# Redes neuronales recurrentes

## Aprendizaje profundo

Departamento de Sistemas Informáticos

E.T.S.I. de Sistemas Informáticos - UPM

En este tema hablaremos de redes neuronales recurrentes (RNN)

- RNN $\rightarrow$ *Recurrent neural networks*
- Permiten tratar con **información secuencial**, pero no se limitan al análisis
- También **pueden generar datos** a partir de una o más entradas

Aprenderemos qué posibilidades nos ofrecen estos modelos y sus limitaciones

- Y técnicas para superar (casi mejor «mitigar») estas limitaciones

Las RNN tienen muchas aplicaciones prácticas en el mundo real

- Sobre todo en el procesamiento de lenguaje natural (NLP)
  - nlp $\rightarrow$ *Natural language processing*
- De hecho ambas áreas han evolucionado a la par en los últimos años

# Series temporales

# ¿Qué es una serie temporal?

Es una **secuencia de observaciones** de una variable medida en el tiempo

- Se **secuencia ordenada de datos** en la que valores consecutivos están correlacionados
- Estos datos pueden ser tanto **numéricos** como **simbólicos**

Las series temporales se clasifican en:

- **Univariantes**: Secuencias de un único valor (p.ej. temperatura, precio de una acción).
- **Multivariantes**: Secuencias con múltiples variables (p.ej. datos de acelerómetros o pistas separadas en una canción).
    - Se suelen representar descomponiéndolas en series univariantes

# ¿Por qué son importantes las series temporales?

La información suele llegar de forma secuencial, no como muestras aisladas.

- En general los datos están intrínsecamente ligados a la noción de tiempo

Los humanos también procesamos la información así $\rightarrow$ **Memoria secuencial**

- *Mecanismo del cerebro que facilita el reconocimiento de patrones secuenciales*
  i. Recita el alfabeto: *ABCDEFGHIJKLMNÑOPQRSTUVWXYZ*
     - Es fácil porque lo hemos aprendido así desde pequeños
  ii. Ahora al revés: *ZYXWVUTSRQPOÑNMLKJIHGFEDCBA*
     - Es muy difícil, a no ser que lo hayamos practicado mucho anteriormente
  iii. Ahora empieza a recitar el alfabeto desde la letra F
     - Al comienzo suele costar un poco (hay que localizar el comienzo del patrón)
     - El resto, una vez reconocido el patrón, sale de forma natural

# Problemas que involucran series temporales

- **Predicción**: Predecir el valor futuro de una variable en base a su pasado
  - P.ej. pronóstico del tiempo, predicción de ventas, etc.
- **Detección de anomalías**: Detectar valores atípicos en una serie de eventos
  - P.ej. detección de fraudes, detección de intrusos, etc.
- **Reconomiento de patrones**: Extraer patrones recurrentes de una serie de eventos
  - P.ej. detección de tendencias, detección de patrones de comportamiento, etc.

# Una advertencia

Cuidado con la **información futura en las características de entrada**

- Es un problema mucho más común de lo que puede parecer

En resumen, ocurre cuando entrenamos un modelo con datos pasados y datos *futuros*

- Especialmente cuando vamos a realizar un test del modelo
- Si separamos la serie en entrenamiento y test:
  - No basta con realizar una separación aleatoria de los ejemplos
  - Hay que separar los datos por tiempo: entrenamiento **ANTES** que test
  - Si no, estamos entrenando con información futura (**mal**)
- Insistimos: **Cuidado con añadir información futura en las entradas**

# Ventanas móviles (I)

Los problemas de predicción se basan en la idea de predecir el futuro dados:

1. El valor actual de la secuencia
2. El conocimiento almacenado en su estado interno

Podemos estimar este conocimiento como el histórico de los valores previos:

1. Tomamos una ventana de tamaño fijo en la secuencia
2. Utilizamos los datos de la ventana para predecir valores futuros de la secuencia
3. Deslizamos la ventana hacia adelante (p.ej. la mitad del tamaño de la ventana)
4. Volvemos al paso 2.

También se conocen como *sliding windows* o *rolling windows*

# Ventanas móviles (II)

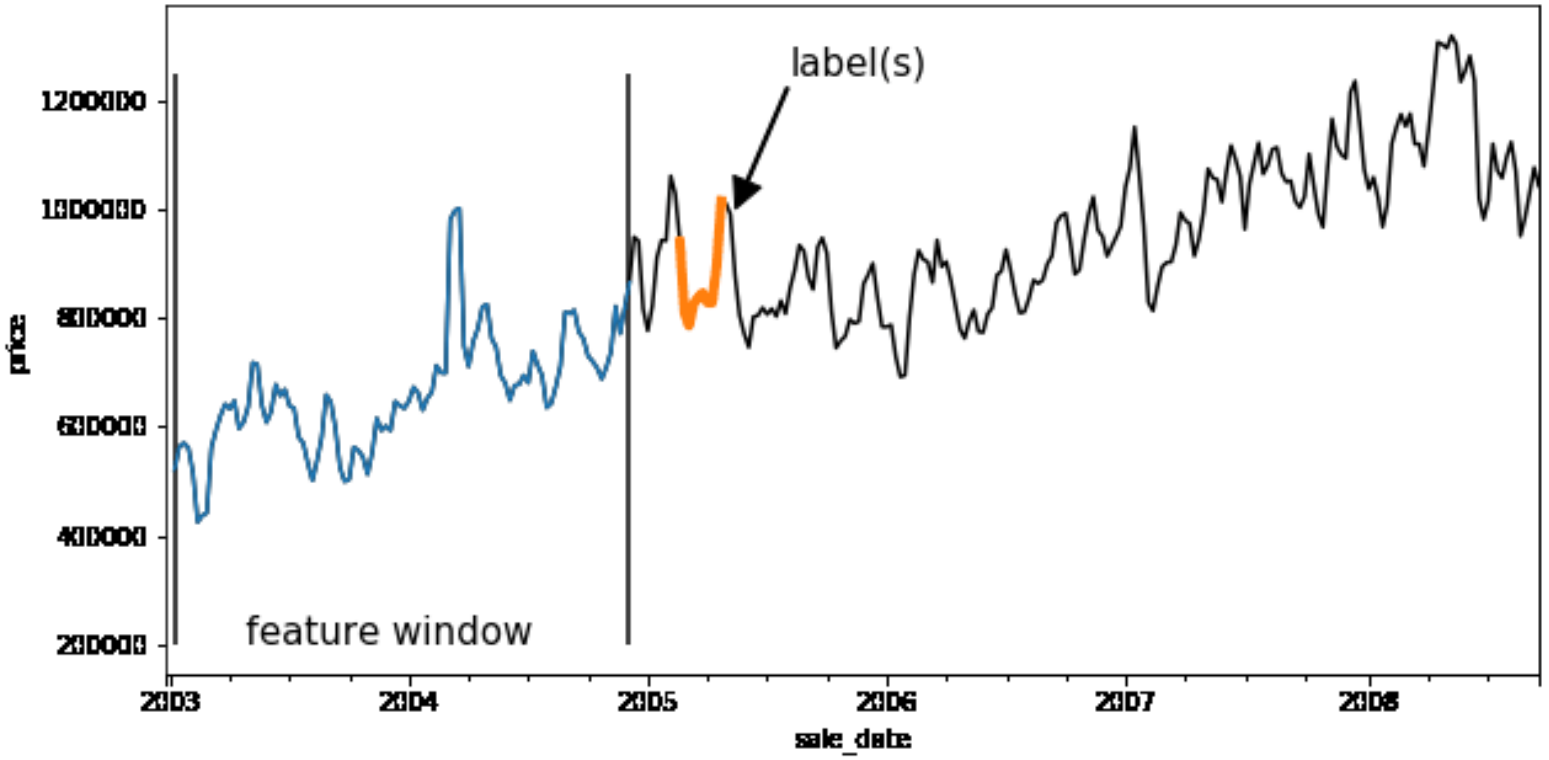


***Figura 1**. Ejemplo de ventana móvil. Fuente: Google Cloud Blog*

# Ventanas móviles (III)

Este enfoque se usa en muchas aplicaciones con éxito

- De hecho es el enfoque más simple y usado en series temporales
- En aprendizaje automático clásico se suele utilizar para mejorar las predicciones
- En el caso concreto del aprendizaje profundo hay dos aproximaciones:

  i. Alimentar la ventana entera a la red neuronal
     - Típico en perceptrones multicapa y también en redes convolucionales

  ii. Alimentar los valores uno a uno e ir manteniendo una memoria
     - ¿Cómo que memoria? Sin estrés, en unos momentos lo vemos

# Un poco de series temporales.ipynb

# Redes neuronales recurrentes

# Intuición

*Una decisión **ahora** no se basa solo en lo que he percibido **ahora***

Razonamos usando información previa, pero las redes *feed-forward*:

- No pueden manejar información de entrada secuencial variable
- En la salida solo se puede usar la información de la entrada actual
- No pueden memorizar entradas pasadas para predicciones

Una red recurrente añade **retroalimentación** a una **red neuronal**

- Aprovechan la naturaleza secuencial de los datos para predecir
- Cada salida depende implícitamente de todas las anteriores
- Mantienen un conocimiento histórico gracias a su memoria interna

# ¿Qué son las redes neuronales recurrentes?

Son redes neuronales que reutilizan *salidas* anteriores como *parte de su entrada*

- Fueron introducidas durante la década de los 80[1]
  - Fueron poco populares en la época por sus requisitos funcionales para entrenar

Su propia salida es parte de la entrada en la siguiente predicción

- Esto permite que la red recuerde información de entradas anteriores
- Mantienen un estado interno (memoria) que se actualiza en cada paso de tiempo
- Son muy útiles cuando el contexto es fundamental

[1] Concretamente con los trabajos de las redes de Hopfield (Neural networks and physical systems with emergent collective computational abilities.) y de Elman (Finding structure in time).

# Intuición del funcionamiento de una RNN (I)

Supongamos que queremos predecir si va a llover a partir de observaciones

- Sin entrar en detalles, todo conceptual

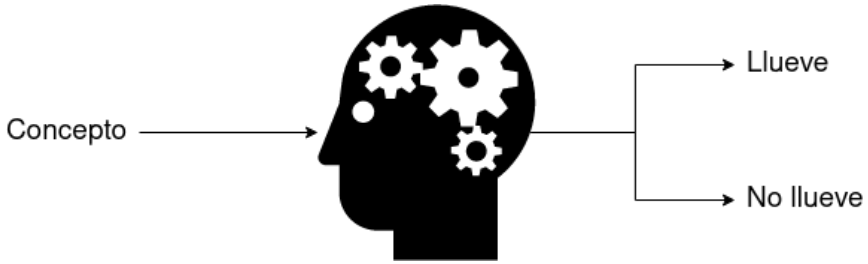


***Figura 2**. Predecimos el tiempo mediante un clasificador que se alimenta con conceptos.*

La naturaleza secuencial de la red implica varios tipos de problemas

- Puede haber una o varias entradas y una o varias salidas
- Lo veremos más adelante explorando los diferentes tipos de problemas en RNN

# Intuición del funcionamiento de una RNN (II)

Observación en $t$: `(niña, jugando, sola)`

- No hemos observado nada previamente y el concepto no indica si va a llover o no
- Sin embargo, **la memoria interna tiene ahora una representación de la entrada**

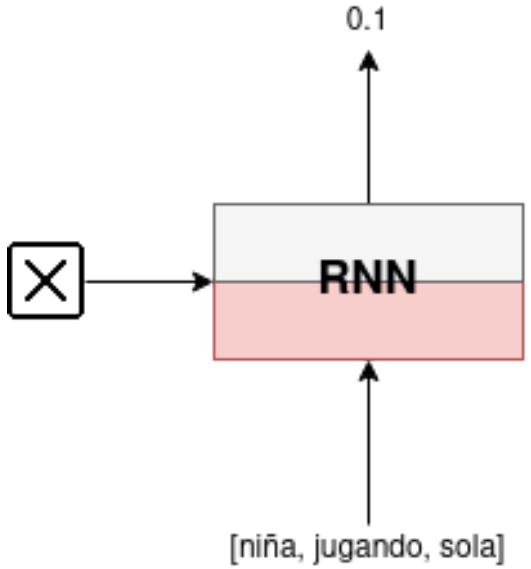


***Figura 3**. En el primer instante de tiempo sólo contamos con la información que acabamos de observar.*

# Intuición del funcionamiento de una RNN (III)

Observación en $t + 1$: `(pasa, vehículo, rojo)`

- La predicción no cambia en absoluto
- Sin embargo, *la memoria va **manteniendo conocimiento** de cada concepto pasados*

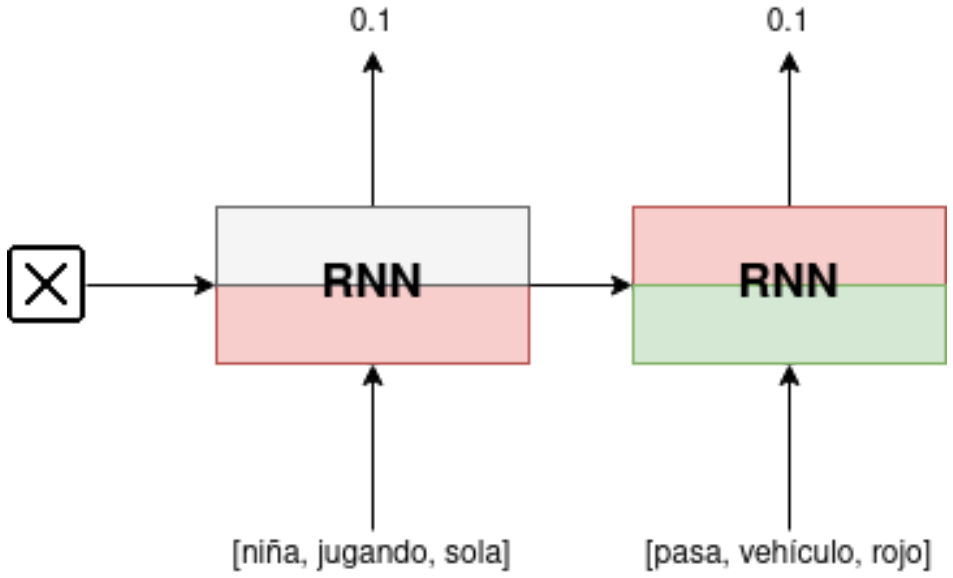


***Figura 4**. El segundo instante de tiempo posee conocimiento en memoria y en entrada inmediata.*

# Intuición del funcionamiento de una RNN (IV)

Observación en $t + 2$: `(pájaros, volando, bajo)`

- La nueva observación aumenta la probabilidad a 0,7
- La memoria mantiene información acerca de las dos observaciones previas

