.

6.6 Notas históricas

Las redes feedforward pueden verse como aproximadores de funciones no lineales eficientes basados en el uso de descenso de gradiente para minimizar el error en la aproximación de una función.

Desde este punto de vista, la red feedforward moderna es la culminación de siglos de progreso en la tarea de aproximación de funciones generales.

La regla de la cadena que subyace al algoritmo de propagación hacia atrás se inventó en el siglo XVII ([Leibniz, 1676](#); [L'Hôpital, 1696](#)). El cálculo y el álgebra se han utilizado durante mucho tiempo para resolver problemas de optimización en forma cerrada, pero el gradiente descendente no se introdujo como técnica para aproximar iterativamente la solución a los problemas de optimización hasta el siglo XIX. ([cauchy, 1847](#)).

A partir de la década de 1940, estas técnicas de aproximación de funciones se utilizaron para motivar modelos de aprendizaje automático como el perceptrón. Sin embargo, los primeros modelos se basaron en modelos lineales. Los críticos, incluido Marvin Minsky, señalaron varias de las fallas de la familia de modelos lineales, como su incapacidad para aprender la función XOR, lo que provocó una reacción violenta contra todo el enfoque de la red neuronal.

El aprendizaje de funciones no lineales requirió el desarrollo de un perceptrón multicapa y un medio para calcular el gradiente a través de dicho modelo. Las aplicaciones eficientes de la regla de la cadena basadas en programación dinámica comenzaron a aparecer en las décadas de 1960 y 1970, principalmente para aplicaciones de control ([Kelley, 1960](#); [Bryson y Denham, 1961](#); [Dreyfus, 1962](#); [bryson y ho, 1969](#); [Dreyfus, 1973](#)) sino también para el análisis de sensibilidad ([Linnainmaa, 1976](#)). [werbos\(1981\)](#) propusieron aplicar estas técnicas al entrenamiento de redes neuronales artificiales. La idea finalmente se desarrolló en la práctica después de ser redescubierta de forma independiente de diferentes maneras ([lecun , 1985](#); [parker, 1985](#); [Rumelhart et al., 1986a](#)). El libro **Procesamiento distribuido en paralelo** presentó los resultados de algunos de los primeros experimentos exitosos con retropropagación en un capítulo ([Rumelhart et al., 1986b](#)) que contribuyó en gran medida a la popularización de la retropropagación e inició un período muy activo de investigación en redes neuronales multicapa. Sin embargo, las ideas presentadas por los autores de ese libro y en particular por Rumelhart y Hinton van mucho más allá de la retropropagación. Incluyen ideas cruciales sobre la posible implementación computacional de varios aspectos centrales de la cognición y el aprendizaje, que recibieron el nombre de "conexionismo" debido a la importancia que esta escuela de pensamiento otorga a las conexiones entre las neuronas como locus del aprendizaje y la memoria. En particular, estas ideas incluyen la noción de representación distribuida ([Hinton et al., 1986](#)).

Tras el éxito de la retropropagación, la investigación de redes neuronales ganó popularidad y alcanzó su punto máximo a principios de la década de 1990. Posteriormente, otras técnicas de aprendizaje automático se hicieron más populares hasta el renacimiento moderno del aprendizaje profundo que comenzó en 2006.

Las ideas centrales detrás de las modernas redes feedforward no han cambiado sustancialmente desde la década de 1980. El mismo algoritmo de retropropagación y el mismo

Los enfoques para el descenso de gradiente todavía están en uso. La mayor parte de la mejora en el rendimiento de las redes neuronales entre 1986 y 2015 se puede atribuir a dos factores. Primero, los conjuntos de datos más grandes han reducido el grado en que la generalización estadística es un desafío para las redes neuronales. En segundo lugar, las redes neuronales se han vuelto mucho más grandes debido a computadoras más poderosas y una mejor infraestructura de software. Sin embargo, una pequeña cantidad de cambios algorítmicos han mejorado notablemente el rendimiento de las redes neuronales.

Uno de estos cambios algorítmicos fue la sustitución del error cuadrático medio por la familia de funciones de pérdida de entropía cruzada. El error cuadrático medio fue popular en las décadas de 1980 y 1990, pero fue reemplazado gradualmente por pérdidas de entropía cruzada y el principio de máxima verosimilitud a medida que las ideas se extendían entre la comunidad estadística y la comunidad de aprendizaje automático. El uso de pérdidas de entropía cruzada mejoró en gran medida el rendimiento de los modelos con salidas sigmoideas y softmax, que anteriormente habían sufrido saturación y aprendizaje lento al usar la pérdida de error cuadrático medio.

El otro cambio algorítmico importante que mejoró en gran medida el rendimiento de las redes feedforward fue la sustitución de las unidades ocultas sigmoideas por unidades ocultas lineales por partes, como las unidades lineales rectificadas. Rectificación mediante $\max\{0, z\}$. La función se introdujo en los primeros modelos de redes neuronales y se remonta al menos hasta Cognitron y Neocognitron (fukushima, 1975, 1980). Estos primeros modelos no usaban unidades lineales rectificadas, sino que aplicaban la rectificación a funciones no lineales. A pesar de la temprana popularidad de la rectificación, la rectificación fue reemplazada en gran medida por sigmoides en la década de 1980, quizás porque los sigmoides funcionan mejor cuando las redes neuronales son muy pequeñas. A principios de la década de 2000, se evitaron las unidades lineales rectificadas debido a la creencia un tanto supersticiosa de que se deben evitar las funciones de activación con puntos no diferenciables. Esto comenzó a cambiar alrededor de 2009. Jarrett et al. (2009) observó que "el uso de una no linealidad rectificadora es el factor individual más importante para mejorar el rendimiento de un sistema de reconocimiento" entre varios factores diferentes del diseño de arquitectura de red neuronal.

Para pequeños conjuntos de datos, Jarrett et al. (2009) observaron que usar la rectificación de no linealidades es aún más importante que aprender los pesos de las capas ocultas. Los pesos aleatorios son suficientes para propagar información útil a través de una red lineal rectificada, lo que permite que la capa clasificadora en la parte superior aprenda a asignar diferentes vectores de características a identidades de clase.

Cuando hay más datos disponibles, el aprendizaje comienza a extraer suficiente conocimiento útil para superar el rendimiento de los parámetros elegidos al azar. gloria et al. (2011a) mostró que el aprendizaje es mucho más fácil en redes lineales rectificadas profundas que en redes profundas que tienen curvatura o saturación de dos lados en sus funciones de activación.

Las unidades lineales rectificadas también son de interés histórico porque muestran que la neurociencia ha seguido teniendo influencia en el desarrollo de algoritmos de aprendizaje profundo. *gloria et al.*(2011a) motivan unidades lineales rectificadas a partir de consideraciones biológicas. La no linealidad semirectificadora pretendía capturar estas propiedades de las neuronas biológicas: 1) Para algunas entradas, las neuronas biológicas están completamente inactivas. 2) Para algunas entradas, la salida de una neurona biológica es proporcional a su entrada. 3) La mayor parte del tiempo, las neuronas biológicas operan en el régimen en el que están inactivas (es decir, deberían haber**activaciones escasas**).

Cuando comenzó el resurgimiento moderno del aprendizaje profundo en 2006, las redes feedforward seguían teniendo mala reputación. Aproximadamente entre 2006 y 2012, se creía ampliamente que las redes feedforward no funcionarían bien a menos que fueran asistidas por otros modelos, como los modelos probabilísticos. Hoy en día, se sabe que con los recursos y las prácticas de ingeniería correctos, las redes feedforward funcionan muy bien. En la actualidad, el aprendizaje basado en gradientes en las redes feedforward se utiliza como herramienta para desarrollar modelos probabilísticos, como el autocodificador variacional y las redes antagónicas generativas, que se describen en el capítulo20. En lugar de ser vista como una tecnología poco confiable que debe ser respaldada por otras técnicas, el aprendizaje basado en gradientes en redes feedforward se considera desde 2012 como una tecnología poderosa que puede aplicarse a muchas otras tareas de aprendizaje automático. En 2006, la comunidad usó el aprendizaje no supervisado para apoyar el aprendizaje supervisado y ahora, irónicamente, es más común usar el aprendizaje supervisado para apoyar el aprendizaje no supervisado.

Las redes feedforward siguen teniendo un potencial sin explotar. En el futuro, esperamos que se apliquen a muchas más tareas y que los avances en los algoritmos de optimización y el diseño de modelos mejoren aún más su rendimiento. En este capítulo se ha descrito principalmente la familia de modelos de redes neuronales. En los capítulos subsiguientes, veremos cómo usar estos modelos, cómo regularizarlos y capacitarlos.

Capítulo 7

Regularización para Deep Learning

Un problema central en el aprendizaje automático es cómo hacer un algoritmo que funcione bien no solo en los datos de entrenamiento, sino también en las nuevas entradas. Muchas estrategias utilizadas en el aprendizaje automático están diseñadas explícitamente para reducir el error de prueba, posiblemente a expensas de un mayor error de entrenamiento. Estas estrategias se conocen colectivamente como regularización. Como veremos, hay muchas formas de regularización disponibles para el profesional del aprendizaje profundo. De hecho, desarrollar estrategias de regularización más efectivas ha sido uno de los principales esfuerzos de investigación en el campo.

Capítulo 5 introdujo los conceptos básicos de generalización, subajuste, sobreajuste, sesgo, varianza y regularización. Si aún no está familiarizado con estos conceptos, consulte ese capítulo antes de continuar con este.

En este capítulo, describimos la regularización con más detalle, centrándonos en las estrategias de regularización para modelos profundos o modelos que pueden usarse como bloques de construcción para formar modelos profundos.

Algunas secciones de este capítulo tratan sobre conceptos estándar en el aprendizaje automático. Si ya está familiarizado con estos conceptos, no dude en omitir las secciones correspondientes. Sin embargo, la mayor parte de este capítulo se ocupa de la extensión de estos conceptos básicos al caso particular de las redes neuronales.

En la sección 5.2.2, definimos la regularización como “cualquier modificación que hacemos a un algoritmo de aprendizaje que pretende reducir su error de generalización pero no su error de entrenamiento”. Hay muchas estrategias de regularización. Algunos imponen restricciones adicionales a un modelo de aprendizaje automático, como agregar restricciones a los valores de los parámetros. Algunos agregan términos adicionales en la función objetivo que se pueden considerar como correspondientes a una restricción suave en los valores de los parámetros. Si se eligen con cuidado, estas restricciones y penalizaciones adicionales pueden conducir a un mejor rendimiento.

en el conjunto de prueba. A veces, estas restricciones y sanciones están diseñadas para codificar tipos específicos de conocimientos previos. Otras veces, estas restricciones y sanciones están diseñadas para expresar una preferencia genérica por una clase de modelo más simple para promover la generalización. A veces, las penalizaciones y las restricciones son necesarias para determinar un problema indeterminado. Otras formas de regularización, conocidas como métodos de conjunto, combinan múltiples hipótesis que explican los datos de entrenamiento.

En el contexto del aprendizaje profundo, la mayoría de las estrategias de regularización se basan en estimadores de regularización. La regularización de un estimador funciona intercambiando un mayor sesgo por una varianza reducida. Un regularizador eficaz es aquel que hace una operación rentable, reduciendo significativamente la varianza sin aumentar demasiado el sesgo. Cuando discutimos la generalización y el sobreajuste en el capítulo 5, nos enfocamos en tres situaciones, donde la familia de modelos que se entrenaba (1) excluía el verdadero proceso de generación de datos, lo que corresponde a un ajuste insuficiente y sesgo inductor, o (2) coincidía con el verdadero proceso de generación de datos, o (3) incluía el proceso de generación pero también muchos otros posibles procesos de generación: el régimen de sobreajuste donde la varianza en lugar del sesgo domina el error de estimación. El objetivo de la regularización es llevar un modelo del tercer régimen al segundo régimen.

En la práctica, una familia de modelos demasiado compleja no incluye necesariamente la función objetivo o el verdadero proceso de generación de datos, ni siquiera una aproximación cercana de ninguno de los dos. Casi nunca tenemos acceso al verdadero proceso de generación de datos, por lo que nunca podemos saber con certeza si la familia de modelos que se estima incluye el proceso de generación o no. Sin embargo, la mayoría de las aplicaciones de los algoritmos de aprendizaje profundo son para dominios en los que el verdadero proceso de generación de datos está casi con seguridad fuera de la familia del modelo. Los algoritmos de aprendizaje profundo generalmente se aplican a dominios extremadamente complicados, como imágenes, secuencias de audio y texto, para los cuales el proceso de generación real consiste esencialmente en simular todo el universo. Hasta cierto punto, siempre intentamos encajar una clavija cuadrada (el proceso de generación de datos) en un agujero redondo (nuestra familia de modelos).

Lo que esto significa es que controlar la complejidad del modelo no es una simple cuestión de encontrar el modelo del tamaño correcto, con la cantidad correcta de parámetros. En cambio, podríamos encontrar, y de hecho en escenarios prácticos de aprendizaje profundo, casi siempre encontramos, que el modelo que mejor se ajusta (en el sentido de minimizar el error de generalización) es un modelo grande que se ha regularizado adecuadamente.

Ahora revisamos varias estrategias sobre cómo crear un modelo tan grande, profundo y regularizado.

7.1 Sanciones de Norma de Parámetros

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

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

$$J(\theta; X, y) = j(\theta; X, y) + \alpha\Omega(\theta) \quad (7.1)$$

dónde $\alpha \in [0, \infty)$ es un hiperparámetro que pondera la contribución relativa del término de penalización norma, Ω , en relación con la función objetivo estándar j . Configuración $\alpha = 0$ da como resultado ninguna regularización. Valores mayores de α corresponden a una mayor regularización.

Cuando nuestro algoritmo de entrenamiento minimiza la función objetivo regularizada J , disminuirá tanto el objetivo original j en los datos de entrenamiento y alguna medida del tamaño de los parámetros θ (o algún subconjunto de los parámetros). Diferentes opciones para la norma del parámetro Ω puede dar lugar a que se prefieran diferentes soluciones. En esta sección, discutimos los efectos de las diversas normas cuando se utilizan como sanciones en los parámetros del modelo.

Antes de profundizar en el comportamiento de regularización de diferentes normas, notamos que para las redes neuronales, generalmente elegimos usar una penalización de norma de parámetro Ω que penaliza *solo los pesos* de la transformación afín en cada capa y deja los sesgos sin regularizar. Los sesgos normalmente requieren menos datos para ajustarse con precisión que los pesos. Cada peso especifica cómo interactúan dos variables. Ajustar bien el peso requiere observar ambas variables en una variedad de condiciones. Cada sesgo controla una única variable. Esto significa que no inducimos demasiada variación dejando los sesgos sin regularizar. Además, la regularización de los parámetros de sesgo puede introducir una cantidad significativa de subajuste. Por lo tanto, usamos el vector w para indicar todos los pesos que deben ser afectados por una sanción de norma, mientras que el vector θ denota todos los parámetros, incluidos ambos w y los parámetros no regularizados.

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

7.1.1 L_2 Regularización de Parámetros

Ya hemos visto, en la sección 5.2.2, uno de los tipos más simples y comunes de penalización de norma de parámetro: la L_2 pena de norma de parámetro comúnmente conocida como **decaimiento de peso**. Esta estrategia de regularización acerca los pesos al origen¹ añadiendo un término de regularización $\Omega(\theta) = \frac{1}{2}w^2$ a la función objetivo. En otras comunidades académicas, L_2 La regularización también se conoce como **regresión de cresta** o **regularización de Tikhonov**.

Podemos obtener una idea del comportamiento de la regularización del decaimiento del peso estudiando el gradiente de la función objetivo regularizada. Para simplificar la presentación, asumimos que no hay parámetro de sesgo, por lo que θ es solo w . Tal modelo tiene la siguiente función objetivo total:

$$\tilde{J}(w; X, y) = w \cdot w + \frac{\alpha}{2} J(w; X, y), \quad (7.2)$$

con el gradiente de parámetro correspondiente

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_J(w; X, y). \quad (7.3)$$

Para tomar un solo paso de gradiente para actualizar los pesos, realizamos esta actualización:

$$w \leftarrow w - (\alpha w + \nabla_J(w; X, y)). \quad (7.4)$$

Escrito de otra manera, la actualización es:

$$w \leftarrow (1 - \alpha)w - \nabla_J(w; X, y). \quad (7.5)$$

Podemos ver que la adición del término de disminución de peso ha modificado la regla de aprendizaje para reducir multiplicativamente el vector de peso por un factor constante en cada paso, justo antes de realizar la actualización de gradiente habitual. Esto describe lo que sucede en un solo paso. Pero, ¿qué sucede durante todo el curso del entrenamiento?

Simplificaremos aún más el análisis haciendo una aproximación cuadrática a la función objetivo en la vecindad del valor de los pesos que obtiene el mínimo costo de entrenamiento no regularizado, $w^* = \text{mínimo de argumento } J(w)$. Si la función objetivo es verdaderamente cuadrática, como en el caso de ajustar un modelo de regresión lineal con

¹De manera más general, podríamos regularizar los parámetros para que estén cerca de cualquier punto específico en el espacio y, sorprendentemente, aún obtener un efecto de regularización, pero se obtendrán mejores resultados para un valor más cercano al verdadero, siendo cero un valor predeterminado que tiene sentido. cuando no sabemos si el valor correcto debe ser positivo o negativo. Dado que es mucho más común regularizar los parámetros del modelo hacia cero, nos centraremos en este caso especial en nuestra exposición.

error cuadrático medio, entonces la aproximación es perfecta. La aproximación \hat{J} es dada por

$$\hat{J}(\theta) = J(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*), \quad (7.6)$$

dónde H es la matriz hessiana de J con respecto a w evaluado en w^* . No hay término de primer orden en esta aproximación cuadrática, porque w^* se define como un mínimo, donde el gradiente desaparece. Así mismo, porque w^* es la ubicación de un mínimo de J , podemos concluir que H es semidefinido positivo.

el mínimo de \hat{J} ocurre donde su gradiente

$$\nabla_w \hat{J}(w) = H(w - w^*) \quad (7.7)$$

es igual a 0.

Para estudiar el efecto de la disminución del peso, modificamos la ecuación 7.7 añadiendo el gradiente de caída de peso. Ahora podemos resolver el mínimo de la versión regularizada de \hat{J} . Usamos la variable \tilde{w} para representar la ubicación del mínimo.

$$\alpha \tilde{w} + H(w - w^*) = 0 \quad (7.8)$$

$$(H + \alpha I)\tilde{w} = \text{cómo } w^* \quad (7.9)$$

$$\tilde{w} = (H + \alpha I)^{-1} \text{cómo } w^*. \quad (7.10)$$

Como α tiende a 0, la solución regularizada \tilde{w} enfoque w^* . Pero lo que sucede como α crece? Porque H es real y simétrico, podemos descomponerlo en una matriz diagonal Λ y una base ortonormal de vectores propios, q , tal que $H = q\Lambda q^T$. Aplicando la descomposición a la ecuación 7.10, obtenemos:

$$\tilde{w} = (q\Lambda q^T + \alpha I)^{-1} q\Lambda q^T w^* \quad (7.11)$$

$$= q(\Lambda + \alpha I)^{-1} q^T q\Lambda q^T w^* \quad (7.12)$$

$$= q(\Lambda + \alpha I)^{-1} \Lambda q^T w^*. \quad (7.13)$$

Vemos que el efecto de la disminución del peso es volver a escalar w^* a lo largo de los ejes definidos por los vectores propios de H . En concreto, el componente de w^* que está alineado con el i -th autovector de H es reescalado por un factor de $\lambda_i + \alpha$. (Es posible que desee revisar cómo funciona este tipo de escalado, explicado primero en la figura 2.3).

A lo largo de las direcciones donde los valores propios de H son relativamente grandes, por ejemplo, donde $\lambda_i \gg \alpha$, el efecto de la regularización es relativamente pequeño. Sin embargo, los componentes con $\lambda_i \ll \alpha$ se encogerán hasta tener una magnitud casi nula. Este efecto se ilustra en la figura 7.1.

