UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

19|20

Trabajo fin de grado

Neural Turing Machines

Alejandro Cabana Suárez

Campus Internacional excelencia UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Neural Turing Machines

Autor: Alejandro Cabana Suárez
Tutor: Luis Fernando Lago Fernández

julio 2020

Alejandro Cabana Suárez
Neural Turing Machines

Alejandro Cabana Suárez

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Actualmente las LSTM son los modelos más utilizados para problemas de análisis de secuencias, pero se ven limitadas en ciertos aspectos al tratar con secuencias largas o problemas en los que es necesario recordar demasiada información. Las NTM proponen una mejora de las capacidades de las redes neuronales recurrentes a través del uso de una memoria externa. El uso de una memoria, mientras sea suficientemente grande, permite almacenar cualquier cantidad de información durante el tiempo que sea necesario. Como toda su estructura es diferenciable, se las puede entrenar de manera eficiente con descenso por gradiente. En este Trabajo de Fin de Grado, desarrollamos desde el principio una implementación de la NTM y la evaluamos en el problema de copiar secuencias, que resulta complejo para las redes tradicionales. Comprobamos que nuestro modelo es capaz de resolver este problema y que podemos comprender por encima la estrategia que sigue para hacerlo.

Palabras clave

Aprendizaje Profundo, Red Neuronal Recurrente, Máquina de Turing Neuronal

Abstract

Nowadays LSTMs are the most widely used models for sequence modeling problems, but they are limited in some aspects when dealing with long sequences or problems that require storing too much information. NTMs propose an improvement of recurrent neural networks' capabilities through the use of an external memory. The use of this memory, if sufficiently big, allows the model to keep any amount of data for as long as needed. As all of the structure is differentiable, NTMs can be efficiently trained with gradient descent. In this Bachelor's thesis, we develop an implementation of the NTM from scratch and evaluate it on the sequence copying task, which turns out challenging for traditional networks. We check that our model is capable of solving this task and that we can understand the strategy it follows to do it.

Keywords

Deep Learning, Recurrent Neural Network, Neural Turing Machine

Índice

Listas

Lista de códigos

Lista de ecuaciones

Lista de figuras

Lista de tablas

1 Introducción

Procesamiento del lenguaje natural, generación de música, generación de texto, conversión texto-voz, todos estos son problemas que se pueden resolver con técnicas de aprendizaje profundo (*deep learning*), concretamente con redes neuronales recurrentes. El estado del arte en este campo está dominado principalmente por las *Long Short-Term Memory (LSTM)* y las *Gated Recurrent Unit (GRU)* , que añaden a las redes recurrentes básicas la capacidad de decidir qué información olvidar o recordar y en qué medida. Para ello, este tipo de redes depende de un estado interno, que puede coincidir o no con la salida de la red, y se transmite y modifica a lo largo del tiempo. Sin embargo, es el modelo el que decide cómo codificar la información en el estado y hay un límite a lo que estos modelos pueden almacenar en él. Una idea para aumentar la capacidad de almacenamiento de información de las redes neuronales son las *Memory-Augmented Neural Network (MANN)* : redes neuronales con acceso a algún tipo de memoria en la que pueden leer y escribir, como podría ser una pila, una cola o, en el caso de la *Neural Turing Machine (NTM)* propuesta por Alex Graves [1], una memoria direccionable de gran tamaño. Esta memoria es precisamente lo que le da su nombre, en analogía a cómo la cinta de una máquina de Turing enriquece las capacidades de una máquina de estados finitos. El objetivo de este trabajo será estudiar estas NTM y su viabilidad en problemas que resultan complejos para redes como las LSTM .

Para hacer esto, desarrollamos una implementación propia de esta arquitectura, basada en la descripción de la propuesta original [1] y otras puestas en práctica de las NTM [2,3]. Una vez completada, la probamos en un problema sencillo pero que depende en gran medida de la capacidad del modelo de recordar información de forma precisa: copiar la cadena de entrada un tiempo después de recibirla. Una vez entrenado el modelo para resolver este problema, intentaremos visualizar la manera en que lo está resolviendo observando el contenido de la memoria y el movimiento de los cabezales de lectura y escritura a lo largo de la ejecución.

Respecto a la estructura del documento, comenzaremos presentando un breve repaso de las bases de las redes neuronales, seguido de una revisión del estado del arte centrado en las redes recurrentes y el análisis de secuencias. Introduciremos después el concepto de las máquinas de Turing neuronales, describiendo sus distintas partes y sus funciones, entre las cuales destaca el uso de una unidad de memoria. Por último, detallaremos el proceso de desarrollar una implementación funcional, los re-

sultados de esta implementación en un problema que requiera el uso de memoria y los problemas encontrados a la hora de llevarla a la práctica.

2

Estado del arte

En este capítulo revisaremos el estado del arte en lo que a redes neuronales se refiere.

2.1. Redes neuronales

Empecemos por el principio. Una red neuronal es un modelo computacional inspirado en el comportamiento del cerebro. Está formada por **perceptrones** o **neuronas artificiales** conectadas entre sí, emulando como su propio nombre indica a las células neuronales.

2.1.1. Perceptrones

Las neuronas biológicas son células con dos partes bien diferenciadas de su cuerpo principal. Por un lado tienen las dendritas, ramificaciones que actúan como sensores para captar estímulos del exterior; por otra parte, el axón, una prolongación más gruesa desde la que se dispara una señal de carácter eléctrico, químico o ambos, hacia las dendritas de otras neuronas cercanas [4].

Emulando este esquema, un perceptrón [5] se estructura como muestra la figura 2.1, contando con una serie de entradas numéricas y un valor de salida. El valor del perceptrón se calcula tomando una suma ponderada de estas entradas (usando unos **pesos**), y haciéndola pasar por una función no lineal. A esta función se la conoce como **función de activación**, ya que es la que decide si la neurona "se activa" o no. En la práctica, las entradas y los pesos se suelen representar con sendos vectores, de tal manera que hacer la suma ponderada equivale a calcular su producto escalar.

La imagen de la función de activación es lo que se considera la salida del perceptrón, y se puede interpretar de varias maneras. De manera discreta, puede fijarse un valor umbral y considerar las salidas mayores como respuestas positivas (la neurona se activa o se "dispara") y las menores como negativas (la neurona no se activa), o dar al rango de la función de activación una interpretación continua. En muchos casos interesa que la salida del perceptrón esté en un rango concreto, por ejemplo sólo números positivos o quizá intervalos más acotados como $(0, 1)$ o $(-1, 1)$. También es conveniente que se mantenga el orden del dominio; es decir, que si tenemos a, b con $a \leq b$ dos puntos a los que

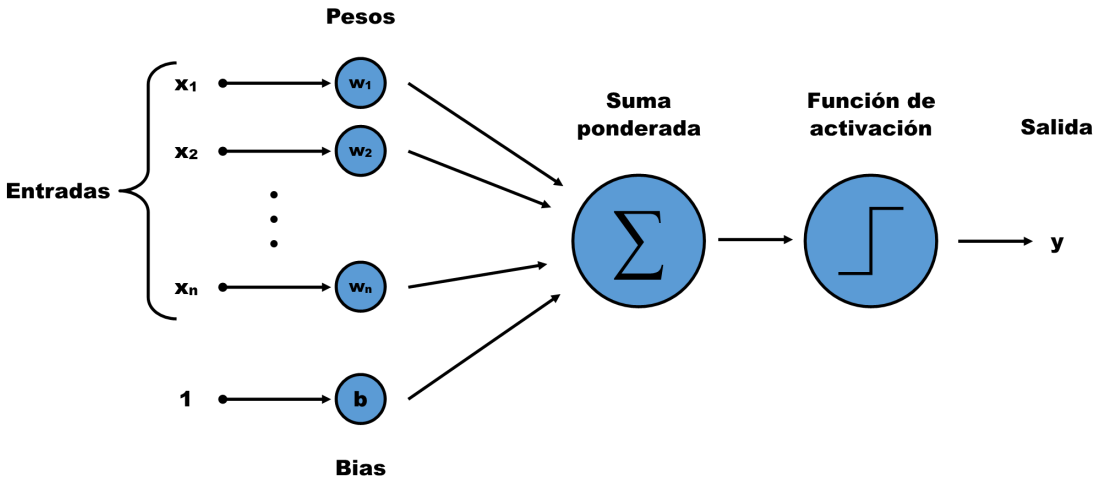


Figura 2.1: Estructura de un perceptrón: Las entradas se multiplican por sus respectivos pesos y se suman. El resultado, al que se le suma el *bias*, pasa por la función de activación para producir la salida.

les aplicamos la función de activación f, entonces siempre se cumpla $f(a) \leq f(b)$. Por esta razón, son populares las funciones de activación que cumplen estas propiedades. Algunos ejemplos de funciones de activación son:

Rectified Linear Unit (ReLU) Una función que se comporta como una función lineal con pendiente m para entradas positivas, pero que asigna 0 a las entradas negativas. Su gráfica se muestra en la figura 2.2.

$$\boxed{\mathrm{ReLU}_m(x) = \max\{0, mx\}} \tag{2.1}$$

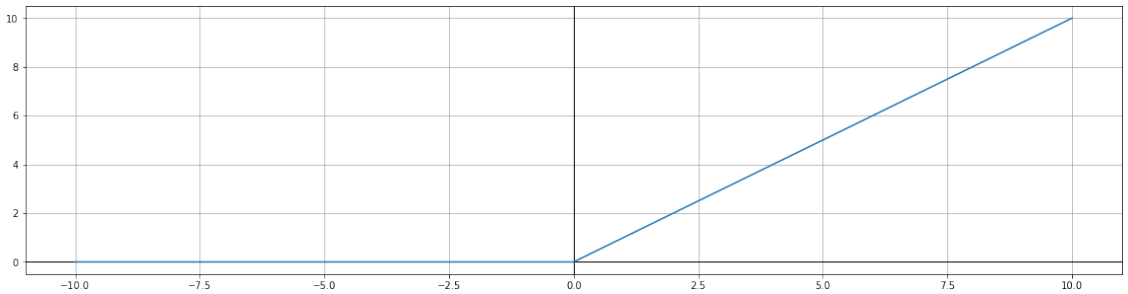


Figura 2.2: Gráfica de la función ReLU .

Sigmoide Una función suave que conserva el orden de su dominio y tiene como rango el intervalo $(0, 1)$. Su gráfica se muestra en la figura 2.3.

$$\boxed{\sigma(x) = \frac{1}{1 + e^{-x}}} \tag{2.2}$$

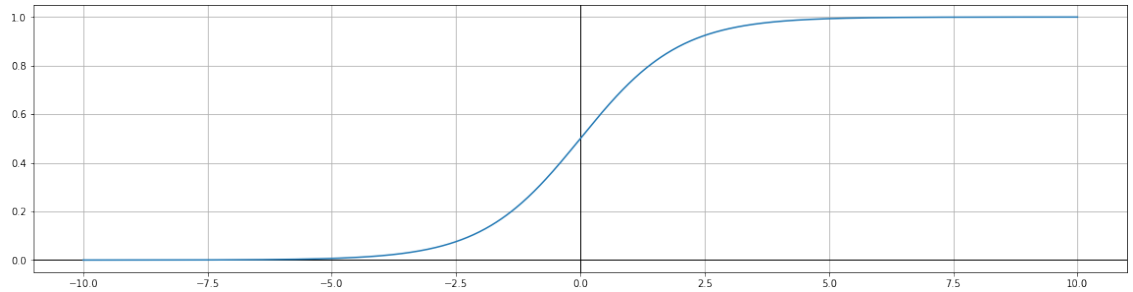


Figura 2.3: Gráfica de la función sigmoide.

