DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Aprendizaje Automático

Clase 9:

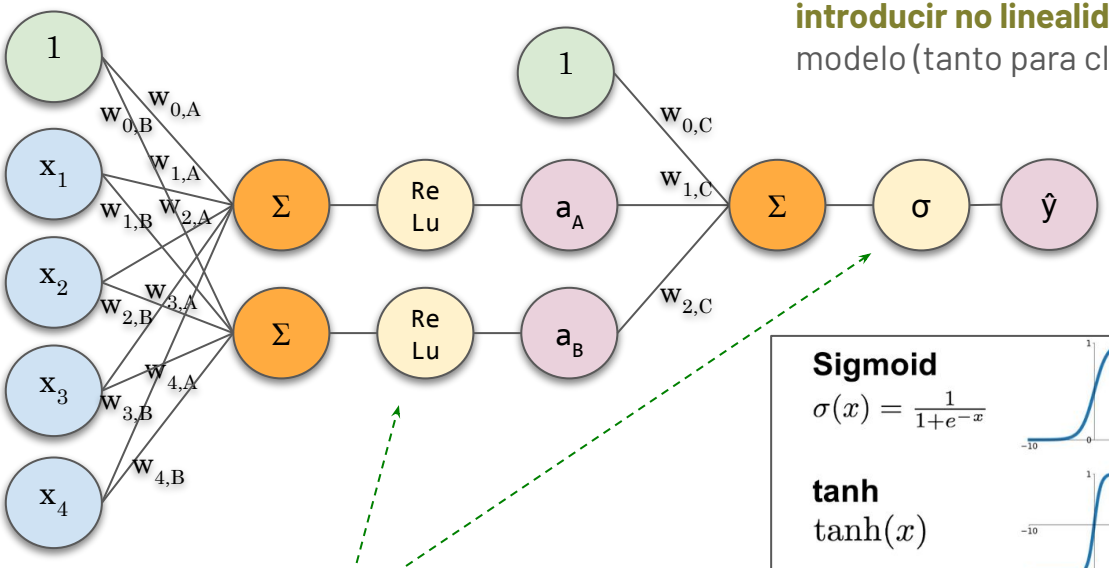
Redes neuronales. Parte II
Predicción de Secuencias

<Repaso>

Redes neuronales multicapa

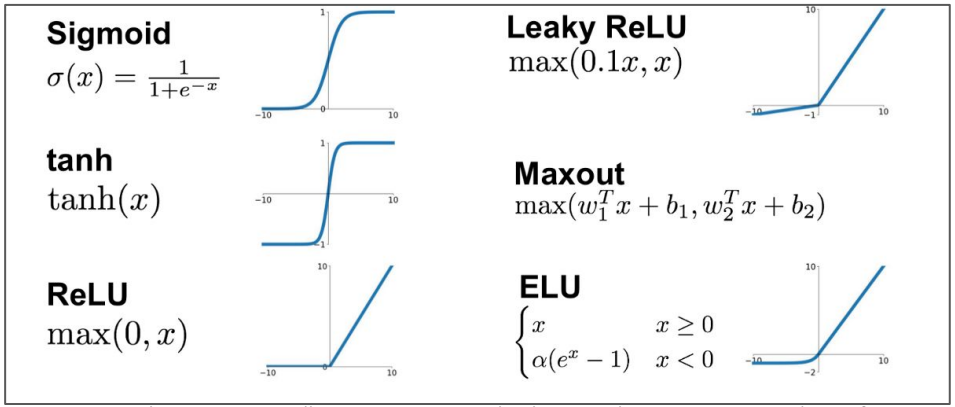
Función de activación

Además de tener una función no lineal como última operación en el caso de clasificación, es común (necesario) **introducir no linealidades** entre las distintas capas del modelo (tanto para clasificación como para regresión).



$\mathbb{R}^{p+1}$ En general, se utiliza la misma función de activación en todas las capas intermedias (en esta caso sólo una capa intermedia). Pero la de la última capa depende del tipo de problema.

Funciones de activación



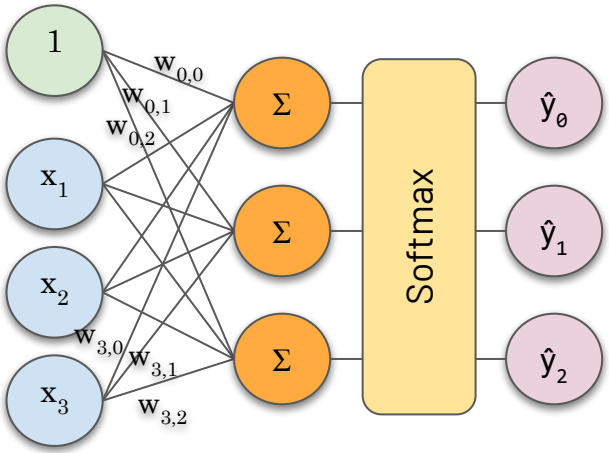
Fuente https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092

Redes neuronales multi-output

Softmax

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

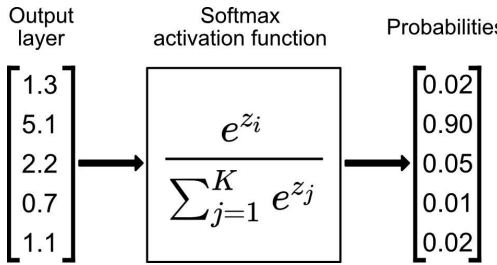
Podríamos usar la siguiente arquitectura:



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**.



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)

$$J_{MLE} = \frac{1}{n}\sum_{i=1}^{n} \text{Multiclass\_CE}^{(i)} = \frac{1}{n}\sum_{i=1}^{n}\left[-\sum_{k=1}^{K} y_k^{(i)} \log(\hat{h}_w^{(k)}(\mathbf{x}^{(i)}))\right]$$

$$J = J_{\text{MLE}} + \lambda\left(\sum_{i,j}\left(W_{i,j}^{(1)}\right)^2 + \sum_{i,j}\left(W_{i,j}^{(2)}\right)^2\right)$$

- 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**.

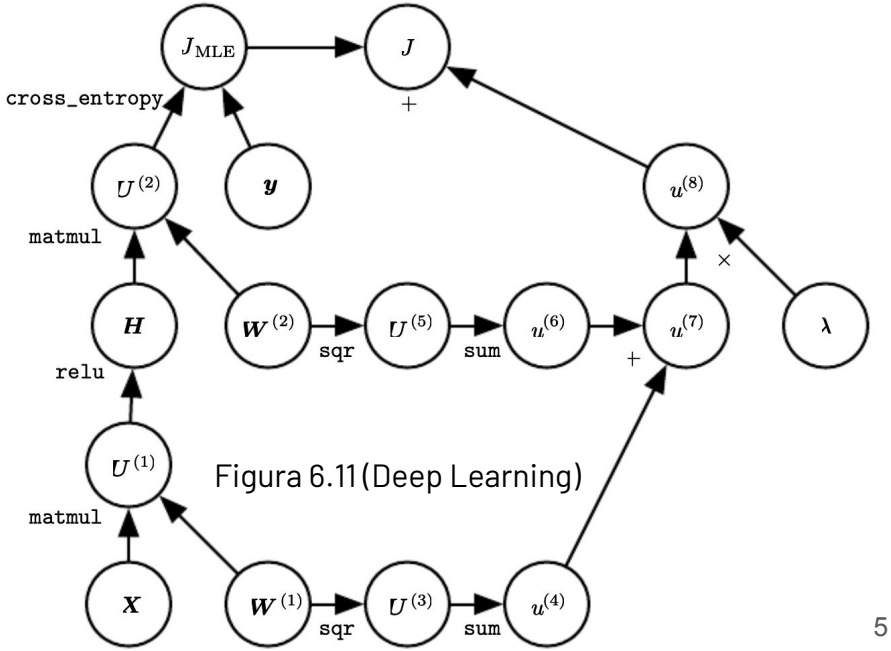


Figura 6.11 (Deep Learning)

</Repaso>

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. $\mathbf{x}^{(i)} \in \mathbb{R}^{10000}$
  - “hola hola cómo estás” → <0, 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 $h : X \rightarrow Y$:
  - $X \in \mathbb{R}^{10000}$
  - $Y \in \{\text{positivo}, \text{negativo}\}$

- 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.

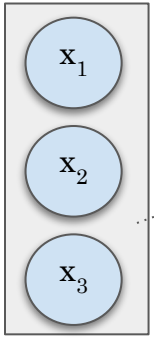
Paréntesis: Una visualización vertical de redes

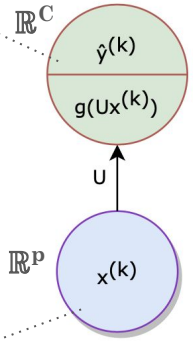
Otra forma de visualizar las redes que conocemos

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

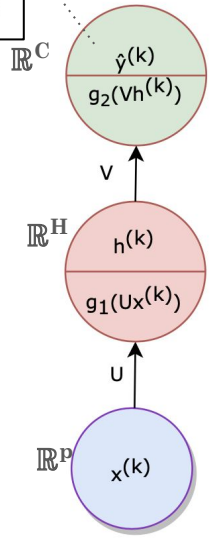
$$\hat{y}^{(k)} = g(Ux^{(k)})$$

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

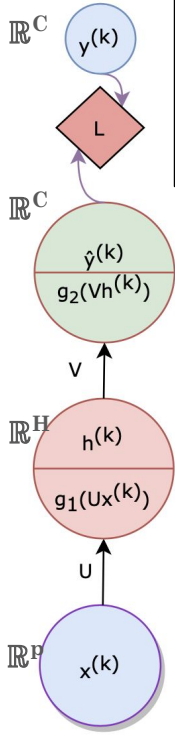




Red neuronal simple



Red neuronal con una capa oculta



Podemos hacer explícita la función de pérdida y la etiqueta real

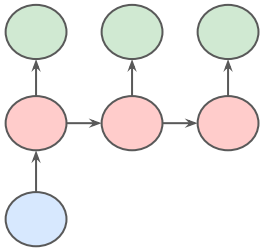
Red neuronal con una capa oculta y su función de pérdida

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.

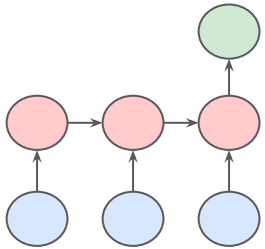
Tipos de modelos:

Uno a muchos



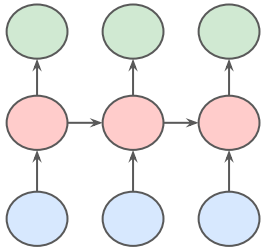
Ej: Descripción imagen
$x^{(k)}$ = imagen
$y^{(k)}$ = [“veo”, “un”, “perro”]

Muchos a uno



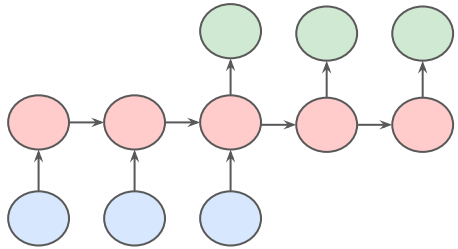
Ej: Predecir el sentimiento
$x^{(k)}$ = [“sí”, “me”, “encantó”]
$y^{(k)}$ = positivo

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

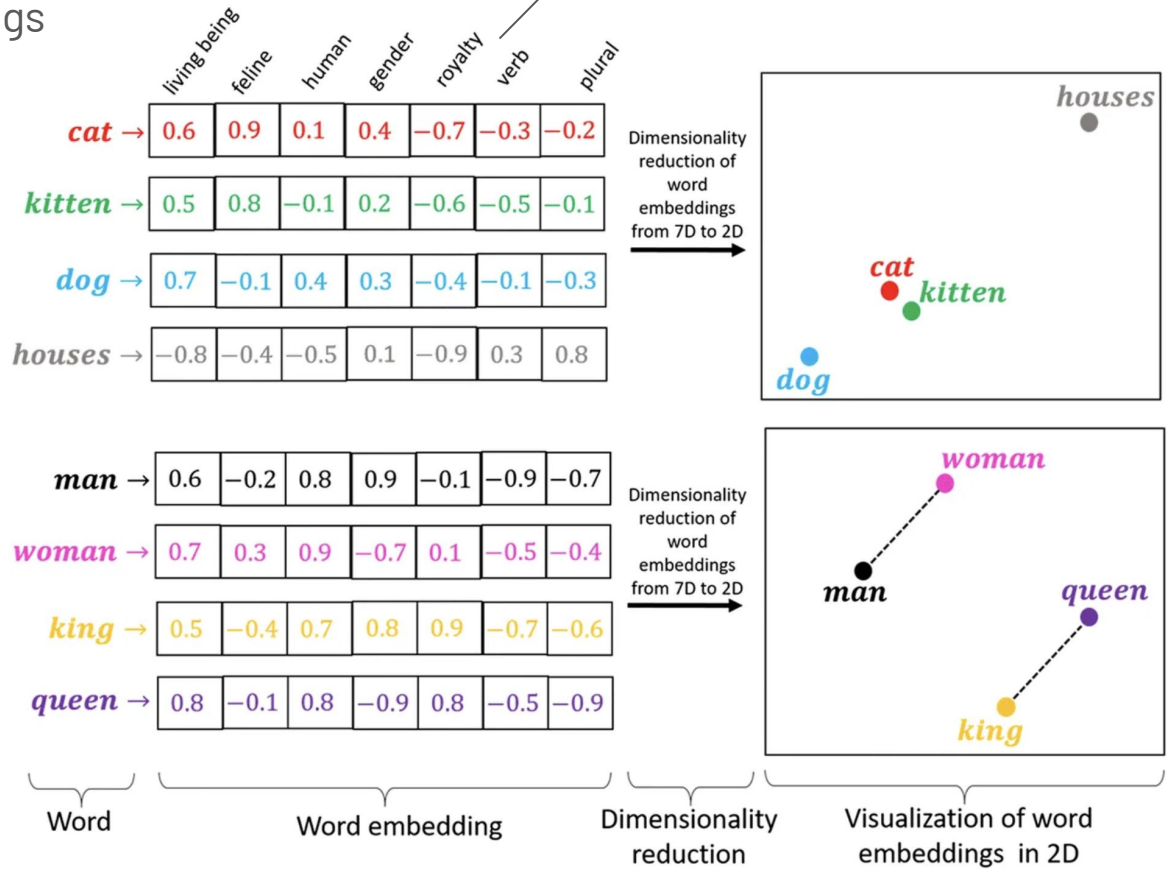
Nota: "Word Embeddings". Transformaciones de palabras a vectores que las representan "bien semánticamente".

