

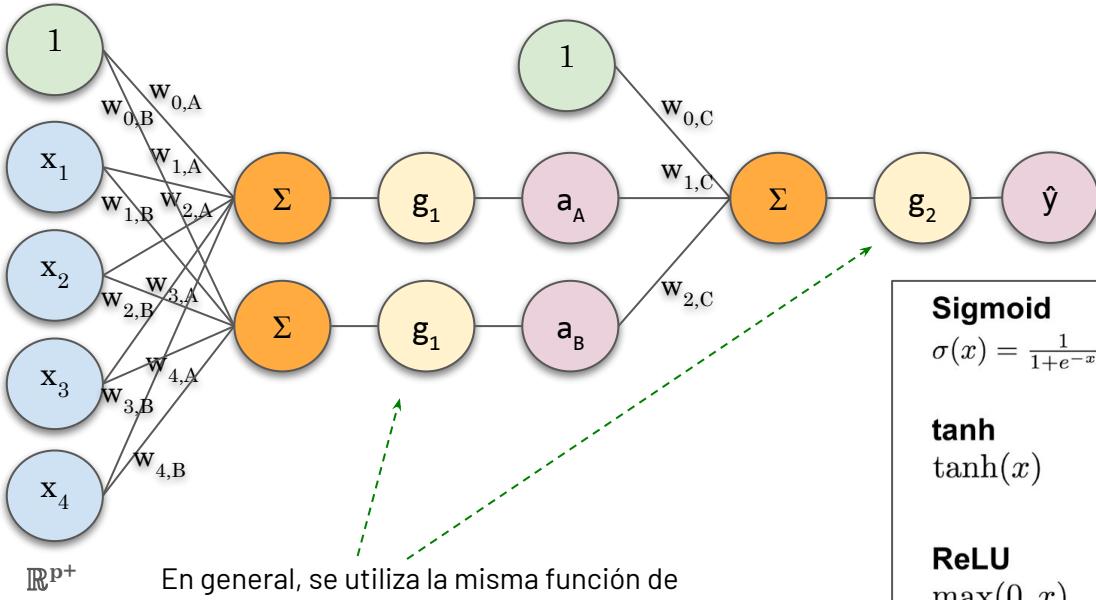
Aprendizaje Automático

Clase 9:

Redes neuronales. Parte II
Predicción de Secuencias

<Repaso>

Redes neuronales



Pero la de la última capa depende del tipo de problema.

Sin capas ocultas:

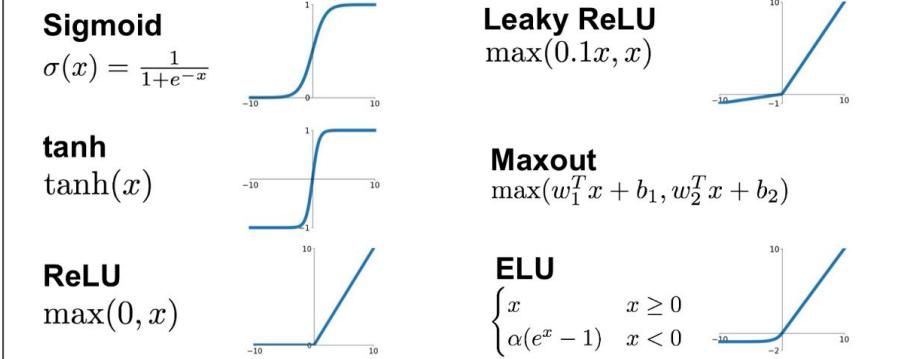
$$g_2 = \sigma \text{ (sigmoidea)} \rightarrow \text{Reg. Logística}$$

$$g_2 = I \text{ (identidad)} \rightarrow \text{Reg. Lineal}$$

Con capas ocultas:

Si las g_1 no son no-lineales, el sistema es equivalente a Reg. Lineal (o logística) según g_2 . $W_3(W_2(W_1)x) = (W_3W_2W_1)x$

Funciones de activación



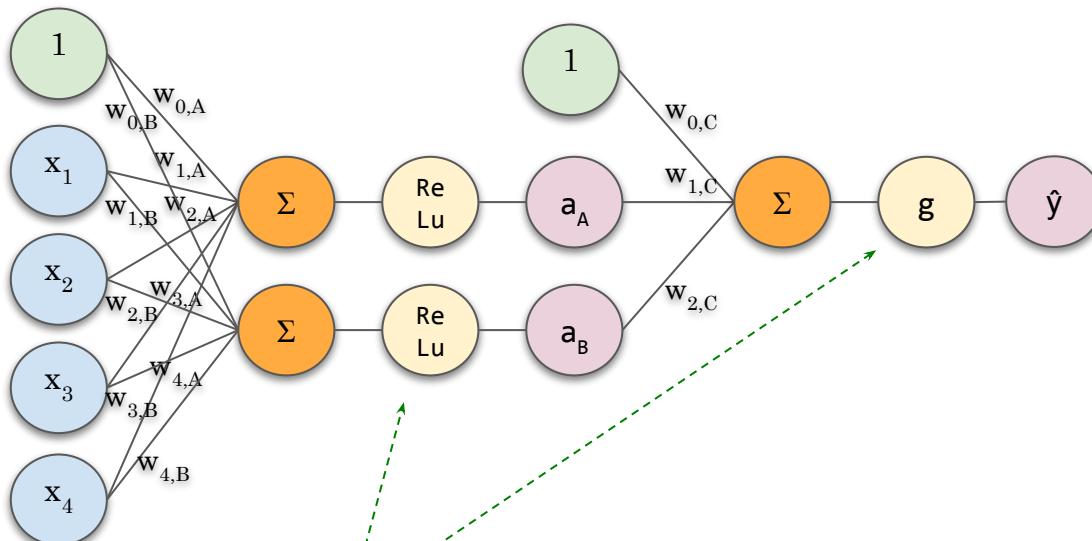
Fuente <https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092>

Redes neuronales

Sin capas ocultas:

$g = \sigma$ (sigmoidea) \rightarrow Reg. Logística

$g = I$ (identidad) \rightarrow Reg. Lineal



\mathbb{R}^{p+}_1

En general, se utiliza la misma función de activación en todas las capas intermedias (en este caso sólo una capa intermedia).

Pero la de la última capa depende del tipo de problema.

Aprender = Parámetros (W s) que minimicen:

¿Cómo?: Descenso por el gradiente (o simil).

Necesitamos la función a minimizar:

- Caso regresión: Costo $MSE_{X,y}(w)$ $J_{Ridge}(w)$
- Caso clasificación binaria $Costo_BinCE_{X,y}$
- Caso clasificación multiclase: **Hoy**
- Caso clasificación multilabel: **Hoy**

Su gradiente en un punto: $\nabla_W J_{X,y}(W)$

$\nabla_w MSE_{X,y}(w)$

$\nabla_w J_{Ridge}(w)$

$\nabla_w Binary_CE^{(i)}$

¿Simples de calcular?

- **No** (sí para reg. lineal o reg. logística)
- ¿Entonces? Backpropagation.

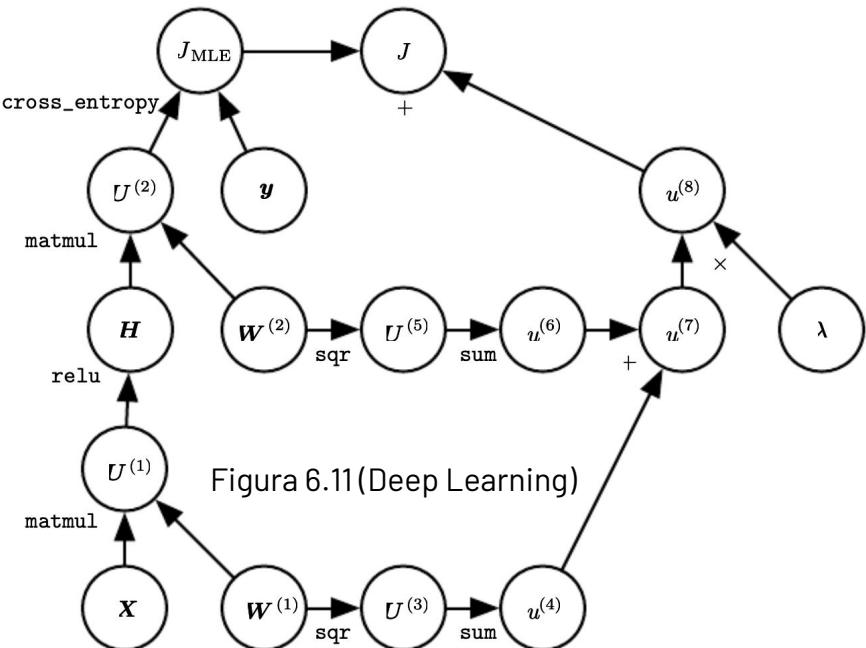
Gradiente de la función de costo final

Una vez definida la arquitectura de la red, incluyendo la función de costo (mse / cross-entropy) e incluyendo regularizaciones y demás operaciones:

Ejemplo (CE + regularización L2)

$$J_{MLE} = \frac{1}{n} \sum_{i=1}^n \text{Multiclass_CE}^{(i)} = \frac{1}{n} \sum_{i=1}^n \left[-\sum_{k=1}^K y_k^{(i)} \log(\hat{h}_w^{(k)}(\mathbf{x}^{(i)})) \right]$$
$$J = J_{MLE} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right)$$

- Se genera un **grafo computacional** (similar a un AST) que representa exactamente la función a minimizar.
- Cada nodo representa un cómputo elemental.
- Hacer una pasada "**forward**" en este grafo permite evaluar la red en un punto dado.
- Hacer una pasada "**backward**" en el grafo subyacente (que contiene las derivadas parciales correspondientes a cada cómputo) permite aplicar la **regla de la cadena** para encontrar la derivada en un punto.
- Utilizando estas estrategias, podemos entrenar la red haciendo **descenso por el gradiente**.



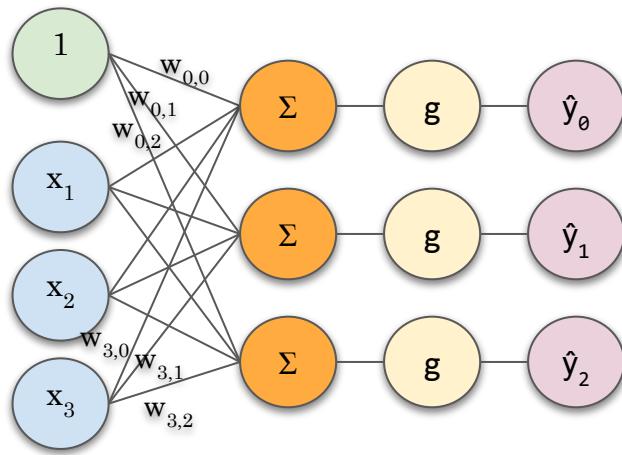
</Repaso>

Redes Neuronales Multi Output

Redes Neuronales Multi Clase

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

Podríamos usar la siguiente arquitectura:

 \mathbb{R}^{p+1} \mathbb{R}^C \mathbb{R}^C

Es decir, podemos utilizarlo para obtener:

$$\hat{y}_0 = P(Y = \text{clase 0} \mid X = x^{(i)})$$

$$\hat{y}_1 = P(Y = \text{clase 1} \mid X = x^{(i)})$$

$$\hat{y}_2 = P(Y = \text{clase 2} \mid X = x^{(i)})$$

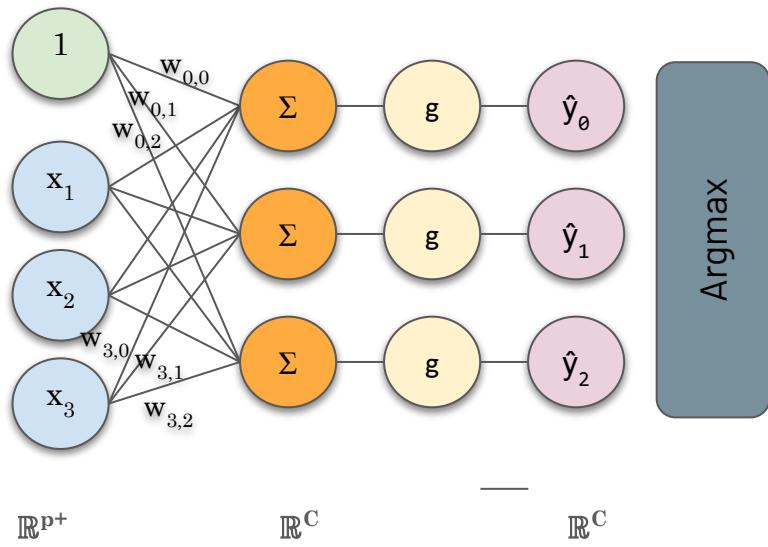
Vimos que si $\mathbf{g}(\mathbf{z}) = \text{sigmoidea}(\mathbf{z})$ obtenemos scores entre 0 y 1. ¿Estaría bien usarla en este caso?

¿Cómo elegimos la clase final?

Redes Neuronales Multi Clase

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

Podríamos usar la siguiente arquitectura:



Es decir, podemos utilizarlo para obtener:

$$\begin{aligned}\hat{y}_0 &= P(Y = \text{clase 0} \mid X = x^{(i)}) \\ \hat{y}_1 &= P(Y = \text{clase 1} \mid X = x^{(i)}) \\ \hat{y}_2 &= P(Y = \text{clase 2} \mid X = x^{(i)})\end{aligned}$$

Una opción sería usar

- $g(z) = z$ o quizás $g(z) = \text{sigmoidea}(z)$

y luego usar $\arg\max(\hat{y})$.

Problemas:

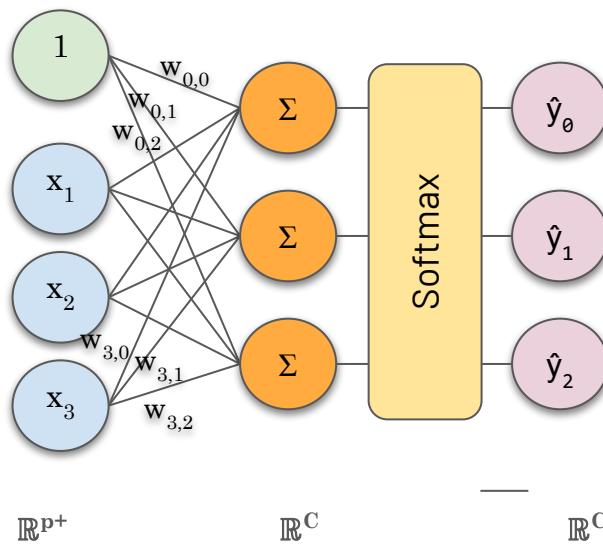
- 1) Perdemos las probabilidades para las distintas clases (**no suman 1**).
- 2) $\arg\max(\hat{y})$ **no es diferenciable**, cuando querremos calcular la loss, no se propaga el gradiente!

Redes Neuronales Multi Clase

Softmax

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

Podríamos usar la siguiente arquitectura:



Es decir, podemos utilizarlo para obtener:

$$\hat{y}_0 = P(Y = \text{clase 0} \mid X = x^{(i)})$$

$$\hat{y}_1 = P(Y = \text{clase 1} \mid X = x^{(i)})$$

$$\hat{y}_2 = P(Y = \text{clase 2} \mid X = x^{(i)})$$

Usamos **Softmax** una **función de activación** (que toma en cuenta los valores del resto de las neuronas). Su salida **suma 1** y es **diferenciable**.