Tangente hiperbólica Una función suave que conserva el orden de su dominio y tiene como rango el intervalo $(-1, 1)$. Su gráfica se muestra en la figura 2.4.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

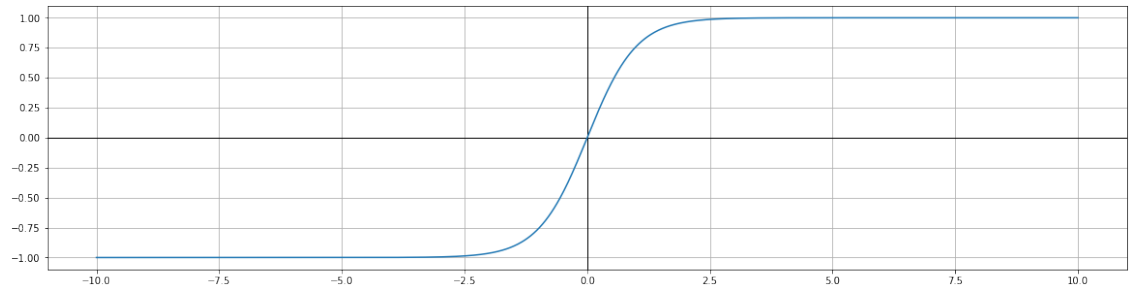


Figura 2.4: Gráfica de la tangente hiperbólica.

Softplus Se trata de una aproximación suave de la función ReLU . Como curiosidad, su derivada es la función sigmoide. Su gráfica se muestra en la figura 2.5.

$$\text{softplus}(x) = \log(1 + e^x) \tag{2.4}$$

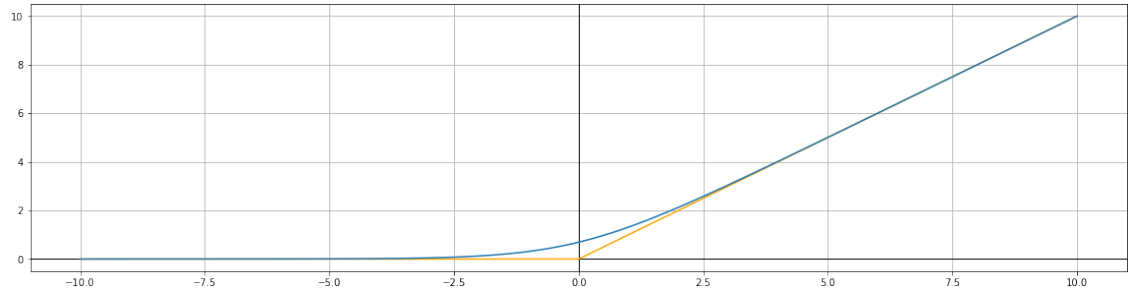


Figura 2.5: Gráfica de la función *softplus* (azul) comparada con ReLU (naranja).

Softmax Al contrario que las anteriores, que se aplican a la salida de cada perceptrón, esta función se aplica a un vector de salidas de varios perceptrones, lo que llamaremos "capa" cuando expliquemos las redes neuronales. La función *softmax* o exponencial normalizada es una generalización de la sigmoide que comprime un vector a uno del mismo tamaño pero con componenetes en [0,1] que además suman 1. Como su nombre sugiere, su ecuación es la siguiente para cada componente i del vector:

$$\text{softmax}(x)[i] = \frac{e^{x[i]}}{\sum_i e^{x[i]}} \tag{2.5}$$

La manera en que un perceptrón **aprende** a solucionar un problema es modificando sus pesos de tal manera que se minimice una función de coste (por ejemplo, la diferencia entre la salida y la salida esperada).

2.1.2. La red

Conectando las salidas de perceptrones con entradas de otros perceptrones se construye lo que llamamos red neuronal. Existen varios tipos de red, dependiendo de la manera en que se conecten las neuronas entre sí. A grandes rasgos, podríamos distinguir las redes en las que las conexiones forman ciclos (**redes recurrentes**) de las redes en las que no (**redes *feed-forward***).

Redes *feed-forward* Es el tipo de red más simple, ya que, al no tener bucles, la información fluye en un sólo sentido, hacia delante. Por supuesto, la más simple entre ellas es la red formada por un solo perceptrón. Otro ejemplo sencillo es el llamado perceptrón multicapa, en el que las neuronas se agrupan en "capas" y las neuronas de una capa sólo se conectan a las de la siguiente. Entre las capas se distinguen la de entrada y la de salida y a las demás se las llama capas ocultas. Si entre dos capas existen todas las conexiones posibles entre neuronas, se dice que la segunda es una capa densa.

Redes recurrentes El hecho de que haya bucles en la red quiere decir que la salida de una neurona de las últimas capas puede influenciar el valor de una neurona de las primeras capas y que, por tanto, la información de un paso de ejecución se puede conservar y utilizar en los siguientes. Esto hace que este tipo de redes sea el que se utilice en problemas de análisis de secuencias como procesamiento de lenguaje natural o audio y vídeo. Hablaremos más de este tipo de redes del apartado 2.2 en adelante.

2.1.3. Aprendizaje

Llamamos aprendizaje al proceso por el que un modelo va adaptándose para resolver mejor un problema concreto. Este proceso consiste en pequeñas correcciones en los pesos que hacen que

se reduzca una **función de coste** que se va evaluando a lo largo del mismo. La magnitud de estas correcciones se conoce como **tasa de aprendizaje** o ***learning rate***. Con una tasa de aprendizaje elevada se avanza más rápido en la dirección deseada, pero también es fácil que estando cerca del mínimo de la función de coste se reaccione exageradamente a cualquier error. Por el contrario, una tasa de aprendizaje pequeña requiere muchos más pasos para realizar la corrección, pero en general el coste se reduce consistentemente en cada paso.

Como el objetivo del aprendizaje es reducir la función de coste, podemos pensar en él como en un problema de optimización. Una técnica muy utilizada es el descenso por gradiente, que consiste en tomar la derivada de la función de coste respecto a cada peso de la red. Pensando en la gráfica de la función de coste como una hipersuperficie en el espacio vectorial de los pesos, estas derivadas nos indican la dirección de máxima pendiente y, por tanto, también la de mínima pendiente. Siguiendo esta pendiente negativa, en algún momento se encontrará un mínimo local, que puede ser o no un coste aceptable para la red. Este algoritmo se conoce como descenso por gradiente [6–8].

Para calcular el gradiente, un metodo muy extendido en redes *feed-forward* (pero que también se adapta a ciertas redes recurrentes) es la **retropropagación** o ***backpropagation*** [9]. Su nombre viene de que, haciendo uso de la regla de la cadena, se comienza a calcular estas derivadas desde la última capa y se avanza hacia atrás calculando cada vez la derivada respecto a los valores de las neuronas de la capa anterior. De esta manera se evita repetir el cálculo de derivadas parciales intermedias.

2.2. RNN básica

El ejemplo más básico de *Recurrent Neural Network (RNN)* es conocido como red de Elman [10] y consiste en un perceptrón multicapa en el que la salida de una de las capas se conecta a su propia entrada o a la de una capa anterior, de tal manera que la información persiste entre pasos de ejecución. A causa de esta retroalimentación, este tipo de redes es más adecuado para analizar secuencias que una red *feed-forward*. Sin embargo, el entrenamiento se vuelve más costoso según crece la longitud de las secuencias a analizar. Este efecto se puede reducir “desenrollando” la red en el tiempo; es decir, construyendo una red no recurrente en la que cada capa se corresponde con un paso temporal de la original. Otro problema que puede surgir es que si la secuencia es demasiado larga, los gradientes pueden crecer demasiado (*exploding gradient*) o reducirse a cada paso hasta desaparecer (*vanishing gradient*) [11]. El primer problema se puede paliar recortando los gradientes que se hacen demasiado grandes (*gradient clipping*), mientras que para el segundo hacen falta estructuras más complejas como las siguientes.

2.3. LSTM

Una celda **LSTM** [12] es una clase de RNN diseñada para lidiar con el *vanishing gradient problem*. La consecuencia de este problema que se pretende evitar, es que cuanto más tiempo transcurre desde un paso, más se "difumina" la información que se recuerda de ese paso. Para evitarlo, la arquitectura que se utiliza es la siguiente.

En la figura 2.6 se puede apreciar que en lugar de ser la salida de la celda lo único que se transmite al siguiente paso de tiempo, se mantiene también un **estado de celda (*cell state*)** con la información que la celda "quiere recordar". Además, se hace pasar la información por tres **puertas (*gates*)** que regulan cómo fluye por la celda:

Puerta de olvido (*forget gate*) En base a la entrada (y la salida del tiempo anterior), decide en qué medida conservar la información del estado de celda. Sigue la siguiente ecuación, en la que x_t es la entrada en tiempo t, h_{t-1} es la salida en tiempo $t-1$, W_f es la matriz de pesos de la capa, R_f es la matriz de pesos recurrentes de la capa, b_f es el *bias* de la capa y σ es la función sigmoide:

$$f_t = \sigma_t(W_f x_t + R_f h_{t-1} + b_f). \tag{2.6}$$

Puerta de entrada (*input gate*) Decide la nueva información de la entrada que añadir al estado de celda. Sigue las siguientes ecuaciones, que comparten x_t y h_t con la anterior y tienen W, R y b como pesos, pesos recurrentes y *bias*, respectivamente. c_{t-1} es el estado de celda en el tiempo $t-1$ y $\circ$ representa el producto punto por punto (*pointwise*):

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i). \tag{2.7a}$$

$$\tilde{c}_t = \tanh(W_c x_t + R_c h_{t-1} + b_c). \tag{2.7b}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t. \tag{2.7c}$$

Puerta de salida (*output gate*) A partir del estado de celda y la entrada, decide qué información dar como salida. Sigue las siguientes ecuaciones, donde de nuevo W, R y b son pesos, pesos recurrentes y *bias* de capas ocultas:

$$o_t = \sigma(W_o x_t + R_o h_{t-1} + b_o). \tag{2.8a}$$

$$h_t = o_t \circ \tanh(c_t). \tag{2.8b}$$

Esta estructura permite que la celda decida cuánta de la información de los tiempos anteriores recordar y en qué medida, lo cual evita que la información antigua se vaya desvaneciendo de forma indeseada. Por este motivo, las LSTM son más adecuadas para el análisis de secuencias que otras arquitecturas como las RNN estándar.

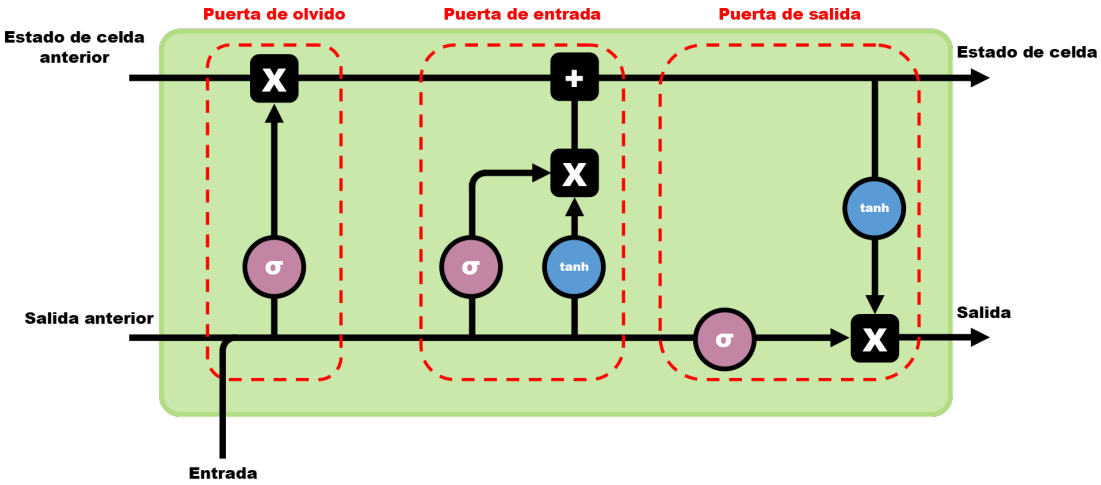


Figura 2.6: Estructura de una LSTM: "σ" y "tanh" representan capas ocultas con funciones de activación sigmoide y tangente hiperbólica, respectivamente. Los bloques "X" y "+" representan producto y suma punto por punto. Las conexiones de lineas perpendiculares representan duplicado, mientras que las conexiones tangentes (como la entrada) implican concatenación.

Por otra parte, el problema evidente de las LSTM es que realizan bastantes operaciones con tensores, y por tanto tardan más y consumen más recursos a la hora de entrenar que una RNN básica.