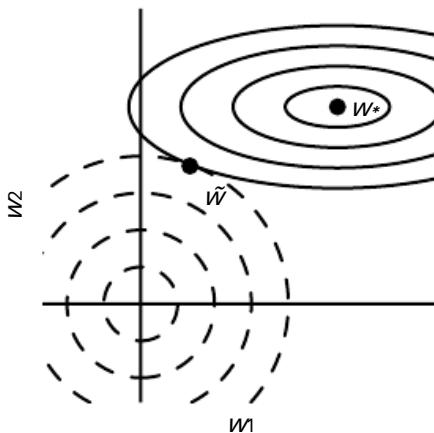


Figura 7.1: Una ilustración del efecto de L_2 (o decaimiento de peso) regularización sobre el valor del óptimo w . Las elipses sólidas representan contornos de igual valor que el objetivo no regularizado. Los círculos punteados representan contornos de igual valor de la L_2 regularizador. En el punto \tilde{w} , estos objetivos en competencia alcanzan un equilibrio. En la primera dimensión, el valor propio de la hessiana de \tilde{w} es pequeño. La función objetivo no aumenta mucho cuando se aleja horizontalmente de w^* . Debido a que la función objetivo no expresa una preferencia fuerte a lo largo de esta dirección, el regularizador tiene un efecto fuerte en este eje. El regularizador tira w cerca de cero. En la segunda dimensión, la función objetivo es muy sensible a los movimientos que se alejan de w^* . El valor propio correspondiente es grande, lo que indica una gran curvatura. Como resultado, la disminución del peso afecta la posición de w relativamente poco

Solo las direcciones a lo largo de las cuales los parámetros contribuyen significativamente a reducir la función objetivo se conservan relativamente intactas. En direcciones que no contribuyen a reducir la función objetivo, un valor propio pequeño de la matriz de Hessiana nos dice que el movimiento en esta dirección no aumentará significativamente el gradiente. Los componentes del vector de peso correspondientes a dichas direcciones sin importancia se descomponen mediante el uso de la regularización a lo largo del entrenamiento.

Hasta ahora hemos discutido el decaimiento del peso en términos de su efecto sobre la optimización de una función de costo cuadrática, general y abstracta. ¿Cómo se relacionan estos efectos con el aprendizaje automático en particular? Podemos averiguarlo estudiando la regresión lineal, un modelo para el cual la verdadera función de costo es cuadrática y, por lo tanto, susceptible al mismo tipo de análisis que hemos usado hasta ahora. Aplicando de nuevo el análisis, podremos obtener un caso especial de los mismos resultados, pero con la solución ahora expresada en términos de los datos de entrenamiento. Para la regresión lineal, la función de costo es

la suma de los errores al cuadrado:

$$(Xw - y) \cdot (Xw - y). \quad (7.14)$$

cuando agregamos L_2 regularización, la función objetivo cambia a

$$(Xw - y) \cdot (Xw - y) + aw \cdot w \cdot \frac{1}{2} \quad (7.15)$$

Esto cambia las ecuaciones normales para la solución de

$$w = (X^T X)^{-1} X^T y \quad (7.16)$$

a

$$w = (X^T X + aI)^{-1} X^T y. \quad (7.17)$$

La matriz $X^T X$ en ecuación 7.16 es proporcional a la matriz de covarianza $\Sigma = \frac{1}{n} X^T X$.

Usando L_2 la regularización reemplaza esta matriz con $X^T X + aI$ en ecuación 7.17. La nueva matriz es la misma que la original, pero con la adición de a a la diagonal. Las entradas diagonales de esta matriz corresponden a la varianza de cada característica de entrada. Podemos ver eso L_2 la regularización hace que el algoritmo de aprendizaje "perciba" la entrada x_i como si tuviera una varianza más alta, lo que hace que se reduzcan los pesos en las características cuya covarianza con el objetivo de salida es baja en comparación con esta varianza agregada.

7.1.2 L_1 regularización

Mientras L_2 el decaimiento de peso es la forma más común de decaimiento de peso, existen otras formas de penalizar el tamaño de los parámetros del modelo. Otra opción es usar L_1 regularización

Formalmente, L_1 regularización en el parámetro del modelo w se define como:

$$\Omega(\theta) = \sum_i |w_i|, \quad (7.18)$$

es decir, como la suma de los valores absolutos de los parámetros individuales.² Ahora hablaremos del efecto de L_1 regularización en el modelo de regresión lineal simple, sin parámetro de sesgo, que estudiamos en nuestro análisis de L_2 regularización. En particular, estamos interesados en delinear las diferencias entre L_1 y L_2 formularios

²Al igual que con L_2 regularización, podríamos regularizar los parámetros hacia un valor que no es cero, sino hacia algún parámetro $w^{(0)}$ que sea el valor $w^{(0)}$. En ese caso el L_1 la regularización sería introducir el término $\Omega(\theta) = \sum_i |w_i - w^{(0)}|$.

de regularización. Al igual que con L_2 caída de peso, L_1 la disminución del peso controla la fuerza de la regularización escalando la penalización Ω utilizando un hiperparámetro positivo α . Así, la función objetivo regularizada $\hat{J}(w; X, y)$ es dado por

$$\hat{J}(w; X, y) = \alpha / \|w\|_1 + J(w; X, y), \quad (7.19)$$

con el gradiente correspondiente (en realidad, sub-gradiente):

$$\nabla_w \hat{J}(w; X, y) = \text{afirmar}(w) + \nabla_w J(X, y; w) \quad (7.20)$$

dónde $\text{firmar}(w)$ es simplemente el signo de w aplicada por elementos.

Al inspeccionar la ecuación 7.20, podemos ver inmediatamente que el efecto de L_1 regularización es bastante diferente de la de L_2 regularización. Específicamente, podemos ver que la contribución de regularización al gradiente ya no escala linealmente con cada w_i ; en cambio, es un factor constante con un signo igual $\text{afirmar}(w_i)$. Una consecuencia de esta forma del gradiente es que no necesariamente veremos soluciones algebraicas limpias para aproximaciones cuadráticas de $J(X, y; w)$ como lo hicimos para L_2 regularización.

Nuestro modelo lineal simple tiene una función de costo cuadrática que podemos representar a través de su serie de Taylor. Alternativamente, podríamos imaginar que esta es una serie de Taylor truncada que se aproxima a la función de costo de un modelo más sofisticado. El gradiente en esta configuración está dado por

$$\nabla_w \hat{J}(w) = H(w - w^*), \quad (7.21)$$

donde, de nuevo, H es la matriz hessiana de J con respecto a w evaluado en w^* .

Porque el L_1 pena no admite expresiones algebraicas limpias en el caso de una arpilla completamente general, también haremos la suposición simplificadora adicional de que la arpilla es diagonal, $H = \text{diag}([H_{1,1}, \dots, H_{n,n}])$, donde cada uno $H_{y,y} > 0$. Esta suposición se mantiene si los datos para el problema de regresión lineal se han preprocesado para eliminar toda correlación entre las características de entrada, lo que se puede lograr mediante PCA.

Nuestra aproximación cuadrática de la L_1 función objetivo regularizada se descompone en una suma sobre los parámetros:

$$\hat{J}(w; X, y) = J(w^*; X, y) + \sum_i \frac{1}{2} H_{y,y} (w_i - w^*_i)^2 + \alpha |w_i|. \quad (7.22)$$

El problema de minimizar esta función de coste aproximado tiene solución analítica (para cada dimensión i), con la siguiente forma:

$$w_i = \text{firmar}(w^*_i) \max(0, w^*_i) - \frac{\alpha}{H_{y,y}}. \quad (7.23)$$

Considere la situación en la que $w^* > 0$ para todos i . Hay dos resultados posibles:

1. El caso en que $w^* \leq \alpha$. Aquí el valor óptimo de w bajo el regularizado el objetivo es simplemente $w=0$. Esto ocurre porque la contribución de $(w; X, y)$ al objetivo regularizado $J(w; X, y)$ está abrumado—en dirección \downarrow por el L_1 regularización que empuja el valor de w a cero.
2. El caso en que $w^* > \alpha$. En este caso, la regularización no desplaza la valor óptimo de w a cero, sino que simplemente lo desplaza en esa dirección una distancia igual a α .

Un proceso similar ocurre cuando $w^* < 0$, pero con el L_1 hacer penaltis w menos negativo por α .

En comparación con L_2 regularización, L_1 la regularización da como resultado una solución que es **más escasa**. La dispersión en este contexto se refiere al hecho de que algunos parámetros tienen un valor óptimo de cero. La escasez de L_1 regularización es un comportamiento cualitativamente diferente al que surge con L_2 regularización Ecuación 7.13 dio la solución \tilde{w} para L_2 regularización Si revisamos esa ecuación usando la suposición de una hessiana definida diagonal y positiva H que introdujimos para nuestro análisis de L_1 regularización, encontramos que $\tilde{w} = H_{yy}^{-1}y - \alpha w^*$. Si w^* era distinto de cero, entonces \tilde{w} restos distinto de cero Esto demuestra que L_2 regularización no hace que los parámetros se vuelvan escasos, mientras que L_1 la regularización puede hacerlo por lo suficientemente grande α .

La propiedad de escasez inducida por L_1 La regularización ha sido ampliamente utilizada como **selección de características** mecanismo. La selección de funciones simplifica un problema de aprendizaje automático al elegir qué subconjunto de las funciones disponibles se debe usar. En particular, el conocido LASSO ([tibshirani, 1995](#)) (menor contracción absoluta y operador de selección) integra un L_1 penalización con un modelo lineal y una función de costo de mínimos cuadrados. El L_1 La penalización hace que un subconjunto de los pesos se convierta en cero, lo que sugiere que las características correspondientes pueden descartarse con seguridad.

En la sección 5.6.1, vimos que muchas estrategias de regularización pueden interpretarse como inferencia MAP Bayesiana, y que en particular, L_2 la regularización es equivalente a la inferencia MAP Bayesian ~~asumiendo~~ previa sobre los pesos. Para L_1 regularización, la pena $\Omega(w) = \alpha \sum_i |w_i|$ utilizado para regularizar una función de costo es equivalente al término logarítmico anterior que se maximiza mediante la inferencia bayesiana de MAP cuando el anterior es una distribución isotrópica de Laplace (ecuación 3.26) encima $w \in \mathbb{R}^{n_{\text{orte}}}$:

$$\text{registro}_{\text{pag}}(w) = -\frac{1}{\alpha} \text{registro de Laplace}(w; 0, \frac{1}{\alpha}) = -\alpha \sum_i |w_i| + n_{\text{orte}} \text{registro}_{\text{a}} - n_{\text{orte}} \text{registro}_{\text{2}}. \quad (7.24)$$

Desde el punto de vista del aprendizaje vía maximización con respecto a w , podemos ignorar la registro a -registro 2 términos porque no dependen de w .

7.2 Sanciones de norma como optimización restringida

Considere la función de costo regularizada por una penalización de norma de parámetro:

$$\mathcal{J}(\theta; X, y) = J(\theta; X, y) + \alpha \Omega(\theta). \quad (7.25)$$

Recuperar de la sección 4.4 que podemos minimizar una función sujeta a restricciones construyendo una función de Lagrange generalizada, que consta de la función objetivo original más un conjunto de penalizaciones. Cada penalización es un producto entre un coeficiente, llamado multiplicador Karush-Kuhn-Tucker (KKT), y una función que representa si se cumple la restricción. Si quisieramos limitar $\Omega(\theta)$ ser menor que alguna constante k , podríamos construir una función de Lagrange generalizada

$$L(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k). \quad (7.26)$$

La solución al problema restringido está dada por

$$\begin{aligned} \theta &= \text{argumento mínimo} \\ \theta & \quad \alpha, \alpha \geq 0 \end{aligned} \quad (7.27)$$

Como se describe en la sección 4.4, resolver este problema requiere modificar ambos θ y α . Sección 4.5 proporciona un ejemplo resuelto de regresión lineal con una L_2 restricción. Son posibles muchos procedimientos diferentes, algunos pueden usar descenso de gradiente, mientras que otros pueden usar soluciones analíticas para donde el gradiente es cero, pero en todos los procedimientos α debe aumentar cada vez que $\Omega(\theta) > k$ y disminuir cada vez que $\Omega(\theta) < k$. todo positivo α alejará $\Omega(\theta)$ reducir. El valor óptimo α alejará $\Omega(\theta)$ para encoger, pero no tan fuerte para hacer $\Omega(\theta)$ volverse menos que k .

Para obtener una idea del efecto de la restricción, podemos corregir α y ver el problema como una función de θ :

$$\begin{aligned} \theta &= \text{mínimo de argumento} \\ \theta & \quad L(\theta, \alpha) = \text{argumento mínimo} \\ \theta & \quad J(\theta; X, y) + \alpha \cdot \Omega(\theta). \end{aligned} \quad (7.28)$$

Esto es exactamente lo mismo que el problema del entrenamiento regularizado de minimizar J . Por lo tanto, podemos pensar en una penalización de norma de parámetro como una restricción sobre los pesos. Si Ω es la L_2 norma, entonces los pesos están obligados a estar en una L_2 pelota. Si Ω es la L_1 norma, entonces los pesos están obligados a estar en una región de

limitado L_1 norma. Por lo general, no conocemos el tamaño de la región de restricción que imponemos al usar el decaimiento de peso con coeficiente α porque el valor de α no nos dice directamente el valor de k . En principio, se puede resolver para k , pero la relación entre k y α depende de la forma de J . Si bien no conocemos el tamaño exacto de la región de restricción, podemos controlarla más o menos aumentando o disminuyendo α para aumentar o reducir la región de restricción. α más grande dará como resultado una región de restricción más pequeña. α menor resultará en una región de restricción más grande.

A veces es posible que deseemos utilizar restricciones explícitas en lugar de sanciones. Como se describe en la sección 4.4, podemos modificar algoritmos como el descenso de gradiente estocástico para dar un paso cuesta abajo en $J(\theta)$ y luego proyectar θ volver al punto más cercano que satisface $\|\theta\| \leq k$. Esto puede ser útil si tenemos una idea de qué valor de k es apropiado y no quiere perder el tiempo buscando el valor de α que corresponde a esto k .

Otra razón para usar restricciones explícitas y reproyección en lugar de imponer restricciones con penalizaciones es que las penalizaciones pueden hacer que los procedimientos de optimización no convexos se atasquen en mínimos locales correspondientes a pequeñas θ . Cuando se entrena redes neuronales, esto generalmente se manifiesta como redes neuronales que entrena con varias "unidades muertas". Estas son unidades que no contribuyen mucho al comportamiento de la función aprendida por la red porque los pesos que entran o salen de ellas son todos muy pequeños. Cuando se entrena con una penalización sobre la norma de los pesos, estas configuraciones pueden ser localmente óptimas, incluso si es posible reducir significativamente J haciendo los pesos más grandes. Las restricciones explícitas implementadas por reproyección pueden funcionar mucho mejor en estos casos porque no fomentan que los pesos se acerquen al origen. Las restricciones explícitas implementadas por reproyección solo tienen efecto cuando los pesos se vuelven grandes e intentan salir de la región de restricción.

Finalmente, las restricciones explícitas con reproyección pueden ser útiles porque imponen cierta estabilidad en el procedimiento de optimización. Cuando se utilizan altas tasas de aprendizaje, es posible encontrar un ciclo de retroalimentación positiva en el que los pesos grandes inducen grandes gradientes que luego inducen una gran actualización de los pesos. Si estas actualizaciones aumentan constantemente el tamaño de los pesos, entonces θ se aleja rápidamente del origen hasta que se produce un desbordamiento numérico. Las restricciones explícitas con la reproyección evitan que este ciclo de retroalimentación continúe aumentando la magnitud de los pesos sin límites. Hinton et al. (2012c) recomiendan el uso de restricciones combinadas con una alta tasa de aprendizaje para permitir una exploración rápida del espacio de parámetros mientras se mantiene cierta estabilidad.

En particular, Hinton et al. (2012c) recomiendan una estrategia introducida por Srebro y Shraibman (2005): restringiendo la norma de cada columna de la matriz de pesos

de una capa de red neuronal, en lugar de restringir la norma de Frobenius de toda la matriz de pesos. Restringir la norma de cada columna por separado evita que cualquier unidad oculta tenga pesos muy grandes. Si convertimos esta restricción en una penalización en una función de Lagrange, sería similar a L_2 -disminución del peso pero con un multiplicador KKT separado para los pesos de cada unidad oculta. Cada uno de estos multiplicadores KKT se actualizaría dinámicamente por separado para hacer que cada unidad oculta obedezca la restricción. En la práctica, la limitación de norma de columna siempre se implementa como una restricción explícita con reprojeción.

7.3 Regularización y problemas de restricciones insuficientes

En algunos casos, la regularización es necesaria para que los problemas de aprendizaje automático se definan correctamente. Muchos modelos lineales en aprendizaje automático, incluida la regresión lineal y PCA, dependen de la inversión de la matriz $X^T X$. Esto no es posible siempre que $X^T X$ sea singular. Esta matriz puede ser singular siempre que la distribución de generación de datos realmente no tenga varianza en alguna dirección, o cuando no haya varianza. *observado* en alguna dirección porque hay menos ejemplos (filas de X) que las características de entrada (columnas de X). En este caso, muchas formas de regularización corresponden a invertir $X^T X + \alpha I$ en cambio. Se garantiza que esta matriz regularizada es invertible.

Estos problemas lineales tienen soluciones de forma cerrada cuando la matriz relevante es invertible. También es posible que un problema sin solución de forma cerrada esté subdeterminado. Un ejemplo es la regresión logística aplicada a un problema donde las clases son linealmente separables. Si un vector de pesos es capaz de lograr una clasificación perfecta, entonces también logrará una clasificación perfecta y una mayor probabilidad. Un procedimiento de optimización iterativo como el descenso de gradiente estocástico aumentará continuamente la magnitud de w , en teoría, nunca se detendrá. En la práctica, una implementación numérica del descenso de gradiente eventualmente alcanzará pesos lo suficientemente grandes como para causar un desbordamiento numérico, momento en el cual su comportamiento dependerá de cómo el programador haya decidido manejar los valores que no son números reales.

La mayoría de las formas de regularización pueden garantizar la convergencia de métodos iterativos aplicados a problemas indeterminados. Por ejemplo, la caída del peso hará que el descenso del gradiente deje de aumentar la magnitud de los pesos cuando la pendiente de la probabilidad sea igual al coeficiente de caída del peso.

La idea de usar la regularización para resolver problemas indeterminados se extiende más allá del aprendizaje automático. La misma idea es útil para varios problemas básicos de álgebra lineal.

Como vimos en la sección 2.9, podemos resolver ecuaciones lineales indeterminadas usando

la pseudoinversa de Moore-Penrose. Recuerde que una definición de la pseudoinversa X^+ de una matriz X es

$$X^+ = \lim_{\alpha \rightarrow 0} (X^T X + \alpha I)^{-1} X^T. \quad (7.29)$$

Ahora podemos reconocer la ecuación 7.29 como realizar una regresión lineal con caída de peso. Específicamente, la ecuación 7.29 es el límite de la ecuación 7.17a medida que el coeficiente de regularización se reduce a cero. Por lo tanto, podemos interpretar la pseudoinversa como la estabilización de problemas indeterminados mediante la regularización.

7.4 Aumento de conjuntos de datos

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

Este enfoque es el más fácil para la clasificación. Un clasificador necesita tomar una entrada complicada y de alta dimensión. Y resúmalo con una sola categoría de identidad. Esto significa que la principal tarea a la que se enfrenta un clasificador es ser invariante frente a una amplia variedad de transformaciones. Podemos generar nuevos (X, y) se empareja fácilmente simplemente transformando las entradas en nuestro conjunto de entrenamiento.

Este enfoque no es tan fácilmente aplicable a muchas otras tareas. Por ejemplo, es difícil generar nuevos datos falsos para una tarea de estimación de densidad a menos que ya hayamos resuelto el problema de estimación de densidad.

El aumento de conjuntos de datos ha sido una técnica particularmente efectiva para un problema de clasificación específico: el reconocimiento de objetos. Las imágenes tienen muchas dimensiones e incluyen una enorme variedad de factores de variación, muchos de los cuales se pueden simular fácilmente. Operaciones como traducir las imágenes de entrenamiento unos pocos píxeles en cada dirección a menudo pueden mejorar en gran medida la generalización, incluso si el modelo ya se ha diseñado para ser parcialmente invariable a la traducción mediante el uso de las técnicas de convolución y agrupación descritas en el capítulo 9. Muchas otras operaciones, como rotar la imagen o escalar la imagen, también han demostrado ser bastante efectivas.

Se debe tener cuidado de no aplicar transformaciones que cambiarían la clase correcta. Por ejemplo, las tareas de reconocimiento óptico de caracteres requieren reconocer la diferencia entre 'b' y 'd' y la diferencia entre '6' y '9', por lo que los giros horizontales y 180° las rotaciones no son formas apropiadas de aumentar los conjuntos de datos para estas tareas.

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

El aumento de conjuntos de datos también es efectivo para tareas de reconocimiento de voz ([Jaityl y Hinton, 2013](#)).