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



Definición del problema: Predicción de secuencias.

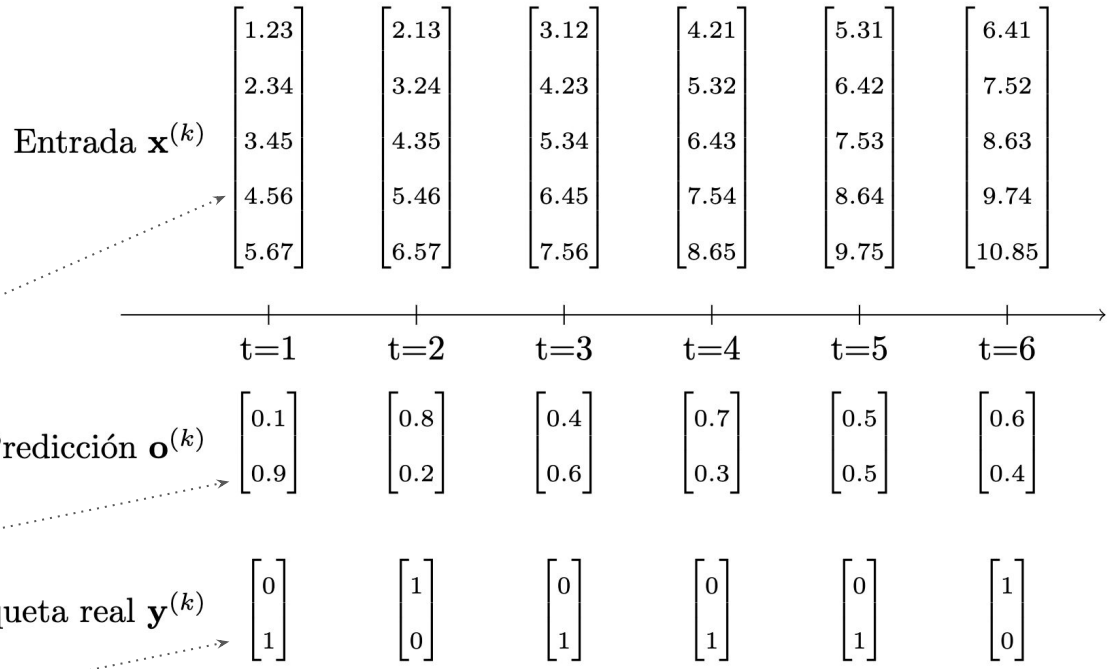
Redefinición de "instancias"

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

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

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

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



En este ejemplo $\mathrm{p} = 5,\ \mathrm{C} = 2,\ \mathrm{T}=6$

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 $\mathbf{J}_{\mathbf{X},\mathbf{y}}$ (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).

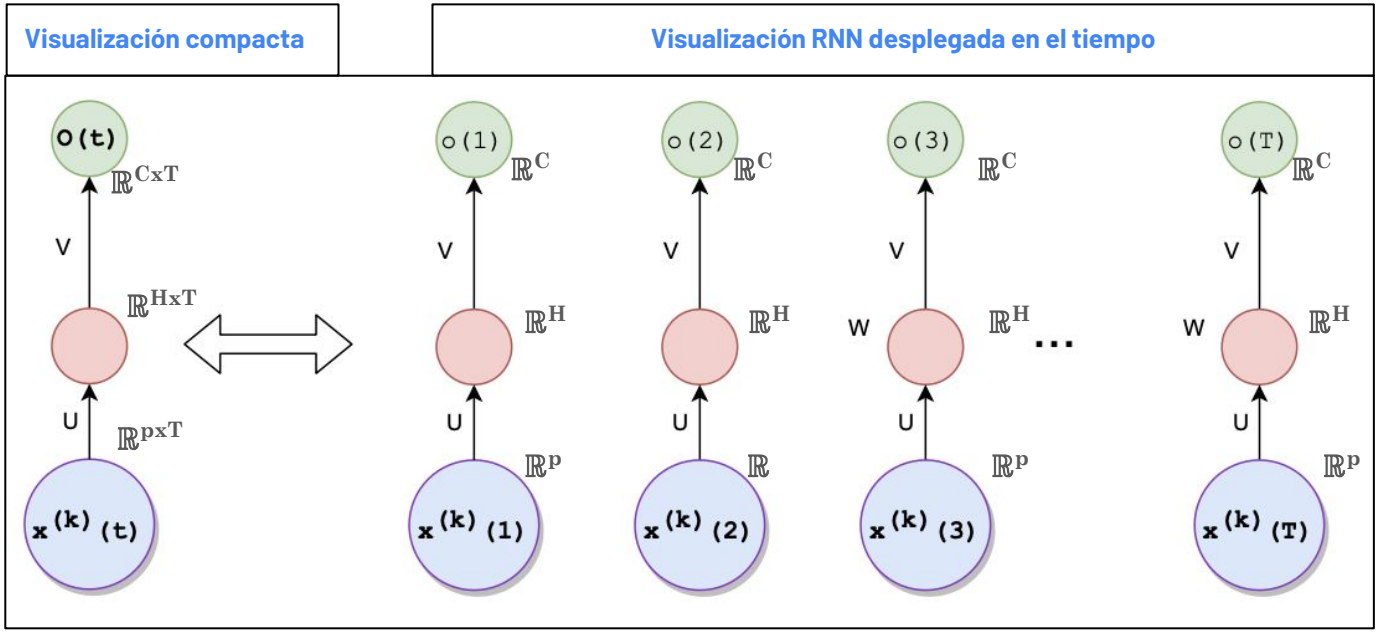
(Ya veremos cómo esto afecta al entrenamiento.)

$$L(\mathbf{x}^{(k)}, \mathbf{y}^{(k)}) = L(\{\mathbf{x}^{(k)}_{\langle 1 \rangle} \dots \mathbf{x}^{(k)}_{\langle T \rangle}\}, \{\mathbf{y}^{(k)}_{\langle 1 \rangle} \dots \mathbf{y}^{(k)}_{\langle T \rangle}\})$$

$$= \sum_{t=1}^{T} L(\mathbf{x}^{(k)}_{\langle t \rangle}, \mathbf{y}^{(k)}_{\langle t \rangle})$$

$$\mathrm{J}_{X,y} = \frac{1}{n} \sum_{i=1}^{n} \mathrm{L}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$$

Redes Neuronales Recurrentes



Si entrenamos esta red **no podría aprender** qué hacer según información anterior.

Necesitamos agregar dependencias temporales.

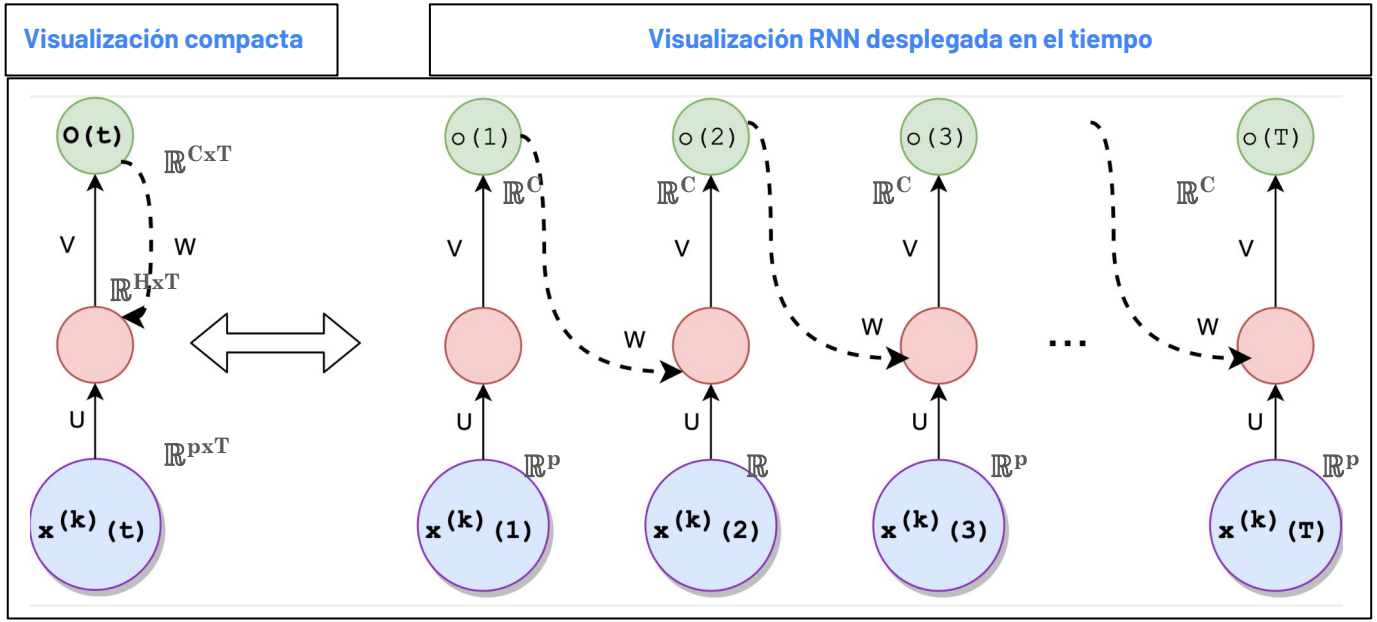
Fórmula que describe esta red:
(omitimos los biases (w_0) y suponemos expansión con 1's como veníamos haciendo)

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

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).

Redes Neuronales Recurrentes



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

¿Qué desventaja tiene esta arquitectura?

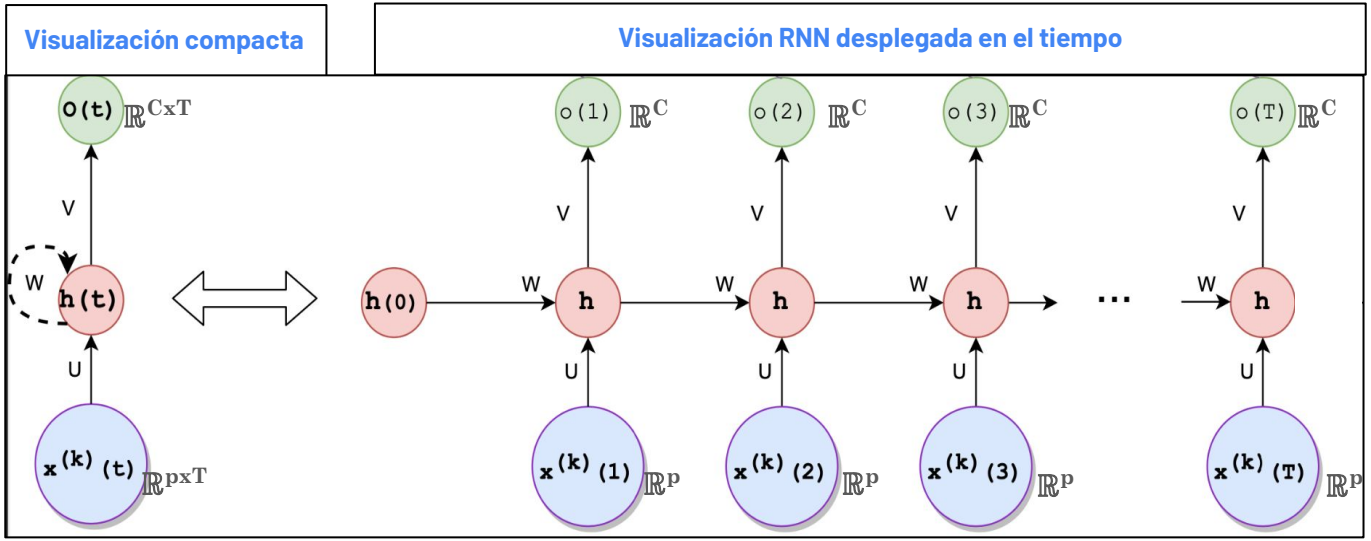
Fórmula que describe esta red:
(omitimos los biases (w_0) y suponemos expansión con 1's como veníamos haciendo)

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

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).

