

Modelado de datos secuenciales mediante redes neuronales recurrentes

En el capítulo anterior, nos centramos en las redes neuronales convolucionales (CNN). Tratamos los bloques de construcción de las arquitecturas CNN y cómo implementar CNN profundas en PyTorch. Por último, hemos aprendido a utilizar las CNN para la clasificación de imágenes. En este capítulo, exploraremos las redes neuronales recurrentes (Recurrent Neural Networks. RNN) y veremos su aplicación en el modelado de datos secuenciales.

Trataremos los siguientes temas:

- Introducción a los datos secuenciales
- RNN para modelar secuencias
- Memoria a corto y largo plazo
- Retropropagación truncada en el tiempo

- Implementación de RNN multicapa para el modelado de secuencias en PyTorch
- Proyecto uno: análisis de sentimiento mediante RNN del conjunto de datos de reseñas de películas de IMDb
- Proyecto dos: modelado de lenguaje a nivel de personajes con celdas LSTM, utilizando datos de texto de *La isla misteriosa*, de Julio Verne
- Uso del recorte de gradiente para evitar la explosión de gradientes

Introducción a los datos secuenciales

Comencemos nuestra discusión sobre las RNN examinando la naturaleza de los datos secuenciales, que se conocen más comúnmente como «datos secuencia» o «secuencias». Examinaremos las propiedades únicas de las secuencias que las diferencian de otros tipos de datos. A continuación, veremos cómo representar los datos secuenciales y exploraremos las distintas categorías de modelos para datos secuenciales, que se basan en la entrada y la salida de un modelo. Esto nos ayudará a explorar la relación entre las RNN y las secuencias en este capítulo.

Modelado de datos secuenciales: el orden importa

Lo que hace que las secuencias sean únicas, en comparación con otros tipos de datos, es que los elementos de una secuencia aparecen en un orden determinado y no son independientes entre sí. Los algoritmos típicos de aprendizaje automático para el aprendizaje supervisado

asumen que la entrada son datos independientes e idénticamente distribuidos (Independent and Identically Distributed, IID), lo que significa que los ejemplos de entrenamiento son mutuamente independientes y tienen la misma distribución subyacente. En este sentido, basándose en la suposición de independencia mutua, el orden en el que se dan los ejemplos de entrenamiento al modelo es irrelevante. Por ejemplo, si tenemos una muestra formada por n ejemplos de entrenamiento, $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}$, el orden en que utilicemos los datos para entrenar nuestro algoritmo de aprendizaje automático no importa. Un ejemplo de este escenario sería el conjunto de datos Iris con el que hemos trabajado anteriormente. En el conjunto de datos de Iris, cada flor se ha medido de forma independiente, y las mediciones sobre una flor no influyen sobre las de otra.

Sin embargo, esta suposición no es válida cuando tratamos con secuencias: por definición, el orden importa. El pronóstico del valor de mercado de una acción concreta sería un ejemplo de este escenario. Por ejemplo, supongamos que tenemos una muestra de n ejemplos de entrenamiento, donde cada ejemplo de entrenamiento representa el valor de mercado de una determinada acción en un día concreto. Si nuestra tarea es pronosticar el valor de mercado de valores para los próximos tres días, tendría sentido considerar los precios anteriores de las acciones en un orden de fecha para derivar las tendencias en lugar de utilizar estos ejemplos de entrenamiento en un orden aleatorio.

Datos secuenciales frente a datos de series temporales

Los datos de series temporales son un tipo especial de datos

secuenciales en los que cada ejemplo está asociado a una dimensión temporal. En los datos de series temporales, las muestras se toman en fechas sucesivas y, por tanto, la dimensión temporal determina el orden entre los puntos de datos. Por ejemplo, las cotizaciones bursátiles y los registros de voz o de un discurso son datos de series temporales.

Por otro lado, no todos los datos secuenciales tienen la dimensión temporal. Por ejemplo, en los datos de texto o las secuencias de ADN, los ejemplos están ordenados, pero el texto o el ADN no se consideran datos de series temporales. Como verá, en este capítulo nos centraremos en ejemplos de procesamiento del lenguaje natural (Natural Language Processing, NLP) y de modelado de texto que no son datos de series temporales. Sin embargo, hay que tener en cuenta que las RNN también pueden utilizarse para datos de series temporales, lo que queda fuera del alcance del libro.

Representación de secuencias

Hemos establecido que el orden entre los puntos de datos es importante en los datos secuenciales, por lo que necesitamos encontrar una manera de aprovechar esta información de orden en un modelo de aprendizaje automático. En este capítulo, representaremos las secuencias como $\langle \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)} \rangle$. Los índices de superíndice indican el orden de las instancias, y la longitud de la secuencia es T . Para un ejemplo sensato de secuencias, considere los datos de series temporales, donde cada punto de ejemplo, $\mathbf{x}^{(t)}$, pertenece a un tiempo particular, t . La [Figura 15.1](#) muestra un ejemplo de datos de series temporales donde tanto las características de entrada (las \mathbf{x}) como las etiquetas de destino (las \mathbf{y}) siguen

naturalmente el orden según su eje temporal; por tanto, tanto las x como las y son secuencias.

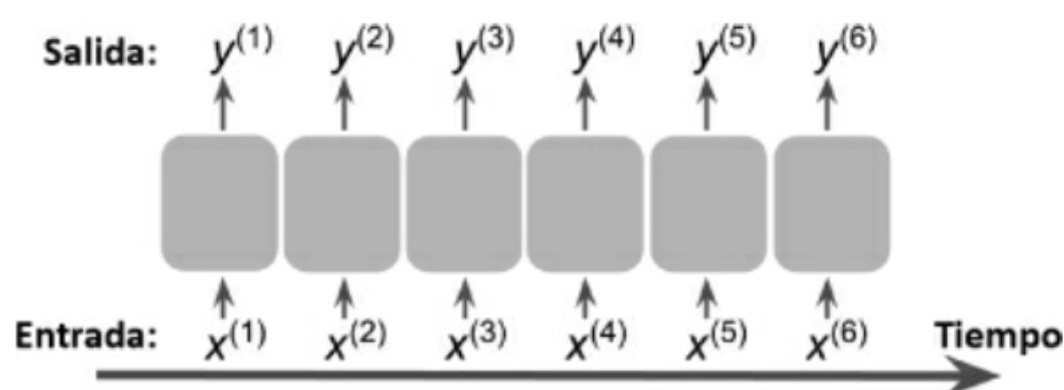


Figura 15.1 *Ejemplo de datos de series temporales.*

Como ya hemos mencionado, los modelos de NN estándar que hemos tratado hasta ahora, como los perceptrones multicapa (MLP) y las CNN para datos de imagen, asumen que los ejemplos de entrenamiento son independientes entre sí y, por tanto, no incorporan información de ordenación. Podemos decir que tales modelos no tienen una memoria de ejemplos de entrenamiento previamente vistos. Por ejemplo, las muestras pasan por los pasos de prealimentación y de retropropagación, y los pesos se actualizan independientemente del orden en que se procesan los ejemplos de entrenamiento.

