Redes Neuronales Recurrentes

Prof. Wílmer Pereira

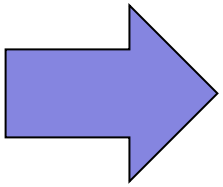
Análisis de texto

La comprensión del lenguaje natural es un tema mítico en IA que ha visto varios fracasos. Una nueva luz se perfila con las redes neuronales profundas aunque, por ahora, para dominios específicos con vocabulario y objetivos bien delimitados …

- En la década de los sesenta, en plena guerra fría, se pretendió realizar un traductor automático de ruso a inglés. Todo el esfuerzo y dinero terminó en resultados decepcionantes. Otro intento fue el proyecto japones de computadoras de 5ta generación cuyo lenguaje ensamblador estaba basado en PROLOG … se canceló sin avances …

- El reconocimiento de lenguaje natural requiere considerar al menos 5 fases:

  1. Fonética
  2. Lexicográfica
  3. Sintáctica
  4. Semántica
  5. Pragmática

  El esfuerzo conceptual y de cálculo es enorme, con dificultades para dar respuesta en tiempos razonables

- A pesar de contar con las redes neuronales profundas, actualmente sólo podemos aspirar a problemas de lenguaje natural reducidos como análisis de sentimientos, clasificación de documentos, identificación de autor (música o texto), responder preguntas (en contexto bien restringidos). Por supuesto, sin verdadero entendimiento del discurso, más bien con correspondencias, mediante estructuras estadísticas, del lenguaje escrito.

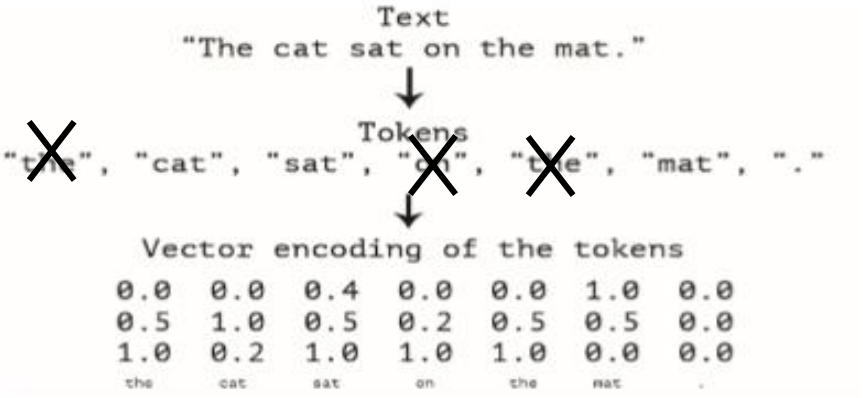
Deep learning y análisis de texto

- Como punto de partida, se debe vectorizar el texto transformándolo en tensores. Los insumos son palabras aisladas o **n-gramas** (secuencias de palabras) para darle contexto a las palabras y resolver la polisemia (banco, bolsa, botín, botón, cabo, capital, …). Por ejemplo:

  El gato sube despacio la escalera
  Con el gato se sube despacio el coche

  ⇒ Con 4-gramas, se resuelve el significado semántico de gato

- En consecuencia el texto debe estar dividido en palabras o n-gramas que llamaremos *tokens*. La idea es asociar más de un número a cada *token*, es decir, un vector (tensor 1D). Esto se logra mediante el *word embedding*. También existe otro mecanismo conocido como *one-hot encoding*



- De ahora en adelante sólo consideraremos palabras o *tokens* … Estas se almacenan en un repositorio, que se comporta como un conjunto, conocido como ***bag-of-words***.

- Además, en algunas circunstancias, puede ser conveniente eliminar las palabras que no aportan significado semántico: artículos, preposiciones, conjunciones, … Este filtrado se realiza gracias a los ***stop words*** que dependen de cada idioma. En ocasiones no se considera este filtrado …

Palabras en el bag-of-word

- Además para reducir el tamaño del *bag-of-word*, se podrían eliminar, por ejemplo, sufijos, prefijos, conjugaciones de verbos, etc … Para ello se propone almacenar sólo un *token* por palabras relacionadas. Este proceso se puede realizar de dos maneras:

  - ***Lemmatization***
  - ***Stemming***

- La *lemmatization* es el proceso donde dada la forma flexionada, retorna la palabra que etimológicamente la representa. Por ejemplo, para la conjugación de un verbo se usa el infinitivo o para un adjetivo el masculino singular, … En consecuencia, la palabra que se incluye en el *bag-of-word* pertenece al lenguaje. Así el lema de canto, canté o cantaríamos, … sería cantar.