Redes Neuronales Recurrentes



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

El estado puede mantener información a largo plazo

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

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

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).

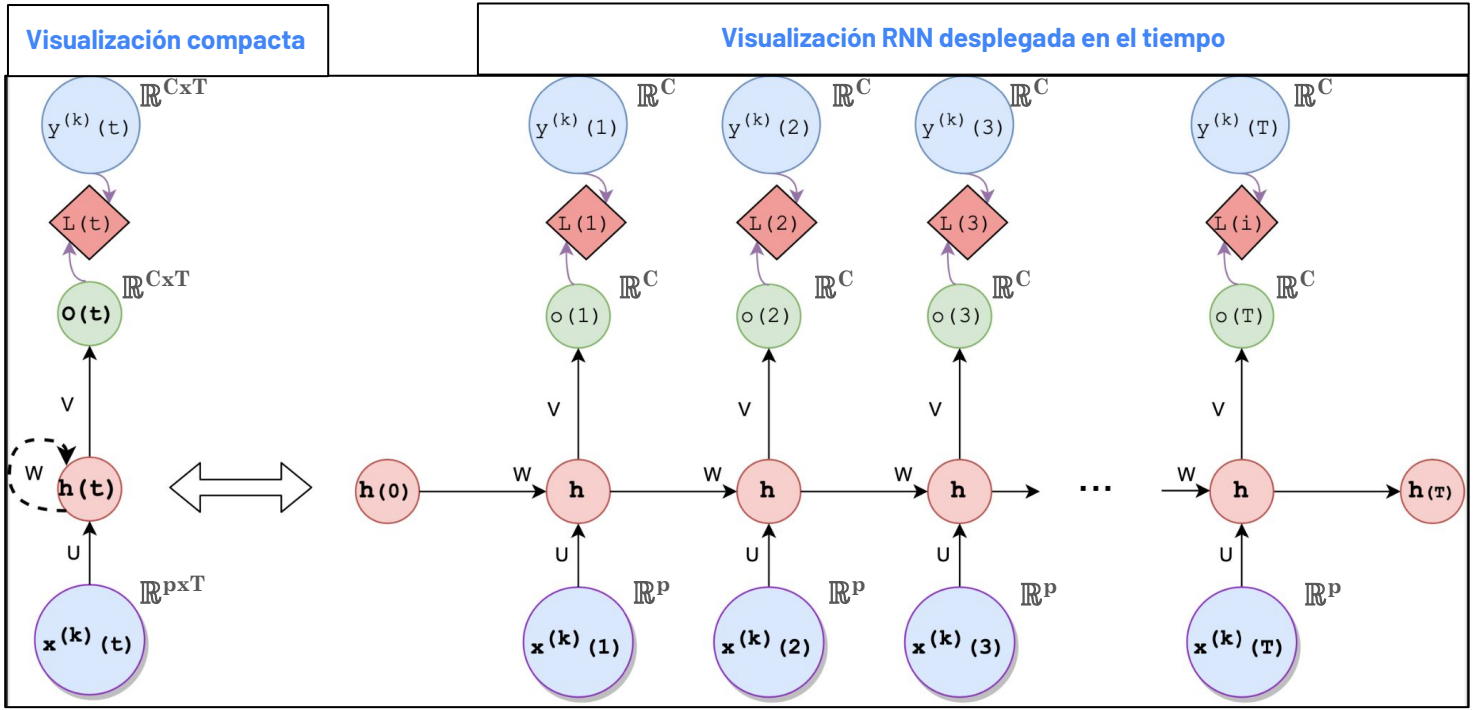
$$L(\mathbf{x}^{(k)}, \mathbf{y}^{(k)}) = L(\{\mathbf{x}^{(k)}_{\langle 1 \rangle} \dots \mathbf{x}^{(k)}_{\langle T \rangle}\}, \{\mathbf{y}^{(k)}_{\langle 1 \rangle} \dots \mathbf{y}^{(k)}_{\langle T \rangle}\})$$

$$= \sum_{t=1}^{T} L(\mathbf{x}^{(k)}_{\langle t \rangle}, \mathbf{y}^{(k)}_{\langle t \rangle})$$

$$\mathrm{J}_{X,y} = \frac{1}{n} \sum_{i=1}^{n} \mathrm{L}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$$

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 Backpropagation / Descenso por el Gradiente?



Redes Neuronales Recurrentes

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

Aunque hay variantes, como TBPTT (**Truncated** BPPT).

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

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. (*)

Problema 1: Cómputo no paralelizable.

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? (igual no es buena idea!)

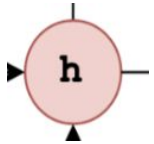
```
1.  class NuestraRNN(nn.Module):  # Código de ejemplo en (quasi-)pytorch
2.      def __init__(self, dim_input, dim_hidden, dim_output):
3.          self.U = nn.Parameter(torch.Tensor(dim_hidden, dim_input))  # mapea Input -> Oculto
4.          self.W = nn.Parameter(torch.Tensor(dim_hidden, dim_hidden)) # mapea Oculto -> Oculto
5.          self.fc = nn.Linear(dim_hidden, dim_output)
6.          self.init_weights()

7.      def forward(self, x):
8.          batch_size, seq_len, input_dim = x.size()
9.          hidden = self.init_hidden() # Inicializamos hidden en h0.
10.         for t in range(seq_len):
11.             x_t = x[:, t, :]
12.             hidden = torch.tanh(x_t @ self.U.T + hidden @ self.W.T) # TanH sería nuestra g1
13.         out = self.fc(hidden)
14.         return out, hidden
```

```
15. model = NuestraRNN(input_size, hidden_size, output_size)
16. criterion = nn.MSELoss()
17. optimizer = optim.SGD(model.parameters(), lr=learning_rate) # SGD = Gradient Descent
18.
19. for epoch in range(num_epochs):
20.     for batch_X, batch_y in dataloader:
21.         outputs, hidden_final = model(batch_X) # Notar que no usamos hidden_final (podríamos)
22.         loss = criterion(outputs, batch_y)
23.         loss.backward()
```

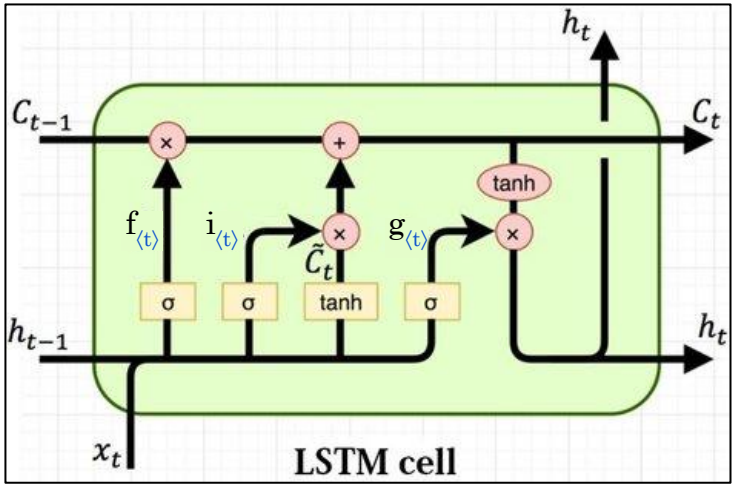
“Memoria” a largo plazo

LSTM

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



cambiamos esta parte de la fórmula anterior por algo más complejo.



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

Ahora los estados ocultos tienen un “Cell State” y un “Hidden State”.

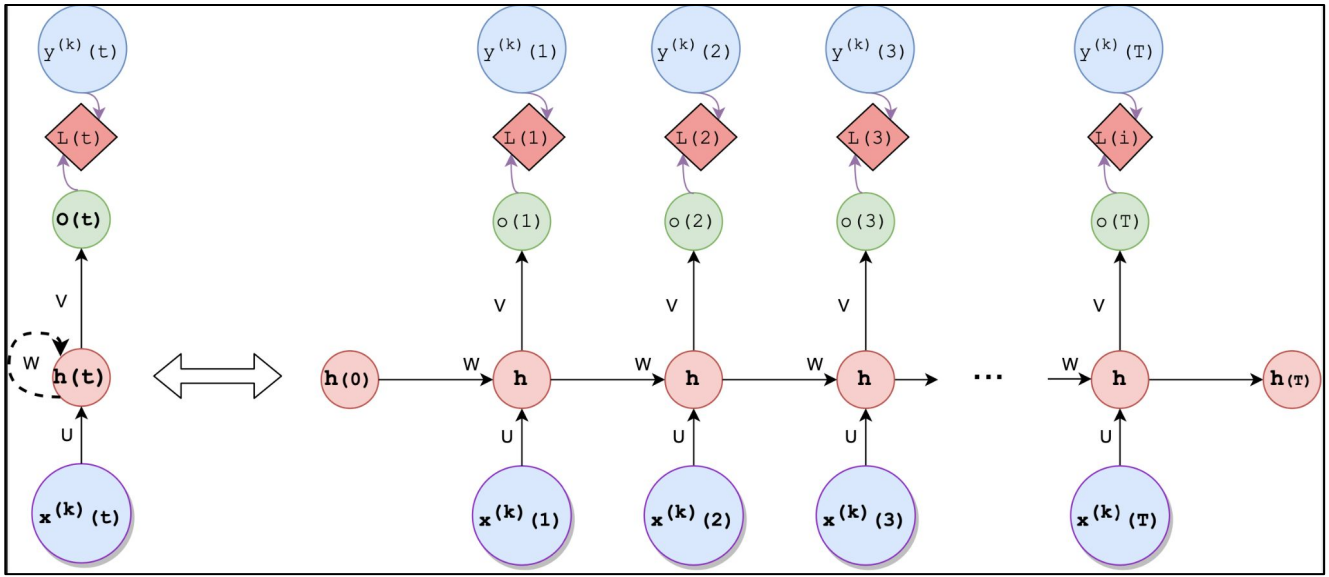
Dos caminos para recordar

El Cell state, para recordar a largo plazo.

El Hidden para corto plazo.

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

También buscar sobre células de tipo GRU

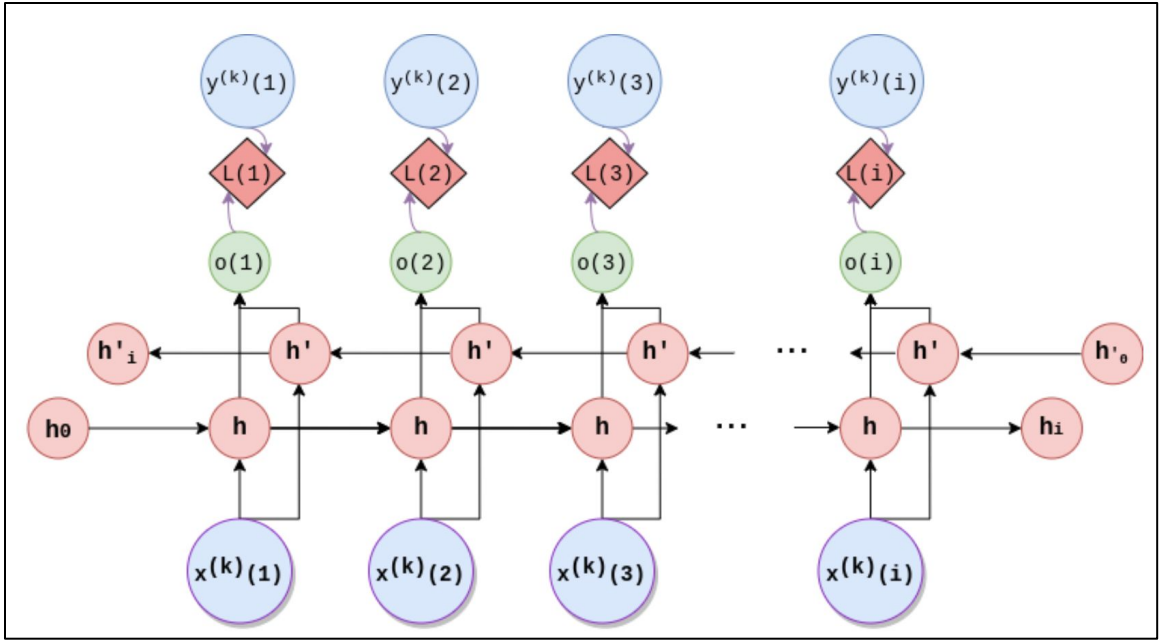


¿Sólo el pasado?

- Las RNN acumulan información a lo largo del tiempo. Las predicciones para cada instante dependen del estado acumulado en función de las entradas anteriores.