Las RNN, por el contrario, están diseñadas para modelar secuencias y son capaces de recordar información pasada y procesar nuevos eventos en consecuencia, lo que supone una clara ventaja cuando se trabaja con datos de secuencias.

Diferentes categorías de modelado de secuencias

El modelado de secuencias tiene muchas aplicaciones fascinantes,

como la traducción de idiomas (por ejemplo, la traducción de textos del inglés al alemán), el subtitulado de imágenes y la generación de textos. Sin embargo, para elegir una arquitectura y un enfoque apropiados, tenemos que entender y ser capaces de distinguir entre estas diferentes tareas de modelado de secuencias. La [Figura 15.2](#), basada en las explicaciones del excelente artículo «The Unreasonable Effectiveness of Recurrent Neural Networks», de Andrej Karpathy, 2015 (<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>), resume las tareas de modelado de secuencias más comunes, que dependen de las categorías de la relación entre los datos de entrada y salida.

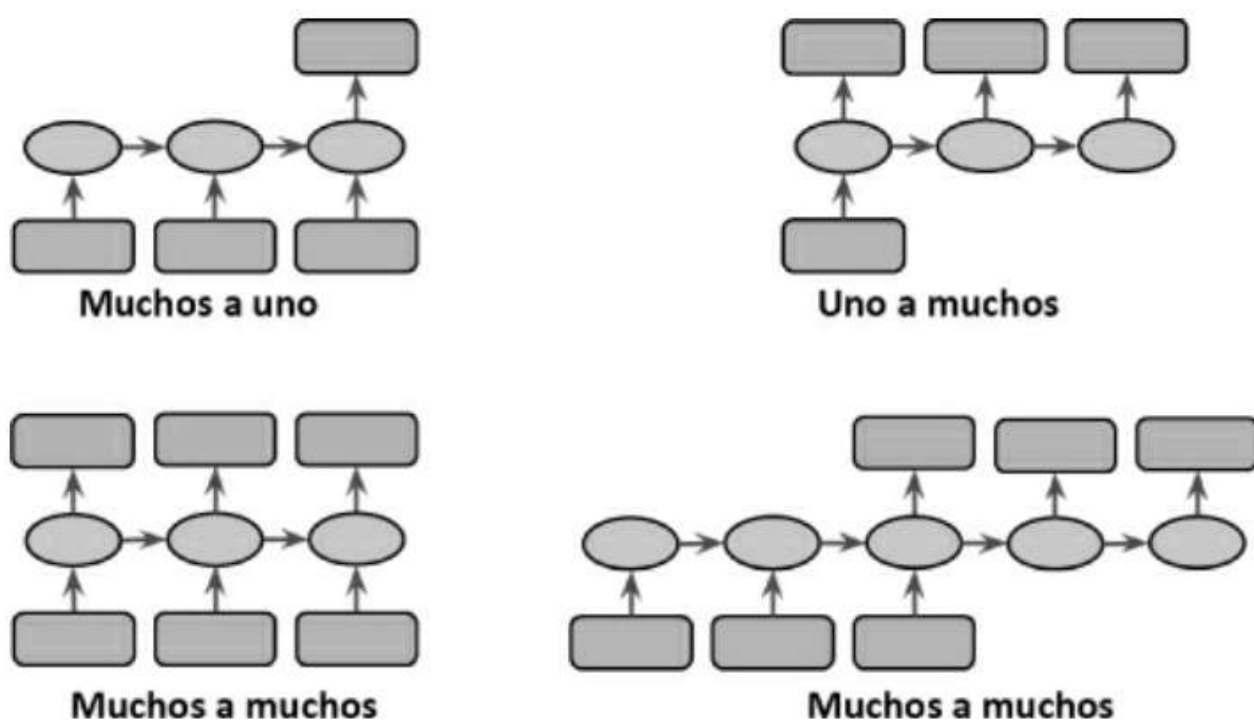


Figura 15.2 Tareas de secuenciación más comunes.

Analicemos con más detalle las diferentes categorías de la relación entre los datos de entrada y de salida, que se han representado en la figura anterior. Si ni los datos de entrada ni los de salida representan secuencias, entonces estamos tratando con datos estándar, y podríamos simplemente utilizar un perceptrón multicapa (u otro

modelo de clasificación previamente tratado en el libro) para modelar tales datos. Sin embargo, si la entrada o la salida es una secuencia, la tarea de modelado probablemente cae en una de estas categorías:

- **Muchos a uno:** los datos de entrada son una secuencia, pero la salida es un vector o escalar de tamaño fijo, no una secuencia. Por ejemplo, en el análisis de sentimiento, la entrada se basa en un texto (por ejemplo, una crítica de una película) y la salida es una etiqueta de clase (por ejemplo, una etiqueta que denota si a un crítico le ha gustado la película).
- **Uno a muchos:** los datos de entrada tienen un formato estándar y no son una secuencia, pero la salida es una secuencia. Un ejemplo de esta categoría es el subtitulado de imágenes: la entrada es una imagen y la salida es una frase en inglés que resume el contenido de esa imagen.
- **Muchos a muchos:** tanto la matriz de entrada como la de salida son secuencias. Esta categoría puede dividirse a su vez en función de si la entrada y la salida están sincronizadas. Un ejemplo de tarea de modelado sincronizado de muchos a muchos es la clasificación de vídeos, en la que se etiqueta cada fotograma. Un ejemplo de tarea de modelado de muchos a muchos con retraso sería la traducción de un idioma a otro. Por ejemplo, una frase entera en inglés debe ser leída y procesada por una máquina antes de que se produzca su traducción al alemán.

Ahora, tras resumir las tres grandes categorías de modelado de secuencias, podemos pasar a discutir la estructura de una RNN.

RNN para modelar secuencias

En esta sección, antes de empezar a implementar las RNN en PyTorch, discutiremos los principales conceptos de las RNN. Empezaremos por ver la estructura típica de una RNN, que incluye un componente recursivo para modelar los datos de la secuencia. A continuación, examinaremos cómo se calculan las activaciones de las neuronas en una RNN típica. Esto creará el contexto para que discutamos los desafíos típicos en el entrenamiento de las RNN, y luego discutiremos las soluciones a estos desafíos, como las LSTM y las unidades recurrentes cerradas (Gated Recurrent Units, GRU).