2.4. GRU

La arquitectura de una GRU [13] es muy similar a la de una LSTM . La principal diferencia entre las dos radica en que GRU no tiene estado de celda, sino que usa la salida (o estado oculto) para transmitir información a los siguientes pasos. Además, también tiene dos puertas en lugar de las tres de una LSTM :

Puerta de actualización (*update gate*) Actúa de manera similar a las puertas de entrada y olvido de una LSTM : decide qué información olvidar y cuál añadir. Sigue la ecuación:

$$z_t = \sigma(W_z x_t + R_z h_{t-1} + b_z), \tag{2.9}$$

donde W_z, R_z y b_z son pesos, pesos recurrentes y *bias* de la capa, x_t y h_t son la entrada en tiempo t y la salida en tiempo $t-1$ y σ la función sigmoide.

Puerta de reinicio (*reset gate*) Otra puerta que decide qué información anterior desechar.. Sigue la ecuación:

$$r_t = \sigma(W_r x_t + R_r h_{t-1} + b_r), \tag{2.10}$$

donde de nuevo W_r, R_r y b_r son pesos, pesos recurrentes y *bias* de la capa, x_t y h_t son

la entrada en tiempo t y la salida en tiempo $t-1$ y σ la función sigmoide.

La salida se calcula siguiendo estas últimas ecuaciones, que tienen W_h, R_h y b_h como pesos, pesos recurrentes y *bias* de la capa y x_t y h_{t-1} como entrada en tiempo t y salida en tiempo $t-1$:

$$\tilde{h}_t = \tanh(W_h x_t + R_h(r_t \circ h_{t-1}) + b_h). \tag{2.11a}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t. \tag{2.11b}$$

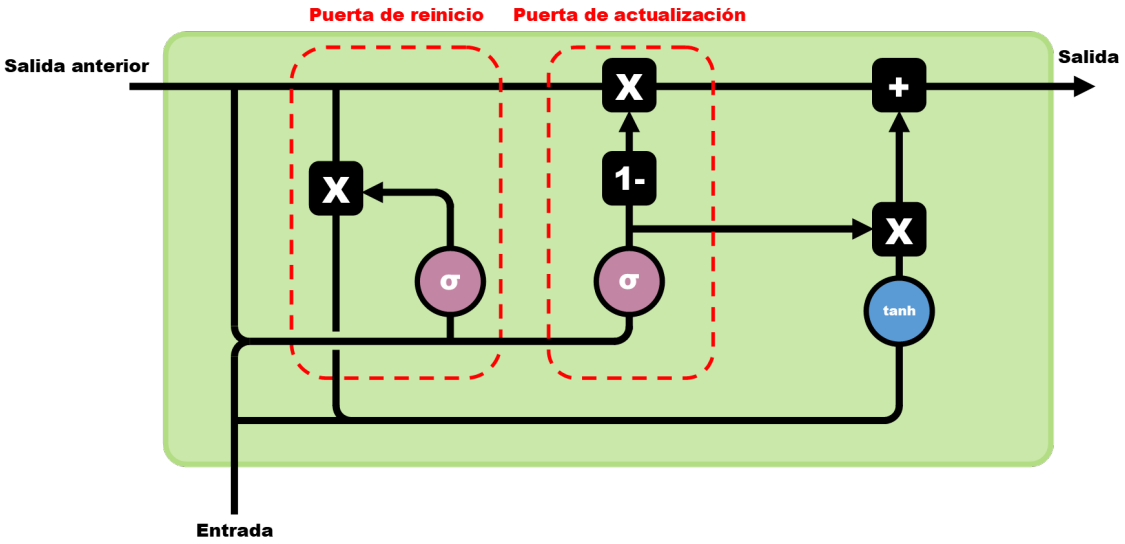


Figura 2.7: Estructura de una GRU: "σ" y "tanh" representan capas ocultas con funciones de activación sigmoide y tangente hiperbólica, respectivamente. Los bloques "X" y "+" representan producto y suma punto por punto. El bloque "1-" produce un vector en el que cada componente es 1 menos la correspondiente del vector de entrada. Las conexiones de lineas perpendiculares representan duplicado, mientras que las conexiones tangentes implican concatenación.

Como se puede comprobar en la figura 2.7, una celda de este tipo tiene menos operaciones con tensores y, por tanto, entrena por lo general más rápido que una LSTM . Sin embargo, no está claro que una de las dos sea mejor que la otra, lo que hace que por lo general se prueben modelos con cada una para elegir la que más se adecue al problema concreto.

2.5. MANN

Otra idea para solucionar el problema de que la información se pierda con el tiempo es dotar a la red de la capacidad de escribir en una memoria externa y leer su contenido en otro momento. A este tipo de redes se las conoce como redes neuronales aumentadas con memoria o MANN [14,15].

Esta memoria puede ser en realidad de cualquier tipo, siempre que sea un elemento diferenciado de la red en el que esta pueda realizar operaciones de lectura y escritura. Una primera aproximación

consiste en añadir una pila o una cola a la red [16, 17]. La adición de este tipo de memoria no supone un gran aumento del número de parámetros de la red, por lo que no añaden un coste elevado de tiempo a la hora de entrenar.

Por otra parte, se puede añadir una unidad de memoria con operaciones de acceso aleatorio. Esta es la particularidad de las NTM , en las que se centrará el siguiente capítulo.

En cualquiera de los casos, en general cuando pensamos en lectura y escritura en una memoria nos imaginamos operaciones discretas, que solo afectan a ciertas posiciones. Sin embargo, uno de los retos en este tipo de redes es volver estas operaciones continuas y diferenciables, de tal manera que los modelos puedan entrenarse con métodos de descenso por gradiente.

3

Neural Turing Machines

Como comentábamos, una NTM se caracteriza por disponer de una memoria en la que puede almacenar información. Como se puede ver en la figura 3.1, una NTM está formada por una red usual a la que llamamos controlador que interactúa con la memoria a través de los mecanismos de lectura y de escritura.

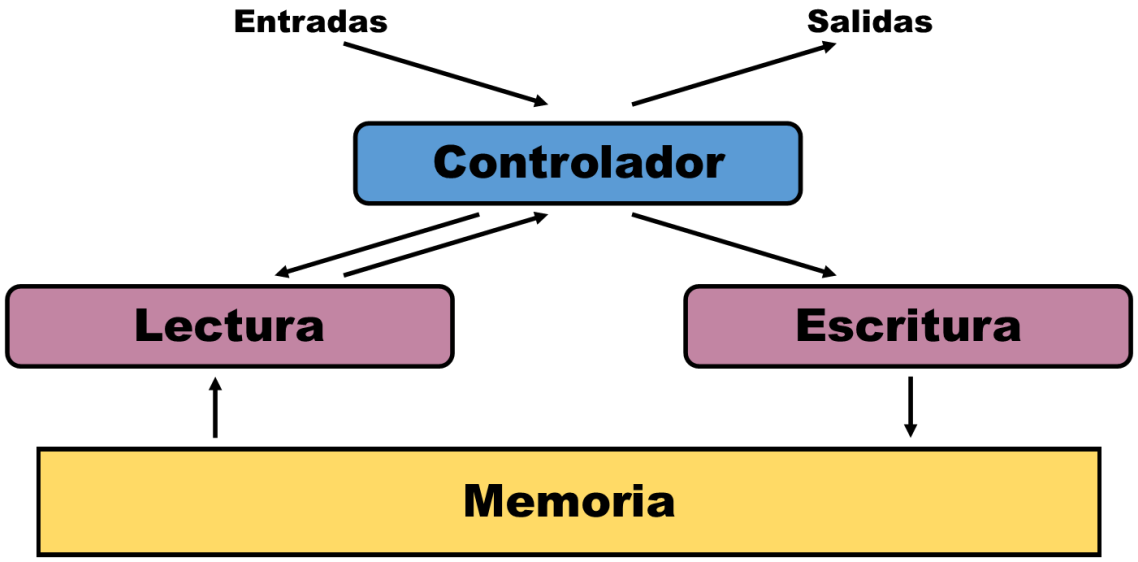


Figura 3.1: Estructura a alto nivel de una NTM: A partir de las entradas y el vector leído de la memoria, el controlador compone una salida y da órdenes a los mecanismos de lectura y escritura, que actualizan la memoria y el vector leído.

En realidad, la red no lee y escribe en una posición concreta de la memoria en cada paso, sino que interactúa con toda la memoria a la vez enfocándose más en ciertas posiciones. Esto hace que la lectura y escritura de las NTM sea “difusa” o “borrosa” y es importante porque permite que todos estos mecanismos sean diferenciables y que, por tanto, se pueda entrenar la red con métodos de descenso por gradiente. El controlador puede elegir en qué posiciones de memoria centrar su atención de dos maneras distintas que puede combinar: puede seleccionar posiciones de memoria en base a su contenido (acceso por contenido), o puede desplazarse en relación a la posición a la que accedió anteriormente (acceso por posición).

3.1. Direccionamiento

El mecanismo de direccionamiento es el que se encarga de generar tanto los pesos con los que la red leerá como los pesos con los que escribirá en cada posición de memoria. Para ambos casos el proceso es el mismo. Digamos que en el tiempo t la memoria $\boldsymbol{M}_t$ es una matriz $N \times M$, donde M es la longitud de cada posición de memoria y N es el número de posiciones de memoria. Los pesos generados serán un vector de longitud N cuyas componentes serán números en el intervalo $[0, 1]$ e indicarán cuánto se enfoca la red para leer o escribir en cada posición de memoria correspondiente. Veamos cómo se obtienen estos pesos pasando por los dos tipos de acceso.

3.1.1. Acceso por contenido

En primer lugar, se produce un vector clave $\boldsymbol{k}_t$ en tiempo t de longitud N que se compara con cada posición de la memoria $\boldsymbol{M}_t[i]$ usando una función de similitud K, en el caso de nuestra implementación la similitud coseno:

$$K(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\| \cdot \|\boldsymbol{v}\|}. \tag{3.1}$$

Este vector de similitudes se multiplica por un escalar positivo β_t que aumenta o disminuye la diferencia entre los pesos y a continuación se pasa por una función exponencial normalizada (*softmax*) para obtener un vector w_t^c con componentes en $[0, 1]$. Todo este proceso se realiza siguiendo esta ecuación para cada $i \in \{0, \cdots, N-1\}$:

$$w_t^c[i] = \frac{e^{\beta_t K(\boldsymbol{k}_t, \boldsymbol{M}_t[i])}}{\sum_{j=0}^{N-1} e^{\beta_t K(\boldsymbol{k}_t, \boldsymbol{M}_t[j])}}. \tag{3.2}$$

Con un β_t elevado ($\beta_t > 1$) el efecto es que se aumenta mucho la distancia entre el peso más grande y todos los demás, haciendo que la red se enfoque más en esa posición. Con un β_t bajo, en cambio, los pesos se acercan más al más elevado, y la atención de la red se desenfoca más.

3.1.2. Acceso por posición

El acceso por posición permite iterar de forma sencilla por las posiciones de memoria. Consiste en rotar los pesos calculados en un paso anterior, de manera que si por ejemplo toda la atención estaba centrada en una sola posición, con una rotación de distancia 1 se cambiaría toda la atención a la siguiente posición de memoria.

Lo primero que se decide es si se va a rotar respecto a los pesos del paso anterior w_{t-1}, los pesos de acceso por contenido recién calculados w_t^c o una combinación de los dos. Esto se hace mediante

un parámetro $g_t \in (0, 1)$ de la siguiente manera:

$$w_t^g = g_t w_t^c + (1 - g_t) w_{t-1}. \tag{3.3}$$

Si g_t se acerca a 0, entonces estamos tomando los pesos del paso anterior y si se acerca a 1, entonces tomamos los pesos del acceso por contenido.

Las rotaciones se realizan en base a unos pesos de rotación s_t. Podemos pensar en s_t como un vector cuyas componentes suman 1 y están entre 0 y 1, en el que cada una de estas componentes indica en qué medida se rotarán los pesos hacia cada distancia. Por ejemplo, si se permiten rotaciones de distancias entre -1 (retroceder una posición) y 1 (avanzar una posición), en ese caso s_t tendrá 3 componentes. Si la correspondiente a -1 es 1 y las demás 0, entonces los pesos se rotarán una posición hacia atrás. La ecuación que rige estas operaciones es la siguiente, para cada $i \in \{0 \cdots N-1\}$:

$$\tilde{w}_t[i] = \sum_{j=0}^{N-1} w_t^g[j] s_t[i-j], \tag{3.4}$$

donde las operaciones con índices son todas módulo N. En realidad podemos pensar en la ecuación 3.4 como rotar w_t^g cada distancia permitida, multiplicar cada una por su peso correspondiente de s_t y a continuación sumarlas todas.