- Hay dos maneras de lematizar: morfológicamente y sintácticamente. Visto morfológicamente la palabra ama tendría dos lemas: el sustantivo ama y el verbo amar. En cambio sintácticamente se puede diferenciar identificando la estructura gramatical de la frase. Por ejemplo, en la frase: "La ama de llaves abrió la puerta", el lema es ama y no amar …

- El *stemming*, por el contrario, no asegura que la palabra del *bag-of-word* pertenezca al lenguaje porque recuperar la raíz común a todas sus declinaciones. Por ejemplo, para canto, canté o cantaríamos sería el token cant … Otra diferencia, con respecto a la *lemmatization*, es que el *token* se obtiene a partir de un proceso lexicográfico …

- También es común utilizar el TF-IDF que es un mecanismo para calcular la frecuencia de aparición de los lemas. Esto ayuda a determinar la relevancia y por ello a construir el *stop word* con los *tokens* o lemas menos importante y el *bag-of-words* con los más importantes..

TF-IDF

- El objetivo es determinar cuan relevante es una palabra o término para un documento dentro de una colección. TF-IDF aumenta proporcionalmente al número de veces que una palabra aparece en el documento, pero es compensada por la frecuencia de la palabra en la colección de documentos.

- TF significa frecuencia del término e IDF frecuencia del inversa del documento. La idea es bajar el peso de las palabras que se usan muy frecuentemente y aumentar el peso de las palabras poco comunes que ocurren solamente en ese documento:

$$TF(término, documento) = \frac{n_i}{\sum_{k=1}^{W} n_k}$$

$$IDF(término) = \log \frac{N}{n_t}$$

$$TF - IDF(término, documento) = TF(término, documento) \times IDF(término)$$

- La motivación es determinar de una colección de documentos cual es el más relevante a la consulta. Los cálculos mostrados no son la única manera de obtener la relevancia del término para el documento.

One-hot encoding

- Es un proceso que construye una matriz binaria dispersa, transformando cada valor categórico de un atributo en una columna de la matriz.

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

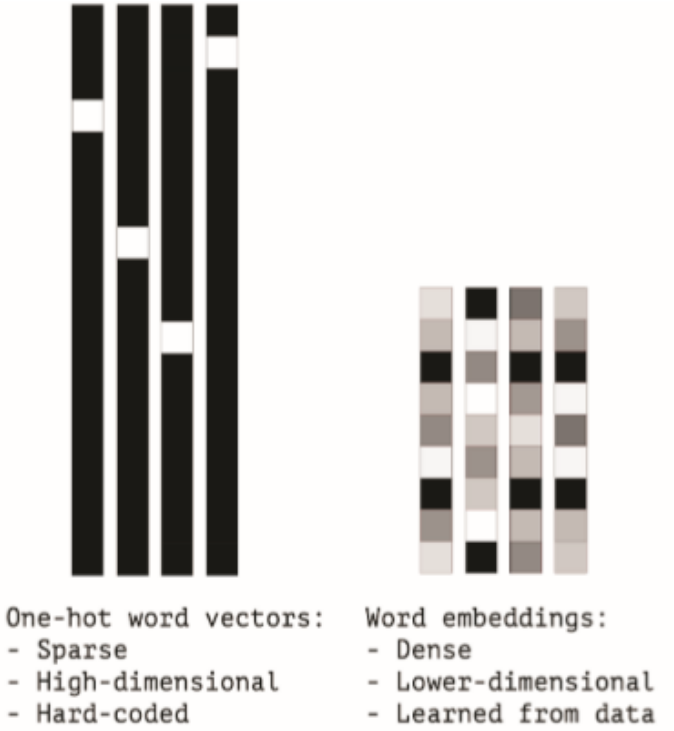
Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

Eliminar atributos categóricos para poder procesar numéricamente

- En Keras se puede realizar muy rápidamente gracias a la clase `Tokenizer`.

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# We create a tokenizer, configured to only take
# into account the top-1000 most common on words
tokenizer = Tokenizer(num_words=1000)
# This builds the word index
tokenizer.fit_on_texts(samples)

# This turns strings into lists of integer indices.
sequences = tokenizer.texts_to_sequences(samples)

# You could also directly get the one-hot binary representations.
# Note that other vectorization modes than one-hot encoding are supported!
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# This is how you can recover the word index that was computed
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

Word embeddings



En lugar de asociar una matriz binaria dispersa, que desperdicia espacio, se define un vector de menor dimensionalidad, aunque generalmente son entre 100 a 1000 valores … baja con respecto a *one-hot-encoding* …

Hay dos maneras de obtener los vectores por palabra:

1. Usar los textos de entrada para obtener los vectores de cada palabra mediante entrenamiento con una red neuronal.
2. Cargar un espacio de palabras con los vectores ya definidos, pre-entrenado con grandes bases de datos de palabras.

