Taller de Aprendizaje Automático

Redes neuronales recurrentes

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

1. Redes neuronales recurrentes
2. Entrenamiento: propagación hacia atrás en del tiempo
3. Implementación de RNN simple
4. Manejando secuencias largas

Redes neuronales recurrentes (RNN)

- Tipo de redes pensadas para trabajar con datos secuenciales o series temporales, e.g.:
  - Precios de acciones en la bolsa
  - Trayectorias de objetos en vehículos autónomos
- Hasta ahora hemos trabajado con entradas de tamaño fijo
- Las RNNs permiten manejar secuencias de largo arbitrario, e.g.: audio, texto, video
- Muy útiles para analizar series de tiempo
- Especialmente útiles para problemas de procesamiento de lenguaje natural, e.g.: traducción automática, *speech to text*.

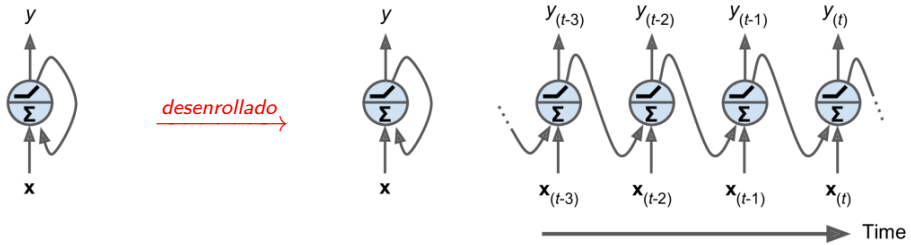
Agenda

- Conceptos básicos
- Cómo entrenarlas mediante propagación hacia atrás en el tiempo
- Cómo utilizarlas para hacer predicción en series temporales
- Analizaremos las dos dificultades mayores y cómo mitigarlas:
  - Gradientes inestables (desvanecimiento y explosión)
  - Memoria de corto plazo limitada y dos estrategias para ampliarla: celdas LSTM y GRU
- Otras técnicas para trabajar con datos secuenciales o series temporales

Una neurona recurrente

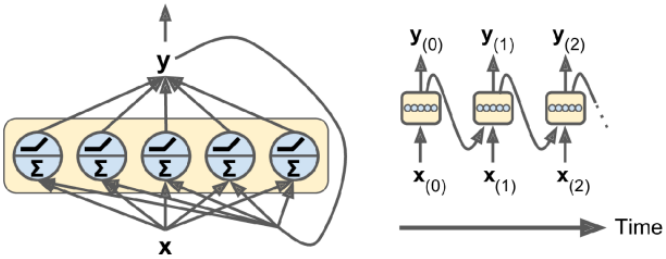
En cada paso temporal t la neurona recibe

- La entrada (escalar o vector) en el instante actual $\mathbf{x}_{(t)}$
- La salida del instante anterior $y_{(t-1)}$



Una capa de neuronas recurrentes

- $\mathbf{y}_{(t)} = \Phi\left(\mathbf{W}_x^\intercal\, \mathbf{x}_{(t)} + \mathbf{W}_y^\intercal\, \mathbf{y}_{(t-1)} + \mathbf{b}\right)$, con $\Phi(\cdot)$ función de activación



- Mini-batch $\mathbf{Y}_{(t)} = \Phi\left(\mathbf{X}_{(t)}\, \mathbf{W}_x + \mathbf{Y}_{(t-1)}\, \mathbf{W}_y + \mathbf{b}\right)$

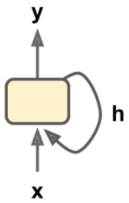
  $\mathbf{Y}_{(t)} : m \times n_{\text{neurons}},\ \mathbf{X}_{(t)} : m \times n_{\text{inputs}},\ \mathbf{b} : n_{\text{neurons}}$

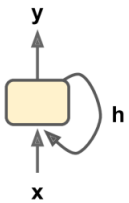
  $\mathbf{W}_x : n_{\text{inputs}} \times n_{\text{neurons}},\ \mathbf{W}_y : n_{\text{neurons}} \times n_{\text{neurons}}$

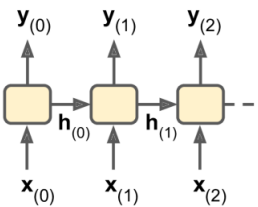
Red neuronal recurrente

Estado de la celda de memoria $\mathbf{h}_{(t)}$

- $\mathbf{h}_{(t)} = f_{\mathbf{W},\mathbf{b}}\left(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)}\right)$ (los pesos se comparten en todos los instantes de tiempo)