Comprender el flujo de datos en las RNN

Empecemos con la arquitectura de las RNN. La [Figura 15.3](#) muestra el flujo de datos en una NN estándar de prealimentación y en una RNN lado a lado para su comparación:

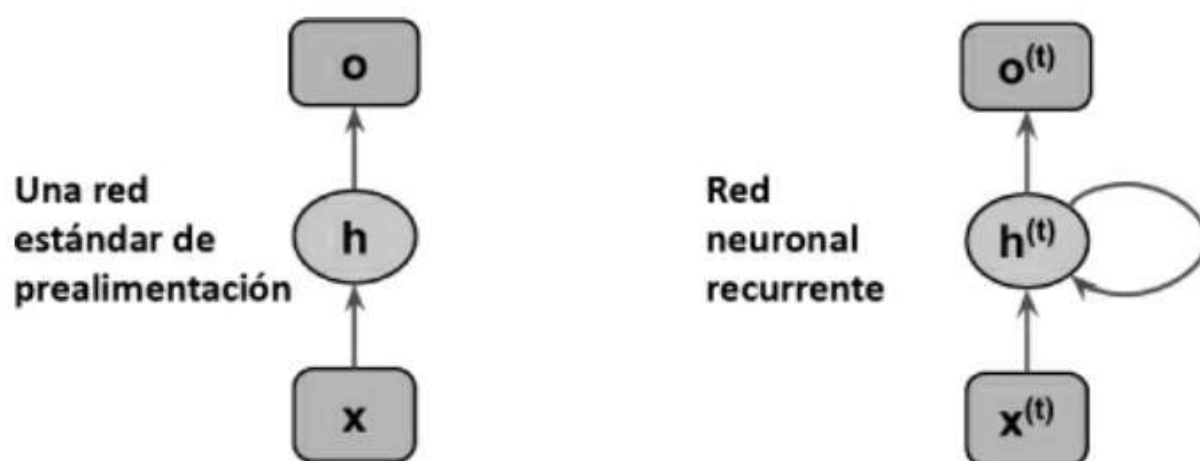


Figura 15.3 Flujo de datos de una NN estándar de prealimentación y una RNN.

Las dos redes tienen una sola capa oculta. En esta representación, las unidades no se muestran, pero suponemos que la capa de entrada (x),

la capa oculta (\mathbf{h}) y la capa de salida (\mathbf{o}) son vectores que contienen muchas unidades.

Determinación del tipo de salida de una RNN



Esta arquitectura RNN genérica podría corresponder a las dos categorías de modelado de secuencias en las que la entrada es una secuencia. Normalmente, una capa recurrente puede devolver una secuencia como salida, $\{\mathbf{o}^{(0)}, \mathbf{o}^{(1)}, \dots, \mathbf{o}^{(T)}\}$, o simplemente devolver la última salida (en $t = T$, es decir, $\mathbf{o}^{(T)}$). Por lo tanto, podría ser de muchos a muchos, o podría ser de muchos a uno si, por ejemplo, solo utilizamos el último elemento, $\mathbf{o}^{(T)}$, como salida final.

Veremos más adelante cómo se maneja esto en el módulo de PyTorch `torch.nn`, cuando echemos un vistazo detallado al comportamiento de una capa recurrente con respecto a devolver una secuencia como salida.

En una red estándar de prealimentación, la información fluye desde la entrada a la capa oculta, y luego desde la capa oculta a la capa de salida. Por otro lado, en una RNN, la capa oculta recibe su entrada tanto de la capa de entrada del instante de tiempo actual como de la capa oculta del instante de tiempo anterior.

El flujo de información en instantes de tiempo adyacentes en la capa oculta permite a la red tener una memoria de eventos pasados. Este flujo de información suele mostrarse como un bucle, también conocido como arista recurrente en la notación gráfica, que es el motivo por el que esta arquitectura general de RNN obtuvo su nombre.

Al igual que los perceptrones multicapa, las RNN pueden constar de varias capas ocultas. Tenga en cuenta que es una convención común referirse a las RNN con una capa oculta como RNN de una capa, que no debe confundirse con las NN de una capa sin capa oculta, como son Adaline o la regresión logística. La [Figura 15.4](#) ilustra una RNN con una capa oculta (arriba) y una RNN con dos capas ocultas (abajo):

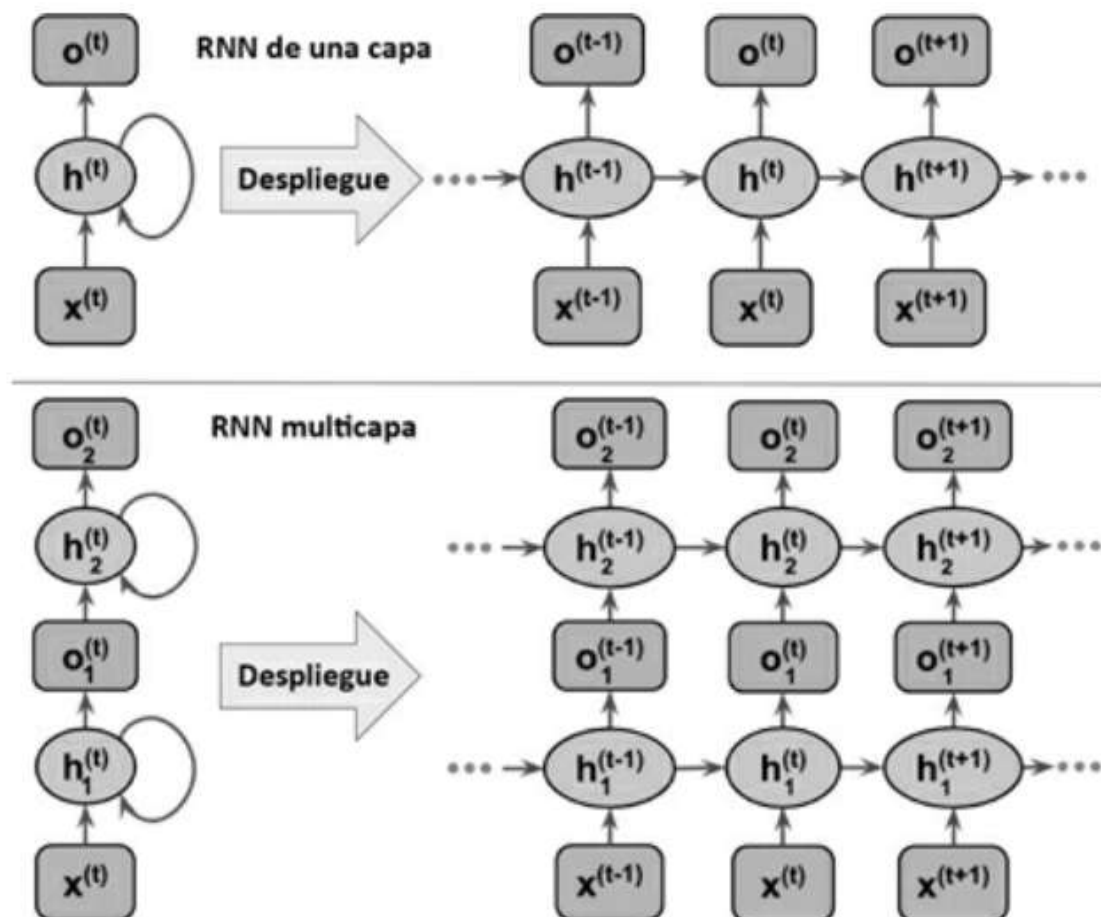


Figura 15.4 Ejemplos de una RNN con una y dos capas ocultas.