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

- Sin embargo, hay problemas en los cuales es necesario conocer información de la entrada en instantes posteriores al momento de la predicción.



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.`

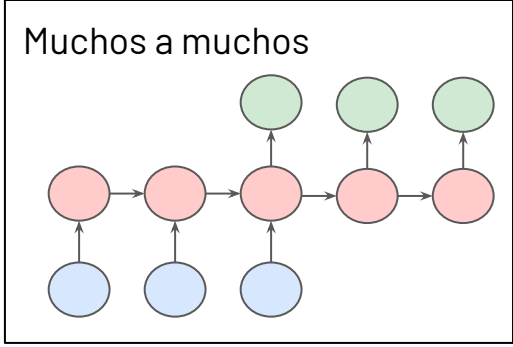
$$\overrightarrow{\mathbf{h}}^{(k)}_{\langle t \rangle} = g_1(W \overrightarrow{\mathbf{h}}^{(k)}_{\langle t-1 \rangle} + U\mathbf{x}^{(k)}_{\langle t \rangle})$$

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

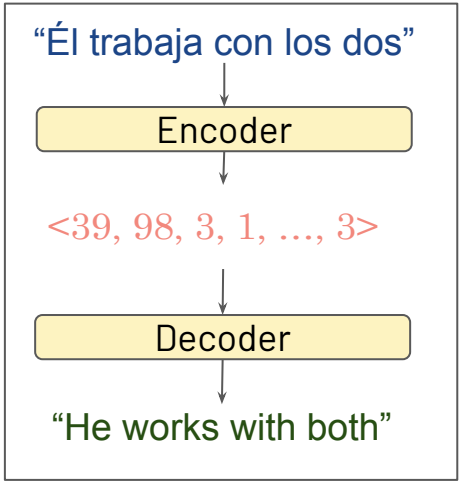
$$\mathbf{o}^{(k)}_{\langle t \rangle} = g_2(V[\overleftarrow{\mathbf{h}}^{(k)}_{\langle t \rangle}, \overrightarrow{\mathbf{h}}^{(k)}_{\langle t \rangle}])$$

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.



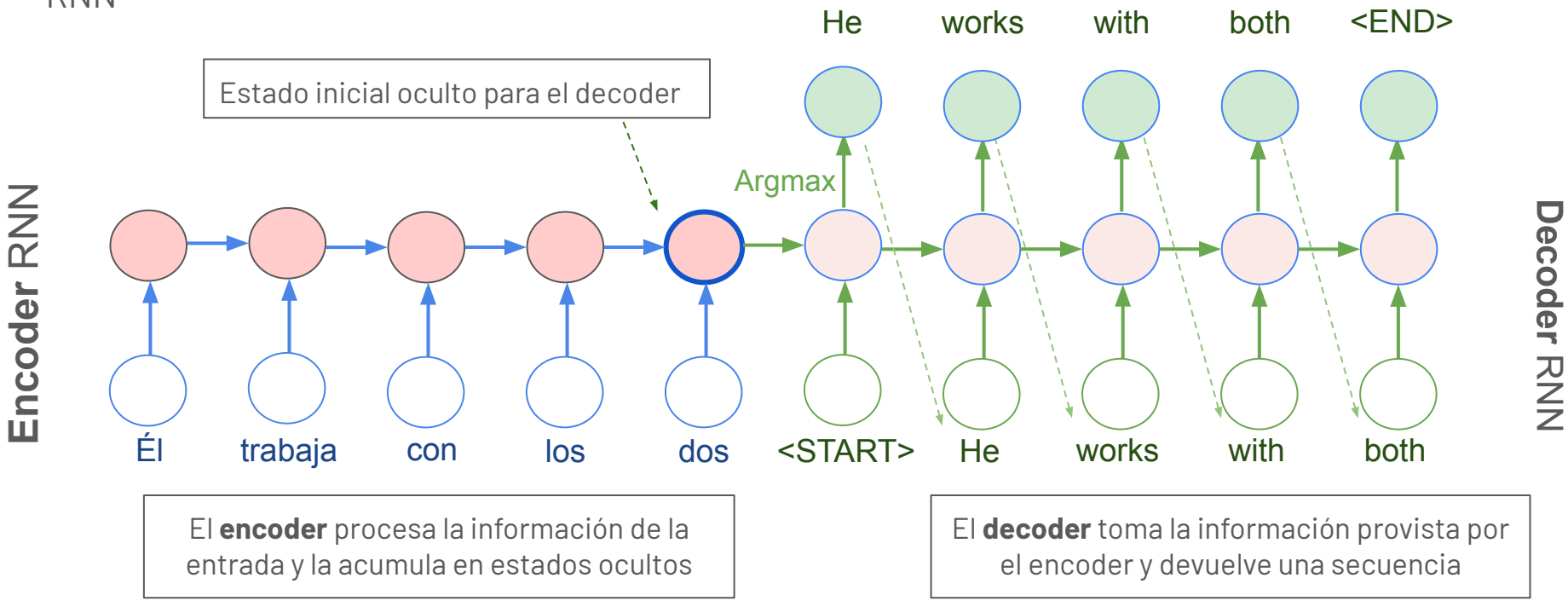
Incluye un **codificador (encoder)** y un **decodificador (decoder)**.

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



Para investigar: **Greedy Decoding** vs **Beam Search** (https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html)

