APRENDIZAJE PROFUNDO

Iker Caballero Bragagnini

Tabla de contenido

Las redes neuronales artificiales: funcionamiento básico

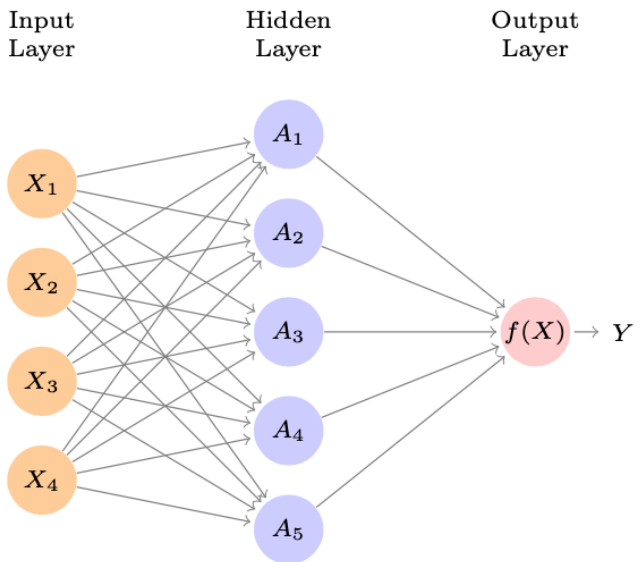
- Las redes neuronales profundas retroalimentadas o *feedforward neural networks* (también llamadas perceptrones multicapa o MLPs) son el elemento esencial de los modelos de aprendizaje profundo. El objetivo de una red neuronal retroalimentada es aproximar alguna función f^*, y esta define $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ y aprende el valor de los parámetros $\boldsymbol{\theta}$ que resulte en la mejor aproximación a f^*

    - Estos modelos se denominan de retroalimentación porque la información fluye a través de la función siendo evaluada desde $\boldsymbol{x}$, a través de las computaciones intermedias usadas para definir f, y finalmente por el resultado $\boldsymbol{y}$

        - No hay conexiones de *feedback* en la que los resultados del modelo se alimenten otra vez a la red. Cuando las redes neuronales retroalimentadas se extienden para incluir conexiones de *feedback*, se llaman redes neuronales recurrentes o *recurrent neural networks*

        - Estas redes son de importancia extrema para los practicantes de aprendizaje automático, ya que forman las bases de las aplicaciones comerciales. Son una piedra conceptual en el camino a las redes neuronales recurrentes, y potencian muchas aplicaciones de lenguaje natural

    - Las redes neuronales se denominan redes porque normalmente se representan componiendo juntamente muchas funciones diferentes. El modelo se asocia con un grafo direccionado acíclico describiendo cómo las funciones se componen



- Si se tiene una secuencia de funciones $f^{(1)}, \dots, f^{(n)}$ conectadas por una cadena para formar $f(\boldsymbol{x}) = f^{(n)}\left(f^{(n-1)}\left(\dots f^{(1)}(\boldsymbol{x})\right)\right)$. Estas estructuras son las más comunes para redes neuronales

- En este caso, $f^{(1)}$ se conoce como la primera capa de la red, $f^{(2)}$ como la segunda capa y así. La longitud total de la cadena da la profundidad del modelo

- Los datos de entrenamiento proporcionan ejemplos aproximados y con ruido de $f^*(\boldsymbol{x})$ evaluado en diferentes puntos de entrenamiento y cada ejemplo de $\boldsymbol{x}$ viene acompañado de una etiqueta $y \approx f^*(\boldsymbol{x})$

  - Los ejemplos especifican directamente lo que la capa de resultados debe de hacer en cada punto $\boldsymbol{x}$ (debe producir un valor cercano a y)

  - El comportamiento de las otras capas no se especifica directamente en los datos de entrenamiento, por lo que el algoritmo de aprendizaje debe decidir cómo usar las capas para producir el resultado deseado, pero los datos de entrenamiento no dicen lo que cada capa individual debería hacer

  - El algoritmo de entrenamiento decide como cómo usar las capas para implementar la mejor aproximación de f^*. Cómo los datos de entrenamiento no muestran el resultado deseado para cada una de estas capas, estas capas se conocen como capas escondidas o *hidden layers*

- Cada capa escondida de la red normalmente es vectorial, y la dimensionalidad de estas capas determina la anchura o *width* del modelo. Cada elemento del vector se puede interpretar de manera análoga como una neurona

  - La capa no representa una sola función de vector a vector, sino que se puede interpretar como una un conjunto de unidades que actúan en paralelo, en donde cada una representa una función de vector a escalar

  - Cada unidad se parece a una neurona en el sentido de que recibe un insumo de otras unidades y calcula su propio valor de activación. La idea de usar muchas capas de representaciones de valor vectorial se extrae de neurociencia

  - La elección de funciones $f^{(i)}(\boldsymbol{x})$ usada para calcular estas representaciones se guía por observaciones neurocientíficas sobre las funciones que las neuronas biológicas. No obstante, la

investigación moderna en redes neuronales se guía de manera más matemática y el objetivo no es modelar el cerebro perfectamente

- Se pueden entender las redes retroalimentadas como máquinas que aproximan funciones que se han diseñado para generalización estadística

- Una manera de entender las redes neuronales retroalimentadas es comenzar con modelos lineales y entender sus limitaciones

  - Los modelos lineales se pueden ajustar eficientemente, gracias a formas cerradas o a algoritmos de optimización convexa. No obstante, la capacidad del modelo se limita a funciones lineales, de modo que el modelo no puede entender las interacciones no lineales entre dos variables de insumo

    - Para extender los modelos lineales a representar funciones no lineales de $\boldsymbol{x}$, se pueden aplicar modelos lineales no a $\boldsymbol{x}$ directamente, pero a los insumos transformados $\phi(\boldsymbol{x})$, donde ϕ es una transformación no lineal

    - Equivalentemente, es posible aplicar el truco del *kernel* para obtener algoritmos de aprendizaje no lineales basados en implícitamente aplicar el mapeado ϕ. Por lo tanto, se puede interpretar a ϕ como un mapeado que proporciona un conjunto de características describiendo $\boldsymbol{x}$, o proporcionando una nueva representación para $\boldsymbol{x}$

  - La pregunta está en cómo escoger el mapeado ϕ adecuado, y uno normalmente tiene tres opciones:

    - Una opción es usar una ϕ común (basada en el principio de suavidad local), tal como una ϕ de dimensión infinita o implícitamente usar una máquina de *kernel* basada en un *kernel* RBF. Si $\phi(\boldsymbol{x})$ es de una dimensión lo suficientemente, siempre se tiene capacidad suficiente para ajustar el conjunto de entrenamiento, pero la generalización al conjunto de comprobación sigue siendo pobre

    - Otra opción es construir manualmente ϕ, y este era el enfoque dominante antes del aprendizaje. Este enfoque requiere de tiempo, en donde los practicantes se especializan en diferentes dominios y con poca transferencia entre ellos

    - Finalmente, la estrategia del aprendizaje profundo es aprender ϕ. En este enfoque, uno quiere modelar $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) =$

$\phi(\boldsymbol{x};\boldsymbol{\theta})^T\boldsymbol{w}$, de modo que ahora se tienen parámetros $\boldsymbol{\theta}$ que se usa para aprender θ de una gran clase de funciones, y parámetros $\boldsymbol{w}$ que mapea de $\phi(\boldsymbol{x})$ al resultado deseado. Este es un ejemplo de red neuronal de retroalimentación, con ϕ definiendo una capa neuronal escondida

- Este ejemplo es el único de los tres que deja de lado la convexidad del problema de entrenamiento (por ejemplo, con máquinas de vectores de soporte, el problema es convexo), pero los beneficios son mayores a las pérdidas. En este enfoque, se parameteriza la representación cómo $\phi(\boldsymbol{x};\boldsymbol{\theta})$ y se usa un algoritmo de optimización para encontrar la $\boldsymbol{\theta}$ correspondiente a una buena representación

- Este enfoque captura el beneficio de la generalidad del primer enfoque (usando una gran familia de $\phi(\boldsymbol{x};\boldsymbol{\theta})$) y el del segundo enfoque, dado que se diseñan familias que se espera qur funcionen bien. La ventaja es que el diseñador humano solo necesita encontrar una familia de funciones generales más que una sola función

- El principio general de mejorar modelos para aprender características extendiéndose más allá de las redes de retroalimentación. Es un tema recurrente en el aprendizaje profundo que plica a muchos tipos de modelos

  - Las redes neuronales retroalimentadas son la aplicación de este principio al aprendizaje determinístico de mapeados de $\boldsymbol{x}$ a $\boldsymbol{y}$ que carecen de conexiones de *feedback*. Otros modelos aplican estos principios a mapeados estocásticos, aprendiendo funciones con *feedback*, y las distribuciones de probabilidad sobre un solo vector

  - El entrenamiento de una red neuronal requiere hacer muchas decisiones de diseño para un modelo lineal: escoger el optimizador, la función de coste y la forma de las unidades de resultados. Además, también se tienen que escoger las funciones de activación y aprender el algoritmo de retropropagación o *back-propagation*

- Con tal de asentar las ideas sobre las redes neuronales retroalimentadas, se analiza un ejemplo de una red totalmente conectada para aprender la función XOR

  - La función XOR es una operación sobre dos valores binarios x_1 y x_2. Cuando exactamente uno de los valores binarios es 1, entonces la función es 1, pero de otro modo devuelve 0

- El modelo de redes neuronales retroalimentadas proporcionaría una función $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ y el algoritmo de aprendizaje adaptaría los parámetros $\boldsymbol{\theta}$ para hacer f lo más similar posible a f^*

- En este ejemplo uno no se debe preocupar de la generalización estadística: se quiere que la red aprenda $\mathcal{X} = \{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$. Se tiene que entrenar la red neuronal sobre estos cuatro puntos, de modo que se tiene que ajustar el conjunto de entrenamiento

- Este problema se puede tratar como un problema de regresión y usar una función de pérdida de error cuadrática media o MSE (con tal de simplificar la matemática, aunque no es la función más apropiada para datos binarios)

  - Evaluado en todo el conjunto de entrenamiento, la función MSE es la siguiente:

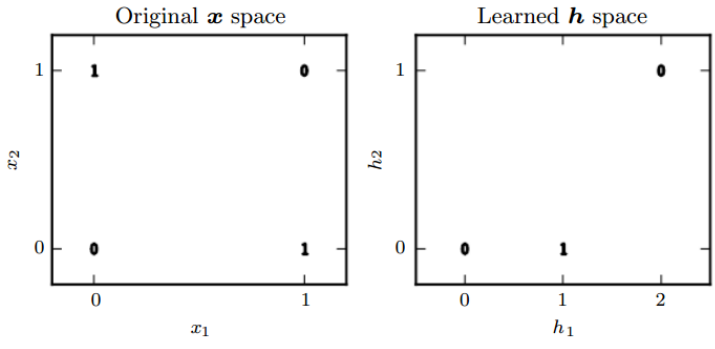
$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathcal{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}) \right)^2$$

  - Ahora se tiene escoger la forma del modelo, $f(\boldsymbol{x}; \boldsymbol{\theta})$. Suponiendo que se escoge un modelo lineal, con $\boldsymbol{\theta}$ consistiendo de $\boldsymbol{w}$ y b, el modelo se define de la siguiente:

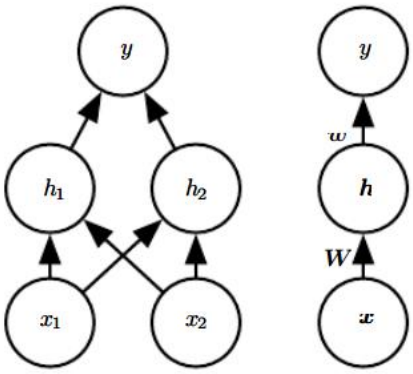
$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

  - Se puede minimizar $J(\boldsymbol{\theta})$ es una forma cerrada con respecto a $\boldsymbol{w}$ y a b usando las ecuaciones normales. Resolviéndolas se obtiene $\boldsymbol{w} = \boldsymbol{0}$ y $b = 1/2$, de modo que el modelo lineal predice 0.5 siempre

  - Esto ocurre porque el modelo lineal no es capaz de representar la función XOR. Una manera de resolver este problema es usar un modelo que aprenda de un espacio de variables diferentes en la que el modelo lineal sea capaz de representar la solución



- Específicamente, se introduce una red neuronal retroalimentada simple con una capa escondida y dos unidades escondidas. Esta red tiene un vector de unidades escondidas $\boldsymbol{h}$ que se han calculado por una función $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$, y estos valores pasan una segunda capa y, finalmente, a la capa de resultados



  - La capa de resultados es un modelo de regresión lineal, pero ahora se aplica a $\boldsymbol{h}$ más que a $\boldsymbol{x}$. La red ahora contiene dos funciones encadenadas juntas, siendo el modelo completo el siguiente:

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) \quad \& \quad y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$$

$$\Rightarrow \quad f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)$$

  - La función $f^{(1)}$ no puede ser lineal porque, si no, el modelo seguiría siendo lineal en su insumo, por lo que no serviría para una representación no lineal. Por lo tanto, se tiene que utilizar una función no lineal que describe las características

$$f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c} \quad \& \quad f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b) = \boldsymbol{h}^T\boldsymbol{w} + b$$

$$\Rightarrow \quad f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = (\boldsymbol{x}^T\boldsymbol{W} + \boldsymbol{c}^T)\boldsymbol{w} + b$$

$$\Rightarrow \quad f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{x}^T\boldsymbol{W}\boldsymbol{w} + (\boldsymbol{c}^T\boldsymbol{w} + b) = \boldsymbol{x}^T\boldsymbol{w}' + b'$$

- La mayoría de redes neuronales lo hacen usando una transformación afín controlada por los parámetros aprendidos, seguida de una función lineal no fija denominada función de activación

  - Se define $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$, donde $\boldsymbol{W}$ proporciona las ponderaciones de una transformación lineal y $\boldsymbol{c}$ los sesgos. La función g es normalmente una función que se aplica elemento por elemento, de modo que $h_i = g\left(\boldsymbol{x}^T\boldsymbol{W}_{:,i} + c_i\right)$

- Usando la unidad lineal rectificada o ReLU, definida por la función $g(z) = \max\{z, 0\}$, se puede especificar la red completamente:

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

- A partir de esto es posible especificar una solución para el problema de la función XOR

  - Primero, se fijan los valores de los siguientes elementos necesarios y de los datos de entrenamiento:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{C} = \begin{bmatrix} 0 & -1 \\ 0 & -1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad \boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{b} = \boldsymbol{0}$$

  - A partir de ahí, se sigue la composición de funciones indicada por la red neuronal para obtener el siguiente resultado:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \Rightarrow \boldsymbol{XW} + \boldsymbol{C} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\Rightarrow \boldsymbol{h} = \max\{\boldsymbol{XW} + \boldsymbol{C}, \boldsymbol{0}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \Rightarrow \boldsymbol{y} = \boldsymbol{hw} + \boldsymbol{b} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

  - En este problema se ha especificado una solución y se ha demostrado que se puede obtener sin error, pero en una situación real hay miles de parámetros y ejemplos de entrenamiento

- En vez de esto, un algoritmo de optimización basado en gradientes puede encontrar parámetros que producen poco error

  - En este caso, la solución era un mínimo global de la función de pérdida, de modo que convergería a ese punto. Hay otras soluciones equivalentes al problema en donde el descenso del gradiente puede llegar

  - El punto de convergencia del método dependerá de los valores iniciales de los parámetros

Las redes neuronales artificiales: aprendizaje basado en el descenso de gradiente

- Diseñar y entrenar una red neuronal no es muy diferente a entrenar otro modelo de aprendizaje automático con el método del gradiente, por lo que se tiene que especificar un procedimiento de optimización, una función de coste y una familia de modelos

    - La mayor diferencia entre los modelos lineales y las redes neuronales es que la no linealidad de una red causa que la mayoría de funciones de pérdida interesantes no sean convexas

        - Esto significa que las redes neuronales normalmente se entrenan utilizando optimizadores basados en gradientes que conducen la función de coste a un valor muy pequeño, más que con optimizadores de ecuaciones lineales usados para entrenar modelos de regresión lineal o los algoritmos de optimización convexa con convergencia global para regresiones logísticas y máquinas de vectores de soporte

        - La optimización convexa converge empezando de un conjunto cualquiera de parámetros iniciales (en teoría, pero en la práctica es muy robusto, pero puede haber problemas numéricos). El descenso de gradiente estocástico aplicado a funciones de pérdida no convexas no garantiza convergencia global y es sensible a los valores iniciales

        - Las redes neuronales retroalimentadas, es importante inicializar todas las ponderaciones a valores aleatoriamente pequeños, mientras que los sesgos se pueden inicializar a cero

        - Uno puede aplicar estos algoritmos a modelos de regresión lineal y máquinas de vectores de soporte (el caso cuando el conjunto de entrenamiento es muy grande). Aplicar esto en una red neuronal es un poco más complicado, pero igualmente se basa en escoger una función de coste y una representación del resultado del modelo

    - Un aspecto importante del diseño de una red neuronal profunda es la selección de una función de coste. Afortunadamente, las funciones de coste para redes neuronales son más o menos las mismas igual que otros modelos parámetricos, tales como los modelos lineales

        - En la mayoría de casos, el modelo parámetrico define una distribución $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ y simplemente se usa el principio de máxima verosimilitud. Esto significa que se usa la entropía

cruzada entre los datos de entrenamiento y los modelos de predicción como la función de coste

- A veces, se toma un enfoque más simple, en donde más que predecir una distribución de probabilidad completa sobre $\boldsymbol{y}$, uno meramente predice un estadístico de $\boldsymbol{y}$ condicionado a $\boldsymbol{x}$. Las funciones de pérdida especializadas permiten entrenar un predictor de estas estimaciones

- La mayoría de redes neuronales se entrenan utilizando máxima verosimilitud, de modo que la función de coste simplemente es la verosimilitud logarítmica negativa, equivalentemente descrita como la entropía cruzada entre los datos de entrenamiento y la distribución del modelo

$$J(\boldsymbol{\theta}) = -E_{\boldsymbol{x},\boldsymbol{y} \sim \hat{p}_{data}}\left[\log\left(p_{model}(\boldsymbol{y}|\boldsymbol{x})\right)\right]$$

  - La forma específica de la función de coste cambia de modelo a modelo, dependiendo de la forma específica de $\log p_{model}$. La expansión de la ecuación da algunos términos que no dependen en los parámetros del modelo y pueden ser descartados

  - A través de la suposición de que $p_{model}(\boldsymbol{y}|\boldsymbol{x}) \sim N(\boldsymbol{y}|f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$ entonces se puede recobrar el coste medio cuadrático multiplicado por 1/2, y el término no depende $\boldsymbol{\theta}$:

$$\sum_{i=1}^{m} \log\left[p\left(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right)\right] = -m\log(\sigma) - \frac{m}{2}\log(2\pi) - \sum_{i=1}^{m}\frac{\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2}{2\sigma^2}$$

$$\Rightarrow \sum_{i=1}^{m} \log\left[p\left(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right)\right] = const - \frac{1}{const}\sum_{i=1}^{m}\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2$$

$$\Rightarrow J(\boldsymbol{\theta}) = -E_{\boldsymbol{x},\boldsymbol{y} \sim \hat{p}_{data}}\left[\log\left(p_{model}(\boldsymbol{y}|\boldsymbol{x})\right)\right]$$

$$\Rightarrow J(\boldsymbol{\theta}) = \frac{1}{2}E_{\boldsymbol{x},\boldsymbol{y} \sim \hat{p}_{data}}(\|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2) + const.$$

$$\Rightarrow J(\boldsymbol{\theta}) = \frac{1}{2}\lim_{m\to\infty}\left(\frac{1}{m}\sum_{i=1}^{m}\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2\right) + const.$$

$$\Rightarrow J(\boldsymbol{\theta}) = \frac{1}{2}\lim_{m\to\infty}(MSE_{train}) + const$$

  - Como se puede ver, el punto que minimiza el MSE_{train} es el mismo que el que minimiza $J(\boldsymbol{\theta})$ pero cambiando la escala, de

modo que se puede usar el MSE_{train} como criterio de estimación por máxima verosimilitud

- Una ventaja de utilizar este enfoque es derivar la función de coste por máxima verosimilitud es que no hace falta diseñar las funciones de coste para cada modelo: especificar un modelo $p_{model}(\boldsymbol{y}|\boldsymbol{x})$ automáticamente determina una función de coste $\log[p(\boldsymbol{y}|\boldsymbol{x})]$

- Un tema recurrente en el diseño de redes neuronales es que el gradiente de la función de coste debe de ser lo suficientemente grande y predecible para servir como un guía del algoritmo de aprendizaje

  - Las funciones que se saturan (que se vuelven muy planas) dificultan el objetivo porque hacen que el gradiente se vuelva muy pequeño en esa zona. En muchos casos esto ocurre porque se saturan las funciones de activación usadas para producir el resultado de las unidades escondidas o de las unidades de resultados

  - La verosimilitud logarítmica negativa ayuda a evitar este problema para muchos modelos. Muchas unidades involucran una función exponencial que se puede saturar cuando su argumento es muy negativo, pero cuando se usa el logaritmo, se deshace la exponencial para algunas unidades

- Una propiedad inusual del coste de entropía cruzada usada para realizar estimación de máxima verosimilitud es que normalmente no tiene un valor mínimo cuando se aplica a modelos usados en la práctica

  - Para variables de resultado discretas, la mayoría de modelos se parametrizan de tal manera que no se puede representar la probabilidad de 0 o 1, pero pueden estar arbitrariamente cerca, como en un modelo logístico. Para variables de resultado continuas, si el modelo puede controlar la densidad de la distribución del resultado (por ejemplo, aprendiendo la varianza de una distribución normal), entonces es posible asignar una gran densidad a los resultados del conjunto de entrenamiento (resultando en que la entropía cruzada llegué a menos infinito)

  - Las técnicas de regularización proporcionan diferentes maneras de modificar el problema de aprendizaje de modo que el modelo no pueda obtener recompensa ilimitada de esta manera

- En vez de aprender toda una distribución de probabilidad $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ uno normalmente quiere aprender un estadístico condicional de $\boldsymbol{y}$ sobre $\boldsymbol{x}$.

Por ejemplo, se puede tener un predictor $f(\boldsymbol{x};\boldsymbol{\theta})$ que quiere predecir la media de $\boldsymbol{y}$

- Si se utiliza una red neuronal lo suficientemente potente, se puede interpretar una red neuronal como un mecanismo que representa cualquier función f de una gran clase de funciones, esta clase estando limitada solo por características tales como continuidad y acotación más que tener una forma paramétrica específica. Desde este punto de vista, se puede entender la función de coste como un funcional (mapeado de funciones a números reales) más que una función

- Por lo tanto, se puede entender el aprendizaje como escoger una función más que escoger los parámetros de una función. Es posible diseñar la función de coste que tenga su mínimo en alguna función específica deseada (por ejemplo, en una función que mapee $\boldsymbol{x}$ al valor esperado de $\boldsymbol{y}$ sobre $\boldsymbol{x}$)

- Resolver un problema de optimización con respecto a una función requiere cálculo de variaciones. Los dos resultados más importantes que se obtienen usando esta herramienta son los siguientes:

  - Resolver el siguiente problema de optimización permite obtener un óptimo que predice la media condicional (siempre que pertenezca a la clase de las funciones sobre las que se optimiza):

$$f^* = \arg\min_f E_{\boldsymbol{x},\boldsymbol{y}\sim p_{data}} (\|\boldsymbol{y} - f(\boldsymbol{x})\|^2)$$

$$\Rightarrow \; f^* = E_{\boldsymbol{y}\sim p_{data}(\boldsymbol{y}|\boldsymbol{x})}(\boldsymbol{y}) = E(\boldsymbol{y}|\boldsymbol{x})$$

  - Este resultado muestra que, si se pudiera entrenar muestras infinitamente de la distribución generadora de datos, minimizar el error medio cuadrático da una función que predice la media condicional de $\boldsymbol{y}$ sobre $\boldsymbol{x}$

  - Diferentes funciones de costes dan diferentes estadísticos. Resolver el siguiente problema de optimización (la función de coste se suele denominar error medio absoluto), por ejemplo, permite obtener una función que predice el valor mediano de $\boldsymbol{y}$ para cualquier $\boldsymbol{x}$ (siempre que pertenezca a la clase de las funciones sobre las que se optimiza):

$$f^* = \arg\min_f E_{\boldsymbol{x},\boldsymbol{y}\sim p_{data}} (\|\boldsymbol{y} - f(\boldsymbol{x})\|_1)$$

  - El error medio cuadrático y el error medio absoluto normalmente lleva a resultados pobres cuando se usa optimización basada en

gradientes. Algunas unidades de resultados que se saturan producen gradientes muy pequeños cuando se combinan con estas funciones de coste, lo cual es una razón por la cual la función de entropía cruzada es más popular que las últimas, aunque no sea necesario estimar toda una distribución $p(\boldsymbol{y}|\boldsymbol{x})$

- La elección de la función de coste está muy relacionada con la elección de la unidad de resultados. La mayoría de tiempo se suele utilizar la entropía cruzada entre la distribución de los datos y la distribución del modelo, pero la elección de la unidad de resultados determina la forma de la función de la entropía cruzada

  - Cualquier tipo de unidad de red neuronal que se puede usar en la capa de resultados se puede utilizar en las capas escondidas

  - Para desarrollar la teoría, se supone que la red retroalimentada proporciona un conjunto de características escondidas definidas por $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$. El rol de la capa de resultados es proporcionar alguna transformación adicional a partir de las características para completar la tarea de la red

- Un tipo de unidad de resultados puede ser una basada en transformaciones afines lineales, llamadas unidades lineales. Dadas las características $\boldsymbol{h}$, una capa de unidades de resultados lineales produce un vector resultante $\hat{\boldsymbol{y}} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$

  - Las unidades lineales normalmente se utilizan cuando se quiere producir la media de una distribución normal condicional, donde maximizar la verosimilitud logarítmica es equivalente a minimizar el error cuadrático medio:

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

    - El marco de la máxima verosimilitud hace que sea directo aprender la covarianza de la distribución normal, o hacer que la covarianza sea una función de un insumo. No obstante, tiene que ser una matriz definida positiva para cualquier insumo, lo cual es difícil de cumplir con una capa lineal, por lo que se suelen utilizar otras unidades de resultados para parametrizar la covarianza

    - Debido a que las unidades lineales no se saturan, no ponen muchas dificultades a la optimización basada en gradientes y se pueden usar en varios algoritmos de optimización

- Muchas tareas requieren predecir el valor de una variable binaria y, que pertenece al dominio de los problemas de clasificación de dos clases o binarios. El enfoque la máxima verosimilitud es definir una distribución de Bernoulli sobre y condicionada a $\boldsymbol{x}$

- Una distribución de Bernoulli solo se determina por un parámetro, de modo que la red neuronal solo necesita predecir solo $P(y = 1|\boldsymbol{x})$, pero debe de estar restringido a $[0,1]$

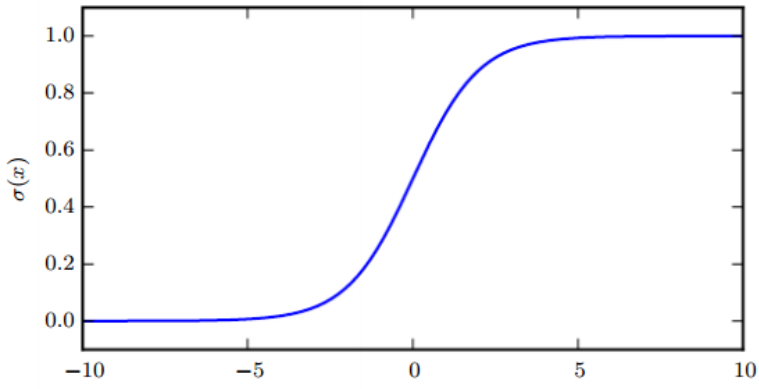
    - Satisfacer la restricción requiere un cuidado especial. Una manera podría ser usar una unidad lineal y se pone un límite para obtener una probabilidad válida

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

    - Esto definiría una distribución condicional válida, pero uno no sería capaz de entrenarla efectivamente con descenso del gradiente: en cualquier momento que $\boldsymbol{w}^T\boldsymbol{h} + b$ salga del intervalo unitario, el gradiente con respecto al parámetro sería $\mathbf{0}$. Un gradiente de $\mathbf{0}$ es problemático porque el algoritmo de aprendizaje deja de tener una guía de cómo mejorar los parámetros correspondientes

- En vez de esto, es mejor utilizar otro enfoque que garantice que haya un gradiente fuerte cuando el modelo está incorrecto. Este enfoque se basa en usar unidades de resultados sigmoides combinadas con máxima verosimilitud

    - Una unidad sigmoide se define por $\hat{y} = \sigma(\boldsymbol{w}^T\boldsymbol{h} + b)$ donde σ es la función logística sigmoide $\sigma(z) = 1/(1 + e^{-z})$. Se puede pensar que esta unidad tiene dos componentes: una capa lineal para calcular $z = \boldsymbol{w}^T\boldsymbol{h} + b$ y después se usa la función de activación logística sigmoide para convertir z en probabilidad



    - Esta distribución se satura para valores muy positivos o muy negativos de z, de modo que la función se vuelve más plana e insensible a pequeños cambios en los insumos

    - La sigmoide también se puede motivar por la construcción de una distribución de probabilidad no normalizada $\tilde{P}(y)$ que no suma a

1, de modo que para normalizarla se necesita dividir por una constante apropiada. Si se empieza por la suposición de que las probabilidades logarítmicas no normalizadas son lineales en y y z, de modo que se pueden exponenciar para obtener las probabilidades no normalizadas y finalmente normalizarlas para ver que se obtiene una distribución de Bernoulli controlada por una transformación sigmoide de z:

$$\log \tilde{P}(y) = yz \;\Rightarrow\; \tilde{P}(y) = e^{yz} \;\Rightarrow\; P(y) = \frac{e^{yz}}{\sum_{y'=0}^{1} e^{y'z}} = \frac{e^{yz}}{1+e^{z}}$$

$$\Rightarrow\; P(y) = \sigma\big((2y-1)z\big) = \frac{e^{(2y-1)z}}{1+e^{(2y-1)z}}$$

$$\Rightarrow P(y) = \sigma(z)^{y}[1-\sigma(z)]^{1-y} = \begin{cases} \dfrac{e^{z}}{1+e^{z}} & when\ y=1 \\ \dfrac{1}{1+e^{z}} & when\ y=0 \end{cases}$$

- Las distribuciones de probabilidad basadas en la exponenciación y normalización son comunes, y una variable z definiendo la distribución sobre variables binarias se denomina *logit*

- Algunas propiedades útiles de las funciones sigmoides son las siguientes:

$$\frac{d}{dz}\sigma(z) = \sigma(z)[1-\sigma(z)]$$

$$1-\sigma(z) = \sigma(-z)$$

$$for\ \forall z \in (0,1), \sigma^{-1}(z) = \log\left(\frac{z}{1-z}\right)$$

- El enfoque de predecir las probabilidades en el espacio logarítmico es natural de usar con el aprendizaje de máxima verosimilitud. Dado que la función de coste usada es $-\log[P(y|\boldsymbol{x})]$, el logaritmo deshace la exponenciación de la sigmoide (sin esto, la saturación de la sigmoide podría evitar los progresos en el aprendizaje del algoritmo)