Por último, para evitar que estos pesos hagan que el enfoque sea muy "borroso" y propiciar que los pesos finales sean muy cercanos a 0 o muy cercanos a 1, se realiza la siguiente operación con un nuevo parámetro escalar $\gamma_t > 1$, para cada $i \in \{0 \cdots N-1\}$:

$$w_t[i] = \frac{\tilde{w}_t[i]^{\gamma_t}}{\sum_{j=0}^{N-1} \tilde{w}_t[j]^{\gamma_t}}. \tag{3.5}$$

De esta forma además, las componentes de w_t están entre 0 y 1 y suman 1.

Todos los parámetros en estas ecuaciones (k_t, β_t, g_t, s_t y γ_t) los genera la red con capas ocultas con sus pesos, *bias* y funciones de activación que dependen del rango en el que se puede mover cada parámetro.

Recapitulando, a la hora de escoger los pesos el modelo puede elegir entre basarse en los pesos que produjo en el tiempo anterior, en los pesos generados por el acceso por contenido o una combinación lineal de ambos. Después puede rotar o no estos pesos a izquierda y/o derecha las veces necesarias y finalmente pasa el resultado por una fase de enfoque. Por tanto, entre las opciones con las que cuenta la NTM están buscar posiciones de memoria parecidas a la entrada y acceder a ellas o a las posiciones cercanas, y acceder a la misma posición que en el paso anterior y las circundantes.

Como nota final, una NTM puede tener varios "cabezales" de lectura y escritura; es decir, puede hacer varios procesos de lectura y escritura independientes en el mismo paso temporal. En caso de

haber más de uno, todo este proceso de generación de pesos (y los siguientes de lectura y escritura) se realiza una vez por cabezal, dando como resultado un vector de pesos diferente para cada uno.

3.2. Lectura

Sea w_t un vector de pesos de longitud N generado como en la sección anterior en un tiempo t. Recordemos que la ecuación 3.5 hace que todos los pesos estén en el intervalo $[0, 1]$ y sumen 1. El vector de lectura r_t será la suma ponderada con estos pesos de todas las posiciones de memoria:

$$r_t = \sum_{i=0}^{N-1} w_t[i] \boldsymbol{M}_t[i] . \tag{3.6}$$

Esta operación, que también se puede interpretar como el producto de matrices de $\boldsymbol{M}_t$ y w_t, es claramente diferenciable.

3.3. Escritura

Inspirándose en las puertas de entrada y olvido de una LSTM , la escritura en las NTM se divide en dos fases: borrado y adición.

Dado un vector de pesos w_t producido como explica la apartado 3.1 en el tiempo t y un vector de borrado e_t de longitud M y con elementos en $(0, 1)$, el contenido de la memoria en el tiempo anterior $\boldsymbol{M}_{t-1}$ se modifica de la siguiente forma, para cada $i \in \{0 \cdots N-1\}$:

$$\tilde{\boldsymbol{M}}_t[i] = \boldsymbol{M}_{t-1}[i] \circ (\mathbf{1} - w_t[i] e_t) . \tag{3.7}$$

En esta ecuación, $\mathbf{1}$ representa un vector fila en el que todas las componentes son 1 y $\circ$ es el producto punto por punto (*pointwise*).

Este mecanismo funciona de tal manera que una posición de memoria se borra completamente sólo si las componentes correspondientes del vector de borrado y el de pesos son las dos 1; si cualquiera de las dos es 0, esa posición de memoria no se ve afectada en esta fase. Al ser el producto conmutativo, en caso de haber varias cabezas de escritura, se pueden realizar todas las fases de borrado en cualquier orden sin generar conflictos.

Para la adición, se genera también un vector de adición a_t de longitud M, que se suma a la memoria siguiendo los pesos una vez completado el borrado:

$$\boldsymbol{M}_t[i] = \tilde{\boldsymbol{M}}_t[i] + w_t[i] a_t . \tag{3.8}$$

De nuevo, el orden en el que se realizan las sumas en caso de haber varias cabezas de escritura no importa. Esta matriz $\boldsymbol{M}_t$ es el contenido final de la memoria en el tiempo t.

4

Diseño

En el capítulo anterior hemos explicado los componentes de una NTM desde un punto de vista teórico. En adelante, veremos el diseño concreto y las decisiones tomadas para nuestra implementación. Gran parte del diseño está basado en la implementación de Mark Collier [2] y las ideas de [3].

En la figura 4.1 podemos ver una estructura más detallada de nuestra NTM . Si bien [1] especifica los parámetros y las ecuaciones que rigen los mecanismos de lectura, escritura y direccionamiento, deja algunas cuestiones de la implementación abiertas. La más clara es el controlador: puede ser una capa recurrente o *feed-forward*, con cualquier número de unidades y cualquier función de activación. Otro detalle que [1] no menciona (pero [2] sí estudia) es el contenido inicial de la memoria, que podría llenarse con ceros, con unos o con cualquier distribución aleatoria. Lo mismo aplica al resto de elementos del estado de celda: el vector leído, el estado del controlador y los pesos de lectura y escritura. También es necesario escoger las funciones de activación de las capas que generan cada parámetro de las ecuaciones. Todos estos detalles los explicaremos en este capítulo.

El *framework* que utilizamos es Keras [18] sobre TensorFlow. Keras es una librería de *Deep Learning* que ofrece clases y funciones de alto nivel que facilitan mucho el trabajo en este campo. Concretamente, nuestra *NTMCell* será una clase que herede de la *SimpleRNNCell* de Keras.

Para Keras, una celda RNN representa una encapsulación de los estados (pesos) y las operaciones que se realizan en cada paso temporal en la evaluación de una RNN . Más concretamente, una celda RNN es una clase que tiene:

- Un método ***call***, que tiene como argumentos la entrada y el estado de celda en el tiempo t y devuelve la salida del tiempo t y el estado del tiempo $t + 1$. Este es el método en el que se ejecuta toda la lógica de la celda y es al que se llama en cada paso de tiempo.
- Un atributo ***state_size***, que indica la forma y las dimensiones del estado de celda que devuelve *call*.
- Un atributo ***output_size***, que indica la forma y las dimensiones de la salida que devuelve *call*.
- Un método ***get_initial_state***, que recibe como argumentos o bien directamente los datos de entrada, o bien el tamaño de los datos de entrada y su tipo de dato. Este método devuelve un

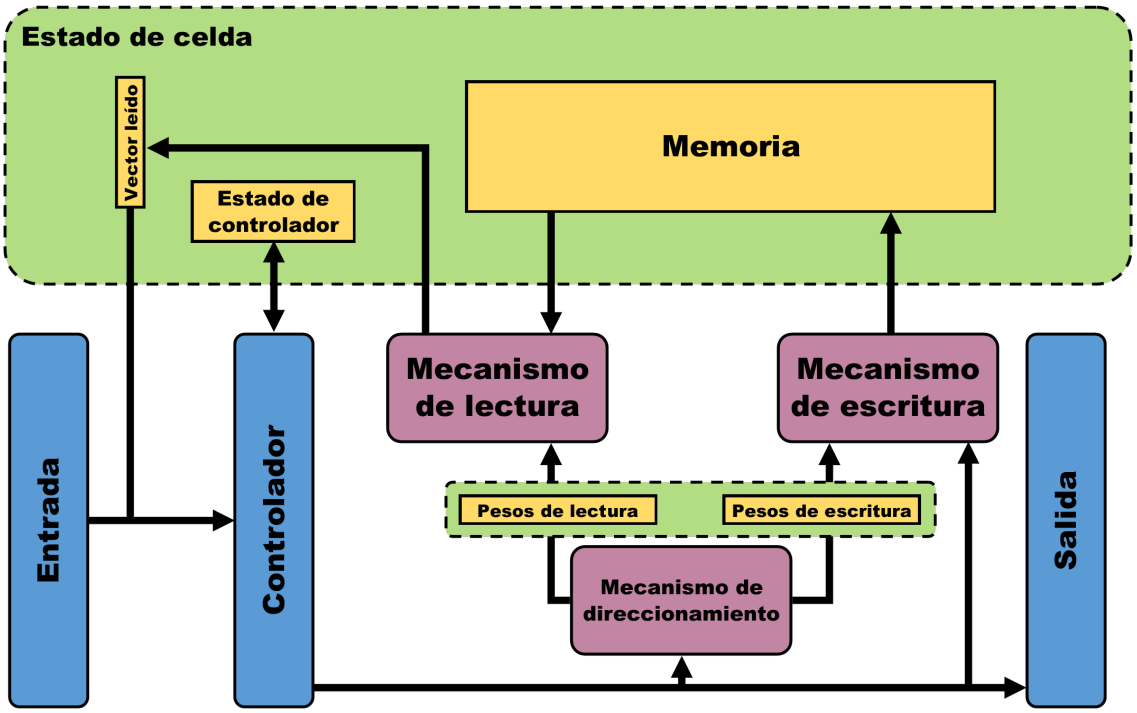


Figura 4.1: Estructura a bajo nivel de una NTM: A partir de las entradas y el vector leído de la memoria, el controlador compone una salida y da órdenes al mecanismo de direccionamiento para que genere pesos de lectura y escritura, que actualizan la memoria y el vector leído. La zona verde se corresponde con la información almacenada en el estado de celda.

tensor de la forma *state_size* con los valores del estado de celda para la primera iteración. Si no se implementa este método, se crea automáticamente un estado inicial en el que todos los valores son 0.

Como cualquier clase, también cuenta con el método *__init__*, en el que se declaran todos los componentes como capas internas o atributos. Además, la clase más general *Layer* dispone del método *build*, que se ejecuta antes de la primera llamada a *call* del modelo y se usa para inicializar los componentes que necesitan conocer el tamaño concreto de la entrada.

En el apéndice A se encuentra el código que implementa todos estos métodos y atributos.

Una vez se dispone de una celda RNN , la clase *RNN* de Keras se encarga de hacer de interfaz para el entrenamiento y la evaluación del modelo e implementa el bucle de tiempo en el que se realizarán las llamadas a *build* y *call*.

4.1. Controlador

Como decíamos, se puede escoger cualquier arquitectura para el controlador, por ejemplo [2] propone una doble capa de LSTM . En nuestro caso, el controlador que escogemos es una instancia de

SimpleRNNCell; es decir, una capa RNN estándar. Hemos elegido una RNN básica en lugar de algo más sofisticado como una LSTM para evitar que sea el propio controlador el que recuerde los datos en lugar de tener que usar la memoria. De esta manera forzamos a que la red escriba y lea de la memoria cuando deja de ser capaz de resolver problemas sólo con las neuronas.

Puesto que nuestro controlador será una capa recurrente, necesitaremos concatenar su estado interno de cada paso a su entrada del paso siguiente. A nivel de la NTM , esto quiere decir que el estado del controlador forma parte del estado de la NTM , ya que esa es la información que se transmite a los siguientes pasos temporales.

Como comentábamos también, el controlador es el que produce los parámetros de las ecuaciones en los apartados 3.1, 3.2 y 3.3. Lo que quiere decir esto es que la salida de la capa recurrente del controlador está conectada a una capa densa por cada uno de los parámetros de estas ecuaciones, de los tamaños con los que se definen. Las salidas de estas capas son lo que usan los mecanismos de direccionamiento, lectura y escritura para hacer sus cálculos. Para forzar que las salidas de estas capas estén en el rango que corresponde a los parámetros, es importante escoger bien sus funciones de activación. En nuestro caso, seguimos los pasos de [2] y utilizamos la sigmoide para e_t y g_t, tangente hiperbólica para a_t y k_t, *softplus* para β_t y *softmax* para s_t y γ_t, aunque a γ_t se le suma 1 para desplazar todo el intervalo por encima de 1.