Inyectar ruido en la entrada a una red neuronal ([Sietsma y Dow, 1991](#)) también puede verse como una forma de aumento de datos. Para muchas tareas de clasificación e incluso algunas de regresión, la tarea aún debería ser posible de resolver incluso si se agrega un pequeño ruido aleatorio a la entrada. Sin embargo, las redes neuronales demuestran no ser muy resistentes al ruido ([Tang y Eliasmith, 2010](#)). Una forma de mejorar la solidez de las redes neuronales es simplemente entrenarlas con ruido aleatorio aplicado a sus entradas. La inyección de ruido de entrada es parte de algunos algoritmos de aprendizaje no supervisados, como el codificador automático de eliminación de ruido ([Vicente et al., 2008](#)). La inyección de ruido también funciona cuando el ruido se aplica a las unidades ocultas, lo que puede verse como un aumento del conjunto de datos en múltiples niveles de abstracción. [piscina et al. \(2014\)](#) mostró recientemente que este enfoque puede ser muy eficaz siempre que la magnitud del ruido se ajuste cuidadosamente. La deserción, una poderosa estrategia de regularización que se describirá en la sección [7.12](#), puede ser visto como un proceso de construcción de nuevos insumos por *multiplicando por ruido*

Al comparar los resultados de las pruebas comparativas de aprendizaje automático, es importante tener en cuenta el efecto del aumento del conjunto de datos. A menudo, los esquemas de aumento de conjuntos de datos diseñados a mano pueden reducir drásticamente el error de generalización de una técnica de aprendizaje automático. Para comparar el rendimiento de un algoritmo de aprendizaje automático con otro, es necesario realizar experimentos controlados. Al comparar el algoritmo de aprendizaje automático A y el algoritmo de aprendizaje automático B, es necesario asegurarse de que ambos algoritmos se evaluaron utilizando los mismos esquemas de aumento de conjuntos de datos diseñados a mano. Suponga que el algoritmo A funciona mal sin aumento del conjunto de datos y el algoritmo B funciona bien cuando se combina con numerosas transformaciones sintéticas de la entrada. En tal caso, es probable que las transformaciones sintéticas hayan causado la mejora del rendimiento, en lugar del uso del algoritmo de aprendizaje automático B. A veces, decidir si un experimento se ha controlado correctamente requiere un juicio subjetivo. Por ejemplo, los algoritmos de aprendizaje automático que inyectan ruido en la entrada están realizando una forma de aumento del conjunto de datos. Por lo general, las operaciones que son de aplicación general (como agregar ruido gaussiano a la entrada) se consideran parte del algoritmo de aprendizaje automático, mientras que las operaciones que son específicas de un dominio de aplicación (como recortar aleatoriamente una imagen) se consideran como parte previa separada. pasos de procesamiento.

7.5 Robustez al ruido

Sección 7.4 ha motivado el uso de ruido aplicado a las entradas como estrategia de aumento de conjuntos de datos. Para algunos modelos, la adición de ruido con varianza infinitesimal en la entrada del modelo equivale a imponer una penalización a la norma de los pesos ([obispo, 1995a,b](#)). En el caso general, es importante recordar que la inyección de ruido puede ser mucho más poderosa que simplemente reducir los parámetros, especialmente cuando el ruido se agrega a las unidades ocultas. El ruido aplicado a las unidades ocultas es un tema tan importante que merece su propia discusión por separado; el algoritmo de abandono descrito en la sección 7.12 es el desarrollo principal de ese enfoque.

Otra forma en que se ha utilizado el ruido al servicio de la regularización de modelos es añadiéndolo a los pesos. Esta técnica se ha utilizado principalmente en el contexto de las redes neuronales recurrentes ([Jim et al., 1996](#); [Tumbas, 2011](#)). Esto puede interpretarse como una implementación estocástica de la inferencia bayesiana sobre los pesos. El tratamiento bayesiano del aprendizaje consideraría que los pesos del modelo son inciertos y representables a través de una distribución de probabilidad que refleje esta incertidumbre. Agregar ruido a los pesos es una forma práctica y estocástica de reflejar esta incertidumbre.

El ruido aplicado a los pesos también puede interpretarse como equivalente (bajo algunos supuestos) a una forma más tradicional de regularización, fomentando la estabilidad de la función a aprender. Considere la configuración de regresión, donde deseamos entrenar una función $\hat{y}(X)$ que mapea un conjunto de características X a un escalar usando la función de costo de mínimos cuadrados entre las predicciones del modelo $\hat{y}(X)$ y los verdaderos valores y :

$$J = \text{mi_pag}(x, y)(\hat{y}(X) - y)^2. \quad (7.30)$$

El conjunto de entrenamiento consta de $metro$ ejemplos etiquetados $\{(X_{(1)}, y_{(1)}), \dots, (X_{(metro)}, y_{(metro)})\}$.

Ahora asumimos que con cada presentación de entrada también incluimos una perturbación aleatoria $w \sim \text{norte}(-0, \eta)$ de los pesos de la red. Imaginemos que tenemos un estándaryo-capas MLP. Denotamos el modelo perturbado como $\hat{y}_w(X)$. A pesar de la inyección de ruido, seguimos interesados en minimizar el error cuadrático de la salida de la red. La función objetivo se convierte así en:

$$J_w = \text{mi_pag}(x, y, -w)(\hat{y}_w(X) - y)^2 \quad (7.31)$$

$$= \text{mi_pag}(x, y, -w) \hat{y}_w^2(X) - 2y\hat{y}_w - wX + y^2. \quad (7.32)$$

Para pequeños η , la minimización de J_w con ruido de peso añadido (con covarianza ηI) es equivalente a la minimización de J con un plazo adicional de regularización:

$\eta_{mi,pag(X,y)} - \nabla_w \hat{J}(X)$ 2. Esta forma de regularización favorece que los parámetros se ir a regiones del espacio de parámetros donde las pequeñas perturbaciones de los pesos tienen una influencia relativamente pequeña en la salida. En otras palabras, empuja el modelo hacia regiones donde el modelo es relativamente insensible a pequeñas variaciones en los pesos, encontrando puntos que no son simplemente mínimos, sino mínimos rodeados de regiones planas (Hochreiter y Schmidhuber, 1995). En el caso simplificado de regresión lineal (w , por ejemplo, $\hat{J}(X) = w \cdot X + b$), este término de regularización colapsa en $\eta_{mi,pag(X)} - X^2$, que no es una función de los parámetros y por lo tanto no contribuyen al gradiente de J con respecto a los parámetros del modelo.

7.5.1 Inyección de ruido en los objetivos de salida

La mayoría de los conjuntos de datos tienen una cierta cantidad de errores en las etiquetas. Puede ser perjudicial maximizar $\text{registro}_{pag}(y / X)$ cuando y es un error. Una forma de evitar esto es modelar explícitamente el ruido en las etiquetas. Por ejemplo, podemos suponer que para alguna pequeña constante ϵ , la etiqueta del conjunto de entrenamiento y es correcta con probabilidad $1 - \epsilon$, y de lo contrario, cualquiera de las otras etiquetas posibles podría ser correcta. Esta suposición es fácil de incorporar analíticamente a la función de costo, en lugar de extraer explícitamente muestras de ruido. Por ejemplo, **suavizado de etiquetas** regulariza un modelo basado en un softmax con k valores de salida reemplazando el disco duro 0 y 1 objetivos de clasificación con objetivos de ϵ y $1 - \epsilon$, respectivamente. La pérdida de entropía cruzada estándar puede luego ser utilizado con estos objetivos blandos. Es posible que el aprendizaje de máxima probabilidad con un clasificador softmax y objetivos duros en realidad nunca converja; el softmax nunca puede predecir una probabilidad de exactamente 0 o exactamente 1, por lo que seguirá aprendiendo pesos cada vez mayores, haciendo predicciones más extremas para siempre. Es posible prevenir este escenario utilizando otras estrategias de regularización como la disminución de peso. El suavizado de etiquetas tiene la ventaja de evitar la búsqueda de probabilidades difíciles sin desalentar la clasificación correcta. Esta estrategia se ha utilizado desde la década de 1980 y sigue ocupando un lugar destacado en las redes neuronales modernas (Szegedy et al., 2015).

7.6 Aprendizaje semisupervisado

En el paradigma del aprendizaje semisupervisado, tanto los ejemplos no etiquetados de $PAG(X)$ y etiquetados ejemplos de $PAG(X, y)$ se utilizan para estimar $PAG(y / X)$ o predecir y de X .

En el contexto del aprendizaje profundo, el aprendizaje semisupervisado generalmente se refiere al aprendizaje de una representación $h = f(X)$. El objetivo es aprender una representación para que

que los ejemplos de la misma clase tienen representaciones similares. El aprendizaje no supervisado puede proporcionar pistas útiles sobre cómo agrupar ejemplos en el espacio de representación. Los ejemplos que se agrupan estrechamente en el espacio de entrada deben asignarse a representaciones similares. Un clasificador lineal en el nuevo espacio puede lograr una mejor generalización en muchos casos (Belkin y Niyogi, 2002; capilla et al., 2003). Una variante de larga data de este enfoque es la aplicación del análisis de componentes principales como un paso de preprocesamiento antes de aplicar un clasificador (en los datos proyectados).

En lugar de tener componentes supervisados y no supervisados separados en el modelo, uno puede construir modelos en los que un modelo generativo de cualquiera $PAG(X)$ o $PAG(X,y)$ comparte parámetros con un modelo discriminativo de $PAG(y/X)$. Entonces se puede compensar el criterio supervisado – registro $PAG(y/X)$ con el no supervisado o generativo (como – registro $PAG(X)$ o – registro $PAG(X,y)$). El criterio generativo expresa entonces una forma particular de creencia previa acerca de la solución al problema de aprendizaje supervisado (Lasserre et al., 2006), a saber, que la estructura de $PAG(X)$ está conectado a la estructura de $PAG(y/X)$ de una manera que es capturada por la parametrización compartida. Al controlar cuánto del criterio generativo se incluye en el criterio total, se puede encontrar una mejor compensación que con un criterio de entrenamiento puramente generativo o puramente discriminatorio (Lasserre et al., 2006; Larochelle y Bengio, 2008).

Salakhutdinov y Hinton (2008) describen un método para aprender la función del núcleo de una máquina del núcleo utilizada para la regresión, en el que el uso de ejemplos no etiquetados para el modelado $PAG(X)$ mejora $PAG(y/X)$ bastante significativamente.

Ver capilla et al. (2006) para obtener más información sobre el aprendizaje semisupervisado.

7.7 Aprendizaje multitarea

Aprendizaje multitarea (caruana, 1993) es una forma de mejorar la generalización al agrupar los ejemplos (que pueden verse como restricciones blandas impuestas a los parámetros) que surgen de varias tareas. De la misma manera que los ejemplos de entrenamiento adicionales ejercen más presión sobre los parámetros del modelo hacia valores que se generalicen bien, cuando parte de un modelo se comparte entre tareas, esa parte del modelo está más restringida hacia buenos valores (suponiendo que el intercambio esté justificado).), a menudo dando una mejor generalización.

Cifra 7.2 ilustra una forma muy común de aprendizaje multitarea, en la que diferentes tareas supervisadas (predecir y_i dado X_i) compartir la misma entrada X , así como alguna representación de nivel intermedio $h_{\text{compartido}}$ capturando un grupo común de

factores El modelo generalmente se puede dividir en dos tipos de partes y parámetros asociados:

1. Parámetros específicos de la tarea (que solo se benefician de los ejemplos de su tarea para lograr una buena generalización). Estas son las capas superiores de la red neuronal en la figura.[7.2](#).
2. Parámetros genéricos, compartidos entre todas las tareas (que se benefician de los datos agrupados de todas las tareas). Estas son las capas inferiores de la red neuronal en la figura.[7.2](#).

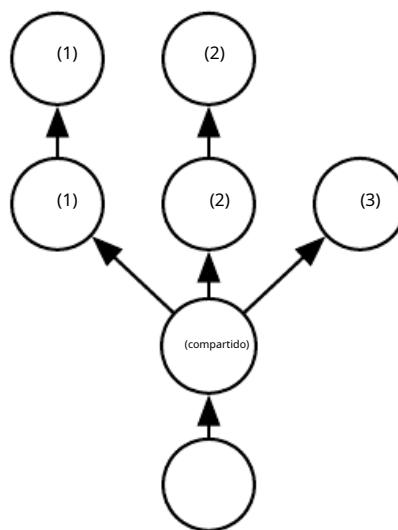


Figura 7.2: El aprendizaje multitarea se puede transmitir de varias maneras en marcos de aprendizaje profundo y esta figura ilustra la situación común en la que las tareas comparten una entrada común pero involucran diferentes variables aleatorias de destino. Las capas inferiores de una red profunda (ya sea supervisada y realimentada o incluye un componente generativo con flechas hacia abajo) se pueden compartir entre dichas tareas, mientras que los parámetros específicos de la tarea (asociados respectivamente con los pesos hacia y desde $h_{(1)}$ y $h_{(2)}$) se pueden aprender además de los que producen una representación compartida $h_{(\text{compartido})}$. La suposición subyacente es que existe un conjunto común de factores que explican las variaciones en la entrada X , mientras que cada tarea está asociada con un subconjunto de estos factores. En este ejemplo, se supone además que las unidades ocultas de nivel superior $h_{(1)}$ y $h_{(2)}$ están especializados para cada tarea (prediciendo respectivamente $y_{(1)}$ y $y_{(2)}$) mientras que alguna representación de nivel intermedio $h_{(\text{compartido})}$ se comparte entre todas las tareas. En el contexto de aprendizaje no supervisado, tiene sentido que algunos de los factores de nivel superior no estén asociados con ninguna de las tareas de salida ($h_{(3)}$): estos son los factores que explican algunas de las variaciones de entrada pero no son relevantes para predecir $y_{(1)}$ o $y_{(2)}$.

Límites de error de generalización y generalización mejorados ([Baxter, 1995](#)) puede lograrse debido a los parámetros compartidos, para los cuales la fuerza estadística puede ser

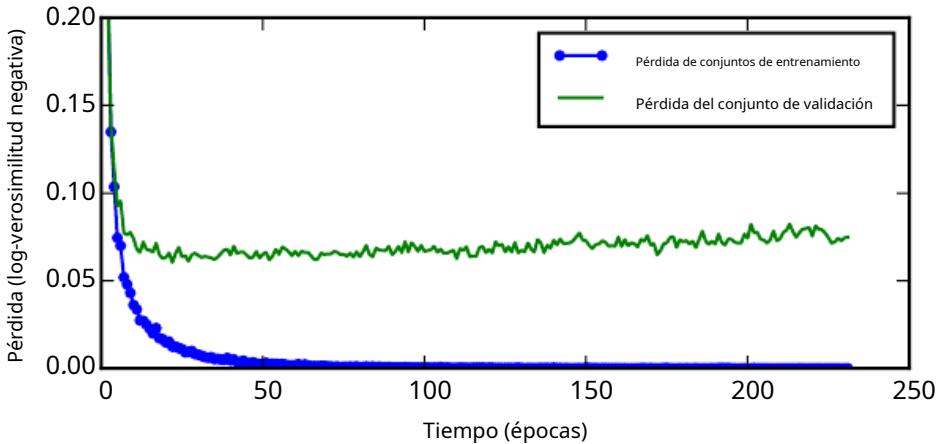


Figura 7.3: Curvas de aprendizaje que muestran cómo cambia la pérdida de probabilidad logarítmica negativa con el tiempo (indicada como número de iteraciones de entrenamiento sobre el conjunto de datos, o **épocas**). En este ejemplo, entrenamos una red maxout en MNIST. Observe que el objetivo de entrenamiento disminuye constantemente con el tiempo, pero la pérdida promedio del conjunto de validación eventualmente comienza a aumentar nuevamente, formando una curva asimétrica en forma de U.

mejorado mucho (en proporción con el mayor número de ejemplos para los parámetros compartidos, en comparación con el escenario de los modelos de una sola tarea). Por supuesto, esto sucederá solo si algunas suposiciones sobre la relación estadística entre las diferentes tareas son válidas, lo que significa que hay algo compartido entre algunas de las tareas.

Desde el punto de vista del aprendizaje profundo, la creencia previa subyacente es la siguiente: *entre los factores que explican las variaciones observadas en los datos asociados a las diferentes tareas, algunos son compartidos entre dos o más tareas.*

7.8 Detención anticipada

Cuando entrenamos modelos grandes con suficiente capacidad de representación para sobreajustar la tarea, a menudo observamos que el error de entrenamiento disminuye constantemente con el tiempo, pero el error del conjunto de validación comienza a aumentar nuevamente. Ver figura 7.3 para ver un ejemplo de este comportamiento. Este comportamiento se produce de forma muy fiable.

Esto significa que podemos obtener un modelo con un mejor error de conjunto de validación (y, por lo tanto, con suerte, un mejor error de conjunto de prueba) volviendo a la configuración del parámetro en el momento con el error de conjunto de validación más bajo. Cada vez que mejora el error en el conjunto de validación, almacenamos una copia de los parámetros del modelo. Cuando finaliza el algoritmo de entrenamiento, devolvemos estos parámetros, en lugar de los últimos parámetros. El

El algoritmo termina cuando ningún parámetro ha mejorado sobre el mejor error de validación registrado para un número de iteraciones pree especificado. Este procedimiento se especifica más formalmente en el algoritmo.7.1.

Algoritmo 7.1 El meta-algoritmo de parada temprana para determinar la mejor cantidad de tiempo para entrenar. Este metaalgoritmo es una estrategia general que funciona bien con una variedad de algoritmos de entrenamiento y formas de cuantificar el error en el conjunto de validación.

Dejar *n* sea el n mero de pasos entre evaluaciones.

Dejar `patience` la "pacienza", el número de veces para observar el empeoramiento del error del conjunto de validación antes de darse por vencido.

Dejar θ_0 ser los parámetros iniciales.

$$\theta \leftarrow \theta_0$$

$i \leftarrow 0$

$j \leftarrow 0$

$V \leftarrow \infty$

$$\theta_* \leftarrow \theta$$

$j^* \leftarrow j$

mientras *j* < *pag* hacer

Actualizar θ ejecutando el algoritmo de entrenamiento para n pasos. $i \leftarrow i + 1$

+norte

$v \leftarrow \text{ValidationSetError}(\theta)$ siv

-<ventones

$j \leftarrow 0$

$$\theta_* \leftarrow \theta$$

$i^* \leftarrow i$

$V \leftarrow V -$

demás

$j \leftarrow j + 1$

terminara si

terminar mientras

Los mejores parámetros son θ_* , el mejor número de pasos de entrenamiento es i_* .

Esta estrategia se conoce como **parada temprana**. Es probablemente la forma de regularización más utilizada en el aprendizaje profundo. Su popularidad se debe tanto a su eficacia como a su sencillez.

Una forma de pensar en la detención anticipada es como un algoritmo de selección de hiperparámetros muy eficiente. En esta vista, el número de pasos de entrenamiento es solo otro hiperparámetro. podemos ver en la figura 7.3 que este hiperparámetro tiene un conjunto de validación en forma de U

curva de rendimiento. La mayoría de los hiperparámetros que controlan la capacidad del modelo tienen una curva de rendimiento del conjunto de validación en forma de U, como se ilustra en la figura 5.3. En el caso de la parada temprana, estamos controlando la capacidad efectiva del modelo determinando cuántos pasos puede tomar para ajustarse al conjunto de entrenamiento. La mayoría de los hiperparámetros deben elegirse mediante un costoso proceso de adivinar y verificar, en el que establecemos un hiperparámetro al comienzo del entrenamiento y luego ejecutamos el entrenamiento en varios pasos para ver su efecto. El hiperparámetro de "tiempo de entrenamiento" es único porque, por definición, una sola ejecución de entrenamiento prueba muchos valores del hiperparámetro. El único costo significativo de elegir este hiperparámetro automáticamente a través de la detención anticipada es ejecutar la evaluación del conjunto de validación periódicamente durante el entrenamiento. Idealmente, esto se hace en paralelo al proceso de entrenamiento en una máquina separada, CPU separada o GPU separada del proceso de entrenamiento principal. Si tales recursos no están disponibles,

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

La detención anticipada es una forma de regularización muy discreta, ya que casi no requiere cambios en el procedimiento de entrenamiento subyacente, la función objetivo o el conjunto de valores de parámetros permitidos. Esto significa que es fácil usar la parada temprana sin dañar la dinámica de aprendizaje. Esto contrasta con el decaimiento de peso, donde se debe tener cuidado de no usar demasiado decaimiento de peso y atrapar la red en un mínimo local incorrecto correspondiente a una solución con pesos patológicamente pequeños.

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

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

Una estrategia (algoritmo 7.2) es inicializar el modelo nuevamente y volver a entrenar en todos

de los datos En este segundo pase de entrenamiento, entrenamos para el mismo número de pasos que el procedimiento de parada anticipada determinado como óptimo en el primer pase. Hay algunas sutilezas asociadas con este procedimiento. Por ejemplo, no hay una buena manera de saber si volver a entrenar para la misma cantidad de actualizaciones de parámetros o la misma cantidad de pasos a través del conjunto de datos. En la segunda ronda de entrenamiento, cada pasada por el conjunto de datos requerirá más actualizaciones de parámetros porque el conjunto de entrenamiento es más grande.