Softmax
activation function

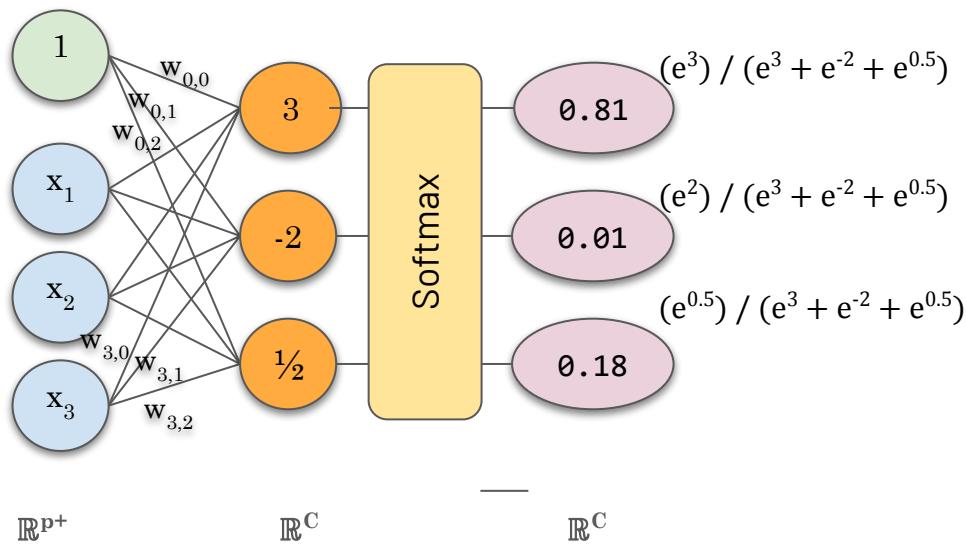
$$\frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Redes Neuronales Multi Clase

Softmax

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

Podríamos usar la siguiente arquitectura:



Es decir, podemos utilizarlo para obtener:

$$\hat{y}_0 = P(Y = \text{clase 0} \mid X = x^{(i)})$$

$$\hat{y}_1 = P(Y = \text{clase 1} \mid X = x^{(i)})$$

$$\hat{y}_2 = P(Y = \text{clase 2} \mid X = x^{(i)})$$

Usamos **Softmax** una **función de activación** (que toma en cuenta los valores del resto de las neuronas). Su salida **suma 1** y es **diferenciable**.

Softmax
activation function

$$\boxed{\frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}}$$

Entropía Cruzada (Multi Clase)

$$\hat{h}(\mathbf{x}^{(i)}) = \text{sigmoid}(f^{(L)}(f^{(L-1)} \dots (\mathbf{x}^{(i)}))) = \hat{P}(Y = 1 | X = \mathbf{x}^{(i)})$$

Entropía cruzada (para caso binario)

La idea de esta métrica es poder calcular el error en un dataset para el cual tenemos la **probabilidad estimada** de que cada instancia pertenezca a la **clase positiva** en el caso de clasificación binaria.

logaritmo natural (base e)

$$J_{X,y}(w) =$$

$$\text{Costo_BinCE}_{X,y} = \frac{1}{n} \sum_{i=1}^n \text{Binary_CE}^{(i)}$$

$$\text{Binary_CE}^{(i)} = -[y^{(i)} \log(\hat{h}^{bin}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{h}^{bin}(x^{(i)}))]$$

$$\text{Binary_CE}^{(i)} = \begin{cases} -\log(\hat{P}(Y = 1 | X = x^{(i)})) & y^{(i)} = 1 \\ -\log(\hat{P}(Y = 0 | X = x^{(i)})) & y^{(i)} = 0 \end{cases}$$

equivalentemente

$P(Y=1 X=x^{(i)})$	$y^{(i)}$	$\text{Binary_CE}^{(i)}$	$\text{Costo_BinCE}_{X,y} = 2.249 / 5 = 0.45$
0.775	1	$-\ln(0.775) \sim 0.255$	
0.116	0	$-\ln(1-0.116) \sim 0.123$	
0.884	1	$-\ln(0.884) \sim 0.123$	
0.744	0	$-\ln(1-0.774) \sim 1.48$	
0.320	0	$-\ln(1-0.320) \sim 0.385$	

Entropía cruzada (multiclas)

$$\hat{h}(x^{(i)}) = \text{softmax}(f^{(L)}(f^{(L-1)} \dots (x^{(i)}))) = \begin{bmatrix} \hat{P}(Y = 0|X = x^{(i)}) \\ \hat{P}(Y = 1|X = x^{(i)}) \\ \vdots \\ \hat{P}(Y = C|X = x^{(i)}) \end{bmatrix}$$

Una métrica muy utilizada en problemas de clasificación que permite calcular el error en un dataset para el cual tenemos la probabilidad estimada **para cada clase**.

$$J_{X,y}(w) =$$

$$\text{Costo_CE}_{X,y} = \frac{1}{n} \sum_{i=1}^n \text{CE}^{(i)}$$

$$\text{CE}^{(i)} = H(y^{(i)}, \hat{h}(x^{(i)})) = - \sum_{c=1}^C y_c^{(i)} \log \hat{h}_c(x^{(i)}) \text{ en donde } \hat{h}_c(x^{(i)}) \text{ representa la probabilidad asignada a la clase } c$$

$y^{(i)}$ representa un vector de ceros salvo en la posición de la clase c , en donde vale 1.

equivalentemente

$$\text{CE}^{(i)} = \begin{cases} -\log(\hat{P}(Y = 0|X = x^{(i)})) & y_1^{(i)} = 1 \\ -\log(\hat{P}(Y = 1|X = x^{(i)})) & y_2^{(i)} = 1 \\ \dots & \\ -\log(\hat{P}(Y = C|X = x^{(i)})) & y_C^{(i)} = 1 \end{cases}$$

Notar que, como en caso binario, sólo un término sobrevive para cada instancia (todos los términos de la sumatoria son cero salvo uno).

$$y_c^{(i)} = \begin{cases} 1 & \text{si } c = c^* \\ 0 & \text{en otro caso} \end{cases}$$

$$H(y^{(i)}, \hat{h}(x^{(i)})) = -\log \hat{h}_{c^*}(x^{(i)})$$

TIP, para investigar: CE y Negative Log-Likelihood (NLL) son lo mismo.

$$H(y^{(i)}, \hat{h}(x^{(i)})) = -\log \hat{h}_{c^*}(x^{(i)})$$

Comentario sobre Cross-Entropy vs KL-Divergence

Encontrarán problemas de deep learning en donde se utiliza **la divergencia de Kullback-Leibler (KL)**. “cuánto se desvía una distribución de probabilidad Q (**por ejemplo, la salida del modelo**) de una distribución verdadera P (**por ejemplo, las etiquetas reales**)” Ver: <https://www.youtube.com/watch?v=KHVR587oW8I>

$$D_{KL}(y^{(i)} \parallel \hat{h}(x^{(i)})) = \sum_{c=1}^C y_c^{(i)} \log \left(\frac{y_c^{(i)}}{\hat{h}_c(x^{(i)})} \right) = H(y^{(i)}, \hat{h}(x^{(i)})) - H(y^{(i)})$$

Y si $y^{(i)}$ es one-hot (es decir, solo una clase c^* tiene valor 1), entonces:

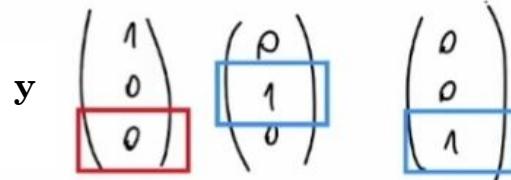
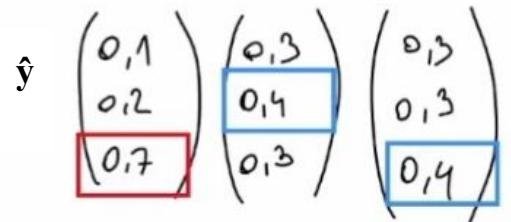
$$H(y^{(i)}) = 0 \quad (\text{porque no hay incertidumbre en un one-hot})$$

$$D_{KL}(y^{(i)} \parallel \hat{h}(x^{(i)})) = -\log \hat{h}_{c^*}(x^{(i)}) \quad \text{Es decir, la Cross Entropy}$$

En segundo lugar, minimizar una significa minimizar la otra (esto vale aunque y no sea one-hot)

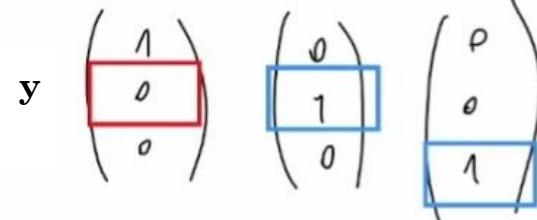
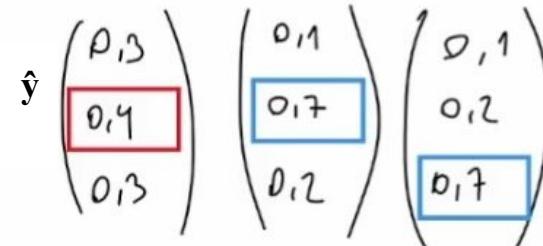
$$\begin{aligned} \nabla \text{KL}(y \parallel p) &= \nabla H(y, p) - \nabla H(y) = \nabla H(y, p) - 0 = \\ \nabla H(y, p) \end{aligned}$$

Entropía cruzada vs Accuracy



Accuracy = $\frac{2}{3}$

Entropía cruzada = 4.14



Accuracy = $\frac{2}{3}$

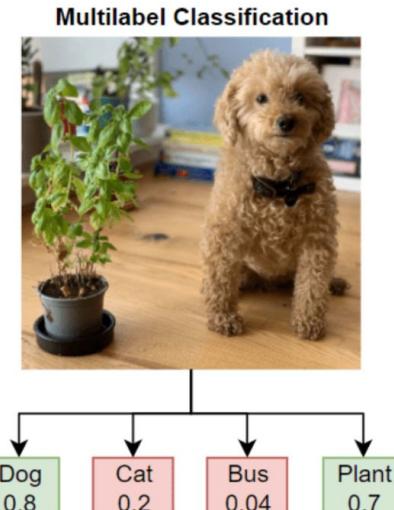
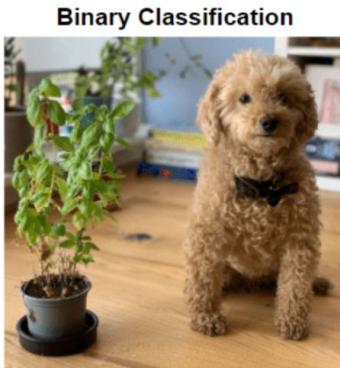
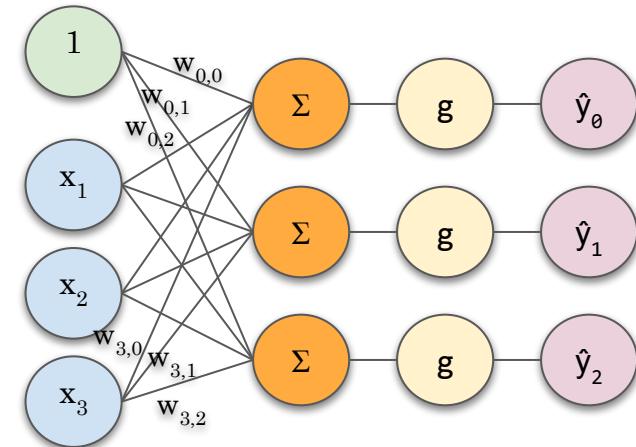
Entropía cruzada = 1.92

Redes Neuronales Multi Label

Redes Neuronales Multi-label

Hay problemas en los cuales tenemos más de una etiqueta por instancia.

- ¿Qué g se les ocurre utilizar en este caso?
- ¿Cómo definirían la función de pérdida?
- ¿Qué métrica se podría usar para medir el error bajo este problema? Buscar por ejemplo: Precision at k (P@k).



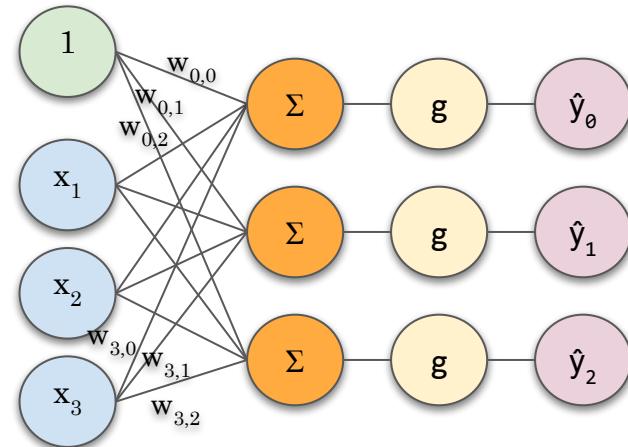
Redes Neuronales Multi-label

Hay problemas en los cuales tenemos más de una etiqueta por instancia.

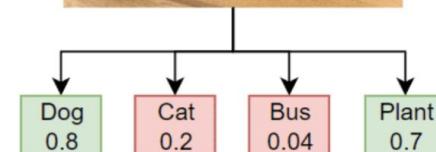
- ¿Qué g se les ocurre utilizar en este caso?
- ¿Cómo definirían la función de pérdida?
- ¿Qué métrica se podría usar para medir el error bajo este problema? Buscar por ejemplo: Precision at k (P@k).

$g = \text{sigmoid}$

Pregunta ¿qué ventaja tiene entrenar todo en una sola red y no múltiple redes binarias?



Multilabel Classification



Predicción de secuencias

Clasificación de documentos

Imaginen que tienen que clasificar el sentimiento (positivo o negativo) para:

Esta tiene que ser una de las peores películas de los años 90. Cuando mis amigos y yo estábamos viendo esta película (siendo el público objetivo al que estaba dirigida), simplemente nos sentamos y miramos la primera media hora con la mandíbula tocando el suelo por lo malo que era en realidad. El resto del tiempo, todos los demás en el cine simplemente empezaban a hablar entre ellos, se iban o, en general, lloraban sobre sus palomitas de maíz...

¿Qué features podemos extraer?

Solución “Bag of Words”

- Bag of words:
 - Los atributos de una instancia, se calculan mirando si una palabra aparece o no entre las 10.000 palabras más frecuentes del idioma. Se almacena el resultado en un vector de dimensión 10.000. $x^{(i)} \in \mathbb{R}^{10000}$
 - “hola hola cómo estás” → <0, 0, 0, 0, 0, ..., 0, 1, 0, ..., 1, ..., 0, 1, 0, .>...
 - Podría ser también la frecuencia relativa (al contar y luego dividir por el largo del documento)
- Entrenar un clasificador $h : X \rightarrow Y$:
 - $X \in \mathbb{R}^{10000}$
 - $Y \in \{\text{positivo, negativo}\}$
- Desventajas:
 - ¿Qué pasa con el orden de las palabras? ¿importa? Por ej, está bien que produzcan el mismo vector:
 - “sí, me gustó mucho, no la vi completa”
 - “sí, no me gustó mucho, la vi completa”
 - Alternativas:
 - A) Usar un modelo como **“bag-of-n-grams”** (en vez de mirar cada palabra por separado, se miran **n** palabras consecutivas. Ej, $n=3$, ¿aparece **(me-gusto-mucho)**? ¿aparece **(no-me-gusto)**? , etc.
 - B) Tratar a la instancia **como una secuencia**, teniendo en cuenta todas las palabras tanto en el contexto de las que la precedieron, como en el de las que le siguen.

Paréntesis: Una visualización vertical de redes

Otra forma de visualizar las redes que conocemos

$$\begin{aligned} h^{(k)} &= g_1(Ux^{(k)}) \\ \hat{y}^{(k)} &= g_2(Vh^{(k)}) \end{aligned}$$

\mathbb{R}^C

$$\hat{y}^{(k)} = g(Ux^{(k)})$$

\mathbb{R}^C

$\hat{y}^{(k)}$

$g(Ux^{(k)})$

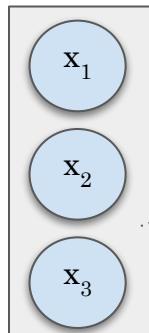
\mathbb{R}^H

$h^{(k)}$

$g_1(Ux^{(k)})$

V

(durante la clase ignoraremos el término bias en las fórmulas, pero siempre está!)