Para examinar la arquitectura de las RNN y el flujo de información, se puede desplegar una representación compacta con una arista recurrente, que se puede ver en la [Figura 15.4](#).

Como sabemos, cada unidad oculta de una NN estándar recibe solo una entrada: la preactivación de la red asociada a la capa de entrada. En cambio, cada unidad oculta de una RNN recibe dos conjuntos

distintos de entradas: la preactivación de la capa de entrada y la activación de la misma capa oculta del instante de tiempo anterior, $t - 1$.

En el primer instante de tiempo, $t = 0$, las unidades ocultas se inicializan con ceros o pequeños valores aleatorios. A continuación, en un instante de tiempo en el que $t > 0$, las unidades ocultas reciben su entrada del punto de datos en el tiempo presente, $\mathbf{x}^{(t)}$, y los valores anteriores de las unidades ocultas en $t - 1$, indicados como $\mathbf{h}^{(t-1)}$.

En el caso de una RNN multicapa, podemos resumir el flujo de información como sigue:

- *layer* = 1: Aquí, la capa oculta se representa como $\mathbf{h}_1^{(t)}$ y recibe su entrada del punto de datos, $\mathbf{x}^{(t)}$, y los valores ocultos en la misma capa, pero del instante de tiempo anterior, $\mathbf{h}_1^{(t-1)}$.
- *layer* = 2: La segunda capa oculta, $\mathbf{h}_2^{(t)}$, recibe sus entradas de las salidas de la capa inferior en el instante de tiempo presente ($\mathbf{o}_1^{(t)}$) y sus propios valores ocultos del instante de tiempo anterior, $\mathbf{h}_2^{(t-1)}$.

Dado que, en este caso, cada capa recurrente debe recibir una secuencia como entrada, todas las capas recurrentes, excepto la última, deben devolver una secuencia como salida (es decir, posteriormente tendremos que establecer `return_sequences=True`). El comportamiento de la última capa recurrente depende del tipo de problema.

Cálculo de activaciones en RNN

Ahora que entiende la estructura y el flujo general de información

en una RNN, vamos a ser más específicos y calcular las activaciones reales de las capas ocultas, así como la capa de salida. En este caso, para simplificar, consideraremos una sola capa oculta; sin embargo, el mismo concepto se aplica a las RNN multicapa.

Cada arista dirigida (las conexiones entre cajas) en la representación de una RNN que acabamos de ver está asociada a una matriz de pesos. Esos pesos no dependen del tiempo, t ; por tanto, son compartidos a lo largo del eje temporal. Las diferentes matrices de pesos en una RNN de una capa son las siguientes:

- W_{xh} : la matriz de pesos entre la entrada, $x^{(t)}$, y la capa oculta, h
- W_{hh} : la matriz de pesos asociada a la arista recurrente
- W_{ho} : la matriz de pesos entre la capa oculta y la capa de salida

Estas matrices de pesos se representan en la [Figura 15.5](#):

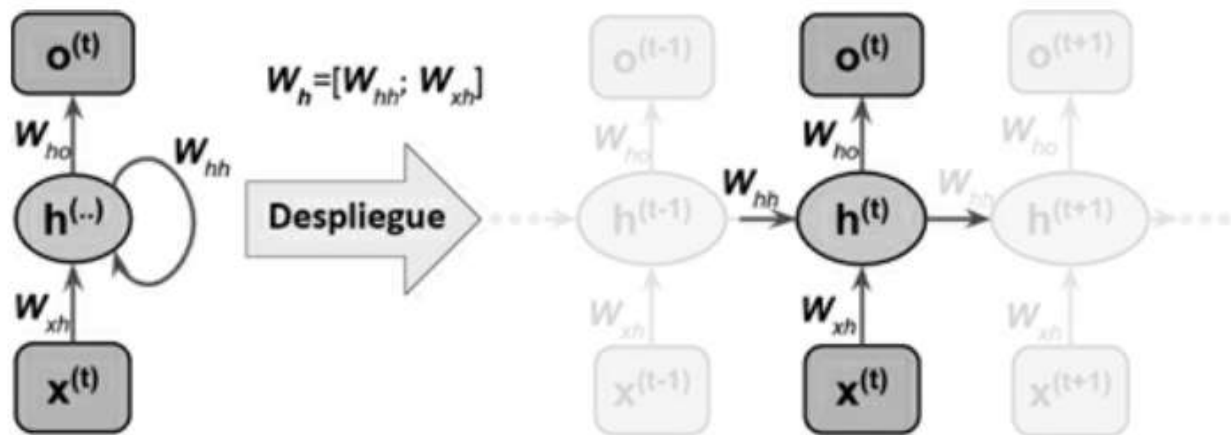


Figura 15.5 Aplicación de pesos a una RNN de una capa.

En ciertas implementaciones, se puede observar que las matrices de pesos, W_{xh} y W_{hh} , se concatenan en una matriz combinada, $W_h = [W_{xh}; W_{hh}]$. Más adelante en esta sección, también utilizaremos esta notación.

El cálculo de las activaciones es muy similar al de los perceptrones multicapa estándar y otros tipos de NN de prealimentación. Para la capa oculta, la entrada de la red, \mathbf{z}_h (preactivación), se calcula mediante una combinación lineal; es decir, calculamos la suma de las multiplicaciones de las matrices de pesos con los vectores correspondientes y añadimos la unidad de sesgo:

$$\mathbf{z}_h^{(t)} = \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h$$

Entonces, las activaciones de las unidades ocultas en el paso de tiempo, t , se calculan como sigue:

$$\mathbf{h}^{(t)} = \sigma_h(\mathbf{z}_h^{(t)}) = \sigma_h(\mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h)$$

Aquí, \mathbf{b}_h es el vector de sesgo para las unidades ocultas y $\sigma(\cdot)$ es la función de activación de la capa oculta.