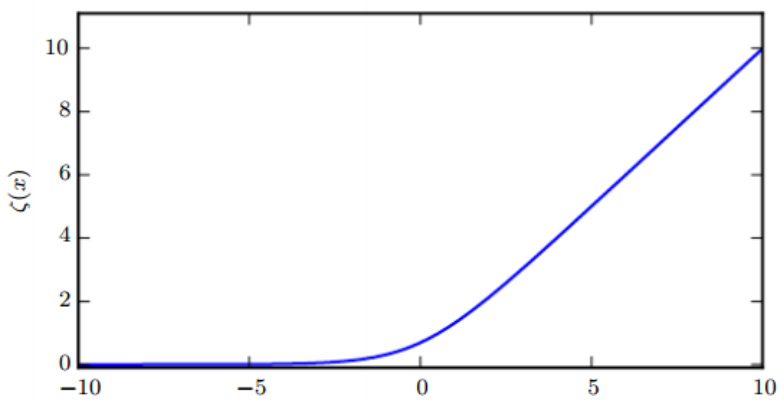
    - La función de pérdida para el aprendizaje por máxima verosimilitud de una Bernoulli parametrizada por una sigmoide es la siguiente:

$$J(\boldsymbol{\theta}) = -\log[P(y|\boldsymbol{x})] = -\log\big[\sigma\big((2y-1)z\big)\big]$$

$$As\ \log[\sigma(z)] = -\log(1+e^{-z}):$$

$$\Rightarrow J(\boldsymbol{\theta}) = \log\left(1 + e^{(1-2y)z}\right)$$

- Esta función se denomina *softplus*, la cual se suele utilizar para obtener el parámetro de varianza σ de una distribución normal porque su rango es de $(0, \infty)$. El nombre proviene de que es una versión suavizada de la función $z^+ = \max\{0, z\}$



- Reescribiendo la pérdida en términos de la función *softplus*, se puede ver que se satura cuando $(1 - 2y)z$ es muy negativa. La saturación, por tanto, ocurre solo cuando el modelo tiene la respuesta correcta ($y = 1$ y z muy positiva o $y = 0$ y z muy negativa)

- Cuando z tiene un signo contrario, entonces el argumento se puede simplificar a $|z|$: mientras $|z|$ se agranda cuando z tiene el signo erróneo, la función *softplus* asintóticamente devuelve $|z|$. Esta propiedad es muy útil porque significa que el aprendizaje basado en gradientes puede actuar para corregir fácilmente una z equivocada

- Cuando se usan otras funciones de pérdida, tales como el error medio cuadrático, la pérdida se puede saturar en cualquier momento en que $\sigma(z)$ se sature, de modo que el gradiente se puede minimizar mucho se esté o no en la respuesta correcta. Por ello, casi siempre se prefiere la máxima verosimilitud para entrenar unidades sigmoides

- Cuando se quiere representar la distribución de probabilidad sobre una variable discreta con n posibles valores, se puede utilizar la función *softmax* (versión suave de la función argmax). Esto se puede ver como una generalización de la función sigmoide que se usaba para representar la distribución de una variable binaria

- Estas funciones se utilizan mucho en la capa de resultados de un clasificador para representar la probabilidad de las n clases, aunque también se puede

    - En el caso de una variable binaria se quiere producir un solo número $\hat{y} = P(y = 1|\boldsymbol{x})$, pero como el número que se necesita debe estar entre 0 y 1 y como se quiere que el logaritmo del número se comporte bien para la optimización de la verosimilitud logarítmica, se escoge predecir $z = \log \tilde{P}(y = 1|\boldsymbol{x})$

- Para generalizar el caso de la variable discreta con n valores, se produce un vector $\hat{\boldsymbol{y}}$ con $\hat{y}_i = P(y = 1|\boldsymbol{x})$ en donde se requiere no solo que $\hat{y}_i$ esté entre 0 y 1, sino que también el vector entero sume 1. El enfoque que función para la distribución de Bernoulli funciona para la multinomial

    - Se predicen las probabilidades logarítmicas no normalizadas $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$ donde $z_i = \log[\tilde{P}(y = 1|\boldsymbol{x})]$ y después se hace la exponencial y se normaliza $\boldsymbol{z}$ para obtener $\hat{\boldsymbol{y}}$ a través de la función *softmax*. El uso de exponenciales funciona muy bien para entrenar la *softmax* para resultar en un valor y usando máxima verosimilitud, y en este caso se quiere maximizar la siguiente verosimilitud logarítmica:

$$\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b} \;\Rightarrow\; softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\Rightarrow \max_{\boldsymbol{b},\boldsymbol{W}} \log[softmax(z_i)] = \max_{\boldsymbol{b},\boldsymbol{W}} z_i - \log\left[\sum_j \exp(z_j)\right]$$

    - El primer término de la ecuación muestra que el insumo tiene una contribución directa en la función de coste, y como este término no se satura (al ser lineal), el aprendizaje puede proceder, aunque el segundo término sea muy pequeño. Cuando se maximiza la función, este primer término impulsa a z_i a crecer, mientras que el segundo término impulsa a todas las $\boldsymbol{z}$ a decrecer

    - El segundo término se puede aproximar por $\max(z_j)$ porque $\exp(z_k)$ es insignificante para cualquier z_k que sea mucho menor a $\max(z_j)$. Esta aproximación permite ver que la función de verosimilitud logarítmica negativa siempre penaliza fuertemente a la predicción incorrecta más activa (más alta en valor), pero si la respuesta correcta tiene ya el mayor insumo de la *softmax*, entonces los términos (en negativo) casi que se cancelan entre sí

- La máxima verosimilitud no regularizada hará que el modelo aprenda parámetros que lleve a la *softmax* a predecir la fracción de las ocurrencias de cada resultado observado en el conjunto de entrenamiento:

$$softmax\big(z_i(\boldsymbol{x}_i;\boldsymbol{\theta})\big) \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i,\boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}$$

  - Debido a que la máxima verosimilitud es un método de estimación consistente, esto se garantiza siempre que la familia sea capaz de representar la distribución de entrenamiento

  - En la práctica, la capacidad limitada del modelo y la optimización imperfecta hacen ver que el modelo solo es capaz de aproximar estas fracciones

- Muchas funciones objetivo diferentes a la verosimilitud logarítmica no funcionan tan bien con la función *softmax*

  - Específicamente, las funciones objetivo que no usan un logaritmo para deshacer la exponencial del fallo *softmax* fallan en aprender cuando el argumento de la exponencial es muy negativo, haciendo que el gradiente desaparezca. En particular, el error cuadrático es una función de pérdida pobre y puede fallar al entrenar el modelo para cambiar sus resultados, aunque el modelo haga predicciones incorrectas altamente confiables

  - La *softmax* se puede saturar cuando las diferencias entre valores de insumo se hacen muy grande. Cuando esta función se satura, muchas funciones de coste basadas en la *softmax* también se saturan a no ser que sean capaces de invertir la función de activación saturada

  - La función *softmax* responde a la diferencia entre insumos porque es invariable al añadir un vector escalar al vector de insumos, de modo que, derivando una versión más estable numéricamente de la *softmax*, se puede observar como la *softmax* depende de la desviación de los diferentes valores de insumo de $\max(z_i)$

$$softmax(\boldsymbol{z}) = softmax(\boldsymbol{z} + \boldsymbol{c})$$

$$\Rightarrow\ softmax(\boldsymbol{z}) = softmax\big[\boldsymbol{z} - \max\big(z_j\big)\big]$$

  - Esta función se satura a 1 cuando el insumo es máximo ($z_i = \max\big(z_j\big)$) y z_i es más grande que todos los otros insumos, mientras que se satura a 0 cuando z_i no es máximo y $\max\big(z_j\big)$ es mucho mayor. Esta es una generalización de como se satura este tipo de función, y puede causar dificultades similares para

aprender si la función de pérdida no se diseña para compensar esto

- El argumento $\boldsymbol{z}$ para la función *softmax* se puede producir de dos maneras diferentes: a través de $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$ o fijando un valor de $\boldsymbol{z}$

  - Aunque el primer enfoque es sencillo, sobreparametriza el modelo debido a su restricción de que n resultados deben sumar 1, ya que eso significa que en verdad solo es necesario estimar $n-1$ probabilidades (el enésimo será tal que haga que la probabilidad sea 1 menos las otras)

  - Imponiendo el requerimiento de que un valor de $\boldsymbol{z}$ sea fijo, se estimarían solo $n-1$ probabilidades

  - Tanto el primer como el segundo enfoque pueden describir el mismo conjunto de probabilidades, pero tienen dinámicas de aprendizaje diferentes. En la práctica no hay mucha diferencia entre usar un argumento u otro, pero es más fácil implementar la versión sobreparametrizada

- Aunque las funciones de activación usadas en el resultado son las más comunes, las redes neuronales se pueden generalizar a casi cualquier capa de resultados. El principio de máxima verosimilitud proporciona una guía de cómo diseñar una buena función de coste para casi cualquier tipo de capa de resultados

  - En general, definiendo una distribución condicional $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$, el principio de máxima verosimilitud sugiere que $-\log[p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})]$ sea la función de coste (negativo de la verosimilitud logarítmica)

    - Como una red neuronal se puede interpretar como una función $f(\boldsymbol{x};\boldsymbol{\theta})$ tal que $f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{w}$, donde los resultados de la función no dan directamente las predicciones de $\boldsymbol{y}$ sino los parámetros de la distribución sobre $\boldsymbol{y}$, entonces la función de coste se puede interpretar como $-\log\big[p\big(\boldsymbol{y};\boldsymbol{w}(\boldsymbol{x})\big)\big]$

  - Por ejemplo, si se quisiera saber la varianza de una distribución condicional Gaussiana para $\boldsymbol{y}$ dado $\boldsymbol{x}$, hay una expresión cerrada dada por el estimador máximo verosímil en el caso escalar univariante

    - Un enfoque más computacionalmente intensivo es incluir la varianza como uno de los parámetros de la distribución $p(\boldsymbol{y}|\boldsymbol{x})$ que está controlado por $\boldsymbol{w} = f(\boldsymbol{x};\boldsymbol{\theta})$. La verosimilitud logarítmica negativa $-\log\big[p\big(\boldsymbol{y};\boldsymbol{w}(\boldsymbol{x})\big)\big]$ proporcionará una función de coste con los términos apropiados necesarios para realizar el procedimiento de optimización para aprender la varianza incrementalmente

- En el caso simple en el que la desviación estándar no depende del insumo, se puede hacer un nuevo parámetro en la red neuronal que se copia directamente a $\boldsymbol{w}$ en $p(\boldsymbol{y}; \boldsymbol{w}(\boldsymbol{x}))$. Este nuevo parámetro puede ser σ o un parámetro v representando σ^2 o uno β representando $1/\sigma^2$ (dependiendo de como se quiera parametrizar la distribución)

- En el caso que se quiera predecir diferente cantidad de varianza en $\boldsymbol{y}$ para diferentes valores $\boldsymbol{x}$, se habla del uso de un modelo heteroscedástico

  - En este caso, solo se hace la especificación de la varianza como uno de los valores resultantes de $f(\boldsymbol{x}; \boldsymbol{\theta})$, lo cual se suele hacer normalmente usando la precisión $1/\sigma^2$ y no su varianza

- En el caso multivariante lo más común es usar una matriz de precisión diagonal $diag(\boldsymbol{\beta})$, la cual funciona bien con el descenso de gradiente porque la verosimilitud logarítmica de la distribución normal parametrizada por $\boldsymbol{\beta}$ requiere solo multiplicación β_i además de sumar $\log(\beta_i)$. El gradiente de la multiplicación, la suma y el logaritmo se comportan bien para el descenso de gradiente

  - En comparación, si se parametrizara en términos de la varianza (o con la desviación estándar), entonces se tendría que usar la división y eso haría que esta tendiera hacia infinito arbitrariamente cerca de cero, posiblemente causando inestabilidad (aunque gradientes grandes ayudan a aprender mejor)

  - Si se usara la desviación, se tendría el mismo problema y el cuadrado (que se tendría que calcular adicionalmente) causaría que, para valores cerca de cero, el gradiente desapareciera, lo cual enseña porque parámetros al cuadrado son difíciles de aprender

- En cualquier caso de los anteriormente expuestos, uno se tiene que asegurar que la matriz de covarianzas de la normal es definida positiva. En el caso de la matriz de precisiones, como los valores propios de esta serán el recíproco de los valores propios de la matriz de covarianza, esto sirve para comprobar si esta última es definida positiva

  - Si se usa una matriz diagonal (o una matriz diagonal multiplicada por un escalar), entonces la única condición que se necesita forzar sobre el resultado del modelo es la positividad

- Si se supone que $\boldsymbol{a}$ es la activación cruda del modelo usado para determinar la precisión diagonal, se puede usar la función *softplus* para obtener el vector de precisiones positivas $\boldsymbol{\beta} = \zeta(\boldsymbol{a})$. Esta estrategia aplica igual si se usa la varianza o la desviación estándar en vez de la matriz de precisión o la matriz diagonal (multiplicada por el escalar)

- Es raro aprender una matriz de covarianzas o de precisión con una estructura más rica que la diagonal. Si la matriz de covarianzas está llena y es condicional, entonces se debe escoger una parametrización que garantiza que la matriz de covarianza predicha es definida positiva, lo cual se puede hacer a través del siguiente producto matricial de una matriz cuadrada no restringida $\boldsymbol{B}$:

$$\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^T(\boldsymbol{x})$$

- Si la matriz tiene rango completo, entonces calcular la verosimilitud es computacionalmente costoso porque se necesitaría una computación de $O(d^3)$ para el determinante y la inversa de $\boldsymbol{\Sigma}(\boldsymbol{x})$ de una matriz cuadrada $d \times d$

- A veces se quiere utilizar una regresión multimodal para predecir valores reales que provienen de una distribución condicional $p(\boldsymbol{y}|\boldsymbol{x})$ que puede tener varios picos en el espacio de $\boldsymbol{y}$ para un mismo valor de $\boldsymbol{x}$. En este caso, normalmente se utiliza un modelo de mezcla normal, y las redes neuronales con mezclas normales como resultado normalmente se conocen como redes de densidad mixta

  - El resultado de una mezcla gaussiana con n componentes se define por la distribución de probabilidad condicional siguiente:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \sum_{i=1}^{n} p(c = i|\boldsymbol{x}) N\left(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})\right)$$

- La red neuronal, por tanto, debe tener tres resultados: un vector definiendo $p(c = i|\boldsymbol{x})$, una matriz que proporcione $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$ para toda i, y un tensor que proporcione $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$ para toda i. Estos resultados deben satisfacer una serie de restricciones:

  - Los componentes de la mezcla $p(c = i|\boldsymbol{x})$ forman una distribución categórica (o *multinoulli distribution*) sobre los n componentes diferentes asociados con la variable latente c (no se observa el componente gaussiano que produce los datos) y se puede obtener por una *softmax* sobre un vector n-dimensional para garantizar que los resultados sean positivos y sumen 1

- Las medias $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$ indican la media asociada con cada componente gaussiano y no están restringidas, y si $\boldsymbol{y}$ es d-dimensional, la matriz resultante de tamaño $n \times d$ debe contener los n vectores d-dimensionales. Aprender estas medias por máxima verosimilitud es más difícil que aprenderlas en el caso en que el resultado solo tiene un componente (solo se quiere actualizar la media de aquel componente que ha producido el resultado)

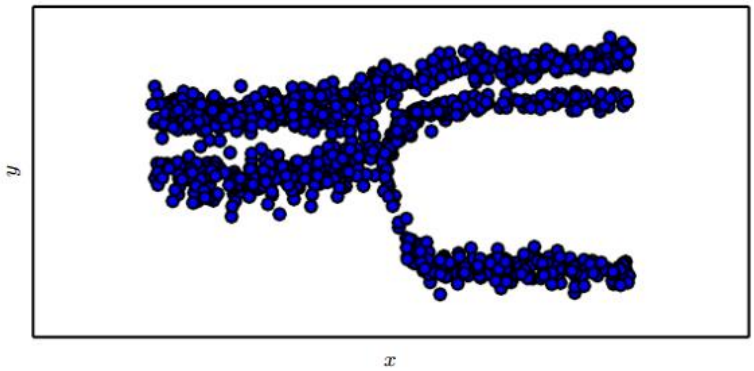
- Como en la práctica uno no sabe qué componente ha producido cada observación, la expresión para la verosimilitud logarítmica naturalmente pondera cada contribución del ejemplo a la pérdida de cada componente por la probabilidad de que el componente haya producido el ejemplo

    - Las covarianzas $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$ especifican la matriz de covarianzas para cada componente, e igual que en el ejemplo gaussiano anteriormente visto, se suele utilizar una matriz diagonal para evitar la necesidad de calcular determinantes. Como aplicar una estimación máximo verosímil es difícil, entonces es necesario asignar responsabilidades parciales a cada componente de la mezcla, y el descenso de gradiente automáticamente seguirá el proceso correcto si se da una especificación correcta de la verosimilitud logarítmica negativa

- Se ha reportado que la optimización basada en gradientes de mezclas gaussianas condicionales (sobre el resultado de las redes neuronales) puede no ser fiable, en parte debido a que se obtienen divisiones (por la varianza) que pueden ser numéricamente inestables. Una solución es utilizar gradientes de clip o escalar los gradientes de manera heurística

    - Los resultados de las mezclas gaussianas son particularmente efectivos en modelos generativos de habla o movimientos de objetos físicos

    - La estrategia de densidad de mezcla da una manera para que la red represente las múltiples modas de la variable de resultado y controlar la varianza del resultado, lo cual es crucial para obtener un alto grado de calidad en estos dominios de valor real

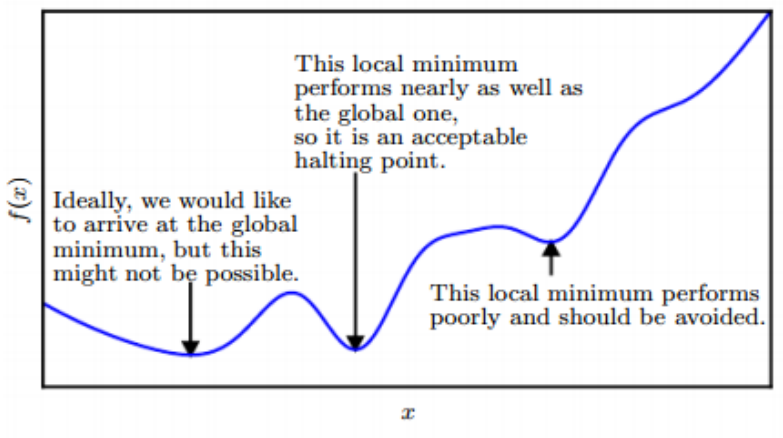


- En general, uno quiere continuar modelando grandes vectores $\boldsymbol{y}$ conteniendo más variables, e imponer estructuras más ricas sobre estas variables de resultado. Por ejemplo, uno puede querer que la red neuronal resulte en una secuencia de caracteres que forme una frase
    - En estos casos, uno puede continuar usando la máxima verosimilitud aplicada a nuestro modelo $p(\boldsymbol{y}; \boldsymbol{w}(\boldsymbol{x}))$, pero el modelo que describe $\boldsymbol{y}$ se vuelve muy complejo

Las redes neuronales artificiales: unidades escondidas

- Anteriormente se ha visto cómo diseñar las redes neuronales que son comunes para la mayoría de modelos paramétricos de aprendizaje estadístico entrenados con optimización basada en gradientes. Ahora se puede estudiar una característica única de las redes neuronales de retroalimentación: cómo escoger el tipo de unidades escondidas para usar en las capas escondidas de la red
    - Las unidades lineales rectificadas o *rectified linear units* (ReLU) es una elección predeterminada muy buena para la unidad escondida, aunque existen muchos otros tipos de unidades escondidas disponiblesç
        - Es difícil determinar cuando usar cada tipo de unidad, pero se pueden describir algunas intenciones básicas que motivan cada tipo de unidades escondidas. Estas pueden ayudar a decidir cuando probar una u otra, aunque suele ser imposible predecir con antelación cual funcionará mejor
        - El proceso de diseño consiste de prueba y error, obteniendo una intuición de qué tipo de unidad escondida puede funcionar bien, y después entrenando una red con ese tipo de unidad escondida y evaluando su rendimiento en un conjunto de validación
    - Algunas de las unidades escondidas no son diferenciables en todos los puntos de insumo, como la ReLU $g(z) = \max\{0, z\}$, que no es diferenciable en el punto $z = 0$

- Aunque esto parezca que invalida g para su uso en un algoritmo de descenso de gradiente, en la práctica, el algoritmo sigue funcionando bien para estos modelos. Esto se debe parcialmente a que los algoritmos de entrenamiento de redes neuronales no suelen llegar a un mínimo local de la función de coste, sino que reducen su valor significativamente y ya



- Como no se espera que se llegue al mínimo en donde el gradiente es $\mathbf{0}$, entonces es aceptable que el mínimo de la función de coste corresponda a puntos con gradiente no definido. Las unidades escondidas que no son diferenciables normalmente no lo son solo en un número pequeño de puntos

- En general, las funciones usadas en el contexto de las redes neuronales tienen derivada por la izquierda y por la derecha definidas. En el caso de la unidad ReLU $g(z) = \max\{0, z\}$ tiene derivada por la izquierda igual a 0 y derivada por la derecha igual a 1

- Las implementaciones de *software* del entrenamiento de redes neuronales normalmente devuelven derivadas laterales más que reportar un error o que la derivada no está definida. Esto se puede justificar heurísticamente debido a que la optimización basada en gradientes está sujeta a error y cuando se pide que se evalúe $g(0)$ este 0 no es exacto

- A no ser que se indique explícitamente, la mayoría de unidades escondidas se pueden describir como unidades que aceptan un vector de insumos $\boldsymbol{x}$, calculando una transformación afín $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$ y aplicando después una función no lineal a cada elemento $g(\boldsymbol{z})$

    - La mayoría de unidades escondidas se distinguen entre ellas solo por la elección de la forma funcional de la función de activación $g(\boldsymbol{z})$

- Como se ha mencionado, las unidades ReLU tienen una función de activación $g(z) = \max\{0, z\}$. Estas unidades son fáciles de optimizar porque son similares a las unidades lineales

    - La única diferencia entre las unidades lineales y las ReLU es que la función de la ReLU resulta en 0 para la mitad de su dominio. Esto hace que las derivadas a través de una ReLU se mantengan grandes siempre que la unidad esté activa y que también son consistentes

        - La segunda derivada de la ReLU es 0 en casi todo su dominio, y la primera derivada es 1 siempre que la unidad esté activa. Esto significa que la dirección del gradiente es más útil para aprender que usando unidades que introdujeran efectos de segundo orden

        - Las unidades ReLU normalmente se utilizan sobre una transformación afín $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$

        - Cuando se inicializan los parámetros de la transformación afín, una buena práctica es fijar todos los elementos de $\boldsymbol{b}$ a un número positivo pequeño como 0.1. Esto hace muy probable que la ReLU esté activa inicialmente para la mayoría de insumos en el conjunto de entrenamiento y permitir que las derivadas pasen

    - Existen varias generalizaciones de la ReLU, las cuales rinden igual como las ReLU y ocasionalmente mejor

        - Una desventaja de las ReLU es que no pueden aprender a través de métodos basados en gradientes en ejemplos cuya activación es nula. Por lo tanto, una variedad de generalizaciones a las ReLU garantiza que recibirán un gradiente en todas partes

        - Hay tres generalizaciones de las ReLU que se basan en usar una pendiente α_i cuando $z_i < 0$, a través de la función $h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$, las cuales pueden ser la rectificación en valor absoluto (fijando $\alpha_i = -1$ para obtener $g(z) = |z|$), la *leaky ReLU* (fijando α_i a un valor pequeño como 0.01) y la *parametric ReLU* (que trata α_i como un parámetro que se puede aprender)

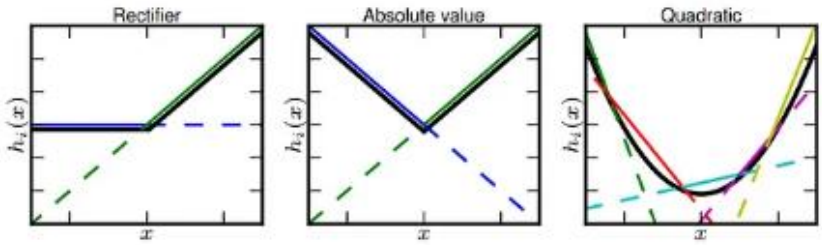
    - Las unidades *maxout* generalizan las unidades lineales rectificadas más allá, dado que en vez de aplicar una función $g(z)$ a cada uno de los n elementos de $\boldsymbol{z}$, estas unidades dividen $\boldsymbol{z}$ en grupos de k elementos (k elementos consecutivos del vector $\boldsymbol{z}$) y resultan en el elemento máximo de uno de esos grupos con k elementos

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

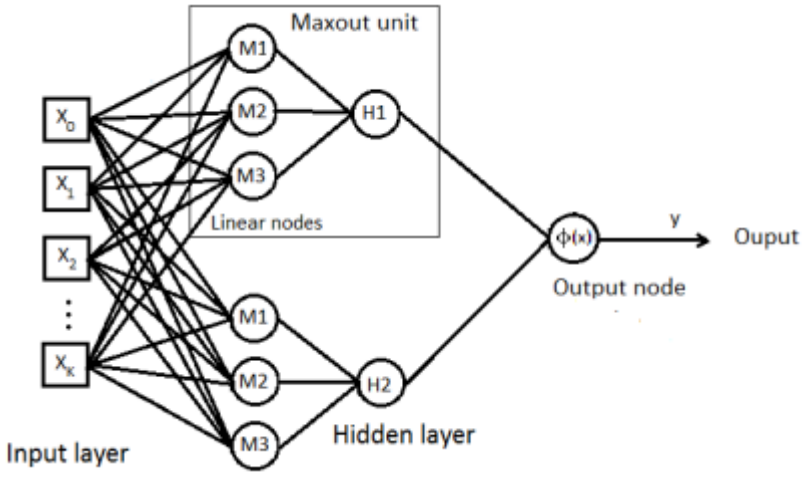
$$where \quad \mathbb{G}^{(i)} = \{(i-1)k+1, \dots, ik\} \quad for \ \ i = 1, \dots, n/k$$

- Esto proporciona una manera de aprender o aproximar una función lineal por piezas (dado que cada elemento en $\boldsymbol{z}$ es una combinación lineal) que responde a múltiples direcciones en el espacio de insumos de $\boldsymbol{x}$ (para cada grupo, correspondiente a un rango de valores de $\boldsymbol{x}$ se escoge una función lineal z_j). El número de unidades dentro de una capa se determina por k, dado que este número será n/k

- Una capa con unidades *maxout* puede aprender una función lineal por piezas convexa de hasta k piezas (1 por cada combinación lineal en $\boldsymbol{z}$), de modo que estas unidades pueden entenderse como unidades que aprenden la función de activación (no solo la relación entre unidades). Con una k lo suficientemente grande, una capa *maxout* puede aprender cualquier función convexa con alta fidelidad, no solo la relación entre las unidades escondidas



- En particular, una capa *maxout* con 2 unidades puede aprender a implementar la misma función que una unidad con ReLU, con una función de rectificación de valor absoluto, con una *leaky ReLU* o con una ReLU paramétrica. La capa, no obstante, se parametrizará diferente de cualquiera de los otros tipos de capas, de modo que las dinámicas de aprendizaje serán diferentes aún en casos donde la capa aprende a implementar la misma función de $\boldsymbol{x}$ como uno de los otros tipos de capas



- Cada unidad *maxout* está parametrizada por k vectores de n elementos en vez de solo 1, de modo que las unidades necesitan más regularización si el conjunto de entrenamiento es grande y el número de piezas por capa se mantiene bajo

- Estas unidades tienen ventajas adicionales, tales como las ventajas estadísticas y computacionales por requerir menos parámetros. Específicamente, si las características capturadas por los n filtros lineales (los n elementos en $\boldsymbol{z}$) se pueden resumir sin perder información tomando el máximo en cada grupo de k valores, entonces la siguiente capa puede trabajar con un número k veces menor de ponderaciones (dado que habrán n/k unidades)

- Como cada unidad funciona con múltiples combinaciones lineales, entonces las unidades *maxout* tienen algo de redundancia que les ayuda a resistir el fenómeno de olvido catastrófico o *catastrophic forgetting*, en donde las redes neuronales olvidan como realizar tareas para las que se han entrenado en el pasado

- Las unidades ReLU y todas sus generalizaciones se basan en el principio de que los modelos son más fáciles de optimizar si su comportamiento está cerca de un comportamiento lineal

    - Este principio general de utilizar un comportamiento lineal para obtener una optimización más sencilla también aplica en otros contextos aparte de redes neuronales lineales profundas

    - Las redes recurrentes pueden aprender de secuencias y producir una secuencia de estados e insumos, y cuando las entrena, uno necesita propagar información a través de diferentes momentos temporales, que es más fácil cuando cálculos lineales (con derivadas con magnitud cercana a 1) están involucradas

        - Una de las mejores arquitecturas de redes neuronales recurrentes, LTSM, propaga la información a través del tiempo a través de sumas (activación lineal simple)

- Antes de la introducción de las unidades ReLU, la mayoría de redes neuronales usaban la función de activación logística sigmoide $g(z) = \sigma(z)$ o la función de activación tangente hiperbólica $g(z) = \tanh(z)$, las cuales están relacionadas a través de $\tanh(z) = 2\sigma(2z) - 1$

    - Anteriormente ya se han visto las unidades sigmoides como unidades de resultados, usadas para predecir probabilidades de una variable binaria. A diferencia de las unidades lineales a trozos, las sigmoides se saturan en la mayoría de su dominio (a un valor grande cuando z es grande o un valor pequeño cuando z es pequeño) y son extremadamente sensibles a su insumo cuando z está cerca de 0

        - La saturación generalizada de las unidades sigmoides puede hacer que el aprendizaje por gradiente sea muy difícil, por lo que su uso como unidades escondidas en redes neuronales no se recomienda

        - Su uso como unidades de resultados es compartible con el uso de entrenamiento basado en gradientes cuando se usa una función de coste apropiada que pueda deshacer la saturación de la sigmoide en la capa de resultado

    - Cuando una función de activación sigmoide se tiene que usar, la función de activación de la tangente hiperbólica típicamente rinde mejor que la logística sigmoide

        - Se parece más a la función identidad, en el sentido de que $\tanh(0) = 0$ mientras que $\sigma(0) = 1/2$

        - Como $\tanh$ es tan similar a la identidad cerca de 0, entrenar una red neuronal $\hat{y} = \boldsymbol{w}^T \tanh(\boldsymbol{U}^T \tanh(\boldsymbol{V}^T \boldsymbol{x}))$ se parece a entrenar un modelo lineal $\hat{y} = \boldsymbol{w}^T \boldsymbol{U}^T \boldsymbol{V}^T \boldsymbol{x}$ siempre que las activaciones de las redes sean pequeñas

        - Esto hace que entrenar una red neuronal con la tangente hiperbólica sea más sencillo

        - Las activaciones sigmoides suelen ser más comunes en otras áreas que no sean redes neuronales de retroalimentación, tales como en redes neuronales recurrentes, autocodificadores y modelos probabilísticos

- Aunque se han visto las unidades ReLU, las sigmoides y las tangentes hiperbólicas, existen otras posibilidades de unidades, aunque se usan menos frecuentemente

    - En general, una variedad amplia de funciones diferenciables rinde muy bien, y algunas de las funciones de activación no publicadas rinden tan bien como las más populares

        - Durante la investigación y el desarrollo de nuevas técnicas, es común probar varias funciones de activación y encontrar varias variantes cuyo rendimiento es comparable

        - Esto significa que nuevas unidades se publican solo si se ha demostrado que proporcionan una mejora significativa

    - Aunque no es práctico enumerar las diferentes funciones que se han propuesto en la literatura, se destacan unos pocos ejemplos que son muy útiles o distintivos

        - Una posibilidad es no tener una función de activación ninguna, sino tener la función identidad, lo cual se ha visto ya anteriormente. Esto haría que la red fuera lineal (si las capas esconidas y la capa de resultados son lineales)