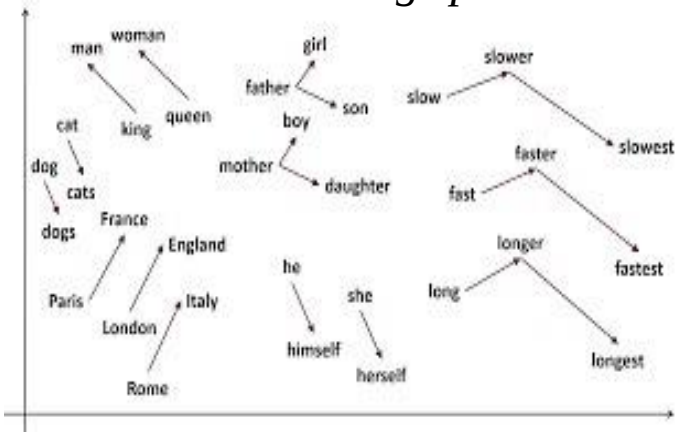
- Los dos espacios *n*-dimensionales con vectores de palabras pre-entrenada para lograr, como en las CNN, transferencia de aprendizaje. Los repositorios más utilizados son:

Word2Vec	Desarrollado por Google en 2013 (tiene cientos de billones de palabras). Cada palabra es un vector o tensor 1D de 300 valores obtenidos del proyecto políglota …
GloVe	Proyecto de Stanford (2014) con funcionalidades similares a Word2Vec.

Intuición de word embedding

- Cada palabra se sitúa en un espacio *n*-dimensional y se agrupan por similaridad. Además dado un vector de transformación se pueden acercar conceptos

embedding space



- ¿Existe un *embedding space* para cualquier tipo de discurso a procesar? Quizás sea posible pero sólo se ha logrado para dominios específicos. Uno de los problemas es la polisemia … Animación …

- Keras ofrece una capa para aprender *embedding space* … la clase se invoca como `Embedding(`*`MAXWORDS`*`,`*`DIMVECTOR`*`)` … inicialmente define un diccionario interno de palabras con vectores de tamaño *`DIMVECTOR`* partiendo con valores aleatorios

`Embedding` toma un tensor 2D (*muestras*, *indicesPalabra*) como entrada y devuelve un tensor 3D (*muestras*, *indicesPalabra*, *`DIMVECTOR`*) … Se entrena con sólo las palabras de los textos de entrada o usando un espacio de palabras pre-entrenado (Word2Vec o GloVE).

- Después de este proceso el tensor 3D sirve de entrada para una red neuronal recurrente o una red neuronal convolucional 1D.

Vectores sólo con los datos de entrada: IMDB

- Si se toman sólo las primeras 20 palabras de la crítica de un conjunto posible de 10,000 palabras. Se buscan vectores de tamaño 8-dimensionales y se obtiene un *accuracy* del 76% ☹ no mejora el 88% de la primera versión …

```
from keras.datasets import imdb
from keras import preprocessing

# Number of words to consider as features
max_features = 10000
# Cut texts after this number of words
# (among top max_features most common words)
maxlen = 20

# Load the data as lists of integers.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# This turns our lists of integers
# into a 2D integer tensor of shape `(samples, maxlen)`
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

model = Sequential()
# We specify the maximum input length to our Embedding layer
# so we can later flatten the embedded inputs
model.add(Embedding(10000, 8, input_length=maxlen))
# After the Embedding layer,
# our activations have shape `(samples, maxlen, 8)`.

# We flatten the 3D tensor of embeddings
# into a 2D tensor of shape `(samples, maxlen * 8)`
model.add(Flatten())

# We add the classifier on top
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

Vocabulario de 10,000 palabras, 8-dimensional y 20 palabras por reseña …

Vectores usando GloVE

- A diferencia del ejemplo anterior, se usa un *embedding space* pre-entrenado, en un espacio 100-dimensional con 400,000 palabras (`glove.6B.100d.txt`).