Como nota adicional, en principio la NTM podría tener varios cabezales de lectura o escritura. Esto querría decir que en cada paso de tiempo la red leería o escribiría varias veces, por lo que necesitaría tantas veces más de estos parámetros. Pongamos como ejemplo que contáramos con un cabezal de lectura y dos de escritura. En ese caso, se escribiría dos veces en la memoria, por lo que necesitaríamos generar dos vectores de pesos. Para esto, necesitamos también dos de cada uno de los parámetros de las ecuaciones del apartado 3.1 (k_t, β_t, g_t, s_t y γ_t). Además, para escribir con esos pesos también necesitamos dos vectores de adición (a_t) y borrado (e_t). Por tanto, por cada uno de estos parámetros habría dos capas que producirían dos versiones distintas de ellos. En nuestra implementación, sin embargo, usaremos un solo cabezal de escritura y uno solo de escritura.

Respecto al número de unidades, para el problema que resolveremos (ver apartado 4.3) es conveniente que el controlador sea capaz de transmitir la entrada y el vector leído a las siguientes capas, por lo que decidimos tener suficientes neuronas para hacer esto de manera sencilla. Concretamente el controlador en nuestras pruebas cuenta con 100 unidades.

4.2. Estado de celda

Hemos mencionado que nuestra *NTMCell* hereda de la *SimpleRNNCell* de Keras. Esto quiere decir que lo único que la clase propaga a los siguientes pasos temporales es el estado de la celda. Por supuesto, necesitamos que la memoria se conserve a lo largo de toda la ejecución y, como vemos en

el apartado 3.1, a cada paso necesitamos también los pesos de lectura y escritura del tiempo anterior para generar los del actual. En consecuencia, aparte del estado de celda del propio controlador, tanto la matriz que forma la memoria como los vectores de pesos de cada paso son lo que se considera el estado de celda de la *NTMCell*. Además, como en cada tiempo se lee un vector de la memoria que se concatena a la entrada de la red, también se incluye este vector leído en el estado de celda. Todos estos elementos son los que están representados en amarillo sobre fondo verde en la figura 4.1.

Precisamente por transmitirse de un tiempo al siguiente, es necesario decidir unos valores iniciales del estado de celda que se recibirán en el primer paso, ya que este no tiene un paso anterior. Por tanto, hay que decidir valores iniciales tanto para la memoria como para los pesos de lectura y escritura y el vector leído. Como argumentan en [2], esta decisión es importante y puede afectar al comportamiento de la red. Ellos prueban tres inicializaciones diferentes: todos los valores a una misma constante, valores aleatorios con una distribución normal y valores aprendidos por la propia red. Su conclusión es que la mejor manera de inicializar la memoria es fijar todas las posiciones a una constante pequeña. En el caso de nuestra implementación, el estado inicial se obtiene como salida del método *get_initial_state*, que por defecto Keras implementa como un método que inicializa todos los valores a 0. Realizaremos las pruebas con un método de implementación propia que inicializa la memoria y el vector leído a 10^{-6} y enfoca los pesos de lectura y escritura totalmente en la primera posición de memoria (1 en la primera posición y 0 en las demás).

4.2.1. Reflexión sobre la inicialización

La inicialización natural que uno utilizaría sin darle muchas vueltas es la que da Keras por defecto: todo 0. Sin embargo, en nuestro caso concreto, esta decisión aparentemente inocente tiene efectos inesperados.

Reflexionemos sobre cómo funciona el mecanismo de direccionamiento para comprender lo que pasaría con esta inicialización, comenzando con el acceso por contenido definido en la ecuación 3.2. Al ser 0 todas las posiciones de la memoria, la similitud coseno siempre da como resultado 0 al comparar cada posición de memoria con cualquier vector, lo que da como resultado un w_t^c en el que todas las componentes son $1/N$. A continuación vendría la ecuación 3.3, que hace una media ponderada entre un w_t^c con todas las componentes iguales y un w_{t-1} que hemos inicializado a 0. El resultado: w_t^g tiene todas las componentes iguales. De esta manera, transponerlo hacia delante y hacia atrás y sumarlo en la ecuación 3.4 sigue dándonos un $\tilde{w}_t$ con todas las componentes iguales. Finalmente, la ecuación 3.5 mantiene todos los pesos iguales pero haciendo que sumen 1, de tal forma que todas las componentes de los pesos generados w_t son $1/N$.

Que los pesos de lectura y escritura sean siempre los mismos es un problema, ya que evita que se pueda escribir y leer en posiciones distintas de memoria, reduciendo en la práctica el tamaño de la memoria a 1.

Esto es lo que ocurre en la primera iteración, en la que todo lo que hay en el estado de celda es 0. Podríamos pensar que con el tiempo este inconveniente se resolverá solo en algún momento; sin embargo, no es así. La situación ahora es que todas las posiciones de la memoria tienen el mismo valor, lo que de nuevo hace que la similitud coseno dé el mismo valor para todas y los pesos del acceso por contenido w_t^c sigan valiendo todos $1/N$ (ecuación 3.2). De nuevo, los pesos del tiempo anterior w_{t-1} tienen todos el mismo valor, por lo que su media ponderada con w_t^c (ecuación 3.3) tiene también el mismo valor en todas las componentes y el ciclo se repite.

De este argumento concluimos que hay dos problemas independientes que no podemos permitir que ocurran a la vez. Por un lado, si todas las posiciones de memoria tienen el mismo valor, el acceso por contenido es inútil, ya que no puede distinguir ninguna por encima de las demás. Por otra parte, si los pesos del tiempo anterior tienen todos el mismo valor, el acceso por posición puro ($g_t = 0$) también pierde su utilidad, ya que se vuelve incapaz de recuperar el enfoque en un número reducido de posiciones de memoria. Basta entonces con solucionar uno de estos problemas para que el modelo pueda funcionar.

Una posible solución es la inicialización alternativa que proponíamos antes. Al enfocar completamente los pesos a la primera posición de memoria, evitamos el segundo problema por completo. Elegimos evitar este y no el primero, ya que esperamos que la red utilice más el acceso por posición para resolver su tarea, que describimos a continuación.

4.3. Tarea: copiado

La tarea para la que realizaremos las pruebas es la de copiar una secuencia. La idea es enseñarle a la red una secuencia de caracteres, que terminará con un carácter especial de control. Este carácter marcará el momento en el que el modelo debe dejar de escuchar y comenzar a contestar. Lo que contestará será la misma secuencia de caracteres en el mismo orden. Es importante remarcar una diferencia: no le pedimos al modelo que dé como salida el carácter que acaba de recibir como entrada en cada tiempo, sino que queremos que lea toda la secuencia y entonces empiece a mostrarla por la salida, un carácter por paso.

Veamos el comportamiento deseado con un ejemplo. Supongamos que los caracteres que estamos usando son los 27 del alfabeto español, y el carácter de control será el símbolo “$”. Entonces, el comportamiento deseado para la entrada “holamundo” es el ilustrado en la tabla 4.1.

t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
Entrada	h	o	l	a	m	u	n	d	o	$								
Salida										h	o	l	a	m	u	n	d	o

Tabla 4.1: Ejemplo de tarea de copiado.

Al usar este sistema con el carácter de control, el modelo se ve obligado a almacenar la cadena durante un tiempo indefinido y después recuperarla en orden. El objetivo es que aprenda a usar la memoria para escribir en ella los caracteres hasta que aparezca el de control y los lea después para sacarlos como salida.

En general, esta clase de problemas son complicados de resolver para una LSTM . El hecho de no tener que memorizar un número fijo de caracteres, sino tener que almacenar un número arbitrario de ellos hasta que aparezca el de control hace complicado para la red que almacene esta información ordenada en el estado interno o que habilite neuronas dedicadas a recordarla. Como se puede ver en las pruebas de [1], una LSTM tarda bastante más que una NTM en resolver el problema de forma relativamente consistente y además no llega a perfeccionarlo.

En la práctica, lo que hasta ahora hemos llamado caracteres lo codificamos como vectores de 9 bits. De estos bits, 8 componen el carácter como tal y el último se usa como bit de control que siempre está a 0. El bit de control sólo se activa cuando se introduce el carácter de control, en el que los 9 bits son 1.

4.4. Función de coste

Para evaluar el desempeño del modelo en esta tarea, tenemos que elegir también una función de coste adecuada. En realidad, podemos pensar en el problema como uno de multi-etiquetado: por cada entrada se decide si se le asigna o no cada una de las ocho etiquetas que son los bits de salida. Además, tener una etiqueta (tener un bit a 1) no indica nada sobre las demás, son independientes. Por tanto, lo podemos considerar como ocho problemas diferentes en los que se elige entre dos clases: ese bit es 0 o ese bit es 1.

La función de coste habitual para evaluar problemas de clasificación binaria como estos es *binary crossentropy*, cuya ecuación es la siguiente:

$$\text{coste}(y, \tilde{y}) = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log(\tilde{y}_i) + (1 - y_i)\log(1 - \tilde{y}_i)) \,. \tag{4.1}$$

Aquí n representa el número de salidas (en nuestro caso 8), y_i es la componente número i del vector de etiquetas (la salida correcta) e $\tilde{y}_i$ es la componente número i del vector de predicción del modelo (la salida del modelo). Si miramos la ecuación con atención, podemos ver que en realidad para cada i sólo uno de los sumandos sobrevive, ya que o bien $y_i = 0$ o bien $1 - y_i = 0$ y eso hace que uno de los dos se anule. Imaginemos ahora que definimos nuevas variables que tomen los valores que sobreviven; es decir,

$$\tilde{y}'_i = \begin{cases} \tilde{y}_i & \text{si } y_i = 1 \\ 1 - \tilde{y}_i & \text{si } y_i = 0 \end{cases} .$$

De esta manera, $\tilde{y}'_i = 1$ indica que la predicción $\tilde{y}_i$ era correcta e $\tilde{y}'_i = 0$, que era incorrecta. Reescribiendo la ecuación 4.1, queda

$$\text{coste}(y, \tilde{y}) = -\frac{1}{n}\sum_{i=1}^{n} \log(\tilde{y}'_i) \,. \tag{4.2}$$

Interpretemos esta ecuación ignorando el $\frac{1}{n}$ por el momento. Para cada i estamos tomando $-\log(\tilde{y}'_i)$. Como sabemos, $-\log(1) = 0$, cual nos da un coste cercano a 0 cuando la predicción está cerca de la correcta. En cambio, si $\tilde{y}'_i$ se acerca a 0 (la predicción está muy equivocada) entonces $-\log(\tilde{y}'_i) \to \infty$ y tenemos un coste muy elevado. Exactamente lo que buscamos. Estamos calculando entonces este valor (que no es otro que la *categorical crossentropy* para dos clases) para cada etiqueta, y por último tomamos la media de todos estos valores para componer el coste final.

En nuestras pruebas, utilizaremos dos funciones de coste distintas basadas en *binary_crossentropy*: una valorará positivamente que la red responda con 0 a todas las entradas hasta el carácter de control y después repita la secuencia, y la otra ignorará la salida de la red antes del carácter de control y sólo la tendrá en cuenta cuando empiece a repetir la secuencia.

4.5. Pruebas

En esta sección describiremos los dos tipos de pruebas que hemos realizado sobre el modelo. Por un lado, queremos probar la parte *feed-forward* del modelo: que con unos pesos adecuados, dada una entrada es capaz de conseguir la salida adecuada. Por otra parte, probaremos también el aprendizaje: que partiendo de unos pesos aleatorios, la red es capaz de ajustarlos para que resuelvan el problema.

4.5.1. Pesos diseñados

Puesto que hemos cambiado un poco el diseño original (especialmente el controlador), un ejercicio interesante es diseñar unos pesos que resuelvan el problema de manera ideal. Esto puede servir para comprobar que esta arquitectura sigue siendo capaz ya no de aprender a resolver el problema, sino de resolverlo; es decir, que existe una solución óptima a la que el modelo aspira a llegar. Además, a la vez puede ayudar a tomar algunas de las decisiones que planteábamos antes, como ciertas funciones de activación o tamaños de capas. Por ejemplo, para los pesos que describiremos a continuación es importante que haya al menos tantas unidades del controlador como bits de entrada más vector leído, y también es conveniente que tenga función de activación sigmoide. Hay que decir que estos pesos están diseñados para la inicialización que comienza con los cabezales de lectura y escritura en la primera posición de la memoria y la función de coste que sólo tiene en cuenta la segunda mitad de la salida.