        - Es aceptable que algunas capas sean puramente lineales: considerando una red $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$ con n insumos y p resultados, se puede reemplazar esta con dos escondidas, una con matriz de ponderaciones $\boldsymbol{U}$ y otra $\boldsymbol{V}$**.** Si la primera capa no tiene función de activación, entonces esencialmente se ha factorizado la matriz de ponderaciones original de la capa original basada en $\boldsymbol{W}$: el enfoque factorizado consiste en calcular $\boldsymbol{h} = g(\boldsymbol{V}^T\boldsymbol{U}^T\boldsymbol{x} + \boldsymbol{b})$

        - Si $\boldsymbol{U}$ produce q resultados, entonces $\boldsymbol{U}$ y $\boldsymbol{V}$ juntos contienen solo $(n + p)q$ parámetros, mientras que $\boldsymbol{W}$ tendría np parámetros. Para una q pequeña, esto puede ser un gran ahorro de parámetros

        - El coste de hacer esto es que se tiene que restringir la transformación lineal para que sea *low-rank*, pero estas relaciones *low-rank* suelen ser suficientes. Las unidades escopndidas lineales, por tanto, ofrecen una manera efectiva de reducir el número de parámetros en una red

    - Las unidades *softmax* son otro tipo de unidades que usualmente se utilizan como unidades de resultados (como se ha visto) pero que también se pueden usar como unidades escondidas

        - Las unidades *softmax* naturalmente representan una distribución de probabilidad sobre una variable discreta con k posibles valores, de modo que se usa como un tipo de interruptor

        - Estos tipos de unidades escondidas normalmente solo se usan en arquitecturas más avanzadas que explícitamente aprenden a manipular memoria

    - Otros tipos de unidades razonablemente comunes son las siguientes:

        - La unidad de base radial o *radial basis function* (RBF) se calcula como $h_i = \exp\left(-\frac{1}{\sigma^2}\left\|\boldsymbol{W}_{:,i} - \boldsymbol{x}\right\|^2\right)$. Esta función se vuelve más activa cuando $\boldsymbol{x}$ está cerca de $\boldsymbol{W}_{:,i}$, y como se satura a 0 para la mayoría de $\boldsymbol{x}$, puede ser difícil de optimizar

        - La unidad *softplus* tiene función de activación $g(a) = \zeta(a) = \log(1 + e^a)$, pero esta versión suave del rectificador lineal no rinde tan bien como la misma ReLU, por lo que no se suele utilizar. La *softplus* demuestra que su rendimiento como unidad escondida puede ser contraintuitivo, dado que uno espera que tenga ventajas sobre la ReLU por su diferenciabilidad en todos los puntos y su menor saturación completa, pero empíricamente no tiene ventajas

        - La unidad $\tanh$ fuerte o *hard* $\tanh$ tiene una forma similar a $\tanh$ y al rectificador lineal, pero a diferencia de este último, está acotada, y tiene función de activación $g(a) = \max(-1, min(1, a))$

Las redes neuronales artificiales: diseño de la arquitectura

- Otra consideración clave de las redes neuronales es determinar la arquitectura, refiriéndose al número de unidades que la red debería tener y cómo conectar estas unidades las unas con las otras

    - La mayoría de redes neuronales se organizan en grupos de unidades llamados capas. La mayoría de arquitecturas de redes neuronales reorganizan estas capas en una estructura de cadena, con cada capa siendo una función de la capa anterior

        - La estructura de las capas sería la siguiente:

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) \Rightarrow \boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)T}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\Rightarrow \ldots \Rightarrow \boldsymbol{h}^{(l)} = g^{(l)}\left(\boldsymbol{W}^{(l)T}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

- En estas arquitecturas basadas en cadenas, la principal consideración arquitectural es la elección de la profundidad de la red y del ancho de cada una de las capas

    - Una red neuronal con solo una capa escondida es suficiente para ajustarse a un conjunto de entrenamiento

    - Las redes más profundas normalmente son capaces de usar menos unidades por capa y muchos menos parámetros, y suelen generalizarse al conjunto de comprobación. No obstante, son más difíciles de optimizar

    - La arquitectura ideal de una red para una tarea se debe encontrar a través de la experimentación, guiada por la monitorización del error en el conjunto de validación

- Un modelo lineal, mapeando características a resultados a través de multiplicaciones matriciales puede, por definición, representar solo funciones lineales. No obstante, a veces se requiere aprender funciones no lineales, pero las redes neuronales proporcionan un marco de aproximación universal para estas sin necesidad de diseñar una familia de modelos especializados

    - Específicamente, el teorema de aproximación universal expresa que una red neuronal retroalimentada con una capa lineal y al menos una capa escondida lineal con cualquier función de activación que comprima el valor de insumo en un rango específico puede aproximar cualquier función medible en el sentido de Borel de un espacio de dimensiones finitas a otro con cualquier cantidad de error diferente de cero, siempre que la red tenga las suficientes unidades

        - Las derivadas de la red neuronal retroalimentada pueden aproximar cualquiera de las derivadas de la función arbitrariamente bien

        - Aunque el concepto de medible en el sentido de Borel es complicado, solo hace falta saber que cualquier función continua en un subconjunto cerrado y acotado de $\mathbb{R}^n$ es medible en el sentido de Borel y, por tanto, se puede aproximar a través de una red neuronal

        - Una red neuronal puede aproximar cualquier función que mapee cualquier espacio de dimensiones finitas discreto a otro

        - Aunque los teoremas originales que se elaboraron primero eran en términos de unidades con funciones de activación que se saturan tanto para valores muy negativos como para valores muy

positivos, pero salieron más teoremas demostrados para una clase mayor de funciones de activación (que incluye la ReLU)

- El teorema de aproximación universal significa que, independientemente de la función que se quiera aprender, se sabe que un MLP grande es capaz de representar esta función

    - No obstante, no está garantizado que el algoritmo de entrenamiento será capaz de aprender esta función en la práctica. Aunque el MLP sea capaz de representar esta función, el aprendizaje puede fallar por no poder encontrar el valor de los parámetros que corresponden a la función deseada o porque se escoja una función errónea debido al sobreajuste

    - Recordando el teorema de *no free lunch* en aprendizaje estadístico, no hay un algoritmo de aprendizaje automático universalmente superior. Las redes neuronales retroalimentadas proporcionan un sistema universal para representar funciones, en el sentido de que, dada una función, existe una red neuronal que representa una función. No hay un procedimiento universal para examinar el conjunto de entrenamiento de ejemplos específicos y escoger una función que generalizará los puntos que no estén en el conjunto de entrenamiento

- El teorema dice que existe una red lo suficientemente grande como para conseguir cualquier grado de exactitud que se desee, pero este teorema no dice qué tan grande tiene que ser la red

    - Barron, en su investigación de 1993, proporciona algunas cotas para el tamaño de una red de una sola capa necesaria para aproximar una gran clase de funciones. Desafortunadamente, en el peor caso, se necesitaría un número exponencial de unidades escondidas (posiblemente con una unidad escondida correspondiente a cada configuración de insumos que se necesita distinguir)

    - Esto se puede ver fácilmente en el caso binario: el número de posibles funciones binarias en los vectores $\boldsymbol{v} \in \{0,1\}^n$ es 2^{2^n} y seleccionar esta función requiere 2^n bits, que generalmente requiere $O(2^n)$ grados de libertad

- Existen familias de funciones que se pueden aproximar eficientemente por una arquitectura con una profundidad mayor a algún valor d, pero la cual requiere un modelo mucho mayor si la profundidad se restringe a menos o igual que d

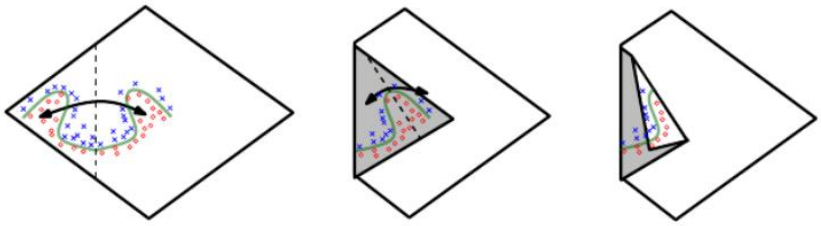
- En muchos casos, el número de unidades requeridas para un modelo simple (una red con una sola capa) es exponencial en n. Estos resultados primero se probaron para modelos que no se parecen a las redes neuronales continuas y diferenciables usadas para aprendizaje automático, pero se han extendido a estos modelos

- Leshno y otros investigadores, en su investigación de 1993, demostraron que las redes neuronales simples con una familia amplia de funciones de activación no polinómicas (incluyendo ReLU) tienen propiedades de aproximación universal, pero estos resultados no contestan a las preguntas de profundidad o eficiencia (solo que pueden representar cualquier función)

- Montufar y otros investigadores, en su investigación de 2014, demostraron que las funciones representables con una red rectificadora puede requerir un número exponencial de unidades escondidas con una red simple

  - Específicamente, demostraron que redes con unidades de funciones lineales por partes pueden representar funciones con un número de regiones que es exponencial en la profundidad de la red neuronal

  - La siguiente ilustración muestra cómo una red con rectificación de valor absoluto crea imágenes espejo de la función calculado por encima de alguna unidad escondida, con respecto al insumo de esa unidad. Cada unidad especifica donde "doblar" el espacio de insumo con tal de crear respuestas espejo, y componiendo estas operaciones de doblado, se obtiene un número exponencial un gran número de regiones lineales por piezas que pueden capturar todo tipo de patrones regulares (repetitivos)



  - De manera más precisa dice que el número de regiones lineales talladas por una red neuronal rectificadora con d insumos, profundidad l y n unidades por capa escondida es exponencial en la profundidad l y es, precisamente, el siguiente número:

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right)$$

- En el caso de redes *maxout* con k filtros lineales por unidad, el número de regiones lineales es $O\left(k^{(l-1)+d}\right)$

- Claramente, no hay garantía de que los tipos de funciones que se quieran aprender en aplicaciones de inteligencia artificial y aprendizaje automático compartan esta propiedad

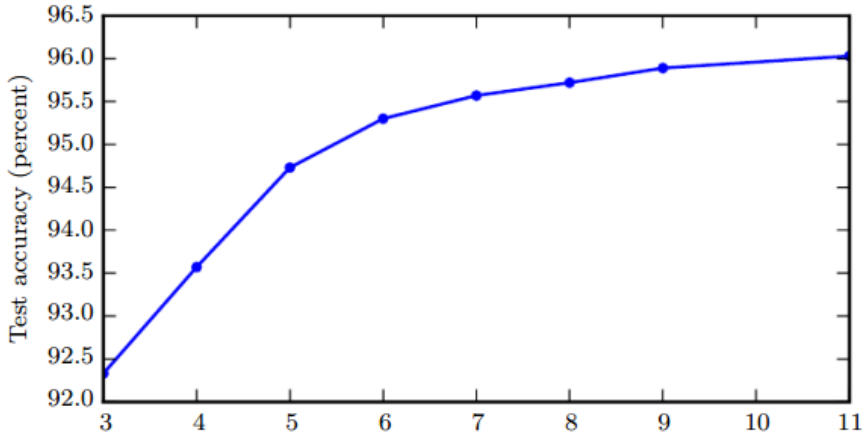
- Puede ser que se quiera seleccionar un modelo de red neuronal profunda por razones estadísticas: cuando se escoge un algoritmo de aprendizaje automático, uno implícitamente muestra unas creencias preconcebidas sobre el tipo de función que el algoritmo debe aprender

  - Escoger un modelo de red neuronal lleva consigo la creencia general de que la función que se quiere aprender involucra la composición de varias funciones lineales simples

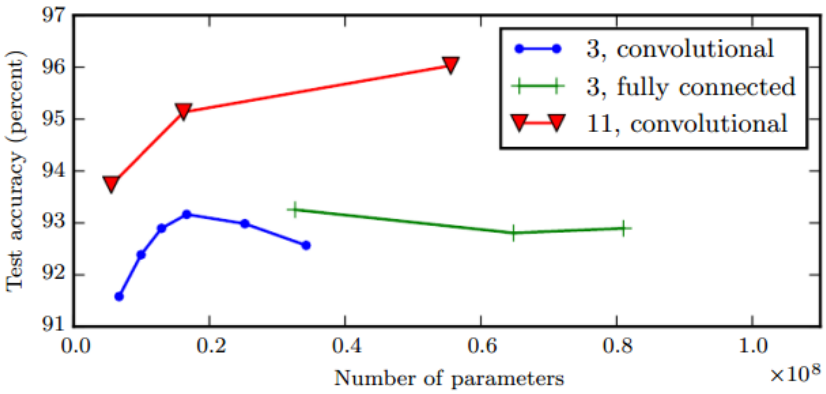
  - Esto se puede interpretar desde el punto de vista del aprendizaje de representación como si se dijera que se cree que el problema de aprendizaje consiste en descubrir un conjunto de factores subyacentes de variación que pueden ser descritas en términos de otros factores subyacentes de variación más simples

  - Alternativamente, se puede interpretar como una creencia de que la función que se quiere aprender es un código computacional consistiendo de varios pasos, donde cada paso usa el resultado del paso previo. Estos pasos intermedios no son necesariamente factores de variación, pero son análogos a *counters* o *pointers* que la red utiliza para organizar su procesamiento interno

- Empíricamente, mayor profundidad no parece resultar en una mejor generalización para una gran variedad de tareas. Esto sugiere que las arquitecturas profundas expresan una distribución *a priori* sobre el espacio de funciones que el modelo aprende



- El experimento de Goodfellow y otros investigadores en 2014 muestra que incrementar el número de parámetros en redes convolucionales sin incrementar su profundidad no es tan efectivo al incrementar el rendimiento en el conjunto de comprobación



- Hasta ahora se han descrito redes neuronales como simples cadenas de capas, con las principales consideraciones siendo la profundidad y la amplitud de cada capa. En la práctica, las redes neuronales muestran mucha más diversidad, y muchas redes neuronales se han desarrollado para tareas específicas (como visión computacional, redes recurrentes, etc.)

    - En general, las capas no necesitan estar conectadas en cadena, aunque es la práctica más común. Muchas arquitecturas construyeron una cadena principal, pero añadieron características arquitecturales adicionales a esta, tales como conexiones de salto o *skip connections*, que van de la capa i a la $i + 2$ o así

        - Saltarse estas conexiones hace más fácil para el gradiente fluir de las capas de entrenamiento a capas más cercas de los insumos

    - Otra consideración importante en el diseño de la arquitectura es cómo conectar exactamente un par de capas entre ellas. En la red neuronal vista descrita por $\boldsymbol{W}$, cada insumo se conecta a cada unidad de resultados

        - Muchas redes neuronales tienen muchas menos conexiones, de modo que cada unidad en la capa de insumo está conectada a solo un pequeño subconjunto de unidades en la capa de resultados

        - Las estrategias para reducir el número de conexiones reducen el número de parámetros y la cantidad de computación requerida para evaluar la red, pero normalmente dependen mucho del problema. Por ejemplo, las redes convolucionales usan patrones especializados de conexiones *sparse* que son muy efectivas para problemas de visión computacional

Las redes neuronales artificiales: *back-propagation*

- Finalmente, es importante motivar y analizar el algoritmo de propagación hacia atrás o *back-propagation* y otros algoritmos de diferenciación que permiten estimar los parámetros de los modelos de redes neuronales de retroalimentación

    - Cuando se usan redes neuronales retroalimentadas para aceptar un insumo $\boldsymbol{x}$ y producir un resultado $\hat{\boldsymbol{y}}$, la información fluye hacia adelante a través de la red

        - El insumo $\boldsymbol{x}$ proporciona la información inicial que se propaga hacia la derecha o hacia arriba de la red a las unidades escondidas en cada capa y finalmente produce $\hat{\boldsymbol{y}}$. A este proceso se le conoce como propagación hacia adelante o *forward propagation*

        - Durante el entranamiento, la propagación hacia adelante puede continuar hasta que produce un coste escalar $J(\boldsymbol{\theta})$ (por el proceso de derivar la función de coste que se usa para entrenar la red neuronal). Esto puede causar problemas para diversas capas posteriores, dado que no se podría obtener la derivada para todas ellas

    - El algoritmo de propagación hacia atrás o *back-propagation*, normalmente conocido como *backprop*, permite que la información sobre el coste fluya hacia atrás a través de la red, con tal de ser capaces de calcular el gradiente

        - El cálculo de la expresión analítica para el gradiente es muy sencillo, pero evaluar numéricamente esta expresión puede ser computacionalmente costoso. El algoritmo de *backpropo* permite hacer esto de manera simple y no costosa

        - Este algoritmo no es todo el algoritmo usado para entrenar la red neuronal multicapa, sino que solo se refiere al método para calcular el gradiente, mientras que otro algoritmo como el descenso del gradiente estocástico se usa para realizar el aprendizaje utilizando este gradiente

        - Este algoritmo, además, no solo se puede usar con redes neuronales, sino que también se puede usar para calcular la derivada de cualquier función. Específicamente, el algoritmo describe como calcular el gradiente $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ para una función arbitraria f, donde $\boldsymbol{x}$ es el conjunto de variables cuyas derivadas se desean, e $\boldsymbol{y}$ es un conjunto adicional de variables que son los insumos de la función, pero para los cuales no se requiere la derivada

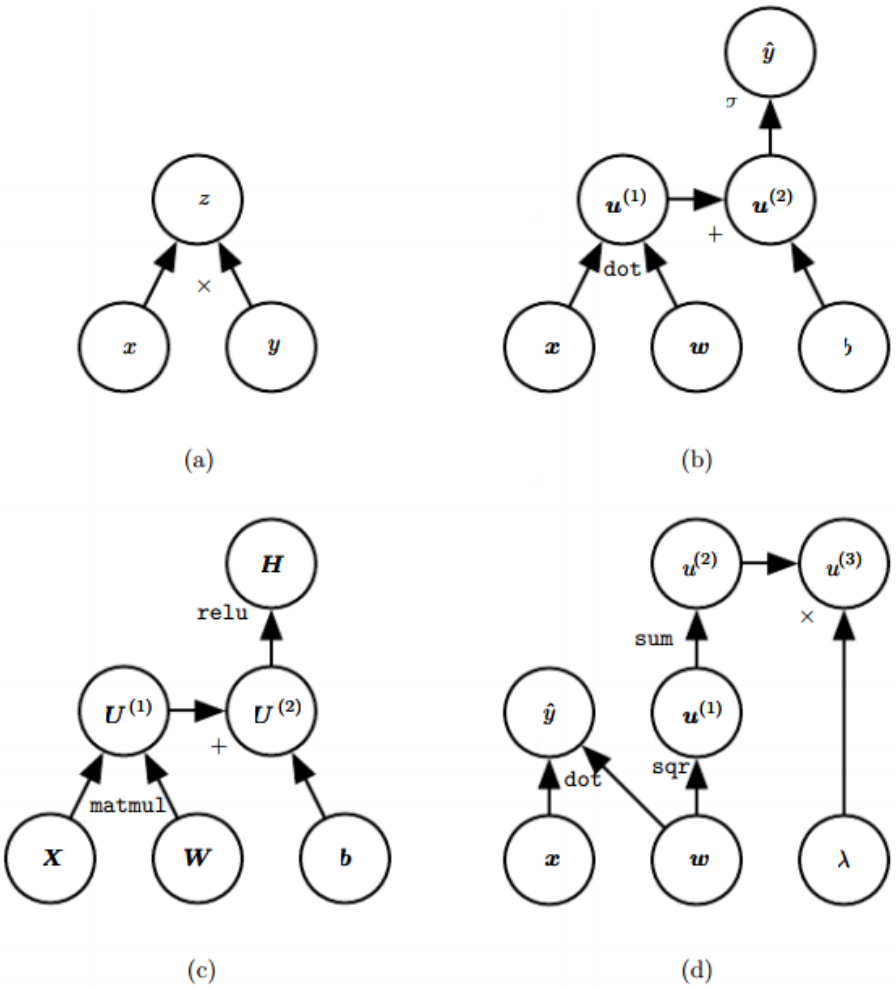
- En muchos algoritmos de aprendizaje, el gradiente que normalmente se requiere es el gradiente de la función de coste con respecto a sus parámetros $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Muchos algoritmos de aprendizaje estadístico involucran calcular otras derivadas para el proceso de aprendizaje o para analizar el modelo aprendido

    - El algoritmo de *backprop* se puede aplicar a estas tareas también y no está restringido a calcular el gradiente de la función de coste con respecto a sus parámetros

    - La idea de calcular derivadas propagando la información a través de la red es muy general, y se puede usar para calcular valores tales como la jacobiana de una función f con múltiples resultados. No obstante, el desarrollo se centra en el caso simple de un solo resultado

- Hasta ahora se ha discutido de manera relativamente informal las redes neuronales con lenguaje de grafos, pero es necesario usar el lenguaje de grafos computacionales con tal de describir el algoritmo de *backprop* detalladamente

    - Existen muchas maneras de formalizar el cálculo como grafos. En este caso, se usa cada nodo del grado para indicar una variable, la cual puede ser un escalar, un vector, una matriz, un tensor, u otro tipo de variable



- Para formalizar los grafos, se necesita introducir la idea de operación: una operación es simplemente una función de una o más variables. Aquellas operaciones más complejas que el conjunto de operaciones que se consideran pueden estar descritas como la composición de varias operaciones juntas

- Sin pérdida de generalidad, se define una operación que devuelve una sola variable de resultado (no pierde generalidad porque la variable de resultado puede tener múltiples entradas como un vector o una matriz). Las implementaciones de *software* del *backprop* normalmente permiten operaciones con múltiples resultados, pero se ignoran en el desarrollo porque introducen detalles irrelevantes para el entendimiento conceptual

- Si una variable y se calcula al aplicar una operación a una variable x, entonces se dibuja un arco direccionado de x a y. A veces se anota el nombre de la operación aplicada en el nodo de resultado, pero depende del contexto

- La regla de la cadena del cálculo se usa para calcular las derivadas de funciones formadas por la composición de otras funciones cuyas

derivadas son conocidas. El *backprop* es un algoritmo que calcula la regla de la cadena, con un orden específico de operaciones que es altamente eficiente

- Esta regla se puede generalizar: suponiendo que $\boldsymbol{x} \in \mathbb{R}^m$ y $\boldsymbol{y} \in \mathbb{R}^n$ y que g es una función que mapea de $\mathbb{R}^m$ a $\mathbb{R}^n$ y que f es una función que mapea de $\mathbb{R}^n$ a $\mathbb{R}$, si $\boldsymbol{y} = g(\boldsymbol{x})$ y $z = f\big(g(\boldsymbol{x})\big) = f(\boldsymbol{y})$, entonces se obtiene el siguiente resultado:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \qquad \nabla_{\boldsymbol{x}} z = [J_{\boldsymbol{x}}(\boldsymbol{y}, \boldsymbol{x})]^T \nabla_{\boldsymbol{y}} z$$

$$where \;\; J_{\boldsymbol{x}}(\boldsymbol{y}, \boldsymbol{x}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

- A partir de este resultado, se puede ver que el gradiente de la variable $\boldsymbol{x}$ se puede obtener multiplicando la matriz jacobiana $J_{\boldsymbol{x}}(\boldsymbol{y}, \boldsymbol{x})$ por un gradiente $\nabla_{\boldsymbol{y}} z$. El algoritmo de *backprop* consiste en realizar el producto entre la Jacobiana y el gradiente para cada operación en el grafo

- Normalmente no se aplica el algoritmo a vectores, sino a tensores de dimensionalidad arbitraria, pero es conceptualmente equivalente que realizar este algoritmo con vectores. La única diferencia está en cómo los números se organizan en una rejilla para formar un tensor

  - Uno puede imaginar que se aplana cada tensor en un vector antes de ejecutar el *backprop*, calculando así un gradiente con valores vectoriales, y después volver a darle forma de tensor al gradiente. De esta manera, el *backprop* sigue consistiendo en multiplicar jacobianas por gradientes

  - Para denotar el gradiente de un valor z con respecto a un tensor $\boldsymbol{X}$, se escribe $\nabla_{\boldsymbol{X}} z$ (como si fuera un vector), y los índices de $\boldsymbol{X}$ ahora tendrán múltiples coordenadas. Esto se puede abstraer usando solo una variable i para representar una tupla de índices, por lo que para todas las posibles tuplas i, $(\nabla_{\boldsymbol{X}} z)_i$ da $\partial z / \partial \boldsymbol{X_i}$

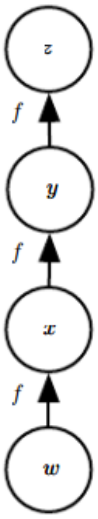
  - Usando esta notación, se puede reescribir la regla de la cadena que aplica a tensores. Si $\boldsymbol{Y} = g(\boldsymbol{X})$ y $z = f(\boldsymbol{Y})$, entonces se obtiene la siguiente expresión:

$$\nabla_{\boldsymbol{X}} z = [J_{\boldsymbol{x}}(\boldsymbol{y}, \boldsymbol{x})]^T \nabla_{\boldsymbol{y}} z$$

$$where \quad J_x(\boldsymbol{y}, \boldsymbol{x}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

- Es sencillo escribir la expresión algebraica para el gradiente de un escalar con respecto a cualquier nodo en el grado computacional que produjo el escalar. No obstante, evaluar realmente la expresión en la computadora introduce consideraciones adicionales

    - Específicamente, muchas subexpresiones pueden estar repetidas múltiples veces dentro de la expresión general del gradiente. Cualquier procedimiento que calcule el gradiente deberá escoger si guardar estas subexpresiones o recalcularlas varias veces



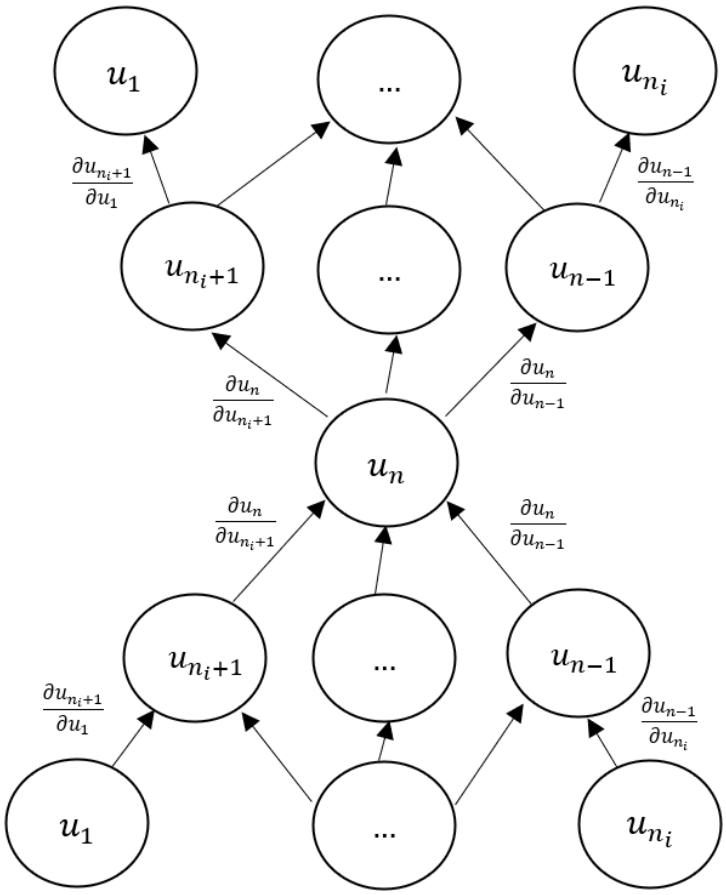
$$\begin{aligned} &\frac{\partial z}{\partial w} \\ =&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\ =&f'(y)f'(x)f'(w) \\ =&f'(f(f(w)))f'(f(w))f'(w) \end{aligned}$$

        - Para grafos complicados, puede haber exponenciales cálculos malgastados, haciendo que una implementación ingenua de la regla de la cadena sea inasequible. En otros casos, calcular la misma subexpresión dos veces puede ser una manera válida de reducir el consumo de la memoria al coste de mayor tiempo de computación

    - Primero se considera una versión del algoritmo de *backprop* que especifica directamente el cálculo del gradiente, en el orden en que se haría realmente y acorde a la aplicación recursiva de la regla de la cadena

        - Considerando un grafo computacional que describe como se calcula un solo escalar $u^{(n)}$ (la pérdida en el ejemplo de entrenamiento, por ejemplo), el cual es la cantidad cuyo gradiente se quiere obtener con respecto a los n_i nodos de insumo $u^{(1)}$ a $u^{(n_i)}$. Por lo tanto, se quiere obtener $\partial u^{(n)}/\partial u^{(i)}$ para $i \in \{1,2,\dots,n_i\}$

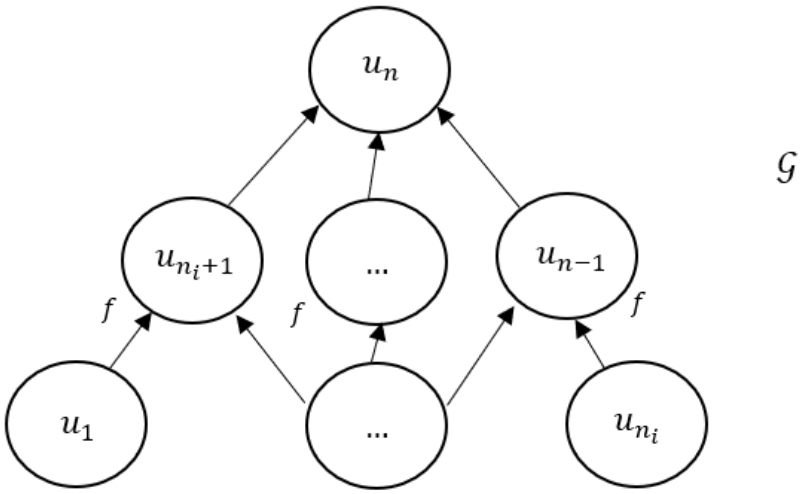
- En la aplicación del *backprop* para calcular gradientes para el descenso del gradiente sobre los parámetros, $u^{(n)}$ será el coste asociado con el ejemplo o *minibatch*, mientras que $u^{(1)}$ a $u^{(n_i)}$ (los insumos) corresponden a los parámetros del modelo



- Se asume que los nodos en el grafo se han ordenado de tal manera que se puede calcular el resultado de uno tras otro, empezando por $u^{(n_i+1)}$ hasta $u^{(n)}$. Tal y como se ha definido en el siguiente algoritmo, cada nodo $u^{(i)}$ se asocia con una operación $f^{(i)}$ y se calcula evaluando la función $u^{(i)} = f\left(\mathbb{A}^{(i)}\right)$, donde $\mathbb{A}^{(i)}$ es el conjunto de todos los nodos que son padres (denotados por $Pa\left(u^{(i)}\right)$) de $u^{(i)}$ (índices $j < i$ con $j \in Pa\left(u^{(i)}\right)$):

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

- En el algoritmo se puede ver cómo, para un nodo i, se crean conjuntos de los nodos padres de un nodo posterior $n_i + 1, \dots, n$ (los cuales estarán entre 1 y n_i para $n_i + 1$ e incluirá los índices posteriors a medida que se avance)