Red neuronal simple

\mathbb{R}^p

$x^{(k)}$

Red neuronal con una capa oculta

\mathbb{R}^p

$x^{(k)}$

\mathbb{R}^H

$h^{(k)}$

$g_1(Ux^{(k)})$

U

Red neuronal con una capa oculta y su función de pérdida

\mathbb{R}^p

$x^{(k)}$

\mathbb{R}^H

$h^{(k)}$

$g_1(Ux^{(k)})$

V

\mathbb{R}^C

$y^{(k)}$

L

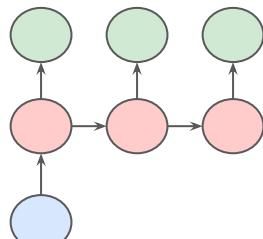
Podemos hacer explícita la función de pérdida y la etiqueta real

Definición del problema: Predicción de secuencias.

Modelos para secuencias de datos, que tienen aplicaciones en tareas como:
pronóstico del tiempo, reconocimiento de voz, traducción de idiomas, predicción de series temporales para finanzas, etc.

Tipos de modelos:

Uno a muchos

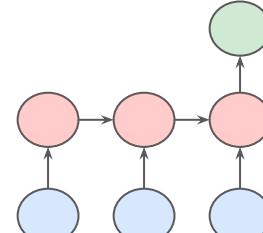


Ej: Descripción imagen

$x^{(k)}$ = imagen

$y^{(k)}$ = ["veo", "un", "perro"]

Muchos a uno

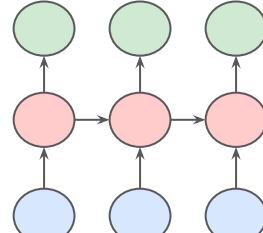


Ej: Predecir el sentimiento

$x^{(k)}$ = ["sí", "me", "encantó"]

$y^{(k)}$ = positivo

Muchos a muchos

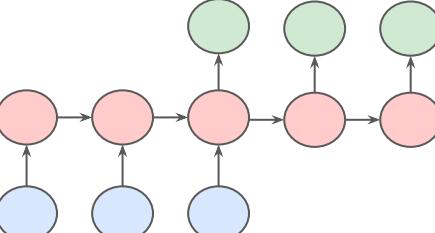


Ej: Precio dólar en un día

$x^{(k)}$ = [x8am, x9am, x10am]

$y^{(k)}$ = [\$800, \$802, \$830]

Muchos a muchos



Ej: Continuar la oración

$x^{(k)}$ = ["sí", "me", "encantó"]

$y^{(k)}$ = ["la", "película", "Raúl"]

En la clase trabajaremos con los modelos como "Muchos a muchos" que es lo más general.

Definición del problema

Paréntesis: Word embeddings

Nota: "Word Embeddings".

Transformaciones de palabras a vectores que las representan "bien semánticamente".

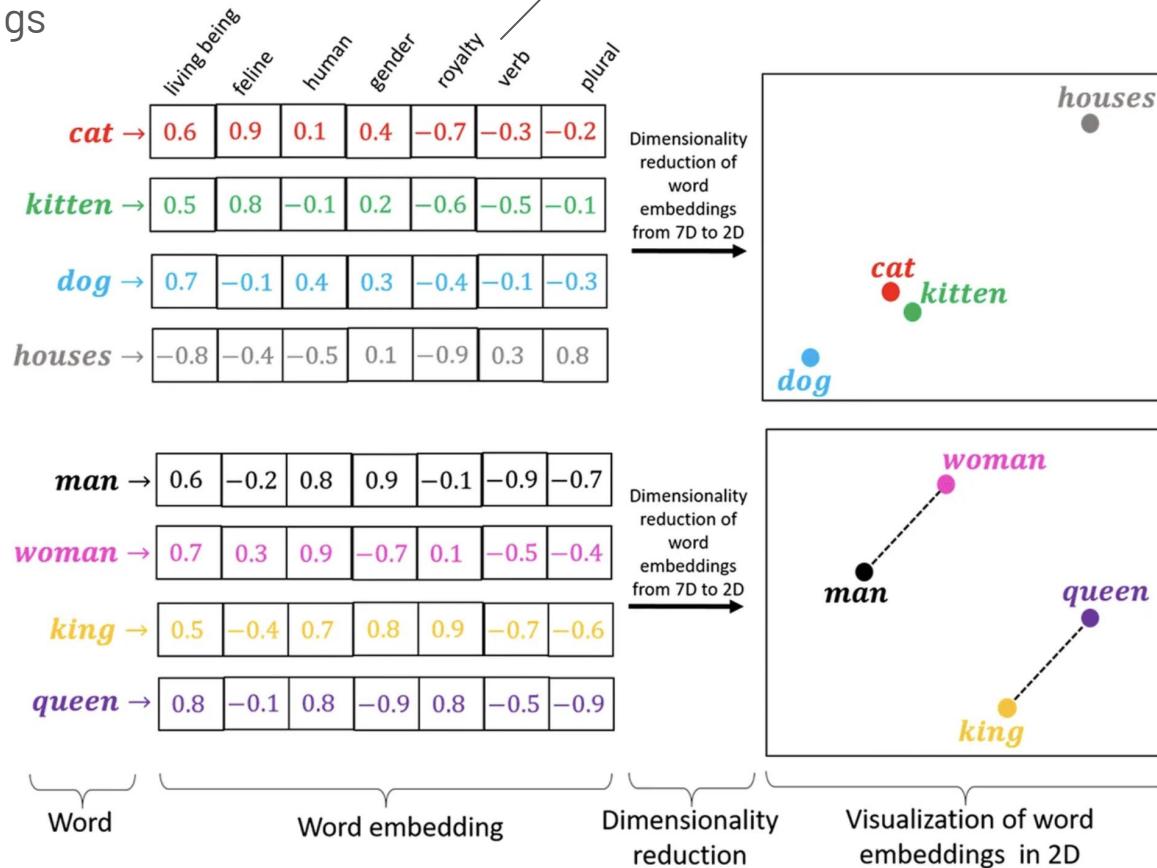
Dos vectores están cerca si su significado es similar.

Word-to-vec, Bert, etc.

```
x(k) = <  
embedding("hola"),  
embedding("cómo"),  
embedding("estás")>
```

en la práctica no ocurre que estas dimensiones representen conceptos tan claros.

fuente: <https://medium.com/@hari4om/word-embedding-d816f643140>



Predicción de secuencias.

Redefinición de "instancias"

La k-ésima instancia es ahora
 $\langle \mathbf{x}^{(k)} \in \mathbb{R}^{pxT}; \mathbf{y}^{(k)} \in \mathbb{R}^{CxT} \rangle$ una serie temporal de entrada junto a su etiqueta, una serie temporal de etiquetas.

Ambas de longitud T.

$\mathbf{x}^{(k)}_{\langle 1 \rangle} \in \mathbb{R}^p$ un vector que representa atributos para el momento 1 de la instancia k.

$\mathbf{o}^{(k)}_{\langle 1 \rangle} \in \mathbb{R}^C$ la predicción en el momento 1 de la instancia k.

$\mathbf{y}^{(k)}_{\langle 1 \rangle} \in \mathbb{R}^C$ la etiqueta en el momento 1 de la instancia k.

	t=1	t=2	t=3	t=4	t=5	t=6
Etiqueta real $\mathbf{y}^{(k)}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
Predicción $\mathbf{o}^{(k)}$	$\begin{bmatrix} 0.1 \\ 0.9 \end{bmatrix}$	$\begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}$	$\begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}$	$\begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$	$\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$	$\begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix}$
Entrada $\mathbf{x}^{(k)}$	$\begin{bmatrix} 1.23 \\ 2.34 \\ 3.45 \\ 4.56 \\ 5.67 \end{bmatrix}$	$\begin{bmatrix} 2.13 \\ 3.24 \\ 4.35 \\ 5.46 \\ 6.57 \end{bmatrix}$	$\begin{bmatrix} 3.12 \\ 4.23 \\ 5.34 \\ 6.45 \\ 7.56 \end{bmatrix}$	$\begin{bmatrix} 4.21 \\ 5.32 \\ 6.43 \\ 7.54 \\ 8.65 \end{bmatrix}$	$\begin{bmatrix} 5.31 \\ 6.42 \\ 7.53 \\ 8.64 \\ 9.75 \end{bmatrix}$	$\begin{bmatrix} 6.41 \\ 7.52 \\ 8.63 \\ 9.74 \\ 10.85 \end{bmatrix}$

En este ejemplo $p = 5, C = 2, T=6$

Predicción de secuencias.

Cálculo del error

El costo para $\mathbf{x}^{(k)}, \mathbf{y}^{(k)}$ es la suma de las pérdidas en todos los pasos de tiempo.

El costo total $J_{X,y}$ (lo que finalmente queremos minimizar con respecto a los pesos) sigue siendo el promedio de los costos de las instancias (que en este caso son secuencias).

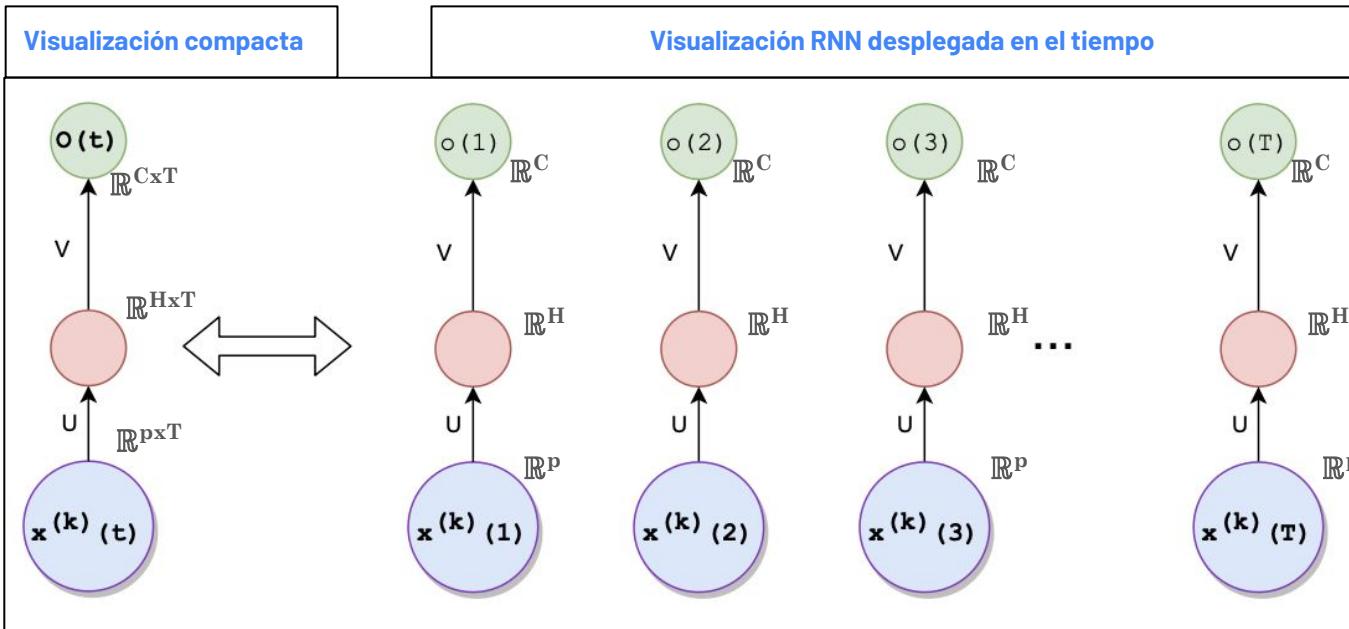
$$\begin{aligned} L(\mathbf{o}^{(k)}, \mathbf{y}^{(k)}) &= L(\{\mathbf{o}_{\langle 1 \rangle}^{(k)} \dots \mathbf{o}_{\langle T \rangle}^{(k)}\}, \{\mathbf{y}_{\langle 1 \rangle}^{(k)} \dots \mathbf{y}_{\langle T \rangle}^{(k)}\}) \\ &= \sum_{t=1}^T L(\mathbf{o}_{\langle t \rangle}^{(k)}, \mathbf{y}_{\langle t \rangle}^{(k)}) \\ J_{X,y} &= \frac{1}{n} \sum_{i=1}^n L(\mathbf{o}^{(i)}, \mathbf{y}^{(i)}) \end{aligned}$$

Etiqueta real $\mathbf{y}^{(k)}$	0	1	0	0	0	1
	[0]	[1]	[0]	[1]	[1]	[0]
Predicción $\mathbf{o}^{(k)}$	0.1	0.8	0.4	0.7	0.5	0.6
	[0.1]	[0.8]	[0.4]	[0.7]	[0.5]	[0.6]
	t=1	t=2	t=3	t=4	t=5	t=6
Entrada $\mathbf{x}^{(k)}$	1.23 2.34 3.45 4.56 5.67	2.13 3.24 4.35 5.46 6.57	3.12 4.23 5.34 6.45 7.56	4.21 5.32 6.43 7.54 8.65	5.31 6.42 7.53 8.64 9.75	6.41 7.52 8.63 9.74 10.85

Redes Neuronales Recurrentes

Redes Neuronales Recurrentes

Computemos el output para cada momento del tiempo como una función de la entrada en ese momento (**independiente al resto**).



Fórmula que describe esta red:
 (omitimos los biases (w_0) y suponemos
 expansión con 1's como veníamos haciendo)

$$\begin{aligned} \mathbf{h}_{\langle t \rangle}^{(k)} &= g_1(U \mathbf{x}_{\langle t \rangle}^{(k)}) \\ \mathbf{o}_{\langle t \rangle}^{(k)} &= g_2(V \mathbf{h}_{\langle t \rangle}^{(k)}) \end{aligned}$$

Notación: $\mathbf{x}_{\langle t \rangle}^{(k)}$

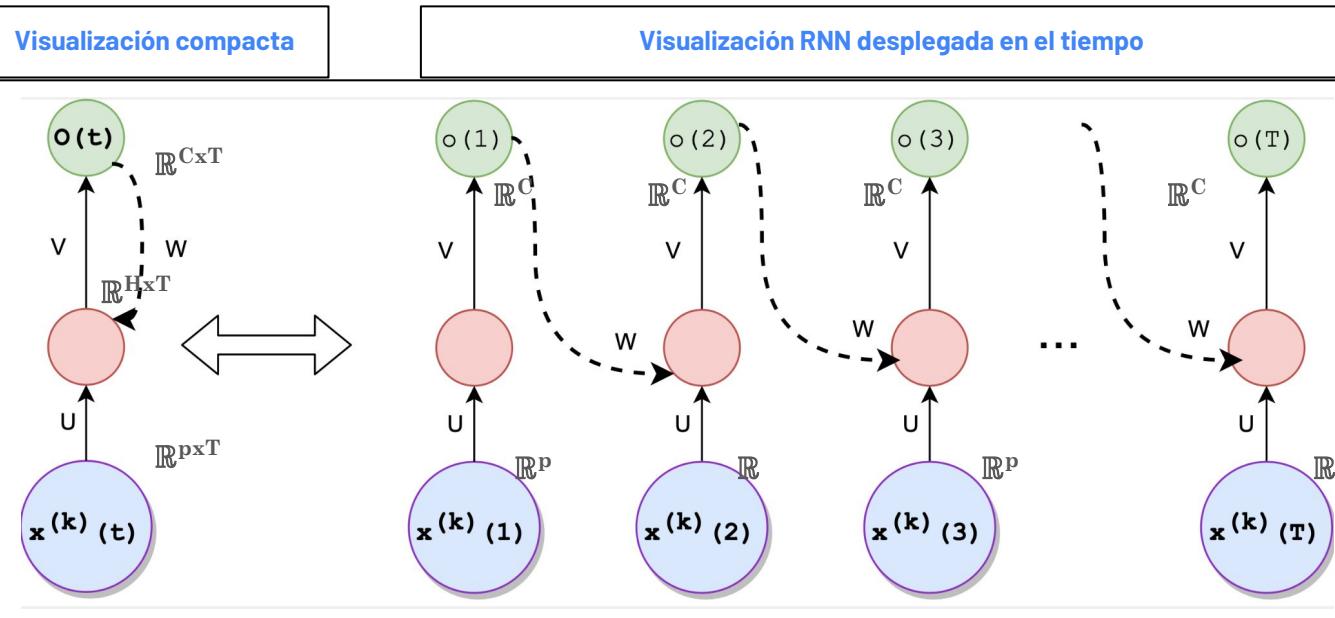
k-ésima instancia (que es una secuencia) en el momento t (un vector de atributos de dimensión p).