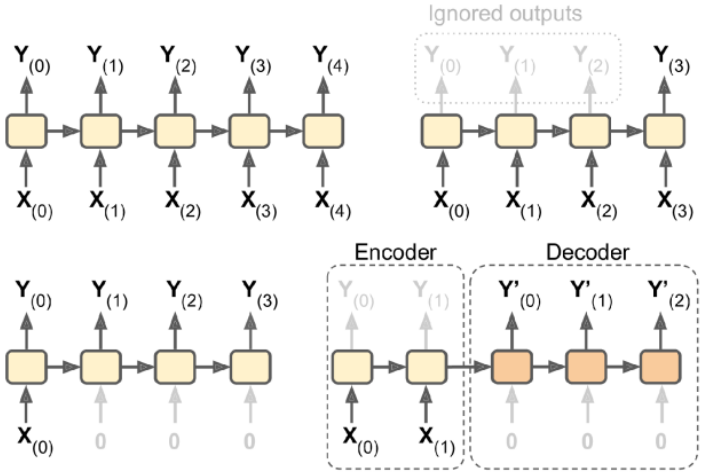


Recurrencia

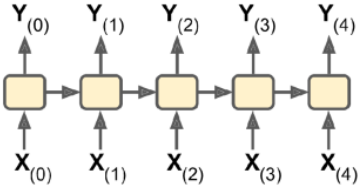
$$
\begin{aligned}
\mathbf{h}_{(t)} &= f\left(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)}\right) \\
&= f\left(f\left(\mathbf{h}_{(t-2)}, \mathbf{x}_{(t-1)}\right), \mathbf{x}_{(t)}\right) \\
&= f\left(f\left(f\left(\mathbf{h}_{(t-3)}, \mathbf{x}_{(t-2)}\right), \mathbf{x}_{(t-1)}\right), \mathbf{x}_{(t)}\right) \\
&\vdots \\
&= \tilde{f}\left(\mathbf{h}_{(0)}, \mathbf{x}_{(1)}, \mathbf{x}_{(2)}, \mathbf{x}_{(3)}, \ldots, \mathbf{x}_{(t)}\right)
\end{aligned}
$$

El estado $\mathbf{h}_{(t)}$ depende del estado inicial $\mathbf{h}_{(0)}$ y todas las entradas anteriores hasta t.

Secuencias de entrada y salida

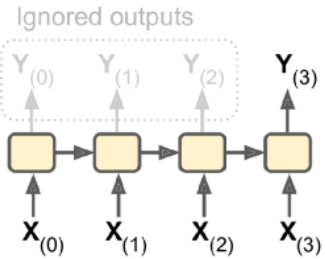


Secuencia a secuencia



Uso típico: predicción de series temporales, imputación de datos faltantes

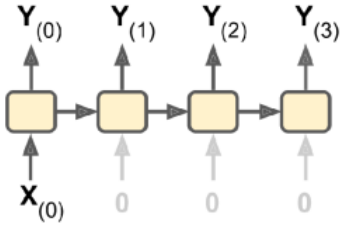
Secuencia a vector



Ejemplos:

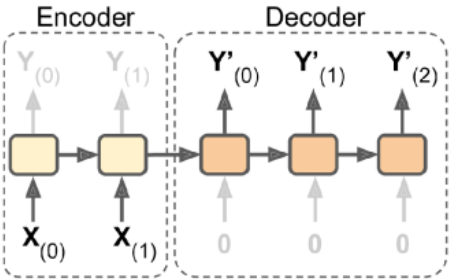
- Análisis de sentimiento a partir de un texto o video
- Score de recomendación de película a partir de texto de reseña de espectador

Vector a secuencia



Ejemplo: *image captioning*. Inferir texto que describa el contenido semántico de una escena.

Encoder - decoder

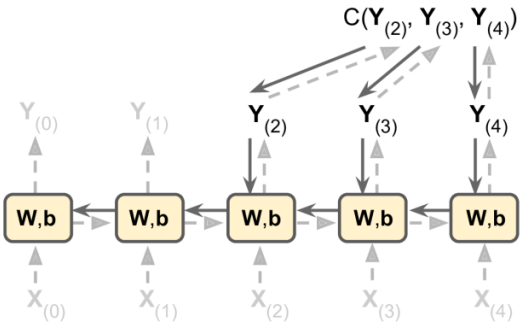


Ejemplo: traducción de texto. Necesitamos ver toda la secuencia entera, para luego sacar toda la secuencia traducida.