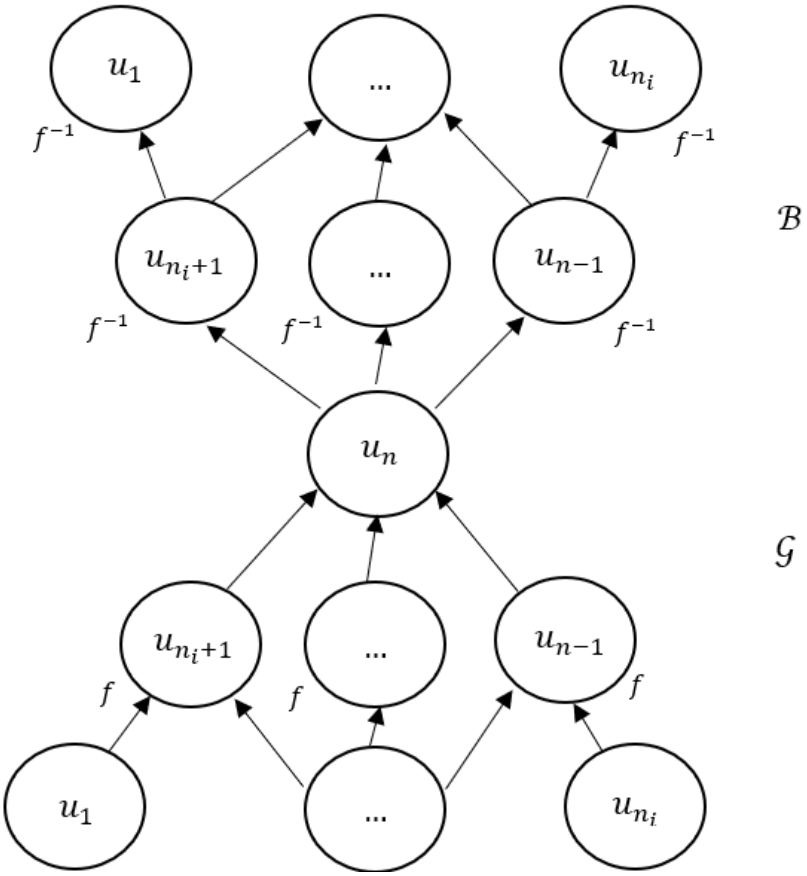


- Este último algoritmo especifica el cálculo de la propagación hacia adelante, que se puede poner en el grafo $\mathcal{G}$. Con tal de realizar el *backprop*, se puede construir un grafo computacional que depende de $\mathcal{G}$ y le añade un conjunto extra de nodos, los cuales forman un subgrafo $\mathcal{B}$ con un nodo por cada nodo de $\mathcal{G}$

  - El cálculo en $\mathcal{B}$ procede de manera exactamente inversa a $\mathcal{G}$ y en cada nodo de $\mathcal{B}$ se calcula la derivada $\partial u^{(n)}/\partial u^{(i)}$ asociada con el nodo del grafo hacia adelante $u^{(i)}$

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

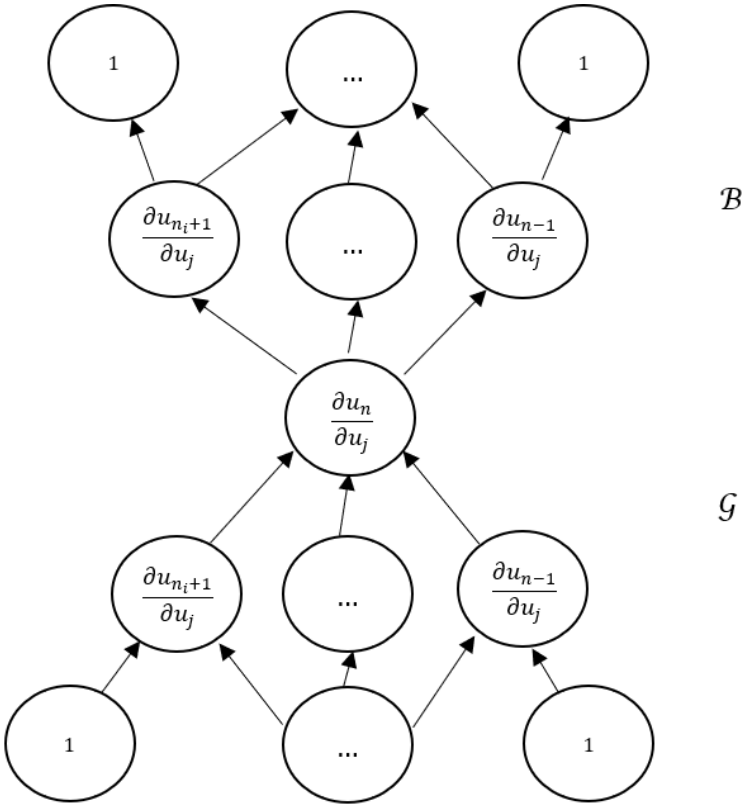
  - El subgrafo $\mathcal{B}$ contiene exactamente un arco para cada arco desde el nodo $u^{(j)}$ al nodo $u^{(i)}$ de $\mathcal{G}$, el cual se asocia con el cálculo de $\partial u^{(i)}/\partial u^{(j)}$. Además, el producto puntual se realiza para cada nodo, entre el gradiente ya calculado con respecto a los nodos $u^{(i)}$ que son hijos de $u^{(j)}$ y el vector que contiene las derivadas parciales $\partial u^{(i)}/\partial u^{(j)}$ para los mismos nodos hijos de $u^{(i)}$



- El algoritmo de *backprop* comienza por ejecutar el algoritmo de propagación hacia adelante anteriormente visto para obtener las activaciones de la red (las funciones f para los diferentes nodos). Después, se inicializa una estructura de datos que guardará las derivadas que se han calculado, y se calculan las derivadas usando la regla de la cadena:

```
grad_table[u^(n)] ← 1
for j = n − 1 down to 1 do
  The next line computes ∂u^(n)/∂u^(j) = Σ_{i:j∈Pa(u^(i))} ∂u^(n)/∂u^(i) ∂u^(i)/∂u^(j) using stored values:
  grad_table[u^(j)] ← Σ_{i:j∈Pa(u^(i))} grad_table[u^(i)] ∂u^(i)/∂u^(j)
end for
return {grad_table[u^(i)] | i = 1, ..., n_i}
```

- Para resumir, la cantidad de cálculos requerida para la realización del *backprop* escala linealmente con el número de arcos de $\mathcal{G}$, donde el cálculo para cada arco corresponde a calcular la derivada parcial (de un nodo con respecto a uno de sus nodos padres) y realizar una multiplicación y una suma. Es posible generalizar este análisis a nodos con valores tensoriales, que solo es una manera de agrupar muchos 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, se ejecuta con orden de un producto jacobiano por nodo en el gráfico

  - Esto se puede ver del hecho de que el *backprop* visita cada arco desde el nodo $u^{(j)}$ hasta el nodo $u^{(i)}$ del grafo exactamente una vez con tal de obtener la derivada parcial $\partial u^{(i)}/\partial u^{(j)}$. Por lo tanto, el *backprop* evita la explosión exponencial en subexpresiones repetidas

  - No obstante, otros algoritmos pueden ser capaces de evitar más subexpresiones al realizar simplificaciones en el grafo computacional, o pueden conservar memoria recalculando en vez de guardando estas subexpresiones

- Con tal de clarificar la definición anterior del cálculo de la propagación hacia atrás, se puede considerar un grafo específico asociado a una red neuronal retroalimentada completamente conectada

  - El algoritmo de la propagación hacia adelante mapea los parámetros de la pérdida supervisada $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ asociada con un solo par de ejemplo de entrenamiento $(\boldsymbol{x}, \boldsymbol{y})$, donde $\hat{\boldsymbol{y}}$ es el resultado de la red neuronal para un insumo $\boldsymbol{x}$

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

- Para obtener el coste total J, la pérdida puede ser sumada a un regularizador $\Omega(\boldsymbol{\theta})$, donde $\boldsymbol{\theta}$ es la matriz de todos los parámetros (ponderaciones y sesgos). En el algoritmo se calculan los valores de las unidades escondidas y, obteniendo un resultado final, se calcula el coste total

- El algoritmo del *backprop* se puede adaptar para el caso de las redes neuronales completamente conectadas de la siguiente manera:

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ **do**
Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
Compute gradients on weights and biases (including the regularization term, where needed):
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for

- El cálculo hacia atrás del algoritmo anterior usa, además del insumo $\boldsymbol{x}$, la variable objetivo $\boldsymbol{y}$. Este cálculo proporciona los gradientes de las activaciones $\boldsymbol{a}^{(k)}$ para cada capa k, comenzando desde la capa de resultados hasta la primera capa escondida

- Para los gradientes calculados, que se pueden interpretar como una indicación de cómo cada capa de resultado debería cambiar para reducir el error, uno puede obtener el gradiente con respecto a los parámetros de cada capa. Los gradientes con respecto a las ponderaciones y sesgos pueden ser inmediatamente usados como parte de la actualización de descenso de gradiente estocástico (realizando la actualización justo después de que los gradientes se hayan calculado) o se

pueden usar con otros métodos de optimización basados en gradientes

Las redes neuronales convolucionales: operación de convolución y agrupación

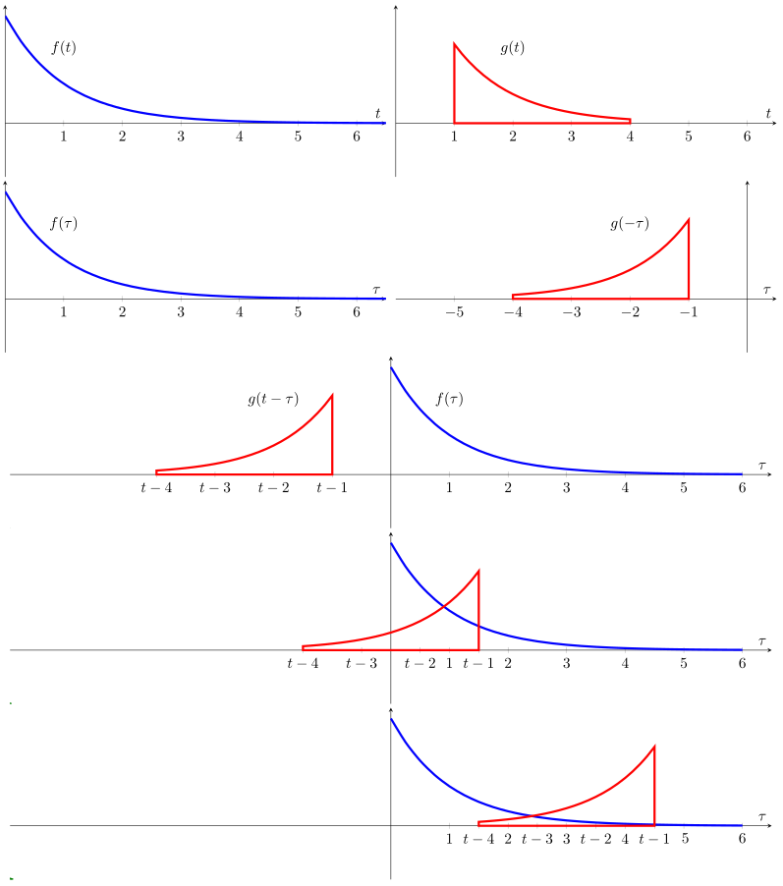
- Las redes neuronales convolucionales o *convolutional neural networks* (CNN) son un tipo de redes especializadas en procesar datos que tiene una topología conocida parecida a la de una reja o *grid*, basadas en la operación lineal de convolución. Las redes convolucionales son simplemente redes neuronales que utilizan convolución en lugar de la multiplicación general de matrices en al menos una de sus capas

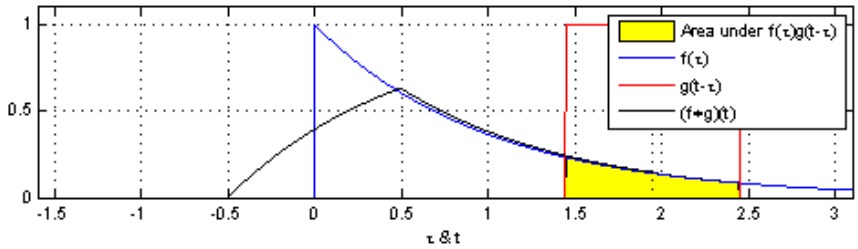
    - En su forma más general, la convolución es una operación sobre dos funciones de un argumento real. Esta operación, en este contexto, se utiliza para promediar conjuntamente diferentes mediciones de manera ponderada

$$s(t) = \int_{-\infty}^{\infty} x(a)w(t-a)\,da$$

        - La convolución se define para cualquier par de funciones para las cuales la integral exista, y es una especie de transformación integral de dos funciones (se transforman en una tercera)

        - Lo que la operación de convolución hace es integrar el producto de dos funciones después de que una de estas (da igual cuál) se refleje sobre el eje vertical y se desplace. La integral se evalúa para todos los valores de desplazamiento, produciendo una función de convolución





- En la terminología de las redes convolucionales, la primera función $x(t)$ se conoce como el insumo y la segunda $w(t)$ se refiere como *kernel*. El resultado de esta operación $s(t)$ se conoce como mapa de características o *feature map*

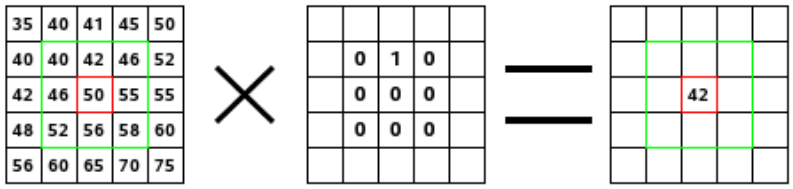
- Normalmente se trabaja con datos computacionales en donde el tiempo está discretizado. Si t solo puede tomar valores enteros y se asume que x y w están definidos solo para valores enteros de t, se puede definir la convolución discreta:

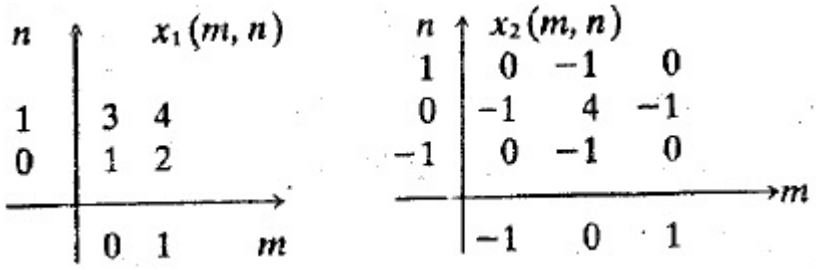
$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

- En aplicaciones de aprendizaje estadístico, el insumo suele ser un arreglo multidimensional de datos (como una imagen) y el *kernel* es usualmente un arreglo multidimensional de parámetros que se adaptan por el

algoritmo de aprendizaje (estos arreglos multidimensionales son tensores)

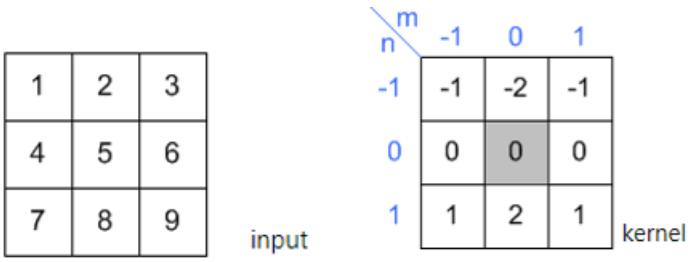
- Como cada elemento del insumo y el *kernel* debe ser guardado explícitamente por separado, normalmente se asumen que estas funciones son cero en todas partes menos por un conjunto finito de puntos (también llamado *padding*) para los que se guardan los valores (la suma pasa de infinita a finita). Los ceros se ponen comenzando desde la izquierda y en dirección descendente, por lo que se suelen indicar las filas y columnas de los elementos finitos concretos de la matriz

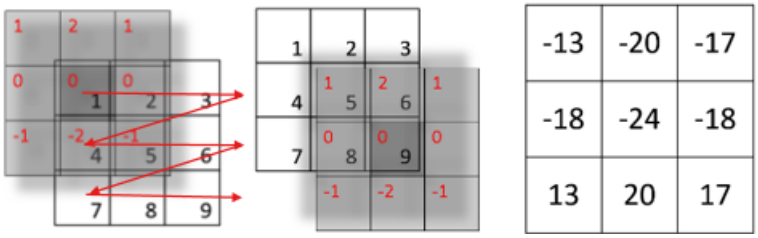




- Normalmente se usan convoluciones sobre más de un eje al mismo tiempo. Si se tiene una imagen bidimensional I como insumo, se querrá usar un *kernel* bidimensional K:

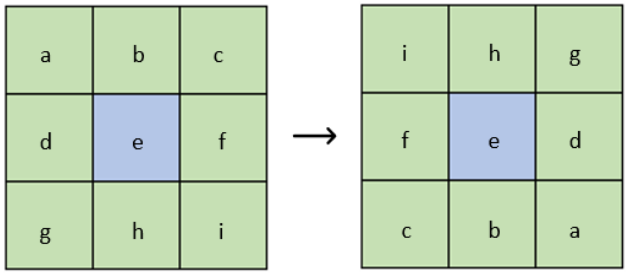
$$S(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$





- La propiedad conmutativa nace de voltear el *kernel* en relación al insumo (las filas se ordenan al revés), en el sentido que cuando m incrementa, el índice de I sube, pero el de K baja. Usualmente se suele usar la fórmula alternativa para la convolución (a través de la propiedad conmutativa), dado que es más fácil de implementar por haber menos variación en el rango de valores válidos de m y n

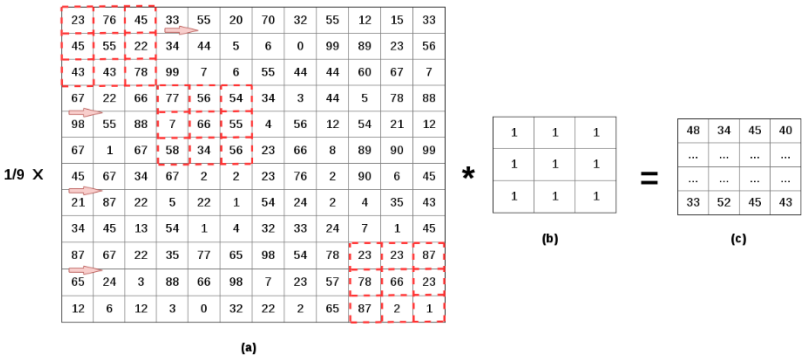
$$S(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$



Kernel Flipped Kernel

- Aunque la propiedad conmutativa es útil para demostraciones, no suele ser una propiedad importante de la implementación de las redes neuronales. Muchas librerías implementan una función relacionada llamada correlación cruzada (llamadas convoluciones en estas librerías), que es lo mismo que la convolución, pero sumando en los índices (no se voltea el *kernel*):

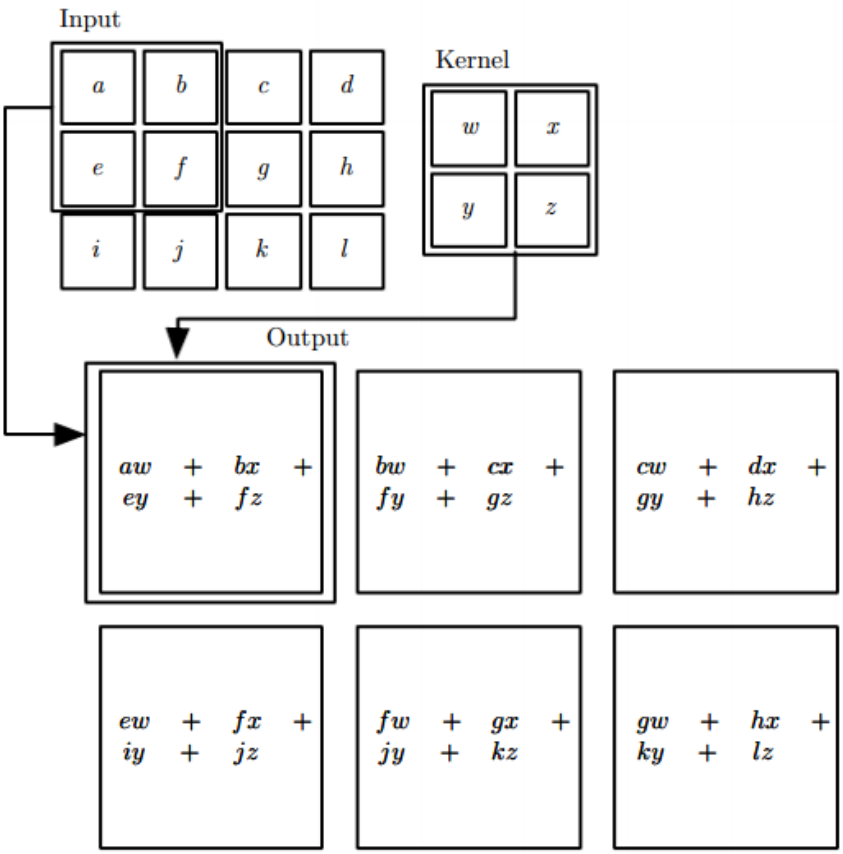
$$S(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$



- En el contexto del aprendizaje automático, el algoritmo de aprendizaje aprende los parámetros apropiados para el *kernel* en cada una de sus celdas, de modo que un algoritmo basado en la

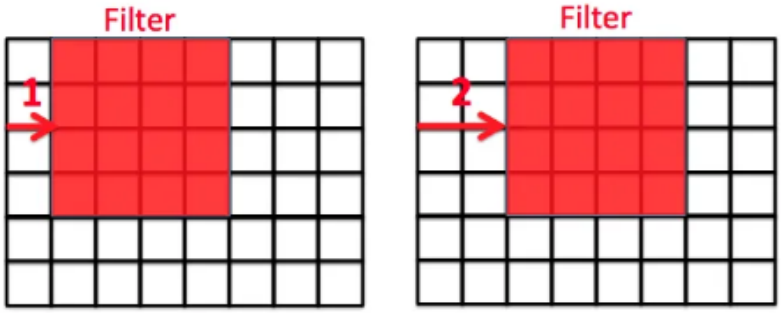
convolución con el *kernel* volteado aprenderá un *kernel* que está volteado relativo al *kernel* aprendido de un algoritmo que lo implementa sin haberle dado la vuelta. Además, es raro que una convolución se use por si sola: normalmente se usa simultáneamente con otras funciones y la combinación de estas funciones no conmuta

- Un esquema del funcionamiento de la convolución cuando no se ha volteado el *kernel* (segundo caso de la convolución). De tener el *kernel* volteado, solo se voltearían los elementos del *kernel*, pero el funcionamiento seguiría siendo el mismo



- Hasta ahora se ha asumido que el filtro o el *kernel* de la convolución se mueve sin dar ningún salto sobre las diferentes regiones de la imagen de insumo. No obstante, esto se puede alterar a través del parámetro de zancada o *stride*

  - El parámetro de zancada o *stride* es un parámetro S que determina que tan lejos el filtro se mueve de una posición a otra. Normalmente se asume que $S = 1$ (no hay saltos), pero se puede fijar para que de saltos



- Las dimensiones no tienen por qué ser las mismas debido al funcionamiento de la operación de convolución (menor a las dimensiones de la imagen)

  - Para calcular las dimensiones del resultado de la convolución, se utilizan las dimensiones del insumo $\boldsymbol{I}$ y del *kernel* $\boldsymbol{K}$ y los parámetros de *stride* S y de *padding* P (el número de filas o columnas para calcular la siguiente fórmula:

$$\dim(\boldsymbol{I}) = m_1 \times n_1 \quad \dim(\boldsymbol{K}) = m_2 \times n_2$$

$$stride = S \qquad padding = P$$

$$\Rightarrow \left(\frac{m_1 - m_2 + 2P}{S} + 1\right) \times \left(\frac{n_1 - n_2 + 2P}{S} + 1\right)$$

- La convolución discreta se puede como una multiplicación de un vector por una matriz, pero en este caso la matriz tiene diversas entradas restringidas a ser iguales a otras entradas

  - Para convoluciones discretas univariantes, cada fila de la matriz se restringe a ser igual que la fila superior pero desplazada por un elemento, mientras que la otra matriz pasa a ser un vector (apilado). Esta matriz que multiplicaría se conoce como matriz de Toeplitz, que es una matriz en donde cada diagonal descendente de izquierda a derecha es constante, y específicamente, la matriz que tendríamos sería una circulante: una matriz cuadrada cuyas filas se componen de los mismos elementos y en cada fila se rota un elemento a la derecha en relación a su fila superior

$$y = h * x = \begin{bmatrix} h_1 & 0 & \cdots & 0 & 0 \\ h_2 & h_1 & & \vdots & \vdots \\ h_3 & h_2 & \cdots & 0 & 0 \\ \vdots & h_3 & \cdots & h_1 & 0 \\ h_{m-1} & \vdots & \ddots & h_2 & h_1 \\ h_m & h_{m-1} & & \vdots & h_2 \\ 0 & h_m & \ddots & h_{m-2} & \vdots \\ 0 & 0 & \cdots & h_{m-1} & h_{m-2} \\ \vdots & \vdots & & h_m & h_{m-1} \\ 0 & 0 & 0 & \cdots & h_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

$$y^T = [\,h_1 \quad h_2 \quad h_3 \quad \cdots \quad h_{m-1} \quad h_m\,] \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n & 0 & 0 & 0 & \cdots & 0 \\ 0 & x_1 & x_2 & x_3 & \cdots & x_n & 0 & 0 & \cdots & 0 \\ 0 & 0 & x_1 & x_2 & x_3 & \cdots & x_n & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & x_1 & \cdots & x_{n-2} & x_{n-1} & x_n & 0 \\ 0 & \cdots & 0 & 0 & 0 & x_1 & \cdots & x_{n-2} & x_{n-1} & x_n \end{bmatrix}.$$

- En dos dimensiones, la convolución de una matriz $\boldsymbol{A}$ y la de una matriz $\boldsymbol{B}$ correspondería a una multiplicación por una matriz circulante de doble bloque $\boldsymbol{X}$ por un vector $\boldsymbol{b}$. La matriz se puede entender como una matriz de submatrices circulantes (que a su vez también es circulante) que se construyen a partir de las filas de $\boldsymbol{A}$ (en orden inverso), y el vector $\boldsymbol{b}$ se construye apilando las filas de la matriz $\boldsymbol{B}$

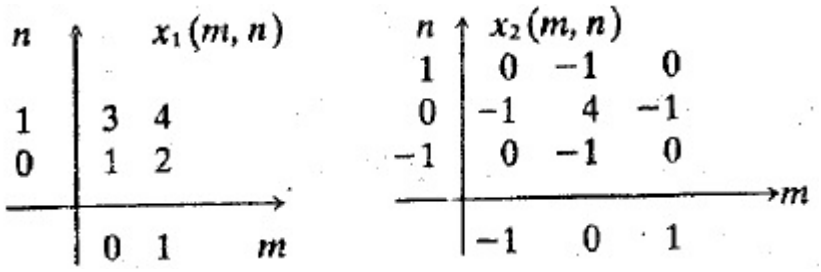
$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \Rightarrow \begin{cases} \boldsymbol{X_0} = circ([0 \;\; -1 \;\; 0 \;\; 0]) \\ \boldsymbol{X_1} = circ([-1 \;\; 4 \;\; -1 \;\; 0]) \\ \boldsymbol{X_2} = circ([0 \;\; -1 \;\; 0 \;\; 0]) \\ \boldsymbol{X_3} = circ([0 \;\; 0 \;\; 0 \;\; 0]) \end{cases}$$

$$\Rightarrow \boldsymbol{X} = \begin{bmatrix} \boldsymbol{X_0} & \boldsymbol{X_3} & \boldsymbol{X_2} & \boldsymbol{X_1} \\ \boldsymbol{X_1} & \boldsymbol{X_0} & \boldsymbol{X_3} & \boldsymbol{X_2} \\ \boldsymbol{X_2} & \boldsymbol{X_1} & \boldsymbol{X_0} & \boldsymbol{X_3} \\ \boldsymbol{X_3} & \boldsymbol{X_2} & \boldsymbol{X_1} & \boldsymbol{X_0} \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \Rightarrow \begin{cases} \boldsymbol{X_0} = circ([0 \;\; -1 \;\; 0 \;\; 0]) \\ \boldsymbol{X_1} = circ([-1 \;\; 4 \;\; -1 \;\; 0]) \\ \boldsymbol{X_2} = circ([0 \;\; -1 \;\; 0 \;\; 0]) \\ \boldsymbol{X_3} = circ([0 \;\; 0 \;\; 0 \;\; 0]) \end{cases}$$

$$\Rightarrow \boldsymbol{X} = \begin{bmatrix} \boldsymbol{X_0} & \boldsymbol{X_3} & \boldsymbol{X_2} & \boldsymbol{X_1} \\ \boldsymbol{X_1} & \boldsymbol{X_0} & \boldsymbol{X_3} & \boldsymbol{X_2} \\ \boldsymbol{X_2} & \boldsymbol{X_1} & \boldsymbol{X_0} & \boldsymbol{X_3} \\ \boldsymbol{X_3} & \boldsymbol{X_2} & \boldsymbol{X_1} & \boldsymbol{X_0} \end{bmatrix}$$

- Es necesario hacer un *padding* de ceros para adaptar las matrices originales al tamaño del resultado de la convolución (se tienen en cuenta las dimensiones del insumo, del *kernel* y el *stride*)



- La convolución apalanca tres ideas importantes que pueden ayudar a un sistema de aprendizaje automático: las interacciones huecas o *sparse*, la compartición de parámetros y las representaciones equivariantes. Además, la convolución proporciona medio para trabajar con insumos de tamaño variable

    - Las capas de redes neuronales tradicionales multiplican una matriz por una matriz de parámetros donde cada uno describe la interacción entre cada unidad de insumo y la unidad de resultado, de modo que cada unidad de insumo interactúa y cada unidad de resultado. No obstante, las redes convolucionales tienen interacciones huecas o *sparse,* y esto ocurre cuando el *kernel* es más pequeño que el insumo

$$\boldsymbol{XW} + \boldsymbol{C} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} + \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \\ c_{41} & c_{42} \end{bmatrix}$$

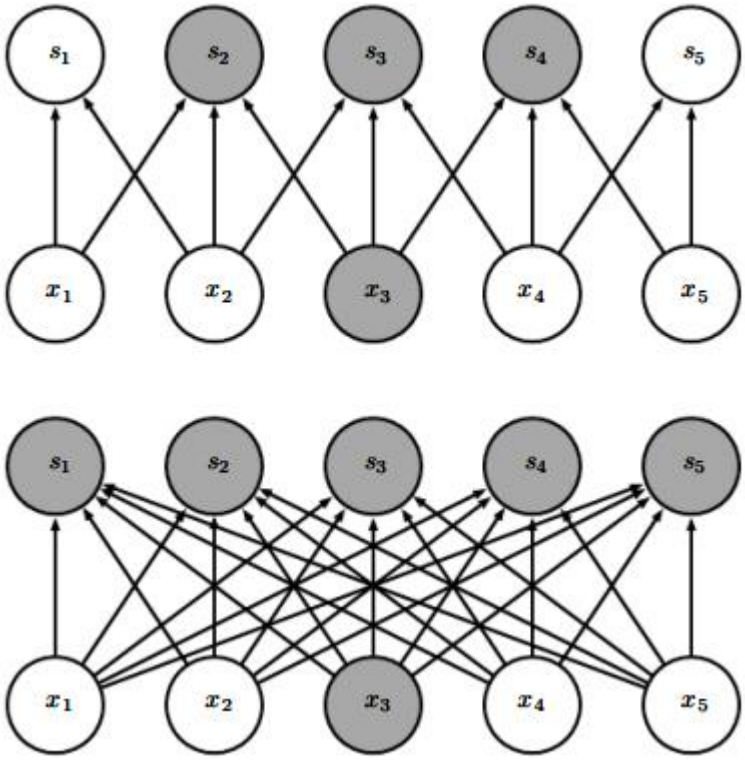
        - Por ejemplo, cuando se procesa una imagen, la imagen de insumo puede tener millones de píxeles, pero se puede detectar características pequeñas como bordes con *kernels* que ocupan solo decenas o cientos de píxeles. Esto significa que se necesitan guardar menos parámetros, que reducen los requerimientos de memoria del modelo y mejora su eficiencia estadística

        - Esto también significa que el cálculo del resultado requiere menos operaciones, y estas mejoras en eficiencia suelen ser muy grandes. Si hay m insumos y n resultados, entonces la multiplicación de matrices requiere $m \times n$ parámetros y los algoritmos usados en la práctica tienen un tiempo de ejecución $O(m \times n)$