Si entrenamos esta red **no podría aprender** qué hacer según información del pasado / futuro de la secuencia.

¿Puede servir?

Redes Neuronales Recurrentes

Computemos el output para cada momento del tiempo como una función de la entrada en ese momento **y el output anterior**.



Fórmula que describe esta red:
(omitimos los biases (w_0) y suponemos
expansión con 1's como veníamos haciendo)

$$\begin{aligned}\mathbf{h}_{\langle t \rangle}^{(k)} &= g_1(W \mathbf{o}_{\langle t-1 \rangle}^{(k)}) + U \mathbf{x}_{\langle t \rangle}^{(k)} \\ \mathbf{o}_{\langle t \rangle}^{(k)} &= g_2(V \mathbf{h}_{\langle t \rangle}^{(k)})\end{aligned}$$

Notación: $\mathbf{x}_{\langle t \rangle}^{(k)}$

k-ésima instancia (que es una secuencia) en el momento t (un vector de atributos de dimensión p).

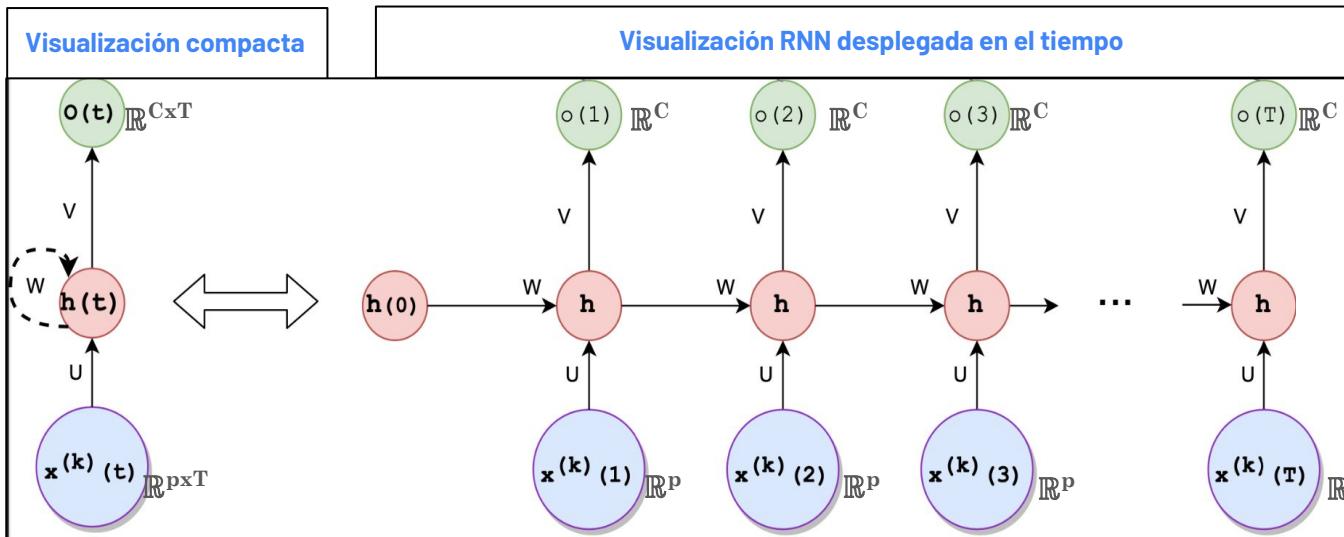
Si entrenamos esta red podría aprender a predecir **según el output anterior**.

¿Qué ventajas / desventajas tiene esta arquitectura?

¿Puede aprender dependencias a más de 1 paso? ¿Cómo?

Redes Neuronales Recurrentes

Computemos el output para cada momento del tiempo como una función de la entrada en ese momento **e información acumulada anterior.**



- **Estado oculto h** con conexiones a la entrada parametrizadas por una matriz de pesos \mathbf{U} .
- **Conexiones recurrentes** parametrizadas por una matriz de pesos \mathbf{W} .
- **Conexiones a la salida** parametrizadas por una matriz de pesos \mathbf{V} .

$$\begin{aligned}\mathbf{h}_{\langle t \rangle}^{(k)} &= g_1(W\mathbf{h}_{\langle t-1 \rangle}^{(k)} + U\mathbf{x}_{\langle t \rangle}^{(k)}) \\ \mathbf{o}_{\langle t \rangle}^{(k)} &= g_2(V\mathbf{h}_{\langle t \rangle}^{(k)})\end{aligned}$$

Notación: $\mathbf{x}_{\langle t \rangle}^{(k)}$

k-ésima instancia (que es una secuencia) en el momento t (un vector de atributos de dimensión p).

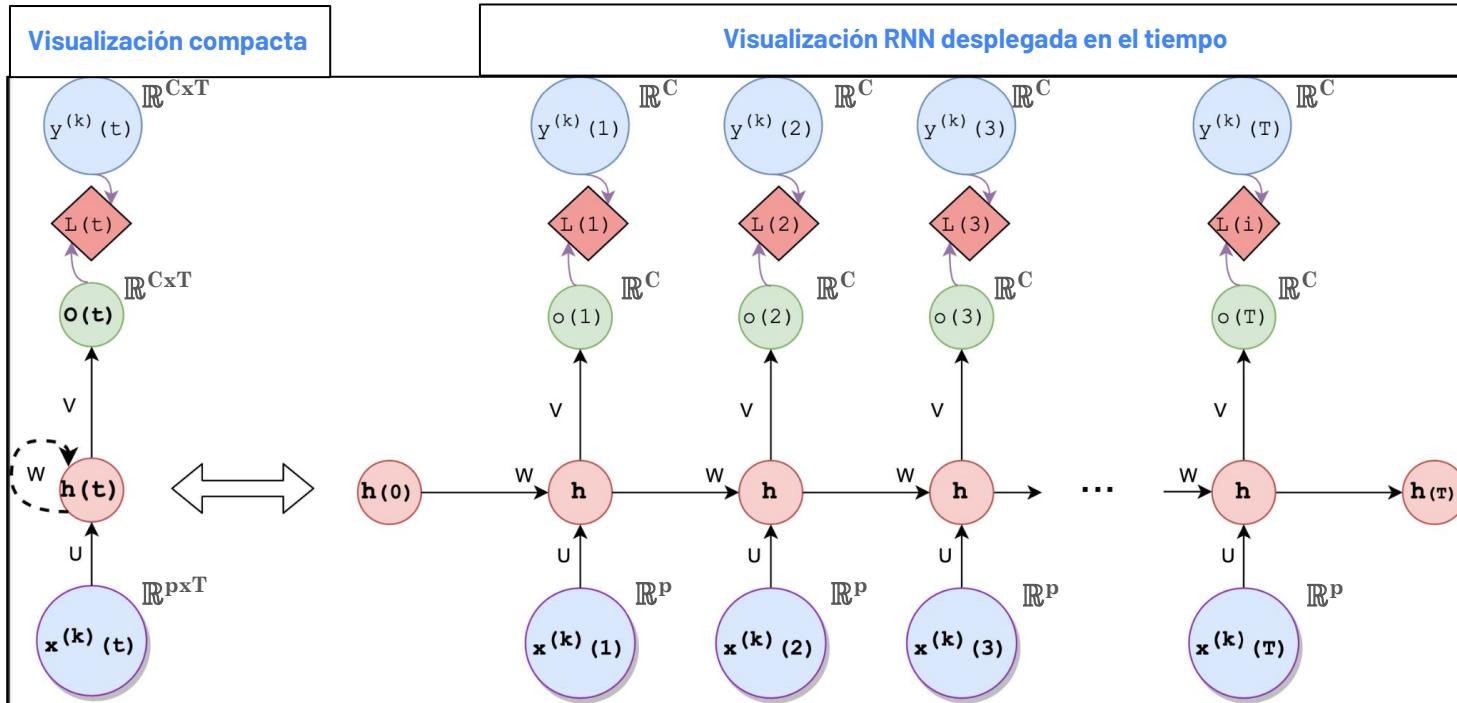
Si entrenamos esta red, podría aprender a predecir **según el estado anterior.**

El estado puede mantener información a largo plazo.

Redes Neuronales Recurrentes

Recordamos, el costo para la serie temporal $\mathbf{x}^{(k)}, \mathbf{y}^{(k)}$
 es la suma de las pérdidas en todos los pasos de tiempo. Entonces:
 ¿Qué habrá que cambiar en el algoritmo de Descenso por el Gradiente?

$$\begin{aligned}
 L(\mathbf{o}^{(k)}, \mathbf{y}^{(k)}) &= L(\{\mathbf{o}_{\langle 1 \rangle}^{(k)} \dots \mathbf{o}_{\langle T \rangle}^{(k)}\}, \{\mathbf{y}_{\langle 1 \rangle}^{(k)} \dots \mathbf{y}_{\langle T \rangle}^{(k)}\}) \\
 &= \sum_{t=1}^T L(\mathbf{o}_{\langle t \rangle}^{(k)}, \mathbf{y}_{\langle t \rangle}^{(k)}) \\
 J_{X,y} &= \frac{1}{n} \sum_{i=1}^n L(\mathbf{o}^{(i)}, \mathbf{y}^{(i)})
 \end{aligned}$$



Redes Neuronales Recurrentes

¿Qué habrá **que cambiar** en el algoritmo de Backpropagation / Descenso por el Gradiente?

CASI NADA

Estas redes definen un grafo computacional en donde **las operaciones conectan** los parámetros de la red con los outputs y por lo tanto se minimiza de igual manera que antes.

Lo que sí sucede es que no podemos **calcular el gradiente en cada momento de tiempo de manera independiente**. Ya que las operaciones del tiempo dependen de las de los tiempos anteriores. (*)

Aunque tiene un nuevo nombre "Backpropagation Through Time" **BPTT**.

Hay variantes, como TBPTT (**Truncated** BPPT).

Problema 1: Cómputo no paraleizable.

Problema 2: Secuencia larga => grafo computacional profundo

Muchas multiplicaciones de los gradientes por matrices + no linealidades.

- **Matrices con máx valor singular < 1** → Vanishing gradient
- **Matrices con máx valor singular > 1** → Exploding gradient

Problema 3: Difícil lograr que la red aprenda dependencias temporales largas.

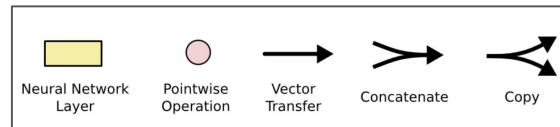
(*) Podríamos para la red que usaba los outputs anteriores como entrada de la siguiente hidden... ¿cómo? buscar sobre: "Teacher Forcing"

```

1. class NuestraRNN(nn.Module): # Código de ejemplo en (quasi-)pytorch (no funciona)
2.     def __init__(self, dim_input, dim_hidden, dim_output):super().__init__()
3.         self.U = nn.Parameter(torch.Tensor(dim_hidden, dim_input)) # mapea Input -> Oculto
4.         self.W = nn.Parameter(torch.Tensor(dim_hidden, dim_hidden)) # mapea Oculto -> Oculto
5.         self.fc = nn.Linear(dim_hidden, dim_output)
6.         self.init_weights()
7.
8.     def forward(self, x):
9.         batch_size, seq_len, input_dim = x.size()
10.        hidden = self.init_hidden() # Inicializamos hidden en h0.
11.        outs = []
12.        for t in range(seq_len):
13.            x_t = x[:, t, :]
14.            hidden = torch.tanh(x_t @ self.U.T + hidden @ self.W.T) # TanH sería nuestra g1
15.            out = self.fc(hidden)
16.            outs.append(out)
17.        return outs, hidden
18.
19. criterion = nn.MSELoss()
20. optimizer = optim.SGD(model.parameters(), lr=learning_rate) # SGD = Gradient Descent
21.
22. for epoch in range(num_epochs):
23.     for batch_X, batch_y in dataloader:
24.         outputs, hidden_final = model(batch_X) # Notar que no usamos hidden_final (podríamos)
25.         loss = criterion(outputs, batch_y)
26.         loss.backward()

```

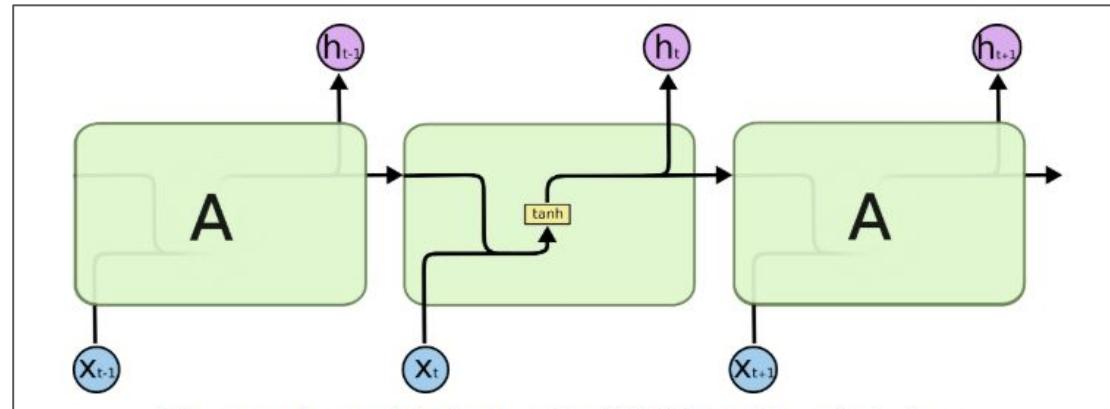
Mecanismos de Memoria



$$\mathbf{h}_{\langle t \rangle}^{(k)} = g_1(W\mathbf{h}_{\langle t-1 \rangle}^{(k)} + U\mathbf{x}_{\langle t \rangle}^{(k)})$$

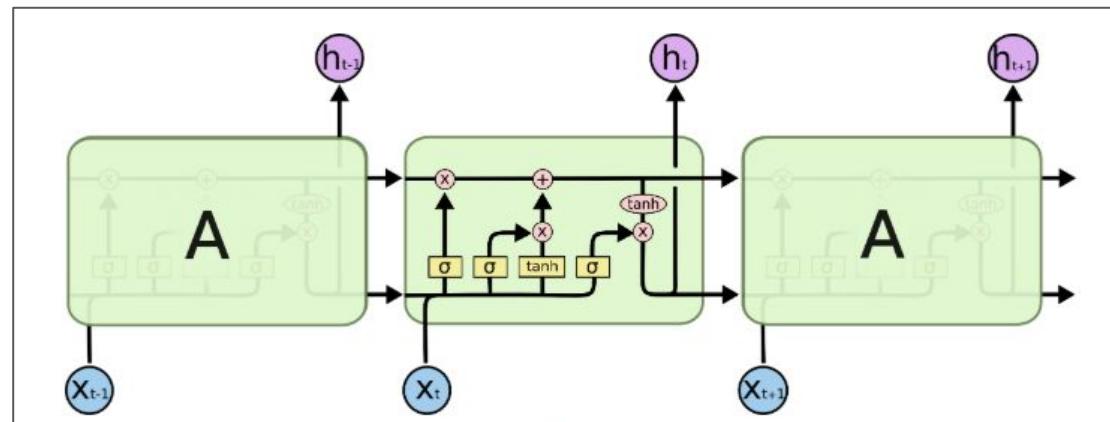
$$\mathbf{o}_{\langle t \rangle}^{(k)} = g_2(V\mathbf{h}_{\langle t \rangle}^{(k)})$$

¿Podemos hacer algo con más capacidad de aprendizaje?