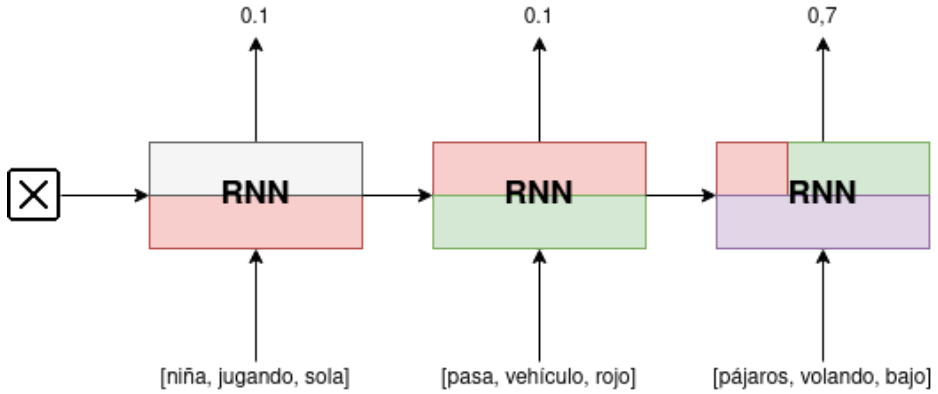


***Figura 5**. La memoria sigue almacenando conocimiento pasado según va llegando información nueva.*

# Intuición del funcionamiento de una RNN (V)

Observación en $t + 3$: `(lavado, de, coche)`

- La red entiende que los últimos dos conceptos aumentan la probabilidad de lluvia
- Se sigue manteniendo el conocimiento de conceptos anteriores
  - Lo malo es que la representación de los primeros se va diluyendo rápidamente

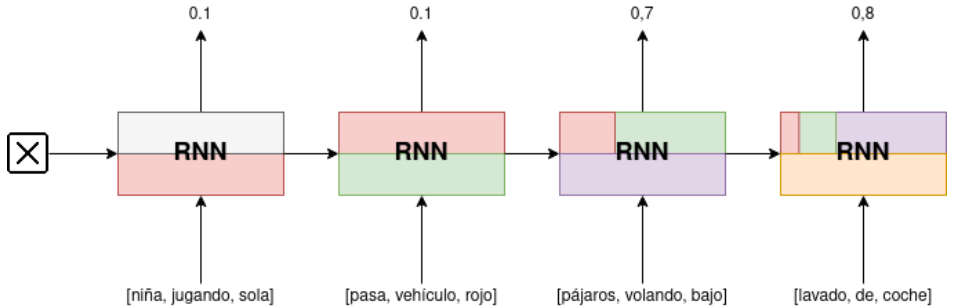


***Figura 6**. El conocimiento pasado se mantiene, pero el más antiguo va disminuyendo en favor del nuevo.*

# Intuición del funcionamiento de una RNN (VI)

Observación en $t + 4$: `(cielo, con, nubarrones)`

Ha aumentado al predicción mucho

- Los conceptos anteriores "se recuerdan" y afectan a la inferencia
- Eso sí, los primeros solo la aumentan marginalmente

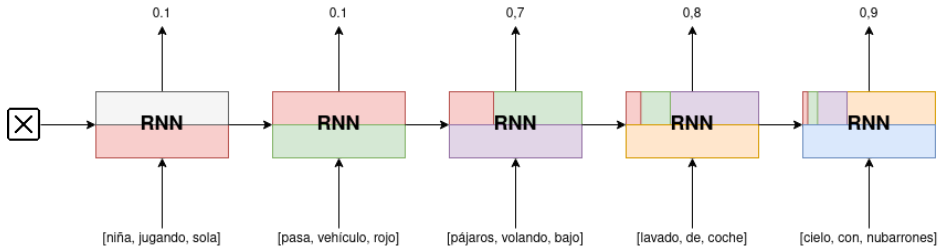


***Figura 6**. La predicción sigue usando conocimiento de la primera, aunque en mucha menor medida.*

# Intuición del funcionamiento de una RNN (VII)

El ejemplo ilustra cómo la red almacena la información

- Se mantiene la información de la salida anterior

Sin embargo, se intuye un problema de esta memoria

- La información anterior almacenada tiende a 0 en pocos pasos
- Ideal → Que la neurona decida qué recordar y qué olvidar
  - Bueno, lo hace (pesos de la retroalimentación) pero no es eficiente
  - Spoiler alert: Veremos una mejora más adelante

# La unidad recurrente simple (SRU) (I)

Recurrencia $\longrightarrow$ Dependencia del valor actual con los valores anteriores

$$y_t = f(x_t, y_{t-1})$$

Las SRU son equivalentes a las neuronas en redes neuronales simples

- De hecho son muy parecidas a las neuronas de una red feed-forward
- Únicamente tienen una retroalimentación de la salida
- Una red neuronal recurrente es una combinación de una o más SRU
    - Sus arquitecturas varían con los tipos de problemas que se quieren resolver

# La unidad recurrente simple (SRU) (II)

Existen dos formas de representar una SRU

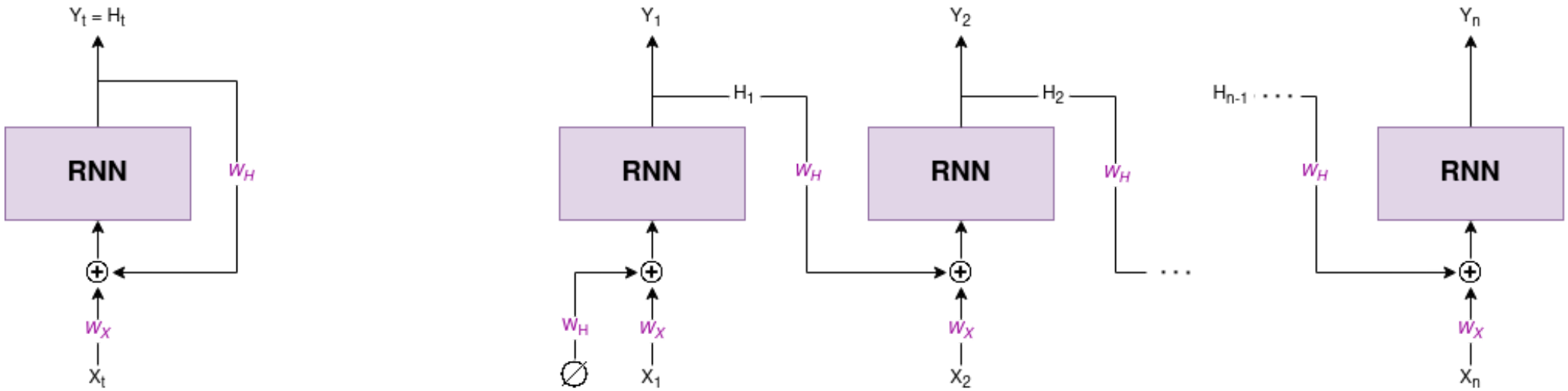


***Figura 7**. Las dos representaciones típicas de una SRU, compacta (izquierda) y desplegada o unrolled (derecha).*

Ambas son equivalentes, se usan indistintamente

- Generalmente se prefiere usar la *unrolled* para facilitar la comprensión

# La unidad recurrente simple (SRU) (III)

Ecuación muy parecida a un MLP:

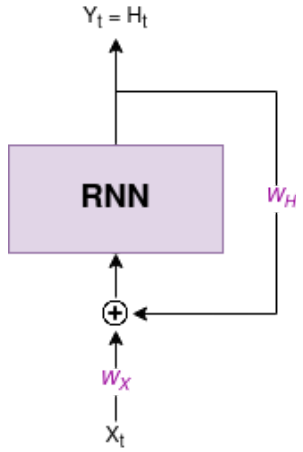


***Figura 8**. Representación compacta de una SRU.*

$$Y_t = f(W_X X_t \frown W_H H_{t-1}) = H_t$$

Siendo:

- $X_t$ e $Y_t$: Entrada y salida de la SRU
- $f$: Activación (ReLU, tanh, sigm, ...)