Entrenamiento: propagación hacia atrás en el tiempo

(*Back-propagation Through Time, BPTT*)

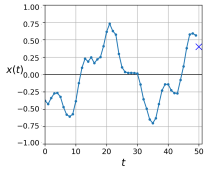
- Desenrollar la red y aplicar back-propagation como en una red feed-forward
- Forward pass en la red desenrollada y cálculo de función de costo C
- Intervienen todas las salidas usadas por la función de costo
- Los gradientes de la función de costo se propagan hacia atrás
- Como los pesos se comparten los gradientes se suman

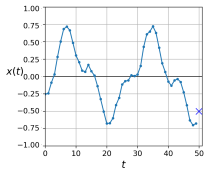


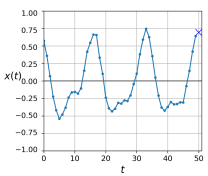
Implementación de RNN simple

- Predicción de series temporales (univariadas)
- Datos sintéticos: suma de dos sinusoides y ruido
- Considerar 50 primeros valores y predecir el siguiente

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)

np.random.seed(42)

n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

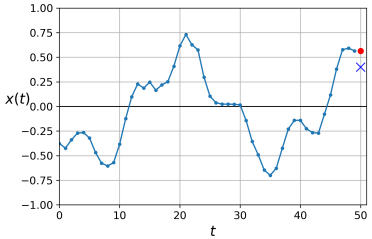


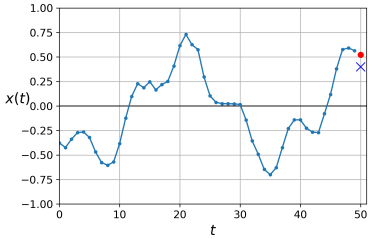




Medidas de referencia

- Predecir con el valor anterior ("persistencia"):
  MSE = 0.020
- Predecir con modelo lineal (FCN):
  MSE = 0.004

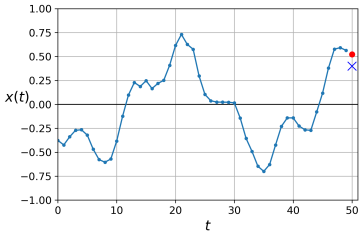





```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

Implementación de RNN simple

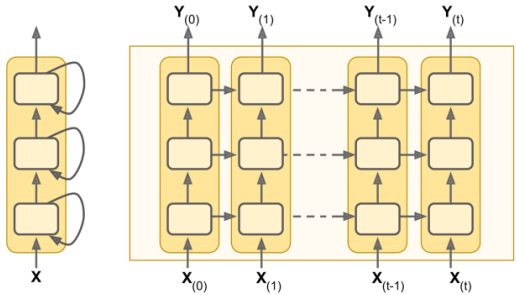
- No es necesario especificar el largo de la entrada
- Modelo: una capa con una única neurona recurrente
- No supera al modelo lineal (MSE = 0.010)
- Pero: muchísimos menos parámetros.
  Por neurona:
  - FCN: 1 parám. por tiempo + 1 bias = 51
  - RNN: 1 parám. por entrada + 1 parám. por estado + 1 bias = 3




```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])

optimizer = keras.optimizers.Adam(lr=0.005)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

Implementación de RNN profundas

- Múltiples capas de neuronas recurrentes
- Retornar secuencias en capas intermedias
- Mejora el modelo lineal (MSE = 0.002)
- La capa de salida puede ser densa:
  - Salida con una sola neurona recurrente no parece muy útil
  - Salida con neurona recurrente requiere activación tanh: salidas en $(-1, 1)$
  - Capa densa: mayor libertad de elección para la activación (no necesariamente saturante)