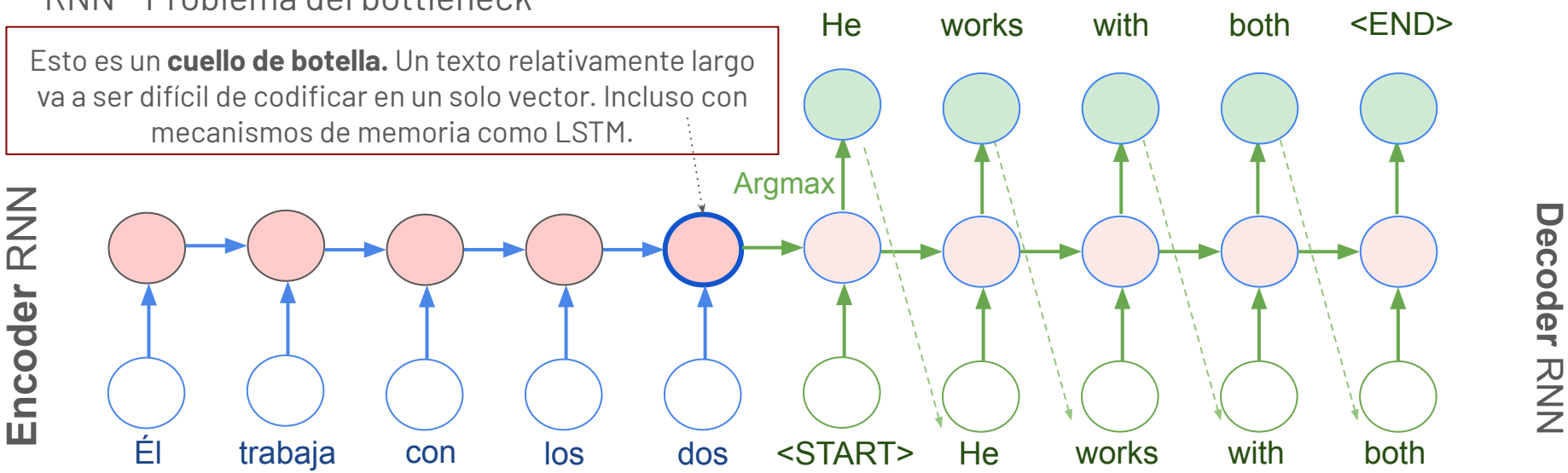
```
class EncoderRNN(nn.Module): # Código de ejemplo en (quasi-)pytorch
    def __init__(self, input_size, dim_hidden):
        ...
        self.rnn = nn.RNN(input_size, dim_hidden)

    def forward(self, x):
        h0 = self.init_hidden()
        output, hidden = self.rnn(x, h0)
        return output, hidden

class DecoderRNN(nn.Module):
    def __init__(self, dim_hidden, dim_output):
        ...
        self.rnn = nn.RNN(dim_output, dim_hidden)
        self.fc = nn.Linear(dim_hidden, dim_output)

    def forward(self, x, hidden):
        output, hidden = self.rnn(x, hidden)
        output = self.fc(output)
        return output, hidden


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

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


for epoch in range(num_epochs):
   for batch_data, batch_targets in dataloader:

        # Pasada por el Encoder
        encoder_outputs, encoder_hidden = encoder(batch_data)

        decoder_outputs = []
        decoder_input = torch.zeros(batch_size, 1, output_size)
        decoder_hidden = encoder_hidden

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

            # "Teacher Forcing": hasta cierto epoch, usamos los y reales y no las predicciones
            decoder_input = batch_targets[:, t]
        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.



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

Atención (ver.2014)

Encoder - Decoder

RNN + Atención

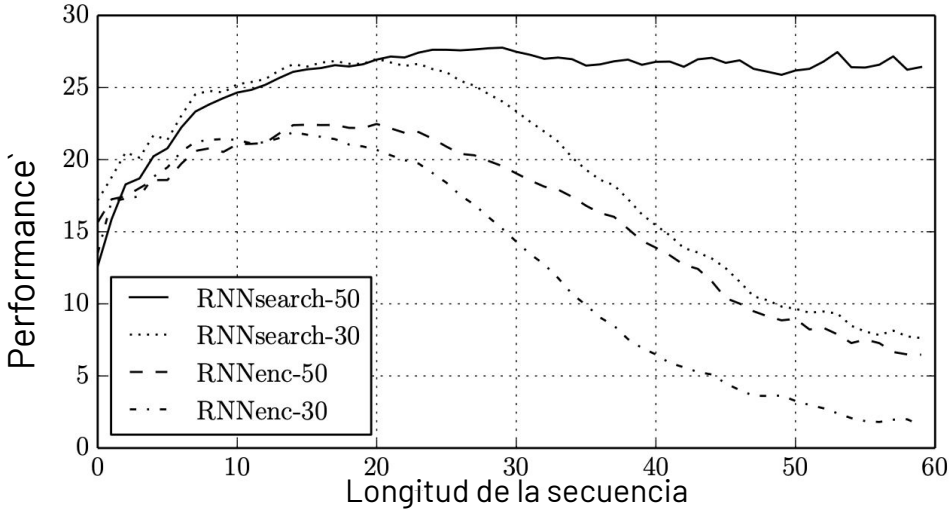
NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

(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 ...**"

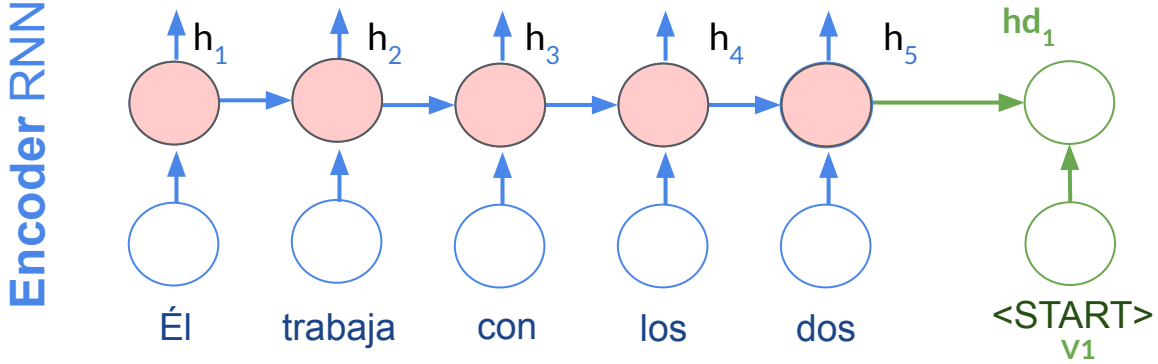


“Atención”: en diferentes pasos, dejar que el modelo se "centre" en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención

Esta vez vamos a usar las salidas de la RNN en cada paso.



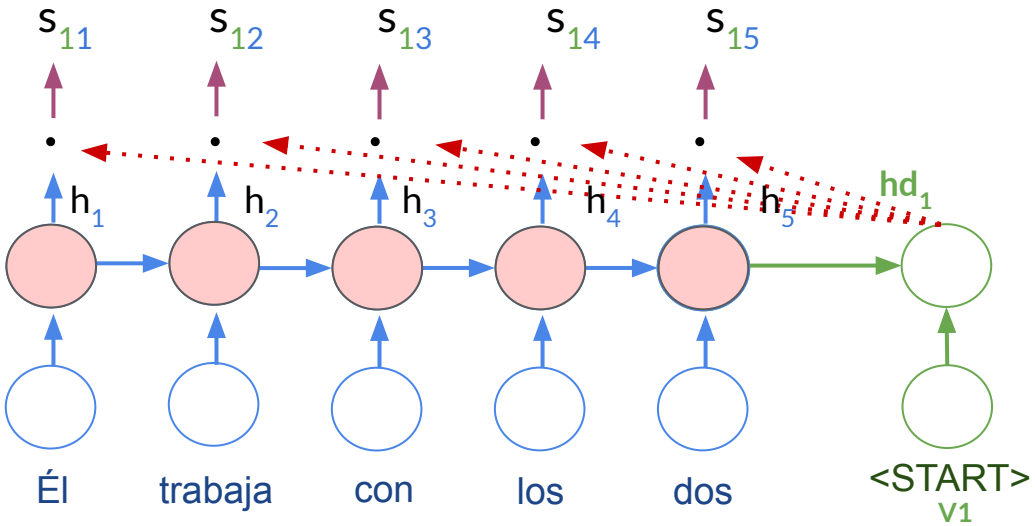
“Atención”: en diferentes pasos, dejar que el modelo se "centre" en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención

Esta vez vamos a usar las salidas de la RNN en cada paso.

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