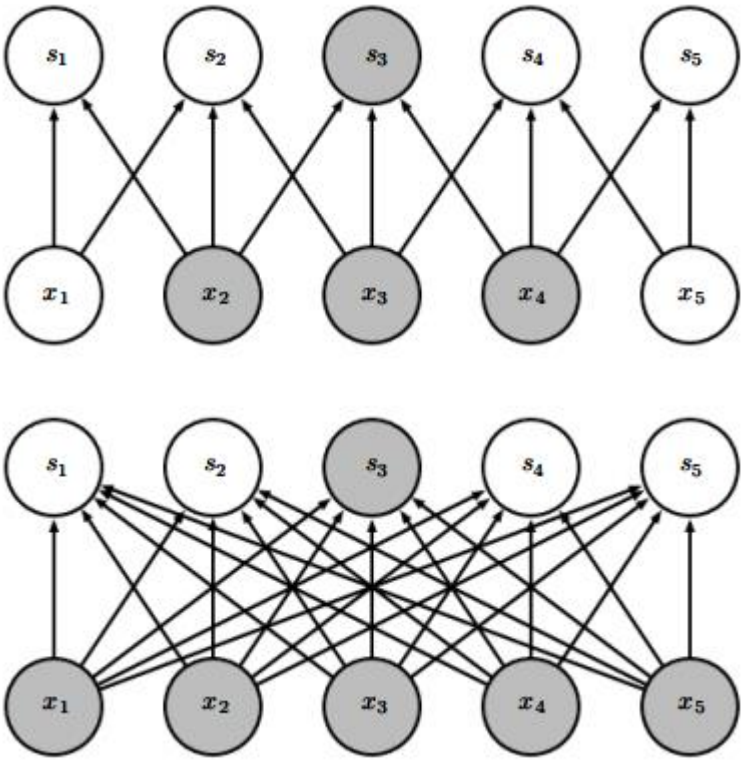
        - Si se limita el número de conexiones que cada insumo puede dar a k, entonces este enfoque de conexión hueca o *sparse* requiere solo $k \times n$ parámetros y tarda $O(k \times n)$. Para muchas aplicaciones prácticas, es posible obtener un buen rendimiento en la tarea de aprendizaje estadístico mientras que se mantiene una k menor en varios órdenes de magnitud a m

    - Los esquemas gráficos de cómo funciona la conectividad hueca o *sparse* permite entender de mejor manera como funciona

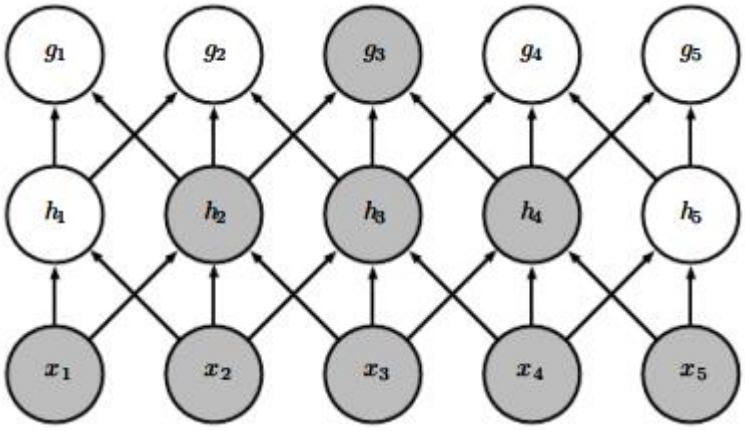
- Cuando se forma la capa $\boldsymbol{s}$ de resultados intermedios o finales por una convolución con un *kernel* de tamaño 3, solo tres resultados se afectan por $\boldsymbol{x}$. En cambio, si se usa una red neuronal completamente conectada (por la multiplicación de matrices), entonces todos los resultados $\boldsymbol{s}$ se ven afectados por x_3



- Cuando se forma el resultado s_3 con la capa de insumo $\boldsymbol{x}$ a través de un *kernel* de tamaño 3, la capa recetiva de s_3 son las tres unidades x_2, x_3 y x_5. En cambio, si se usa una red neuronal completamente conectada (por la multiplicación de matrices), entonces todos los insumos $\boldsymbol{x}$ afectan a s_3



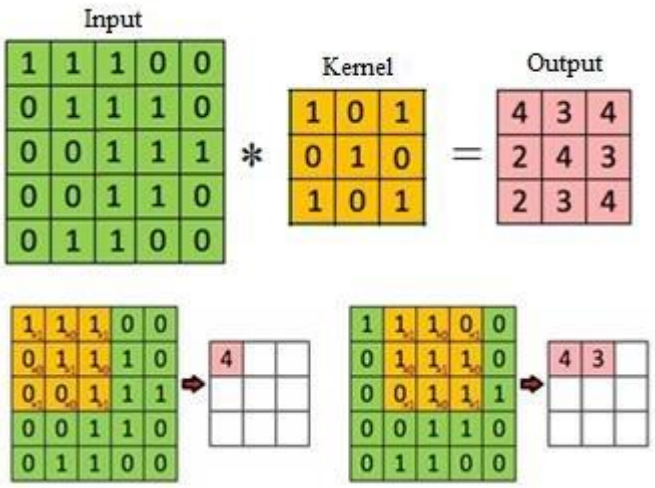
- La capa receptora de las unidades en las capas más profundas (más a la izquierda) de una red convolucional es más grande que el campo receptivo de las unidades en las capas menos profundas (más a la derecha). Este efecto incrementa si la red incluye arquitectura de características como la convolución zancada (o *strided convolution*) o agrupación (o *pooling*), por lo que las conexiones directas en una red convolucional son más huecas, las unidades en las primeras capas pueden estar indirectamente conectadas a toda o a la mayoría de la imagen de insumo



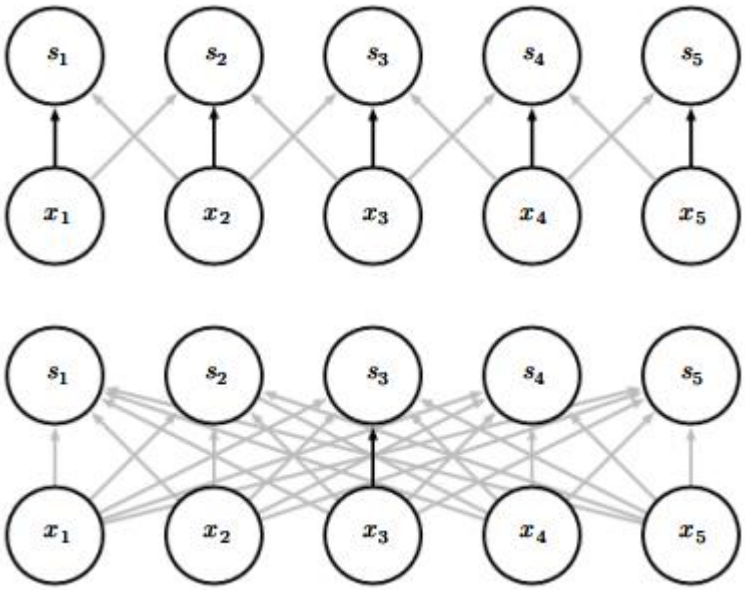
- La compartición de parámetros se refiere al uso del mismo parámetro para más de una función en un modelo. En una red neuronal tradicional, cada elemento de la matriz de ponderaciones se usa exactamente una

vez cuando se calcula el resultado de una capa (se multiplica por un elemento de la capa de insumo y no se vuelve a usar)

- En una red neuronal convolucional, cada miembro del *kernel* se usa en cada posición del insumo (excepto posiblemente en algunos de los píxeles de los bordes, dependiendo de las decisiones de diseño de los bordes). La compartición de parámetros usada por la operación de convolución significa que no tiene que aprender un conjunto separado de parámetros para todas las localizaciones, sino que solo se aprende un conjunto



- Esto no afecta al tiempo de ejecución de la propagación hacia adelante (sigue siendo $O(k \times n)$), pero sí que reduce los requerimientos de memoria del modelo a k parámetros únicamente, siendo k menor en varios órdenes de magnitud a m. Como m y n son normalmente de la misma medida, k es prácticamente insignificante comparado a $m \times n$ y la convolución es muchísimo más eficiente que una multiplicación de matrices usual en términos de requerimientos de memoria y eficiencia estadística



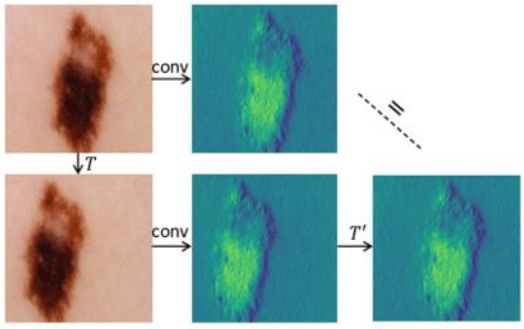
- En el caso de la convolución, la forma particular de compartición paramétrica causa que la capa tenga una propiedad llamada equivarianza a la translación

  - Que una función sea equivariante significa que, si el insumo cambia, el resultado cambia de la misma manera. Específicamente, una función $f(x)$ es equivariante a una función g si $f\big(g(x)\big) = g\big(f(x)\big)$

  - En el caso de la convolución, si se asume que g es cualquier función que traslada o desplaza el insumo, entonces la función de convolución es equivariante a g

  - Por ejemplo, siendo I una función que da brillo a la imagen en coordenadas enteras y g una función que mapea una función de imagen a otra función de imagen, de modo que $I' = g(I)$ es la función de imagen tal que $I'(x, y) = I(x - 1, y)$ (cada pixel desplaza una unidad a la derecha para obtener I'), si se aplica esta transformación a I y se aplica la convolución, el resultado será el mismo que si se aplica la convolución a I' y se aplica g al resultado

$$conv\big(g(I)\big) \quad \Leftrightarrow \quad g\big(conv(I')\big)$$

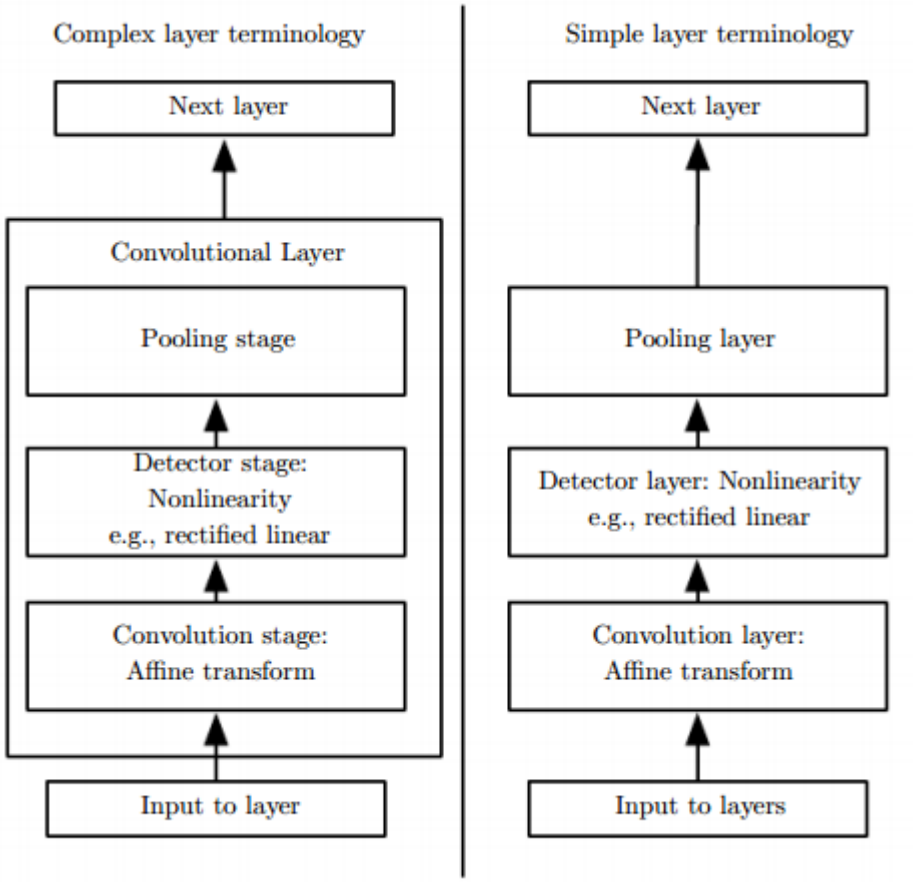


(a) Conv is translation equivariant

- Cuando se procesa una serie temporal, esto significa que la convolución produce un tipo de línea temporal que muestra cuando las diferentes características aparecen en el insumo. Si se mueve un evento a un momento posterior en el tiempo, la misma representación de este aparecerá en el resultado, pero en un momento posterior en el tiempo

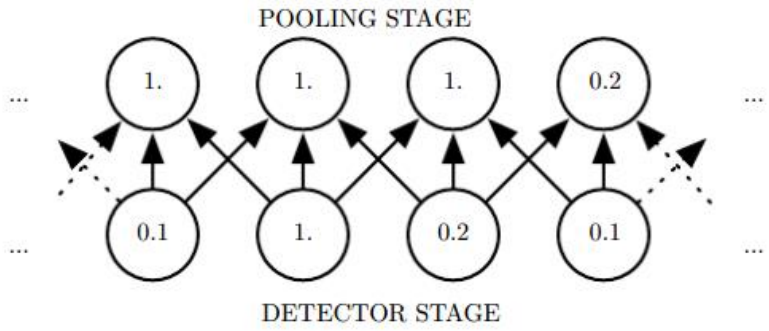
- La convolución no es naturalmente equivariante a otras transformaciones como cambios de escala y la rotación de una imagen. Otros mecanismos son necesarios para manejar este tipo de transformaciones

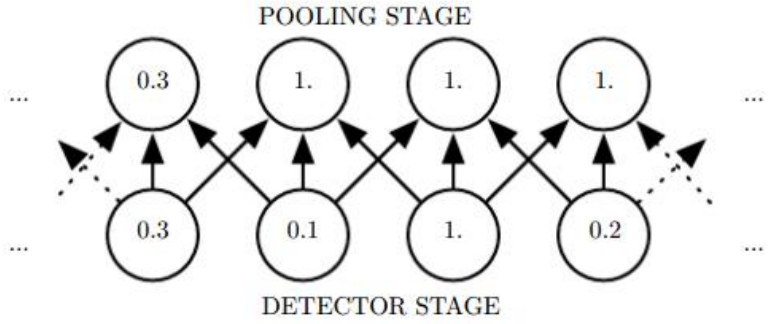
- Una capa típica de una red convolucional consiste de tres fases. En la primera, la capa realiza varias convoluciones en paralelo para producir un conjunto de activaciones lineales; en la segunda fase, cada activación lineal se mueve a través de una función de activación no lineal como una ReLU (fase de detector); y en la tercera fase, se usa una función de agrupación o *pooling* para modificar más el resultado de la capa



- Una función de agrupación o *pooling* reemplaza el resultado de una red en una cierta localización con un estadístico de resumen de los resultados más cercanos
  es
  - Por ejemplo, el *max pooling* es una operación que reporta el máximo resultado dentro de una vecindad rectangular. Otras funciones populares de *pooling* incluyen el promedio de una vecindad rectangular, la norma L^2 para una vecindad triangular, o un promedio ponderado basado en la distancia desde el píxel central

  - En todos los casos, el *pooling* permite hace que la representación sea aproximadamente invariable a pequeñas translaciones en el insumo. La invarianza a la translación significa que, si se puede trasladar el insumo por una pequeña cantidad, los valores de la mayoría de resultados no cambian

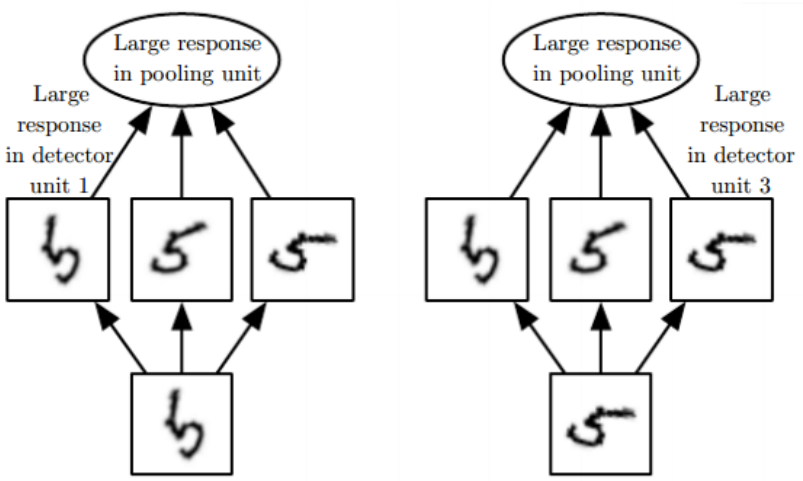




- La invarianza a la translación local puede ser una propiedad muy útil si a uno le importa más que una característica esté presente más que su localización exacta. Por ejemplo, cuando se determina si una imagen contiene una cara, no es necesario saber la localización de los ojos con exactitud, solo saber que hay un ojo en el lado izquierdo y derecho de la cara, aunque en otros contextos si sea necesario saber la localización exacta

- Se puede considerar que el uso del *pooling* agrega una distribución *a priori* infinitamente fuerte de que la función que aprende la capa debe ser invariable para translaciones pequeñas. Cuando esta suposición es correcta, puede mejorar en gran medida la eficiencia estadística de la red

- Hacer *pooling* o agregar regiones espaciales produce invarianza a la translación, pero si se agrega sobre los resultados de convoluciones parametrizadas separadamente, las características pueden aprender a que transformaciones ser invariable



- Una unidad de *pooling* o de agregación que agrega sobre múltiples características que se han aprendido con parámetros separados puede aprender a ser invariable a transformaciones del insumo. Aunque el *max pooling* es invariable a translaciones de manera natural, se usa este enfoque multicanal cuando se quiere ser invariable a otras transformaciones

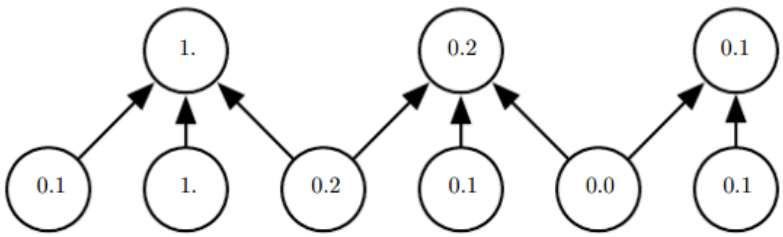
- En el gráfico, un conjunto de tres filtros aprendidos y una unidad de *max pooling* pueden aprender a ser invariables a la rotación: cada uno de los filtros intenta buscar coincidencias entre las posibles direcciones del filtro, causando una gran activación en aquel detector donde haya más coincidencia. La unidad de *max pooling* tendrá una gran activación independientemente de cuál sea el detector con mayor activación, por lo que, aunque se usen dos insumos distintos, uno de los detectores tiene la máxima activación y la unidad de *max pooling* es casi la misma

- Este principio se puede apalancar a través de redes neuronales *maxout* y otras redes convolucionales

- Debido a que el *pooling* resume las respuestas sobre toda una vecindad, es posible usar menos unidades de *pooling* que unidades detectoras a través de reportar estadísticos de resumen para regiones de agregación con un espacio entre ellas de k píxeles (más que solo 1 píxel como suele ser)

    - Esto mejora la eficiencia computacional de la red porque la siguiente capa tiene k veces menos insumos que procesar. Cuando el número de parámetros en la siguiente capa es una función del tamaño del insumo (como cuando la red está totalmente conectada y se basa en multiplicación de matrices), esta reducción en el tamaño del insumo puede provocar una mayor eficiencia estadística y menos requerimientos de memoria para almacenar los parámetros

- Usando *max pooling* para un ancho de agregación (cuantos insumos agrega) de 3 y un *stride* entre agregaciones de dos insumos, se reduce el tamaño de representación por un factor de dos, lo cual reduce la presión computacional y estadística en la siguiente capa. La región de agregación a la derecha de todo tiene un menor tamaño (de dos), pero se tiene que incluir si no se quiere ignorar algunas de las unidades de detección



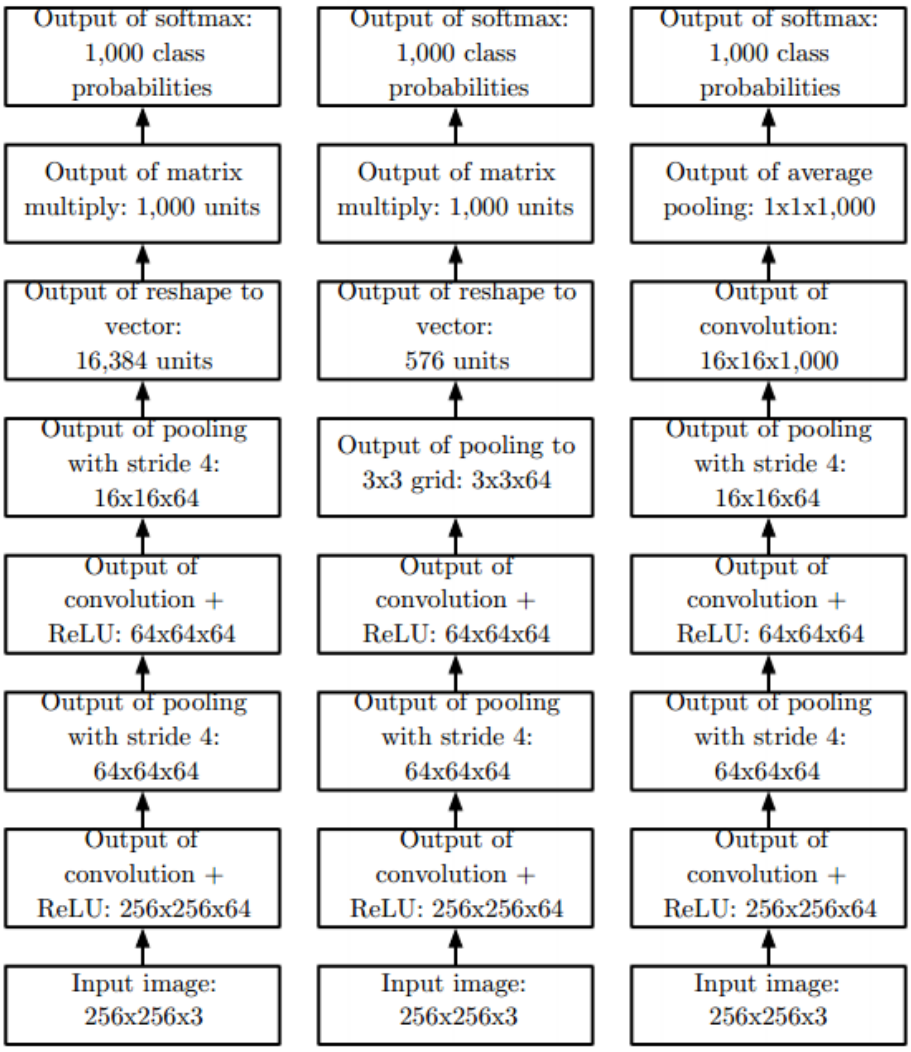
- Algunos avances teóricos permiten usar estas técnicas en diferentes contextos y de diferentes maneras

  - Es posible agregar dinámicamente características, por ejemplo, ejecutando un algoritmo de clusterización en las localizaciones de las características interesantes. Este enfoque da un conjunto diferente de regiones de agregación para cada imagen

  - También es posible aprender una sola estructura de agregación o *pooling* y que se aplica a todas las imágenes

  - El *pooling* puede complicar algunos tipos de arquitecturas neuronales que usan información *top-down*, tales como las máquinas de Boltzmann y los autoencoders

- Algunos ejemplos de arquitecturas neuronales completas para clasificación usando convoluciones y *pooling* se presentan en el siguiente esquema:



- Los *strides* y profundidades específicas pueden variar, dado que estos esquemas son solo de propósito ilustrativo. La mayoría de redes convolucionales involucran mucha cantidad de ramificaciones (a diferencia de las que se ven aquí)

- La arquitectura de la izquierda procesa la imagen, y después de alternar entre convoluciones y *pooling*, el tensor se redimensiona o se aplana a un vector para eliminar las dimensiones más altas. A partir de ahí, el resto de la red tiene la arquitectura de una red neuronal retroalimentada normal para clasificación

- La segunda es una red convolucional que procesa una imagen de tamaño variable, pero aún mantiene una sección completamente conectada. Esta red utiliza una operación de agrupación con grupos de tamaño variable, pero con un número fijo de grupos, para proporcionar un vector de tamaño fijo de 576 unidades a la parte completamente conectada de la red

        - Finalmente, la arquitectura de la derecha es una red convolucional que no tiene ninguna capa de peso completamente conectada, sino una última capa convolucional que genera un mapa de características por clase. Presumiblemente, el modelo aprende un mapa de la probabilidad de que ocurra cada clase en cada ubicación espacial

        - Promediar un mapa de características hasta un solo valor proporciona el argumento para el clasificador *softmax* en la parte superior

Las redes neuronales convolucionales: variantes, datos y algoritmos

- Las redes convolucionales se pueden usar para obtener como resultado un objeto de altas dimensiones estructurado más que una etiqueta de clasificación o un valor para una tarea de regresión

    - Típicamente, este objeto es solo un tensor, emitido por una capa convolucional estándar

        - Por ejemplo, el modelo puede emitir un tensor $\boldsymbol{S}$, en donde S_{ijk} es la probabilidad que el píxel (j, k) del insumo para la red pertenece a la clase i. Esto permite que el modelo etiquete cada píxel en una imagen y permita dibujar máscaras precisas que siguen los contornos de los objetos individuales

    - Un problema que suele ocurrir es que el plano resultante puede ser más pequeño que el plano de insumo. En los tipos de arquitecturas usadas para clasificación de un solo objeto en una imagen, la mayor reducción en las dimensiones espaciales de la red proviene del uso de capas de *pooling* con un *stride* grande

        - Con tal de producir un mapa resultante de un tamaño similar al del insumo, uno puede evitar el *pooling*. Otras estrategias consisten en emitir una tabla de etiquetas de una menor resolución o usar el operador de *pooling* con un *stride* unitario

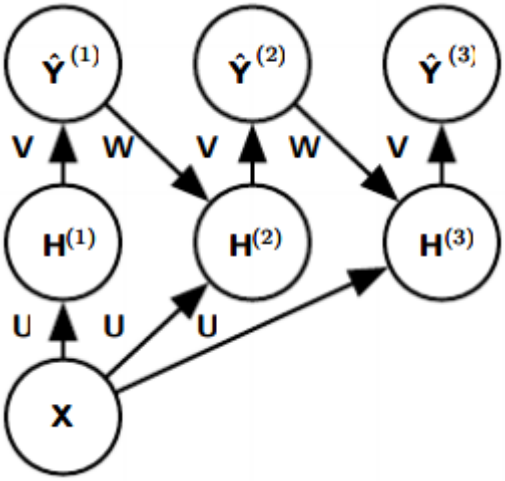
    - Una estrategia para el etiquetado por pixel de las imágenes es producir una elección inicial de las etiquetas de las imágenes y refinar esta selección inicial usando las interacciones entre píxeles vecinos

        - Repitiendo este paso de refinamiento varias veces corresponde a usar las mismas convoluciones en cada fase, compartiendo ponderaciones entre las últimas capas y las más profundas. Esto hace que la secuencia de cálculos realizada por capas

convolucionales sucesivas con ponderaciones compartidas entre capas un tipo particular de red recurrente

- Teniendo una imagen de insumo $\boldsymbol{X}$ con ejes correspondientes a las filas, columnas y canales (rojo, verde y azul), el objetivo es obtener como resultado un tensor de etiquetas $\hat{Y}$ con una distribución de probabilidad sobre las etiquetas de cada pixel y que tiene como ejes las filas de la imagen, sus columnas y las diferentes clases. En vez de dar un resultado $\hat{Y}$ en un solo tiro, la red neuronal recurrente iterativamente refina su estimación $\hat{Y}$ como insumo para crear una nueva estimación, usando los mismos parámetros para cada actualización y refinando la estimación tantas veces como se quieran

- El tensor de *kernels* de convolución $\boldsymbol{U}$ se usan en cada paso para calcular la representación escondida dada la imagen de insumo, y el tensor de *kernel* $\boldsymbol{V}$ se usa para producir una estimación de las etiquetas dados los valores escondidos. En todos los pasos menos el primero, los *kernels* $\boldsymbol{W}$ se convolucionan sobre $\hat{Y}$ para proporcionar el insumo a la capa escondida (en el perimer paso el término se reemplaza por 0)



- Una vez se hace la predicción para cada pixel, se pueden usar varios métodos para procesar más estas predicciones con tal de obtener una segmentación de la imagen en regiones

  - La idea general es asumir que grandes grupos de píxeles contiguos tienden a estar asociados con la misma etiqueta

  - Los modelos gráficos pueden describir las relaciones probabilísticas entre píxeles vecinos. Alternativamente, la red convolucional se puede entrenar para maximizar una aproximación del objetivo de entrenamiento del modelo gráfico

- Los datos usados en una red convolucional normalmente consisten de varios canales, donde cada canal es una observación de una cantidad diferente en algún punto en el espacio o tiempo

	Single channel	Multi-channel
1-D	Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step.	Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a "skeleton" over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character's skeleton. Each channel in the data we feed to the convolutional model represents the angle about one axis of one joint.
2-D	Audio data that has been preprocessed with a Fourier transform: We can transform the audio waveform into a 2D tensor with different rows corresponding to different frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time. Using convolution across the frequency axis makes the model equivariant to frequency, so that the same melody played in a different octave produces the same representation but at a different height in the network's output.	Color image data: One channel contains the red pixels, one the green pixels, and one the blue pixels. The convolution kernel moves over both the horizontal and vertical axes of the image, conferring translation equivariance in both directions.
3-D	Volumetric data: A common source of this kind of data is medical imaging technology, such as CT scans.	Color video data: One axis corresponds to time, one to the height of the video frame, and one to the width of the video frame.

  - Una ventaja de usar redes convolucionales es que puede procesar insumos de dimensiones espaciales variantes (no como antes, en donde todos los ejemplos tienen las mismas dimensiones espaciales)

    - Estos tipos de insumo no pueden representarse por redes basadas en multiplicaciones de matriz. Esto proporciona una

razón para usar redes convolucionales, aunque el coste computacional y el sobreajuste no sean problemas significativos

- Considerando una colección de imágenes con diferentes dimensiones, no es claro como modelar estos insumos con una matriz de ponderaciones de tamaño fijo. No obstante, la convolución se puede aplicar de manera directa: el *kernel* se aplica un número diferente de veces dependiendo del tamaño del insumo, y el resultado de la operación de convolución se escalará de manera acorde

- La convolución puede verse como una multiplicación de matrices; el mismo *kernel* de convolución induce un tamaño diferente de matriz circulante de doble bloque para cada tamaño de entrada

    - A veces el resultado o el insumo puede tener tamaño variable, por ejemplo, si se quiere asignar una etiqueta de clase para cada pixel del insumo. En este caso, no es necesario ningún trabajo de diseño

    - En otro caso, la red debe producir algunos resultados de tamaño fijo, por ejemplo, si se quiere asignar una sola etiqueta de clase a toda la imagen. En este caso se tienen que hacer pasos de diseño adicionales, como insertar una capa de *pooling* cuyas regiones de *pooling* escalan de manera proporcional al tamaño del insuo, con tal de mantener un número fijo de resultados de la agregación

- El uso de las convoluciones para procesar insumos variables solo tiene sentido para insumos con tamaño variable que contengan cantidades variables de observación del mismo tipo de cosa (diferentes longitudes de grabaciones sobre el tiempo, diferentes anchos de las observaciones en el espacio, etc.)