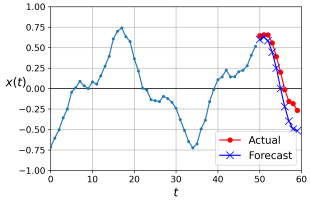


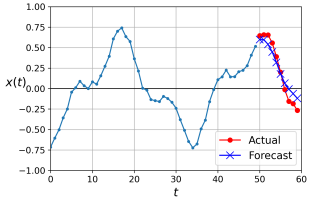

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
```

Predecir varios pasos hacia adelante

Supongamos que queremos predecir 10 pasos hacia adelante. Exploremos tres opciones:





1. Predicción de a un paso con el modelo anterior
   ⇒ Acumula los errores de predicción
2. Predecir 10 valores hacia adelante a partir de los 50 valores anteriores:
   Supera modelo lineal, pero podemos hacer mejor

```
n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:]
```

Manejando secuencias largas

- Entrenar con secuencias largas implica muchos pasos temporales y una red muy profunda
- Como cualquier red profunda una RNN puede sufrir el problema del gradiente inestable
- Además cuando una RNN procesa una secuencia muy larga olvida los primeros pasos

Flujo de gradientes en RNN

Al hacer backpropagation τ pasos se multiplica por $\mathbf{W}_{hh}^{\mathsf{T}}$ τ veces

- Si $\lambda_{\max}(\mathbf{W}_{hh}) > 1$: Gradiente explota
  funciones de activación staturadas (e.g. tanh), gradient clipping, inicialización
- Si $\lambda_{\max}(\mathbf{W}_{hh}) < 1$: Gradiente se desvanece

Por esta razón, no se puede usar ReLU ni cualquier activación sin saturación

Manejando secuencias largas: el problema de los gradientes inestables

Para mitigar este problema, se pueden usar técnicas que ya vimos para redes profundas:

- Inicializaciones adecuadas para el tipo de red
- Optimizadores con inercia, Adam, etc.
- Dropout
- Truncated BPTT, gradient clipping.

Estrategias de normalización:

- *Batch Normalization*
  - No es efectivo dentro de una celda: en cada instante de tiempo se utilizaría la misma normalización, independientemente de los verdaderos valores de $\mathbf{X}_{(t)}$ y $\mathbf{h}_{(t)}$
  - Se puede usar entre capas, pero se ha visto que no proporciona una mejora sustancial*.

*C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio, "Batch normalized recurrent neural networks," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2657–2661, 2016

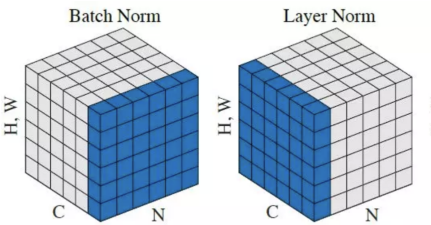
Manejando secuencias largas: el problema de los gradientes inestables

Estrategias de normalización:

- *Layer Normalization**

– Se normaliza en cada capa oculta. Por cada instancia, para cada capa oculta, se calcula media y varianza de las entradas a las neuronas de la capa, y se normaliza para tener media cero y varianza unitaria

– Ventaja adicional: al aplicarse a cada instancia, se comporta igual en training y en testing (a diferencia de Batch Normalization).



*L. J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. abs/1607.06450, 2016

Manejando secuencias largas: el problema de los gradientes inestables

Layer Normalization:

```
from tensorflow.keras.layers.experimental import LayerNormalization

class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = LayerNormalization()
        self.activation = keras.activations.get(activation)
    def get_initial_state(self, inputs=None, batch_size=None, dtype=None):
        if inputs is not None:
            batch_size = tf.shape(inputs)[0]
            dtype = inputs.dtype
        return [tf.zeros([batch_size, self.state_size], dtype=dtype)]
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
```

- Se customiza una celda hereda de `keras.layers.Layer`
- Constructor:
  1. Entradas: cantidad de unidades, tipo de activación
  2. Setea `state_size`, `output_size`
  3. Crea una `SimpleRNNCell` sin activación, ya que `LayerNormalization` se aplica luego de la operación lineal y antes de la activación
  4. Luego se aplica la activación

Método `Call()`:

1. Aplica la celda RNN simple, que calcula la combinación lineal de la entrada actual y los estados previos.
2. Devuelve el resultado dos veces (porque en la RNN simple el estado es igual a la salida); ignoramos `new_states[0]`.
3. Aplica `LayerNormalization` + activación
4. Devuelve la salida dos veces (como output y como nuevos estados).

Manejando secuencias largas: el problema de la memoria de corto plazo

- Al avanzar el tiempo, al pasar los datos por la RNN, en cada paso se va perdiendo información.
- Ejemplo: si queremos traducir una frase larga o transcribir una frase de memoria, olvidamos más fácilmente los comienzos.
- Solución: celdas específicamente diseñadas para extender la memoria de corto plazo:
  - LSTM (*Long Short-Term Memory*)*
  - GRU (*Gated Recurrent Unit*)†

*S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997

†K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014. cite arxiv:1406.1078Comment: EMNLP 2014

Manejando secuencias largas: el problema de la memoria de corto plazo

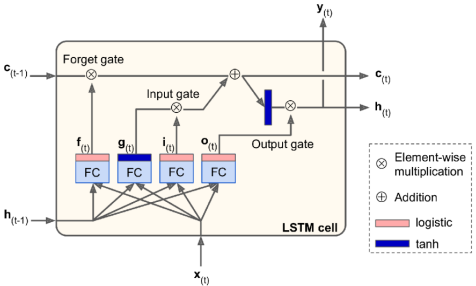
- Ambas celdas se pueden utilizar como caja negra, en lugar de la celda básica. Aceleran la convergencia en entrenamiento, y capturan dependencias en los datos de mayor rango.