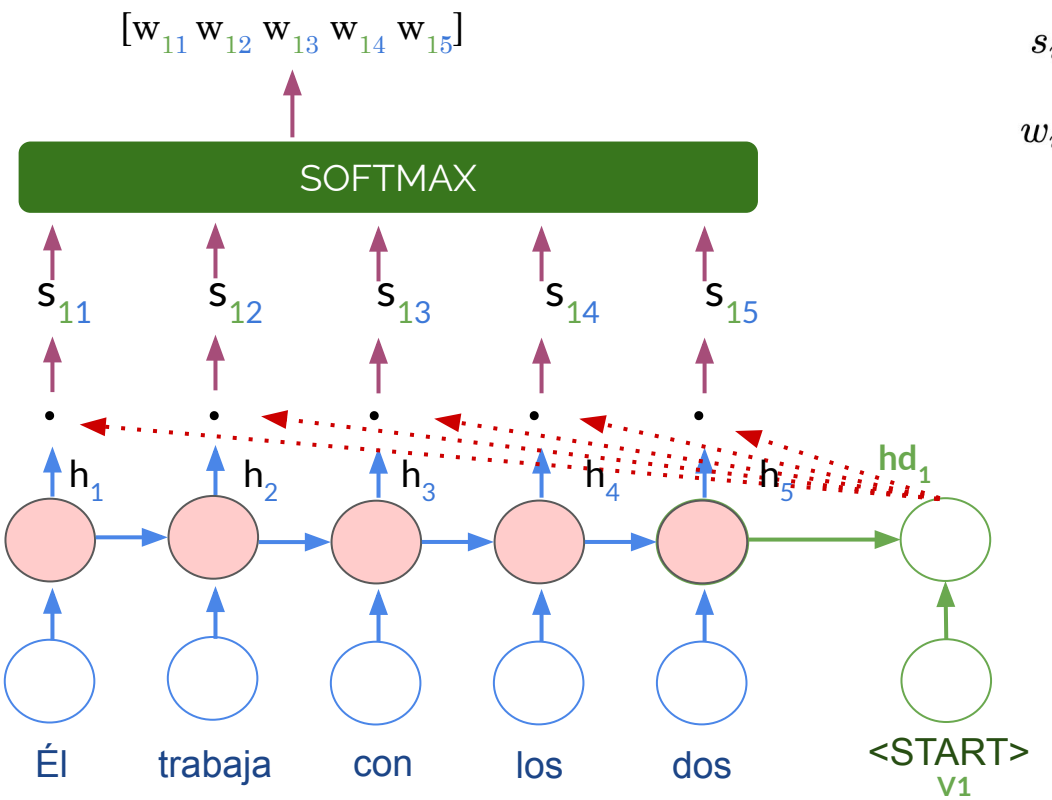


“Atención”: en diferentes pasos, dejar que el modelo se "centre" en diferentes partes de la entrada.

Encoder - Decoder

RNN + Atención

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



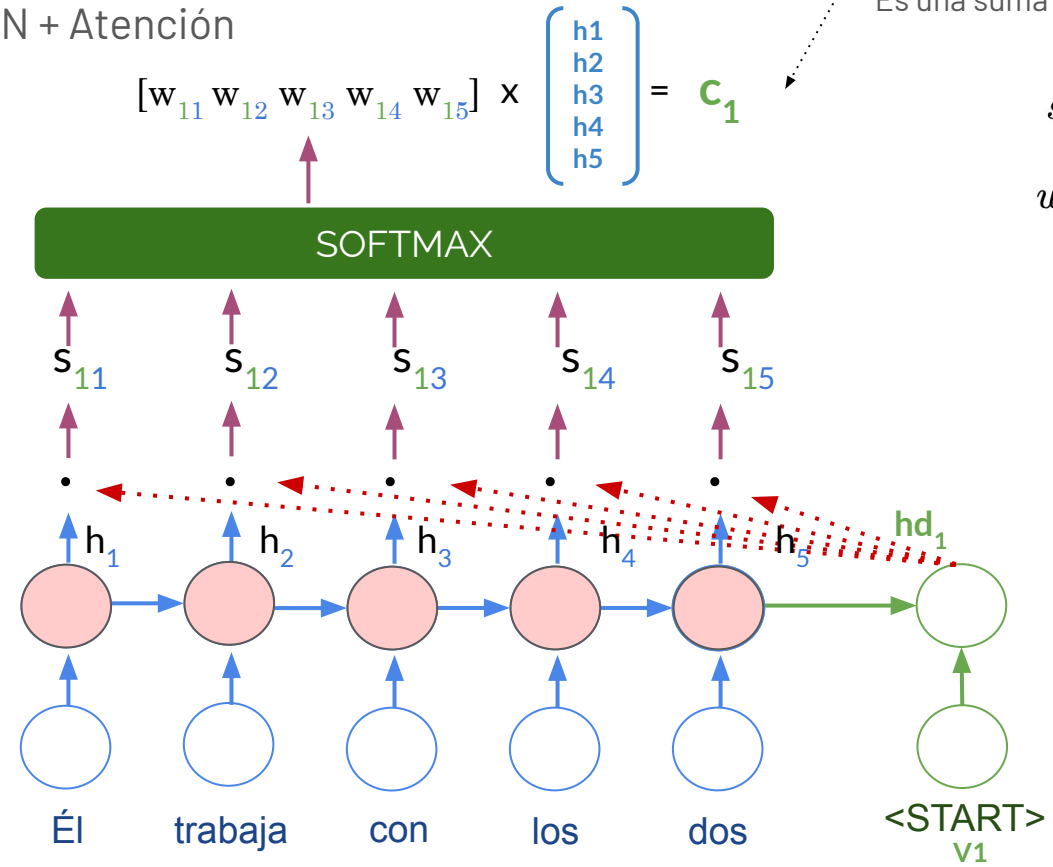
$$s_{ij} = \text{similitud}(\mathbf{h_d}_{\langle i \rangle}^{(k)}, \mathbf{h}_{\langle j \rangle}^{(k)}) = \mathbf{h_d}_{\langle i \rangle}^{(k)} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$

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

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

Encoder - Decoder

RNN + Atención



Vector de contexto

$\mathbf{c}_1$: representación útil para $\mathbf{hd}_1$ de la entrada completa.

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

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

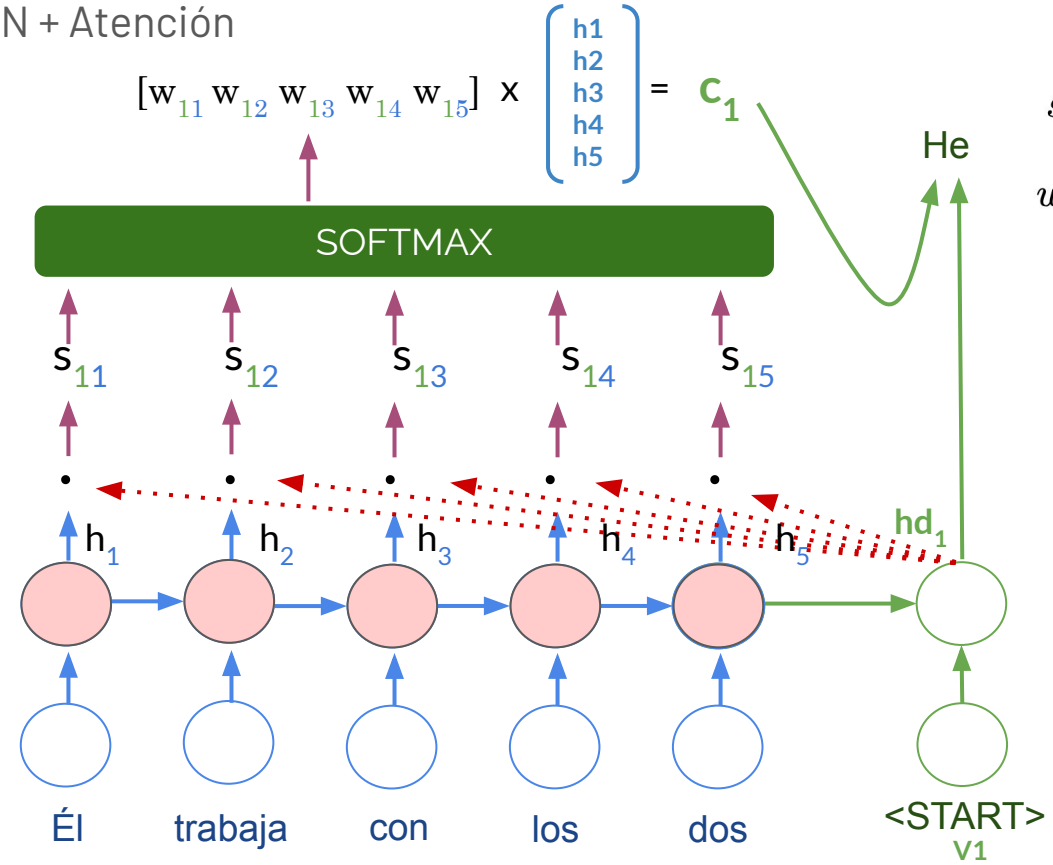
$$w_{ij} = softmax(s_i)[j] = \frac{e^{s_{ij}}}{\sum_{j'} e^{s_{ij'}}}$$

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

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

Encoder - Decoder

RNN + Atención



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

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

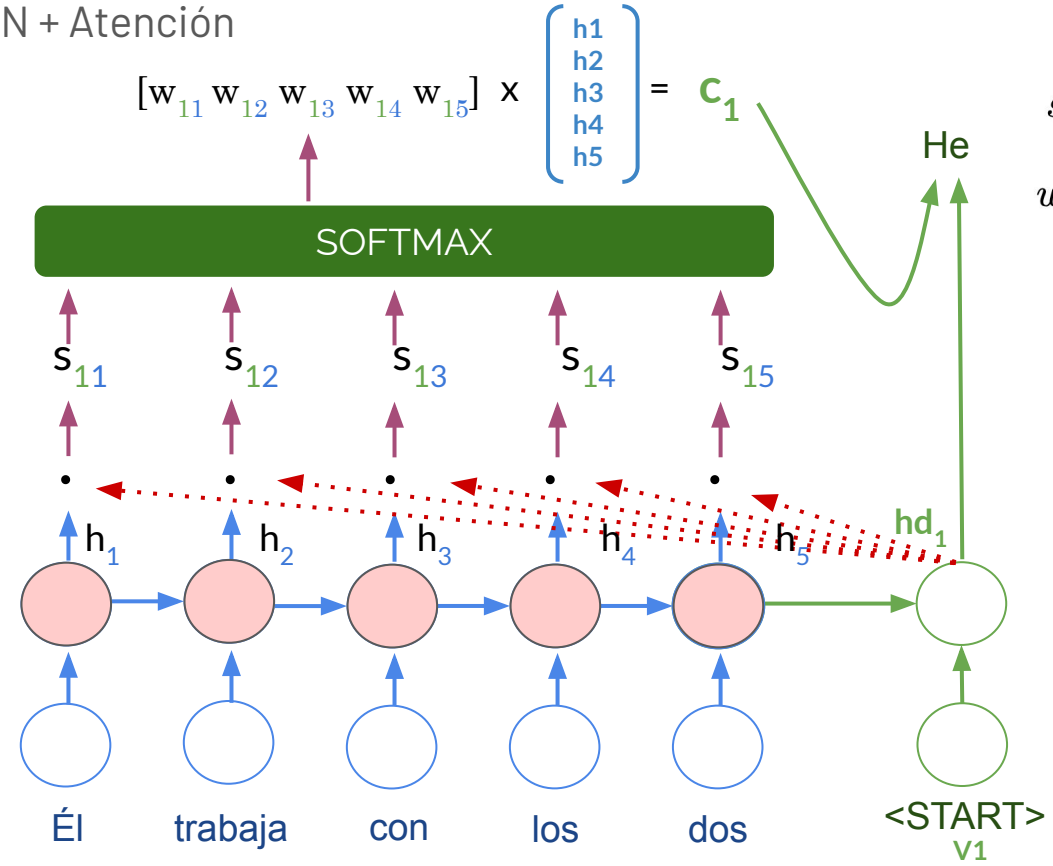
$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$ vector de contexto

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

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

Encoder - Decoder

RNN + Atención



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

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

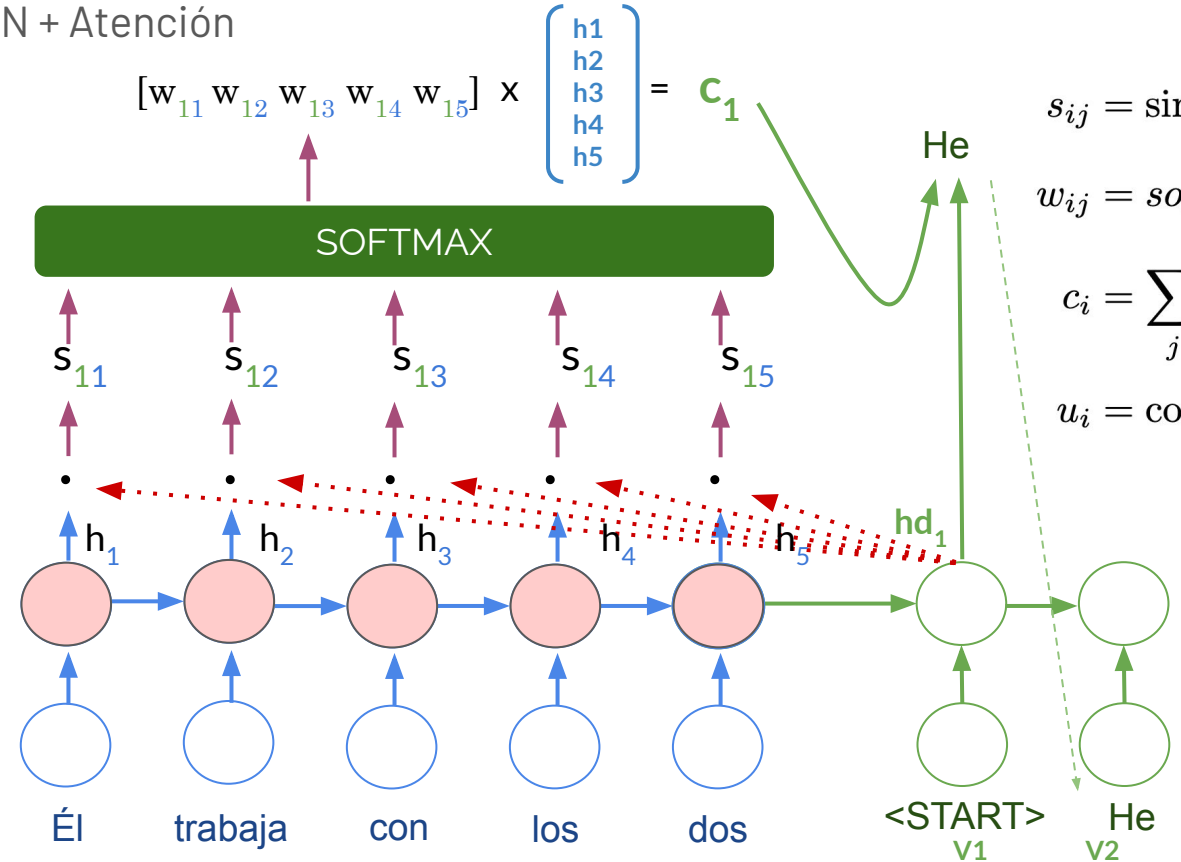
$$c_i = \sum_j w_{ij} \cdot \mathbf{h}_{\langle j \rangle}^{(k)}$$ vector de contexto

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

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

Encoder - Decoder

RNN + Atención



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

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

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

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

Encoder - Decoder

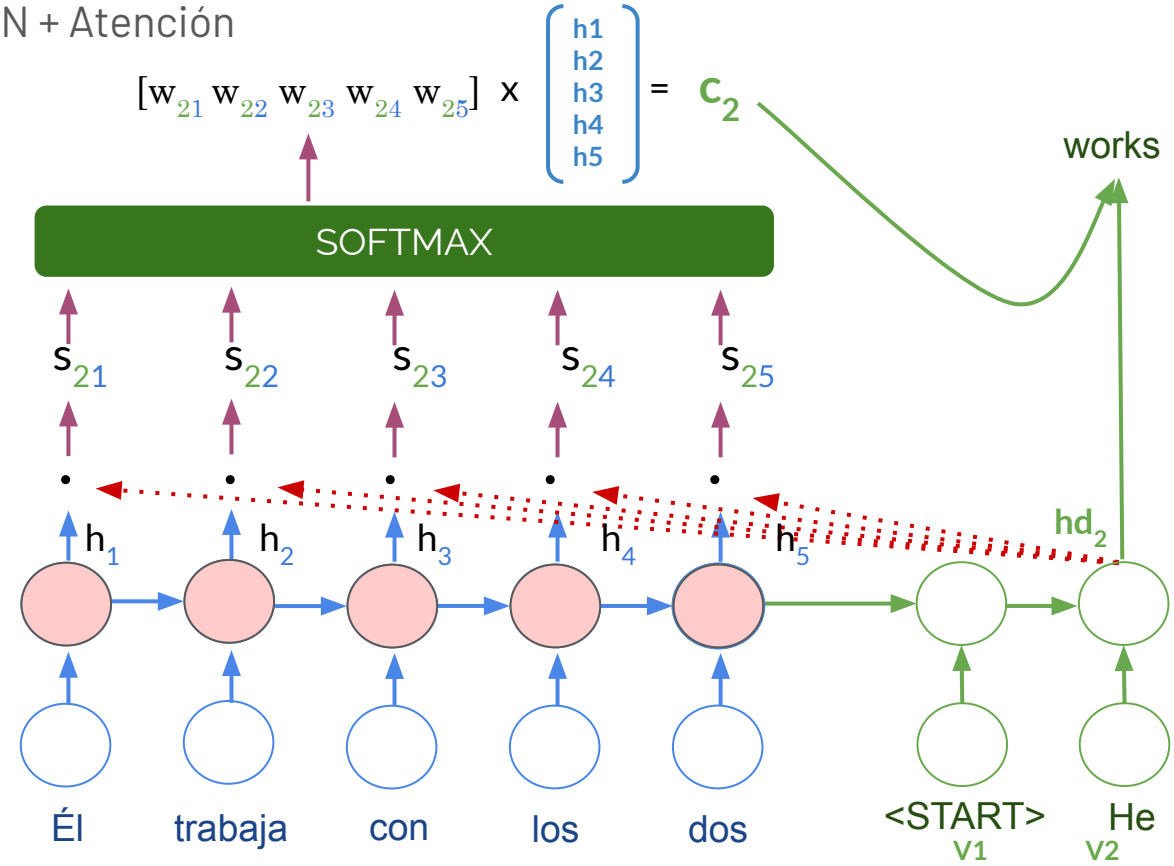