```
glove_dir = '/Users/fchollet/Downloads/glove.6B'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector
```

10,000 palabras

Video sobre `embedding_matrix`

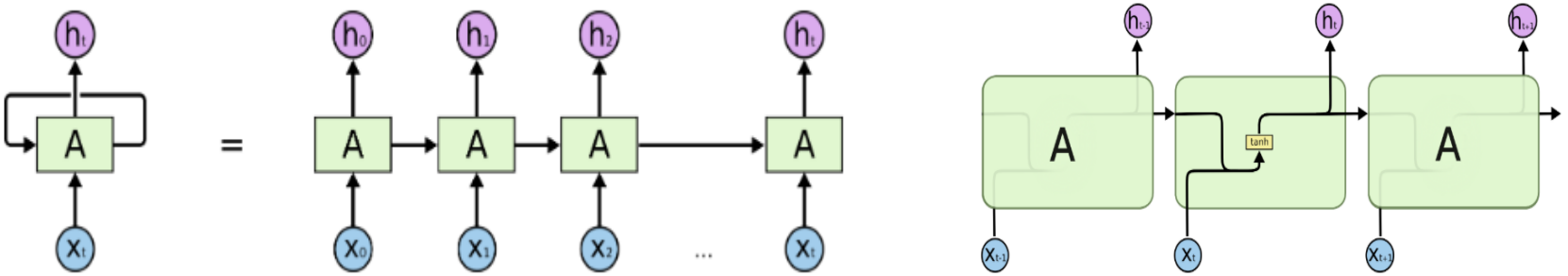
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

- Se congela la capa `embedding_matrix` pre-entrenada y el *accuracy* mejora en un 82%, pero sigue sin llegar al 88% con la red simple … ☹ … falta la historia …

Redes neuronales recurrentes

Las mayoría de redes neuronales son *feedforward*. Al incluir lazos de retroalimentación, se recupera el valor de la salida para mantener el estado de la información pasada. Esto es imprescindible cuando se procesan series de tiempo (finanzas, meteorología, ...)



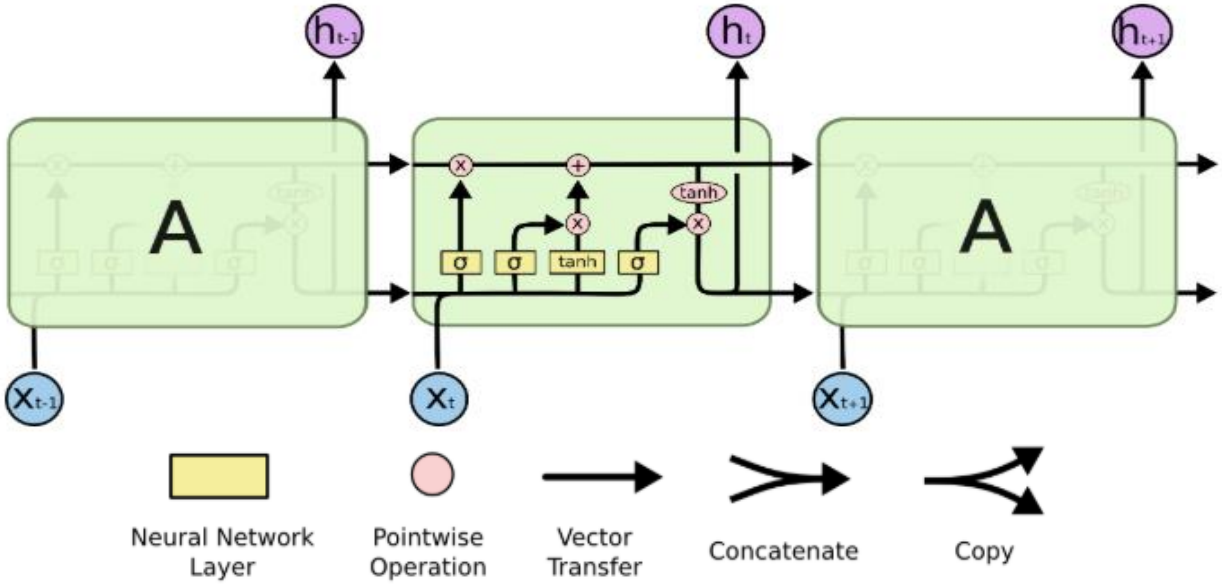
- Estas redes (RNN) son ideales en el procesamiento de texto, series de tiempo, …

  … pero …

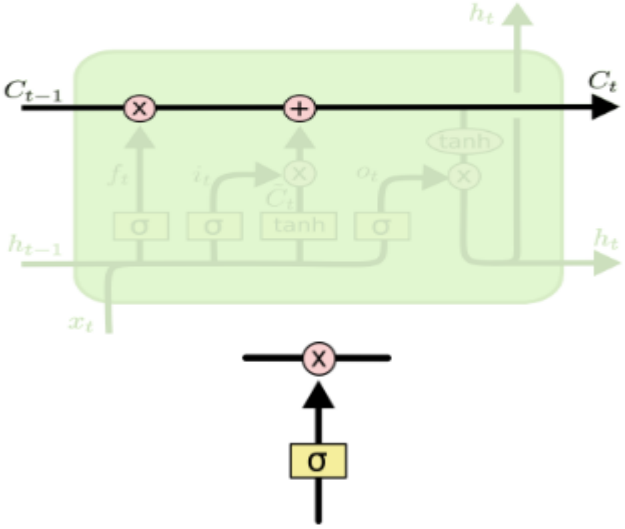
  … tienen problemas de desvanecimiento de gradiente …

- Como la dependencia a largo plazo se diluye, se desarrolló una RNN que mantiene por más tiempo el valor de los estados precedentes … Estas redes se conocen como LSTM (*Long Short Term Memory Network*) a veces mantienen el estado de largo plazo y a veces lo olvidan para ser substituido por el estado más reciente … Fue propuesta por Hochreiter&Schmidhuber (1997).

Principios de LSTM



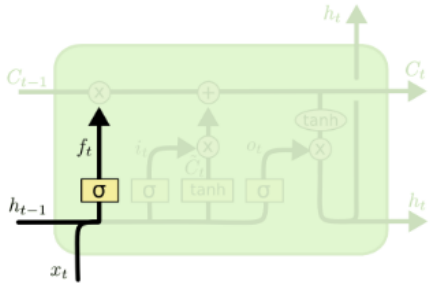
LSTM amplía la versatilidad de la neurona con la aplicación de varias funciones de activación sobre la entrada (parte inferior). La parte superior es el procesamiento de la salida de la neurona en el estado anterior



La parte superior mantiene el pasado a lo largo del tiempo durante el entrenamiento de esa neurona. Las operaciones x y + son las que regulan que tanto se olvidará o recordará el estado anterior

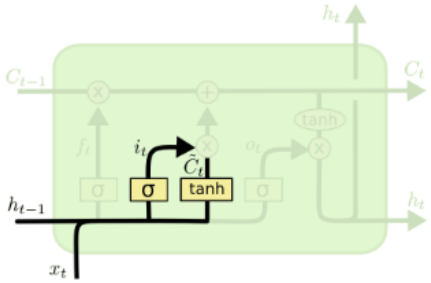
Las funciones de activación que se aplican son sigmoide y tangente hiperbólica. La sigmoide, en particular, retorna una salida binaria por lo que para LSTM representa una **puerta** de activación o desactivación de la entrada que recibe

Puertas y transferencia de información



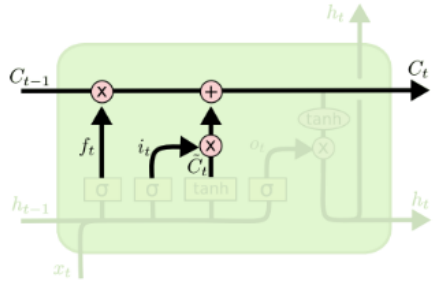
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Esta primera fase representa la **puerta de olvido**, ya que, a partir del estado anterior y la entrada actual, decide que tanto efecto tendrá sobre la línea del pasado de la red.



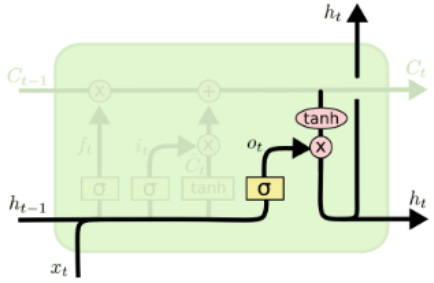
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Esta fase pretende transferir a la línea del pasado el efecto de la entrada, mediado por lo que se debe olvidar. El resultado irá a la línea del tiempo de la red.



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

Dado lo que se olvidó y lo que se aprendió … se modifica la línea del tiempo. Esto representa la influencia de la información a corto plazo sobre la información a largo plazo

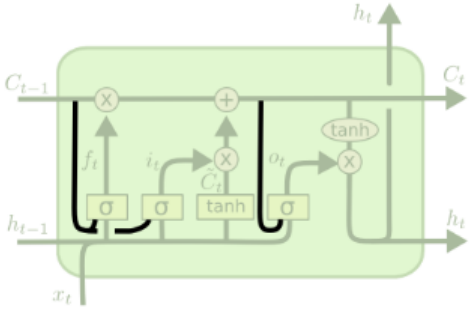


$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Finalmente se combina la información de lo que se olvidó de la entrada con el pasado de la red y se conforma la salida efectiva de esa neurona. La salida, a su vez, se combina de la entrada en el próximo estado.

Variantes LSTM

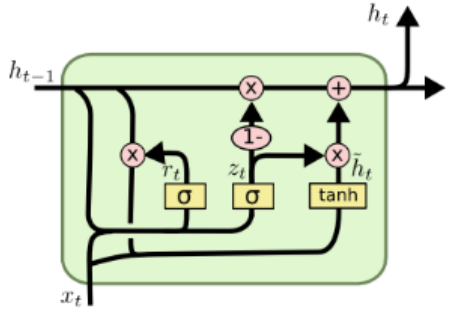
- Existen muchas modificaciones al diseño original, unas propuestas por el mismo equipo de Schmidhuber y otras de diferentes autores …



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Esta versión agrega **conexiones rendija**, es decir, que se permite a cada modificación sobre la entrada, ver el pasado o la línea de tiempo de la neurona …

- Una variante más conocida es la propuesta por Cho et al (2014) conocida como GRU (*Gated Recurrent Unit*)



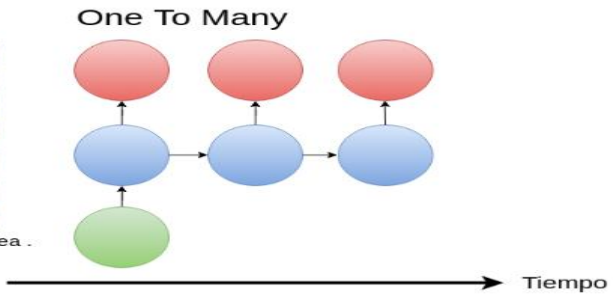
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

GRU opera primero la influencia del pasado sobre las operaciones de la entrada actual. Este modelo es más simple que el LSTM estándar y ha ganado mucha popularidad

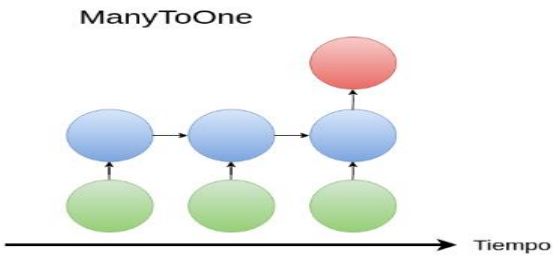
- ¿Cuál es la mejor arquitectura? Según un artículo de Josefowicz et al (2015) donde comparó decenas de miles de RNN, constató que todas, en general, se comportan más o menos igual, con ciertas diferencias entre ellas para ciertos tipos de problemas …

Arquitecturas RNN

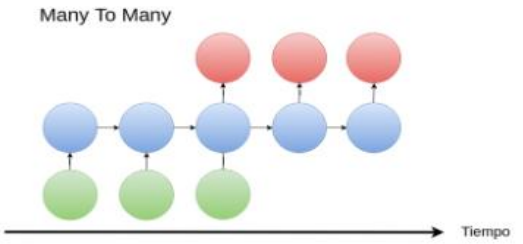
a group of giraffes standing in a grassy area .
Fuente: https://www.analyticsvidhya.com



Dada una imagen obtener un texto que la describa …



Dado una secuencia de texto clasificarlos como en el aná-lisis de sentimiento



Traducción automática entre diferentes idiomas …

Video sencillo sobre RNN y otro video más

RNN con Keras

- Existen muchos tipos de redes neuronales recurrentes. La más simple, conocida como `SimpleRNNLayer`, puede configurarse para que retorne una sóla salida al final de la recurrencia o una salida por cada serie de tiempo o entrada