En caso de querer utilizar la matriz de pesos concatenados, $\mathbf{W}_h = [\mathbf{W}_{xh}; \mathbf{W}_{hh}]$, la fórmula para calcular las unidades ocultas cambiará:

$$\mathbf{h}^{(t)} = \sigma_h\left([\mathbf{W}_{xh}; \mathbf{W}_{hh}] \begin{bmatrix} \mathbf{x}^{(t)} \\ \mathbf{h}^{(t-1)} \end{bmatrix} + \mathbf{b}_h\right)$$

Una vez calculadas las activaciones de las unidades ocultas en el instante de tiempo actual, se calcularán las activaciones de las unidades de salida:

$$\mathbf{o}^{(t)} = \sigma_o(\mathbf{W}_{ho}\mathbf{h}^{(t)} + \mathbf{b}_o)$$

Para ayudar a aclarar lo anterior, la [Figura 15.6](#) muestra el proceso de cálculo de estas activaciones con ambas formulaciones:

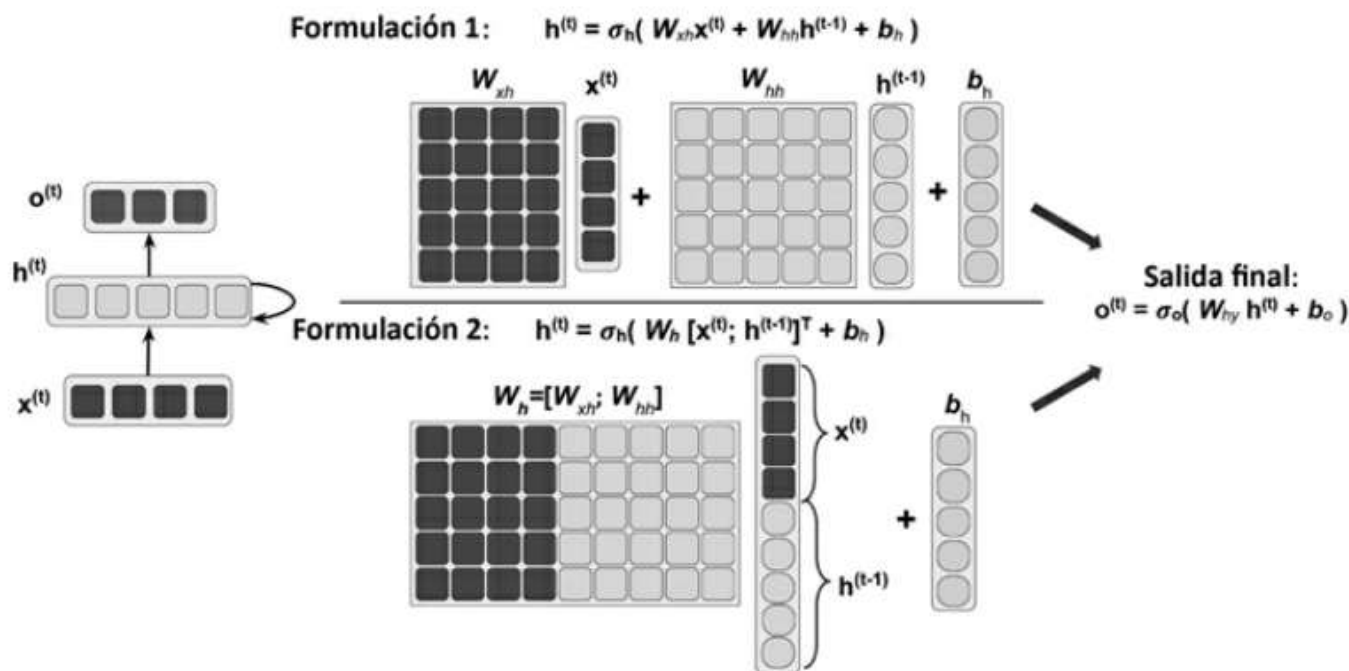


Figura 15.6 Cálculo de las activaciones.

Entrenamiento de RNN mediante retropropagación en el tiempo (BackPropagation Through Time, BPTT)

El algoritmo de aprendizaje para las RNN se introdujo en 1990: «Backpropagation Through Time: What It Does and How to Do It», Paul Werbos, *Proceedings of IEEE*, 78(10): 1550-1560, 1990.

El cálculo de los gradientes puede ser un poco complicado, pero la idea básica es que la pérdida global, L , es la suma de todas las funciones de pérdida en los tiempos de $t = 1$ a $t = T$:

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

Como la pérdida en el tiempo t depende de las unidades ocultas en todos los instantes de tiempo anteriores $1 : t$, el gradiente se calculará como sigue:

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \times \frac{\partial o^{(t)}}{\partial h^{(t)}} \times \left(\sum_{k=1}^t \frac{\partial h^{(k)}}{\partial h^{(t)}} \times \frac{\partial h^{(k)}}{\partial W_{hh}} \right)$$

Aquí, $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$ se calcula mediante una multiplicación de instantes de tiempo adyacentes:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^t \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

Recurrencia oculta frente a recurrencia de salida

Hasta ahora, se han visto redes recurrentes en las que la capa oculta tiene la propiedad recurrente. Sin embargo, hay un modelo alternativo en el que la conexión recurrente proviene de la capa de salida. En este caso, las activaciones netas de la capa de salida en el instante de tiempo anterior, \mathbf{o}^{t-1} , pueden añadirse de una de las dos siguientes maneras:

- A la capa oculta en el instante de tiempo presente, \mathbf{h}^t (mostrado en la [Figura 15.7](#) como recurrencia de salida a oculta)
- A la capa de salida en el instante de tiempo presente, \mathbf{o}^t (mostrado en la [Figura 15.7](#) como recurrencia de salida a salida)