RNN + Atención

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

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

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

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



Encoder RNN

Decoder RNN

Encoder - Decoder

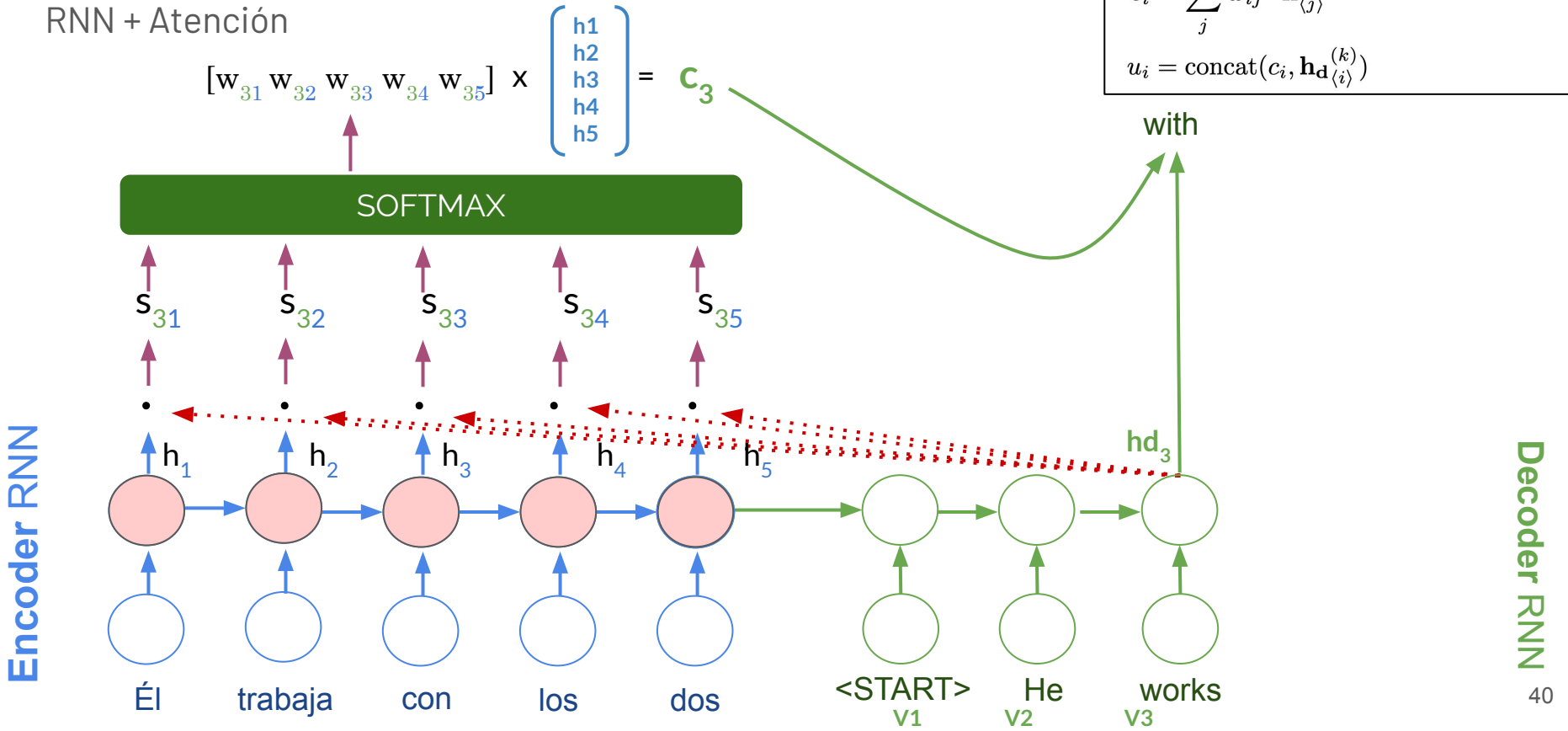
RNN + Atención

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

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

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

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



Encoder - Decoder

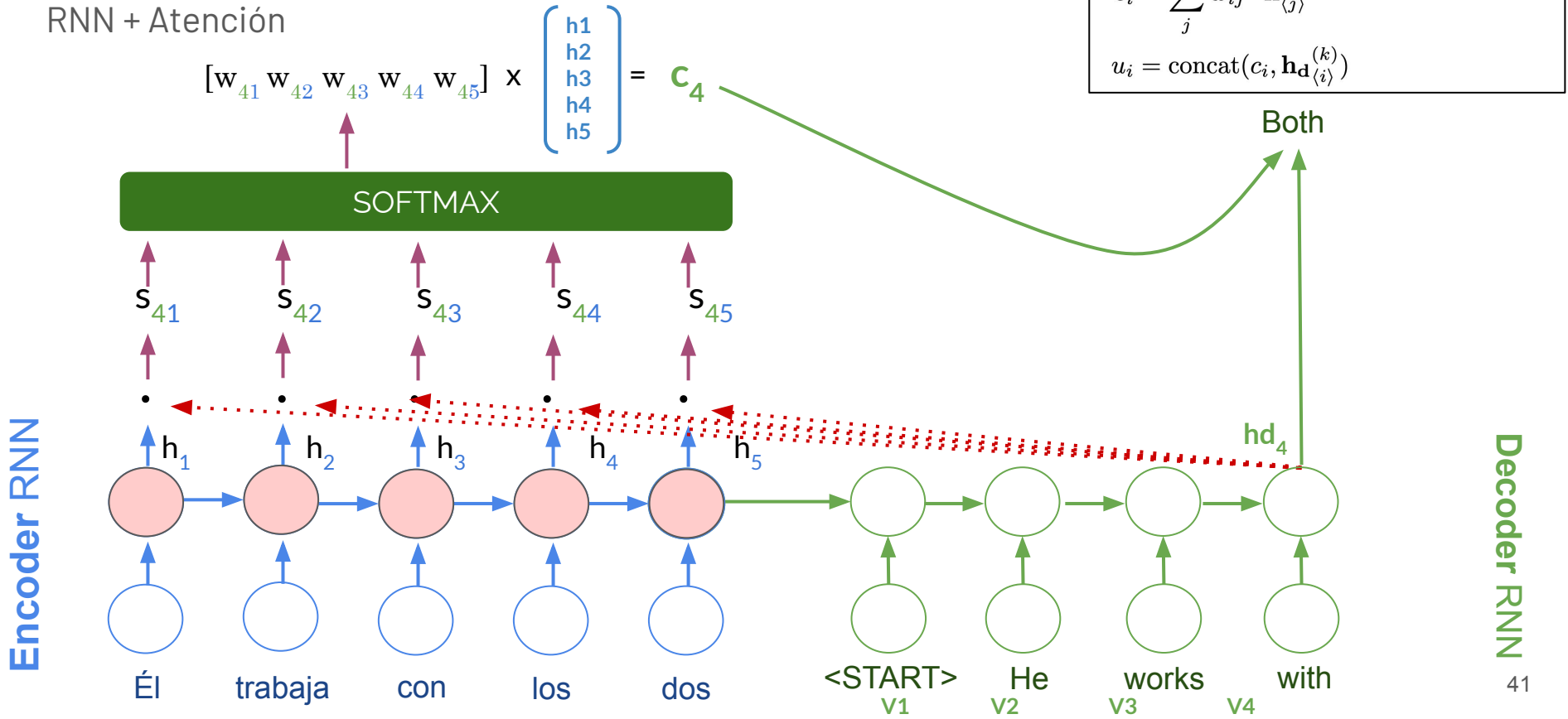
RNN + Atención

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

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

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

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



Encoder - Decoder

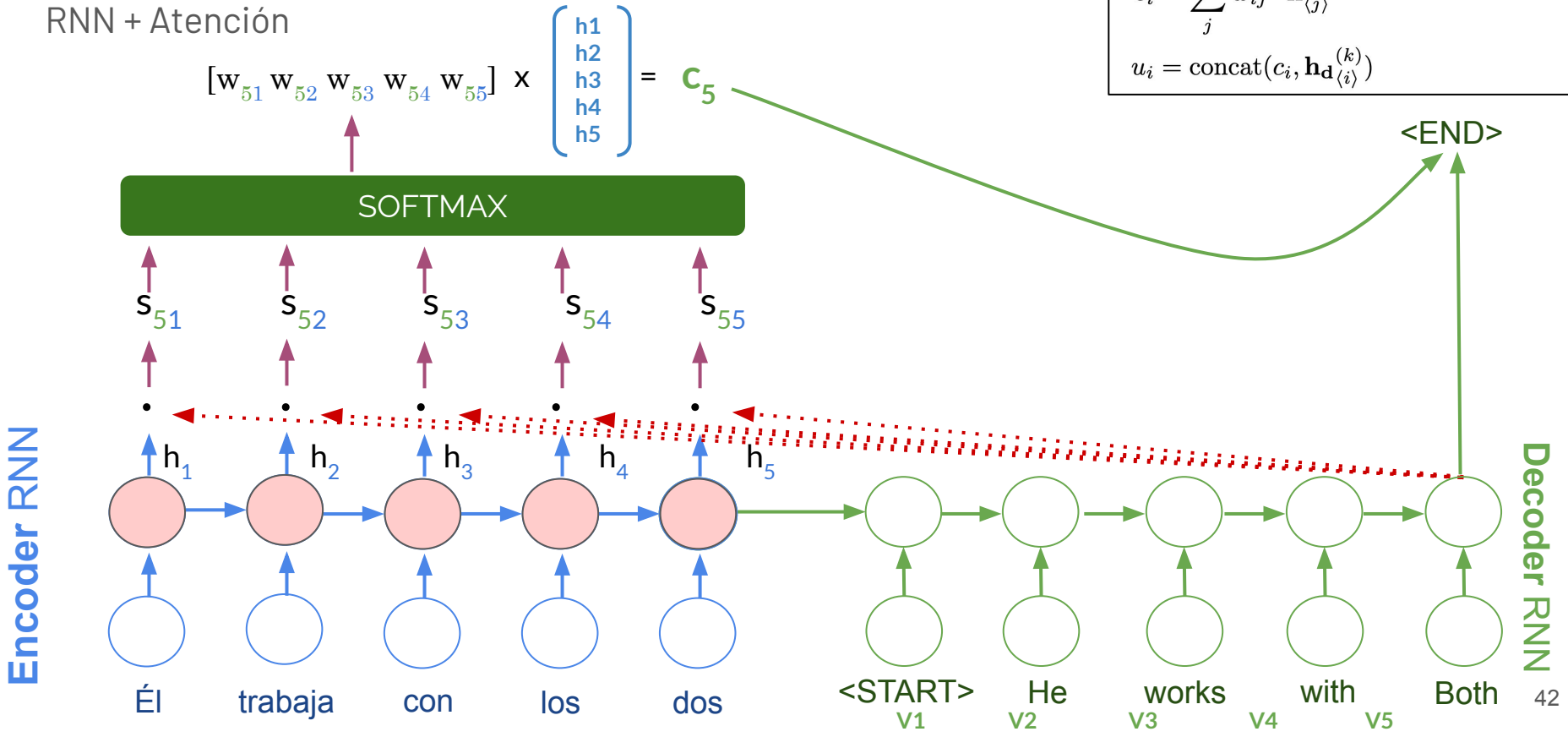
RNN + Atención

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

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

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

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



Encoder - Decoder

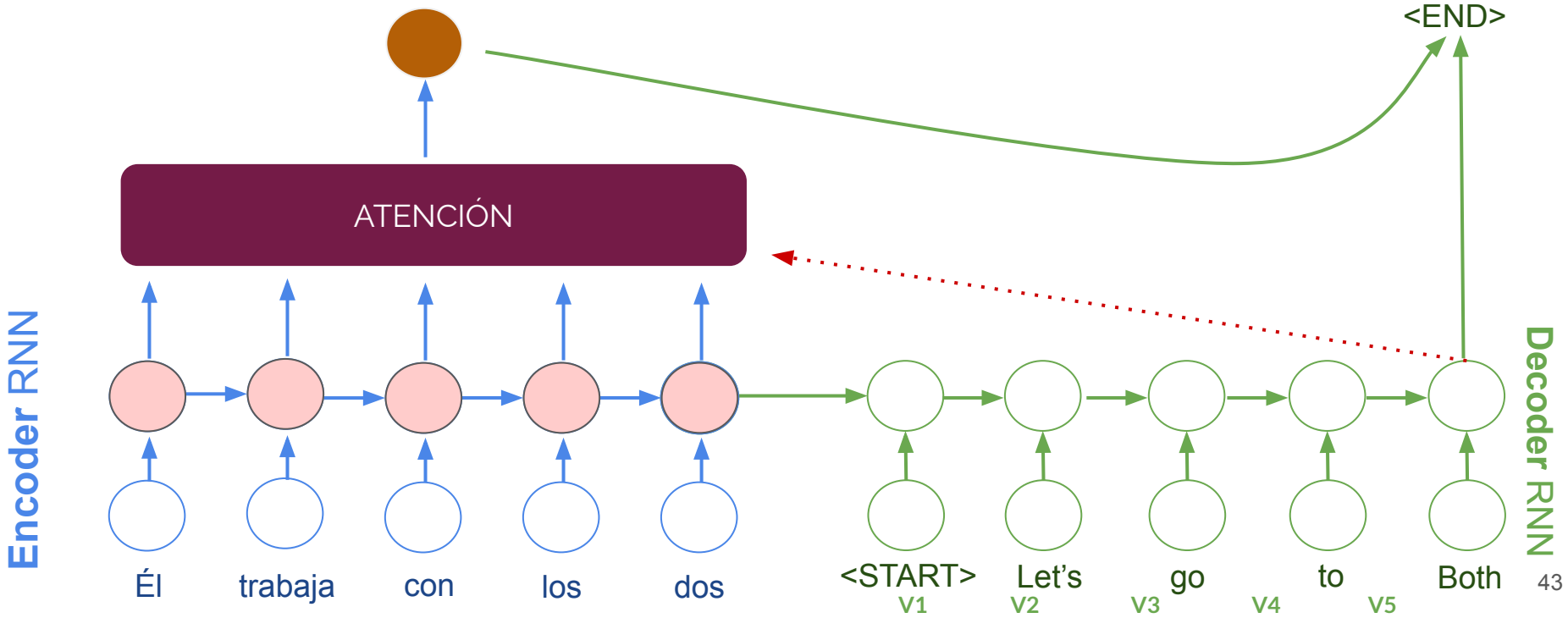
RNN + Atención

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

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

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

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



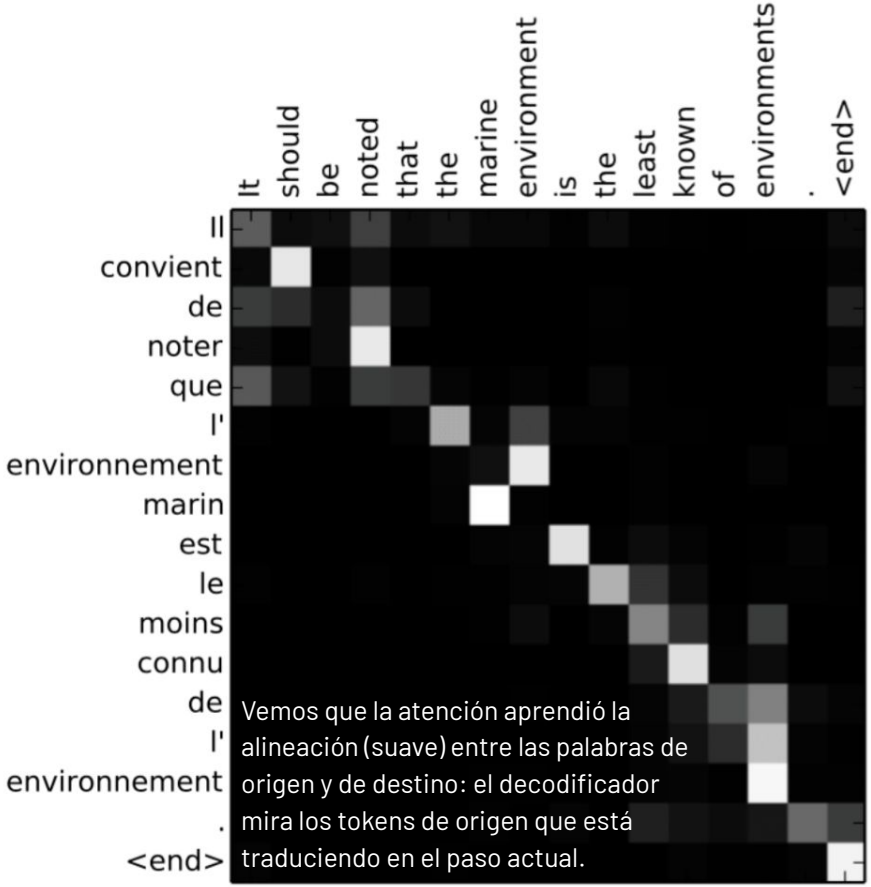
Encoder - Decoder

Resumen

En cada paso del decodificador:

- Recibe entrada de atención: un estado de decodificador $\mathbf{hd}_i$ y todos los estados del codificador $\mathbf{h}_1\ \mathbf{h}_2\ \mathbf{h}_3 \dots\ \mathbf{h}_T$
- Calcula **puntuaciones de atención**:
  - Para cada estado del codificador $\mathbf{h}_j$, la atención calcula su "relevancia" para el estado $\mathbf{hd}_i$ del decodificador: $\mathbf{similitud(hd}_i, \mathbf{h}_j\mathbf{)}$
    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”.

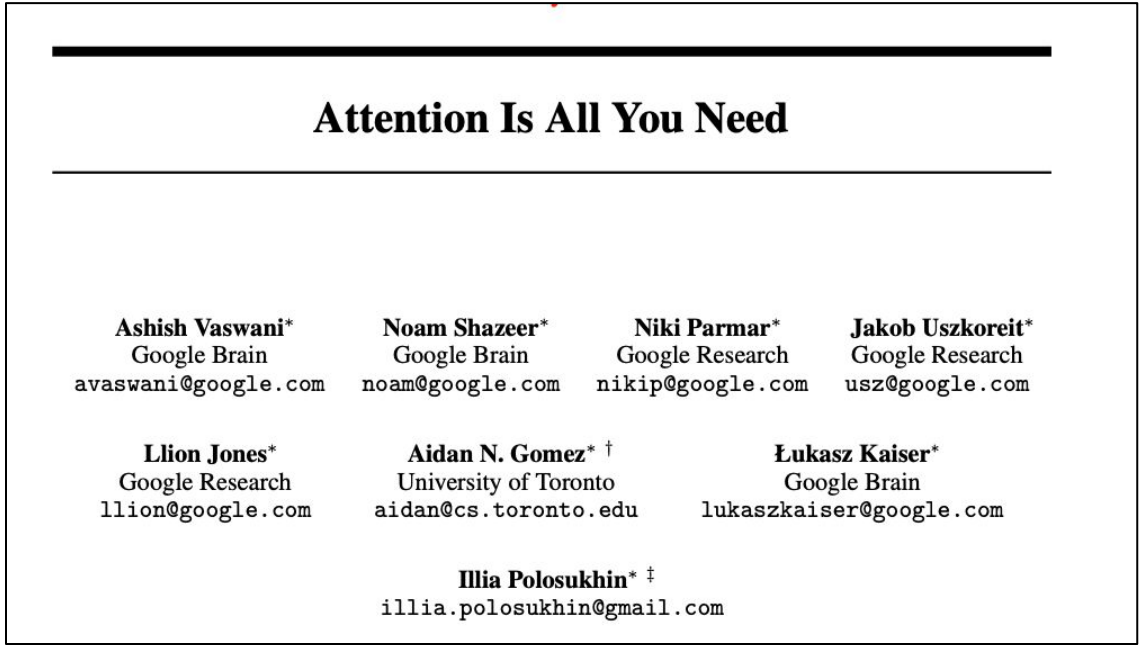
[Neural Machine Translation by Jointly Learning to Align and Translate]