    - La convolución no tiene sentido si el insumo tiene un tamaño variable debido a que se pueden incluir opcionalmente diferentes tipos de observaciones

    - Por ejemplo, si se está procesando aplicaciones universitarias, y otras características consisten de notas y exámenes estandarizados, pero no todos los aplicantes hacen estos últimos, entonces no tiene sentido hacer una convolución con las mismas ponderaciones sombre ambos tipos de características correspondientes a notas y exámenes estandarizados

Las redes neuronales secuenciales: redes recurrentes

- Las redes neuronales recurrentes (RNNs) son una familia de redes neuronales para procesar datos secuenciales. Igual que una red neuronal convolucional, las redes recurrentes se especializan en procesar una secuencia de valores $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$ y que pueden escalar a secuencias más grandes de lo que sería posible si no se tiene una arquitectura recurrente

    - Para ir de las redes multicapa a las redes recurrentes, se necesita aprovecharse de la compartición de parámetros a través de diferentes partes del modelo

        - La compartición de parámetros hace posible extender y aplicar el modelo a ejemplos de diferentes formas (diferentes longitudes) y generalizar entre ellas

        - Si se tuvieran parámetros separados para cada valor del índice temporal, no se podría generalizar la longitud de secuencias no vistas durante el entrenamiento, ni tampoco compartir la fortaleza estadística entre diferentes longitudes de secuencias y diferentes posiciones en el tiempo

    - Esta compartición es particularmente importante cuando una pieza de información específica puede ocurrir en múltiples posiciones dentro de una secuencia (se quiere que la posición en la secuencia no sea relevante, sino la pieza de información que realmente lo es)

        - Si se quisiera procesar esta secuencia con una red neuronal retroalimentada totalmente conectada, entonces, como se tendrían parámetros separados para cada insumo, la red necesitaría aprender todos los patrones o reglas separadamente en cada posición de la secuencia

        - En comparación, una red neuronal recurrente comparte las mismas ponderaciones a través de varios pasos temporales

    - Una idea relacionada es usar una red convolucional a través de una secuencia temporal unidimensional. La operación de convolución permite que la red aprenda parámetros a través del tiempo, pero es simple (solo tiene una capa)

        - El resultado de la convolución es una secuencia donde cada miembro del resultado es una función de un pequeño número de miembros vecinos del insumo

        - La idea de la compartición de parámetros se manifiesta en la aplicación del mismo *kernel* convolucional en cada momento del tiempo. Las redes recurrentes comparten parámetros de otra manera: cada miembro del resultado es una función de los

miembros previos del resultado, y cada resultado se produce usando la misma regla de actualización que se aplicó a los resultados anteriores

- Esta formulación recurrente resulta en la compartición de parámetros a través de un grafo computacional muy profundo

- Para la simplicidad de la exposición, se refiere a las RNNs como operando en una secuencia que contiene vectores $\boldsymbol{x}^{(t)}$ con el índice de paso del tiempo $t \in [1, \tau]$

  - En la práctica, las redes recurrentes operan en *minibatches* de estas secuencias, con una longitud de secuencia diferente τ para cada miembro del *minibatch*. Se suelen omitir los índices del *minibatch* para ahorrar notación

  - Las RNN se pueden aplicar en dos dimensiones a lo largo de datos espaciales como imágenes, o a datos que involucran tiempo, la red puede tener conexiones que van hacia atrás en el tiempo, siempre que la secuencia se observe antes de que se proporcione a la red

- Como un grafo computacional formaliza la estructura de un conjunto de cálculos, se usa este para explicar la idea de despliegue de un cálculo recursivo o recurrente en un grafo computacional que tiene una estructura repetitiva, típicamente correspondiendo a un conjunto de eventos

  - Desplegar un grafo computacional resulta en la compartición de parámetros a lo largo de la estructura de la red neuronal. Considerando la clásica forma de un sistema dinámico, donde $\boldsymbol{s}(t)$ es el estado del sistema, se puede ver como la ecuación es recurrente porque $\boldsymbol{s}^{(t)}$ depende de $\boldsymbol{s}^{(t-1)}$

$$\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}\right)$$

  - Para un número finito de pasos temporales τ, el grafo se puede desplegar aplicando la definición $\tau - 1$ veces. Si se despliega la ecuación para $\tau = 3$ pasos de tiempo, se obtienen los siguientes resultados:

$$\boldsymbol{s}^{(3)} = f\left(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}\right) = f\left(f\left(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right)$$

  - Desplegando la ecuación a través de la definición recursiva permite tener una expresión que no requiere recurrencia. Esta expresión ahora se puede representar por un grafo computacional dirigido acíclico

- Otro ejemplo que se puede considerar es el de un sistema dinámico que está dirigido por una señal externa $\boldsymbol{x}^{(t)}$, donde se puede ver que el estado ahora contiene información sobre toda la secuencia pasada

$$\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

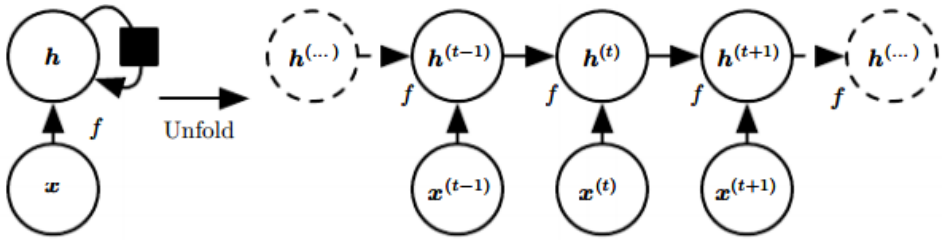
- Las redes neuronales recurrentes se pueden construir de diferentes maneras: igual que casi cualquier función se puede considerar una red neuronal retroalimentada, cualquier función que involucre recurrencia puede ser considerada una red neuronal recurrente

  - Muchas redes neuronales recurrentes pueden usar la siguiente ecuación o una similar para definir los valores de sus unidades escondidas:

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

  - Las redes neuronales típicas añaden características arquitectónicas adicionales tales como las capas de resultados que leen la información del estado $\boldsymbol{h}$ para hacer predicciones

- Las RNN se pueden entender gráficamente de dos maneras diferentes: a través de un diagrama de nodos (grafo computacional) o a través de un grado computacional desplegado



  - Una manera de dibujar la red es a través de un diagrama que contiene un nodo por cada componente que pueda existir en la implementación física del modelo. De este modo, la red define un circuito que opera en tiempo real, con partes físicas cuyos estados corrientes pueden influenciar su estado futuro, como a la izquierda de la imagen

  - El cuadrado negro indica que una interacción toma lugar con un retraso de un solo paso temporal, del estado en el tiempo t al tiempo $t + 1$

  - La otra manera de dibujar la red es a través de un grafo computacional desplegado, en el que cada componente se representa por varias variables diferentes, con una variable por

paso temporal, representando el estado del componente en ese punto en el tiempo. Cada variable para cada paso temporal se dibuja en un nodo separado del grado computacional, como a la derecha de la imagen

- A lo que se le llama despliegue es la operación que mapea un circuito como el de la izquierda de la imagen al de la derecha. El grafo computacional resultante ahora dependerá de la longitud de la secuencia

- Tanto el grafo recurrente como el grafo desplegado tienen usos útiles: el grafo recurrente es pequeño y explicativo, mientras que el gráfico desplegado proporciona una descripción explícita de qué cálculos realizar, además de ilustrar la idea del flujo de la información hacia adelante (calculando resultados y pérdidas) y hacia atrás (calculando gradientes) en el tiempo

- El proceso de despliegue tiene dos ventajas: independientemente de la longitud de la secuencia, el modelo aprendido siempre tiene el mismo tamaño de insumos (se especifica en términos de la transición de un estado a otro estado, y no por el número de variables) y es posible usar la misma función de transición f con los mismos parámetros en cada paso temporal

  - Se puede representar la recurrencia desplegada después de t pasos con una función $g^{(t)}$:

    $$\boldsymbol{h}^{(t)} = g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right) = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$

  - La función $g^{(t)}$ coge toda la secuencia pasada como insumo y produce el estado actual, pero la estructura de recurrencia desplegada permite factorizar $g^{(t)}$ en diferentes aplicaciones repetidas de una función f

  - Estos dos factores hacen posible que se pueda aprender un solo modelo f que opera en todos los pasos temporales y todas las longitudes de las secuencias, en vez de necesitar aprender un modelo separado $g^{(t)}$ para todos los posibles pasos temporales

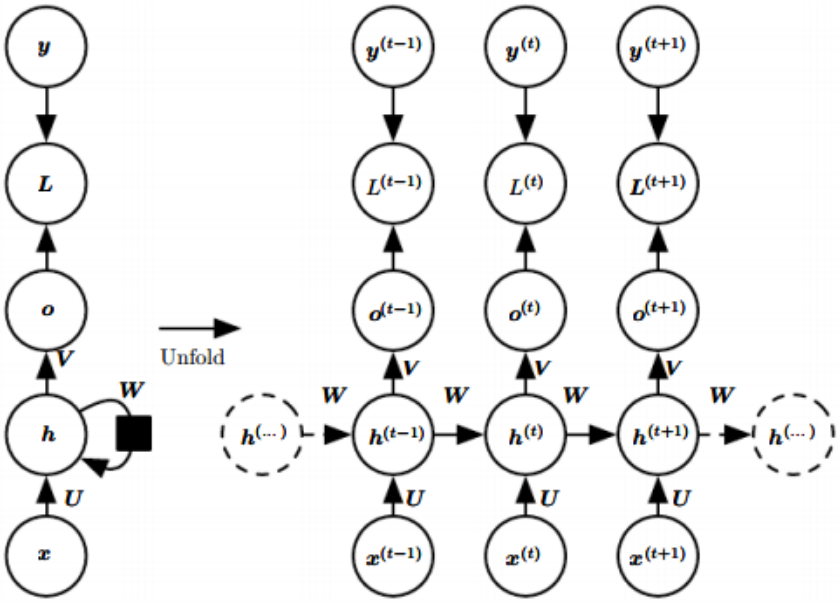
  - Aprender un solo modelo compartido permite la generalización a longitudes de las secuencias que no han aparecido en el conjunto de entrenamiento, y permite que el modelo se estime con menos ejemplos de entrenamiento de los que se requeriría sin compartición de parámetros

Las redes neuronales secuenciales: arquitecturas recurrentes

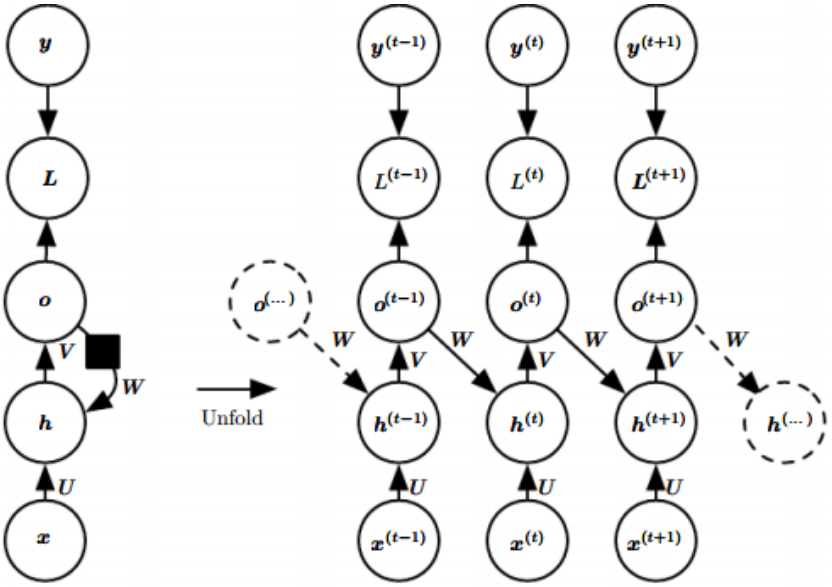
- A partir de las ideas del despliegue de grados y de la compartición de parámetros, vistas anteriormente, se pueden diseñar una gran variedad de redes neuronales recurrentes

    - Algunos ejemplos de patrones de diseños importantes para redes neuronales recurrentes son los siguientes:

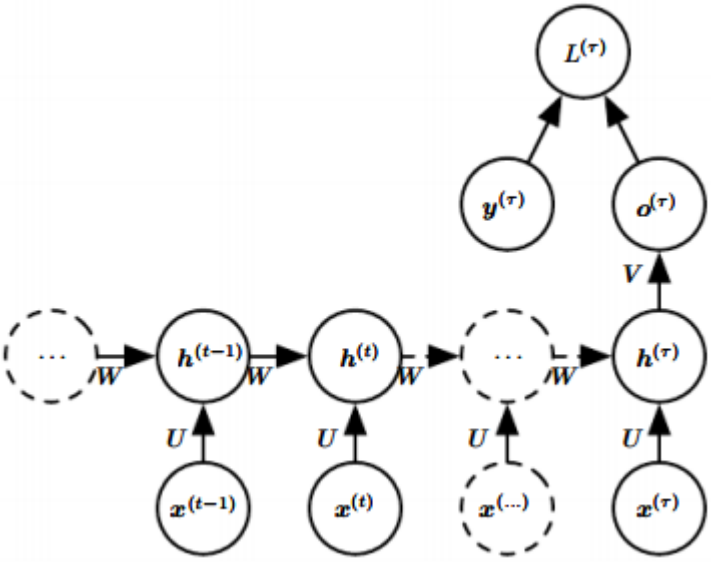
        - Las redes recurrentes que producen un insumo en cada momento del tiempo y tienen conexiones recurrentes entre unidades escondidas. En este caso, $\boldsymbol{o}$ son los resultados del modelo, $\boldsymbol{L}$ son los valores de la función de pérdida y $\boldsymbol{y}$ son los ejemplos de entrenamiento para la variable objetivo



        - Las redes recurrentes que producen un resultado en cada momento del tiempo y solo tienen conexiones recurrentes del resultado en un momento a la unidad escondida del siguiente momento. En este caso, $\boldsymbol{o}$ son los resultados del modelo, $\boldsymbol{L}$ son los valores de la función de pérdida y $\boldsymbol{y}$ son los ejemplos de entrenamiento para la variable objetivo



- Las redes recurrentes con conexiones recurrentes entre las unidades escondidas, que leen una secuencia entera y producen un solo resultado. En este caso, $\boldsymbol{o}$ son los resultados del modelo, $\boldsymbol{L}$ son los valores de la función de pérdida y $\boldsymbol{y}$ son los ejemplos de entrenamiento para la variable objetivo



- La red neuronal recurrente con conexiones recurrentes entre unidades escondidas y que producen un resultado en cada paso y la siguiente ecuación es universal, en el sentido que cualquier función calculable por una máquina de Turing puede ser calculada por una red neuronal de tamaño finito:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}$$

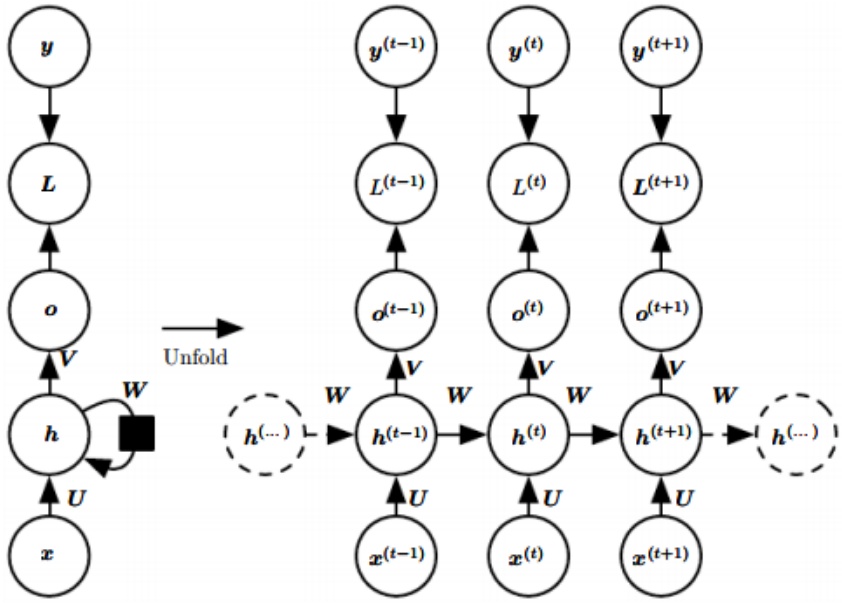
- El resultado se puede leer desde la RNN después de un número de pasos temporales que es asintóticamente lineal en el número

de pasos usados por una máquina de Turing y asintóticamente lineal en la longitud del insumo

- Las funciones calculables por una máquina de Turing son discretas, de modo que los resultados se interpretan como la implementación exacta de la función, no una aproximación. La RNN, cuando se usa como una máquina de Turing, toma una secuencia binaria como un insumo y sus resultados se deben discretizar para proporcionar un resultado binario

- El insumo de una máquina de Turing es una especificación de la función a ser calculada, de modo que la misma red que simula esta máquina de Turing es suficiente para todos los problemas. La RNN teórica usada para la demostración puede simular un *stack* no acotado a través de representar sus activaciones y ponderaciones con números racionales de precisión no acotada

- Ahora es posible desarrollar las ecuaciones de la RNN que se está estudiando, asumiendo una función de activación tangente hiperbólica y que el resultado es discreto



- Una manera natural de representar variables discretas es pensar en el resultado $\boldsymbol{o}$ como un vector que da las probabilidades logarítmicas no normalizadas de cada posible valor de la variable discreta. Después, se puede aplicar la operación *softmax* como un paso de post-procesamiento para obtener el vector $\hat{\boldsymbol{y}}$ de probabilidades normalizadas sobre el resultado

- La propagación hacia adelante comienza con la especificación del estado inicial $\boldsymbol{h}^{(0)}$, y entonces, para cada paso temporal desde $t = 1$ a $t = \tau$, se aplican las siguientes ecuaciones de

actualización, donde $\boldsymbol{b}$ y $\boldsymbol{c}$ son sesgos y $\boldsymbol{U}$, $\boldsymbol{V}$ y $\boldsymbol{W}$ son las matrices de ponderaciones:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}$$

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

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}$$

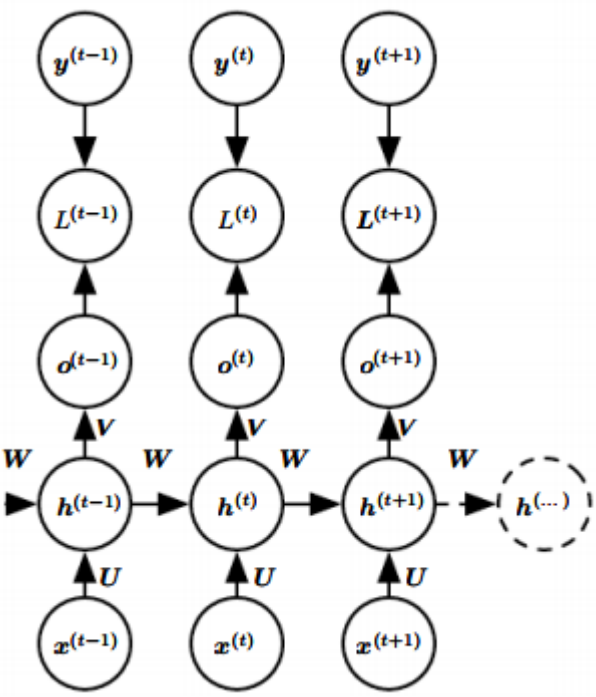
$$\hat{\boldsymbol{y}}^{(t)} = softmax\left(\boldsymbol{o}^{(t)}\right)$$

- La pérdida total para una secuencia dada de valores de $\boldsymbol{x}$ emparejados con una secuencia de valores $\boldsymbol{y}$ sería la suma de las pérdidas sobre todos los pasos temporales. Por ejemplo, si $L^{(t)}$ es la verosimilitud logarítmica negativa de $y^{(t)}$ dado $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}$, entonces se obtiene el siguiente resultado, donde $p_{model}\left(y^{(t)} \middle| \{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\}\right)$ se obtiene leyendo la entrada para $y^{(t)}$ para el vector de resultados $\hat{\boldsymbol{y}}^{(t)}$ del modelo:

$$L\left(\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)}\}\right) = \sum_t L^{(t)} =$$

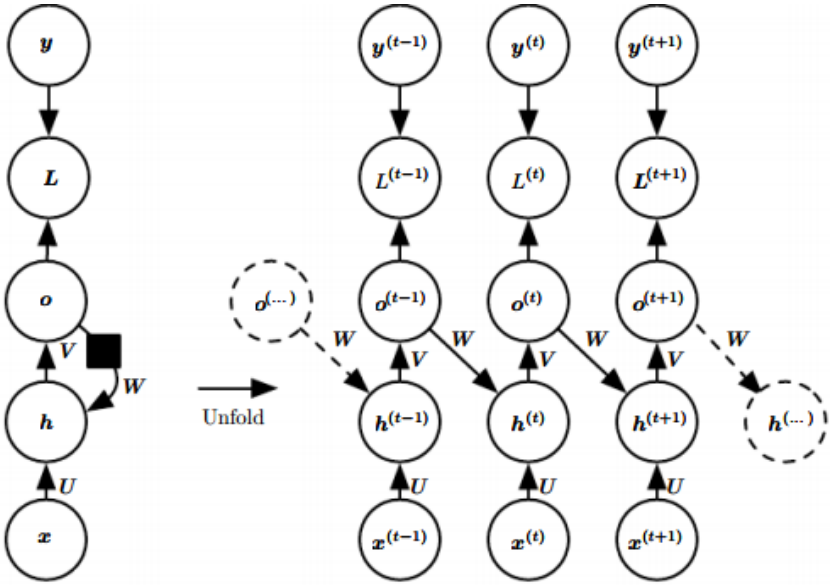
$$= -\sum_t \log\left[p_{model}\left(y^{(t)} \middle| \{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\}\right)\right]$$

- Calcular el gradiente de esta función de pérdida con respecto a sus parámetros es costoso, dado que requiere realizar la propagación hacia adelante moviéndose de la izquierda a la derecha, y después hacer el *backprop* de la derecha a la izquierda. El algoritmo de *backprop* aplicado a un grafo desplegado con coste $O(\tau)$ se conoce como *backprop through time* (BPTT)



- El tiempo de ejecución es $O(\tau)$ y no se puede reducir por paralelización porque la propagación hacia adelante es inherentemente secuencial: cada paso temporal solo se puede calcular después del anterior. Los estados calculados en el paso de la propagación hacia adelante se deben guardar hasta que se usen durante el *backprop*, con un coste de memoria también $O(\tau)$

- La red con conexiones recurrentes solo desde el resultado en un paso temporal a las unidades escondidas en el siguiente paso temporal es estrictamente menos poderosa porque no tiene conexiones recurrentes de unidad escondida a unidad escondida (no pueden simular una máquina de Turing universal)



- Como esta red no tiene recurrencia de unidad escondida a unidad escondida, requiere que las unidades de resultado capturen toda la información sobre el pasado que la red usará para predecir el futuro. Antes, la información del pasado podía transmitirse en el futuro a través de $\boldsymbol{h}$, pero ahora solo se puede pasar $\boldsymbol{o}$

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{o}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}$$

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

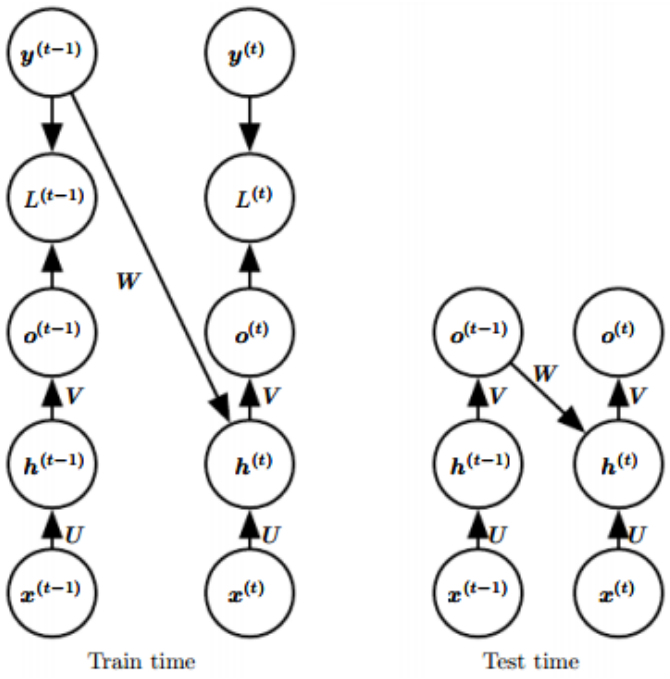
$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}$$

  - Como las unidades de resultado se entrenan específicamente para coincidir con la variable objetivo en el conjunto de entrenamiento, es poco probable que capturen la información necesaria sobre el historial pasado del insumo, a menos que el usuario sea capaz de describir el estado entero del sistema y lo proporcione como parte de los ejemplos objetivo en el conjunto de entrenamiento

  - La ventaja de eliminar la recurrencia entre unidades escondidas es que, para cualquier función de pérdida basada en comparar la predicción en el momento t al objetivo de entrenamiento en t, todos los pasos temporales se pueden desacoplar. Esto hace posible la paralelización, permitiendo que el gradiente de cada paso t se calcule aisladamente

  - No hay necesidad de calcular el resultado del primer momento primero, dado que el conjunto de entrenamiento proporciona el valor ideal para ese resultado

- Los modelos que tienen conexiones recurrentes desde sus resultados a sus unidades escondidas pueden entrenarse con la técnica de *teacher forcing*, el cual es un procedimiento que emerge del criterio de máxima verosimilitud y que, durante el entrenamiento, el modelo recibe el verdadero resultado $y^{(t)}$ es un insumo en el momento $t + 1$



- Esto se puede ver mejor considerando el criterio de máxima verosimilitud para dos pasos:

$$\log\left[p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \middle| \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)\right] =$$

$$= \log\left[p\left(\boldsymbol{y}^{(2)} \middle| \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) p\left(\boldsymbol{y}^{(1)} \middle| \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)\right] =$$

$$= \log\left[p\left(\boldsymbol{y}^{(2)} \middle| \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)\right] + \log\left[p\left(\boldsymbol{y}^{(1)} \middle| \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)\right]$$

- En el ejemplo, se ve que en $t = 2$ el modelo se entrena para maximizar la probabilidad condicional de $\boldsymbol{y}^{(2)}$ dada la secuencia $\boldsymbol{x}$ hasta ese momento y el valor previo $\boldsymbol{y}^{(1)}$ del conjunto de entrenamiento. Por lo tanto, la máxima verosimilitud especifica que, durante el entrenamiento, más que proporcionar su resultado, estas conexiones deberían ser proporcionadas los valores objetivo especificando cuál debería ser el resultado correcto

- Este método no introduce ningún sesgo porque los insumos son los valores verdaderos para el paso de tiempo anterior, de modo que no hay posibilidad de que se transfiera información del futuro y que memorice así la secuencia entera de valores verdaderos. Además, como se usan los verdaderos valores, no se extienden los posibles errores de predicción del modelo en un paso del tiempo a través del tiempo

- Aunque originalmente se ha motivado el *teacher forcing* para evitar el BPTT en modelos que no tienen conexiones recurrentes entre unidades escondidas, este se puede aplicar a modelos con este tipo de conexiones

siempre que también tengan conexiones del resultado en un paso temporal a la unidad escondida en el siguiente paso

- Tan pronto como las unidades escondidas se vuelven una función de las unidades escondidas de los pasos temporales anteriores, el algoritmo BPTT es necesario, y algunos modelos se pueden entrenar usando *teacher forcing* y BPTT (pero no se pueden paralelizar)

- La desventaja del *teacher forcing* estricto ocurre cuando la red se va a usar después en un modo de bucle abierto, con los resultados de la red (o muestras de la distribución de resultados) proporcionándose otra vez como insumo. En este caso, el tipo de insumos que la red ve durante el entrenamiento puede ser muy diferente al tipo de insumos que ve durante la comprobación

  - Una manera de mitigar este problema es entrenar con insumos del *teacher forcing* y con insumos libres, por ejemplo, a través de predecir la variable objetivo correctamente en una serie de pasos en el futuro a través de los caminos recurrentes desplegados de resultado a insumo. Se despliegan más pasos que el número de pasos que se conocen τ (donde hay datos del objetivo que se pueden usar) para que se puedan usar los resultados de la red en vez de los reales para los pasos adicionales

  - De esta manera, la red puede aprender a tener en cuenta las condiciones de insumo (como las que genera ella misma en el modo de funcionamiento libre) que no se ven durante el entrenamiento y cómo mapear el estado hacia uno que hará que la red genere resultados adecuados después de unos cuantos pasos

  - Otro enfoque para mitigar la diferencia entre insumos en el proceso de entrenamiento y los insumos que se ven en la comprobación es escoger aleatoriamente valores generados o datos reales como insumo. Este enfoque explota una estrategia de aprendizaje de currículo para gradualmente usar más de los valores generados como insumos

- Calcular el gradiente a través de una red neuronal recurrente es sencillo, dado que uno debe simplemente aplicar el algoritmo de *backprop* generalizado al grafo computacional desplegado

  - Para ganar una intuición de cómo se comporta el algoritmo del BPTT, se proporciona un ejemplo de cómo calcular los gradientes por BPTT para las ecuaciones de la RNN anteriores

- En esta derivación, se asume que los resultados $\boldsymbol{o}^{(t)}$ se usan como argumentos para la función *softmax* para obtener el vector de probabilidades sobre el resultado $\hat{\boldsymbol{y}}$. Además, también se asume que la pérdida es la verosimilitud logarítmica negativa del objetivo verdadero $y^{(t)}$ dado el insumo actual

- Para cada nodo $\boldsymbol{\mathcal{N}}$ se necesita calcular el gradiente $\nabla_{\boldsymbol{\mathcal{N}}} L$ recursivamente, basado en el gradiente calculado en nodos que lo siguen en el grado. Se comienza la recursión con los nodos inmediatamente precediendo la pérdida final:

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

- El gradiente $\nabla_{\boldsymbol{o}^{(t)}} L$ sobre los resultados en el momento t para toda i, t es el siguiente:

$$\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$

- Ahora, se puede comenzar desde el final de la secuencia para motivar el algoritmo. En el paso temporal final τ, $\boldsymbol{h}^{(\tau)}$ solo tiene $\boldsymbol{o}^{(\tau)}$ como descendiente, de modo que su gradiente es el siguiente:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^T \nabla_{\boldsymbol{o}^{(\tau)}} L$$

- Se puede iterar hacia atrás para hacer *backprop* de los gradientes a través del tiempo, desde $t = \tau - 1$ hasta $t = 1$, notando como $\boldsymbol{h}^{(t)}$ (para $t < \tau$) tiene como descendientes a $\boldsymbol{o}^{(t)}$ y $\boldsymbol{h}^{(t+1)}$. Su gradiente es el siguiente:

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left[J\left(\boldsymbol{h}^{(t+1)}, \boldsymbol{h}^{(t)}\right)\right]^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \left[J\left(\boldsymbol{o}^{(t)}, \boldsymbol{h}^{(t)}\right)\right]^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) =$$

$$= \boldsymbol{W}^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) diag\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)$$

$$where \quad diag\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) = \begin{bmatrix} 1 - \left(h_1^{(t+1)}\right)^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 - \left(h_\tau^{(t+1)}\right)^2 \end{bmatrix}$$

- Una vez que los gradientes en los nodos internos del grafo computacional se han obtenido, se pueden obtener los gradientes en los nodos de parámetros. Como los parámetros se comparten a través de los

diferentes pasos temporales, se tiene que tener cuidado con la notación para las operaciones de cálculo con estas variables

- Las ecuaciones que se quieren implementar usan el método *backprop* anteriormente visto, que calcula la contribución de un único arco en el grado computacional para el gradiente. No obstante, el operador $\nabla_{\boldsymbol{W}} f$ tiene en cuenta la contribución de $\boldsymbol{W}$ en f de todos los arcos del grafo

- Para resolver la ambigüedad, se introducen las variables $\boldsymbol{W}^{(t)}$ que se han definido como las copias de $\boldsymbol{W}$ pero cada $\boldsymbol{W}^{(t)}$ solo se usa en el momento t. Con esto, se puede denotar como $\nabla_{\boldsymbol{W}^{(t)}}$ para denotar la contribución de los pesos en el momento t al gradiente (la contribución de un solo arco a la función)