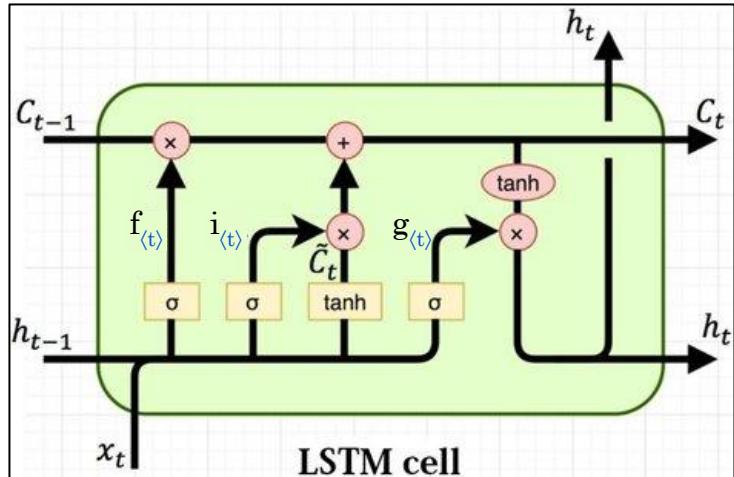
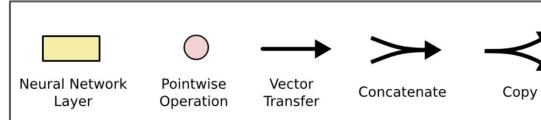
Ahora los estados ocultos producen un "**Cell State**" y un "**Hidden State**".

El hidden sigue siendo utilizado para el output y para el siguiente estado oculto, el cell solo para el próximo estado oculto



Mecanismos de Memoria

Ejemplo: LSTM



$$\begin{aligned}
 \mathbf{i}_{\langle t \rangle}^{(k)} &= \sigma(W^i \mathbf{h}_{\langle t-1 \rangle}^{(k)} + U^i \mathbf{x}_{\langle t \rangle}^{(k)}) \\
 \mathbf{f}_{\langle t \rangle}^{(k)} &= \sigma(W^f \mathbf{h}_{\langle t-1 \rangle}^{(k)} + U^f \mathbf{x}_{\langle t \rangle}^{(k)}) \\
 \mathbf{g}_{\langle t \rangle}^{(k)} &= \sigma(W^g \mathbf{h}_{\langle t-1 \rangle}^{(k)} + U^g \mathbf{x}_{\langle t \rangle}^{(k)}) \\
 \tilde{\mathbf{C}}_{\langle t \rangle}^{(k)} &= \tanh(W^{\tilde{C}} \mathbf{h}_{\langle t-1 \rangle}^{(k)} + U^{\tilde{C}} \mathbf{x}_{\langle t \rangle}^{(k)}) \\
 \mathbf{C}_{\langle t \rangle}^{(k)} &= \mathbf{f}_{\langle t \rangle}^{(k)} \odot \mathbf{C}_{\langle t-1 \rangle}^{(k)} + \mathbf{i}_{\langle t \rangle}^{(k)} \odot \tilde{\mathbf{C}}_{\langle t \rangle}^{(k)} \\
 \mathbf{h}_{\langle t \rangle}^{(k)} &= \mathbf{g}_{\langle t \rangle}^{(k)} \odot \tanh(\mathbf{C}_{\langle t \rangle}^{(k)})
 \end{aligned}$$

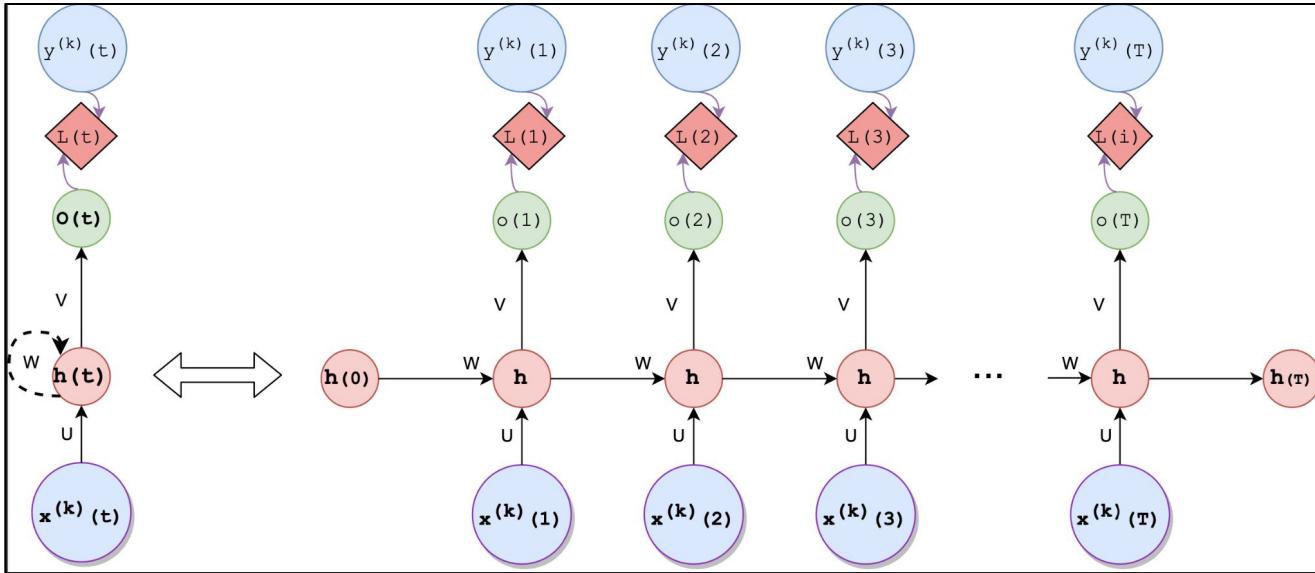
Dos caminos para recordar
 El Hidden para corto plazo. El
 Cell state, para recordar a
 largo plazo.

pd. No pretendemos que sepan esta fórmula de memoria, pero sí la intuición.

Recomiendo: <https://www.youtube.com/watch?v=k6fSgUaWUF8>
 También <http://colah.github.io/posts/2015-08-Understanding-LSTMs>

$$\mathbf{o}_{\langle t \rangle}^{(k)} = g_2(V \mathbf{h}_{\langle t \rangle}^{(k)})$$

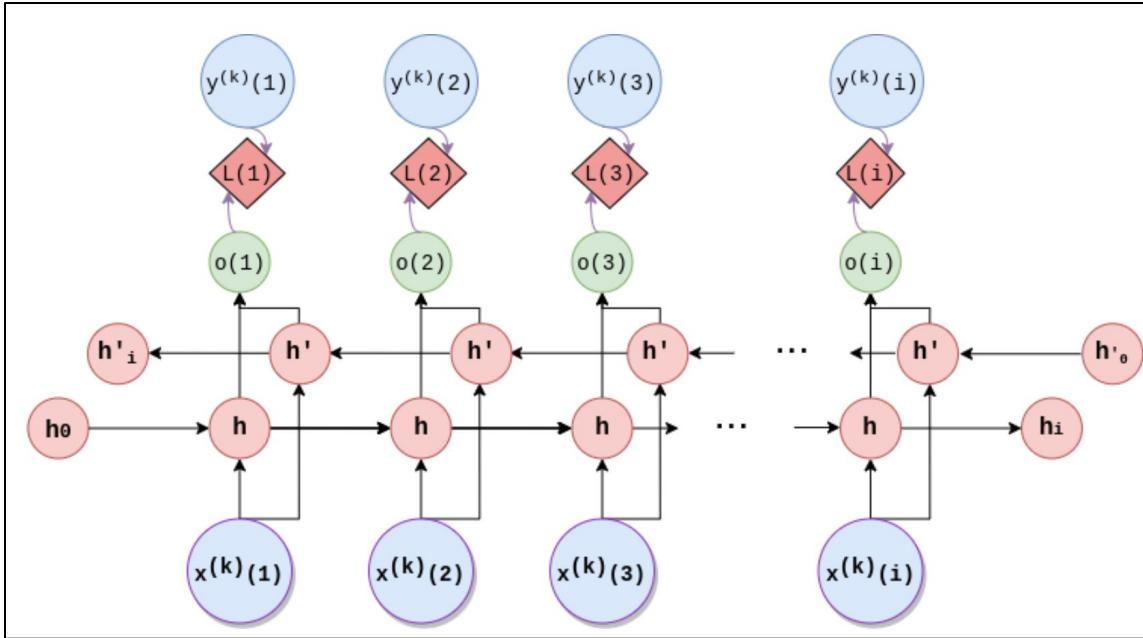
También buscar sobre **células de tipo GRU**



¿Sólo el pasado?

- Las RNN acumulan información a lo largo del tiempo. Las predicciones para cada instante dependen del estado acumulado en función de las entradas anteriores.
- Sin embargo, hay problemas en los cuales es necesario conocer información de la entrada en instantes posteriores al momento de la predicción.

$$\begin{aligned}\mathbf{h}_{\langle t \rangle}^{(k)} &= g_1(W\mathbf{h}_{\langle t-1 \rangle}^{(k)} + U\mathbf{x}_{\langle t \rangle}^{(k)}) \\ \mathbf{o}_{\langle t \rangle}^{(k)} &= g_2(V\mathbf{h}_{\langle t \rangle}^{(k)})\end{aligned}$$



Redes recurrentes bidireccionales (BRNN) (Schuster & Paliwal, 1997).

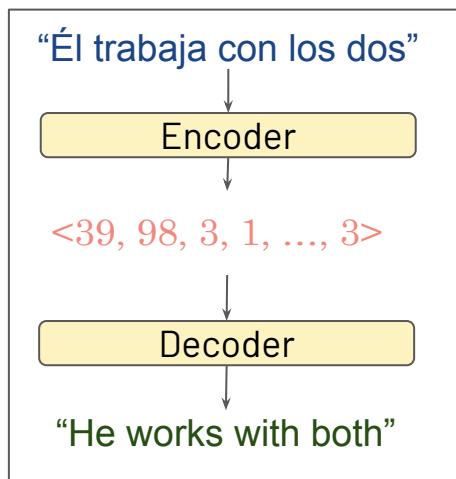
- En una BRNN existen dos estados ocultos, los cuales acumulan información del pasado de la serie temporal y del futuro de la serie, respectivamente.
- Las BRNNs pueden entrenarse utilizando algoritmos similares a las RNNs, debido a que las dos neuronas direccionales no tienen ninguna interacción entre sí.'

$$\begin{aligned}\overrightarrow{\mathbf{h}}_{\langle t \rangle}^{(k)} &= g_1(W \overrightarrow{\mathbf{h}}_{\langle t-1 \rangle}^{(k)} + U \mathbf{x}_{\langle t \rangle}^{(k)}) \\ \overleftarrow{\mathbf{h}}_{\langle t \rangle}^{(k)} &= g_1(W \overleftarrow{\mathbf{h}}_{\langle t+1 \rangle}^{(k)} + U \mathbf{x}_{\langle t \rangle}^{(k)}) \\ \mathbf{o}_{\langle t \rangle}^{(k)} &= g_2(V[\overleftarrow{\mathbf{h}}_{\langle t \rangle}^{(k)}, \overrightarrow{\mathbf{h}}_{\langle t \rangle}^{(k)}])\end{aligned}$$

Arquitecturas Encoder - Decoder

Encoder - Decoder

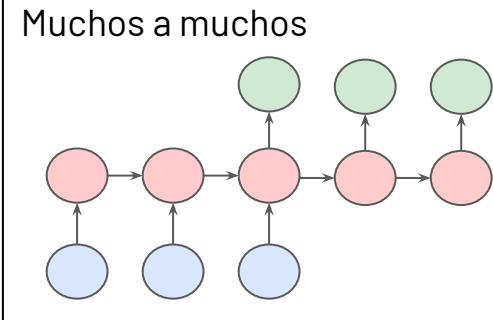
Tipo de arquitectura que se utiliza en tareas como traducción automática, generación de texto, resumen de texto, reconocimiento de voz y más.



Incluye un **codificador (encoder)** y un **decodificador (decoder)**.

El encoder procesa la información de la entrada y la acumula en **estados ocultos** (generando una nueva representación).

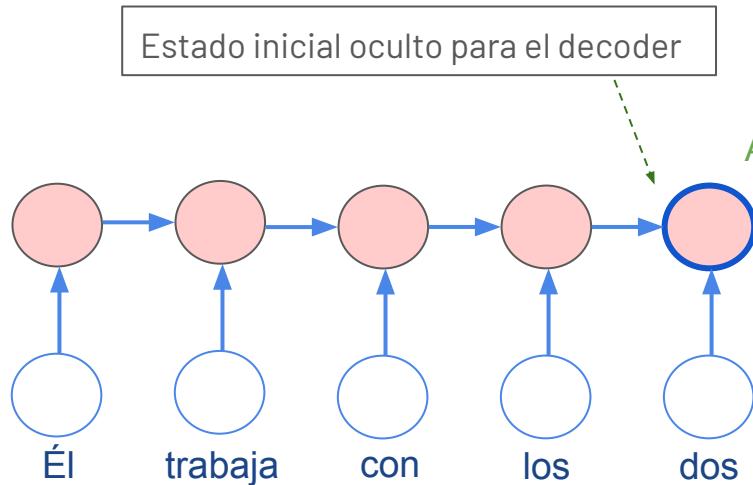
El decoder toma la información provista por el encoder (la nueva representación de la entrada) y **devuelve una secuencia**.



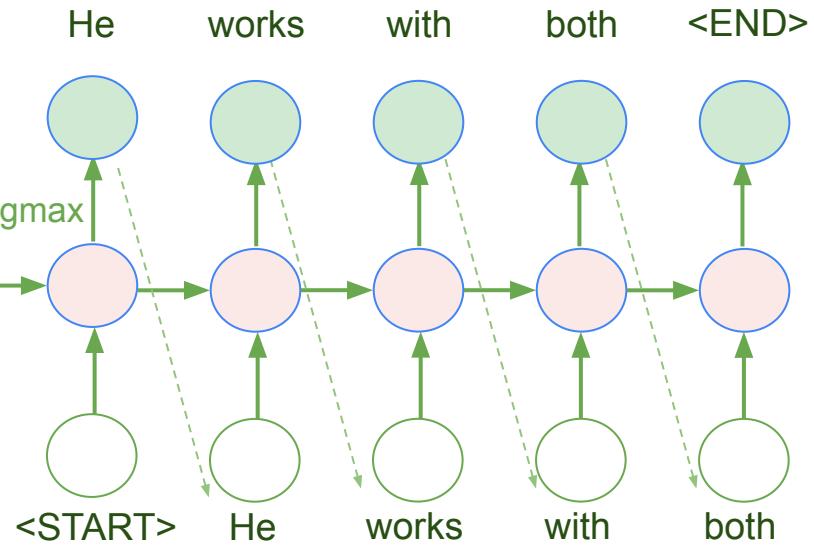
Encoder - Decoder

RNN

Encoder RNN



El **encoder** procesa la información de la entrada y la acumula en estados ocultos



El **decoder** toma la información provista por el encoder y devuelve una secuencia

Decoder RNN

```
1. class EncoderRNN(nn.Module): # Código de ejemplo en (quasi-)pytorch
2.     def __init__(self, input_size, dim_hidden):
3.         ...
4.         self.rnn = nn.RNN(input_size, dim_hidden)
5.
6.     def forward(self, x):
7.         h0 = self.init_hidden()
8.         output, hidden = self.rnn(x, h0)
9.         return output, hidden
10.
11. class DecoderRNN(nn.Module):
12.     def __init__(self, dim_hidden, dim_output):
13.         ...
14.         self.rnn = nn.RNN(dim_output, dim_hidden)
15.         self.fc = nn.Linear(dim_hidden, dim_output)
16.
17.     def forward(self, x, hidden):
18.         output, hidden = self.rnn(x, hidden)
19.         output = self.fc(output)
20.         return output, hidden
```

```
# Create DataLoader for mini-batch gradient descent
dataset = TensorDataset(data, targets)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
1. encoder = EncoderRNN(input_size, hidden_size, num_layers)
2. decoder = DecoderRNN(hidden_size, output_size, num_layers)
3. criterion = nn.MSELoss()
4. encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
5. decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate)
6.

7. for epoch in range(num_epochs):
8.     for batch_data, batch_targets in dataloader:
9.
10.         # Pasada por el Encoder
11.         encoder_outputs, encoder_hidden = encoder(batch_data)
12.
13.         decoder_outputs = []
14.         decoder_input = torch.zeros(batch_size, 1, output_size)
15.         decoder_hidden = encoder_hidden

16.         for t in range(batch_targets.size(1)):
17.             decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden)
18.             loss += criterion(decoder_output, batch_targets[:, t])
19.             decoder_outputs.append(decoder_output)

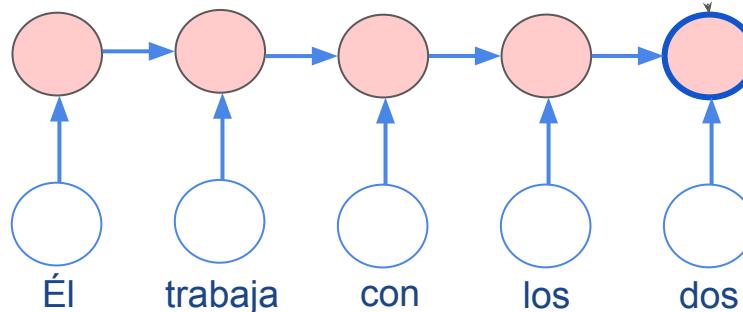
20.             # "Teacher Forcing": hasta cierto epoch, usamos los y reales y no las predicciones
21.             decoder_input = batch_targets[:, t]
22.             loss.backward()
```