Si pensamos en $H_{t-1}$ como memoria

- $W_H \rightarrow$ ¿Qué recuerdos considerar?

¿Y cómo aprenden? $\rightarrow$ **Backpropagation through time (BPTT)**[2]

[2] No entramos en la implementación porque es compleja y no aporta nada al curso. Dos recursos interesantes del concepto los podemos encontrar en la entrada de la Wikipedia y el post A Gentle Introduction to Backpropagation Through Time de Jason Brownlee.

# Un detalle al que prestar atención

Hemos visto que el valor de salida en $t$ se obtiene con la siguiente fórmula:

$$H_t = f(W_X X_t \frown W_H H_{t-1})$$

Fijémonos en que:

- Sólo intervienen dos pesos:
  - $W_X$ que afecta a la entrada actual y $W_H$ que afecta al valor anterior de la salida $h_{t-1}$
- Los pesos **no** dependen del tiempo
  - $W_X$ (¿A qué debo prestar atención ahora?) y $W_H$ (¿Qué debo recordar de lo que pasó?)

Pero las RNN sólo funcionan bien para la memoria a corto plazo

- Sobre todo si están basadas en unidades unidades recurrentes simples

# Implementando redes recurrentes simples.ipynb

# Oye, pero he leído por ahí que ...

... está demostrado que un MLP es capaz de representar cualquier función

Sí, pero terminaríamos con *muchos parámetros* y *ninguna estructura*

- Las RNN **aprovechan la estructura secuencial de los datos** en su arquitectura
- Es la misma historia que al comparar un MLP con una CNN
  - La CNN tiene una estructura interna que aprovecha la estructura espacial de los datos
- Al final utilizamos una fracción de los pesos de una forma más inteligente

También está el problema del tamaño de entrada constante

- Las redes feed-forward **deben tener un tamaño de entrada constante**
- En el mundo real tanto imágenes como secuencias tienen distintos tamaños

# Pero entonces un MLP normal vale, ¿no?

¡Por supuesto! Para eso están las *sliding windows*

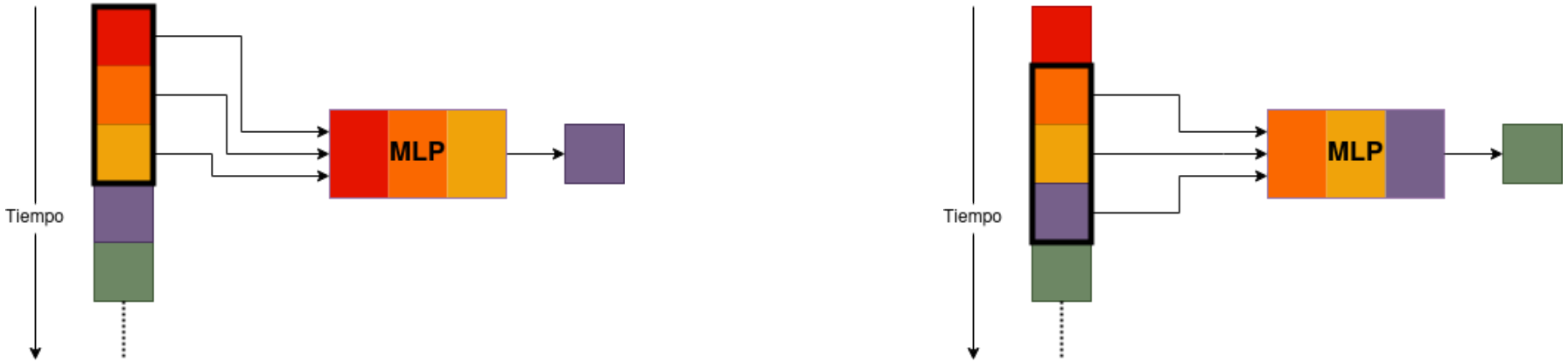


***Figura 9**. Al conjunto de datos se le llama overlapping windows dataset.*

Lo malo es que tienen ciertos inconvenientes

- Requieren un tamaño de entrada y salida fijos (o reentrenar)
- En realidad no hay memoria, solo lo que hay en la ventana

# Además, echando cuentas ...

Supongamos un problema de clasificación de secuencias con:

- ... una longitud de secuencia de $T = 20$, ...
- ... una entrada de dimensión $D = 30$, ...
- ... una capa oculta de tamaño $M = 16$, ...
- ... y $k = 3$ clases de salida

En un MLP

$$I \to H: \quad T \times D \times M = 9600$$
$$H \to O: \quad M \times k = 48$$

Total: *9648* parámetros a ajustar

En una RNN

$$I \to H: \quad (D + M) \times M = 736$$
$$H \to O: \quad M \times k = 48$$

Total: *784* parámetros a ajustar

# *Stacked* RNN

Apilar una SRU sobre otra nos lleva al concepto de RNN profundas (DRNN)

- Le damos los valores de entrada a la primera unidad ...
- ... la salida de la primera a la segunda ...
- ... y así sucesivamente ...

$$
\begin{aligned}
H_t^1 &= f(W_X^1 X_t + W_H^1 H_{t-1}^1) \\
H_t^2 &= f(W_i^2 H_{t-1}^1 + W_H^2 H_{t-1}^2) \\
&\dots \\
H_t^n &= f(W_i^n H_{t-(n-1)}^{n-1} + W_H^n H_{t-(n-1)}^n)
\end{aligned}
$$

En realidad es prácticamente lo mismo que añadir capas en un MLP

# Apilando redes recurrentes.ipynb

# Ventajas e inconvenientes de estas arquitecturas

## Ventajas

- Posibilidad de procesar entradas de longitud variable
- El tamaño del modelo no crece con el tamaño de la entrada
- La inferencia tiene en cuenta la información histórica
- Los pesos se comparten a lo largo del tiempo

## Inconvenientes

- Procesamiento es mucho más lento
- Es difícil acceder a información lejana

# Implementación en PyTorch (I)

En RNN se devuelve la secuencia de valores y el último estado oculto:

```
rnn = nn.RNN(input_size=..., hidden_size=128)
o, h = rnn(i)

gru = nn.GRU(input_size=..., hidden_size=128)
o, h = gru(i)

lstm = nn.LSTM(input_size=..., hidden_size=128)
o, (h, c) = lstm(i)
```

- `o` → Secuencia para cada paso en la forma `(batch, seq_len, hidden_size)`
- `h` (o `(h , c)`) → Estado oculto final

En arquitecturas simples unidireccionales, `o[:,-1,:]` ≡ `h`

- Aun así es buena práctica extraer explícitamente la salida que necesitamos

# Implementación en PyTorch (II)

Hay tres capas RNN integradas en PyTorch:

- `torch.nn.SimpleRNN`: Unidad recurrente simple, donde la salida de la unidad en $t-1$ es la entrada de la unidad en $t$
- `torch.nn.LSTM` , Propuesta por Hochreiter y Schmidhuber[3] en 1997
- `torch.nn.GRU`: Propuesta por primera vez por Cho et al.[4] en 2014.