Algoritmo 7.2 Un meta-algoritmo para usar la detención temprana para determinar cuánto tiempo entrenar y luego volver a entrenar en todos los datos.

Dejar $X_{(tren)}$ y $y_{(tren)}$ ser el conjunto de entrenamiento.

Dividir $X_{(tren)}$ y $y_{(tren)}$ en $(X_{(\text{subtren})}, X_{(\text{válido})})$ y $(y_{(\text{subtren})}, y_{(\text{válido})})$

respectivamente.

Ejecutar parada anticipada (algoritmo 7.1) a partir de al azar θ usando $X_{(\text{subtren})}$ y $y_{(\text{subtren})}$ para datos de entrenamiento y $X_{(\text{válido})}$ y $y_{(\text{válido})}$ para datos de validación. esto vuelve i^* , el número óptimo de pasos. Colocar θ nuevamente a valores aleatorios.

Entrenar en $X_{(tren)}$ y $y_{(tren)}$ para i^* pasos.

Otra estrategia para usar todos los datos es mantener los parámetros obtenidos de la primera ronda de entrenamiento y luego *continuar* entrenando pero ahora usando todos los datos. En esta etapa, ya no tenemos una guía sobre cuándo detenerse en términos de una serie de pasos. En cambio, podemos monitorear la función de pérdida promedio en el conjunto de validación y continuar entrenando hasta que caiga por debajo del valor del objetivo del conjunto de entrenamiento en el que se detuvo el procedimiento de parada anticipada. Esta estrategia evita el alto costo de volver a entrenar el modelo desde cero, pero no se comporta tan bien. Por ejemplo, no hay ninguna garantía de que el objetivo en el conjunto de validación alguna vez alcance el valor objetivo, por lo que ni siquiera se garantiza que esta estrategia termine. Este procedimiento se presenta más formalmente en el algoritmo 7.3.

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

Cómo la parada temprana actúa como un regularizador: Hasta ahora hemos dicho que la parada temprana es una estrategia de regularización, pero hemos respaldado esta afirmación solo mostrando curvas de aprendizaje donde el error del conjunto de validación tiene una curva en forma de U. Qué

Algoritmo 7.3 Meta-algoritmo que utiliza la parada temprana para determinar en qué valor objetivo comenzamos a sobreajustar, luego continuamos entrenando hasta alcanzar ese valor.

Dejar $X_{(tren)}$ y $y_{(tren)}$ ser el conjunto de entrenamiento.

Dividir $X_{(tren)}$ y $y_{(tren)}$ en $(X_{(\text{subtren})}, X_{(\text{válido})})$ y $(y_{(\text{subtren})}, y_{(\text{válido})})$

respectivamente.

Ejecutar parada anticipada (algoritmo 7.1) a partir de al azar θ usando $X_{(\text{subtren})}$ y $y_{(\text{subtren})}$ para datos de entrenamiento y $X_{(\text{válido})}$ y $y_{(\text{válido})}$ para datos de validación. esto actualiza θ

$\cdot \leftarrow J(\theta, X_{(\text{subtren})}, y_{(\text{subtren})})$

mientras $J(\theta, X_{(\text{válido})}, y_{(\text{válido})}) > -\text{hacer}$

Entrenar en $X_{(\text{tren})}$ y $y_{(\text{tren})}$ para n pasos. **terminar**

mientras

Cuál es el mecanismo real por el cual la interrupción temprana regulariza el modelo? [obispo \(1995a\)](#) y [Sjöberg y Ljung \(1995\)](#) argumentó que la detención anticipada tiene el efecto de restringir el procedimiento de optimización a un volumen relativamente pequeño de espacio de parámetros en la vecindad del valor del parámetro inicial θ_0 , como se ilustra en la figura 7.4. Más específicamente, imagine tomar τ pasos de optimización (correspondientes a iteraciones de entrenamiento) y con tasa de aprendizaje $-\tau$. Podemos ver el producto $-\tau$ como una medida de la capacidad efectiva. Suponiendo que el gradiente está acotado, restringir tanto el número de iteraciones como la tasa de aprendizaje limita el volumen del espacio de parámetros accesible desde θ_0 . En este sentido, $-\tau$ se comporta como si fuera el recíproco del coeficiente utilizado para la caída del peso.

De hecho, podemos mostrar cómo, en el caso de un modelo lineal simple con una función de error cuadrático y descenso de gradiente simple, la parada anticipada es equivalente a L_2 regularización

Para comparar con la clásica L_2 regularización, examinamos una configuración simple donde los únicos parámetros son pesos lineales ($\theta = w$). Podemos modelar la función de costo $J(w)$ con una aproximación cuadrática en la vecindad del valor empíricamente óptimo de los pesos w^* :

$$\hat{J}(\theta) = J(w^*) + (w - w^*)^T H(w - w^*), \quad (7.33)$$

dónde H es la matriz hessiana de $J(w)$ con respecto a w evaluado en w^* . Dada la suposición de que w^* es un mínimo de $J(w)$, lo sabemos H es semidefinido positivo. Bajo una aproximación de la serie local de Taylor, el gradiente viene dado por:

$$\nabla_w \hat{J}(w) = H(w - w^*). \quad (7.34)$$

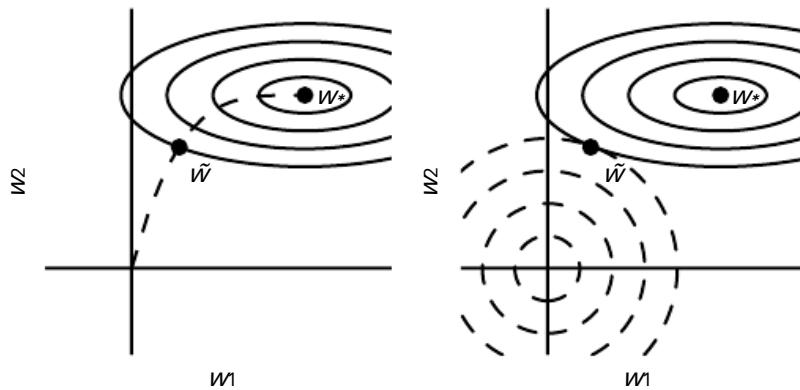


Figura 7.4: Una ilustración del efecto de detenerse antes de tiempo.(Izquierda) Las líneas de contorno sólidas indican los contornos de la verosimilitud logarítmica negativa. La línea discontinua indica la trayectoria tomada por SGD a partir del origen. En lugar de detenerse en el punto w^* que minimiza el costo, la parada temprana da como resultado que la trayectoria se detenga en un punto anterior \tilde{w} . (Derecha) Una ilustración del efecto de L_2 regularización para la comparación. Los círculos discontinuos indican los contornos de la L_2 penalización, que hace que el mínimo del costo total se encuentre más cerca del origen que el mínimo del costo no regularizado.

Vamos a estudiar la trayectoria seguida por el vector de parámetros durante el entrenamiento. Para simplificar, establezcamos el vector de parámetros inicial en el origen,³ eso es $w(0)=0$. Estudiemos el comportamiento aproximado del descenso de gradiente en \hat{f} analizando el descenso de gradiente en \hat{f} :

$$w(\tau) = w(\tau-1) - \nabla_{w(\tau)} \hat{f}(w(\tau-1)) \quad (7.35)$$

$$= w(\tau-1) - H(w(\tau-1) - w^*) \quad (7.36)$$

$$w(\tau) - w^* = (y_0 - -H)(w(\tau-1) - w^*). \quad (7.37)$$

Reescribamos ahora esta expresión en el espacio de los vectores propios de H , aprovechando la descomposición propia de H : $H = q\Lambda q^T$, donde Λ es una matriz diagonal y q es una base ortonormal de vectores propios.

$$w(\tau) - w^* = (y_0 - -q\Lambda q^T)(w(\tau-1) - w^*) q^T (w(\tau) - w^*) = \quad (7.38)$$

$$(y_0 - -\Lambda)q^T (w(\tau-1) - w^*) \quad (7.39)$$

³Para las redes neuronales, para obtener la ruptura de simetría entre unidades ocultas, no podemos inicializar todos los parámetros para 0, como se discutió en la sección 6.2. Sin embargo, el argumento es válido para cualquier otro valor inicial $w(0)$.

Asumiendo que $w(0)=y$ y eso es elegido para ser lo suficientemente pequeño para garantizar $\|y - w\|_2 < 1$, la trayectoria del parámetro durante el entrenamiento después de actualizaciones de parámetros es la siguiente:

$$w(\tau) = [y - (\gamma - \lambda)\tau]w^*. \quad (7.40)$$

Ahora, la expresión para $w(\tau)$ en ecuación 7.13 para L_2 -regularización se puede reorganizar como:

$$w(\tau) = (\gamma + \alpha I)^{-1} \gamma w^* \quad (7.41)$$

$$w(\tau) = [y - (\gamma + \alpha I)^{-1} \alpha]w^* \quad (7.42)$$

Ecuación de comparación 7.40 y ecuación 7.42, vemos que si los hiperparámetros, α , y γ se eligen de tal manera que

$$(\gamma - \lambda)\tau = (\gamma + \alpha I)^{-1} \alpha, \quad (7.43)$$

entonces la regularización y la detención anticipada pueden verse como equivalentes (al menos bajo la aproximación cuadrática de la función objetivo). Yendo aún más lejos, tomando logaritmos y usando la expansión en serie para registro $(1 + X)$, podemos concluir que si todos λ_i son pequeños (es decir, $-\lambda_i < 1$ y $\lambda_i/\alpha < 1$) entonces

$$\tau \approx \frac{1}{-\alpha}, \quad (7.44)$$

$$\alpha \approx \frac{1}{\tau}. \quad (7.45)$$

Es decir, bajo estos supuestos, el número de iteraciones de entrenamiento juega un papel inversamente proporcional a la L_2 -parámetro de regularización, y el inverso de τ juega el papel del coeficiente de decaimiento del peso.

Los valores de los parámetros correspondientes a direcciones de curvatura significativa (de la función objetivo) se regularizan menos que las direcciones de menor curvatura. Por supuesto, en el contexto de la detención temprana, esto realmente significa que los parámetros que corresponden a direcciones de curvatura significativa tienden a aprender temprano en relación con los parámetros correspondientes a direcciones de menor curvatura.

Las derivaciones en esta sección han demostrado que una trayectoria de longitud τ termina en un punto que corresponde a un mínimo de la L_2 -objetivo regularizado. Por supuesto, la detención anticipada es más que la mera restricción de la longitud de la trayectoria; en cambio, la detención temprana generalmente implica monitorear el error del conjunto de validación para detener la trayectoria en un punto particularmente bueno en el espacio. Por lo tanto, la detención temprana tiene la ventaja sobre la disminución del peso de que la detención temprana determina automáticamente la cantidad correcta de regularización, mientras que la disminución del peso requiere muchos experimentos de entrenamiento con diferentes valores de su hiperparámetro.

7.9 Vinculación de parámetros y uso compartido de parámetros

Hasta ahora, en este capítulo, cuando hemos discutido la adición de restricciones o penalizaciones a los parámetros, siempre lo hemos hecho con respecto a una región o punto fijo. Por ejemplo, la regularización (o disminución del peso) penaliza los parámetros del modelo por desviarse del valor fijo de cero. Sin embargo, a veces podemos necesitar otras formas de expresar nuestro conocimiento previo sobre los valores adecuados de los parámetros del modelo. A veces, es posible que no sepamos con precisión qué valores deben tomar los parámetros, pero sabemos, por el conocimiento del dominio y la arquitectura del modelo, que debe haber algunas dependencias entre los parámetros del modelo.

Un tipo común de dependencia que a menudo queremos expresar es que ciertos parámetros deben estar cerca unos de otros. Considere el siguiente escenario: tenemos dos modelos que realizan la misma tarea de clasificación (con el mismo conjunto de clases) pero con distribuciones de entrada algo diferentes. Formalmente, tenemos modelo A con parámetros w_A y modelo B con parámetros w_B , los dos modelos asigne la entrada a dos salidas diferentes pero relacionadas: $\hat{y}_A = f(w_A, X)$ y $\hat{y}_B = g(w_B, X)$.

Imaginemos que las tareas son lo suficientemente similares (quizás con distribuciones de entrada y salida similares) que creemos que los parámetros del modelo deberían estar cerca el uno al otro: $\forall i, w_A^i$ debería estar cerca de w_B^i . Podemos aprovechar esta información a través de la regularización. Específicamente, podemos usar una norma de parámetro de penalización de la forma: $\Omega(w_A, w_B) = -w_A - w_B - 2$. Aquí usamos un L_2 penalti, pero otros las opciones también son posibles.

Este tipo de enfoque fue propuesto por [Lasserre et al. \(2006\)](#), quien regularizó los parámetros de un modelo, entrenado como clasificador en un paradigma supervisado, para estar cerca de los parámetros de otro modelo, entrenado en un paradigma no supervisado (para capturar la distribución de los datos de entrada observados). Las arquitecturas se construyeron de manera que muchos de los parámetros del modelo clasificador pudieran emparejarse con los parámetros correspondientes del modelo no supervisado.

Si bien una penalización de norma de parámetro es una forma de regularizar los parámetros para que estén cerca unos de otros, la forma más popular es usar restricciones: *forzar conjuntos de parámetros para que sean iguales*. Este método de regularización a menudo se denomina **intercambio de parámetros**, porque interpretamos que los diversos modelos o componentes del modelo comparten un conjunto único de parámetros. Una ventaja significativa de compartir parámetros sobre la regularización de los parámetros para que estén cerca (a través de una penalización de norma) es que solo un subconjunto de los parámetros (el conjunto único) debe almacenarse en la memoria. En ciertos modelos, como la red neuronal convolucional, esto puede conducir a una reducción significativa en la huella de memoria del modelo.

Redes neuronales convolucionales Con mucho, el uso más popular y extenso de la compartición de parámetros ocurre en **redes neuronales convolucionales** (CNNs) aplicadas a la visión artificial.

Las imágenes naturales tienen muchas propiedades estadísticas que son invariantes a la traducción. Por ejemplo, una foto de un gato sigue siendo una foto de un gato si se traslada un píxel a la derecha. Las CNN tienen en cuenta esta propiedad al compartir parámetros en múltiples ubicaciones de imágenes. La misma característica (una unidad oculta con los mismos pesos) se calcula en diferentes ubicaciones en la entrada. Esto significa que podemos encontrar un gato con el mismo detector de gatos si el gato aparece en la columna/o columna $\neq 1$ en la imagen

El intercambio de parámetros ha permitido que las CNN reduzcan drásticamente la cantidad de parámetros de modelo únicos y aumenten significativamente los tamaños de red sin requerir un aumento correspondiente en los datos de entrenamiento. Sigue siendo uno de los mejores ejemplos de cómo incorporar efectivamente el conocimiento del dominio en la arquitectura de la red.

Las CNN se analizarán con más detalle en el capítulo 9.

7.10 Representaciones dispersas

El decaimiento del peso actúa colocando una penalización directamente en los parámetros del modelo. Otra estrategia es aplicar una penalización a las activaciones de las unidades en una red neuronal, fomentando que sus activaciones sean escasas. Esto impone indirectamente una penalización complicada en los parámetros del modelo.

Ya hemos discutido (en la sección 7.1.2) cómo la penalización induce una parametrización escasa, lo que significa que muchos de los parámetros se vuelven cero (o casi cero). La escasez representacional, por otro lado, describe una representación en la que muchos de los elementos de la representación son cero (o casi cero). Una vista simplificada de esta distinción se puede ilustrar en el contexto de la regresión lineal:

$$\begin{array}{ccccccccc}
 & & & & & & & & \\
 & 18 & & 4 & 0 & 0 & -20 & 0 & 2 \\
 & 5 & & 0 & 0 & -103 & 0 & -3 & - \\
 & 15 & & 0 & 5 & 0 & 0 & 0 & -2 \\
 & -9 & & -1 & 0 & 0 & -10 & -4 & -5 \\
 & -3 & & 1 & 0 & 0 & 0 & -50 & -1 \\
 & & & & & & & & 4
 \end{array} \tag{7.46}$$

$y \in \mathbb{R}_{\text{metro}}$ $A \in \mathbb{R}_{\text{metro} \times \text{norte}}$ $X \in \mathbb{R}_{\text{norte}}$

$$\begin{array}{ccccccccc}
 & -14 & & 3 & -1 & 2 & -5 & 4 & \\
 & 1 & & -4 & 2 & -3 & -1 & 1 & 3 \\
 & 19 & = & -1 & 5 & 4 & 2 & -3 & -2 \\
 & -2 & & -3 & 1 & 2 & -3 & 0 & -3 \\
 & 23 & & & -5 & 4 & -2 & 2 & -5 & -1 \\
 & & & & & & & & 0 \\
 & y \in \mathbb{R}_{\text{metro}} & & B \in \mathbb{R}_{\text{metro} \times \text{norte}} & & h \in \mathbb{R}_{\text{norte}} & & &
 \end{array} \tag{7.47}$$

En la primera expresión, tenemos un ejemplo de un modelo de regresión lineal escasamente parametrizado. En el segundo, tenemos regresión lineal con una representación escasa h de los datos X . Eso es, h es una función de X que, en cierto sentido, representa la información presente en X , pero lo hace con un vector disperso.

La regularización representacional se logra mediante los mismos tipos de mecanismos que hemos usado en la regularización de parámetros.

Norma sanción regularización de representaciones se realiza sumando a la función de pérdida una pena normal en el *representación*. Esta pena se denota $\Omega(h)$. Como antes, denotamos la función de pérdida regularizada por J :

$$J(\theta, X, y) = J(\theta, X, y) + \alpha \Omega(h) \tag{7.48}$$

dónde $\alpha \in [0, \infty)$ pondera la contribución relativa del término de penalización norma, con valores mayores de α correspondiente a una mayor regularización.

como un L_1 penalización en los parámetros induce a la escasez de parámetros, un L_1 penalización en el ℓ_1 elementos de la representación inducen a la escasez representacional: $\Omega(h) = \|h\|_1 = \sum_i |h_i|$. Por supuesto, el L_1 la pena es sólo una opción de pena que puede resultar en una representación escasa. Otras incluyen la sanción derivada de un Student- t previo a la representación (Olshausen y campo, 1996; Bergstra, 2011) y penalizaciones por divergencia KL (Larochelle y Bengio, 2008) que son especialmente útiles para representaciones con elementos restringidos a estar en el intervalo unitario. Sotavento et al. (2008) y Buen compañero et al. (2009) ambos proporcionan ejemplos de estrategias basado en la regularización de la activación promedio a través de varios ejemplos, $\frac{1}{n} \sum_i h(i)$, a estar cerca de algún valor objetivo, como un vector con .01 para cada entrada.

Otros enfoques obtienen escasez de representación con una fuerte restricción en los valores de activación. Por ejemplo, **persecución de emparejamiento ortogonal** (patiet al., 1993) codifica una entrada X con la representación h que resuelve el problema de optimización con restricciones

$$\begin{aligned}
 & \text{mínimo de argumento-}x -Qu-2, \\
 & S.S-0 < k
 \end{aligned} \tag{7.49}$$

dónde $-h$ es el número de entradas distintas de cero de h . Este problema puede resolverse eficientemente cuando W está restringida a ser ortogonal. Este método a menudo se llama

OMP- k con el valor de k especificado para indicar el número de características distintas de cero permitidas. Coates y Ng (2011) demostró que OMP-1 puede ser un extractor de características muy efectivo para arquitecturas profundas.

Esencialmente, cualquier modelo que tenga unidades ocultas puede hacerse escaso. A lo largo de este libro, veremos muchos ejemplos de regularización de escasez utilizados en una variedad de contextos.

7.11 Embolsado y otros métodos de conjunto

Harpillera (corto para **agregación de arranque**) es una técnica para reducir el error de generalización mediante la combinación de varios modelos (Breiman, 1994). La idea es entrenar varios modelos diferentes por separado, luego hacer que todos los modelos voten sobre la salida para ejemplos de prueba. Este es un ejemplo de una estrategia general en aprendizaje automático llamada **promedio del modelo**. Las técnicas que emplean esta estrategia se conocen como **métodos de conjunto**.

La razón por la que funciona el promedio del modelo es que los diferentes modelos no suelen cometer los mismos errores en el conjunto de prueba.