Encoder - Decoder

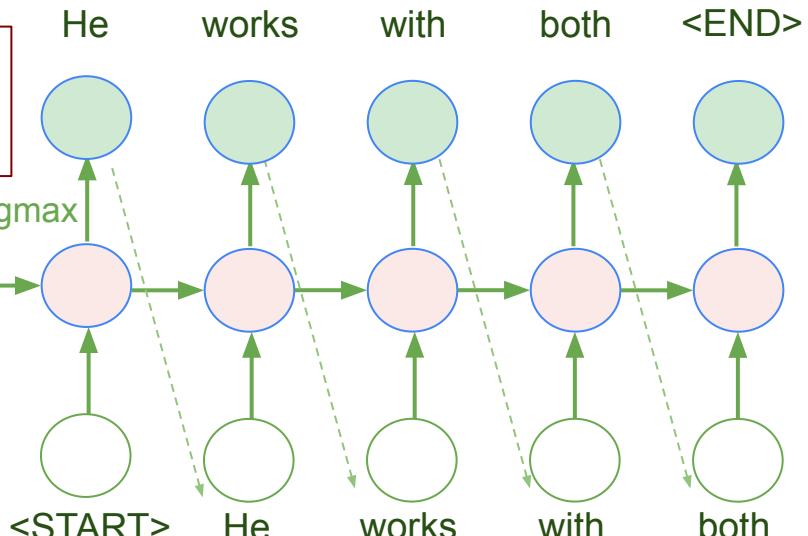
RNN - Problema del bottleneck

Esto es un **cuello de botella**. Un texto relativamente largo va a ser difícil de codificar en un solo vector. Incluso con mecanismos de memoria como LSTM.

Encoder RNN



El **encoder** procesa la información de la entrada y la acumula en estados ocultos



El **decoder** toma la información provista por el encoder y devuelve una secuencia

Decoder RNN

Atención (ver.2014)

Encoder - Decoder

RNN + Atención

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

(2014)

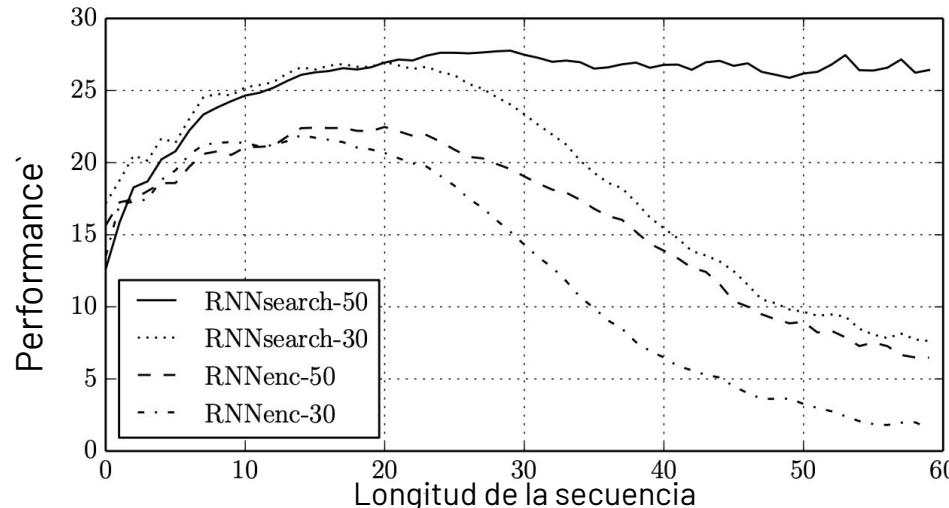
Dzmitry Bahdanau

Jacobs University Bremen, Germany

KyungHyun Cho Yoshua Bengio*

Université de Montréal

Del abstract: "... conjeturamos que el uso de un **vector de longitud fija es un cuello de botella para el aprendizaje** ... proponemos ampliarlo permitiendo que el modelo **busque automáticamente partes de una oración fuente que son relevantes para predecir una palabra objetivo ...**"



Encoder - Decoder

Resumen

En cada paso del decodificador:

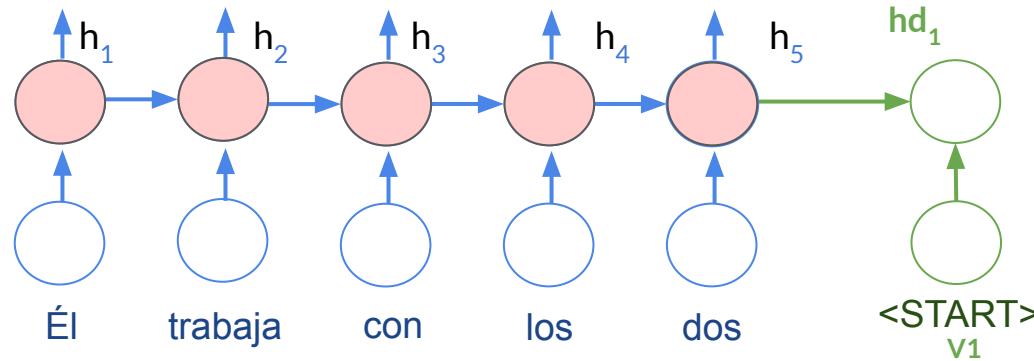
- Recibe entrada de atención: un estado de decodificador hd_i , y todos los estados del codificador $h_1, h_2, h_3 \dots h_T$
- Calcula puntuaciones de atención:
 - Para cada estado del codificador h_j , la atención calcula su "relevancia" para el estado hd_i del decodificador: $\text{similitud}(hd_i, h_j)$Formalmente, aplica una función de atención que recibe un estado de decodificador y un estado de codificador y devuelve un valor escalar.
- Calcula pesos de alineación: una distribución de probabilidad (softmax aplicado a puntuaciones);
- Calcula el vector de contexto: la suma ponderada de los estados del codificador y los pesos.

"Atención": en diferentes pasos, dejar que el modelo se "centre" en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención

Encoder RNN



Decoder RNN

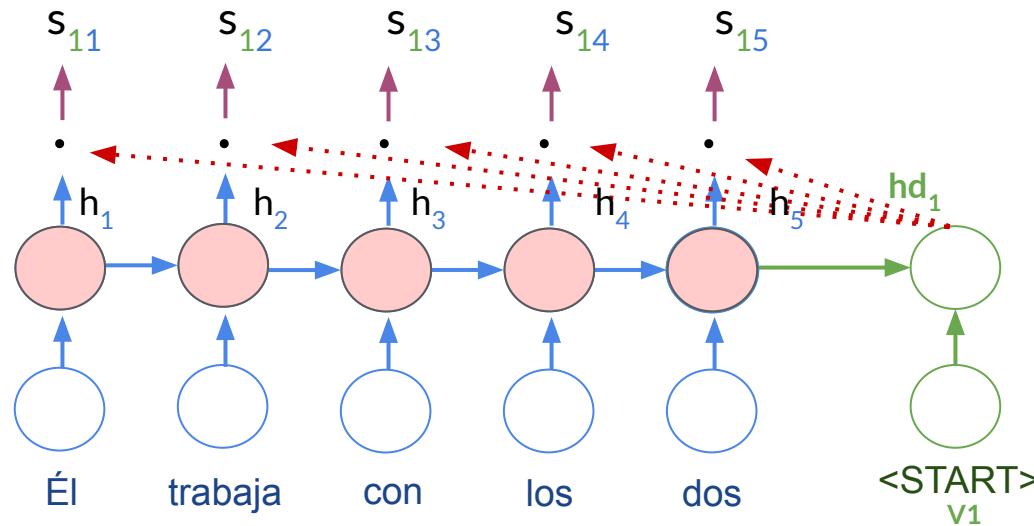
"Atención": en diferentes pasos, dejar que el modelo se "centre" en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención

$$s_{ij} = \text{similitud}(\mathbf{h}_{\mathbf{d}\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{\mathbf{d}\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

Encoder RNN

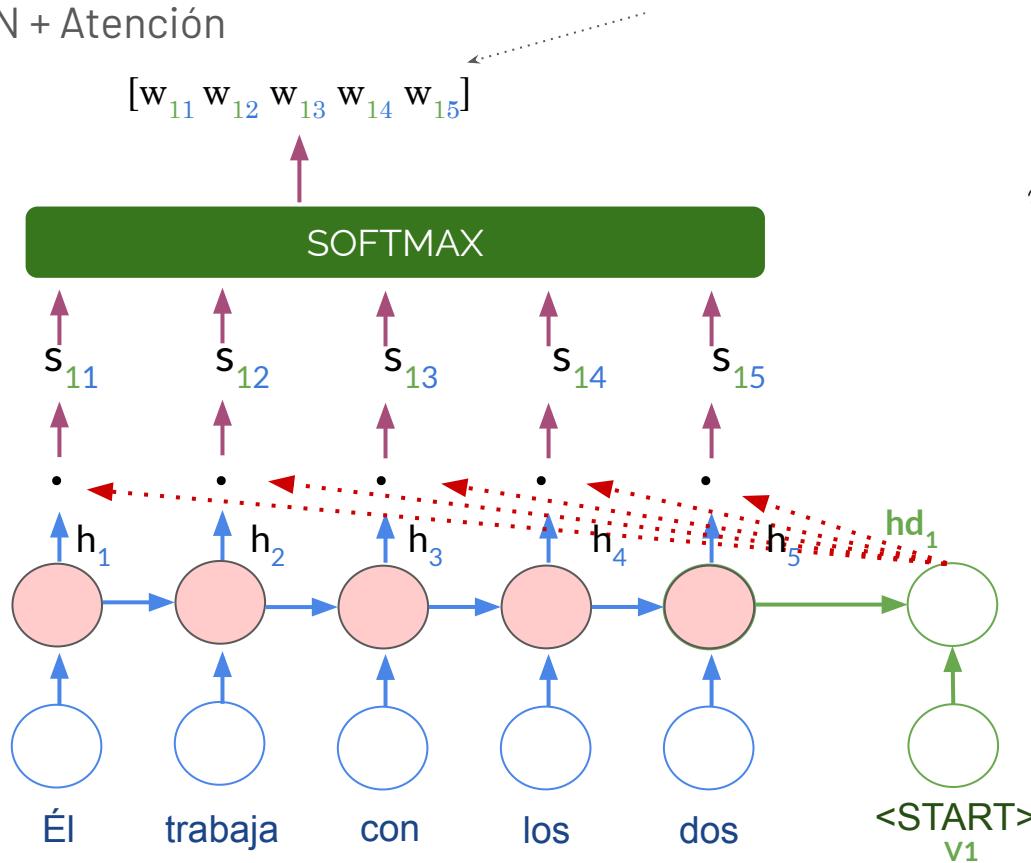


Decoder RNN

“Atención”: en diferentes pasos, dejar que el modelo se “centre” en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención



Los w_{ij} (pesos de alineación) representan la

w_{13} : importancia relativa del estado 3 del encoder para decodificar en el instante 1.
No son pesos de la red, me pareció bien usar w de todas maneras (se usa a en general).

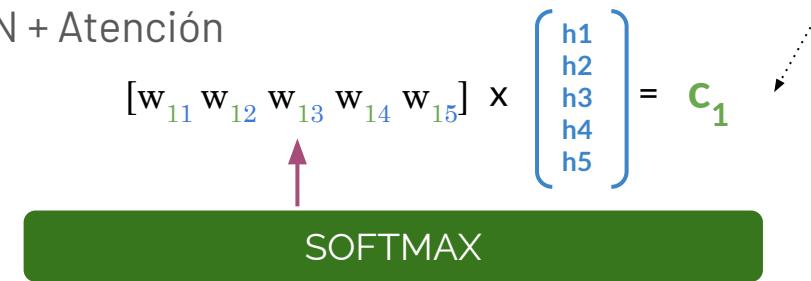
$$s_{ij} = \text{similitud}(\mathbf{h}_{d\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{d\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}}$$

“Atención”: en diferentes pasos, dejar que el modelo se “centre” en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención



Vector de contexto

c_1 : representación útil para hd_1 de la entrada completa.

Es una suma ponderada $\dim(c_i) = \dim(h_{\langle j \rangle})$

$$s_{ij} = \text{similitud}(\mathbf{h}_{d_{\langle i \rangle}}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{d_{\langle i \rangle}} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

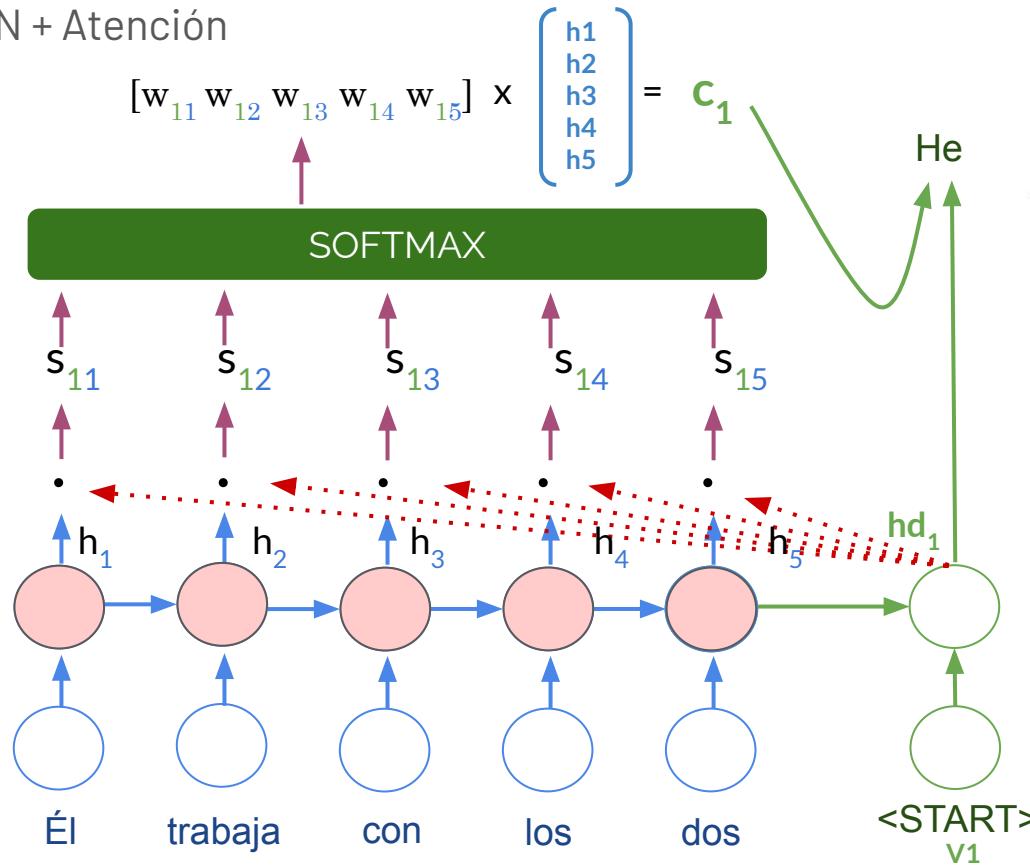
$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}$$