La primeras implementaciones de código abierto de LSTM y GRU fueron en 2015

[3] Artículo: *Long short-term memory*
[4] Artículo: *Learning phrase representations using RNN encoder-decoder for statistical machine translation*

# Implementación de RNN

Lo más habitual es definir una subclase de `torch.nn.Module`

- Generalmente los modelos suelen ser más complejos que una única capa RNN

```
import torch.nn as nn

class RNNModel(nn.Module):
    def __init__(self, vocab_size=1000, embedding_dim=64, *args, **kwargs):
        super().__init__(*args,)

        self.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim)
        self.rnn1 = nn.RNN(input_size=embedding_dim, hidden_size=256, batch_first=True)
        self.fc = nn.Linear(in_features=256, out_features=10)

    def forward(self, x):       # x: (batch, seq_len)
        x = self.embedding(x)   # x: (batch, seq_len, embedding_dim)
        x, _ = self.rnn(x)      # x: (batch, seq_len, 256)
        x = x[:, -1, :]         # x: (batch, 128)
        x = self.fc(x)          # x: (batch, 10)
        return x
```

# Una nota acerca de las secuencias

En PyTorch las RNN trabajan con ***inputs* de longitud fija** $\rightarrow N \times T \times D$

- Esto es, $N$ ejemplos de longitud de secuencia $T$ con dimensión de entrada $D$
- Esta configuración permite operaciones en *batch*, lo que acelera el entrenamiento

Sin embargo, al fijar $T$$ hay que tomar una decisión

- $T$ **demasiado largo**: Secuencias cortas se rellenan (*padding*) con ceros
  - *Son muy raras*[5] por lo que conllevan derroche de espacio y de tiempo
- $T$ **demasiado corto**: Secuencias largas se truncan y podemos perder información

PyTorch permite secuencias de longitud variable, pero se usele preferir $T$ fijo

- `pack_padded_sequence` y `pad_packed_sequence`: Implica lógica adicional y el manejo explícito de la longitud de cada secuencia

[5] La larga estela.

# Más allá de las unidades recurrentes simples

# Sobre las dependencias a largo plazo

Supongamos la frase *«Las **nubes** están en el ...»*

- La respuesta más obvia sería **cielo**
- Casi no necesitamos más contexto que la palabra **nubes**

Ahora supongamos la frase: *«He residido en **España** los últimos 10 años, durante los cuales he visitado gran parte del país y disfrutado de su gastronomía. Puedo hablar con bastante fluidez el ...»*

- La información relevante de la respuesta **español**, pero la información relevante está muy lejos para poder predecir

Las arquitecturas LSTM y GRU nos ayudan a ***mitigar*** este problema

- Implementan técnicas para aprender **qué** elementos **retener** y **cuáles olvidar**

# Long Short-Term Memory (LSTM)

Son un tipo especial de SRU capaz de aprender dependencias a largo plazo

- Diseñadas **expresamente** para el problema de las dependencias
- Tratan de entender *qué datos deberían ser recordados* y *qué datos pueden olvidarse*

Son un **recambio directo de las SRU**

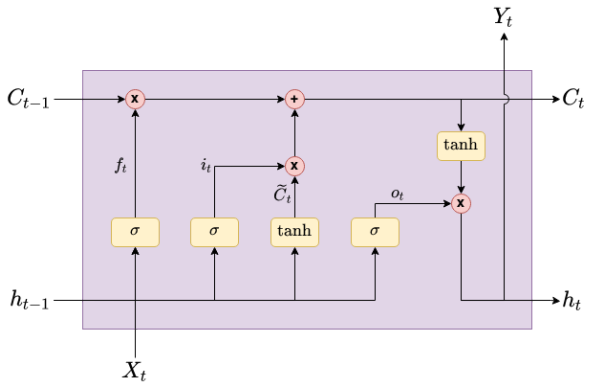


***Figura 18**.Esquema de la arquitectura de una unidad LSTM.*

# El estado de la unidad LSTM

La clave de las LSTM es la línea horizontal superior

- Es el **estado de la unidad** y la información fluye por ella, generalmente sin cambios

La unidad tiene la **capacidad de añadir o quitar información** al estado

- Esto se gestiona a través de las estructuras denominadas *puertas*
- Son una forma de dejar (o no) pasar información
- Están compuestas por unos pesos, una activación $\sigma$ y un producto punto a punto
  - La activación determina la cantidad de cada componente que "pasa"
  - Va de $0$ (no pasa nada) a $1$ (pasa totalmente)

# 1. ¿Qué información descartamos?

Comprueba $h_{t-1}$ y $X_t$ y devuelve $f_t \in (0, 1)$ para cada componente de $C_{t-1}$

- De $0$ (olvidar completamente) a $1$ (recordar completamente)

***Forget gate layer***

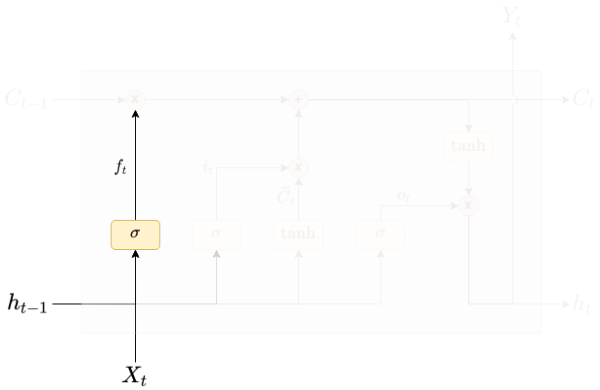


***Figura 18**. Forget gate.*

**Ejemplo**

El estado mantiene el género:

- Así usa los pronombres correctos
- Si viene un nuevo sujeto, probablemente querremos olvidar el anterior

$$f_t = \sigma(W_f[h_{t-1}, X_t] + b_f)$$

# 2. ¿A qué información prestamos atención?

Comprueba $h_{t-1}$ y $X_t$ y devuelve:

- Valores que vamos a actualizar (\textit{input gate layer)}
- Valores candidatos que \textit{podrían} se añadidos al estado

### *Input gate layer*

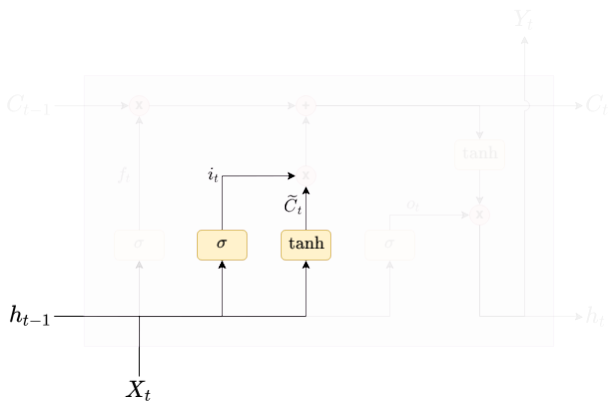


***Figura 19**. Input gate.*

### Sobre el ejemplo anterior

- Se actualizaría la información del género
- Se añadiría información si fuese necesario

$$i_t = \sigma(W_i[h_{t-1,}X_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C[h_{t-1,}X_t] + b_C)$$