Consideremos por ejemplo un conjunto de k modelos de regresión. Supongamos que cada modelo comete un error v_i en cada ejemplo, con los errores extraídos de una media cero distribución normal multivariada con varianzas $\text{MI}[v_i] = v$ y covarianzas $\text{MI}[v_i v_j] = C$. Entonces el error cometido por la predicción promedio de todos los modelos de conjunto es $\frac{1}{k} \sum_i v_i$. El error cuadrático esperado del predictor de conjunto es

$$\text{mi} \left(\frac{1}{k} \sum_i v_i \right)^2 = \frac{1}{k^2} \text{mi} \left(\sum_i v_i^2 + \sum_{i \neq j} v_i v_j \right) \quad (7.50)$$

$$= \frac{1}{k} v^2 + \frac{k-1}{k} C. \quad (7.51)$$

En el caso de que los errores estén perfectamente correlacionados y $C=v$, el error cuadrático medio se reduce a v , por lo que el promedio del modelo no ayuda en absoluto. En el caso de que los errores no estén perfectamente correlacionados y $C=0$, el error cuadrático esperado de la el conjunto es solo $1/kv$. Esto significa que el error cuadrático esperado del conjunto disminuye linealmente con el tamaño del conjunto. En otras palabras, en promedio, el conjunto se desempeñará al menos tan bien como cualquiera de sus miembros, y si los miembros cometen errores independientes, el conjunto se desempeñará significativamente mejor que sus miembros.

Diferentes métodos de conjunto construyen el conjunto de modelos de diferentes maneras. Por ejemplo, cada miembro del conjunto podría formarse entrenando a un

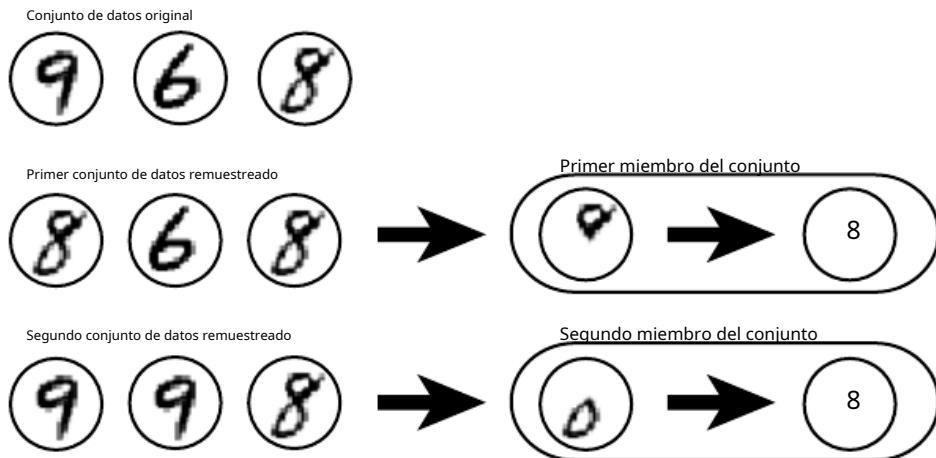


Figura 7.5: Una representación de dibujos animados de cómo funciona el embolsado. Supongamos que entrenamos un detector 8 en el conjunto de datos que se muestra arriba, que contiene un 8, un 6 y un 9. Supongamos que creamos dos conjuntos de datos remuestreados diferentes. El procedimiento de entrenamiento de embolsado consiste en construir cada uno de estos conjuntos de datos mediante muestreo con reemplazo. El primer conjunto de datos omite el 9 y repite el 8. En este conjunto de datos, el detector aprende que un bucle en la parte superior del dígito corresponde a un 8. En el segundo conjunto de datos, repetimos el 9 y omitimos el 6. En este caso, el detector aprende que un bucle en la parte inferior del dígito corresponde a un 8. Cada una de estas reglas de clasificación individuales es frágil, pero si promediamos su salida, entonces el detector es robusto y logra la máxima confianza solo cuando ambos bucles del 8 están presentes.

diferente tipo de modelo usando un algoritmo diferente o función objetivo. El embolsado es un método que permite reutilizar varias veces el mismo tipo de modelo, algoritmo de entrenamiento y función objetivo.

Específicamente, embolsar implica construir k diferentes conjuntos de datos. Cada conjunto de datos tiene la misma cantidad de ejemplos que el conjunto de datos original, pero cada conjunto de datos se construye mediante muestreo con reemplazo del conjunto de datos original. Esto significa que, con alta probabilidad, a cada conjunto de datos le faltan algunos de los ejemplos del conjunto de datos original y también contiene varios ejemplos duplicados (en promedio, alrededor de 2/3 de los ejemplos del conjunto de datos original se encuentran en el conjunto de entrenamiento resultante, si tiene el mismo tamaño que el original). El modelo luego se entrena en el conjunto de datos i . Las diferencias entre los ejemplos que se incluyen en cada conjunto de datos dan como resultado diferencias entre los modelos entrenados. Ver figura 7.5 para un ejemplo.

Las redes neuronales alcanzan una variedad lo suficientemente amplia de puntos de solución que, a menudo, pueden beneficiarse del promedio del modelo, incluso si todos los modelos están entrenados en el mismo conjunto de datos. Las diferencias en la inicialización aleatoria, la selección aleatoria de minibatches, las diferencias en los hiperparámetros o los diferentes resultados de las implementaciones no deterministas de las redes neuronales suelen ser suficientes para hacer que diferentes miembros de la

conjunto para cometer errores parcialmente independientes.

El promedio del modelo es un método extremadamente poderoso y confiable para reducir el error de generalización. Por lo general, se desaconseja su uso cuando se comparan algoritmos para artículos científicos, porque cualquier algoritmo de aprendizaje automático puede beneficiarse sustancialmente del promedio del modelo al precio de una mayor computación y memoria. Por esta razón, las comparaciones de referencia generalmente se realizan utilizando un solo modelo.

Los concursos de aprendizaje automático generalmente se ganan mediante métodos que utilizan un promedio de modelos sobre docenas de modelos. Un ejemplo destacado reciente es el Gran Premio de Netflix ([Koren,2009](#)).

No todas las técnicas para construir conjuntos están diseñadas para que el conjunto sea más regularizado que los modelos individuales. Por ejemplo, una técnica llamada **impulsar**([Freund y Schapire,1996b,a](#)) construye un conjunto con mayor capacidad que los modelos individuales. El impulso se ha aplicado para construir conjuntos de redes neuronales ([Schwenk y Bengio,1998](#)) mediante la adición incremental de redes neuronales al conjunto. El impulso también se ha aplicado interpretando una red neuronal individual como un conjunto ([bengio et al.,2006a](#)), agregando gradualmente unidades ocultas a la red neuronal.

7.12 Abandono

Abandonar([Srivastava et al.,2014](#)) proporciona un método computacionalmente económico pero poderoso para regularizar una amplia familia de modelos. En una primera aproximación, se puede pensar en el abandono como un método para hacer que el empaquetamiento sea práctico para conjuntos de muchas redes neuronales grandes. El embolsado implica entrenar varios modelos y evaluar varios modelos en cada ejemplo de prueba. Esto parece poco práctico cuando cada modelo es una gran red neuronal, ya que entrenar y evaluar dichas redes es costoso en términos de tiempo de ejecución y memoria. Es común usar conjuntos de cinco a diez redes neuronales:[Szegedy et al.\(2014a\)](#) usó seis para ganar el ILSVRC, pero más que esto rápidamente se vuelve difícil de manejar. Dropout proporciona una aproximación económica para entrenar y evaluar un conjunto empaquetado de muchas redes neuronales exponencialmente.

Específicamente, dropout entrena al conjunto que consta de todas las subredes que se pueden formar eliminando unidades que no son de salida de una red base subyacente, como se ilustra en la figura[7.6](#). En la mayoría de las redes neuronales modernas, basadas en una serie de transformaciones afines y no linealidades, podemos eliminar de manera efectiva una unidad de una red multiplicando su valor de salida por cero. Este procedimiento requiere algunas ligeras modificaciones para modelos tales como redes de función de base radial, que toman

la diferencia entre el estado de la unidad y algún valor de referencia. Aquí, presentamos el algoritmo de abandono en términos de multiplicación por cero para simplificar, pero puede modificarse trivialmente para que funcione con otras operaciones que eliminan una unidad de la red.

Recuerde que para aprender con bagging, definimos k diferentes modelos, construir k diferentes conjuntos de datos tomando muestras del conjunto de entrenamiento con reemplazo, y luego entrene el modelo i en conjunto de datos i . Dropout pretende aproximarse a este proceso, pero con un número exponencialmente grande de redes neuronales. Específicamente, para entrenar con abandono, usamos un algoritmo de aprendizaje basado en minilotes que realiza pequeños pasos, como el descenso de gradiente estocástico. Cada vez que cargamos un ejemplo en un minilote, probamos aleatoriamente una máscara binaria diferente para aplicarla a todas las unidades de entrada y ocultas en la red. La máscara de cada unidad se muestrea independientemente de todas las demás. La probabilidad de muestrear un valor de máscara de uno (lo que hace que se incluya una unidad) es un hiperparámetro fijado antes de que comience el entrenamiento. No es una función del valor actual de los parámetros del modelo o del ejemplo de entrada. Por lo general, se incluye una unidad de entrada con una probabilidad de 0,8 y una unidad oculta con una probabilidad de 0,5. Luego ejecutamos la propagación hacia adelante, la propagación hacia atrás, y la actualización de aprendizaje como de costumbre. Cifra 7.7 ilustra cómo ejecutar la propagación directa con abandono.

Más formalmente, supongamos que un vector de máscara μ especifica qué unidades incluir, y $j(\theta, \mu)$ define el costo del modelo definido por parámetros θ y máscara μ . Entonces el entrenamiento de deserción consiste en minimizar $m_j(\theta, \mu)$. La expectativa contiene exponencialmente muchos términos, pero podemos obtener una estimación imparcial de su gradiente muestreando valores de μ .

El entrenamiento de abandono no es lo mismo que el entrenamiento de embolsado. En el caso del embolsado, los modelos son todos independientes. En el caso de abandono, los modelos comparten parámetros y cada modelo hereda un subconjunto diferente de parámetros de la red neuronal principal. Este intercambio de parámetros hace posible representar un número exponencial de modelos con una cantidad manejable de memoria. En el caso del embolsado, cada modelo se entrena hasta la convergencia en su respectivo conjunto de entrenamiento. En el caso de abandono, por lo general, la mayoría de los modelos no se entrenan explícitamente en absoluto; por lo general, el modelo es lo suficientemente grande como para que no sea factible muestrear todas las subredes posibles dentro de la vida útil del universo. En su lugar, una pequeña fracción de las posibles subredes están entrenadas para un solo paso, y la compartición de parámetros hace que las subredes restantes lleguen a buenos ajustes de los parámetros. Estas son las únicas diferencias. Más allá de estos, la deserción sigue el algoritmo de embolsado. Por ejemplo, el conjunto de entrenamiento encontrado por cada subred es de hecho un subconjunto del conjunto de entrenamiento original muestreado con reemplazo.

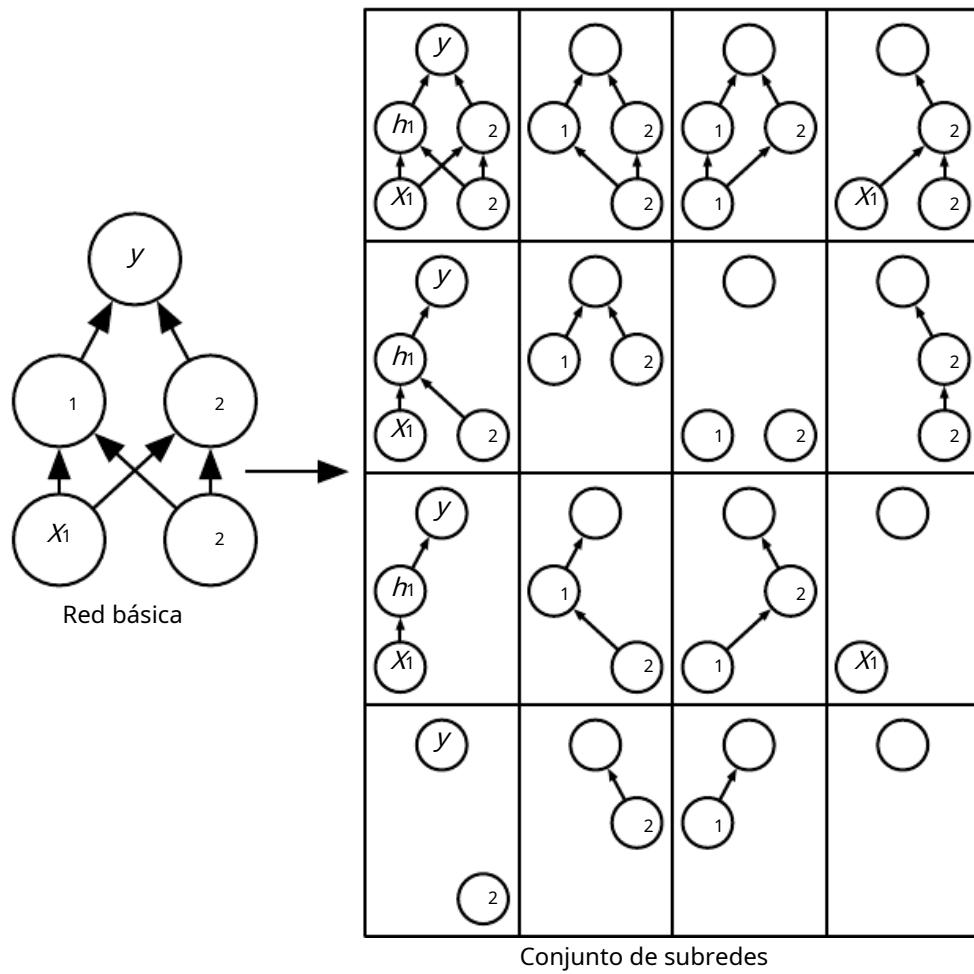


Figura 7.6: Dropout entrena un conjunto que consta de todas las subredes que se pueden construir eliminando unidades que no son de salida de una red base subyacente. Aquí, comenzamos con una red base con dos unidades visibles y dos unidades ocultas. Hay dieciséis subconjuntos posibles de estas cuatro unidades. Mostramos todas las dieciséis subredes que pueden formarse eliminando diferentes subconjuntos de unidades de la red original. En este pequeño ejemplo, una gran proporción de las redes resultantes no tienen unidades de entrada o ninguna ruta que conecte la entrada con la salida. Este problema se vuelve insignificante para las redes con capas más anchas, donde la probabilidad de descartar todos los caminos posibles de las entradas a las salidas se vuelve más pequeña.

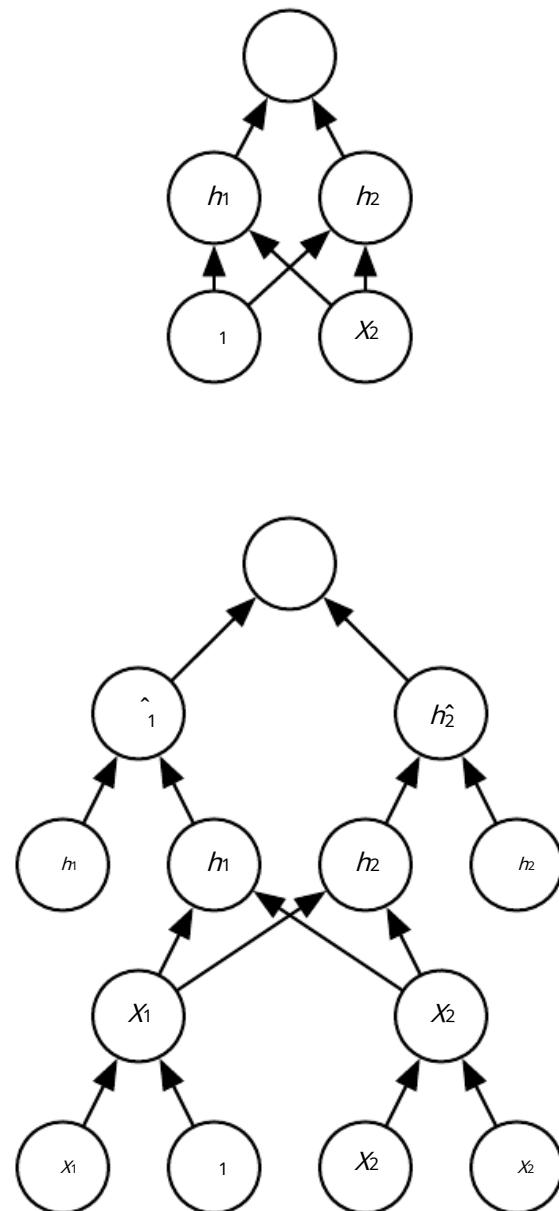


Figura 7.7: Un ejemplo de propagación hacia adelante a través de una red de alimentación hacia adelante utilizando abandono.(Arriba)En este ejemplo, usamos una red feedforward con dos unidades de entrada, una capa oculta con dos unidades ocultas y una unidad de salida.(Abajo)Para realizar la propagación directa con abandono, muestreamos aleatoriamente un vector μ con una entrada para cada entrada o unidad oculta en la red. las entradas de μ son binarios y se muestran independientemente unos de otros. La probabilidad de que cada entrada sea 1 es un hiperparámetro, generalmente 0.5 para las capas ocultas y 0.8 para la entrada. Cada unidad en la red se multiplica por la máscara correspondiente y luego la propagación directa continúa a través del resto de la red como de costumbre. Esto es equivalente a seleccionar aleatoriamente una de las subredes de la figura 7.6 y ejecutar la propagación hacia adelante a través de él.

Para hacer una predicción, un conjunto en bolsa debe acumular votos de todos sus miembros. Nos referimos a este proceso como **inferencia** en este contexto. Hasta ahora, nuestra descripción de embolsado y abandono no ha requerido que el modelo sea explícitamente probabilístico. Ahora, asumimos que el rol del modelo es generar una distribución de probabilidad. En el caso de embolsado, cada modelo produce una distribución de probabilidad $\text{pag}(y/X)$. La predicción del conjunto viene dada por la media aritmética de todas estas distribuciones,

$$\frac{1}{k} \sum_{j=1}^k \text{pag}(y/X_j). \quad (7.52)$$

En el caso de abandono, cada submodelo definido por vector de máscara μ define una distribución de probabilidad $\text{pag}(y/x, \mu)$. La media aritmética de todas las máscaras está dada por

$$\frac{\sum_{\mu} \text{pag}(\mu) \text{pag}(y/x, \mu)}{\mu} \quad (7.53)$$

dónde $\text{pag}(\mu)$ es la distribución de probabilidad que se utilizó para muestrear μ en tiempo de entrenamiento.

Debido a que esta suma incluye un número exponencial de términos, es difícil de evaluar, excepto en los casos en que la estructura del modelo permita alguna forma de simplificación. Hasta el momento, no se sabe que las redes neuronales profundas permitan ninguna simplificación tratable. En su lugar, podemos aproximar la inferencia con el muestreo, promediando la salida de muchas máscaras. Incluso 10-20 máscaras suelen ser suficientes para obtener un buen rendimiento.

Sin embargo, existe un enfoque aún mejor, que nos permite obtener una buena aproximación a las predicciones de todo el conjunto, a costa de una sola propagación directa. Para hacerlo, cambiamos a usar la media geométrica en lugar de la media aritmética de las distribuciones predichas de los miembros del conjunto. [Warde-Farley et al. \(2014\)](#) presentan argumentos y evidencia empírica de que la media geométrica se comporta de manera comparable a la media aritmética en este contexto.

No se garantiza que la media geométrica de distribuciones de probabilidad múltiples sea una distribución de probabilidad. Para garantizar que el resultado sea una distribución de probabilidad, imponemos el requisito de que ninguno de los submodelos asigne probabilidad 0 a ningún evento, y renormalizamos la distribución resultante. La probabilidad no normalizada la distribución definida directamente por la media geométrica está dada por

$$\text{pag}_{\text{conjunto}}(y/X) = \prod_{\mu} \text{pag}(y/x, \mu) \quad (7.54)$$

dónde d es el número de unidades que se pueden soltar. Aquí usamos una distribución uniforme sobre μ para simplificar la presentación, pero las distribuciones no uniformes son

también es posible. Para hacer predicciones debemos volver a normalizar el conjunto:

$$pag_{\text{conjunto}}(y/X) = \frac{\underset{-\text{conjunto}}{PAG}(y/X)}{\underset{y}{\sum} pag_{\text{conjunto}}(y/X)}. \quad (7.55)$$

Una idea clave (Hinton et al., 2012c) involucrados en la deserción es que podemos aproximar pag_{conjunto} evaluando $pag(y/X)$ en un modelo: el modelo con todas las unidades, pero con los pesos saliendo de la unidad i multiplicado por la probabilidad de incluir la unidad i . La motivación de esta modificación es capturar el valor esperado correcto de la salida de esa unidad. Llamamos a este enfoque **el regla de inferencia de escala de peso**. Todavía no existe ningún argumento teórico para la precisión de esta regla de inferencia aproximada en redes no lineales profundas, pero empíricamente funciona muy bien.