```
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(1))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_10_time_steps_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

```
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.GRU(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(1))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_10_time_steps_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

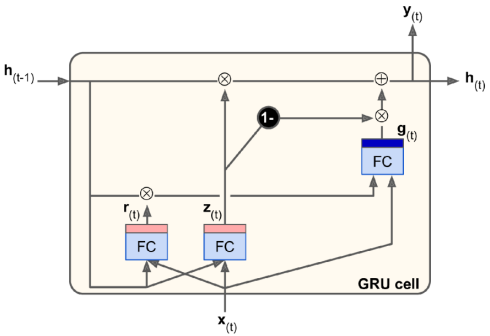
Celda LSTM

- Divide el estado en:
  - $\mathbf{h}_{(t)}$: captura la memoria corta
  - $\mathbf{c}_{(t)}$: captura la memoria a mayor rango
- Compuerta $\mathbf{g}_{(t)}$ lleva el flujo de memoria (tanh)
- Las compuertas $\mathbf{f}_{(t)}$ (*forget gate*), $\mathbf{i}_{(t)}$ (*input gate*) y $\mathbf{o}_{(t)}$ (*output gate*) actúan como selectores modulando el flujo de memoria (activación logística, salida en el intervalo $(0, 1)$), a partir de la memoria corta $\mathbf{h}_{(t-1)}$ y la entrada actual $\mathbf{x}_{(t)}$
- $\mathbf{c}_{(t-1)}$ pasa por la *forget gate* eliminando algunas memorias cortas seleccionadas de $\mathbf{h}_{(t-1)}$. Luego, agregando memorias nueva seleccionadas de $\mathbf{h}_{(t-1)}$ por el *input gate* se genera $\mathbf{c}_{(t)}$
- $\mathbf{h}_{(t)}$ se genera modificando $\mathbf{c}_{(t)}$ con una tanh, y filtrando con la *output gate*.



Celda GRU

- Simplifica la celda LSTM, logrando igual desempeño
- Ambos vectores de estado (memoria corta y memoria larga) se unen en un mismo $\mathbf{h}_{(t)}$
- Una única compuerta controladora $\mathbf{z}_{(t)}$ controla los factores de olvido y de salida:
  - Si $\mathbf{z}_{(t)} \simeq 1$: se abre la *forget gate* y se cierra la *input gate* $(1 - \mathbf{z}_{(t)} \simeq 0)$
- La compuerta $\mathbf{r}_{(t)}$ controla qué parte del estado previo se muestra a la compuerta principal $\mathbf{g}_{(t)}$



Celdas LSTM y GRU: observaciones

- Decisivas en el éxito de las RNN
- De todas formas su memoria es limitada y les cuesta capturar patrones de larga escala en secuencias superiores a los 100 pasos
- Una forma de mitigar este efecto es acortando las secuencias mediante capas convolucionales 1D.

Referencias

C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio, "Batch normalized recurrent neural networks," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2657–2661, 2016.

L. J. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. abs/1607.06450, 2016.

S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.
cite arxiv:1406.1078Comment: EMNLP 2014.

A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*.
O'Reilly Media, Inc., 2019.

A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *IEEE Intelligent Systems*, vol. 24, no. 2, pp. 8–12, 2009.