La estrategia general será la siguiente, diferenciada claramente en dos fases. En la primera fase, hasta el carácter de control, queremos que la red escriba las entradas en distintas posiciones de la memoria, preferiblemente adyacentes. Recordemos que la manera de decidir lo que se escribe es producir los vectores de borrado (e_t) y adición (a_t). Estos vectores los producen las capas conectadas a la salida del controlador, por lo que necesitamos que el controlador sea capaz de dar como salida los 8 bits del carácter de entrada sin modificar (en principio la red podría comprimir la información en menos neuronas y ser capaz de recuperarla después, pero por simplicidad lo pensaremos así). El vector de adición será entonces una copia de estos 8 bits y todas las componentes del de borrado deberán ser 1 para que se limpie por completo la posición de memoria antes de la suma. Durante esta fase, el cabezal de escritura irá avanzando una posición de memoria cada paso de tiempo. Esto lo conseguiremos a través del acceso por posición, concretamente usando s_t. En la segunda fase, cuando se lea el carácter de control, comenzaremos a leer de la memoria desde la primera posición en la que escribimos, avanzando también una posición de memoria cada paso de tiempo. Este vector leído será exactamente lo que querremos dar como salida. Puesto que el vector leído se concatena a la entrada del controlador, que a su vez está conectado a la salida, necesitaremos de nuevo que el controlador copie el vector leído sin modificarlo, para que la salida pueda copiarlo también.

Con esto en mente, pasemos entonces por encima de los pesos de la *NTMCell*:

Controlador Como ya hemos visto, necesitamos que el controlador copie sus entradas (la entrada del modelo y el vector leído) a sus neuronas para que las siguientes capas puedan acceder a ellas. Como las salidas representan bits, irán entre 0 y 1 y elegiremos la sigmoide como función de activación. Sin tener en cuenta la función de activación, querríamos que los pesos fueran algo parecido a la matriz identidad, pero a causa de la sigmoide para conseguir un 1 en la neurona necesitamos un valor muy alto, y para conseguir un 0 necesitamos un valor igualmente alto pero negativo. Lo que haremos entonces será que los pesos sean la matriz identidad (recortada al tamaño adecuado) pero con 20 en lugar de 1 y todo el vector de *bias* será -10. De esta manera a cada entrada se la multiplica por 20 y se le resta 10; las que son 1 dan como resultado 10 y las que son 0 dan resultado -10 y tras aplicar la función de activación quedan 1 y 0, respectivamente.

Respecto a la parte recurente, lo único que nos interesa recordar es si estamos en la fase de lectura o de escritura, y esto nos lo indica el bit de control de la entrada del modelo. Este bit es 0 en todo momento salvo en el cambio de fase, cuando llega el carácter de control y se vuelve 1. A partir de este momento queremos que la neurona correspondiente a ese bit siga valiendo 1. Para lograr esto, basta con que los pesos recurrentes sean todos 0 salvo el que conecta esta "neurona de control" consigo misma. Este peso concreto será 20, siguiendo la misma lógica que antes.

El código 4.1 muestra concretamente los valores de cada parámetro (*kernel* representa la matriz de pesos de la capa).

Código 4.1: Pesos diseñados para el controlador.

```
kernel = np.eye(ninputs+memposlen,units,dtype=dtype)
kernel = kernel *20

recurrent_kernel = np.zeros((units,units),dtype=dtype)
recurrent_kernel[ctrl,ctrl] = 20.

bias = np.ones(units,dtype=dtype)
bias = bias *(-10.)
```

Direccionamiento El mecanismo de direccionamiento es el que decide dónde se escribe y de dónde se lee en cada paso. Lo que queremos es que hasta que llegue el carácter de control, el cabezal de escritura vaya avanzando de una posición en una posición y el de lectura se quede fijo en la primera. Una vez llegue el carácter de control, el cabezal de escritura se detendrá y el de lectura comenzará a moverse al mismo ritmo. Para hacer esto no necesitamos para nada el acceso por contenido, por lo que no nos importarán los valores de los parámetros k_t, β_t ni γ_t. Para usar siempre el acceso por posición e ignorar estos parámetros, el valor de g_t siempre deberá ser 0, lo que conseguimos con un peso de 0 y un *bias* de -10 (por la función de activación) tanto para lectura como para escritura (recordemos que este mecanismo esta duplicado: uno produce direcciones de escritura y otro de lectura). Ahora todo depende de la elección de s_t que hagamos, que será distinta para el caso de lectura y para el de escritura.

Direccionamiento de escritura Como hemos planteado, queremos que el cabezal de escritura avance mientras aún no haya llegado el carácter de control; es decir, mientras la neurona de control tenga valor 0. En ese caso, s_t debería ser el vector $(0, 0, 1)$, que indica que no retrocede, no queda en la misma posición pero sí avanza. Podríamos hacer que se detuviera el cabezal cuando llegara el carácter de control, pero en realidad no importa que siga avanzando ya que no vamos a volver a escribir. Por simplicidad, fijamos que el *bias* s_t valga siempre $(0, 0, 10)$, como se puede comprobar en el código 4.2. Al aplicar la función de activación, este vector se transforma en $(0, 0, 1)$

Código 4.2: Pesos diseñados para el mecanismo de direccionamiento de escritura.

```
s_w_kernel = np.zeros((units,3),dtype=dtype)
s_w_bias = np.zeros(3,dtype=dtype)
s_w_bias[2] = 10.
```

Direccionamiento de lectura La lectura es el caso contrario: debe detenerse el cabezal ($s_t = (0, 1, 0)$) mientras la neurona de control sea 0, y después debe

comenzar a avanzar ($s_t = (0, 0, 1)$) mientras la neurona de control valga 1. Teniendo en cuenta la función de activación (esta vez *softmax*) y resolviendo un par de ecuaciones lineales sencillas, la conclusión a la que llegamos es que $s_t[1] = -10c + 10$ y $s_t[2] = 10c$, donde c es el valor de la neurona de control. Se pueden ver los valores concretos asignados a cada peso en el código 4.3.

Código 4.3: Pesos diseñados para el mecanismo de direccionamiento de lectura.

```
1  s_r_kernel = np.zeros((units,3),dtype=dtype)
2  s_r_kernel[ctrl,1] = -10.
3  s_r_kernel[ctrl,2] = 10.
4  s_r_bias = np.zeros(3,dtype=dtype)
5  s_r_bias[1] = 10.
```

Parámetros de escritura A nivel de escritura no nos hace falta distinguir la primera fase de la segunda, podemos escribir siempre la entrada del modelo en la memoria mientras tengamos espacio suficiente. Esto nos da la restricción de que la memoria tenga al menos tantas posiciones como la longitud de la cadena de entrada. Teniendo esto en mente, en todos los pasos querremos borrar por completo la posición de memoria que indique el mecanismo de direccionamiento, y por tanto e_t será siempre 1 en todas sus componentes. Esto lo conseguimos con 0 en todos los pesos y 10 en el *bias*, de tal forma que dé un número muy cercano a 1 al pasar por la función de activación. Para el vector de activación a_t seguimos una lógica similar a la de los pesos del controlador: como queremos copiar los primeros 8 bits de la entrada, los pesos serán una matriz con sólo 20 en la diagonal hasta la octava fila y el resto 0. El *bias* será también -10 pero sólo hasta la octava componente, como se puede comprobar en el código 4.4.

Código 4.4: Pesos diseñados para los parámetros del mecanismo de escritura.

```
1  e_kernel = np.zeros((units,memposlen),dtype=dtype)
2  e_bias = np.ones(memposlen,dtype=dtype)
3  e_bias = e_bias *10
4
5  a_kernel = np.eye(units,memposlen,dtype=dtype)
6  a_kernel[ctrl:,:] = 0.
7  a_kernel = a_kernel *10
8  a_bias = np.zeros(memposlen,dtype=dtype)
```

Salida La capa de salida simplemente debe copiar el vector leído, ya que la respuesta que querremos dar siempre saldrá de la memoria. Bastará con coger una matriz identidad desplazada las posiciones adecuadas. De nuevo, deberemos tener en cuenta la función de activación y cambiar los 1 de la matriz identidad por 20 y poner un *bias* fijo de -10.

En el código 4.5 podemos ver de manera concreta cómo se asignan los valores a estos pesos.

Código 4.5: Pesos diseñados para la capa de salida.

```
1 output_kernel = np.eye(units,noutputs,dtype=dtype)
2 output_kernel = np.roll(output_kernel,ctrl+1,axis=0)
3 output_kernel = output_kernel *20
4 output_bias = np.full(noutputs,-10.,dtype=dtype)
```

4.5.2. Aprendizaje

El objetivo de estas pruebas es analizar la capacidad del modelo para aprender a resolver el problema partiendo de unos pesos aleatorios.

El modelo de estas pruebas contará con un controlador de 100 unidades y una memoria con 128 posiciones de 20 bits cada una. En cada época se entrenará con 200 *batches* de 10 secuencias cada uno, y cada secuencia será de 20 caracteres de longitud (41 teniendo en cuenta el carácter de control y el tiempo de respuesta).

El optimizador elegido para las pruebas es Adam [19], una variante del descenso por gradiente explicado en el apartado 2.1.3 que ajusta la tasa de aprendizaje (*learning rate*) durante el entrenamiento. La tasa de aprendizaje inicial será 10^{-3} y se recortarán los gradientes a un valor de 1. Esto quiere decir que si el valor de una componente del gradiente supera 1, se sustituye por 1.

Las funciones de coste serán las descritas al final del apartado 4.4 y la función de precisión será *binary_accuracy* restringida a la respuesta del modelo. Esta última simplemente mide el porcentaje de veces que la salida del modelo coincide con las etiquetas, interpretando los valores superiores a 0.5 como 1 y los inferiores como 0.

Tras cada época, se almacena tanto la media de los costes como la media de las precisiones al entrenar en cada secuencia de la época. Esto nos da el coste y la precisión de entrenamiento de la época. Además, también tras cada época, se evalúa el modelo en 640 secuencias de prueba que sirven para comprobar su desempeño fuera del conjunto de entrenamiento. Las medias de los resultados de coste y precisión de estas evaluaciones nos dan el coste y la precisión de prueba.

Visualizaremos los resultados con gráficas de los costes y las precisiones de entrenamiento y de prueba frente al número de épocas. En un escenario ideal, los costes deberían acercarse a 0 cuando aumentara el número de pruebas y las precisiones deberían acercarse a 1.

5 Resultados

En este capítulo visualizaremos y comentaremos los resultados de las pruebas descritas en el apartado 4.5.

5.1. Pesos diseñados

Tras construir un modelo y cargarle los pesos diseñados en el apartado 4.5.1, lo evaluamos en varias secuencias y observamos que el modelo proporciona el resultado esperado. Se intuía que funcionaría correctamente ya que no surgieron contratiempos a nivel teórico a la hora de diseñar los pesos, pero era necesario comprobarlo empíricamente.

La figura 5.1 muestra un ejemplo de evaluación del modelo en una secuencia. La primera imagen representa la salida del modelo con los pesos diseñados, seguida de la salida correcta y la diferencia entre las dos. Cada franja horizontal representa los 8 bits del carácter y el eje vertical avanza en el tiempo de arriba a abajo.