Debido a que usualmente usamos una probabilidad de inclusión de $\frac{1}{2}$, la regla de escala de peso por lo general equivale a dividir los pesos por 2 al final del entrenamiento, y luego usando el modelo como de costumbre. Otra forma de lograr el mismo resultado es multiplicar los estados de las unidades por 2 durante el entrenamiento. De cualquier manera, el objetivo es asegurarse de que la entrada total esperada para una unidad en el momento de la prueba sea aproximadamente la misma que la entrada total esperada para esa unidad en el momento del tren, aunque en promedio falte la mitad de las unidades en el momento del tren.

Para muchas clases de modelos que no tienen unidades ocultas no lineales, la regla de inferencia de escala de peso es exacta. Para un ejemplo simple, considere un clasificador de regresión softmax con n variables de entrada representadas por el vector v :

$$PAG(y = y/v) = \underset{y}{\text{softmax}} Wv + b. \quad (7.56)$$

Podemos indexar en la familia de submodelos mediante la multiplicación por elementos de la entrada con un vector binario d :

$$PAG(y = y/v; d) = \underset{y}{\text{softmax}} W(d \cdot v) + b. \quad (7.57)$$

El predictor del conjunto se define volviendo a normalizar la media geométrica de las predicciones de todos los miembros del conjunto:

$$PAG_{\text{conjunto}}(y = y/v) = \frac{\underset{y}{\text{PAG}}(y = y/v) PAG_{\text{conjunto}}(y \cdot \bar{v})}{\underset{y}{\sum} PAG_{\text{conjunto}}(y \cdot \bar{v})} \quad (7.58)$$

dónde

$$PAG_{\text{conjunto}}(y = y/v) =_{2 \text{norte}} \frac{\underset{d \in \{0,1\}^n}{\text{PAG}}(y = y/v; d)}{\underset{d \in \{0,1\}^n}{\text{PAG}}(y = y/v; d)}. \quad (7.59)$$

Para ver que la regla de escala de peso es exacta, podemos simplificar PAG_{conjunto} :

$$\frac{PAG_{\text{conjunto}}(y = y / v) =_{2 \text{norte}}}{d \in \{0, 1\}_{\text{norte}}} PAG(y = y / v; d) \quad (7.60)$$

$$=_{2 \text{norte}} \frac{\text{softmax}(W(d \cdot v) + b)_y}{d \in \{0, 1\}_{\text{norte}}} \quad (7.61)$$

$$=_{2 \text{norte}} \frac{\frac{\text{Exp } W_{y,:}(d \cdot v) + b_y}{d \in \{0, 1\}_{\text{norte}}}}{y \text{Exp } W_{y,:}(d \cdot v) + b_y} \quad (7.62)$$

$$= \frac{\frac{2 \text{norte}}{d \in \{0, 1\}_{\text{norte}}} \text{Exp } W_{y,:}(d \cdot v) + b_y}{y \text{Exp } W_{y,:}(d \cdot v) + b_y} \quad (7.63)$$

Porque PAG se normalizará, podemos ignorar con seguridad la multiplicación por factores que son constantes con respecto a y :

$$\frac{PAG_{\text{conjunto}}(y = y / v) \alpha_{2 \text{norte}}}{d \in \{0, 1\}_{\text{norte}}} \frac{\text{Exp } W_{y,:}(d \cdot v) + b_y}{\quad \quad \quad} \quad (7.64)$$

$$= \exp -1 \frac{W_{y,:}(d \cdot v) + b_y}{d \in \{0, 1\}_{\text{norte}}} \quad (7.65)$$

$$= \exp \frac{1}{2} W_{y,:}v + b_y. \quad (7.66)$$

Sustituyendo esto de nuevo en la ecuación 7.58 obtenemos un clasificador softmax con pesos $\frac{1}{2}W$.

La regla de escala de peso también es exacta en otras configuraciones, incluidas las redes de regresión con salidas condicionalmente normales y redes profundas que tienen capas ocultas sin no linealidades. Sin embargo, la regla de escala de peso es solo una aproximación para modelos profundos que no tienen linealidades. Aunque la aproximación no se ha caracterizado teóricamente, a menudo funciona bien empíricamente. [Buen compañero et al.](#) (2013a) encontró experimentalmente que la aproximación de escala de peso puede funcionar mejor (en términos de precisión de clasificación) que las aproximaciones de Monte Carlo al predictor de conjunto. Esto se mantuvo incluso cuando se permitió la aproximación de Monte Carlo para muestrear hasta 1000 subredes. [Gal y Ghahramani](#) (2015) encontró que algunos modelos obtienen una mejor precisión de clasificación usando veinte muestras y

la aproximación de Montecarlo. Parece que la elección óptima de la aproximación de inferencia depende del problema.

Srivastava *et al.*(2014) mostró que la deserción es más eficaz que otros regularizadores estándar de bajo costo computacional, como la disminución del peso, las restricciones de la norma del filtro y la regularización de la actividad escasa. La deserción también puede combinarse con otras formas de regularización para lograr una mejora adicional.

Una ventaja de la deserción es que es muy barato desde el punto de vista computacional. El uso de abandono durante el entrenamiento requiere solo $O(n)$ cálculo por ejemplo por actualización, para generar n números binarios aleatorios y multiplicarlos por el estado. Dependiendo de la implementación, también puede requerir $O(n)$ memoria para almacenar estos números binarios hasta la etapa de propagación hacia atrás. Ejecutar la inferencia en el modelo entrenado tiene el mismo costo por ejemplo que si no se usara el abandono, aunque debemos pagar el costo de dividir los pesos por 2 una vez antes de comenzar a ejecutar la inferencia en los ejemplos.

Otra ventaja significativa de la deserción es que no limita significativamente el tipo de modelo o procedimiento de entrenamiento que se puede utilizar. Funciona bien con casi cualquier modelo que utilice una representación distribuida y se puede entrenar con descenso de gradiente estocástico. Esto incluye redes neuronales feedforward, modelos probabilísticos como máquinas de Boltzmann restringidas (Srivastava *et al.*,2014), y redes neuronales recurrentes (Bayer y Osendorfer,2014;Pascanu *et al.*,2014a). Muchas otras estrategias de regularización de potencia comparable imponen restricciones más severas a la arquitectura del modelo.

Aunque el costo por paso de aplicar la deserción a un modelo específico es insignificante, el costo de usar la deserción en un sistema completo puede ser significativo. Debido a que la deserción es una técnica de regularización, reduce la capacidad efectiva de un modelo. Para compensar este efecto, debemos aumentar el tamaño del modelo. Por lo general, el error de conjunto de validación óptimo es mucho menor cuando se usa el abandono, pero esto tiene el costo de un modelo mucho más grande y muchas más iteraciones del algoritmo de entrenamiento. Para conjuntos de datos muy grandes, la regularización confiere poca reducción en el error de generalización. En estos casos, el costo computacional de usar modelos de deserción y más grandes puede superar el beneficio de la regularización.

Cuando hay muy pocos ejemplos de capacitación etiquetados disponibles, la deserción es menos efectiva. redes neuronales bayesianas (Neal,1996) superan la deserción en el conjunto de datos de empalme alternativo (Xiong *et al.*,2011) donde hay menos de 5.000 ejemplos disponibles (Srivastava *et al.*,2014). Cuando hay datos adicionales sin etiquetar disponibles, el aprendizaje de funciones no supervisado puede obtener una ventaja sobre la deserción.

Apostar *et al.*(2013) mostró que, cuando se aplica a la regresión lineal, la deserción es equivalente a L_2 decaimiento de peso, con un coeficiente de decaimiento de peso diferente para

cada función de entrada. La magnitud del coeficiente de disminución del peso de cada característica está determinada por su varianza. Resultados similares se mantienen para otros modelos lineales. Para modelos profundos, la caída no es equivalente a la caída del peso.

La estocasticidad utilizada durante el entrenamiento con abandono no es necesaria para el éxito del enfoque. Es solo un medio para aproximar la suma de todos los submodelos. [Wang y Manning\(2013\)](#) derivaron aproximaciones analíticas a esta marginación. Su aproximación, conocida como **abandono rápido** resultó en un tiempo de convergencia más rápido debido a la estocasticidad reducida en el cálculo del gradiente. Este método también se puede aplicar en el momento de la prueba, como una aproximación más basada en principios (pero también más costosa desde el punto de vista computacional) al promedio de todas las subredes que la aproximación de escala de peso. El abandono rápido se ha utilizado para casi igualar el rendimiento del abandono estándar en problemas de redes neuronales pequeñas, pero aún no ha producido una mejora significativa ni se ha aplicado a un problema grande.

Así como la estocasticidad no es necesaria para lograr el efecto regularizador de la deserción, tampoco es suficiente. Para demostrar esto, [Warde-Farley et al.\(2014\)](#) diseñaron experimentos de control usando un método llamado **aumento de la deserción** que diseñaron para usar exactamente el mismo ruido de máscara que el abandono tradicional pero carecen de su efecto de regularización. El impulso de abandono entrena a todo el conjunto para maximizar conjuntamente la probabilidad de registro en el conjunto de entrenamiento. En el mismo sentido en que la deserción tradicional es análoga al embolsado, este enfoque es análogo al impulso. Según lo previsto, los experimentos con aumento de la deserción casi no muestran ningún efecto de regularización en comparación con el entrenamiento de toda la red como un modelo único. Esto demuestra que la interpretación de abandono como embolsado tiene valor más allá de la interpretación de abandono como robustez al ruido. El efecto de regularización del conjunto empaquetado solo se logra cuando los miembros del conjunto muestreados estocásticamente están entrenados para desempeñarse bien independientemente unos de otros.

La deserción ha inspirado otros enfoques estocásticos para entrenar conjuntos exponencialmente grandes de modelos que comparten pesos. DropConnect es un caso especial de deserción donde cada producto entre un solo peso escalar y un solo estado de unidad oculta se considera una unidad que se puede descartar ([Pálido et al., 2013](#)). La agrupación estocástica es una forma de agrupación aleatoria (consulte la sección 9.3) para construir conjuntos de redes convolucionales con cada red convolucional atendiendo a diferentes ubicaciones espaciales de cada mapa de características. Hasta ahora, el abandono sigue siendo el método de conjunto implícito más utilizado.

Una de las ideas clave del abandono es que entrenar una red con un comportamiento estocástico y hacer predicciones promediando varias decisiones estocásticas implementa una forma de embolsado con parámetros compartidos. Anteriormente, describimos

abandono como embolsado de un conjunto de modelos formado por unidades de inclusión o exclusión. Sin embargo, no es necesario que esta estrategia de promediación del modelo se base en la inclusión y la exclusión. En principio, cualquier tipo de modificación aleatoria es admisible. En la práctica, debemos elegir familias de modificación que las redes neuronales puedan aprender a resistir. Idealmente, también deberíamos usar familias de modelos que permitan una regla de inferencia aproximada rápida. Podemos pensar en cualquier forma de modificación parametrizada por un vector μ como formación de un conjunto formado por $p_{\text{avg}}(y|x, \mu)$ para todos los valores posibles de μ . No hay requisito de que μ tengan un número finito de valores. Por ejemplo, μ puede ser de valor real. Srivastava et al. (2014) mostró que multiplicando los pesos por $\mu \sim \text{norte}(1, I)$ puede superar la deserción en función de las máscaras binarias. Porque MI[μ] = 1 la red estándar implementa automáticamente la inferencia aproximada en el conjunto, sin necesidad de escalar pesos.

Hasta ahora hemos descrito la omisión puramente como un medio para realizar un embolsado aproximado eficiente. Sin embargo, hay otra visión de la deserción que va más allá. Dropout entrena no solo un conjunto de modelos en bolsas, sino un conjunto de modelos que comparten unidades ocultas. Esto significa que cada unidad oculta debe poder funcionar bien independientemente de qué otras unidades ocultas estén en el modelo. Las unidades ocultas deben estar preparadas para intercambiarse e intercambiarse entre modelos. Hinton et al. (2012c) se inspiraron en una idea de la biología: la reproducción sexual, que implica el intercambio de genes entre dos organismos diferentes, crea una presión evolutiva para que los genes no solo se vuelvan buenos, sino que se intercambien fácilmente entre diferentes organismos. Tales genes y tales características son muy resistentes a los cambios en su entorno porque no pueden adaptarse incorrectamente a las características inusuales de ningún organismo o modelo. Por lo tanto, la deserción regulariza cada unidad oculta para que no sea simplemente una buena característica, sino una característica que es buena en muchos contextos. Warde-Farley et al. (2014) compararon el entrenamiento de deserción con el entrenamiento de grandes conjuntos y concluyeron que la deserción ofrece mejoras adicionales al error de generalización más allá de las obtenidas por conjuntos de modelos independientes.

Es importante comprender que una gran parte de la potencia de abandono surge del hecho de que el ruido de enmascaramiento se aplica a las unidades ocultas. Esto puede verse como una forma de destrucción adaptativa altamente inteligente del contenido de información de la entrada en lugar de la destrucción de los valores brutos de la entrada. Por ejemplo, si el modelo aprende una unidad oculta h que detecta una cara encontrando la nariz, luego soltando h corresponde a borrar la información de que hay una nariz en la imagen. El modelo debe aprender otro h , ya sea que codifique de manera redundante la presencia de una nariz, o que detecte la cara por otra característica, como la boca. Las técnicas tradicionales de inyección de ruido que agregan ruido no estructurado en la entrada no pueden borrar aleatoriamente la información sobre una nariz de una imagen de una cara a menos que la magnitud del ruido sea tan grande que casi toda la información en

la imagen es eliminada. Destruir las características extraídas en lugar de los valores originales permite que el proceso de destrucción haga uso de todo el conocimiento sobre la distribución de entrada que el modelo ha adquirido hasta el momento.

Otro aspecto importante de la deserción es que el ruido es multiplicativo. Si el ruido fuera aditivo con escala fija, entonces una unidad oculta lineal rectificada h con ruido añadido-simplemente podría aprender a tener h volverse muy grande para hacer el ruido adicional-insignificante en comparación. El ruido multiplicativo no permite una solución tan patológica al problema de la robustez del ruido.

Otro algoritmo de aprendizaje profundo, la normalización por lotes, reparametriza el modelo de una manera que introduce ruido tanto aditivo como multiplicativo en las unidades ocultas en el momento del entrenamiento. El objetivo principal de la normalización por lotes es mejorar la optimización, pero el ruido puede tener un efecto de regularización y, a veces, hace que la deserción sea innecesaria. La normalización por lotes se describe más adelante en la sección 8.7.1.

7.13 Entrenamiento contradictorio

En muchos casos, las redes neuronales han comenzado a alcanzar el rendimiento humano cuando se evalúan en un conjunto de pruebas iid. Por lo tanto, es natural preguntarse si estos modelos han obtenido una verdadera comprensión a nivel humano de estas tareas. Para probar el nivel de comprensión que tiene una red de la tarea subyacente, podemos buscar ejemplos en los que el modelo clasifica erróneamente. [Szegedy et al. \(2014b\)](#) encontró que incluso las redes neuronales que funcionan con precisión a nivel humano tienen una tasa de error de casi el 100% en ejemplos que se construyen intencionalmente mediante el uso de un procedimiento de optimización para buscar una entrada X cerca de un punto de datos \tilde{X} tal que la salida del modelo es muy diferente en X . En muchos casos, X puede ser tan similar a \tilde{X} que un observador humano no puede notar la diferencia entre el ejemplo original y el **ejemplo adversario**, pero la red puede hacer predicciones muy diferentes. Ver figura 7.8 para un ejemplo.

Los ejemplos adversarios tienen muchas implicaciones, por ejemplo, en la seguridad informática, que están más allá del alcance de este capítulo. Sin embargo, son interesantes en el contexto de la regularización porque se puede reducir la tasa de error en el conjunto de prueba iid original a través de **entrenamiento adversario**—entrenamiento en ejemplos adversariamente perturbados del conjunto de entrenamiento ([Szegedy et al., 2014b; Buen compañero et al., 2014b](#)).

[Buen compañero et al. \(2014b\)](#) mostró que una de las principales causas de estos ejemplos contradictorios es la linealidad excesiva. Las redes neuronales se construyen a partir de bloques de construcción principalmente lineales. En algunos experimentos, la función general que implementan resulta ser altamente lineal como resultado. Estas funciones lineales son fáciles

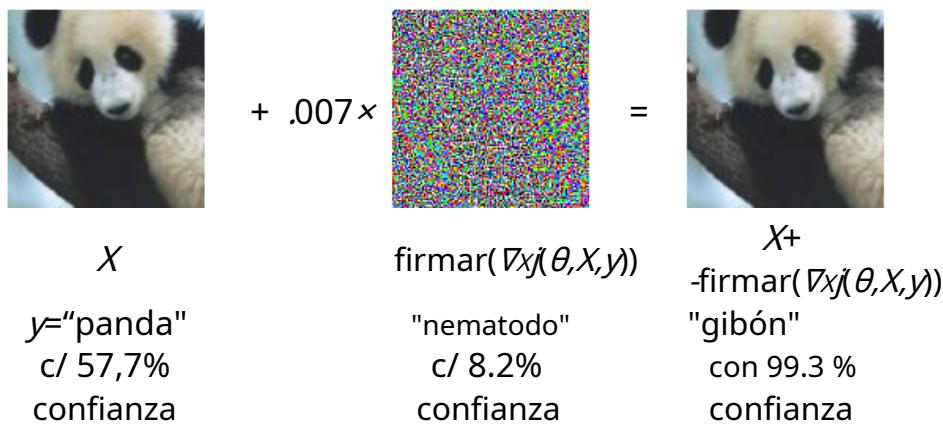


Figura 7.8: Una demostración de generación de ejemplos contradictorios aplicada a GoogLeNet ([Szegedy et al., 2014a](#)) en ImageNet. Al agregar un vector imperceptiblemente pequeño cuyos elementos son iguales al signo de los elementos del gradiente de la función de costo con respecto a la entrada, podemos cambiar la clasificación de la imagen de GoogLeNet. Reproducido con permiso de [Buen compañero et al. \(2014b\)](#).

Para optimizar. Desafortunadamente, el valor de una función lineal puede cambiar muy rápidamente si tiene numerosas entradas. Si cambiamos cada entrada por-, entonces una función lineal con pesos w puede cambiar tanto como $-||w||/1$, que puede ser una cantidad muy grande si w es de alta dimensión. El entrenamiento adversario desalienta este comportamiento lineal local altamente sensible al alentar a la red a ser localmente constante en la vecindad de los datos de entrenamiento. Esto puede verse como una forma de introducir explícitamente una constancia local previa en las redes neuronales supervisadas.

El entrenamiento adversario ayuda a ilustrar el poder de usar una gran familia de funciones en combinación con una regularización agresiva. Los modelos puramente lineales, como la regresión logística, no pueden resistir los ejemplos adversarios porque se ven obligados a ser lineales. Las redes neuronales pueden representar funciones que pueden variar desde casi lineales hasta casi localmente constantes y, por lo tanto, tienen la flexibilidad de capturar tendencias lineales en los datos de entrenamiento mientras aprenden a resistir la perturbación local.

Los ejemplos adversarios también proporcionan un medio para lograr el aprendizaje semisupervisado. en un punto x que no está asociado con una etiqueta en el conjunto de datos, el propio modelo asigna alguna etiqueta \hat{y} . La etiqueta del modelo \hat{y} puede que no sea la verdadera etiqueta, pero si el modelo es de alta calidad, entonces \hat{y} tiene una alta probabilidad de proporcionar la etiqueta verdadera. Podemos buscar un ejemplo adversario x que hace que el clasificador genere una etiqueta y con $y = \hat{y}$. Los ejemplos adversarios generados utilizando no la etiqueta verdadera sino una etiqueta proporcionada por un modelo entrenado se denominan **ejemplos adversarios virtuales** ([Miyato et al., 2015](#)). El clasificador puede entonces ser entrenado para asignar la misma etiqueta a x . Esto anima al clasificador a aprender una función que es

robusto a pequeños cambios en cualquier lugar a lo largo de la variedad donde se encuentran los datos sin etiquetar. La suposición que motiva este enfoque es que las diferentes clases generalmente se encuentran en variedades desconectadas, y una pequeña perturbación no debería poder saltar de una variedad de clase a otra variedad de clase.

7.14 Clasificador de distancia de tangente, prop de tangente y tangente múltiple

Muchos algoritmos de aprendizaje automático tienen como objetivo superar la maldición de la dimensionalidad asumiendo que los datos se encuentran cerca de una variedad de baja dimensión, como se describe en la sección 5.11.3.