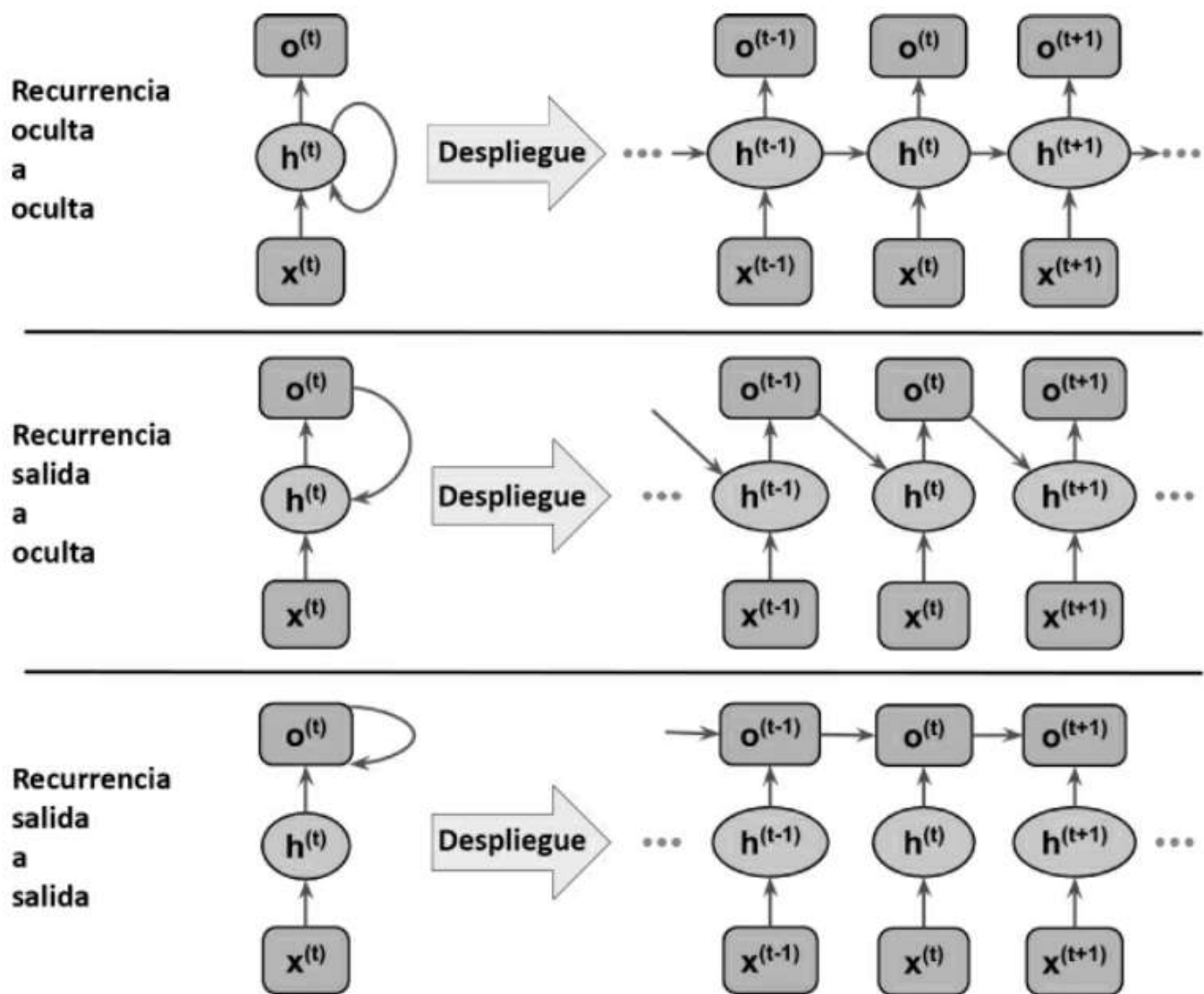


Figura 15.7 *Diferentes modelos de conexión recurrente.*

Como se muestra en la [Figura 15.7](#), las diferencias entre estas arquitecturas pueden verse claramente en las conexiones recurrentes. Siguiendo nuestra notación, los pesos asociados a la conexión recurrente se denotarán para la recurrencia oculta a oculta por W_{hh} , para la recurrencia de salida a oculta por W_{oh} , y para la recurrencia de salida a salida por W_{oo} . En algunos artículos de la literatura, los pesos asociados a las conexiones recurrentes también se denotan por W_{rec} .

Para ver cómo funciona lo anterior en la práctica, vamos a calcular manualmente el paso hacia delante para uno de estos tipos

recurrentes. Usando el módulo `torch.nn`, se puede definir una capa recurrente a través de RNN, que es similar a la recurrencia oculta a oculta. En el siguiente código, crearemos una capa recurrente de RNN y realizaremos un paso hacia delante en una secuencia de entrada de longitud 3 para calcular la salida. También calcularemos manualmente el paso hacia delante y compararemos los resultados con los de la RNN.

En primer lugar, vamos a crear la capa y a asignar los pesos y sesgos para nuestros cálculos manuales:

```
>>> import torch
>>> import torch.nn as nn
>>> torch.manual_seed(1)

>>> rnn_layer = nn.RNN(input_size=5, hidden_size=2,
...                      num_layers=1, batch_first=True)
>>> w_xh = rnn_layer.weight_ih_l0
>>> w_hh = rnn_layer.weight_hh_l0
>>> b_xh = rnn_layer.bias_ih_l0
>>> b_hh = rnn_layer.bias_hh_l0
>>> print('W_xh shape:', w_xh.shape)
>>> print('W_hh shape:', w_hh.shape)
>>> print('b_xh shape:', b_xh.shape)
>>> print('b_hh shape:', b_hh.shape)
W_xh shape: torch.Size([2, 5])
W_hh shape: torch.Size([2, 2])
b_xh shape: torch.Size([2])
b_hh shape: torch.Size([2])
```

La forma de entrada para esta capa es $((\text{batch_size}, \text{sequence_length}, 5))$, donde la primera dimensión es la dimensión del lote (ya que establecemos `batch_first=True`), la segunda dimensión corresponde a la secuencia, y la última dimensión corresponde a las características. Obsérvese que obtendremos una secuencia de salida que, para una secuencia de entrada de longitud 3, dará como resultado la secuencia de salida $\langle \mathbf{o}^{(0)}, \mathbf{o}^{(1)}, \mathbf{o}^{(2)} \rangle$. Además, la RNN utiliza una capa por defecto, y se puede configurar `num_layers` para apilar múltiples capas de RNN para formar una RNN apilada.

Ahora, llamaremos al paso hacia delante en `rnn_layer`, calcularemos manualmente las salidas en cada instante de tiempo y las compararemos:

```
>>> x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()
>>> ## output of the simple RNN:
>>> output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))
>>> ## manually computing the output:
>>> out_man = []
>>> for t in range(3):
...     xt = torch.reshape(x_seq[t], (1, 5))
...     print(f'Time step {t} =>')
...     print('    Input          : ', xt.numpy())
...
...     ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_hh
...     print('    Hidden          : ', ht.detach().numpy())
...
...     if t > 0:
...         prev_h = out_man[t-1]
...     else:
...         prev_h = torch.zeros((ht.shape))
```



```

...     ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) \
...         + b_hh
...     ot = torch.tanh(ot)
...     out_man.append(ot)
...     print('    Output (manual) :', ot.detach().numpy())
...     print('    RNN output      :', output[:, t].detach().numpy())
...     print()
Time step 0 =>
    Input          : [[1.  1.  1.  1.  1.]]
    Hidden          : [[-0.4701929  0.5863904]]
    Output (manual) : [[-0.3519801  0.52525216]]
    RNN output      : [[-0.3519801  0.52525216]]

Time step 1 =>
    Input          : [[2.  2.  2.  2.  2.]]
    Hidden          : [[-0.88883156  1.2364397 ]]
    Output (manual) : [[-0.68424344  0.76074266]]
    RNN output      : [[-0.68424344  0.76074266]]

Time step 2 =>
    Input          : [[3.  3.  3.  3.  3.]]
    Hidden          : [[-1.3074701  1.886489 ]]
    Output (manual) : [[-0.8649416  0.90466356]]
    RNN output      : [[-0.8649416  0.90466356]]

```

En nuestro cálculo directo manual, utilizamos la función de activación tangente hiperbólica (tanh), ya que también se utiliza en RNN (la activación por defecto). Como puede ver en los resultados impresos, las salidas de los cálculos manuales hacia delante coinciden exactamente con la salida de la capa RNN en cada instante de tiempo. Esperamos que esta tarea práctica le haya ilustrado sobre los misterios de las redes recurrentes.