```python
class EncoderRNN(nn.Module): # Igual que antes.

class Attention(nn.Module):
    def __init__(self, hidden_size):
        self.attn = nn.Linear(hidden_size * 2, hidden_size)
        self.v = nn.Parameter(hidden_size)

    def forward(self, hidden, encoder_outputs):
        energy = torch.tanh(self.attn(torch.cat([h, encoder_outputs], 2)))
        energy = self.v @ energy
        attention_weights = torch.softmax(energy)
        return attention_weights

class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, num_layers=1):
        self.rnn = nn.RNN(hidden_size, hidden_size, num_layers, batch_first=True)
        self.attention = Attention(hidden_size)
        self.fc = nn.Linear(hidden_size * 2, output_size)

    def forward(self, x, hidden, encoder_outputs):
        attn_weights = self.attention(hidden[-1], encoder_outputs)
        context = attn_weights @ encoder_outputs
        rnn_input = torch.cat([x, context], 2)
        output, hidden = self.rnn(rnn_input, hidden)
        output = self.fc(torch.cat([output, context]))
        return output, hidden
```

Atención (ver.2017)
Q-K-V

Encoder - Decoder

¿Nos podemos **deshacer** de las RNNs?



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

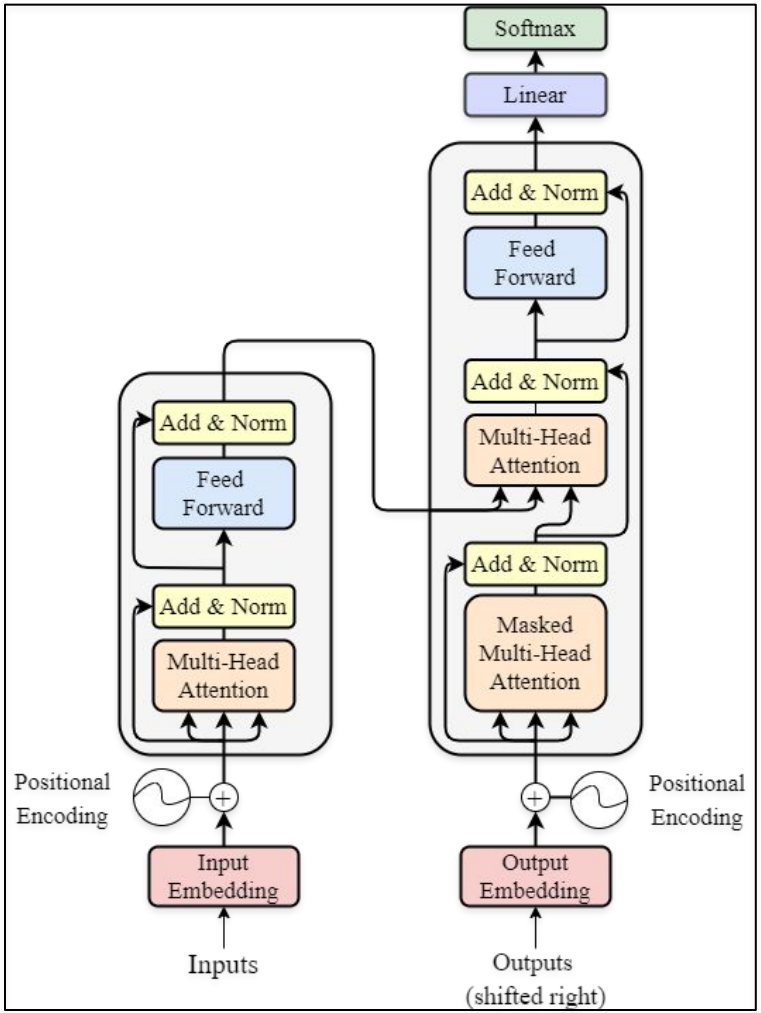
Transformers

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017).
Attention is all you need. *Advances in neural information processing systems*, *30*.

En el paper se presenta la arquitectura “Transformers”. Los transformers siguen la arquitectura Encoder-Decoder.

Pero sin recurrencia, sin convoluciones. **Se procesa toda la serie al mismo tiempo.**

Tiene muchas componentes que lo hacen funcionar. Hoy nos concentramos sólo en el **proceso de atención**



Encoder - Decoder

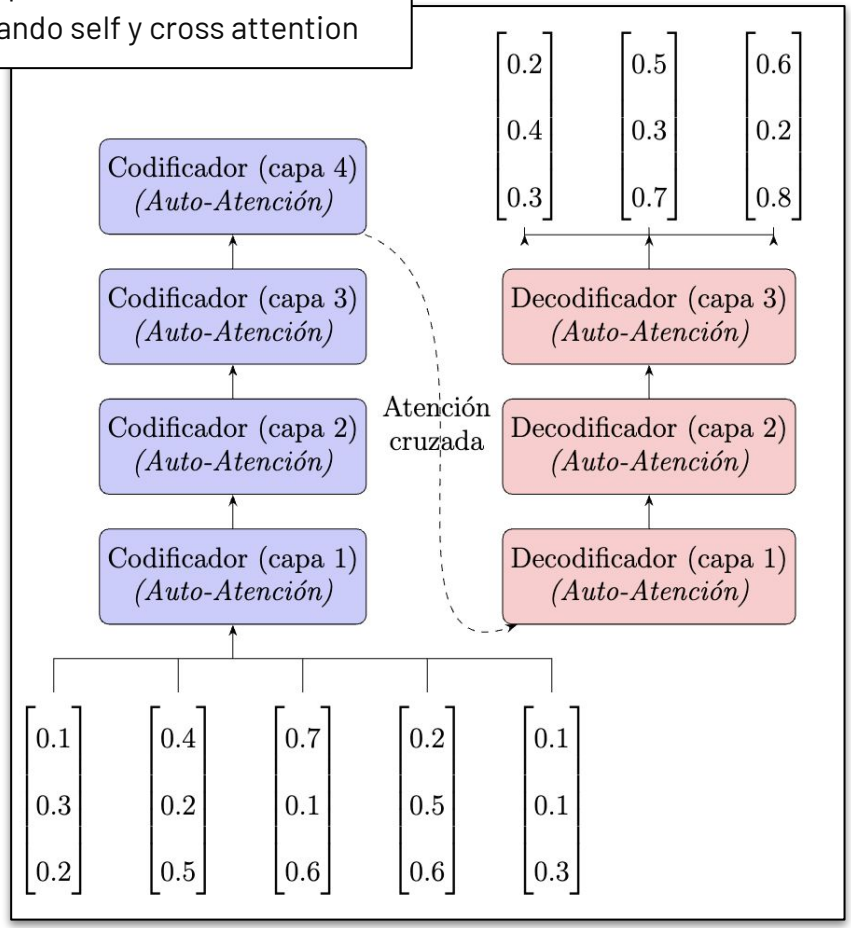
¿Cómo codifica/decodifica?
Utilizando varias capas consecutivas apiladas.

Cada capa genera una nueva representación de la serie temporal.

Autoatención: "Self-Attention": Genera representaciones contextuales de una secuencia.

Atención Cruzada: "Cross-Attention": permite que el modelo pondere la importancia de la salida del codificador para la decodificación. (como vimos en RNNs)

Esquema **Encoder-Decoder** usando self y cross attention



Autoatención (Self-Attention)

Procedimiento (versión simplificada). Idea: Entran embeddings, salen embeddings con más contexto.

1. Cada palabra (su embedding) en la secuencia se multiplica con el resto para obtener similitudes entre pares.

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

(se parece mucho a lo que hacíamos con RNNs, pero ahora son los embeddings directo, no los hidden de una RNN)

2. Puntuaciones de Atención: se aplica una función softmax para obtener los pesos de atención

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

(en RNNs, lo que llamábamos pesos de alineación)

3. Suma Ponderada: Multiplicar los pesos de atención por los inputs para obtener la salida para cada token.

$$\mathbf{new\_token}_{\langle j \rangle}^{(k)} = \sum_j w_{ij} \cdot \mathbf{token}_{\langle j \rangle}^{(k)}$$

(en RNNs, el vector de contexto)

4. Repetir “cantidad de capas”-veces, tomando $\text{new\_token}_{<j>}$ como entrada en el tiempo <j> de la siguiente capa.

Autoatención (Self-Attention)

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

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

$$\mathbf{new\_token}_{\langle j \rangle}^{(k)} = \sum_j w_{ij} \cdot \mathbf{token}_{\langle j \rangle}^{(k)}$$

¿Y qué es lo que se aprende? (hasta acá no hay parámetros del modelo - recordar que los w_{ij} son escalares)

query key

$$s_{ij} = \text{similitud}(\mathbf{token}_{\langle i \rangle}^{(k)}, \mathbf{token}_{\langle j \rangle}^{(k)}) = (W^q \mathbf{token}_{\langle i \rangle}^{(k)})^T \cdot (W^k \mathbf{token}_{\langle j \rangle}^{(k)})$$

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

$$\mathbf{new\_token}_{\langle j \rangle}^{(k)} = \sum_j w_{ij} \cdot (W^v \mathbf{token}_{\langle j \rangle}^{(k)})$$

value

Self-Attention

Recomiendo este paso a paso
https://colab.research.google.com/drive/1rPk3ohrmVclghH7uQ7qys4oznDdAhpzF

La versión completa (matricial) tiene esta pinta:

$$\text{Atención}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

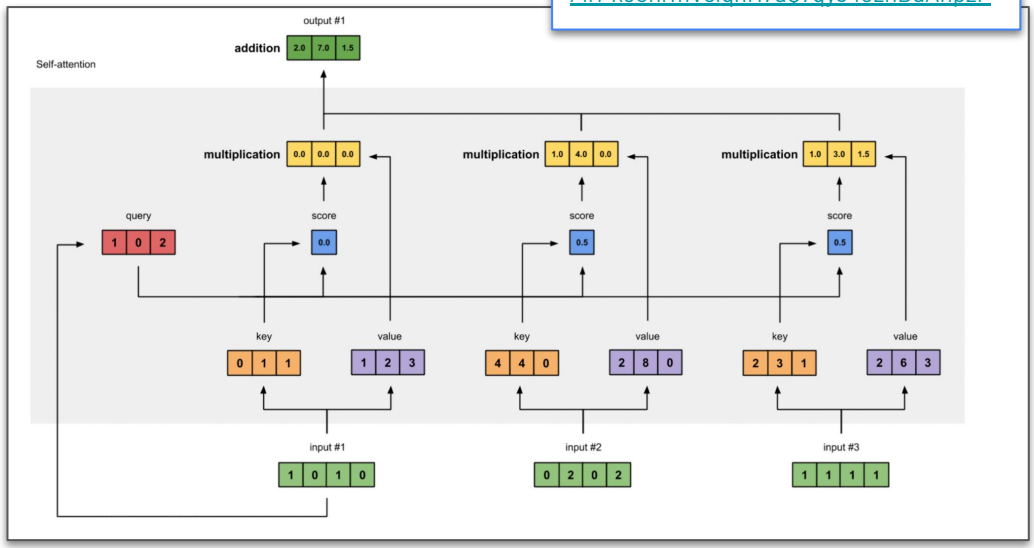
$Q = XW^q$ (Matriz de consultas, Queries)

$K = XW^k$ (Matriz de claves, Keys)

$V = XW^v$ (Matriz de valores, Values)

$d_k =$ Dimensión de las claves

$X =$ Entrada o Salida de la capa anterior de atención



Resumen:

- Vimos una idea intuitiva del mecanismo de self-attention.
- **Cross-attention:** Muy similar, pero k, q y v no parten del mismo vector de entrada.
- Este mecanismo permite aprender representaciones contextuales y útiles de secuencias (de longitud fija)

Recomiendo indagar más en el tema: **Understanding Deep Learning** (Simon J.D. Prince): **Capítulo 12 (transformers)**
Temas como *Positional Encoding; Multihead Attention; Conecciones Residuales, etc .*

Tarea

Completar el formulario

Lecturas obligatorias:

➢ Deep Learning: **Capítulo 10** (hasta sección **10.5 inclusive**).

Opcional (pero necesario para la vida).

- Paper 1: **"Neural Machine Translation by Jointly Learning to Align and Translate"**
- Paper 2: **"Attention Is All You Need".**
- Paper 3: **"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"**

Opcional:

- Understanding Deep Learning (Simon J.D. Prince): **Capítulo 12 (transformers)**
- Deep Learning: Resto del **Capítulo 10**.
- Blogs recomendados:
  - https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html
  - https://distill.pub/2016/augmented-rnns/
  - https://colab.research.google.com/drive/1rPk3ohrmVclqhH7uQ7qys4oznDdAhpzF