- Usando esta notación, el gradiente de los parámetros restantes son los siguientes:

$$\nabla_{\boldsymbol{c}} L = \sum_t \left[J\left(\boldsymbol{o}^{(t)}, \boldsymbol{c}\right)\right]^T \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left[J\left(\boldsymbol{h}^{(t)}, \boldsymbol{b}^{(t)}\right)\right]^T \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left[J\left(L, \boldsymbol{o}_i^{(t)}\right)\right]^T \nabla_{\boldsymbol{V}} \boldsymbol{o}_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)^T \left[\boldsymbol{h}^{(t)}\right]^T$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left[J\left(L, h_i^{(t)}\right)\right]^T \nabla_{\boldsymbol{W}} h_i^{(t)} = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \left[\boldsymbol{h}^{(t-1)}\right]^T$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left[J\left(L, h_i^{(t)}\right)\right]^T \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \left[\boldsymbol{x}^{(t)}\right]^T$$

- En el ejemplo desarrollado hasta ahora, las pérdidas $L^{(t)}$ eran entropías cruzadas entre las variables objetivo $\boldsymbol{y}^{(t)}$ y los resultados $\boldsymbol{o}^{(t)}$. Igual que en una red neuronal retroalimentada, es posible utilizar casi cualquier pérdida, dependiendo de la tarea

  - Cuando se usa un objetivo de entrenamiento de verosimilitud logarítmica predictiva, igual que con $L\left(\left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\right\}, \left\{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)}\right\}\right)$, se entrena la RNN para estimar la distribución condicional del siguiente elemento de la secuencia $\boldsymbol{y}^{(t)}$ dado los insumos pasados

    - Esto significa que se quiere maximizar la verosimilitud logarítmica $\log\left[p\left(\boldsymbol{y}^{(t)} \middle| \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\right)\right]$ o, si un modelo incluye conexiones desde el resultado en un paso temporal al siguiente, se quiere maximizar $\log\left[p\left(\boldsymbol{y}^{(t)} \middle| \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(t-1)}\right)\right]$

- Descomponer la probabilidad conjunta sobre la secuencia de los valores de $\boldsymbol{y}$ como una serie de predicciones probabilísticas es una de las maneras de capturar la distribución conjunta completa a lo largo de la secuencia

    - Cuando no se alimentan valores pasados de $\boldsymbol{y}$ como insumos que condicionan la predicción en el siguiente paso, el modelo gráfico direccionado no contiene arcos de ninguna $\boldsymbol{y}^{(i)}$ en el pasado a una $\boldsymbol{y}^{(t)}$ actual. En este caso, los resultados $\boldsymbol{y}$ son condicionalmente independientes dada la secuencia de valores de $\boldsymbol{x}$

    - Cuando se alimentan valores pasados (reales) de $\boldsymbol{y}$ como insumos que condicionan la predicción en el siguiente paso, el modelo gráfico direccionado contiene arcos de $\boldsymbol{y}^{(i)}$ en el pasado a una $\boldsymbol{y}^{(t)}$ actual. En este caso, los resultados $\boldsymbol{y}$ no son condicionalmente independientes dada la secuencia de valores de $\boldsymbol{x}$

- Como un ejemplo simple, se considera una RNN para una secuencia de variables aleatorias escalares $\mathbb{Y} = \{\mathrm{y}^{(1)}, \dots, \mathrm{y}^{(\tau)}\}$, sin insumos adicionales (el insumo será el resultado generado por la red en el siguiente paso temporal)

    - Por lo tanto, la RNN define un modelo gráfico direccional sobres las variables y y se parametriza la distribución conjunta de las observaciones de usando la regla de la cadena para las probabilidades condicionales:

$$P(\mathbb{Y}) = \prod_{t=1}^{\tau} P\left(\mathbf{y}^{(t)} \middle| \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \dots, \mathbf{y}^{(2)}, \mathbf{y}^{(1)}\right)$$

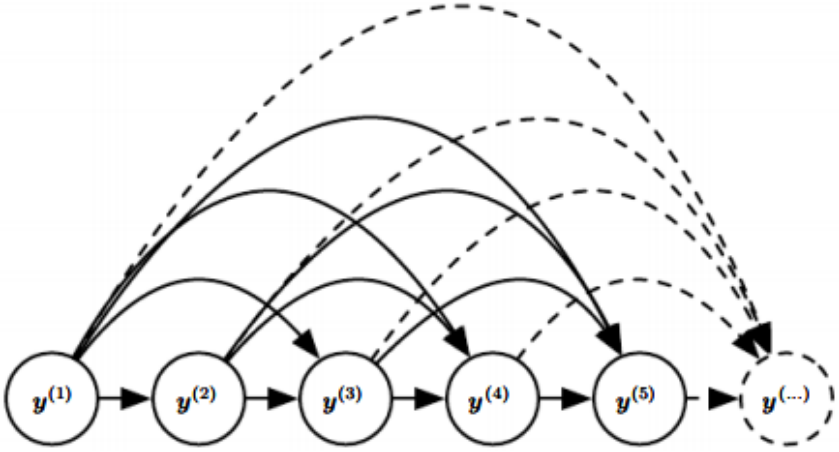
    - Por lo tanto, la verosimilitud logarítmica negativa del conjunto de valores $\{y^{(1)}, \dots, y^{(\tau)}\}$ acorde a este modelo es la siguiente:

$$L = \sum_{t} L^{(t)} = -\sum_{t} \log P\left(\mathrm{y}^{(t)} = y^{(t)} \middle| y^{(t-1)}, y^{(t-2)}, \dots, y^{(1)}\right)$$

- Los arcos en un modelo gráfico indican qué variables dependen directamente de otras variables, y se suelen omitir arcos que no indiquen una fuerte relación o interacción entre variables. No obstante, en algunos casos, a veces todos los insumos deberían tener una influencia en el siguiente elemento de la secuencia

- Las RNNs son útiles cuando se cree que la distribución sobre $\mathrm{y}^{(t)}$ puede depender del valor de $\mathrm{y}^{(i)}$ del pasado distante de una manera no capturada por el efecto de $\mathrm{y}^{(i)}$ sobre $\mathrm{y}^{(t-1)}$ (como la derivada parcial)

- Una manera de interpretar una RNN como un modelo grádico es verla como definiendo un modelo gráfico cuya estructura es la de un grafo completo, permitiendo la representación de dependencias directas entre cualquier par de valores de y. La interpretación gráfica completa de la RNN se basa en ignorar las unidades escondidas $\boldsymbol{h}^{(t)}$ marginalizándolas fuera del modelo



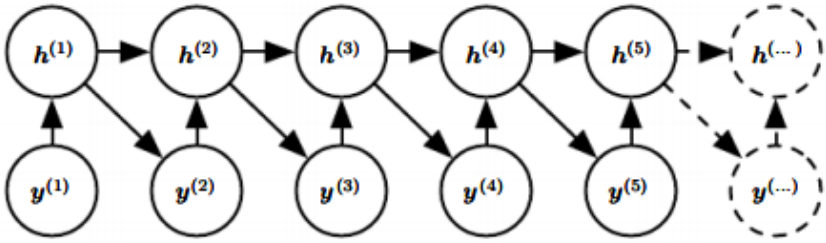
- Es más interesante considerar la estructura de modelo gráfico de las RNNs que resultan de interpretar las unidades escondidas $\boldsymbol{h}^{(t)}$ como variables aleatorias. Incluir las unidades escondidas en el modelo gráfico revela que la RNN proporciona una parametrización muy eficiente de la distribución conjunta sobre las observaciones

    - Suponiendo que se ha representado una distribución conjunta arbitraria sobre valores discretos con una representación tabular (un arreglo conteniendo una entrada separada para cada posible asignación de valores, siendo esta la probabilidad de que ocurra la asignación), si y puede tomar k diferentes valores, la representación tabular tendría $O(k^{\tau})$ parámetros

    - En comparación, debido a la compartición de parámetros, el número de parámetros en la RNN es $O(1)$ como una función de la longitud de la secuencia. El número de parámetros en la RNN se puede ajustar para controlar la capacidad del modelo, pero no se fuerza a que escale con la longitud de la secuencia

    - La ecuación $\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}; \boldsymbol{\theta}\right)$ muestra que la RNN parametriza las relaciones a largo plazo a través de recurrencias con una sola función f y unos mismos parámetros $\boldsymbol{\theta}$ para cada paso temporal.

Incorporando los nodos $\boldsymbol{h}^{(t)}$ en el modelo gráfico hace que se desacople el pasado y el futuro, actuando como una cantidad intermedia entre ellas. Una variable $y^{(i)}$ en el pasado distante puede influenciar a una variable $y^{(t)}$ a través de su efecto en $\boldsymbol{h}$



- La estructura del gráfico muestra que el modelo se puede parametrizar eficientemente usando las mismas distribuciones de probabilidad condicional en cada momento en el tiempo, y cuando todas las variables se observan, la probabilidad de la asignación conjunta de todas las variables se puede evaluar eficientemente

- Aún con la parametrización eficiente del modelo gráfico, algunas operaciones siguen siendo difíciles computacionalmente. Por ejemplo, es difícil predecir valores perdidos en medio de la secuencia

- El precio que las redes recurrentes pagan por un número reducido de parámetros es que optimizar los parámetros puede ser difícil

  - La compartición de parámetros usada en las RNN se basa en la suposición de que los mismos parámetros se pueden usar para diferentes pasos de tiempo. Equivalentemente, la suposición es que la distribución condicional sobre las variables es estacionaria

  - En principio, sería posible usar t como un insumo extra en cada momento de tiempo y dejar que el aprendiz descubra cualquier dependencia del tiempo mientras que aprende lo que más pueda entre pasos temporales. Esto sería mucho mejor que utilizar una distribución de probabilidad condicional diferente para cada t, pero la red tendría que extrapolar cuando se encuentre con nuevos valores de t

- Para completar la visión de las RNN como modelos gráficos, se tiene que describir como muestrear del modelo. La operación principal que se necesita realizar es la de muestrear de la distribución condicional en cada momento de tiempo

- No obstante, hay una complicación adicional: la RNN debe tener algunos mecanismos para determinar la longitud de la secuencia. Esto puede ser conseguido de varias maneras

- En este caso, cuando el resultado es un símbolo tomado de un vocabulario, uno puede añadir un símbolo especial correspondiente al final de la secuencia. Cuando se genera este símbolo, el proceso de muestreado para, y en el conjunto de entrenamiento se inserta este símbolo como un miembro extra de la secuencia, inmediatamente después de $\boldsymbol{x}^{(\tau)}$ en cada ejemplo de entrenamiento

- Otra opción es introducir un resultado de Bernoulli al modelo para representar la decisión de continuar o parar la generación en cada momento del tiempo. Este enfoque es más general que el enfoque de añadir un símbolo extra al vocabulario, porque se pueda aplicar a cualquier RNN, más solo a RNNs cuyo resultado sea una secuencia de símbolos (se puede aplicar a secuencias de números reales)

- La nueva unidad de resultado normalmente suele ser una unidad sigmoide entrenada con una pérdida de entropía cruzada. En este enfoque, la sigmoide se entrena para maximizar la probabilidad logarítmica de la predicción correcta de si la secuencia acaba o continua en cada paso temporal

- Otra manera de determinar la longitud de la secuencia τ para añadir un resultado extra al modelo que predice un entero τ. El modelo puede muestrear un valor de τ y entonces muestrea τ pasos de datos

- Este enfoque requiere un añadir un insumo extra a la actualización recurrente para cada paso temporal, de modo que la actualización recurrente tenga en cuenta de si está cerca del final de la secuencia generada. Este insumo extra puede consistir del valor de τ o se puede consistir de $\tau - t$, el número de pasos temporales restantes

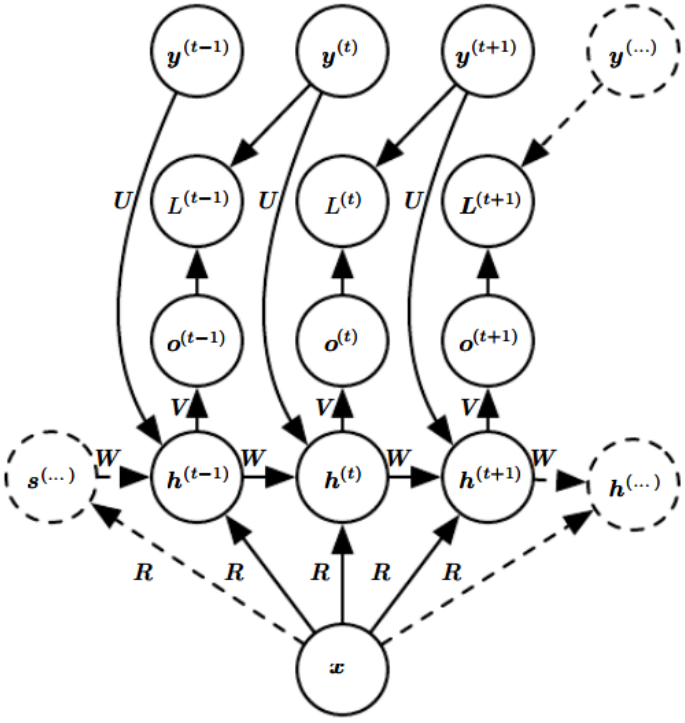
- Sin este insumo, la RNN puede generar secuencias que acaben abruptamente, tales como una oración que acabe antes de que se complete. Este enfoque se basa en la descomposición condicional de la probabilidad conjunta:

$$P\left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\right) = P(\tau)P\left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)} \middle| \tau\right)$$

- En la sección anterior se ha descrito como una RNN puede corresponder a un modelo gráfico direccionado sobre una secuencia de variables aleatorias $y^{(t)}$ sin

ningún insumo $\boldsymbol{x}$. Por supuesto, se puede incluir la secuencia $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$ y así hacer una extensión del modelo gráfico no solo para representar la distribución conjunta sobre las variables y sino también la distribución condicional de y sobre $\boldsymbol{x}$

- Cualquier modelo que representa una variable $P(\boldsymbol{y}; \boldsymbol{\theta})$ puede ser reinterpretado como un modelo representando una distribución condicional $P(\boldsymbol{y}|\boldsymbol{w})$ donde $\boldsymbol{w} = \boldsymbol{\theta}$
    - Se puede extender este modelo como representando una distribución $P(\boldsymbol{y}|\boldsymbol{x})$ usando el mismo principio, solo que ahora $\boldsymbol{w}$ será una función de $\boldsymbol{x}$. En el caso de una RNN, esto se puede conseguir de diferentes maneras
- Anteriormente se han discutido RNNs que tienen una secuencia de vectores $\boldsymbol{x}^{(t)}$ para $t = 1, \ldots, \tau$ como insumo. Otra opción es tomar un solo vector $\boldsymbol{x}$ como insumo: cuando $\boldsymbol{x}$ es un vector de tamaño fijo, se puede hacer que sea un insumo extra a la RNN que genera la secuencia $\mathbf{y}$
    - Las maneras más comunes de proporcionar un insumo extra a la RNN son proporcionarlo como un insumo extra en cada paso temporal, iniciarlo como el estado inicial $\boldsymbol{h}^{(0)}$ o ambos. El enfoque más común es el primero, ilustrado en el siguiente esquema:



- La interacción entre el insumo $\boldsymbol{x}$ y cada vector de la unidad escondida $\boldsymbol{h}^{(t)}$ se parametriza por una matriz de ponderaciones nueva $\boldsymbol{R}$ que estaba ausente del modelo de solo la secuencia de valores de y. El mismo producto $\boldsymbol{x}^T\boldsymbol{R}$ se añade como un insumo adicional para las unidades escondidas en cada momento del tiempo

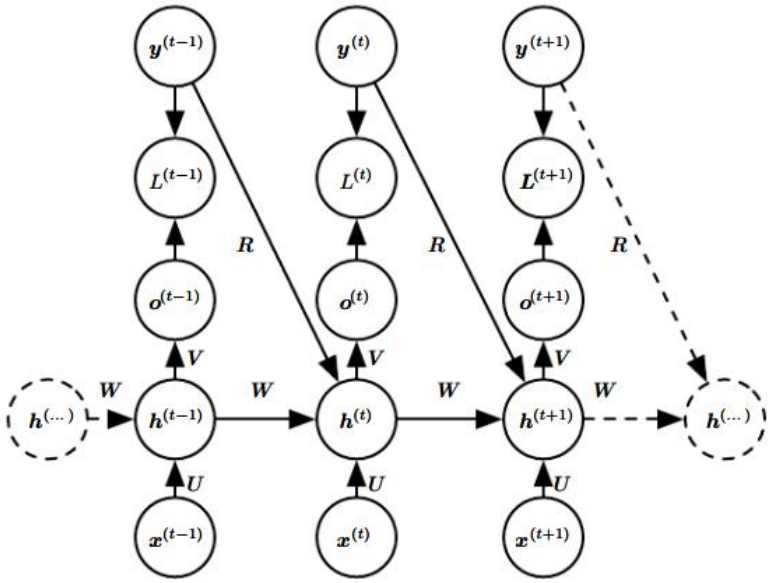
- Se puede pensar en la elección de $\boldsymbol{x}$ como determinando el valor de $\boldsymbol{x}^T\boldsymbol{R}$ que es, efectivamente, un nuevo parámetro de sesgo usado para cada una de las unidades escondidas. Las ponderaciones siguen siendo independientes del insumo

- Este modelo se puede interpretar como un modelo que toma $\boldsymbol{\theta}$ del modelo no condicional y hacer que sean $\boldsymbol{w}$, donde los parámetros de sesgo dentro de $\boldsymbol{w}$ son una nueva función del insumo

- Más que recibir solo un vector $\boldsymbol{x}$ como insumo, la RNN puede recibir una secuencia de vectores $\boldsymbol{x}^{(t)}$ como insumo. La RNN descrita en secciones anteriores corresponde a la distribución condicional $P\left(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)} \middle| \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\right)$ que hace una suposición de que se cumple la independencia condicional, factorizando la distribución de la siguiente manera:

$$\prod_t P\left(\boldsymbol{y}^{(t)} \middle| \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\right)$$

  - Para eliminar la suposición de independencia condicional, se pueden añadir conexiones del resultado en el momento t a la unidad escondida en el momento $t+1$. El modelo puede representar distribuciones arbitrarias sobre la secuencia $\boldsymbol{y}$



- Este tipo de modelo representando una distribución sobre una secuencia dada otra secuencia sigue teniendo una restricción, que es que la longitud de ambas secuencias debe ser la misma

Las redes neuronales secuenciales: redes bidireccionales y arquitecturas de codificación-descodificación

- Cuando se quiere que las predicciones de una RRN dependan de una secuencia dependa del pasado, pero también del futuro (de una secuencia entera), entonces se tienen que usar redes neuronales recurrentes bidireccionales

    - Todas las redes recurrentes consideradas hasta ahora tienen una estructura causal, dado que el estado en t captura información del pasado $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t-1)}$ y el insumo presente $\boldsymbol{x}^{(t)}$

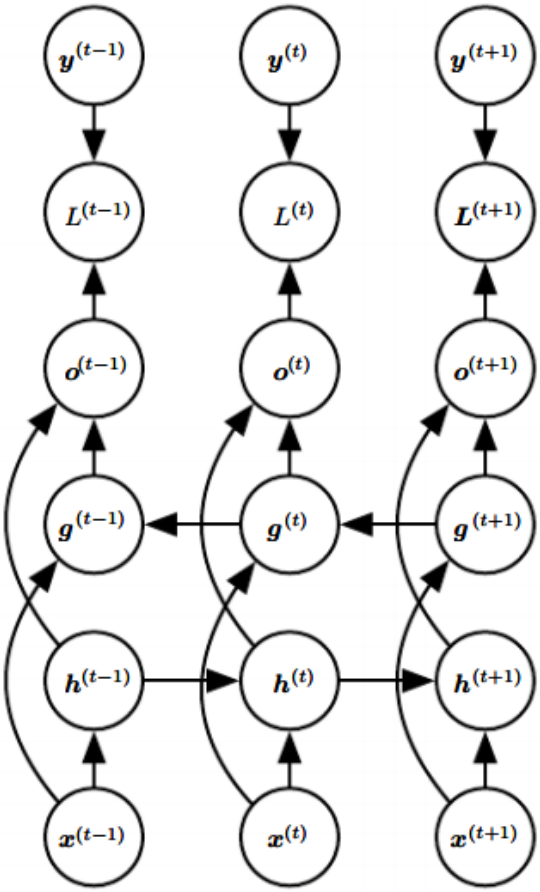
        - Algunos modelos discutidos también permiten información de valores $\boldsymbol{y}$ pasados para afectar al estado actual cuando los valores de $\boldsymbol{y}$ están disponibles

        - No obstante, en muchas aplicaciones, se quiere que se resulte en una predicción $\boldsymbol{y}^{(t)}$ que pueden depender de toda una secuencia de insumos. Por ejemplo, en reconocimiento del habla, la interpretación correcta del sonido como un fonema puede depender de los siguientes fonemas por la coarticulación y la dependencia lingüística (se tiene que mirar a las palabras futuras y sonidos futuros para poder desambiguar la interpretación correcta)

    - Las redes neuronales recurrentes bidireccionales o RNNs bidireccionales se inventaron para dirigirse a la necesidad de la dependencia de la predicción de una secuencia entera de insumos

        - Las RNN bidireccionales combinan una RNN que se mueve hacia adelante en el tiempo comenzando desde el comienzo de la secuencia con otra RNN que se mueve atrás en el tiempo comenzando por el final de la secuencia

        - Una RNN bidireccional común usa $\boldsymbol{h}^{(t)}$ para el estado de la sub-RNN que se mueve hacia adelante y $\boldsymbol{g}^{(t)}$ para el estado de la sub-RNN que se mueve hacia atrás



- Esto permite que las unidades de resultado $\boldsymbol{o}^{(t)}$ calculen una representación que depende tanto del pasado como del futuro, pero es más sensible a valores cercanos a t sin tener que especificar una ventana de tamaño específico alrededor de t (como uno tendría que hacer con una red neuronal retroalimentada, una red convolucional o una RNN con un *buffer* de tamaño fijo para tener *look-ahead*)

- Esta idea se puede extender naturalmente a un insumo bidimensional, tales como imágenes, teniendo cuatro RNNs, cada una yendo en una de las cuatro direcciones

  - En un punto (i, j) del *grid* bidimensional, un resultado $O_{i,j}$ puede calcular una representación que capturaría la mayoría de la información local pero también podría depender de insumos de largo alcance si la RNN puede aprender a llevar esta información

  - Comparado con una red convolucional, las RNNs aplicadas a las imágenes son típicamente más caras, pero permiten que las interacciones laterales de largo alcance entre características en el mismo mapa de características. Las ecuaciones de propagación hacia adelante para estas RNN se escriben de una forma que muestran que usan una convolución que calcula el insumo de

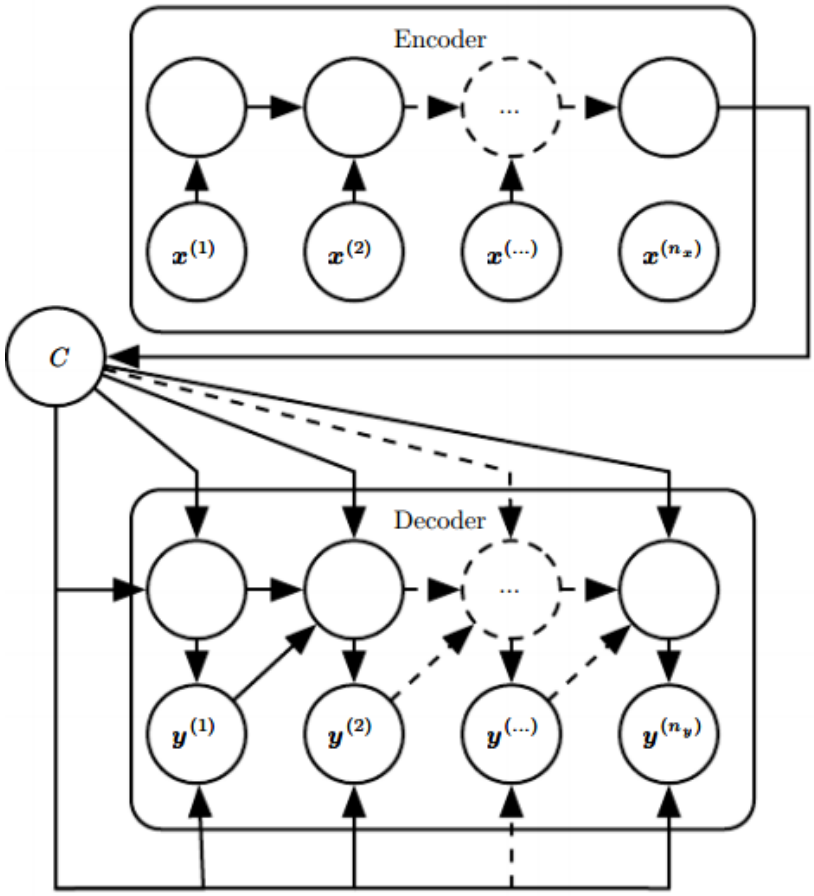
abajo a arriba para cada capa, antes de que la propagación recurrente a través del mapa de características incorpore las interacciones laterales

- Anteriormente, se ha visto que una RNN puede mapear una secuencia de insumo a un vector de tamaño fijo y viceversa y cómo se puede mapear una secuencia de insumo a una secuencia de resultado del mismo tamaño. Ahora se discute como una RNN puede mapear una secuencia de insumo a una secuencia de resultado no necesariamente del mismo tamaño

    - Normalmente se conoce a la secuencia de insumo de la RNN como contexto, y se quiere producir una representación de este contexto C

        - Este contexto C puede ser un vector o secuencia de vectores que resumen la secuencia de insumo $\boldsymbol{X} = \left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)}\right)$

    - La arquitectura de RNN más simple para mapear secuencias de longitud variable es la arquitectura de codificador-descodificador o secuencia a secuencia



        - Una RNN codificadora o lectora o de insumo procesa la secuencia de insumo, emitiendo un contexto C, normalmente como una función simple de su estado final escondido

        - Una RNN descodificadora o escritora o de resultados se condiciona a ese vector de tamaño fijo para generar una secuencia $\boldsymbol{Y} = \left(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n_y)}\right)$. La innovación en este tipo de arquitectura sobre las que se han presentado es que las longitudes de las secuencias n_x y n_y puede variar de una a otra, mientras que las otras arquitecturas se restringen a $n_x = n_y = \tau$

        - En una arquitectura secuencial secuencia a secuencia, las dos RNNs se entrenan conjuntamente para maximizar la media de $\log\left[P\left(\boldsymbol{y}^{(1)}, \ldots \boldsymbol{y}^{(n_y)} \middle| \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n_x)}\right)\right]$ sobre todos los pares de secuencias $\boldsymbol{x}$ e $\boldsymbol{y}$ en el conjunto de entrenamiento. El último estado $\boldsymbol{h}_{n_x}$ de la RNN codificadora típicamente se usa como una representación C de la secuencia de insumo que se proporciona como un insumo a la RNN decodificadora

        - Si el contexto C es un vector, entonces la RNN decodificadora es simplemente una RNN vector a secuencia para recibir el insumo. Como se ha visto, hay al menos dos maneras de hacer que se recibe un insumo en una RNN de vector a secuencia: se pueden usar como un insumo extra en cada paso temporal o se puede fijar como el estado inicial $\boldsymbol{h}^{(0)}$

        - No hay restricciones de que el codificador deba tener sobre el mismo tamaño de capas escondidas que el descodificador

    - Una limitación clara de la arquitectura es cuando el resultado del contexto C por la RNN codificadora tiene una dimensión que es demasiado pequeña para resumir adecuadamente una secuencia larga

        - Este fenómeno se ha observado en el contexto de la traducción automática

        - Se propuso hacer que C sea una secuencia de longitud variable y no un vector de tamaño fijo. Adicionalmente, se introdujo un mecanismo de atención que aprende a asociar elementos de la secuencia C a elementos de la secuencia resultante

Las redes neuronales secuenciales: profundidad y recursividad

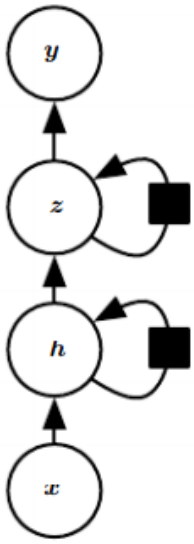
- Hasta ahora se han visto varias arquitecturas recurrentes para redes neuronales en donde solo se considera una capa. La evidencia experimental muestra que es necesario tener una profundidad suficiente para realizar los mapeados correspondientes

- El calculo en la mayoría de RNN se puede descomponer entres bloques de parámetros y transformaciones asociadas: del insumo al estado escondido, del estado escondido previo al siguiente, y del estado escondido al resultado

    - Con la primera arquitectura vista de RNNs, cada uno de estos tres bloques se asocia a una sola matriz de ponderaciones. Cuando la red se despliega, cada uno de los bloques corresponde a una transformación simple

    - Una representación simple se entiende como una transformación que se representaría por una sola capa en una red neuronal retroalimentada. Típicamente esta es una transformación representada por una transformación afín aprendida seguida por una no linealidad fija

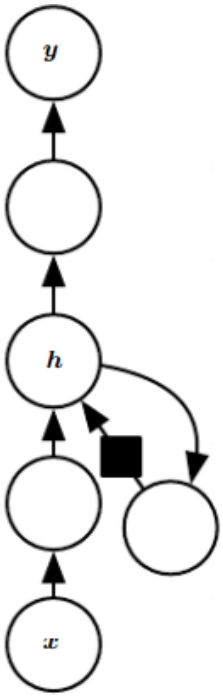
- Graves y otros investigadores, en su investigación de 2013, fueron los primeros en mostrar los beneficios de descomponer el estado de una RNN en múltiples capas



(a)

    - Se pueden interpretar las capas más bajas en la jerarquía como que juegan un papel importante en transformar el insumo “crudo” en una representación que es más apropiada para los niveles superiores o subsiguientes del estado escondido

- Pascanu y otros investigadores, en su investigación de 2014, propusieron usar una red neuronal separada (posiblemente profunda) para cada uno de los tres bloques enumerados anteriormente



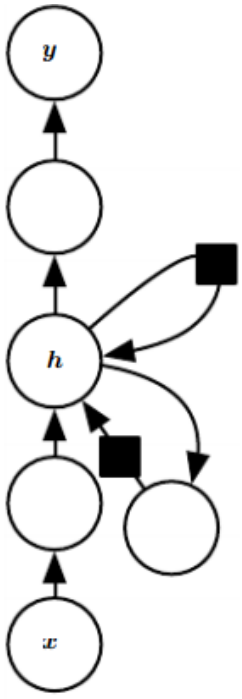
(b)

- Las consideraciones de la capacidad representacional sugieren que se asigne suficiente capacidad a cada uno de estos tres pasos, pero añadir esta profundidad puede dañar el aprendizaje haciendo que la optimización sea difícil

- En general, es más fácil optimizar redes neuronales simples, por lo que añadir la profundidad extra hace que el camino más corto de una variable en el paso temporal t a una variable en el paso $t + 1$ sea más largo

- Por ejemplo, si se usa una red neuronal retroalimentada con una sola capa para una transición de estado a estado, entonces se ha doblado la longitud del camino más corto entre variables entre dos oasis temporales diferentes, comparada con la arquitectura más simple de las RNN vistas anteriormente