Retos del aprendizaje de las interacciones de largo alcance

BPTT, que ya se ha mencionado brevemente, introduce algunos retos nuevos. Debido al factor multiplicativo, $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$, en el cálculo de los gradientes de la función de pérdida, surgen los denominados «problemas de gradiente evanescente y explosivo».

Estos problemas se explican en los ejemplos de la [Figura 15.8](#), que, para simplificar, muestra una RNN con una sola unidad oculta:

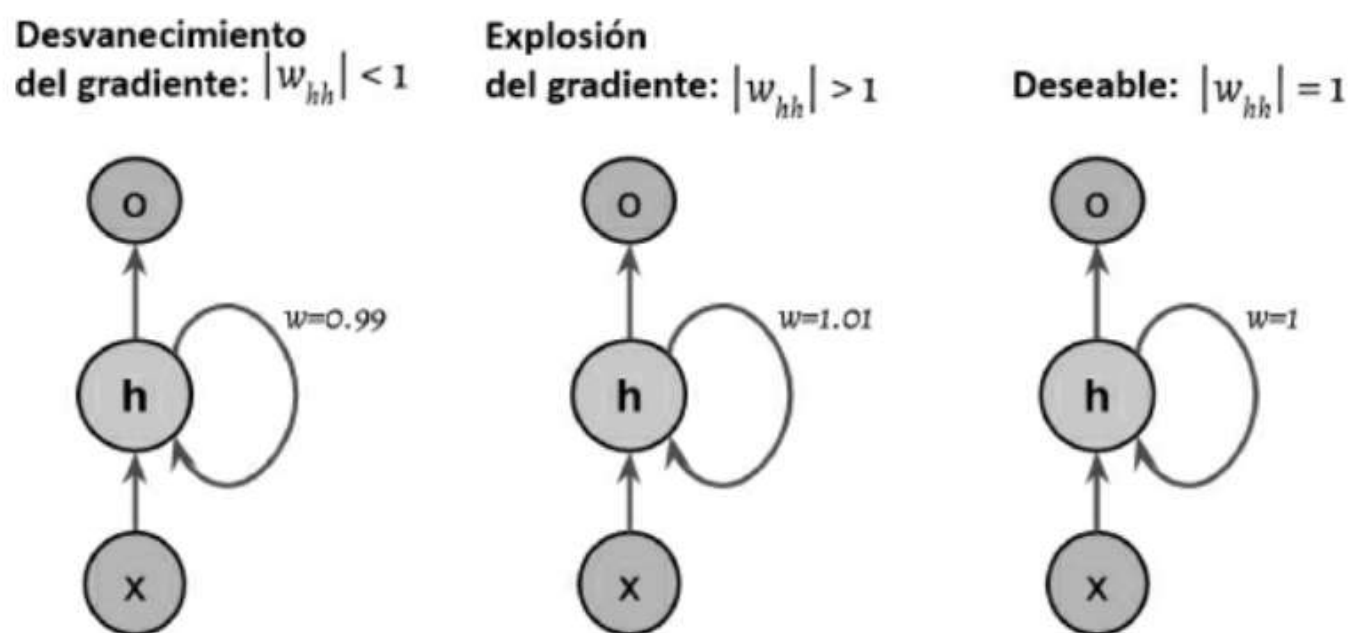


Figura 15.8 Problemas para calcular los gradientes de la función de pérdida.

Básicamente, $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$ tiene $t - k$ multiplicaciones; por lo tanto, al multiplicar el peso, w , por sí mismo $t - k$ veces se obtiene un factor, w^{t-k} . Como resultado, si $|w| < 1$, este factor se vuelve muy pequeño cuando $t - k$ es grande. Por otro lado, si el peso de la arista recurrente es $|w| > 1$, entonces w^{t-k} se vuelve muy grande cuando $t - k$ es grande.

Nótese que un $t - k$ grande se refiere a dependencias de largo alcance. Podemos ver que se puede alcanzar una solución ingenua para evitar que los gradientes se desvanezcan o exploten asegurando que $|w| = 1$. Si está interesado y quiere investigar esto con más detalle, lea *On the difficulty of training recurrent neural networks*, de R. Pascanu, T. Mikolov e Y. Bengio, 2012 (<https://arxiv.org/pdf/1211.5063.pdf>).

En la práctica, existen al menos tres soluciones a este problema:

- Recorte de gradientes
- Retropropagación truncada en el tiempo (Truncated Backpropagation Through Time, TBPTT)
- LSTM

Con el recorte de gradientes, especificamos un valor de corte o umbral para los gradientes, y asignamos este valor de corte a los valores de gradientes que superan este valor. Por el contrario, TBPTT simplemente limita el número de instantes de tiempo que la señal puede retropropagar después de cada paso hacia delante. Por ejemplo, aunque la secuencia tenga 100 elementos o pasos, solo podemos retropropagar los 20 pasos de tiempo más recientes.

Aunque tanto el recorte de gradiente como TBPTT pueden resolver el problema del gradiente explosivo, el truncamiento limita el número de pasos en los que el gradiente puede retroceder eficazmente y actualizar adecuadamente los pesos. Por otro lado, LSTM, diseñada en 1997 por Sepp Hochreiter y Jürgen Schmidhuber, ha tenido más éxito en los problemas de gradiente de fuga y de explosión, y modela las dependencias de largo alcance mediante el uso de celdas de memoria. Analicemos LSTM con más detalle.

Células de memoria a corto y largo plazo

Como se ha dicho anteriormente, las LSTM se introdujeron por primera vez para superar el problema del gradiente evanescente («Long Short-Term Memory», de S. Hochreiter y J. Schmidhuber, *Neural Computation*, 9(8): 1735-1780, 1997). El bloque de construcción de las LSTM es una célula de memoria, que esencialmente representa o reemplaza la capa oculta de las RNN estándar.

En cada celda de memoria, hay una arista recurrente que tiene el peso deseable, $w = 1$, como ya hemos comentado, para superar los problemas de gradiente de fuga y explosión. Los valores asociados a esta arista recurrente se denominan colectivamente «estado de la célula». La estructura desplegada de una célula LSTM moderna se muestra en la [Figura 15.9](#):