# 3. Actualización del estado

Se actualiza el estado $C_{t-1}$ con la nueva información

- En los anteriores pasos **se decide** qué hacer con las entradas
- **Ahora** únicamente tenemos que **hacerlo**

### *Actualización del estado*

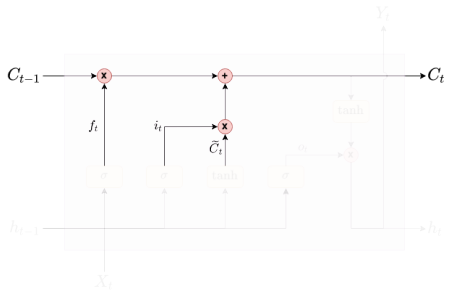


***Figura 20**. Actualización del estado.*

### Sobre el ejemplo anterior

- Aquí se eliminaría la información sobre el género del antiguo sujeto

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Este nuevo estado se le pasará a la neurona en la siguiente iteración

# 4. ¿Qué información damos de salida?

Decidimos la salida (filtrando el estado que estamos pasando):

- Capa sigmoide que decide qué partes del estado sacar
- Tangente del estado para transformar al intervalo $(-1, 1)$
- Producto de ambas para sólo sacar las partes decididas

### *Decisión de salida*

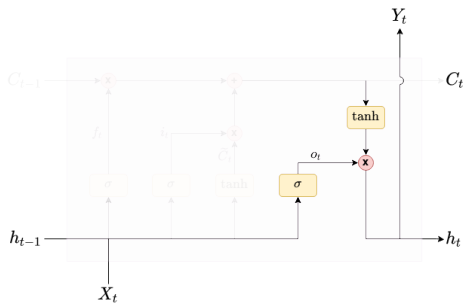


***Figura 21**. Decisión de salida.*

### Sobre el ejemplo anterior

- Da información relevante a:
  - ... género para construir un adjetivo
  - ... número para conjugar un verbo

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

# Unidades *Gated Recurrent Unit* (GRU) (I)

Se pueden considerar como una **versión simplificada de las** unidades **LSTM**

- Mismo objetivo que las LSTM $\rightarrow$ mitigar el problema de la memoria a largo plazo

Su rendimiento es muy similar al de las LSTM

- Empíricamente parece que GRU se comporta mejor con *datasets* pequeños
- Sin embargo, algunos autores recomiendan combinar ambas:
  - Capacidad de aprender asociaciones a largo plazo para la LSTM
  - Capacidad de aprender de patrones a corto plazo para la GRU
- ¿Cuál usar? $\rightarrow$ **Ensayo y error**

# Unidades *Gated Recurrent Unit* (GRU) (y II)

### *Unidad recurrente GRU*

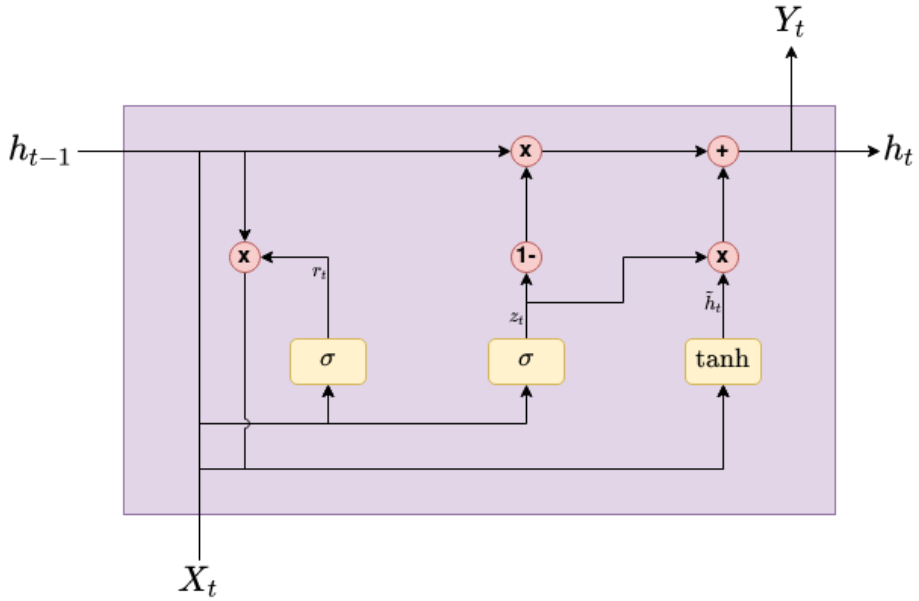


***Figura 22**. Esquema de unidad GRU.*

### Características

1. No existe entrada de estado
2. Dos puertas para gestionar entradas
    i. Salida anterior y entrada actual
    ii. Salida anterior y actual

$$z_t = \sigma(W_z[h_{t-1}, X_t] + b_z)$$
$$r_t = \sigma(W_r[h_{t-1}, X_t] + b_r)$$
$$\widetilde{h_t} = \tanh(W_h[r_t \odot h_{t-1}, x_t])$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h_t}$$

# Resumen de unidades recurrentes

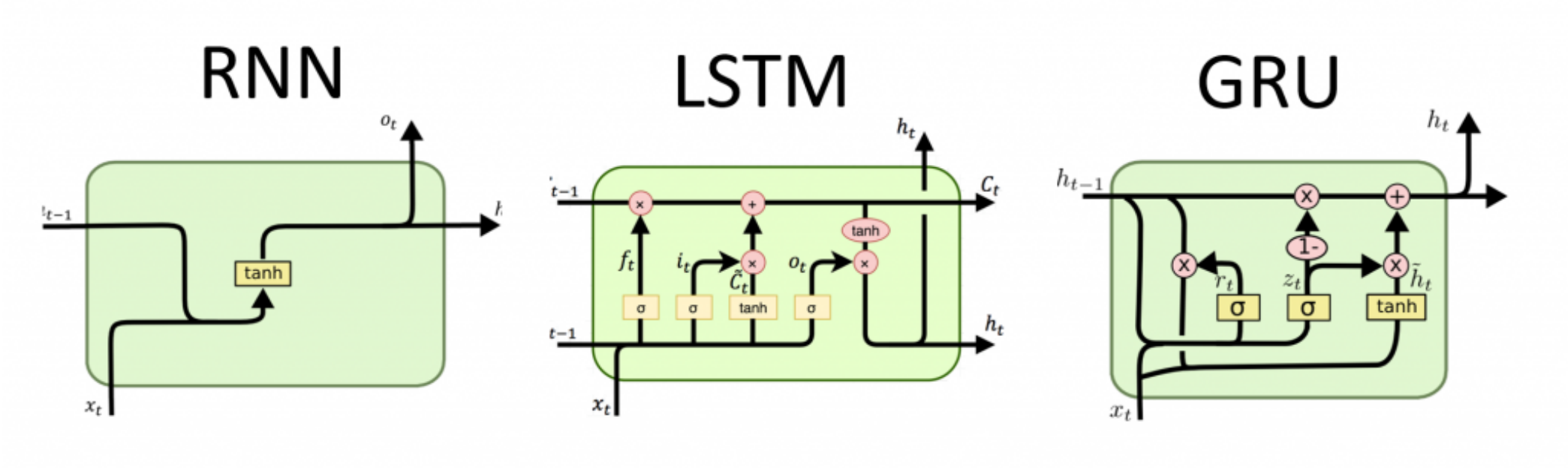


***Figura 22**. Comparativa de los diferentes modelos de redes recurrentes.*

# Predicción de casos diarios de COVID-19

# Arquitecturas de redes recurrentes

# Sobre los difentes tipos de problema

Las redes feed-forward tienen siempre la misma estructura

- Se presenta **un input**, se obtiene **un output**
- Prácticamente todos los problemas vistos hasta ahora son así
- Se conoce como ***one-to-one***

Ahora tenemos potencialmente **entradas y salidas como secuencias**

- **Entrada única, secuencia de salida**: P.ej. Subtitulado de imágenes
- **Secuencia de entrada, única salida**: P.ej. Clasificación de malware
- **Secuencia de entrada, secuencia de salida**: P.ej. Traducción automática o *name entity recognition*

Utilizaremos la notación $T_x$ y $T_y$ para la longitud de entrada y salida

# Arquitectura one-to-many ($T_x = 1, T_y > 1$)

**Generación de secuencias** a partir de una única entrada

- Aplicaciones: generación de texto, de tráfico de red, etiquetado de imágenes, ...