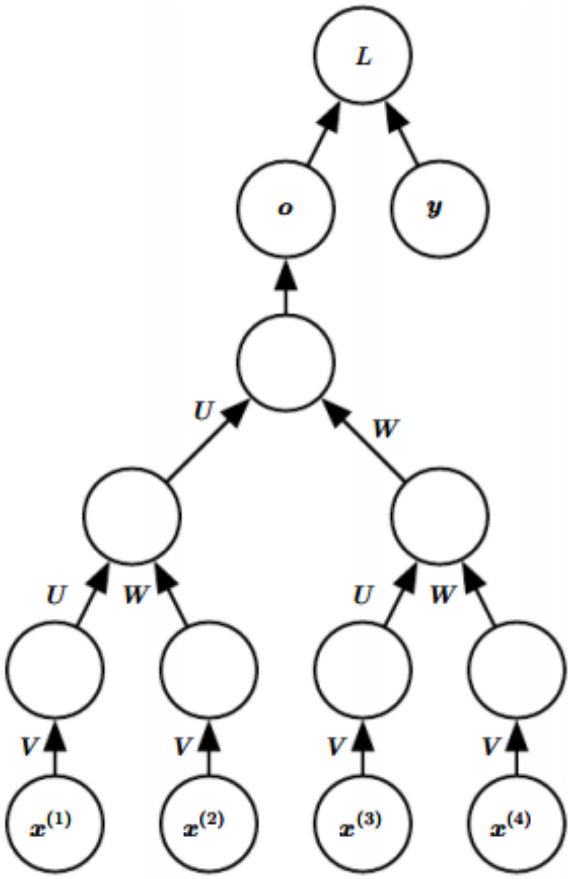
- No obstante, como los investigadores discutieron, se puede mitigar este efecto de la adición de longitud extra a través de la introducción de *skip connections* en el camino de unidad escondida a unidad escondida. Por lo tanto, en el grafo computacional se indica una recurrencia tanto en el nodo de $\boldsymbol{h}$ como en el nodo que indica la red neuronal retroalimentada simple



(c)

- Una generalización interesante de las redes neuronales recurrentes son las redes neuronales recursivas, las cuales tienen otro diseño y son útiles para aplicaciones específicas

    - Las redes neuronales recursivas son una generalización de las redes recurrentes, con un tipo diferente de grafo computacional, que se estructura como un árbol profundo, más que en una estructura de cadena como las RNNs



- Las redes neuronales recursivas se han aplicado con éxito al procesamiento de estructuras de datos como un insumo para las redes neuronales, además de en visión computacional

- Una clara ventaja de las redes recursivas sobre las redes recurrentes es que la secuencia de una misma longitud τ y profundidad (medida como el número de composiciones de operaciones no lineales) pueden ser drásticamente reducidas de τ a $O(\log \tau)$, lo cual puede ayudar a lidiar con dependencias a largo plazo

- Una pregunta abierta es cómo es la mejor manera de estructurar el árbol

  - Una opción es tener una estructura de árbol que no dependa de los datos, tal como un árbol binario balanceado

  - En algunos dominios, los métodos externos pueden sugerir una estructura de árbol apropiada. Por ejemplo, cuando se procesan oraciones, la estructura de árbol para la red recursiva puede fijarse a la estructura del *parse tree* (un árbol que representa la estructura lógica de una oración) de la oración proporcionada por el *natural language parser* (usado para extraer datos de la oración)

        - Idealmente, uno querría que el aprendiz mismo pudiera descubrir e inferir la estructura del árbol que es apropiada según su insumo

    - Existen muchas variantes de la idea de la red neuronal recursiva

        - Algunos investigadores asociaron los datos con la estructura de árboles, y asocian los insumos y las variables objetico con nodos individuales del árbol. El cálculo realizado por cada nodo no tiene por qué ser el cálculo realizado por una red neuronal artificial tradicional (transformación afín de todos los insumos seguidos de una no linealidad monótona)

        - Socher y otros investigadores, en su investigación de 2013, propusieron el uso de operaciones tensoriales y formas bilineares, que han sido útiles para modelar las relaciones entre conceptos, cuando los conceptos se representan por vectores continuos (*embeddings* o incrustaciones)

Las redes neuronales secuenciales: largo plazo, *echo states* y unidades *leaky*

- Uno de los problemas más importantes en el aprendizaje profundo relacionado a la recurrencia es la dificultad de aprender dependencias a largo plazo, lo cual está estrechamente relacionado con la optimización y el entrenamiento de la red neuronal

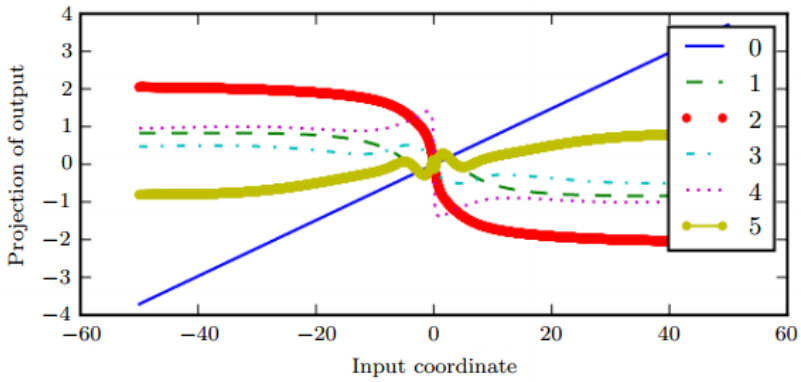
    - La dificultad matemática de aprender dependencias a largo plazo en redes recurrentes es que los gradientes propagados a través de varias fases tienden a desvanecerse (la mayoría del tiempo) o explotar (menos común)

        - Aún si se asume que los parámetros son tales que la red recurrente es estable (puede guardar memorias con gradientes que no explotan), la dificultad con las dependencias en el largo plazo aparece a partir de las ponderaciones exponencialmente pequeñas dadas las interacciones en el largo plazo (requiriendo la multiplicación de muchos Jacobianos) comparado con las interacciones en el corto plazo

    - Las redes neuronales recurrentes involucran la composición de la misma función múltiples veces, una por cada paso temporal. Estas composiciones pueden resultar en un comportamiento no lineal extremo

        - Esto se puede visualizar a través de proyectar los estados escondidos en una sola dimensión y con las coordenadas del insumo indicando las coordenadas del estado inicial a lo largo de

una dirección aleatoria en el espacio n-dimensional del estado. El gráfico, por lo tanto, se interpreta como una sección transversal de una función de altas dimensiones



- En particular la función de composición empleada por las redes recurrentes neuronales se parece a una multiplicación de matrices. Se puede pensar en una relación de recurrencia como en una red neuronal recurrente simple sin función de activación y sin insumos $\boldsymbol{x}$

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t-1)}$$

- Esta relación de recurrencia esencialmente describe el método de potencias, de modo que se puede simplificar y, si $\boldsymbol{W}$ admite una descomposición de valores propios de la forma $\boldsymbol{W} = \boldsymbol{Q}^T\boldsymbol{\Lambda}\boldsymbol{Q}$ con $\boldsymbol{Q}$ ortogonal, la relación de recurrencia se puede simplificar aún más

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t-1)} \Rightarrow \boldsymbol{h}^{(t)} = (\boldsymbol{W}^t)^T\boldsymbol{h}^{(0)}$$

$$\Rightarrow \boldsymbol{h}^{(t)} = \boldsymbol{Q}^T\boldsymbol{\Lambda}^t\boldsymbol{Q}\boldsymbol{h}^{(0)}$$

- Los valores propios se elevan a la potencia de t, causando que valores propios con magnitudes menores a 1 decaigan a cero y que los valores propios con magnitudes mayores a 1 exploten. Cualquier componente de $\boldsymbol{h}^{(0)}$ que no esté alineado con el vector propio será eventualmente descartado

- Este problema es particular a las redes recurrentes y se puede entender mejor a través de considerar el caso escalar en el que se multiplica la ponderación w por si misma múltiples veces

  - En este caso, w^t explotará o desaparecerá dependiendo de la magnitud de w. No obstante, si se hace una red no recurrente que tiene una ponderación diferente $w^{(t)}$ para cada paso temporal, entonces la situación es diferente

        - Si el estado inicial se diera por 1, entonces el estado en el momento t se da por $\prod_t w^{(t)}$. Suponiendo que los valores $w^{(t)}$ se generan aleatoriamente, independientemente uno del otro y con media 0 y varianza v, entonces la varianza de este producto es $O(v^n)$

        - Si se quisiera obtener una varianza deseada v^*, entonces se tendrían que escoger ponderaciones individuales con varianza $v = (v^*)^{1/n}$. Las redes neuronales retroalimentadas muy profundas con una escala seleccionada cuidadosamente, por lo tanto, pueden evitar el problema de la explosión y la desaparición de parámetros

    - Uno puede esperar que el problema se evite simplemente estando en una región del espacio paramétrico en donde los gradientes no exploten ni desaparezcan, pero para almacenar las memorias de manera que sean robustas a pequeñas perturbaciones, las RNN deben entrar en regiones en donde los gradientes desaparecen

        - Específicamente, cuando el modelo es capaz de representar dependencias a largo plazo, el gradiente de la interacción a largo plazo tiene exponencialmente menor magnitud que el gradiente de la interacción en el corto plazo

        - Esto no dice que es imposible aprender, pero dice que puede tomar mucho tiempo aprender dependencias a largo plazo, dado que la señal sobre estas dependencias tenderá a estar escondidas en las pequeñas fluctuaciones que aparecen en las dependencias a corto plazo

        - En la práctica, los experimentos muestran que cuanto más se incrementa el rango de dependencias que se necesita capturar, la optimización basada en gradientes se vuelve muy difícil, con una probabilidad de que haya un entrenamiento satisfactorio de una RNN tradicional via SGD llegando a 0 rápidamente en secuencias de longitud 10 o 20

Las aplicaciones del aprendizaje profundo: aprendizaje profundo a gran escala

- El aprendizaje automático se basa en la filosofía del conexionismo: mientras que una neurona biológica no es inteligente, grandes poblaciones de neuronas o características juntas pueden exhibir comportamiento inteligente. La escala juega un papel fundamental en los avances en el área y se requiere computación de alto rendimiento e infraestructura de *software*

- Tradicionalmente, las redes neuronales se entrenaban usando la CPU de una sola máquina, pero ahora se suele usar computación con GPUs o los CPUs de un clúster. Los investigadores en el área demostraron como la CPU era insuficiente para la carga computacional requerida por las redes

    - Una implementación adecuada en familias de CPUs puede proporcionar grandes mejoras. En 2011, Vanhoucke y otros investigadores propusieron una implementación que era más rápida usando aritmética de punto fijo que aritmética de punto flotante

    - Cada nuevo modelo de CPU tiene diferentes características de rendimiento, por lo que algunas implementaciones con aritmética de punto flotante pueden ser más rápidas. El punto más importante es que la especialización cuidadosa de rutinas de computación numérica puede dar grandes recompensas

    - Otras estrategias, aparte de escoger si usar aritmética de punto fijo o flotante, incluyen la optimización de estructuras de datos para evitar fallos de caché y usar instrucciones vectoriales. Muchos investigadores ignoran estos detalles de implementación, pero cuando el rendimiento de la implementación restringe el tamaño del modelo, la exactitud del modelo empeora

- La mayoría de implementaciones de redes neuronales se basan en GPUs, que se desarrollaron principalmente para propósitos gráficos pero que se aplican a otros propósitos debido a sus capacidades

    - En las aplicaciones gráficas se requiere el cálculo de diversas operaciones en paralelo en un corto periodo de tiempo. Las GPUs tienen que hacer multiplicaciones matriciales y divisiones de muchos vértices en paralelo para convertir coordenadas tridimensionales en coordenadas bidimensionales en la pantalla

    - Estos cálculos son simples y no tienen tantas ramificaciones como la carga computacional a la que la CPU suele estar expuesta. Por ejemplo, cada vértice en un objeto rígido está multiplicado por la misma matriz (no se necesita calcular previamente qué matriz multiplicar). Además, los cálculos son completamente independientes la una de la otra, por lo que se pueden paralelizar fácilmente

    - Las GPUs se diseñaron para que tuvieran un gran nivel de paralelismo y una banda ancha de memoria amplia, al coste de tener menos velocidad en tiempo de reloj y menor capacidad de ramificación relativamente a las CPUs tradicionales

- Los algoritmos de redes neuronales requieren las mismas características de rendimiento que los algoritmos gráficos descritos: normalmente requieren grandes y numerosos *buffers* de parámetros, valores de activación, y valores de gradiente

    - Estos *buffers* son lo suficientemente grandes para que no se clasifican como caché de una computadora tradicional, de modo que la banda ancha del sistema normalmente se vuelve un factor limitante. Las GPUs ofrecen una ventaja sobre las CPUs debido a que tienen una mayor banda ancha de memoria

    - Los algoritmos de entrenamiento de las redes neuronales normalmente no requieren mucha ramificación o control sofisticado, de modo que son apropiados para el *hardware* de la GPU. Como las redes neuronales se pueden dividir en múltiples neuronas individuales que se pueden procesar independientemente de las otras neuronas en la misma capa, de modo que las redes se benefician fácilmente de la computación en GPU

    - La popularidad de las GPUs para las redes neuronales se disparó con el advenimiento de las GPUs de propósito general, que podían ejecutar código arbitrario

- Escribir código eficiente para GP-GPUs es difícil porque las técnicas requeridas para poder obtener un buen rendimiento en la GPU son diferentes de las que se utilizan para obtener un buen rendimiento en la CPU

    - En la CPU se diseñan los algoritmos para que se acceda al caché siempre que se pueda, mientras en la GPU, como las localizaciones de memoria no están en cachés, puede ser más rápido calcular el mismo valor dos veces. Además, el código de GPU tiene naturalmente múltiples *threads* coordianados, de modo que se consideran aspectos como la lectura o escritura de valores simultáneamente para los *threads* o asegurarse que se ejecuta la misma instrucción de manera simultánea

    - Debido a la dificultad de escribir código de alto rendimiento en GPUs, los investigadores deberían estructurar su flujo de trabajo para evitar la necesidad de escribir código de GPU nuevo para comprobar nuevos modelos o algoritmos. Normalmente, uno construirá librerías de operaciones de alto rendimiento para poder trabajar más cómodamente, tales como *cuda-covnet* o PyLearn2

- En muchos casos, los recursos computacionales disponibles en una sola máquina son insuficientes, de modo que se quiere distribuir la carga de entrenamiento e inferencia (predicción) entre varias máquinas

    - La distribución de la inferencia es simple (cada insumo se puede procesar en una máquina diferente) a través del paralelismo de datos, pero también se puede hacer un paralelismo de modelo, donde múltiples máquinas trabajan en un mismo punto de datos, pero se ejecuta una parte diferente del modelo (asequible tanto para inferencia como entrenamiento)

        - El paralelismo de datos durante el entrenamiento es más difícil: uno puede incrementar el tamaño del *minibatch* usado para el paso de descenso de gradiente estocástico, pero normalmente se obtienen menos que rendimientos lineales en términos del rendimiento de la optimización

        - Sería mejor permitir que múltiples máquinas calcularan diversos pasos del algoritmo de descenso en paralelo, pero la definición del descenso de gradiente es un algoritmo secuencial

    - Esto último se puede solucionar a través del descenso de gradiente estocástico asíncrono, en donde varios núcleos de los procesadores de la memoria representan parámetros, y cada núcleo lee parámetros sin un *lock* (para los *threads*), calculan el gradiente e incrementa los parámetros sin un *lock*

        - Esto reduce la cantidad media de mejora que cada paso del descenso de gradiente da (algunos núcleos sobrescriben el progreso de los otros), pero la tasa incrementada de pasos causa que el proceso de entrenamiento sea más rápido en general

        - Dean y otros investigadores, en 2012, realizaron una implementación multimáquina sin *lock* para el descenso del gradiente, donde los parámetros se manejaban por un servidor de parámetros más que guardándose en la memoria compartida

- En muchas aplicaciones comerciales, es más importante que el coste temporal y de memoria de ejecutar la inferencia en el modelo de aprendizaje sea bajo que el nivel de coste de tiempo y memoria del entrenamiento. Esto implica que se necesita algún tipo de compresión del modelo

    - En aplicaciones donde no se requieren, uno puede entrenar un modelo una vez, ponerlo en marcha y ser usado por millones de usuarios

        - En muchos casos, el usuario final tiene menos recursos que el desarrollador. Por ejemplo, uno puede entrenar un modelo en un clúster computacional para implementarlo en teléfonos móviles

    - Una estrategia clave para reducir el coste de la inferencia es la compresión del modelo: la idea básica consiste en reemplazar el modelo original con un modelo más pequeño que requiere menos memoria y tiempo de ejecución para evaluarse

        - La compresión del modelo es aplicable cuando el tamaño del modelo original es dirigido principalmente por la necesidad de prevenir el sobreajuste. En la mayoría de casos, el modelo con el menor error de generalización es un ensamblado de varios modelos de entrenamiento independientes

    - Evaluar las n muestras es costoso, y a veces un solo modelo generaliza mejor si es más grande (usando, por ejemplo, regularización con *dropout*)

        - Estos modelos grandes aprenden una función $f(\boldsymbol{x})$ usando muchos más parámetros de los necesarios para la tarea. Su tamaño es necesario solo por el número limitado de ejemplos de entrenamiento

        - En cuanto se ha ajustado esta función $f(\boldsymbol{x})$ en estos puntos, se puede generar un conjunto de entrenamiento teniendo infinitos ejemplos, simplemente aplicando f a puntos $\boldsymbol{x}$ muestreados aleatoriamente. Entonces, se entrena un nuevo modelo más pequeño para poder hacer coincidir $f(\boldsymbol{x})$ en estos puntos

        - Con tal de usar de manera más eficiente la capacidad de este modelo más pequeño, lo mejor es muestrear los nuevos puntos $\boldsymbol{x}$ de una distribución parecida a la de los insumos de comprobación o *test* que serán proporcionados al modelo después. Esto se puede hacer corrompiendo ejemplos de entrenamiento o dibujando puntos de un modelo generativo entrenado en el conjunto de entrenamiento original

        - Alternativamente, uno puede entrenar un modelo más pequeño solo con los puntos de entrenamiento, pero entrenarlo para copiar otras características del modelo, tales como su distribución posterior sobre las clases incorrectas

- Una estrategia para acelerar los sistemas de procesamientos de datos en general es construir sistemas que tengan una estructura dinámica en el gráfico que describe el cálculo necesario para el procesamiento de los insumos

- Los sistemas de procesamiento de datos pueden determinar dinámicamente qué subconjunto de redes neuronales debería ejecutarse para un insumo concreto

    - Las redes neuronales individuales también pueden exhibir estructura dinámica internamente al determinar que subconjunto de características (unidades escondidas) calcular dada la información sobre el insumo. Esta forma de estructura dinámica interna se conoce como cálculo condicional

    - Debido a que muchos componentes de la arquitectura pueden ser relevantes solo para una cantidad pequeña de insumos posibles, el sistema puede ejecutarse más rápido al calcular solo las unidades escondidas cuando sea necesario

- La estructura dinámica de los cálculos es un principio básico de la ciencia computacional aplicado a la ingeniería de *software*. Las versiones más simples de la estructura dinámica aplicadas a redes neuronales se basan en determinar el subconjunto delo mismo grupo de redes neuronales (u otros modelos de aprendizaje automático) deberían aplicarse a un insumo particular

    - Una estrategia venerable para acelerar la inferencia en un clasificador es usar una cascada de clasificadores. Esta estrategia se puede aplicar cuando el objetivo es detectar la presencia de un objeto o evento raro

    - Para saber de manera segura que un objeto está presente, uno debe utilizar un clasificador sofisticado con alta capacidad, el cual es costoso de ejecutar. No obstante, porque el objeto es raro, normalmente se puede utilizar menos coste computacional para rechazar insumos que no contienen ese objeto

    - Los primeros clasificadores en la secuencia tienen una capacidad baja, y se entrenan para que tengan un *recall* alto (para que no se rechace erróneamente un insumo con el objeto presente), mientras que e modelo final se entrena para tener gran precisión. En el momento de comprobación, se ejecuta la inferencia ejecutando los clasificadores en una secuencia, abandonando cualquier ejemplo en el momento en el que un modelo de la cascada lo rechace

    - Esto permite verificar la presencia de objetos con alta confianza, usando un modelo de alta capacidad, pero no fuerza al modelador a pagar el coste de la inferencia completa para cada ejemplo

- Hay dos maneras en que la cascada puede conseguir una capacidad más alta:

    - Una manera es hacer que los miembros finales de la cascada tengan una alta capacidad individualmente. En este caso, el sistema tendrá alta capacidad porque sus miembros la tienen

    - Otra manera es hacer una cascada en la que cada modelo individual tiene poca capacidad, pero el sistema tiene una alta capacidad debido a la combinación de varios modelos

    - Alternativamente, otra versión de cascadas utiliza los primeros modelos de la secuencia para implementar un tipo de mecanismo de atención fuerte: los primeros modelos de la cascada localizan el objeto y los miembros finales de la cascada realizan procesamiento adicional dada la localización del objeto

- Los árboles de decisión son un ejemplo de estructura dinámica, porque cada nodo en el árbol determina cuáles de sus subárboles debería evaluarse para cada insumo

    - Una manera simple de unir el aprendizaje profundo y la estructura dinámica sirve para entrenar un árbol de decisión en el que cada nodo usa una red neuronal para dividir la decisión, aunque esto no se ha usado mucho para no ralentizar los cálculos de la inferencia

    - Con la misma razón, uno puede usar una red neuronal, llamada *gater* para seleccionar una de las posibles redes expertas que se usarán para usar el cálculo del resultado, dado el insumo actual

    - La primera versión de esta idea se llamó mezcla de expertos, en donde el *gater* resulta en un conjunto de probabilidades o ponderaciones (una por experto), y el resultado final se obtiene por una combinación ponderada de los resultados de los expertos. En ese caso el *gater* no ofrece una reducción en un coste computacional, pero si se escoge solo un experto para cada ejemplo, entonces se obtiene una mezcla de expertos fuerte, que puede acelerar el entrenamiento y la inferencia

    - Esta estrategia funciona cuando el número de decisiones del *gater* es pequeño porque no es combinatorio. Pero cuando se quieren seleccionar diferentes subconjuntos de unidades o parámetros, no es posible usar un tipo de “interruptor suave” porque requiere enumerar (y calcular los resultados de) todas las configuraciones del *gater*

- Para poder solucionar este problema, se han explorado varios enfoques para entrenar *gaters* combinatorios, explorando estimadores del gradiente en las probabilidades de que se seleccione una red, o usando técnicas de aprendizaje de refuerzo para aprender el *dropout* condicional en bloques de unidades escondidas y tener una reducción real del coste computacional sin impactar negativamente en la calidad de la aproximación

- Otro tipo de estructura dinámica es un interruptor, donde una unidad escondida puede recibir un insumo de diferentes unidades dependiendo del contexto

    - Este enfoque de rutina dinámica se puede interpretar como un mecanismo de atención

    - Hasta ahora, el uso de un interruptor fuerte no ha sido efectivo en aplicaciones a gran escala. Enfoques actuales, en cambio, usan promedios ponderados sobre muchos insumos posibles, y por tanto no consiguen todos los beneficios computacionales posibles de la estructura dinámica

- Un obstáculo muy grande para utilizar sistemas estructurados dinámicamente es el bajo nivel de paralelismo que resulta de un sistema siguiendo diferentes ramificaciones de código para diferentes insumos

    - Esto significa que pocas operaciones en la red se pueden describir como una multiplicación de matrices o una convolución de *batches* en un *minibatch* de ejemplos. Se pueden escribir subrutinas más especializadas que hagan una convolución de cada ejemplo con diferentes *kernels* o multipliquen cada fila de una matriz de diseño por un conjunto de columnas de ponderaciones diferente

    - Desafortunadamente, estas rutinas son difíciles de implementar eficientemente: implementaciones en la CPU serán más lentas debido a la falta de coherencia de caché, y las implementaciones de la GPU serán lentas debido a que se necesitan serializar los grupos de *threads* cuando sus miembros toman diferentes ramificaciones

    - En algunos casos, estos problemas se pueden mitigar partiendo los ejemplos en grupos que tomen la misma ramificación, y procesar estos grupos de ejemplos simultáneamente. Esto puede ser aceptable para minimizar el tiempo requerido para procesar una cantidad fija de ejemplos en una configuración *offline*

        - En una configuración en tiempo real donde los ejemplos se procesan continuamente, la partición de la carga puede resultar en problemas de balanceo de carga. Por ejemplo, si se asigna una máquina para procesar el primer paso de una cascada y otra para procesar el último, entonces el primero tenderá a estar sobrecargado y el último al revés

El aprendizaje de representación: pre-entrenamiento

- Muchas tareas de procesamiento de información pueden ser muy fáciles o muy difíciles dependiendo de como se representa la información. Por lo tanto, el aprendizaje de representación es una de las áreas más importantes
    - Es posible cuantificar el tiempo de ejecución asintótico de de varias operaciones usando representaciones apropiadas o inapropiadas
        - Por ejemplo, insertar un número en la posición correcta en una lista ordenada de números es una operación $O(n)$ si la lista se representa como una lista vinculada, pero solo $O(\log n)$ si la lista se representa como un árbol *red-black*
        - En aprendizaje estadístico, una buena representación es aquella que hace una tarea subsiguiente más fácil. La elección de la representación normalmente dependerá de la elección de la tarea de aprendizaje subsiguiente
    - Entrenar con tareas supervisadas naturalmente lleva a que haya una representación en cada una de las capas escondidas de una red neuronal retroalimentada (sobre todo en las capas finales), teniendo propiedades que hacen la clasificación o la regresión más fácil
        - En el caso de la clasificación, por ejemplo, insumos que no son linealmente separables pueden ser linealmente separables en la última capa
        - Dependiendo del tipo de la última capa, las unidades escondidas de la penúltima capa deben aprender diferentes propiedades
    - El entrenamiento supervisado de redes neuronales supervisadas no involucra la imposición de una condición sobre las características intermedias aprendidas. Otros tipos de algoritmos de aprendizaje de representación normalmente se diseñan específicamente para formar la representación de una manera determinada
        - Si, por ejemplo, se quiere aprender una representación para hacer la estimación de la densidad más fácil, las distribuciones con más independencia serán más fáciles de modelar, de modo que

se podría diseñar una función objetivo que motive a los elementos del vector de representación $\boldsymbol{h}$ a ser independientes

- Igual que en redes supervisadas, los algoritmos de aprendizaje profundo no supervisado tienen un objetivo principal de entrenamiento, pero también aprenden una representación como efecto secundario

- Independientemente de cómo se ha obtenido la representación, se puede usar para otra tarea. Alternativamente, múltiples tareas (supervisadas o no supervisadas) se pueden aprender a la vez con alguna representación interna compartida

- La mayoría de problemas de representación presenta un *tradeoff* entre preservar tanta información del insumo como se pueda y conseguir buenas propiedades (como la independencia)

  - El aprendizaje de representación es particularmente interesante porque proporciona una manera de realizar aprendizaje no supervisado o semi-supervisado

    - Normalmente se tienen grandes cantidades de datos de entrenamiento sin etiquetas y relativamente pocos datos etiquetados. Entrenar técnicas de aprendizaje supervisado solo en el subconjunto de ejemplos etiquetados normalmente suele resultar nen un sobreajuste severo

    - El aprendizaje semi-supervisado ofrece la posibilidad de resolver este problema de sobreajuste a través de aprender también de los datos no etiquetados. Específicamente, se pueden aprender buenas representaciones de datos no etiquetados, y usar estas representaciones para la tarea de aprendizaje supervisado

- El aprendizaje no supervisado ha sido muy importante para que los investigadores entrenen redes neuronales supervisadas sin requerir especializaciones arquitectónicas (como la convolución o la recurrencia). A esto se le llama pre-entrenamiento no supervisado o *greedy layer-wise unsupervised pretraining*, que es un ejemplo canónico de como una representación aprendida para una tarea (aprendizaje no supervisado) puede ser útil para otra (aprendizaje supervisado)

  - El pre-entrenamiento no supervisado se apoya en un algoritmo de aprendizaje de representación de una sola capa tal como RBM, un autocodificador de una sola capa, un modelo de codificación *sparse* u otro modelo que aprenda representaciones latentes

- Cada capa se pre-entrena usando aprendizaje no supervisado, tomando el resultado de la capa anterior para producir como resultado una nueva representación de los datos, cuya distribución (o su relación con otras variables) es, deseablemente, más sencilla

- Dado un algoritmo de aprendizaje estadístico no supervisado $\mathcal{L}$ que toma como conjunto de entrenamiento de ejemplos y devuelve un codificador o función de características f, se puede plantear de manera más formal el siguiente algoritmo:

$f \leftarrow$ Identity function
$\tilde{\boldsymbol{X}} = \boldsymbol{X}$
for $k = 1, \ldots, m$ **do**
  $f^{(k)} = \mathcal{L}(\tilde{\boldsymbol{X}})$
  $f \leftarrow f^{(k)} \circ f$
  $\tilde{\boldsymbol{X}} \leftarrow f^{(k)}(\tilde{\boldsymbol{X}})$
end for
if *fine-tuning* ***then***
  $f \leftarrow \mathcal{T}(f, \boldsymbol{X}, \boldsymbol{Y})$
end if
Return f

  - Los datos de insumo $\boldsymbol{X}$ con una fila por ejemplo y $f^{(1)}(\boldsymbol{X})$ es el resultado de la primera fase del codificador en $\boldsymbol{X}$, habiendo un total de m fases

  - Si se realiza *fine-tuning*, se usa un aprendiz $\mathcal{T}$ que toma una función inicial f, ejemplos de insumo $\boldsymbol{X}$ (y en caso de aprendizaje supervisado, asociados a la variable objetivo $\boldsymbol{Y}$) y devuelve la función modificada

- Los procedimientos de *greedy layer-wise training* basados en criterios no supervisados para poder sobrepasar la dificultad de entrenar conjuntamente capas de una red neuronal para una tarea supervisada

  - Este pre-entrenamiento se clasifica como *greedy* porque es un algoritmo avaricioso (optimiza cada pieza de la solución independientemente o en cada momento, no conjuntamente) y como *layer-wise* porque las piezas independientes son las capas de la red

  - Específicamente, el algoritmo procede con una capa en cada momento, entrenando la capa k mientras que se mantienen las otras fijas. En particular, las capas más bajas (que se entrenan primero) no se adaptan después de que las capas superiores se introduzcan

        - Aunque se clasifica como no supervisado debido a que se entrena cada capa con un algoritmo de aprendizaje de representación no supervisado, también se le llama de pre-entrenamiento porque se supine que solo es un primer paso antes de que se aplique el algoritmo de entrenamiento conjunto para el *fine-tuning* de todas las capas a la vez

        - En el contexto del aprendizaje supervisado, se puede ver como un regularizador (en algunos experimentos, el pre-entrenamiento reduce el error de comprobación sin reducir el de entrenamiento) y una forma de inicializar los parámetros

    - Es común utilizar la palabra pre-entrenamiento para referirse no solo a esta fase misma sino también al protocolo entero de dos fases que combina la fase de pre-entrenamiento y la fase de aprendizaje supervisado

        - La fase del aprendizaje supervisado puede involucrar el entrenamiento de un clasificador simple encima de las características aprendidas en la fase de pre-entrenamiento, o puede involucrar *fine-tunning* de la red neuronal entera

    - No importa qué tipo de algoritmo de aprendizaje no supervisado o qué tipo de modelo se emplea, en la gran mayoría de casos, el esquema de entrenamiento general es más o menos el mismo. Mientras que la elección del algoritmo de aprendizaje no supervisado afectará a los detalles, la mayoría de aplicaciones de pre-entrenamiento no supervisado seguirán más o menos este protocolo

        - Además, este pre-entrenamiento puede ser usado como inicialización para otros algoritmos de aprendizaje no supervisado, tales como autocodificadores profundos y modelos probabilísticos con muchas capas de variables latentes. Tales modelos pueden incluir redes de creencias profundas y máquinas de Boltzmann profundas

        - Es posible tener un algoritmo de pre-entrenamiento supervisado como el visto, basado en la premisa de que entrenar una red neuronal simple (de una sola capa) es más fácil que entrenar uno profundo, que parece validarse en varios contextos