$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)} \quad \text{vector de contexto}$$

"Atención": en diferentes pasos, dejar que el modelo se "centre" en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención



$$s_{ij} = \text{similitud}(\mathbf{h}_d^{(k)}_{\langle i \rangle}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_d^{(k)}_{\langle i \rangle} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}}$$

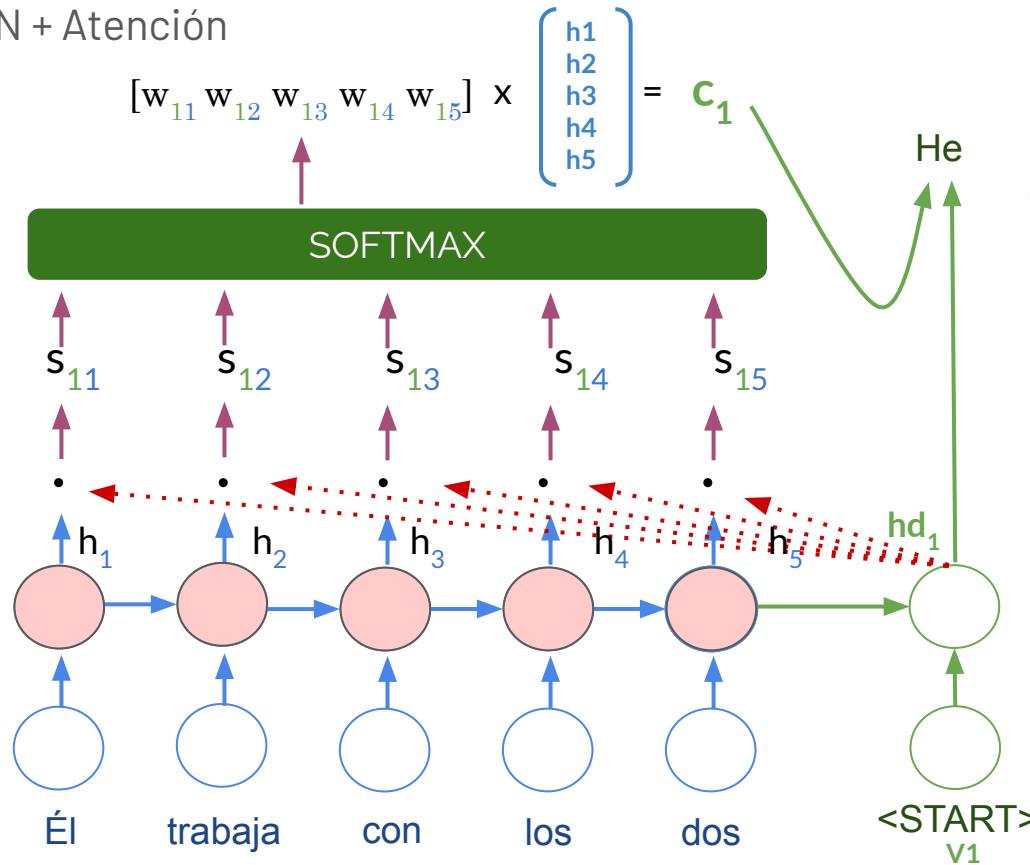
$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)} \quad \text{vector de contexto}$$

$$u_i = \text{concat}(c_i, \mathbf{h}_d^{(k)}_{\langle i \rangle})$$

“Atención”: en diferentes pasos, dejar que el modelo se “centre” en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención



$$s_{ij} = \text{similitud}(\mathbf{h}_{d\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{d\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}}$$

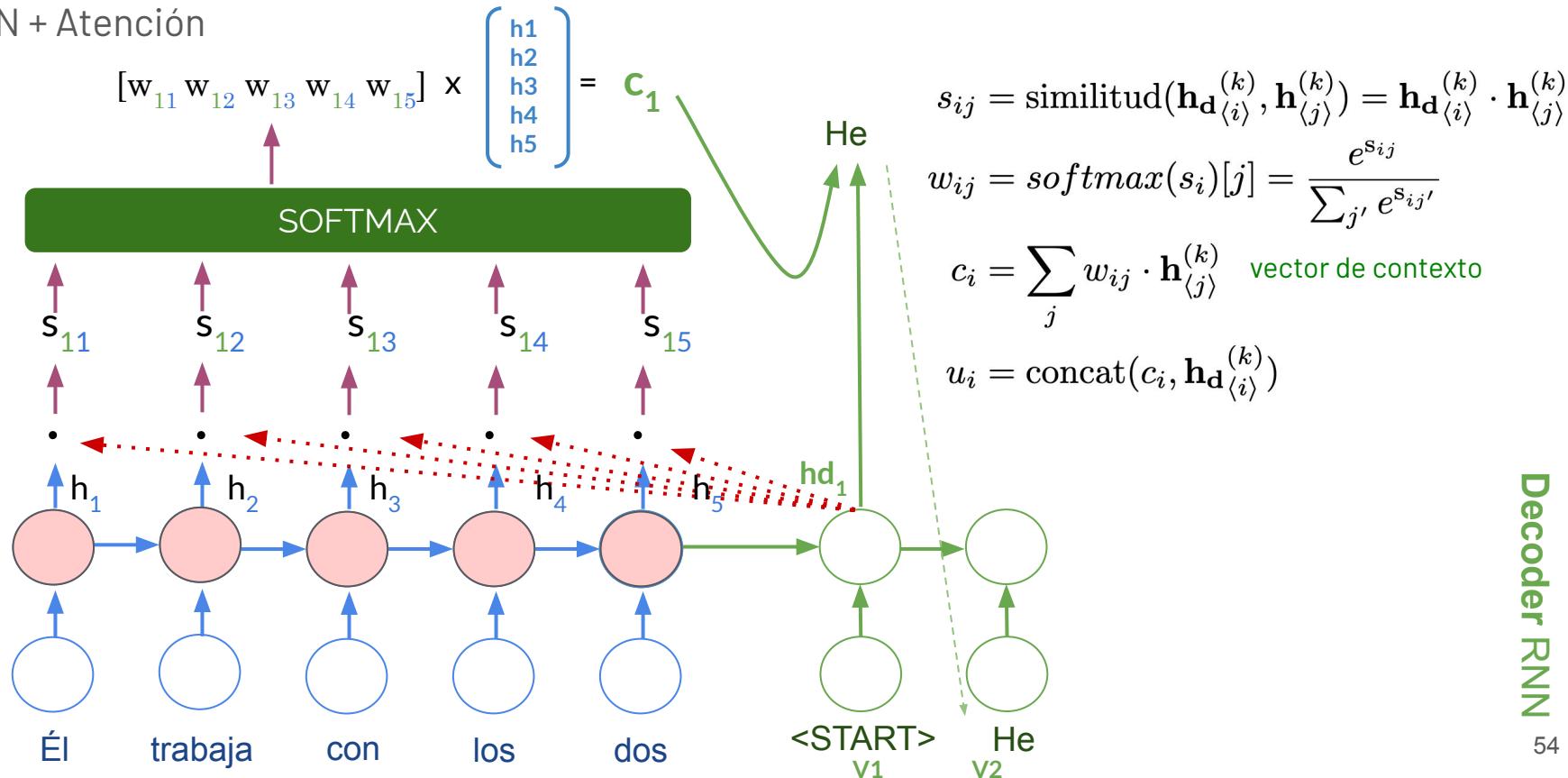
$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)} \quad \text{vector de contexto}$$

$$u_i = \text{concat}(c_i, \mathbf{h}_{d\langle i \rangle}^{(k)})$$

“Atención”: en diferentes pasos, dejar que el modelo se “centre” en diferentes partes de la entrada.

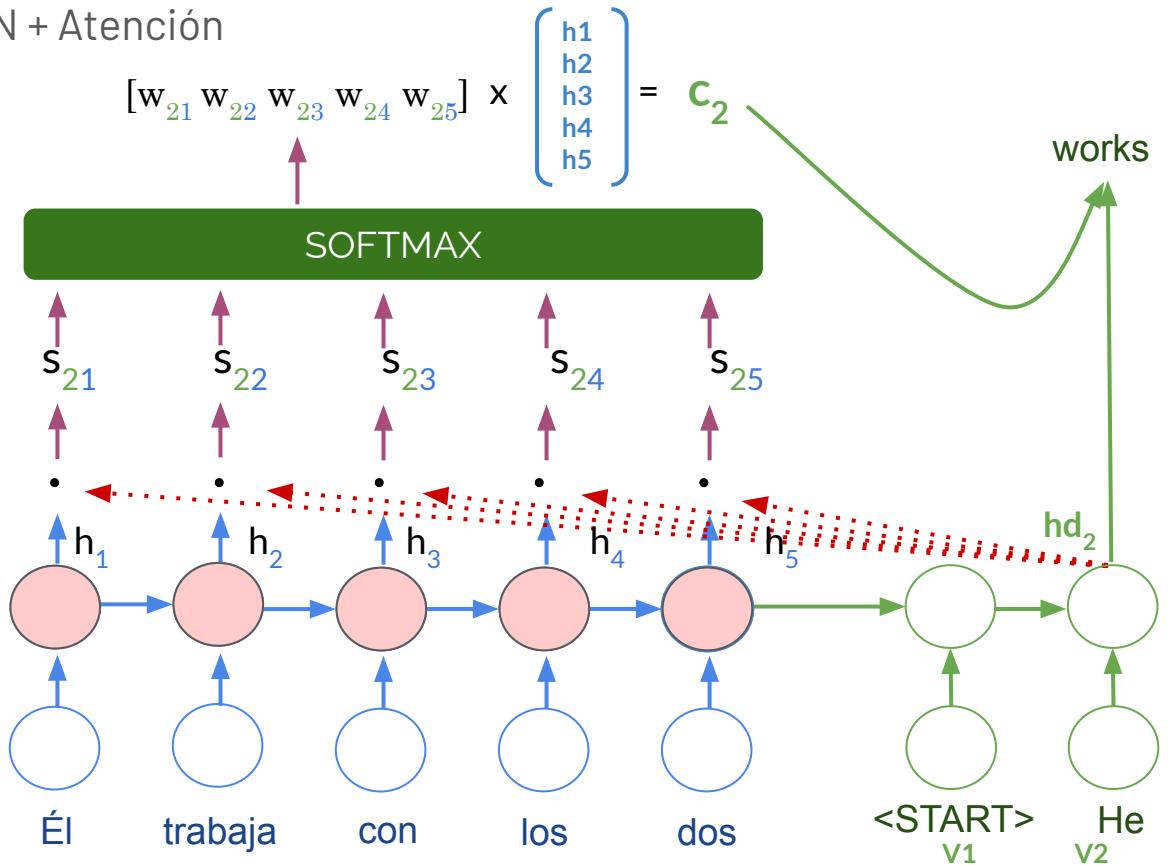
Encoder - Decoder

RNN + Atención



Encoder - Decoder

RNN + Atención



$$s_{ij} = \text{similitud}(\mathbf{h}_{d\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{d\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

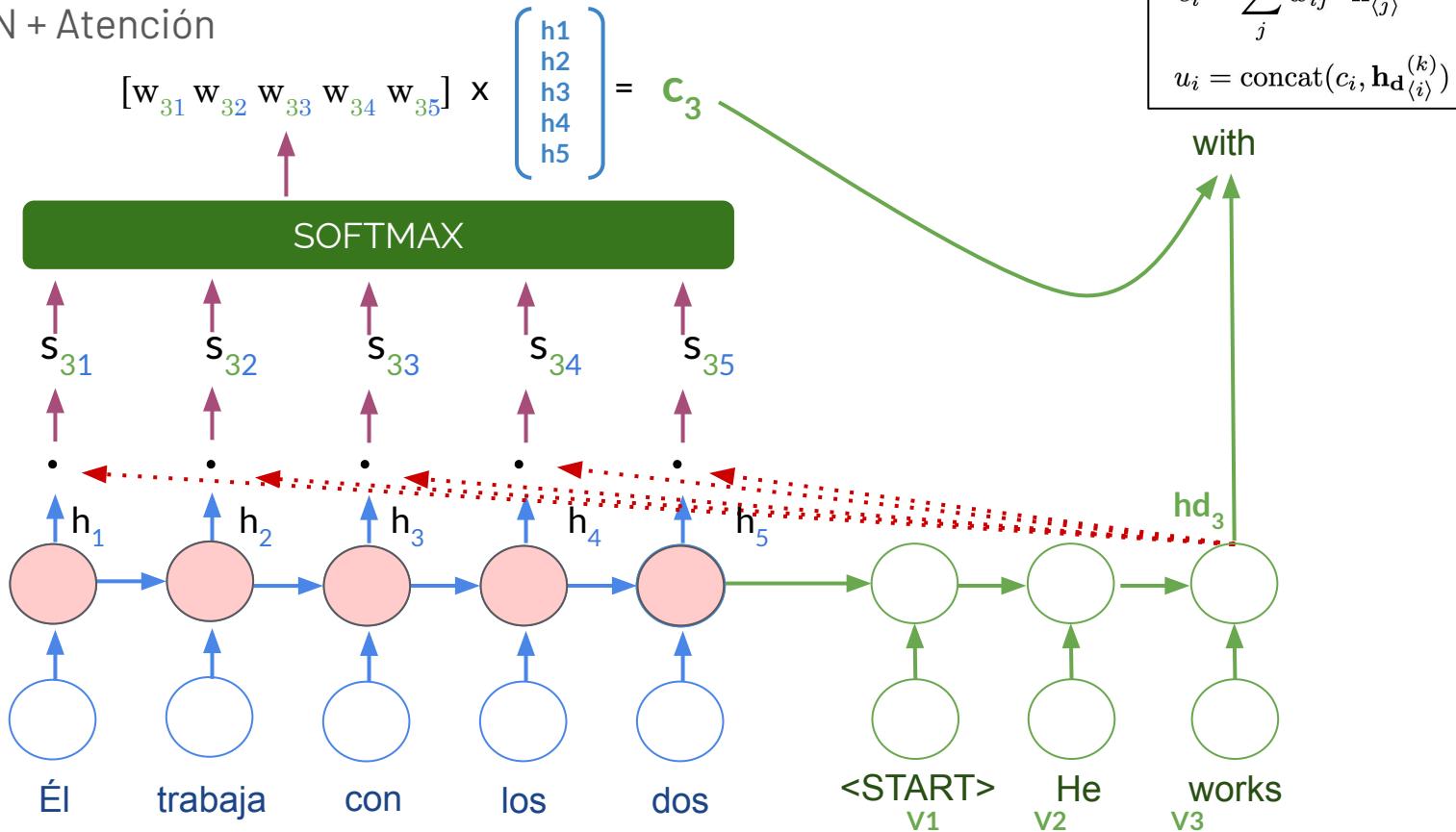
$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}$$

$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

$$u_i = \text{concat}(c_i, \mathbf{h}_{d\langle i \rangle}^{(k)})$$

Encoder - Decoder

RNN + Atención



$$s_{ij} = \text{similitud}(\mathbf{h}_{d\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{d\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