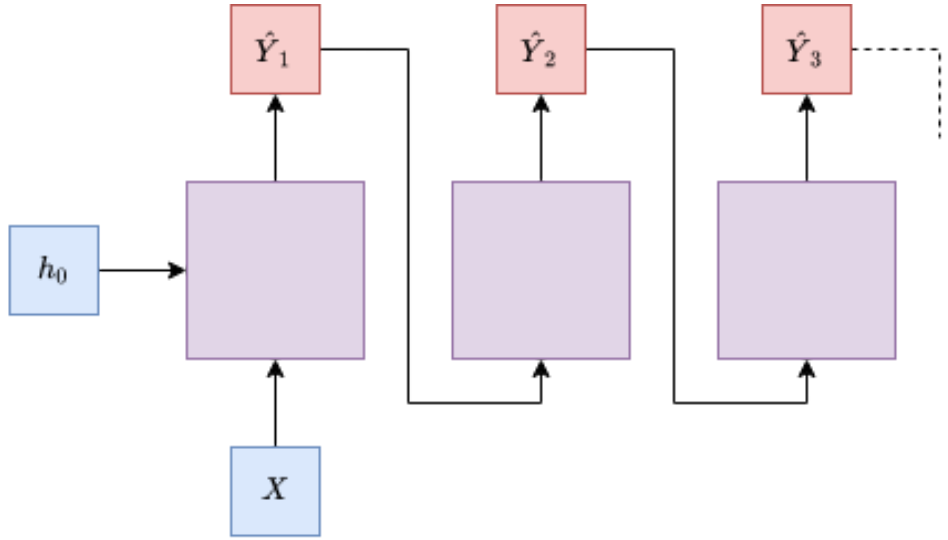


***Figura 10**. Ilustración de la arquitectura one-to-many.*

# Generación de música.ipynb

# Arquitectura many-to-one ($T_x > 1, T_y = 1$)

**Una única respuesta** para una secuencia de entrada (típico para clasificación)

Ejemplo - Análisis de malware de $2KiB$: In: $X_1, \ldots, X_{2048}$, Out: $Y_1, \ldots, Y_{2048}$

- Qué $Y_i$ cogemos? Tiene sentido el último, ya que así ha visto todo el ejecutable

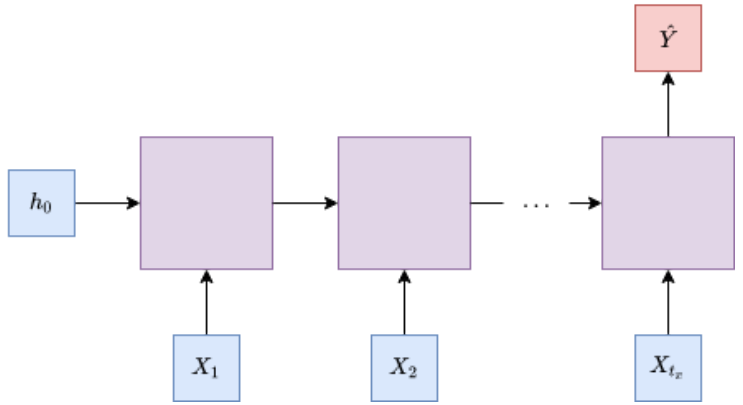


***Figura 11**. Ilustración de la arquitectura many-to-one.*

# Arquitectura many-to-one - Otra vuelta de tuerca

¿Por qué coger el último estado? ¡podríamos estar perdiendo información clave!