```
>>> from keras.models import Sequential
>>> from keras.layers import Embedding, SimpleRNN
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32))
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_22 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_10 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_23 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_11 (SimpleRNN)     (None, None, 32)          2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

```
>>> model = Sequential()
>>> model.add(Embedding(10000, 32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32))  # This last layer only returns the last outputs.
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_24 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_12 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_13 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_14 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_15 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

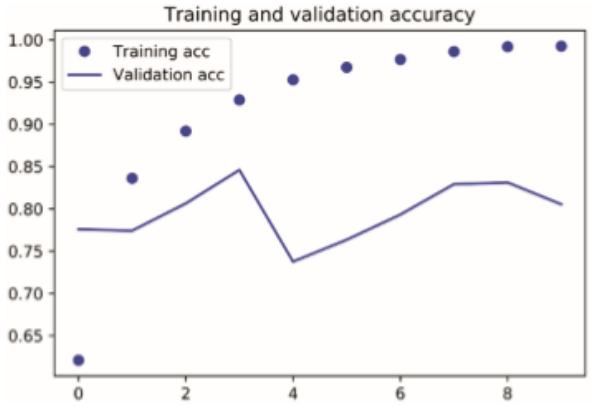
También son posible varias capas recurrentes consecutivas

IMBD (reseñas de cine) con RNN

- Con redes neuronales convencionales (16/16/1) se obtuvo un *accuracy* del 88% … veamos si mejora con las RNN …