Uno de los primeros intentos de aprovechar la hipótesis múltiple es el **distancia tangente**algoritmo ([Simard et al., 1993, 1998](#)). Es un algoritmo de vecino más cercano no paramétrico en el que la métrica utilizada no es la distancia euclídea genérica sino que se deriva del conocimiento de las variedades cerca de las cuales se concentra la probabilidad. Se supone que estamos tratando de clasificar ejemplos y que los ejemplos en la misma variedad comparten la misma categoría. Dado que el clasificador debe ser invariante a los factores locales de variación que corresponden al movimiento en la variedad, tendría sentido utilizar como distancia entre puntos al vecino más cercano X_1 y X_2 la distancia entre los colectores $METRO_1$ y $METRO_2$ que pertenecen respectivamente. Aunque eso puede ser computacionalmente difícil (requeriría resolver un problema de optimización para encontrar el par de puntos más cercano en $METRO_1$ y $METRO_2$), una alternativa barata que tiene sentido localmente es aproximar $METRO_i$ por su plano tangente en X_i medir la distancia entre las dos tangentes, o entre un plano tangente y un punto. Eso se puede lograr resolviendo un sistema lineal de baja dimensión (en la dimensión de las variedades). Por supuesto, este algoritmo requiere uno para especificar los vectores tangentes.

En un espíritu relacionado, el**puntal tangente**algoritmo ([Simard et al., 1992](#)) (cifra 7.9) entrena un clasificador de red neuronal con una penalización adicional para hacer que cada salida $F(X)$ de la red neuronal localmente invariante a factores de variación conocidos. Estos factores de variación corresponden al movimiento a lo largo de la variedad cerca de la cual se concentran los ejemplos de la misma clase. La invariancia local se logra requiriendo $\nabla_X F(X)$ ser ortogonal a los vectores tangentes múltiples conocidos v_i en X , o de manera equivalente que la derivada direccional de F en X en las direcciones v_i ser pequeño añadiendo una sanción de regularización Ω :

$$\Omega(F) = \sum_i \frac{(\nabla_X F(X)) \cdot v_i}{\|v_i\|^2} . \quad (7.67)$$

Este regularizador, por supuesto, se puede escalar mediante un hiperparámetro apropiado y, para la mayoría de las redes neuronales, necesitaríamos sumar muchas salidas en lugar de la única salida. $\mathcal{R}(\mathbf{X})$ descrito aquí por simplicidad. Al igual que con el algoritmo de distancia tangente, los vectores tangentes se derivan a priori, generalmente del conocimiento formal del efecto de transformaciones como traslación, rotación y escala en imágenes. La propiedad tangente se ha utilizado no solo para el aprendizaje supervisado (Simard et al., 1992) sino también en el contexto del aprendizaje por refuerzo (Thrun, 1995).

La propagación tangente está estrechamente relacionada con el aumento de conjuntos de datos. En ambos casos, el usuario del algoritmo codifica su conocimiento previo de la tarea especificando un conjunto de transformaciones que no deberían alterar la salida de la red. La diferencia es que, en el caso del aumento de conjuntos de datos, la red se entrena explícitamente para clasificar correctamente distintas entradas que se crearon al aplicar más que una cantidad infinitesimal de estas transformaciones. La propagación tangente no requiere visitar explícitamente un nuevo punto de entrada. En cambio, regulariza analíticamente el modelo para resistir la perturbación en las direcciones correspondientes a la transformación especificada. Si bien este enfoque analítico es intelectualmente elegante, tiene dos inconvenientes principales. Primero, solo regulariza el modelo para resistir la perturbación infinitesimal. El aumento explícito de conjuntos de datos confiere resistencia a perturbaciones más grandes. En segundo lugar, el enfoque infinitesimal plantea dificultades para los modelos basados en unidades lineales rectificadas. Estos modelos solo pueden reducir sus derivados apagando unidades o reduciendo sus pesos. No son capaces de encoger sus derivados saturando a un valor alto con pesos grandes, como sigmoides obroneado las unidades pueden. El aumento de conjuntos de datos funciona bien con unidades lineales rectificadas porque diferentes subconjuntos de unidades rectificadas pueden activarse para diferentes versiones transformadas de cada entrada original.

La propagación tangente también está relacionada con **puntal doble** (Drucker y Le Cun, 1992) y entrenamiento adversario (Szegedy et al., 2014b; Buen compañoero et al., 2014b). Double backprop regulariza el jacobiano para que sea pequeño, mientras que el entrenamiento contradictorio encuentra entradas cerca de las entradas originales y entrena el modelo para producir la misma salida en estas que en las entradas originales. Tanto la propagación de tangente como el aumento de conjuntos de datos mediante transformaciones especificadas manualmente requieren que el modelo sea invariable en ciertas direcciones de cambio especificadas en la entrada. El entrenamiento con doble apoyo y el adversario requieren que el modelo sea invariable para *todo* dirección de cambio en la entrada siempre que el cambio sea pequeño. Así como el aumento de conjuntos de datos es la versión no infinitesimal de la propagación tangente, el entrenamiento adversario es la versión no infinitesimal del backprop doble.

El clasificador tangente múltiple (rifai et al., 2011c), elimina la necesidad de conocer a priori los vectores tangentes. Como veremos en el capítulo 14, los codificadores automáticos pueden

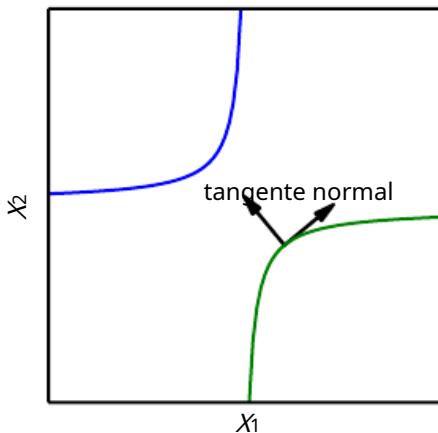


Figura 7.9: Ilustración de la idea principal del algoritmo de la tangente (Simard et al., 1992) y clasificador tangente múltiple (rifai et al., 2011c), que regularizan la función de salida del clasificador $f(X)$. Cada curva representa la variedad para una clase diferente, ilustrada aquí como una variedad unidimensional incrustada en un espacio bidimensional. En una curva, hemos elegido un solo punto y dibujado un vector que es tangente a la variedad de clase (paralelo a la variedad y tocando la variedad) y un vector que es normal a la variedad de clase (ortogonal a la variedad). En múltiples dimensiones puede haber muchas direcciones tangentes y muchas direcciones normales. Esperamos que la función de clasificación cambie rápidamente a medida que se mueve en la dirección normal a la variedad, y que no cambie a medida que se mueve a lo largo de la variedad de clase. Tanto la propagación tangente como el clasificador tangente múltiple regularizan $f(X)$ para no cambiar mucho como X se mueve a lo largo de la variedad. La propagación tangente requiere que el usuario especifique manualmente las funciones que calculan las direcciones tangentes (como especificar que las pequeñas traducciones de imágenes permanezcan en la misma variedad de clase), mientras que el clasificador tangente múltiple estima las direcciones tangentes múltiples entrenando un codificador automático para que se ajuste a los datos de entrenamiento. El uso de autocodificadores para estimar variedades se describirá en el capítulo 14.

estimar los múltiples vectores tangentes. El clasificador tangente múltiple hace uso de esta técnica para evitar la necesidad de vectores tangentes especificados por el usuario. Como se ilustra en la figura 14.10, estos vectores tangentes estimados van más allá de las invariantes clásicas que surgen de la geometría de las imágenes (como la translación, la rotación y el escalado) e incluyen factores que deben aprenderse porque son específicos del objeto (como las partes móviles del cuerpo). El algoritmo propuesto con el clasificador tangente múltiple es, por lo tanto, simple: (1) usar un autocodificador para aprender la estructura múltiple mediante aprendizaje no supervisado, y (2) usar estas tangentes para regularizar un clasificador de red neuronal como en prop tangente (ecuación 7.67).

Este capítulo ha descrito la mayoría de las estrategias generales utilizadas para regularizar las redes neuronales. La regularización es un tema central del aprendizaje automático y, como tal,

será revisado periódicamente por la mayoría de los capítulos restantes. Otro tema central del aprendizaje automático es la optimización, que se describe a continuación.

Capítulo 8

Optimización para entrenar modelos profundos

Los algoritmos de aprendizaje profundo implican optimización en muchos contextos. Por ejemplo, realizar inferencias en modelos como PCA implica resolver un problema de optimización. A menudo usamos la optimización analítica para escribir pruebas o diseñar algoritmos. De todos los muchos problemas de optimización involucrados en el aprendizaje profundo, el más difícil es el entrenamiento de redes neuronales. Es bastante común invertir días o meses en cientos de máquinas para resolver incluso una sola instancia del problema de entrenamiento de redes neuronales. Debido a que este problema es tan importante y costoso, se ha desarrollado un conjunto especializado de técnicas de optimización para resolverlo. Este capítulo presenta estas técnicas de optimización para el entrenamiento de redes neuronales.

Si no está familiarizado con los principios básicos de la optimización basada en gradientes, le sugerimos que revise el capítulo 4. Ese capítulo incluye una breve descripción de la optimización numérica en general.

Este capítulo se centra en un caso particular de optimización: encontrar los parámetros θ de una red neuronal que reduce significativamente una función de costo $J(\theta)$, que normalmente incluye una medida de rendimiento evaluada en todo el conjunto de entrenamiento, así como términos de regularización adicionales.

Comenzamos con una descripción de cómo la optimización utilizada como algoritmo de entrenamiento para una tarea de aprendizaje automático difiere de la optimización pura. A continuación, presentamos varios de los desafíos concretos que dificultan la optimización de las redes neuronales. Luego definimos varios algoritmos prácticos, incluidos los propios algoritmos de optimización y las estrategias para inicializar los parámetros. Los algoritmos más avanzados adaptan sus tasas de aprendizaje durante el entrenamiento o aprovechan la información contenida en

las segundas derivadas de la función de costo. Finalmente, concluimos con una revisión de varias estrategias de optimización que se forman al combinar algoritmos de optimización simples en procedimientos de nivel superior.

8.1 En qué se diferencia el aprendizaje de la optimización pura

Los algoritmos de optimización utilizados para el entrenamiento de modelos profundos difieren de los algoritmos de optimización tradicionales en varios aspectos. El aprendizaje automático suele actuar de forma indirecta. En la mayoría de los escenarios de aprendizaje automático, nos preocupamos por alguna medida de rendimiento PAG , que se define con respecto al conjunto de prueba y también puede ser intratable. Por lo tanto, optimizamos PAG sólo indirectamente. Reducimos una función de costo diferente $J(\theta)$ con la esperanza de que al hacerlo mejore PAG . Esto contrasta con la optimización pura, donde minimizar J es un objetivo en sí mismo. Los algoritmos de optimización para entrenar modelos profundos también suelen incluir alguna especialización en la estructura específica de las funciones objetivas de aprendizaje automático.

Por lo general, la función de costo se puede escribir como un promedio sobre el conjunto de entrenamiento, como

$$J(\theta) = \mathbb{E}_{(X,y) \sim p_{\text{datos}}} L(F(X; \theta), y), \quad (8.1)$$

dónde L es la función de pérdida por ejemplo, $F(X; \theta)$ es la salida predicha cuando la entrada es X , p_{datos} es la distribución empírica. En el caso del aprendizaje supervisado, y es la salida de destino. A lo largo de este capítulo, desarrollaremos el caso supervisado no regularizado, donde los argumentos para L son $F(X; \theta)$ y y . Sin embargo, es trivial extender este desarrollo, por ejemplo, para incluir θ o X como argumentos, o para excluir y como argumentos, con el fin de desarrollar diversas formas de regularización o aprendizaje no supervisado.

Ecuación 8.1 define una función objetivo con respecto al conjunto de entrenamiento. Por lo general, preferiríamos minimizar la función objetivo correspondiente donde la expectativa se toma a través de la *distribución de generación de datos* p_{datos} en lugar de solo sobre el conjunto de entrenamiento finito:

$$J^*(\theta) = \mathbb{E}_{(X,y) \sim p_{\text{datos}}} L(F(X; \theta), y). \quad (8.2)$$

8.1.1 Minimización empírica del riesgo

El objetivo de un algoritmo de aprendizaje automático es reducir el error de generalización esperado dado por la ecuación 8.2. Esta cantidad se conoce como **riesgo**. Enfatizamos aquí que la expectativa se toma sobre la verdadera distribución subyacente p_{datos} . Si conociéramos la verdadera distribución $p_{\text{datos}}(X, y)$, la minimización del riesgo sería una tarea de optimización

resoluble por un algoritmo de optimización. Sin embargo, cuando no sabemos $p_{\text{datos}}(X, y)$ pero solo tenemos un conjunto de muestras de entrenamiento, tenemos un problema de aprendizaje automático.

La forma más sencilla de convertir un problema de aprendizaje automático en un problema de optimización es minimizar la pérdida esperada en el conjunto de entrenamiento. Esto significa reemplazar la verdadera distribución $p_{\text{datos}}(X, y)$ con la distribución empírica $p_{\text{datos}}(X, y)$ definida por el conjunto de entrenamiento. Ahora minimizamos el **riesgo empírico**

$$\min_{\theta} \sum_{i=1}^m L(F(X_i; \theta), y_i) = \frac{1}{m} \sum_{i=1}^m L(F(X_i; \theta), y_i) \quad (8.3)$$

dónde m es el número de ejemplos de entrenamiento.

El proceso de entrenamiento basado en minimizar este error de entrenamiento promedio se conoce como **minimización empírica del riesgo**. En este entorno, el aprendizaje automático sigue siendo muy similar a la optimización directa. En lugar de optimizar el riesgo directamente, optimizamos el riesgo empírico y esperamos que el riesgo también disminuya significativamente. Una variedad de resultados teóricos establecen condiciones bajo las cuales se puede esperar que el riesgo real disminuya en varias cantidades.

Sin embargo, la minimización empírica del riesgo es propensa al sobreajuste. Los modelos con alta capacidad pueden simplemente memorizar el conjunto de entrenamiento. En muchos casos, la minimización empírica del riesgo no es realmente factible. Los algoritmos de optimización modernos más efectivos se basan en el descenso de gradiente, pero muchas funciones de pérdida útiles, como la pérdida 0-1, no tienen derivadas útiles (la derivada es cero o indefinida en todas partes). Estos dos problemas significan que, en el contexto del aprendizaje profundo, rara vez usamos la minimización empírica del riesgo. En su lugar, debemos utilizar un enfoque ligeramente diferente, en el que la cantidad que realmente optimizamos es aún más diferente de la cantidad que realmente queremos optimizar.

8.1.2 Funciones de pérdida sustituta y detención anticipada

A veces, la función de pérdida que realmente nos importa (por ejemplo, el error de clasificación) no se puede optimizar de manera eficiente. Por ejemplo, minimizar exactamente la pérdida esperada de 0-1 suele ser intratable (exponencial en la dimensión de entrada), incluso para un clasificador lineal ([Marcotte y Savard, 1992](#)). En tales situaciones, normalmente se optimiza un **función de pérdida sustituta** en cambio, que actúa como un proxy pero tiene ventajas. Por ejemplo, la probabilidad logarítmica negativa de la clase correcta se suele utilizar como sustituto de la pérdida de 0-1. La probabilidad logarítmica negativa permite que el modelo estime la probabilidad condicional de las clases, dada la entrada, y si el modelo puede hacerlo bien, puede elegir las clases que producen el menor error de clasificación esperado.

En algunos casos, una función de pérdida sustituta en realidad da como resultado poder aprender más. Por ejemplo, la pérdida 0-1 del conjunto de prueba a menudo continúa disminuyendo durante mucho tiempo después de que la pérdida 0-1 del conjunto de entrenamiento haya llegado a cero, cuando se entrena con el sustituto de probabilidad logarítmica. Esto se debe a que incluso cuando la pérdida esperada de 0-1 es cero, se puede mejorar la solidez del clasificador separando aún más las clases entre sí, obteniendo un clasificador más seguro y confiable, extrayendo así más información de los datos de entrenamiento de la que se obtendría. han sido posibles simplemente minimizando la pérdida promedio de 0-1 en el set de entrenamiento.

Una diferencia muy importante entre la optimización en general y la optimización tal como la usamos para entrenar algoritmos es que los algoritmos de entrenamiento no suelen detenerse en un mínimo local. En cambio, un algoritmo de aprendizaje automático generalmente minimiza una función de pérdida sustituta, pero se detiene cuando se aplica un criterio de convergencia basado en la detención anticipada (sección 7.8) Está satisfecho. Por lo general, el criterio de detención anticipada se basa en la verdadera función de pérdida subyacente, como la pérdida 0-1 medida en un conjunto de validación, y está diseñado para hacer que el algoritmo se detenga cada vez que comience a ocurrir un sobreajuste. El entrenamiento a menudo se detiene mientras la función de pérdida sustituta todavía tiene grandes derivadas, lo que es muy diferente de la configuración de optimización pura, donde se considera que un algoritmo de optimización ha convergido cuando el gradiente se vuelve muy pequeño.

8.1.3 Algoritmos por lotes y minilotes

Un aspecto de los algoritmos de aprendizaje automático que los separa de los algoritmos de optimización generales es que la función objetivo generalmente se descompone como una suma sobre los ejemplos de entrenamiento. Los algoritmos de optimización para el aprendizaje automático normalmente calculan cada actualización de los parámetros en función de un valor esperado de la función de costo estimada utilizando solo un subconjunto de los términos de la función de costo total.

Por ejemplo, los problemas de estimación de máxima verosimilitud, cuando se ven en el espacio de registro, se descomponen en una suma sobre cada ejemplo:

$$\theta_{ML} = \underset{\theta}{\text{argmax}} \sum_{i=1}^n \text{registro}_{\text{pag}}(\text{model}(x_i; \theta)). \quad (8.4)$$

Maximizar esta suma es equivalente a maximizar la expectativa sobre la distribución empírica definida por el conjunto de entrenamiento:

$$J(\theta) = \underset{y \sim p_{\text{datos}}}{\text{mix}} \text{registro}_{\text{pag}}(\text{model}(x; \theta)). \quad (8.5)$$

La mayoría de las propiedades de la función objetivo utilizados por la mayoría de nuestros algoritmos de optimización también son expectativas sobre el conjunto de entrenamiento. por ejemplo, el

La propiedad más utilizada es el gradiente:

$$\nabla_{\theta} J(\theta) = \text{mix}_{y \sim p_{\text{datos}}} \nabla_{\theta} \text{registro}_{\text{pag}}(\text{modelo}(X, y; \theta)). \quad (8.6)$$

Calcular exactamente esta expectativa es muy costoso porque requiere evaluar el modelo en cada ejemplo en el conjunto de datos completo. En la práctica, podemos calcular estas expectativas tomando muestras al azar de una pequeña cantidad de ejemplos del conjunto de datos y luego tomando el promedio solo de esos ejemplos.

Recuerde que el standard error de la media (ecuación 5.46) estimado a partir de n muestras está dada por σ/\sqrt{n} , donde σ es la verdadera desviación estándar del valor de las muestras. El denominador \sqrt{n} muestra que hay retornos menores que lineales al usar más ejemplos para estimar el gradiente. Compare dos estimaciones hipotéticas del gradiente, una basada en 100 ejemplos y otra basada en 10 000 ejemplos. Este último requiere 100 veces más cálculo que el primero, pero reduce el error estándar de la media solo por un factor de 10. La mayoría de los algoritmos de optimización convergen mucho más rápido (en términos de cálculo total, no en términos de número de actualizaciones) si son permitidos calcular rápidamente estimaciones aproximadas del gradiente en lugar de calcular lentamente el gradiente exacto.

Otra consideración que motiva la estimación estadística del gradiente a partir de un pequeño número de muestras es la redundancia en el conjunto de entrenamiento. En el peor de los casos, todos los *metros* muestras en el conjunto de entrenamiento podrían ser copias idénticas entre sí. Una estimación basada en muestreo del gradiente podría calcular el gradiente correcto con una sola muestra, usando m veces menos computación que el enfoque ingenuo. En la práctica, es poco probable que realmente encontramos esta situación del peor de los casos, pero podemos encontrar una gran cantidad de ejemplos que hacen contribuciones muy similares al gradiente.

Los algoritmos de optimización que utilizan todo el conjunto de entrenamiento se denominan **lote** o **determinista** métodos de gradiente, porque procesan todos los ejemplos de entrenamiento simultáneamente en un lote grande. Esta terminología puede ser algo confusa porque la palabra "lote" también se usa a menudo para describir el minilote utilizado por el descenso de gradiente estocástico de minilote. Por lo general, el término "descenso de gradiente por lotes" implica el uso del conjunto de entrenamiento completo, mientras que el uso del término "por lotes" para describir un grupo de ejemplos no lo hace. Por ejemplo, es muy común usar el término "tamaño de lote" para describir el tamaño de un mini lote.

Los algoritmos de optimización que usan un solo ejemplo a la vez a veces se denominan **estocástico** o **en línea** métodos. El término en línea generalmente se reserva para el caso en que los ejemplos se extraen de una secuencia de ejemplos creados continuamente en lugar de un conjunto de entrenamiento de tamaño fijo sobre el que se realizan varias pasadas.