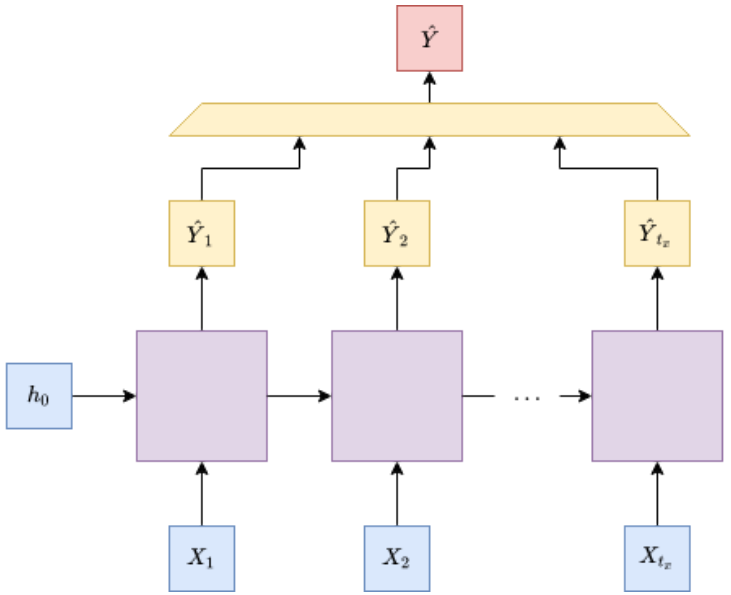


***Figura 12**. Ilustración de la arquitectura many-to-one usando max pooling para determinar la salida.*

# Arquitectura many-to-many ($T_x = T_y$)

**Dos secuencias** de entrada y salida, ambas **del mismo tamaño**

- Típicas de problemas de etiquetado gramatical y similares

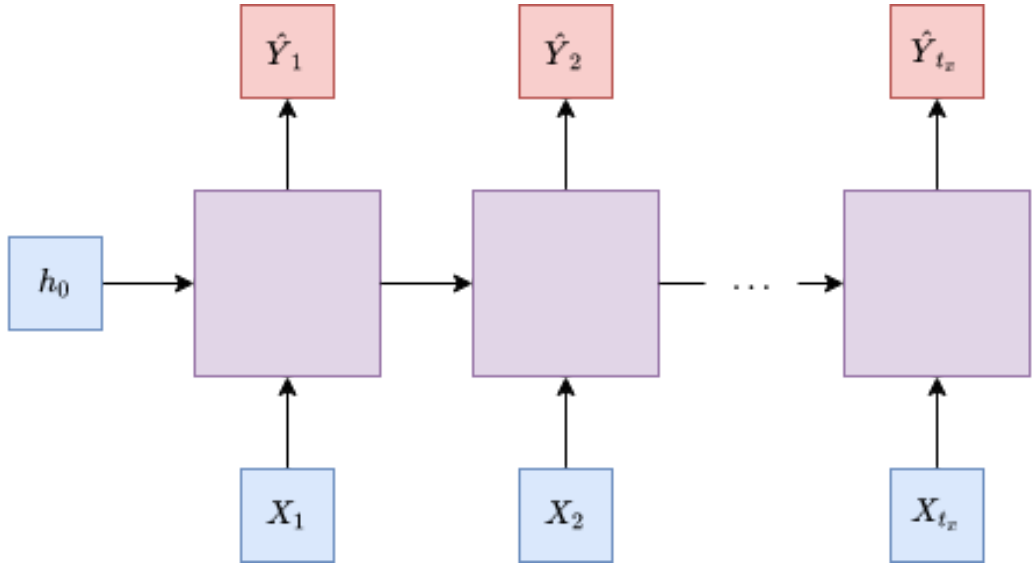


***Figura 13**. Ilustración de la arquitectura many-to-many con el mismo tamaño de entrada-salida.*

# Arquitectura many-to-many ($T_x \neq T_y$)

**Secuencia de entrada** que genera una de **salida** de, generalmente, **distinto tamaño**

- Típicas de problemas de traducción automática

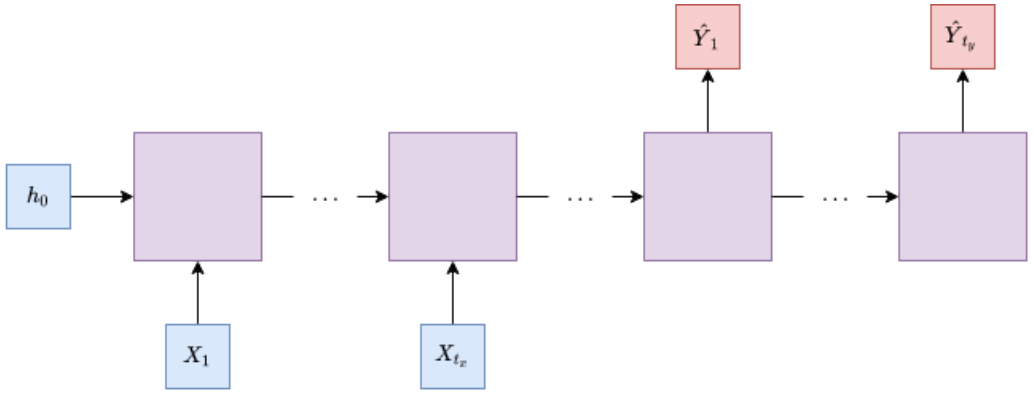


***Figura 14**. Ilustración de la arquitectura many-to-many con (potencial de tener) diferente tamaño de entrada-salida.*

# Tipos de arquitecturas de redes neuronales

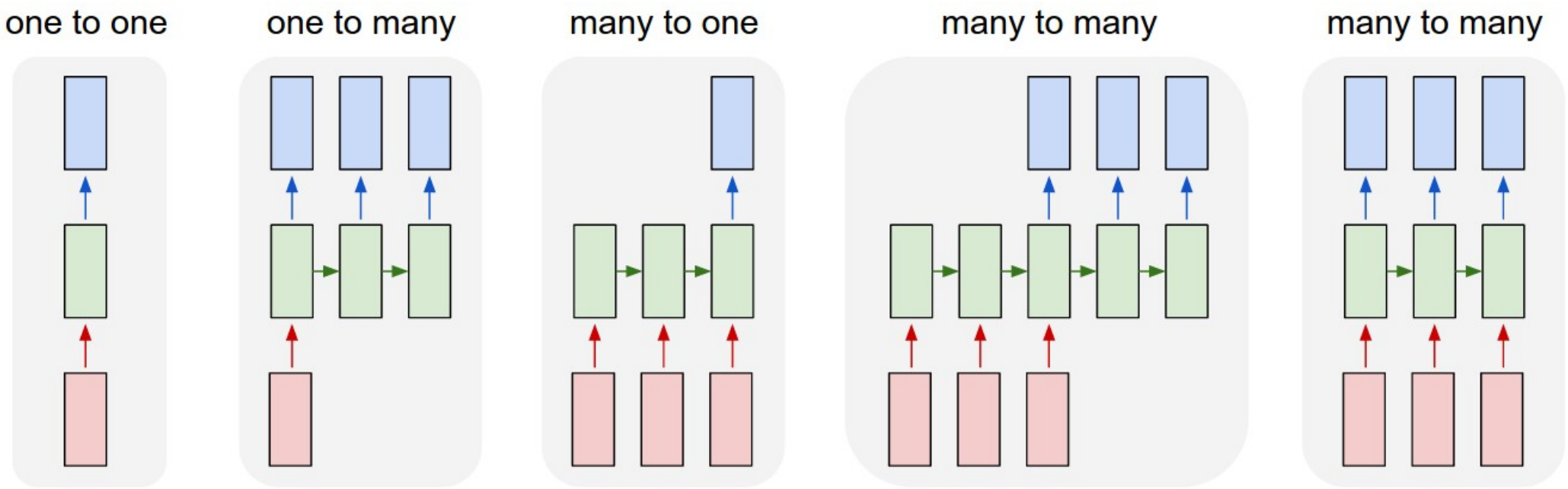


***Figura 15**. Diferentes problemas en redes neuronales recurrentes. Autor: Andrej Karpathy[6].*

[6] Extraído de la entrada del blog del autor The Unreasonable Effectiveness of Recurrent Neural Networks.

# ¿Se diferencian las RNN de las CNN? (I)

Ambas han impulsado el rendimiento de la inteligencia artificial en los últimos años

Las redes convolucionales son redes de tipo ***feed-forward***

- Utilizan capas de filtros y de *pooling*
- Los tamañas de entrada y salida son fijos
- Se utilizan habitualmente sobre datos espaciales (p.ej. imagenes)

Por otro lado, redes recurrentes **retroalimentan** los resultados a la red

- Son adecuadas para datos secuenciales (p.ej. texto o vídeo)
- El tamaño de la entrada y la salida resultante pueden variar
- Sus casos de uso incluyen el procesamiento del lenguaje natural, la detección de anomalías o la generación de secuencias

# ¿Se diferencian las RNN de las CNN? (II)

¿Qué pasa si queremos entender qué ocurre en las imágenes?

***Figura 16**. ¿Hacia dónde se mueven la cabeza y la pelota?.*

La secuencia de imágenes pasada le da contexto a la predicción

# ¿Se diferencian las RNN de las CNN? (III)

***Figura 16**.Los eventos pasados añaden información y contexto.*

Las RNN están diseñadas precisamente para recordar la información previa

# ¿Se diferencian las RNN de las CNN? (y IV)

Hemos aprendido que las redes recurrentes tienen en cuenta píxeles adyacentes

- ¿Y el mismo mecanismo para eventos adyacentes en el tiempo?
  - No es que no funcione, de hecho se usa bastante
  - De hecho es una muy buena forma de arrancar una prueba de concepto
  - Es un enfoque innecesariamente indirecto
    - Es usar un modelado espacial para captar un fenómeno temporal
    - Requiere mucho más esfuerzo y memoria para realizar la misma tarea

Ambas técnicas se complementan $\rightarrow$ Redes de convolución recurrentes (CRNN):

- Etiquetado de vídeos, reconocimiento de gestos, *sentiment analysis* en lenguas logográficas, ...

# Licencia



Puedes encontrar su código en el siguiente enlace:
https://github.com/blazaid/aprendizaje-profundo