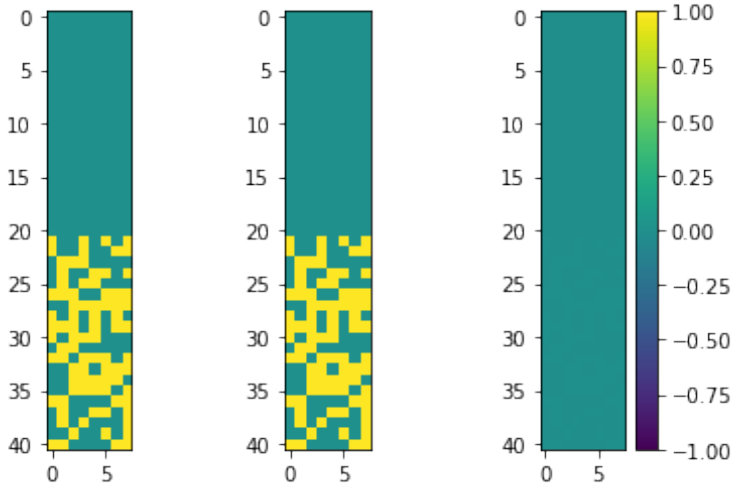
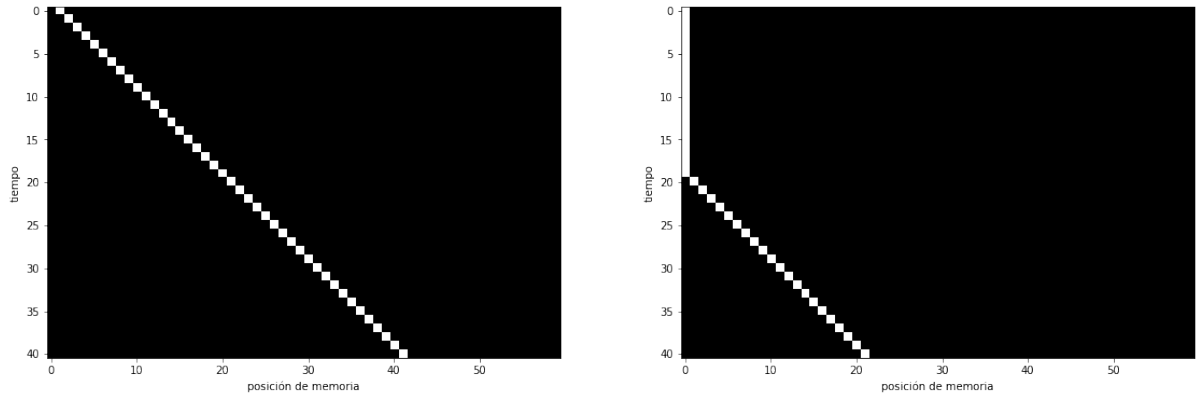


Figura 5.1: Evaluación del modelo con pesos diseñados.

Este comportamiento de la red demuestra que efectivamente el modelo es capaz de resolver el

problema con los pesos adecuados. Esto quiere decir que potencialmente podría llegar a aprender a resolver el problema por sí mismo partiendo de pesos aleatorios.

Veamos ahora cómo utiliza el modelo la memoria. Como esperábamos, en la figura 5.2(a) podemos ver que el cabezal de escritura comienza a avanzar desde el principio, mientras que el cabezal de lectura de la figura 5.2(b) queda fijo. Una vez llega el carácter de control en el tiempo 20, el cabezal de escritura pasa de largo, lo cual no afecta, y el de lectura comienza a moverse y leer secuencialmente las posiciones de memoria que se escribieron antes. En la figura 5.3 se muestran los contenidos de la memoria en el momento en que el modelo recibe el carácter de control. Estos coinciden exactamente con los caracteres de entrada. El modelo comienza a escribir en la posición 1 de memoria en lugar de la 0 debido a que por diseño el cabezal primero se desplaza y luego actualiza la memoria.



(a) Cabezal de escritura

(b) Cabezal de lectura

Figura 5.2: Evolución de los cabezales de lectura y escritura a lo largo del tiempo en el modelo con pesos diseñados. El color negro representa 0 y el blanco 1. Para facilitar la visualización, los pesos están recortados hasta el 60, los que han sido omitidos eran todos 0.

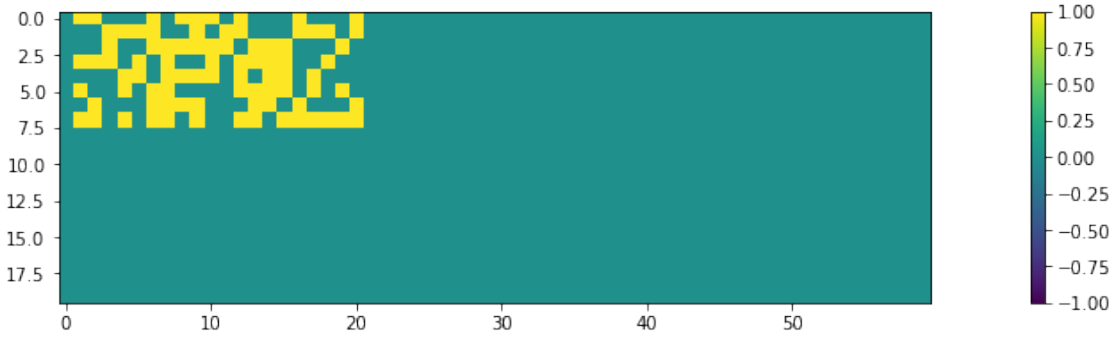


Figura 5.3: Contenido de la memoria tras la primera fase del modelo con pesos diseñados. Para facilitar la visualización, se muestran sólo las posiciones de memoria hasta la 60. El contenido de las omitidas era 0 en todos los casos.

5.2. Aprendizaje

Veremos ahora los resultados de las pruebas de aprendizaje del modelo.

5.2.1. Coste de salida completa

Realizamos las pruebas descritas en el apartado 2.1.3 con la función de coste que tiene en cuenta la salida completa, incluida la primera fase en la que el modelo está escribiendo en memoria. La evolución del coste (*loss*) y la precisión (*accuracy*) del modelo se muestran en la figura 5.4.

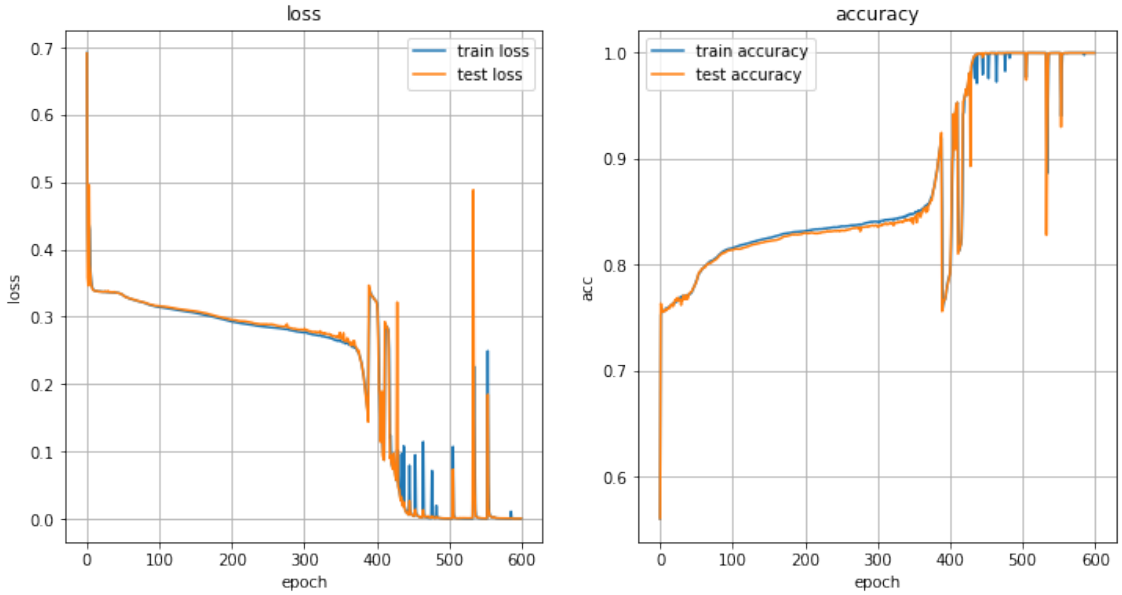


Figura 5.4: Evolución de coste y precisión en entrenamiento y pruebas para el modelo que usa la salida completa para el coste.

Como se puede observar, sólo después de unas 500 épocas consigue el modelo reducir a 0 el coste y aumentar la precisión a 1. Alrededor del 0.3 de coste y 0.83 de precisión, el modelo tiene un momento en el que claramente le cuesta más seguir aprendiendo. Esto puede deberse a que el modelo se enfoca demasiado en responder 0 durante la primera fase y esto hace que le cueste corregir durante la segunda. Sin embargo, aun con esta limitación es capaz de acertar el aprender a resolver el problema por completo. Comparémoslo ahora con la otra función de coste.

5.2.2. Coste de final de la salida

Comenzando en unos pesos aleatorios, realizamos 200 épocas de entrenamiento con el modelo descrito en el apartado 2.1.3 y la función de coste que sólo tiene en cuenta la segunda mitad de la salida. La figura 5.5 ilustra la evolución del coste (*loss*) y la precisión (*accuracy*) del modelo a lo largo

del tiempo.

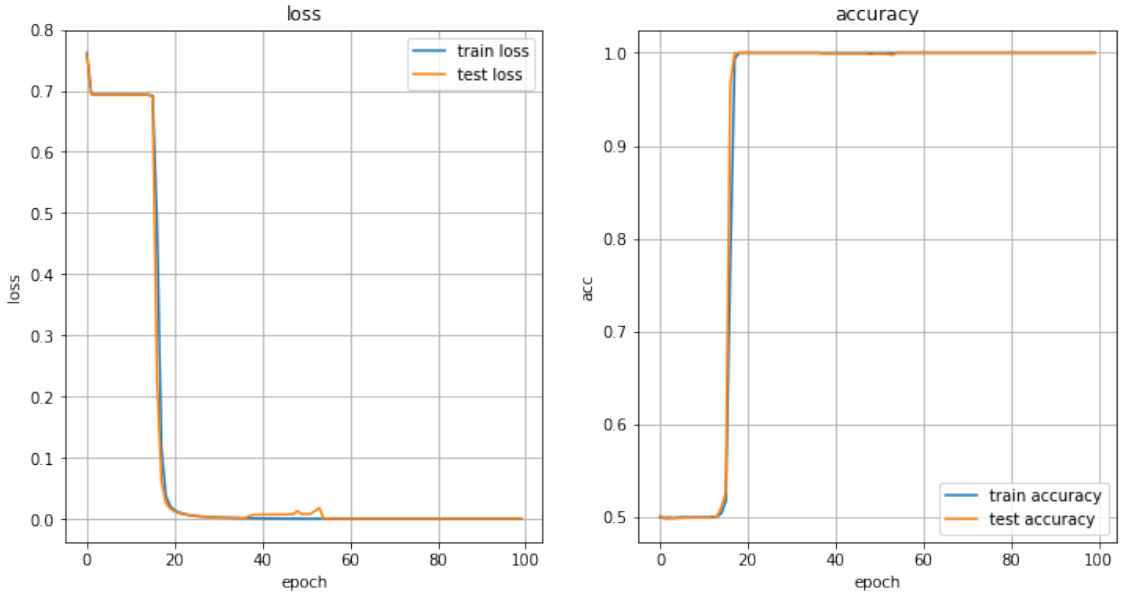


Figura 5.5: Evolución de coste y precisión en entrenamiento y pruebas para el modelo cuya función de coste sólo tiene en cuenta la segunda mitad de la salida.

Como se puede ver, antes incluso de llegar a las 200 épocas (alrededor de las 30) el modelo ya consigue un coste de 0 y una precisión del 100 %. Queda claro que el hecho de sólo tener en cuenta la parte de la salida en la que la red responde facilita en gran medida el aprendizaje del modelo.

En la figura 5.6 podemos ver la evaluación de este modelo entrenado sobre una secuencia. De nuevo, la primera imagen representa la salida del modelo con los pesos aprendidos, seguida de la salida correcta y la diferencia entre las dos. Cada franja horizontal representa los 8 bits del carácter y el eje vertical avanza en el tiempo de arriba a abajo.