```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Si se muestra el comportamiento durante el entrenamiento y se calcula el *accuracy* con los datos de prueba



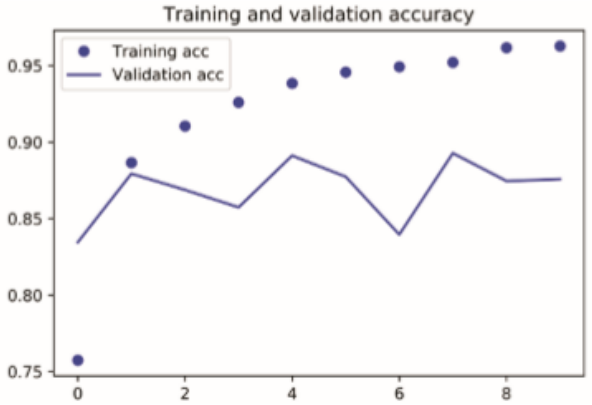
- Ahora con una RNN (32/1) se obtuvo un *accuracy* del 85% y desafortunadamente con *overfitting* … las razones pueden ser por tratarse de una red neuronal más pequeña y/o, la más probable, que este tipo de RNN simple no se adecúa a texto …

IMDB con LSTM

- Afortunadamente son LSTM si se supera el 88% de *accuracy* de la red simple !!!