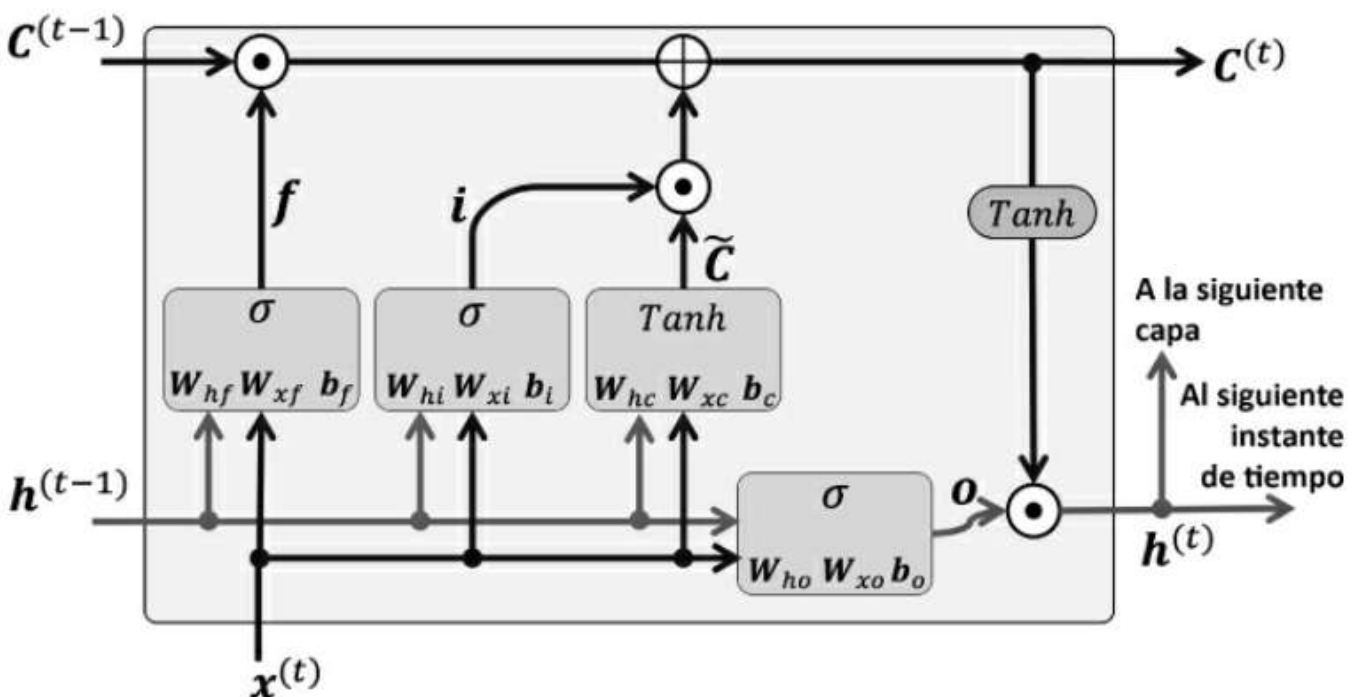


Figura 15.9 Estructura de una célula LSTM.

Obsérvese que el estado de la célula del instante de tiempo anterior, $\mathbf{C}^{(t-1)}$, se modifica para obtener el estado de la célula en el paso de tiempo presente, $\mathbf{C}^{(t)}$, sin que se haya multiplicado directamente por ningún factor de peso. El flujo de información en esta célula de memoria lo controlan varias unidades de cálculo (a menudo llamadas «puertas»), que se describirán aquí. En la figura, \odot se refiere al producto término a término (multiplicación término a término) y \oplus significa suma término a término (adición término a término). Además, $\mathbf{x}(t)$ se refiere a los datos de entrada en el tiempo t , y $\mathbf{h}^{(t-1)}$ indica las unidades ocultas en el tiempo $t - 1$. Se señalan cuatro cajas con una función de activación, ya sea la función sigmoide (σ) o \tanh , y un conjunto de pesos; estas cajas aplican una combinación lineal realizando multiplicaciones matriciales-vectoriales en sus entradas (que son $\mathbf{h}^{(t-1)}$ y $\mathbf{x}^{(t)}$). Estas unidades de cómputo con funciones de activación sigmoideas, cuyas unidades de salida pasan por \odot , se denominan «puertas».

En una célula LSTM, hay tres tipos diferentes de puertas, que se conocen como la puerta del olvido, la puerta de entrada y la puerta de salida.

La puerta del olvido (f_t) permite a la célula de memoria restablecer el estado de la célula sin crecer indefinidamente. De hecho, la puerta del olvido decide qué información se deja pasar y qué información se suprime. Ahora, f_t se calcula de la siguiente manera:

$$f_t = \sigma(\mathbf{W}_{xf}\mathbf{x}^{(t)} + \mathbf{W}_{hf}\mathbf{h}^{(t-1)} + \mathbf{b}_f)$$

Obsérvese que la puerta del olvido no formaba parte de la célula LSTM original; se añadió unos años después para mejorar el modelo original («Learning to Forget: Continual Prediction with LSTM», de F. Gers,

J. Schmidhuber, y F. Cummins, *Neural Computation* 12, 2451-2471, 2000).

La puerta de entrada (i_t) y el valor candidato (\tilde{c}_t) se encargan de actualizar el estado de la célula. Se calculan como sigue:

$$\begin{aligned}i_t &= \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \\ \tilde{c}_t &= \tanh(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c)\end{aligned}$$

El estado de la célula en el momento t se calcula de la siguiente manera:

$$c^{(t)} = (c^{(t-1)} \odot f_t) \oplus (i_t \odot \tilde{c}_t)$$

La puerta de salida (o_t) decide cómo actualizar los valores de las unidades ocultas:


$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o)$$

Dado lo anterior, las unidades ocultas en el instante de tiempo presente se calculan como sigue:

$$h^{(t)} = o_t \odot \tanh(c^{(t)})$$

La estructura de una célula LSTM y sus cálculos subyacentes pueden parecer muy complejos y difíciles de implementar. Sin embargo, la buena noticia es que PyTorch ya lo ha implementado todo en funciones envoltorio (wrapper) optimizadas; y eso nos permite definir nuestras celdas LSTM de forma fácil y eficiente. Más adelante en este capítulo aplicaremos las RNN y las LSTM a conjuntos de datos del mundo real.

Otros modelos RNN avanzados



Las LSTM ofrecen un enfoque básico para modelar las dependencias de largo alcance en las secuencias. Sin embargo, es importante señalar que hay muchas variaciones de LSTM descritas en la literatura («An Empirical Exploration of Recurrent Network Architectures», de Rafal Jozefowicz, Wojciech Zaremba, e Ilya Sutskever, *Proceedings of ICML*, 2342-2350, 2015). También cabe destacar un enfoque más reciente, la unidad recurrente cerrada (Gated Recurrent Unit, GRU), que se propuso en 2014. Las GRU tienen una arquitectura más sencilla que las LSTM; por lo tanto, son computacionalmente más eficientes, mientras que su rendimiento en algunas tareas, como el modelado de música polifónica, es comparable al de las LSTM. Si está interesado en saber más sobre estas arquitecturas modernas de RNN, consulte el artículo «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling», de Junyoung Chung y otros, 2014 (<https://arxiv.org/pdf/1412.3555v1.pdf>).

Implementación de RNN para el modelado de secuencias en PyTorch

Ahora que hemos tratado la teoría que subyace tras las RNN, estamos listos para pasar a la parte más práctica de este capítulo: implementar las RNN en PyTorch. Durante el resto de este capítulo, aplicaremos las RNN a dos tareas problemáticas que son frecuentes:

1. Análisis de opiniones
2. Modelado de idiomas