La mayoría de los algoritmos utilizados para el aprendizaje profundo se encuentran en algún punto intermedio, utilizando más

que uno pero menos que todos los ejemplos de entrenamiento. Éstos se llamaban tradicionalmente **minilote** o **minilote estocástico** métodos y ahora es común llamarlos simplemente **estocástico** métodos.

El ejemplo canónico de un método estocástico es el descenso de gradiente estocástico, que se presenta en detalle en la sección 8.3.1.

Los tamaños de los minilotes generalmente dependen de los siguientes factores:

- Los lotes más grandes proporcionan una estimación más precisa del gradiente, pero con rendimientos menos que lineales.
- Las arquitecturas multinúcleo suelen estar infrautilizadas por lotes extremadamente pequeños. Esto motiva el uso de un tamaño de lote mínimo absoluto, por debajo del cual no hay reducción en el tiempo para procesar un mini lote.
- Si todos los ejemplos del lote se van a procesar en paralelo (como suele ser el caso), la cantidad de memoria aumenta con el tamaño del lote. Para muchas configuraciones de hardware, este es el factor limitante en el tamaño del lote.
- Algunos tipos de hardware logran un mejor tiempo de ejecución con tamaños específicos de arreglos. Especialmente cuando se usan GPU, es común que la potencia de 2 tamaños de lote ofrezca un mejor tiempo de ejecución. La potencia típica de 2 tamaños de lote varía de 32 a 256, y a veces se intentan 16 para modelos grandes.
- Los lotes pequeños pueden ofrecer un efecto regularizador ([Wilson y Martínez, 2003](#)), quizás por el ruido que añaden al proceso de aprendizaje. El error de generalización suele ser mejor para un tamaño de lote de 1. El entrenamiento con un tamaño de lote tan pequeño puede requerir una tasa de aprendizaje pequeña para mantener la estabilidad debido a la gran variación en la estimación del gradiente. El tiempo de ejecución total puede ser muy alto debido a la necesidad de realizar más pasos, tanto por la tasa de aprendizaje reducida como porque se necesitan más pasos para observar todo el conjunto de entrenamiento.

Diferentes tipos de algoritmos usan diferentes tipos de información del minilote de diferentes maneras. Algunos algoritmos son más sensibles al error de muestreo que otros, ya sea porque usan información que es difícil de estimar con precisión con pocas muestras o porque usan la información de maneras que amplifican más los errores de muestreo. Métodos que calculan las actualizaciones basándose únicamente en el gradiente *gramo* generalmente son relativamente robustos y pueden manejar lotes más pequeños como 100. Métodos de segundo orden, que también usan la matriz hessiana H y computar actualizaciones tales como $H^{-1} \text{gramo}$, por lo general requieren tamaños de lote mucho más grandes, como 10,000. Estos grandes tamaños de lote son necesarios para minimizar las fluctuaciones en las estimaciones de $H^{-1} \text{gramo}$. Suponer que H se estima perfectamente pero tiene un mal número de estado. Multiplicación por

H^{-1} su inversa amplifica errores preexistentes, en este caso, errores de estimación engramo . Cambios muy pequeños en la estimación degramo por lo tanto, puede causar grandes cambios en la actualización $H^{-1}\text{gramo}$, incluso si H se estimaron perfectamente. Por supuesto, H se estimará solo aproximadamente, por lo que la actualización $H^{-1}\text{gramo}$ contendrá aún más error de lo que predeciríamos al aplicar una operación pobremente condicionada a la estimación degramo .

También es crucial que los minilotes se seleccionen al azar. Calcular una estimación no sesgada del gradiente esperado de un conjunto de muestras requiere que esas muestras sean independientes. También deseamos que dos estimaciones de gradiente posteriores sean independientes entre sí, por lo que dos minilotes de ejemplos posteriores también deberían ser independientes entre sí. Muchos conjuntos de datos se organizan de manera más natural de una manera en la que los ejemplos sucesivos están altamente correlacionados. Por ejemplo, podríamos tener un conjunto de datos de datos médicos con una larga lista de resultados de análisis de muestras de sangre. Esta lista podría organizarse de modo que primero tengamos cinco muestras de sangre tomadas en diferentes momentos del primer paciente, luego tengamos tres muestras de sangre tomadas del segundo paciente, luego las muestras de sangre del tercer paciente, y así sucesivamente. Si tuviéramos que sacar ejemplos en orden de esta lista, entonces cada uno de nuestros minilotes estaría extremadamente sesgado, porque representaría principalmente a un paciente de los muchos pacientes en el conjunto de datos. En casos como estos, donde el orden del conjunto de datos tiene alguna importancia, es necesario mezclar los ejemplos antes de seleccionar los minilotes. Para conjuntos de datos muy grandes, por ejemplo, conjuntos de datos que contienen miles de millones de ejemplos en un centro de datos, puede ser poco práctico muestrear ejemplos de manera verdaderamente uniforme al azar cada vez que queremos construir un minilote. Afortunadamente, en la práctica suele ser suficiente mezclar el orden del conjunto de datos una vez y luego almacenarlo de forma aleatoria. Esto impondrá un conjunto fijo de posibles minilotes de ejemplos consecutivos que todos los modelos entrenados a partir de entonces utilizarán, y cada modelo individual se verá obligado a reutilizar este orden cada vez que pase por los datos de entrenamiento. Sin embargo, esta desviación de la verdadera selección aleatoria no parece tener un efecto perjudicial significativo. Si nunca se mezclan los ejemplos de ninguna manera, se puede reducir seriamente la efectividad del algoritmo.

Muchos problemas de optimización en el aprendizaje automático se descomponen en ejemplos lo suficientemente bien como para que podamos calcular actualizaciones separadas completas en diferentes ejemplos en paralelo. En otras palabras, podemos calcular la actualización que minimiza $J(\boldsymbol{\theta})$ para un minilote de ejemplos \mathcal{X} al mismo tiempo que calculamos la actualización de varios otros minilotes. Estos enfoques asíncronos distribuidos en paralelo se analizan más adelante en la sección 12.1.3.

Una motivación interesante para el descenso de gradiente estocástico de minilotes es que sigue el gradiente del verdadero *error de generalización* (ecuación 8.2) siempre que no se repitan ejemplos. La mayoría de las implementaciones de gradiente estocástico de minilotes

descender mezcla el conjunto de datos una vez y luego lo pasa varias veces. En el primer paso, cada minilote se usa para calcular una estimación imparcial del verdadero error de generalización. En el segundo paso, la estimación se sesga porque se forma volviendo a muestrear valores que ya se han utilizado, en lugar de obtener nuevas muestras justas de la distribución de generación de datos.

El hecho de que el descenso de gradiente estocástico minimiza el error de generalización es más fácil de ver en el caso de aprendizaje en línea, donde se extraen ejemplos o minilotes de un **arroyo** de datos. En otras palabras, en lugar de recibir un conjunto de entrenamiento de tamaño fijo, el aprendizaje es similar a un ser vivo que ve un nuevo ejemplo a cada instante, con cada ejemplo (X, y) procedente de la distribución de generación de datos $p_{\text{datos}}(X, y)$. En este escenario, los ejemplos nunca se repiten; cada experiencia es una muestra justa de p_{datos} .

La equivalencia es más fácil de obtener cuando ambos X y y son discretos. En este caso, el error de generalización (ecuación 8.2) se puede escribir como una suma

$$j^*(\theta) = \sum_{X, y} p_{\text{datos}}(X, y) L(F(X; \theta), y), \quad (8.7)$$

con la pendiente exacta

$$\text{gramo} = \nabla_{\theta} j^*(\theta) = \sum_{X, y} p_{\text{datos}}(X, y) \nabla_{\theta} L(F(X; \theta), y). \quad (8.8)$$

Ya hemos visto el mismo hecho demostrado para el log-verosimilitud en la ecuación 8.5 y ecuación 8.6; observamos ahora que esto es válido para otras funciones L además de la probabilidad. Un resultado similar puede obtenerse cuando X y y son continuos, bajo supuestos leves con respecto a p_{datos} y L .

Por lo tanto, podemos obtener un estimador no sesgado del gradiente exacto del error de generalización muestreando un minilote de ejemplos $\{X(1), \dots, X(metro)\}$ con los objetivos correspondientes $y(i)$ de la distribución de generación de datos p_{datos} , y calculando el gradiente de pérdida con respecto a los parámetros para ese minilote:

$$\text{gramo} = \frac{1}{metro} \sum_i L(F(X(i); \theta), y(i)). \quad (8.9)$$

Actualizando θ en la dirección de gramo realiza SGD en el error de generalización.

Por supuesto, esta interpretación solo se aplica cuando los ejemplos no se reutilizan. No obstante, normalmente es mejor hacer varias pasadas a través del conjunto de entrenamiento, a menos que el conjunto de entrenamiento sea extremadamente grande. Cuando se utilizan múltiples épocas de este tipo, solo la primera época sigue el gradiente no sesgado del error de generalización, pero

por supuesto, las épocas adicionales generalmente brindan suficiente beneficio debido a la disminución del error de entrenamiento para compensar el daño que causan al aumentar la brecha entre el error de entrenamiento y el error de prueba.

Dado que algunos conjuntos de datos crecen rápidamente en tamaño, más rápido que el poder de cómputo, cada vez es más común que las aplicaciones de aprendizaje automático usen cada ejemplo de entrenamiento solo una vez o incluso que realicen un paso incompleto a través del conjunto de entrenamiento. Cuando se usa un conjunto de entrenamiento extremadamente grande, el sobreajuste no es un problema, por lo que el ajuste insuficiente y la eficiencia computacional se convierten en las preocupaciones predominantes. Ver también [Bottou y Bousquet\(2008\)](#) para una discusión sobre el efecto de los cuellos de botella computacionales en el error de generalización, a medida que crece el número de ejemplos de entrenamiento.

8.2 Desafíos en la optimización de redes neuronales

La optimización en general es una tarea extremadamente difícil. Tradicionalmente, el aprendizaje automático ha evitado la dificultad de la optimización general mediante el diseño cuidadoso de la función objetivo y las restricciones para garantizar que el problema de optimización sea convexo. Cuando entrenamos redes neuronales, debemos enfrentar el caso general no convexo. Incluso la optimización convexa no está exenta de complicaciones. En esta sección, resumimos varios de los desafíos más destacados involucrados en la optimización para entrenar modelos profundos.

8.2.1 Mal condicionamiento

Algunos desafíos surgen incluso cuando se optimizan funciones convexas. De estos, el más destacado es el mal acondicionamiento de la matriz de Hesse, H . Este es un problema muy general en la mayoría de las optimizaciones numéricas, convexas o no, y se describe con más detalle en la sección [4.3.1](#).

En general, se cree que el problema del mal condicionamiento está presente en los problemas de entrenamiento de redes neuronales. El mal condicionamiento puede manifestarse al hacer que SGD se "atasque" en el sentido de que incluso pasos muy pequeños aumentan la función de costo.

Recuperar de la ecuación [4.9](#) que una expansión en serie de Taylor de segundo orden de la función de costo predice que un paso de descenso de gradiente de $-g$ agregaría

$$\frac{1}{2} \cdot g^T H g - g^T g \quad (8.10)$$

al costo. El mal acondicionamiento del gradiente se convierte en un problema cuando $\lambda^{-1} H$ excede $\gamma^{-1} g^T g$. Para determinar si el mal acondicionamiento es perjudicial para una tarea de entrenamiento de redes neuronales, se puede monitorear la norma de gradiente cuadrática $g^T g$ y

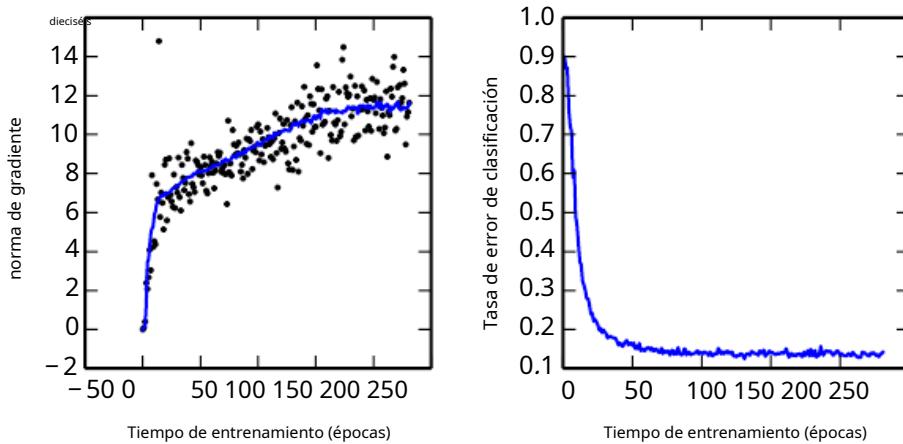


Figura 8.1: El descenso de gradiente a menudo no llega a un punto crítico de ningún tipo. En este ejemplo, la norma de gradiente aumenta a lo largo del entrenamiento de una red convolucional utilizada para la detección de objetos.(Izquierda)Un diagrama de dispersión que muestra cómo se distribuyen las normas de las evaluaciones de gradientes individuales a lo largo del tiempo. Para mejorar la legibilidad, solo se traza una norma de gradiente por época. El promedio móvil de todas las normas de gradiente se representa como una curva sólida. La norma de gradiente aumenta claramente con el tiempo, en lugar de disminuir como cabría esperar si el proceso de entrenamiento convergiera en un punto crítico.(Bien)A pesar del gradiente creciente, el proceso de formación es razonablemente exitoso. El error de clasificación del conjunto de validación disminuye a un nivel bajo.

el *gramo-Hgtérmino*. En muchos casos, la norma del gradiente no se reduce significativamente a lo largo del aprendizaje, pero *lagromo-Hgtérmino* crece en más de un orden de magnitud. El resultado es que el aprendizaje se vuelve muy lento a pesar de la presencia de un fuerte gradiente porque la tasa de aprendizaje debe reducirse para compensar una curvatura aún más fuerte. Cifra8.1 muestra un ejemplo del gradiente que aumenta significativamente durante el entrenamiento exitoso de una red neuronal.

Aunque el mal condicionamiento está presente en otros entornos además del entrenamiento de redes neuronales, algunas de las técnicas utilizadas para combatirlo en otros contextos son menos aplicables a las redes neuronales. Por ejemplo, el método de Newton es una herramienta excelente para minimizar funciones convexas con matrices hessianas mal condicionadas, pero en las secciones siguientes argumentaremos que el método de Newton requiere una modificación significativa antes de que pueda aplicarse a las redes neuronales.

8.2.2 Mínimos locales

Una de las características más destacadas de un problema de optimización convexo es que puede reducirse al problema de encontrar un mínimo local. Cualquier mínimo local es

garantizado que sea un mínimo global. Algunas funciones convexas tienen una región plana en la parte inferior en lugar de un único punto mínimo global, pero cualquier punto dentro de dicha región plana es una solución aceptable. Al optimizar una función convexa, sabemos que hemos llegado a una buena solución si encontramos un punto crítico de cualquier tipo.

Con funciones no convexas, como las redes neuronales, es posible tener muchos mínimos locales. De hecho, casi cualquier modelo profundo está esencialmente garantizado para tener un número extremadamente grande de mínimos locales. Sin embargo, como veremos, esto no es necesariamente un problema importante.

Las redes neuronales y cualquier modelo con múltiples variables latentes parametrizadas de manera equivalente tienen múltiples mínimos locales debido a la **identificabilidad del modelo** problema. Se dice que un modelo es identificable si un conjunto de entrenamiento lo suficientemente grande puede descartar todos los parámetros del modelo excepto uno. Los modelos con variables latentes a menudo no son identificables porque podemos obtener modelos equivalentes intercambiando variables latentes entre sí. Por ejemplo, podríamos tomar una red neuronal y modificar la capa 1 intercambiando el vector de peso entrante por unidad/ α el vector de peso entrante por unidad/ β , luego haciendo lo mismo para los vectores de peso salientes. Si tenemos m capas y n unidades cada uno, entonces hay $(m+1)n^2$ formas de disponer las unidades ocultas. Este tipo de no identificabilidad se conoce como **simetría del espacio de peso**.

Además de la simetría del espacio de peso, muchos tipos de redes neuronales tienen causas adicionales de no identificabilidad. Por ejemplo, en cualquier red lineal rectificada o maxout, podemos escalar todos los pesos y sesgos entrantes de una unidad por α y también escalamos todos sus pesos salientes por β . Esto significa que—si el costo de la función no incluye términos como el decaimiento del peso que depende directamente de los pesos en lugar de los resultados de los modelos: cada mínimo local de una red lineal rectificada o maxout se encuentra en un $(m+1)n^2$ -dimensional hipérbola de mínimos locales equivalentes.

Estos problemas de identificabilidad del modelo significan que puede haber una cantidad extremadamente grande o incluso incontablemente infinita de mínimos locales en una función de costo de red neuronal. Sin embargo, todos estos mínimos locales que surgen de la no identificabilidad son equivalentes entre sí en el valor de la función de costo. Como resultado, estos mínimos locales no son una forma problemática de no convexidad.

Los mínimos locales pueden ser problemáticos si tienen un alto costo en comparación con el mínimo global. Se pueden construir pequeñas redes neuronales, incluso sin unidades ocultas, que tengan mínimos locales con un costo mayor que el mínimo global (Sontag y Sussman, 1989; Brady et al., 1989; Gori y Tesi, 1992). Si los mínimos locales con alto costo son comunes, esto podría plantear un problema grave para los algoritmos de optimización basados en gradientes.

Sigue siendo una pregunta abierta si hay muchos mínimos locales de alto costo

para redes de interés práctico y si los algoritmos de optimización las encuentran. Durante muchos años, la mayoría de los profesionales creían que los mínimos locales eran un problema común que afectaba a la optimización de redes neuronales. Hoy en día, ese no parece ser el caso. El problema sigue siendo un área activa de investigación, pero los expertos ahora sospechan que, para redes neuronales suficientemente grandes, la mayoría de los mínimos locales tienen un valor de función de bajo costo, y que no es importante encontrar un verdadero mínimo global en lugar de encontrar un punto en espacio de parámetros que tiene un costo bajo pero no mínimo ([Sajonia et al., 2013](#); [Delfin et al., 2014](#); [Buen compañero et al., 2015](#); [Choromanska et al., 2014](#)).

Muchos profesionales atribuyen casi todas las dificultades con la optimización de redes neuronales a los mínimos locales. Alentamos a los profesionales a probar cuidadosamente los problemas específicos. Una prueba que puede descartar mínimos locales como problema es trazar la norma del gradiente a lo largo del tiempo. Si la norma del gradiente no se reduce a un tamaño insignificante, el problema no son mínimos locales ni ningún otro tipo de punto crítico. Este tipo de prueba negativa puede descartar mínimos locales. En espacios de grandes dimensiones, puede ser muy difícil establecer positivamente que los mínimos locales son el problema. Muchas estructuras distintas de los mínimos locales también tienen pequeños gradientes.

8.2.3 Mesetas, Puntos de Silla y Otras Regiones Planas

Para muchas funciones no convexas de alta dimensión, los mínimos (y máximos) locales son, de hecho, raros en comparación con otro tipo de punto con gradiente cero: un punto de silla. Algunos puntos alrededor de un punto de silla tienen un costo mayor que el punto de silla, mientras que otros tienen un costo menor. En un punto de silla, la matriz hessiana tiene valores propios positivos y negativos. Los puntos que se encuentran a lo largo de vectores propios asociados con valores propios positivos tienen un costo mayor que el punto silla, mientras que los puntos que se encuentran a lo largo de valores propios negativos tienen un valor más bajo. Podemos pensar en un punto silla como un mínimo local a lo largo de una sección transversal de la función de costo y un máximo local a lo largo de otra sección transversal. Ver figura [4.5](#) para una ilustración.

Muchas clases de funciones aleatorias exhiben el siguiente comportamiento: en espacios de baja dimensión, los mínimos locales son comunes. En espacios de dimensiones superiores, los mínimos locales son raros y los puntos de silla son más comunes. Para una función $F: \mathbb{R}^n \rightarrow \mathbb{R}$ de este tipo, la relación esperada entre el número de puntos de silla y los mínimos locales crece exponencialmente con n . Para comprender la intuición detrás de este comportamiento, observe que la matriz hessiana en un mínimo local solo tiene valores propios positivos. La matriz hessiana en un punto de silla tiene una mezcla de valores propios positivos y negativos. Imagina que el signo de cada valor propio se genera al lanzar una moneda. En una sola dimensión, es fácil obtener un mínimo local lanzando una moneda y obteniendo cara una vez. En n -dimensional espacio, es exponencialmente improbable que todos los lanzamientos de monedas