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

Es sólo una capa 32/1, sin usar capas pre-entrenadas …

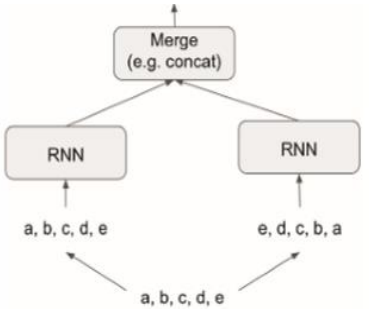


- A pesar de continuar el *overfitting* en pocas épocas, con LSTM si se logra un 89% y aún falta sin incluir una red pre-entrenada de GloVe o Word2Vec … También habría que corregir el *overfitting* con *dropout* o regularización …

RNN bidireccional

- La idea de este tipo de red neuronal consiste en evaluar la secuencia de datos cronológica y anticronológicamente y mezclarlas… Esto permitirá ver la secuencia desde dos perspectivas aunque el tiempo naturalmente no lo percibimos de esta manera … Intentemos con IMDB …

```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=10, batch_size=128, validation_split=0.2)
```



- La opción bidireccional crea automáticamente dos secuencias: cronológica y anticronológica y las procesa separadamente … Mejora ligeramente a 90% que es mejor que el obtenido en la última versión de IMDB … aunque continúa con el *overfitting* …
- En cambio, con la base de datos de temperaturas, el error no disminuye significativamente …

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Bidirectional(
    layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

No parece ofrecer mejoras porque, en realidad las RNN bidireccionales se adecúan al tratamiento de lenguaje natural y sólo a algunos problemas particulares de secuencia de tiempo …

Resumen final de IMDB

La referencia es con un red neuronal superficial se obtuvo un *accuracy* del 88%

Embedding 8-dimensional, sin pre-entrenamiento	76%
Embedding pre-entrenado (glove) 100-dimensional	82%
RNN simple (32/1) sin *embedding*	85%
LSTM (32/1) sin *embedding*	89%
RNN bidireccional (32/1) y *embedding* no pre-entrenado	90%

- Tanto en los ejemplos de la base de datos de temperaturas como en IMDB no se consideraron muchas otras variantes que pueden mejorar el desempeño:

    1. Ajustar la tasa de aprendizaje de los optimizadores
    2. Intentar GRU y/o LSTM
    3. Empilar capas densas antes de la última capa de regresión
    4. Probar regularización en lugar de *dropout*.

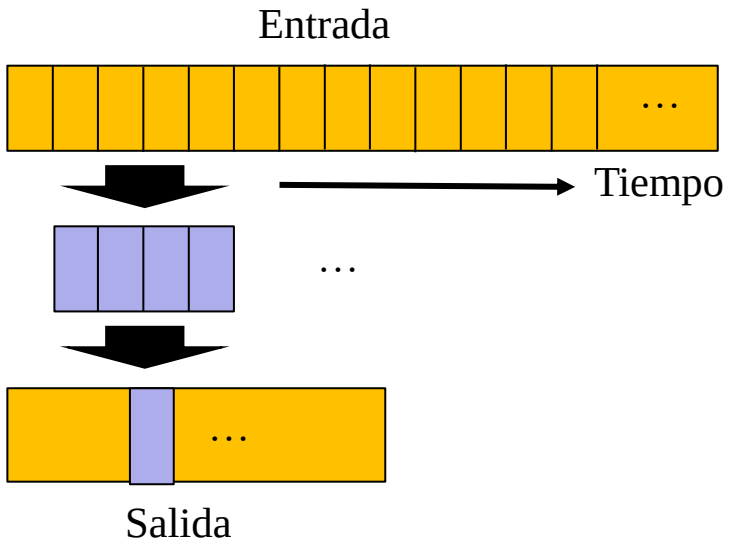
Es claro que *deep learning* tiene un fuerte componente de arte más que ciencia al momento de entonar las soluciones

Procesamiento secuencias: Conv1D

El proceso de convolución de texto (1D) es análogo a la convolución para imágenes (2D).
La diferencia radica en que los filtros para imágenes son 2D y para texto son 1D.
La gran ventaja, con respecto a LSTM o GRU, requieren menos tiempo de ejecución.

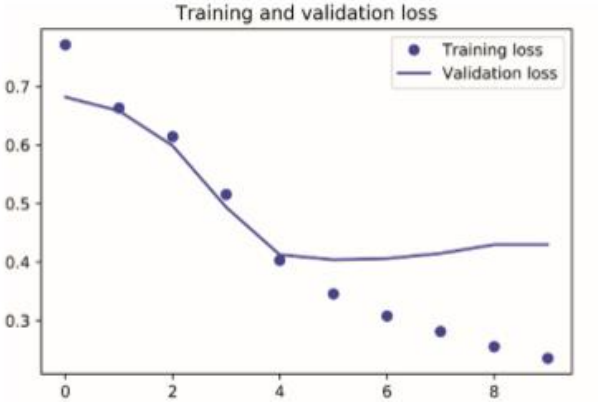
- El ancho del filtro lineal determina que tanto se explora del texto, lo que sugiere un revisión de la secuencia de caracteres en el pasado y futuro. No obstante no discrimina explícitamente entre pasado y futuro pero es invariante a la traslación en una dimensión
- Al igual que para las imágenes, se usan operaciones de *max-pooling* o *avg-pooling* para reducir la dimensionalidad de las salidas y compactar la información. También se usa una capa `GlobalMaxPooling1D` que equivale al `Flatten`. Tratemos, de nuevo, con IMDB …