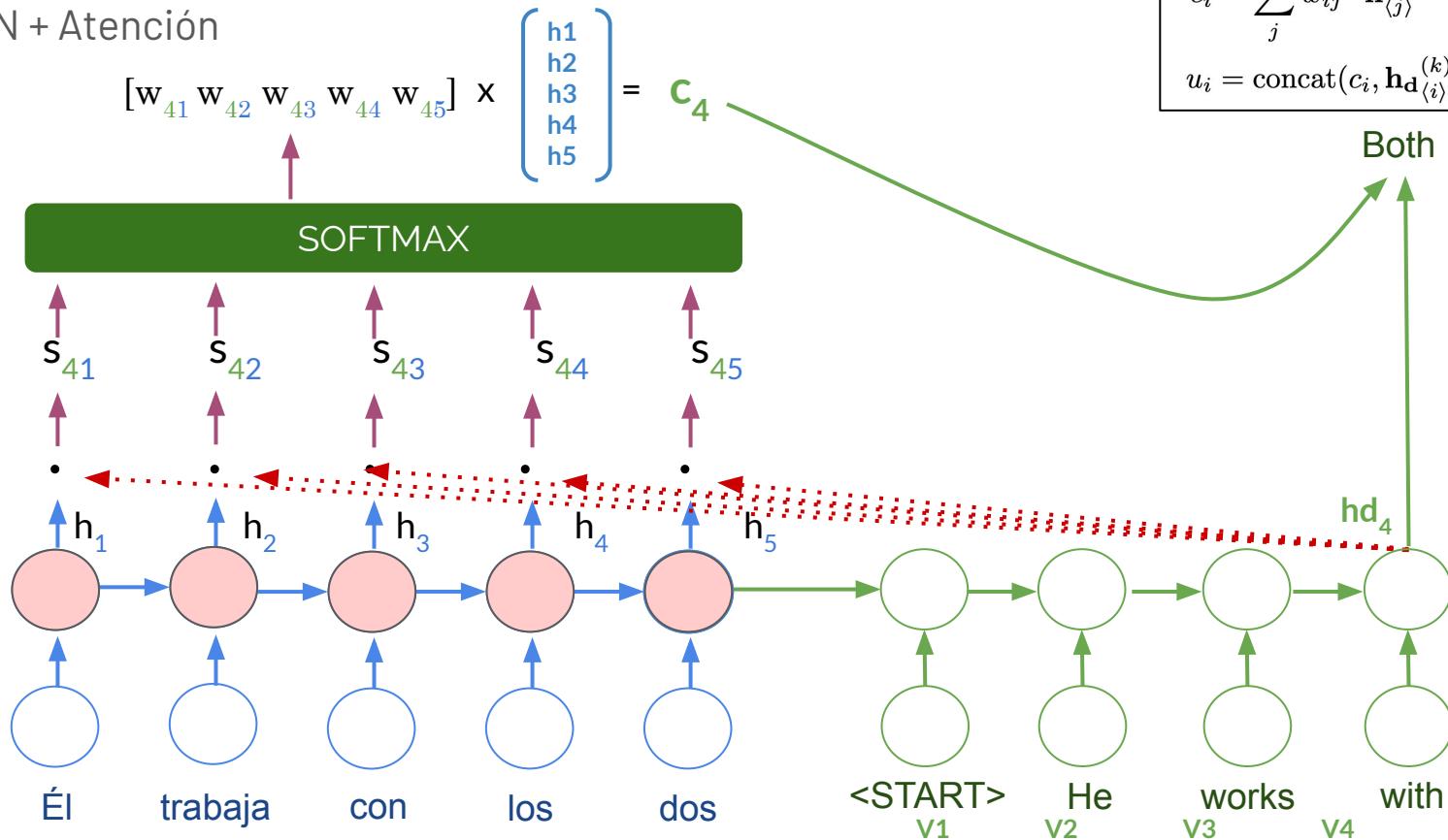
$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}$$

$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

$$u_i = \text{concat}(c_i, \mathbf{h}_{d\langle i \rangle}^{(k)})$$

Encoder - Decoder

RNN + Atención



$$s_{ij} = \text{similitud}(\mathbf{h}_{d\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{d\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

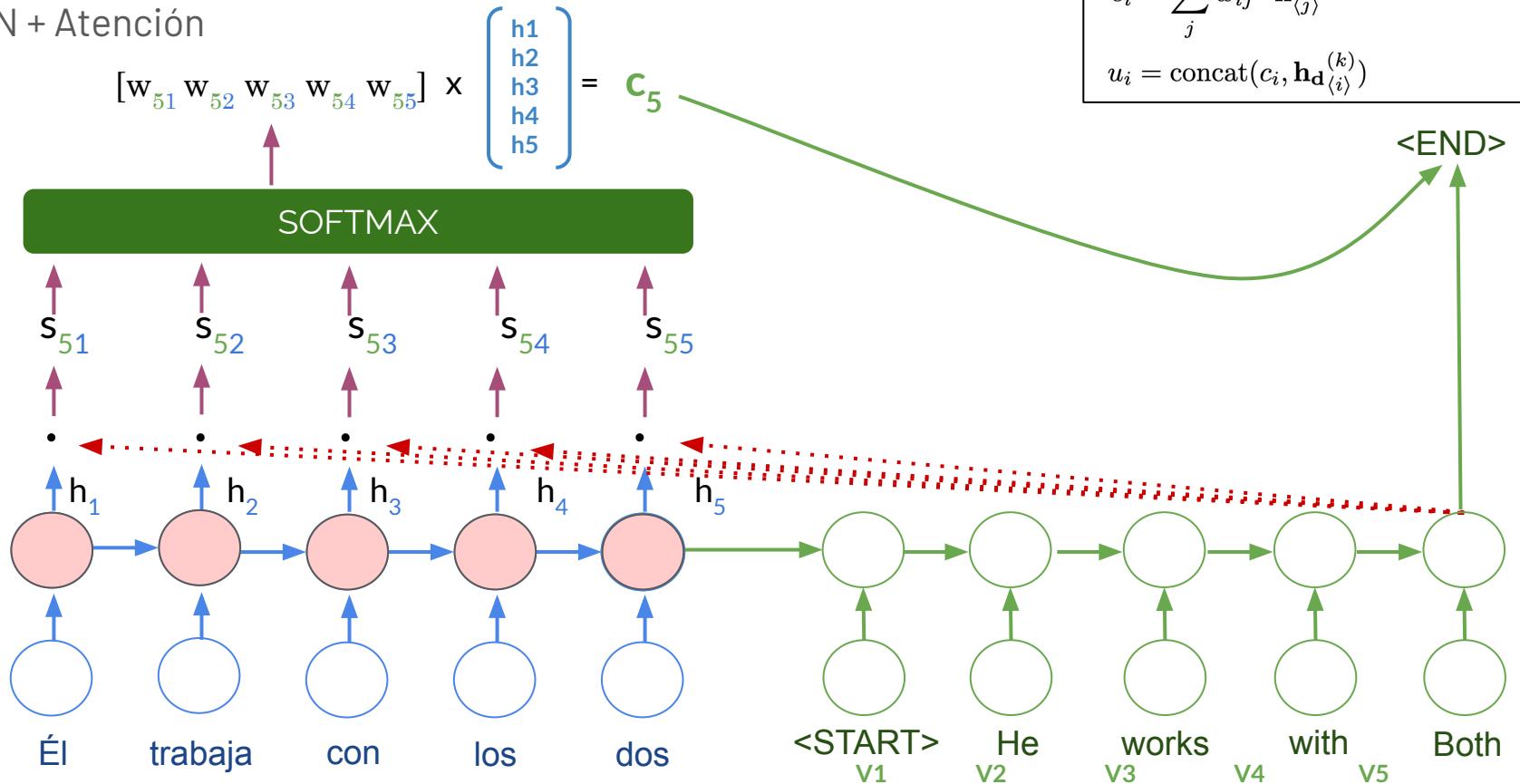
$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}$$

$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

$$u_i = \text{concat}(c_i, \mathbf{h}_{d\langle i \rangle}^{(k)})$$

Encoder - Decoder

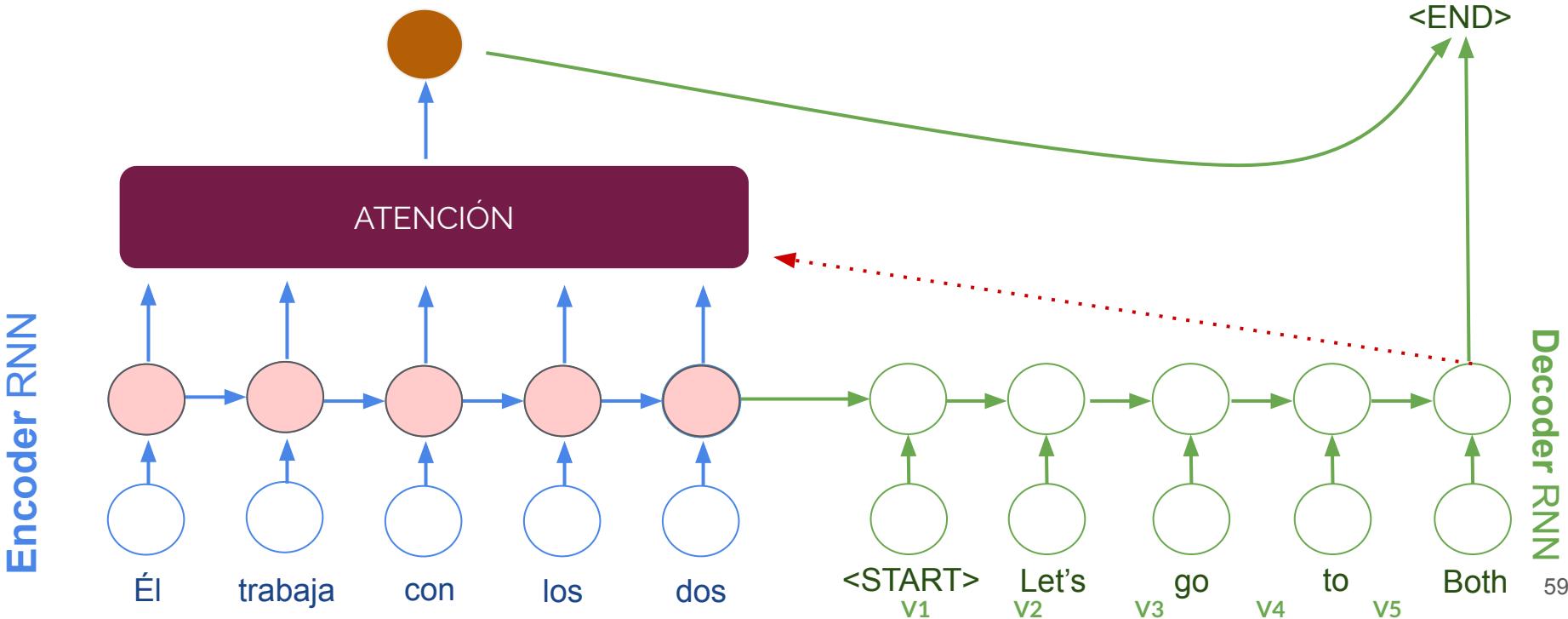
RNN + Atención



Encoder - Decoder

RNN + Atención

$$s_{ij} = \text{similitud}(\mathbf{h}_{\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h}_{\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$
$$w_{ij} = \text{softmax}(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}$$
$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$
$$u_i = \text{concat}(c_i, \mathbf{h}_{\langle i \rangle}^{(k)})$$

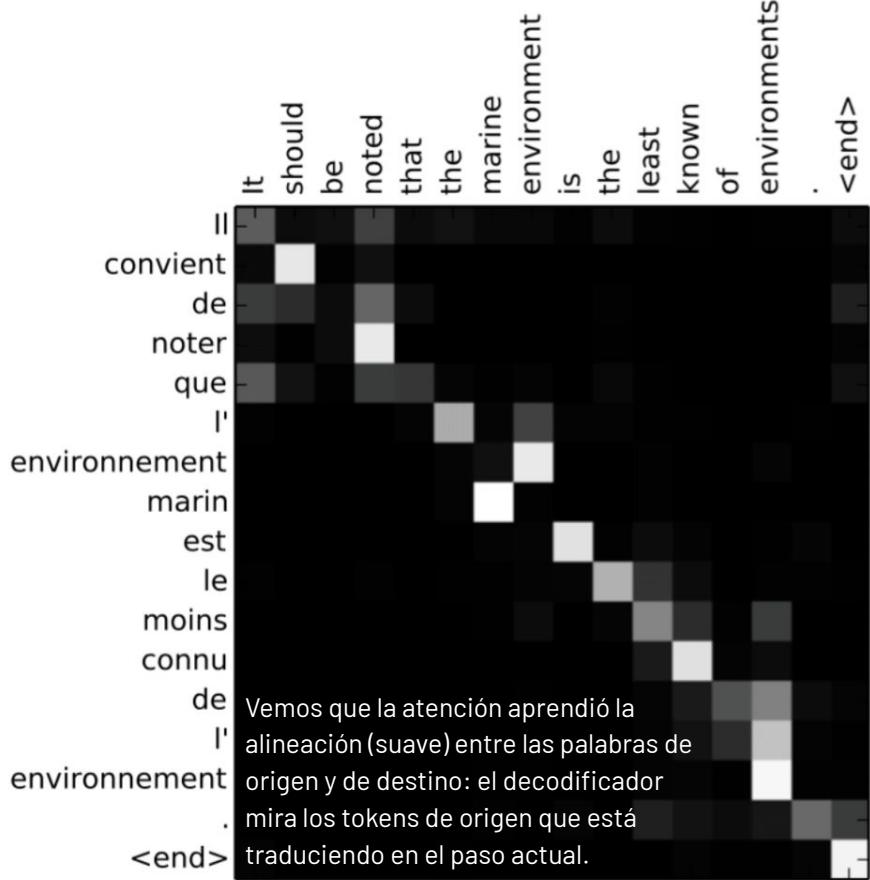


Encoder - Decoder

Resumen

En cada paso del decodificador:

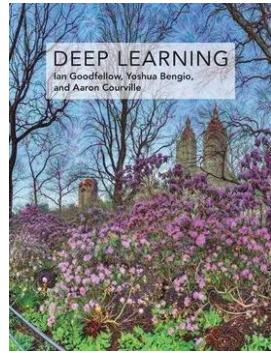
- Recibe entrada de atención: un estado de decodificador hd_i , y todos los estados del codificador $h_1, h_2, h_3 \dots h_T$
- Calcula puntuaciones de atención:
 - Para cada estado del codificador h_j , la atención calcula su "relevancia" para el estado hd_i del decodificador: **similitud(hd_i, h_j)**Formalmente, aplica una función de atención que recibe un estado de decodificador y un estado de codificador y devuelve un valor escalar.
- Calcula pesos de alineación: una distribución de probabilidad (softmax aplicado a puntuaciones);
- Calcula el vector de contexto: la suma ponderada de los estados del codificador y los pesos.



Aprende "Alineamientos".

[Neural Machine Translation by Jointly Learning to Align and Translate]

```
1. class EncoderRNN(nn.Module): # Igual que antes.
2.
3. class Attention(nn.Module):
4.     def __init__(self, hidden_size):
5.         self.attn = nn.Linear(hidden_size * 2, hidden_size)
6.         self.v = nn.Parameter(hidden_size)
7.
8.     def forward(self, hidden, encoder_outputs):
9.         energy = torch.tanh(self.attn(torch.cat([hidden, encoder_outputs], 2)))
10.        energy = self.v @ energy
11.        attention_weights = torch.softmax(energy)
12.        return attention_weights
13.
14. class DecoderRNN(nn.Module):
15.     def __init__(self, hidden_size, output_size, num_layers=1):
16.         self.rnn = nn.RNN(hidden_size, hidden_size, num_layers, batch_first=True)
17.         self.attention = Attention(hidden_size)
18.         self.fc = nn.Linear(hidden_size * 2, output_size)
19.
20.     def forward(self, x, hidden, encoder_outputs):
21.         attn_weights = self.attention(hidden[-1], encoder_outputs)
22.         context = attn_weights @ encoder_outputs
23.         rnn_input = torch.cat([x, context], 2)
24.         output, hidden = self.rnn(rnn_input, hidden)
25.         output = self.fc(torch.cat([output, context]))
26.         return output, hidden
```



Tarea

Completar el formulario

Lecturas obligatorias:

- Deep Learning (Goodfellow, Bengio, Courville):
 - **Capítulo 10 “Sequence Modeling”** (hasta sección **10.7 inclusive**, pueden saltar 10.2.3 y 10.2.4 y 10.6).

Opcional.

- Paper: **“Neural Machine Translation by Jointly Learning to Align and Translate”**

Opcional:

- Deep Learning: Resto del **Capítulo 10 (que incluye temas como LSTM, clipping gradients, memoria explícita, etc)**.
- Blogs recomendados:
 - https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html
 - <https://distill.pub/2016/auqmented-rnns/>