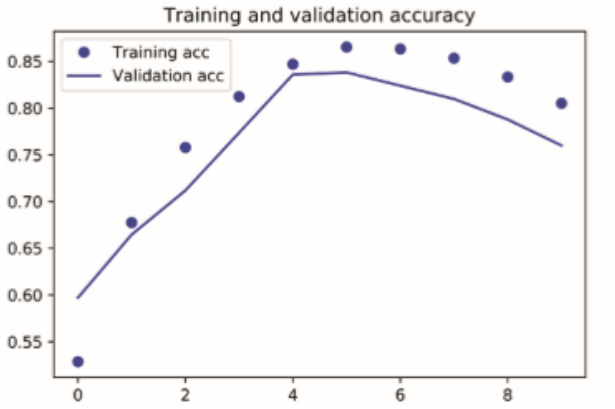
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```



Conv1D

- Aunque el *accuracy* es un poco menor que con la utilización de LSTM, esta red `Conv1D` se ejecuta mucho más rápido tanto en CPU como en GPU … sin embargo hay *overfitting* más allá de la 4ta época.






```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
cnn_history = model.fit_generator(train_gen,
                                  steps_per_epoch=500,
                                  epochs=20,
                                  validation_data=val_gen,
                                  validation_steps=val_steps)
```

Para la BD de temperaturas, `mae` es en promedio de 0.45, lo cual es mucho peor que cualquiera de las opciones anteriores que se usaron para Jena …

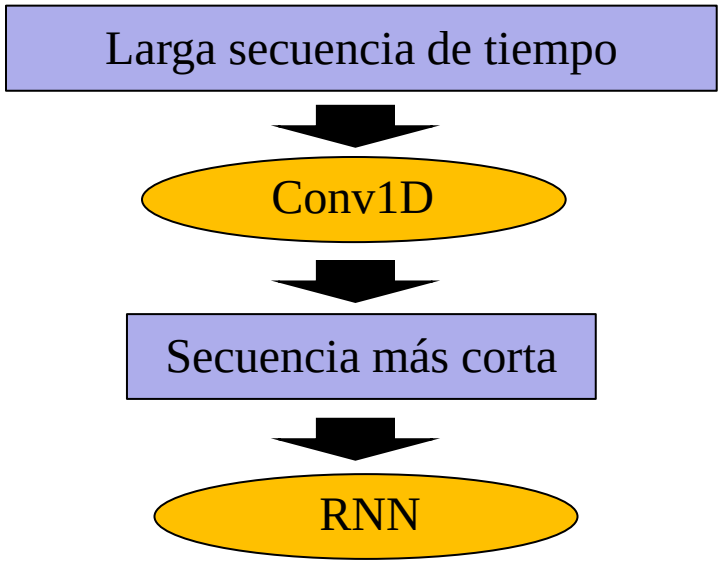
… la razón es …

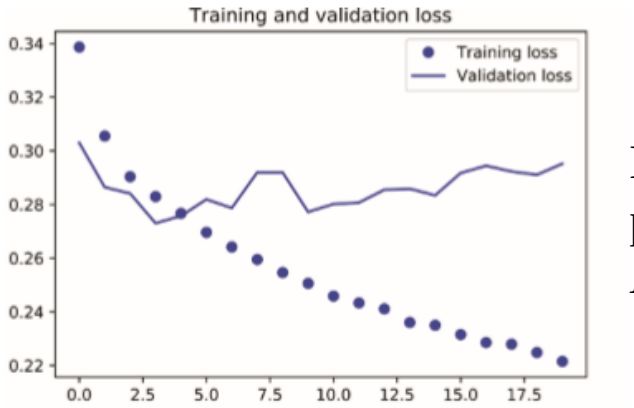
Como se mencionó al comienzo, `Conv1` no maneja bien la temporalidad … No ocurrió así con IMDB porque las reseñas son independientes en el tiempo…

Conv1D + RNN

- Con `Conv1D` se gana en rapidez si la secuencia de tiempo es muy larga porque reduce las salidas (`max-pooling`). El manejo de tiempo por supuesto se logra con las RNN (LSTM o GRU). Se intentará la mezcla con la BD de temperaturas de Jena …

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```





No es tan bueno comparado con GRU+ *dropout* pero la respuesta es significativamente más rápida. Además hay *overfitting* …

RNN simple y CNN+RNN

- En primer lugar una explicación alternativa en inglés de las RNN (1 hora …)
- Un buen ejemplo para RNN son las series de tiempo de las manchas solares … En este video (13 minutos en inglés) se muestra la programación básica en Keras y luego el ejemplo de las manchas solares
- Una aplicación para video con CNN+RNN en Keras en un video (22 minutos en inglés)
- Una alternativa más rápida para las RNN las redes neuronales transformacionales que no son cubiertas en esta presentación pero si en el siguiente video (13 minutos en inglés). No tienen ejemplos de código sólo teoría …
- En *Deep Learning* están aún las redes neuronales generativas como son las *Variational Autoencoders* y las GAN (*Generative Adversarial Network*) que también son de gran interés …