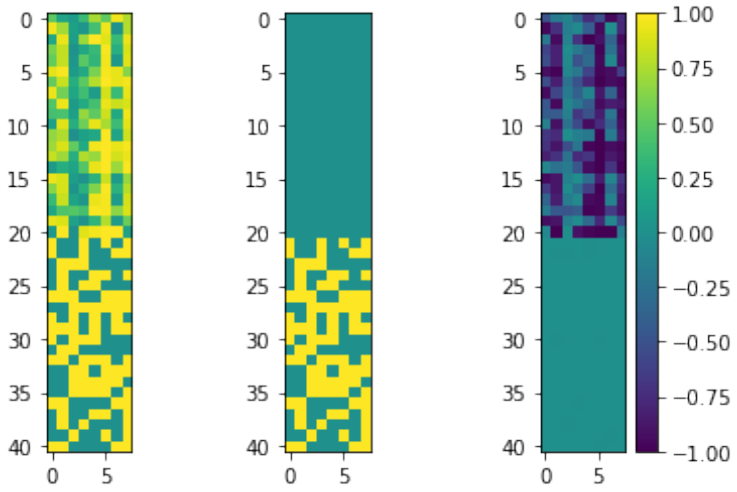


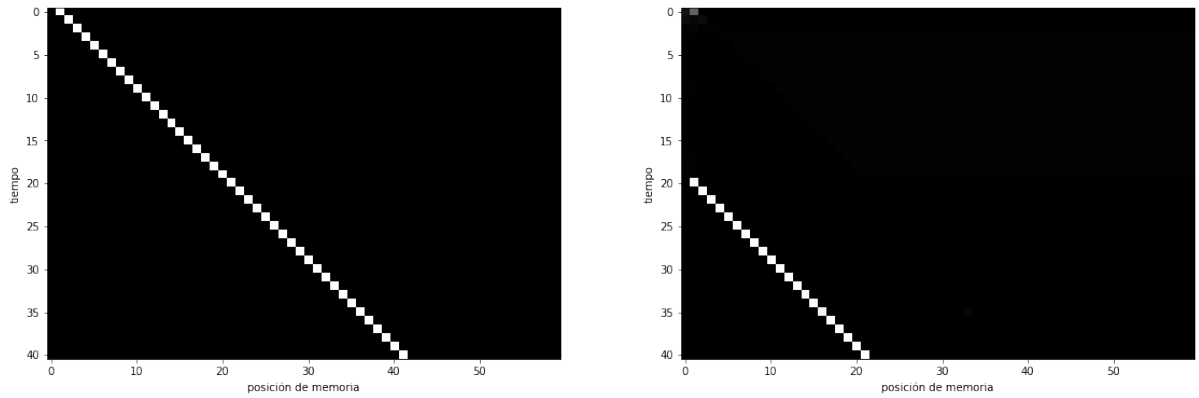
Figura 5.6: Evaluación del modelo entrenado.

Si bien los caracteres hasta el 20 no son todos 0 como en la salida esperada, al no tenerlos en cuenta la función de coste, esto no afecta al entrenamiento del modelo. En cambio las salidas del 21 al 40 son exactamente las mismas, lo que encaja con la precisión de 100 % del modelo.

Observemos cómo actúan los cabezales de lectura y escritura sobre la memoria.

El comportamiento del cabezal de escritura es más o menos el mismo que en la prueba con pesos diseñados. En la figura 5.7(a) podemos ver que al principio avanza de posición en posición escribiendo en la memoria hasta que aparece el carácter de control en el tiempo 20, y entonces deja de ser relevante mientras no sobrescriba las posiciones anteriores.

Como muestra la figura 5.7(b), el cabezal de lectura también hace lo que esperaríamos: a partir del tiempo 20 comienza a leer las posiciones que se escribieron en la primera fase. Sin embargo, hay una pequeña variación con respecto a la solución con pesos diseñados del apartado 5.1. En lugar de fijar el cabezal de lectura en su valor inicial (la primera posición de memoria) hasta que llegue el momento de leer, lo que hace es apagarlo por completo (poner todos los pesos de lectura a 0) y volver a encenderlo cuando llega el momento. Esto resulta curioso porque al apagarlo, ya no tiene manera de acceder a la primera posición de memoria utilizando acceso por posición. Eso quiere decir que el modelo debe estar usando acceso por contenido para recuperar esa primera posición de memoria en el tiempo 20.



(a) Cabezal de escritura

(b) Cabezal de lectura

Figura 5.7: Evolución de los cabezales de lectura y escritura a lo largo del tiempo en el modelo con pesos entrenados. El color negro representa 0 y el blanco 1. Para facilitar la visualización, los pesos están recortados hasta el 60. Los que han sido omitidos eran todos 0.

Resulta sorprendente lo mucho que se parece la estrategia que aprende el modelo a la que diseñamos anteriormente. Para empezar, tanto los pesos de escritura como los de lectura están completamente enfocados en una única posición de memoria en cada paso, no hace ninguna escritura ni lectura borrosas. Por otra parte, empieza a escribir en la primera posición de memoria y continúa con posiciones adyacentes hacia delante. Que empiece en la primera posición tiene una explicación sencilla: es como inicializamos los pesos. En cambio, que utilice posiciones adyacentes de memoria sí es más

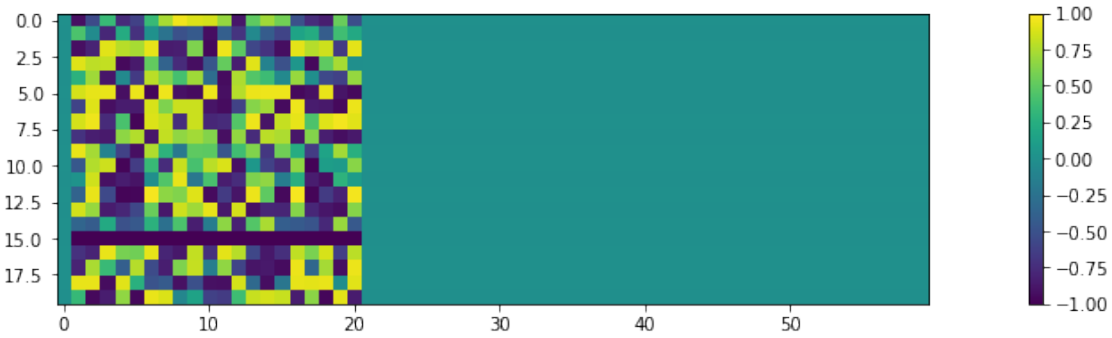


Figura 5.8: Contenido de la memoria tras la primera fase del modelo con pesos entrenados. Para facilitar la visualización, se muestran sólo las posiciones de memoria hasta la 60. El contenido de las omitidas era 0 en todos los casos.

digno de mención, ya que, aunque sólo permitimos saltos de -1, 0 o 1 posiciones, en ningún momento se desvía de los saltos de 1, ni tampoco se deja distraer por el acceso por contenido. Para terminar, si bien es cierto que los contenidos de la memoria de la figura 5.8 son ilegibles a simple vista, esto se debe a que el modelo codifica cada entrada de cierta manera que le permite recuperar la información más adelante. Concluimos que este modelo es capaz de desarrollar algoritmos sencillos a partir de ejemplos de entrada-salida.

6

Conclusiones

Hemos realizado una implementación de la NTM propuesta por Graves [1] partiendo de cero, tomando las decisiones de diseño que quedan abiertas con el problema de copiado en mente. Este es un problema que resulta complejo para las arquitecturas tradicionales. Sin embargo, hemos comprobado que nuestro modelo no sólo tiene la estructura necesaria para resolver este problema, sino que además es capaz de aprender a resolverlo partiendo de una inicialización aleatoria de sus pesos. Esto deja claro que el modelo puede aprender a acceder a la memoria de manera secuencial, por lo que puede implementar estructuras sencillas como pilas o colas.

Como explican en [20], el potencial de esta arquitectura radica en su capacidad para separar el cómputo del almacenamiento de información. En este sentido, la estructura de las NTM es similar a la de un ordenador. La memoria de la red cumple la misma función que la memoria de acceso aleatorio (RAM): almacena la información que ordena el controlador. Por su parte, el controlador de la red actúa como el procesador, decidiendo qué operaciones se hacen con las entradas y qué se lee y escribe en la memoria. Además, al ser el controlador una red recurrente de por sí, también dispone de su estado de celda, que podríamos comparar a los registros internos del procesador.

En cierta manera, podríamos pensar que la ausencia de esta separación es lo que limita el resultado de diseños como las LSTM o GRU , que mezclan el procesamiento con su manera de conservar información. Sin embargo, ya que estos modelos de aprendizaje profundo vienen demostrando resultados muy prometedores en tareas que no requieren demasiada memoria, el hecho de añadirle una memoria externa complementa perfectamente sus carencias sin afectar a sus virtudes. Además, como hemos podido comprobar en las pruebas, el hecho de separar estos dos aspectos puede ayudar a esclarecer el comportamiento interno de los modelos, que tradicionalmente se han venido viendo como cajas negras que de alguna manera producen resultados que funcionan.

6.1. Siguientes pasos

El primer paso que viene a la cabeza es probar el modelo en otros problemas, comenzando por los propuestos en [1]. Uno de ellos es el mismo problema de copiado salvo que se pide al modelo que

repita la salida un número determinado de veces que se le indica como entrada. Con esto se pretende comprobar si el modelo puede aprender funciones anidadas e implementar bucles. Otra tarea que proponen es la llamada *associative recall*. Se le presenta a la red una serie de patrones y posteriormente se realiza una consulta con uno de esos patrones; el modelo debe responder con el siguiente patrón de la secuencia. Con esto se consigue probar la capacidad del modelo para almacenar objetos que "apuntan" a otros, como en una lista.

Otra idea interesante es comprobar la capacidad de generalización para estos problemas. En el caso del problema de copiado hemos comprobado que el modelo decide cómo mover los cabezales relativamente a la posición a la que los movió en el paso de tiempo anterior. Esto quiere decir que esta decisión está localizada, no le afecta directamente lo que ocurra en otros pasos de tiempo ni lo que ocurra en otras posiciones de memoria. Por tanto, sería interesante comprobar si la red es capaz de resolver el mismo problema para cadenas más largas sin necesidad de reentrenarla. De la misma manera, se podría estudiar la posibilidad de aumentar el tamaño de la memoria una vez entrenada la red, lo que permitiría ajustar a posteriori los recursos que necesita para su ejecución.

Por último, otro aspecto que queda pendiente de explorar es el controlador. Hasta ahora hemos realizado nuestras pruebas con una única capa recurrente como controlador, con buenos resultados. Sería interesante investigar cómo y cuánto afectaría utilizar como controlador una red más profunda, con otro tipo de arquitectura como una LSTM o una combinación de las dos.

Bibliografía

[1] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines," oct 2014.

[2] M. Collier and J. Beel, "Implementing Neural Turing Machines," jul 2018.

[3] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, pp. 471–476, oct 2016.

[4] E. R. Kandel, J. H. Schwartz, and T. M. Jessell, *Principles of Neural Science, fourth addition*. 2000.

[5] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, 1958.

[6] C. Lemaréchal, "Cauchy and the Gradient Method," *Documenta Mathematica*, vol. ISMP, pp. 251–254, 2012.

[7] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944.

[8] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *Advances in Neural Information Processing Systems 20 - Proceedings of the 2007 Conference*, 2009.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, 1986.

[10] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[11] J. F. Kolen and S. C. Kremer, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies," in *A Field Guide to Dynamical Recurrent Networks*, IEEE, 2009.

[12] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pp. 1724–1734, 2014.

[14] M. Collier, "Memory-Augmented Neural Networks for Machine Translation," 2019.

[15] C. Gulcehre and S. Chandar, "Memory Augmented Neural Networks for Natural Language Processing," *EMNLP-Tutorial*, 2017.

[16] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom, "Learning to transduce with unbounded memory," *Advances in Neural Information Processing Systems*, vol. 2015-Janua, pp. 1828–1836, 2015.

[17] A. Joulin and T. Mikolov, “Inferring algorithmic patterns with stack-augmented recurrent nets,” *Advances in Neural Information Processing Systems*, vol. 2015-Janua, pp. 190–198, 2015.

[18] F. Chollet *et al.*, “Keras.” `https://keras.io`, 2015.

[19] D. P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization,” *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pp. 1–15, 2015.

[20] H. Jaeger, “Deep neural reasoning,” *Nature*, vol. 538, pp. 467–468, oct 2016.

Acrónimos

GRU Gated Recurrent Unit.

LSTM Long Short-Term Memory.

MANN Memory-Augmented Neural Network.

NTM Neural Turing Machine.

RAM Random Access Memory.

ReLU Rectified Linear Unit.

RNN Recurrent Neural Network.

Apéndices

A

Implementación NTM

Este es el código de nuestra implementación propia de la celda NTM que utilizamos para las pruebas.

Código A.1: Código de la implementación de *NTMCell*.

```
import numpy as np
import tensorflow as tf
import tensorflow.keras.backend as K
import matplotlib.pyplot as plt

from collections import namedtuple

from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import SimpleRNNCell
from tensorflow.keras.layers import concatenate
from tensorflow.keras.losses import cosine_similarity
from tensorflow.keras.activations import softmax


def _normalize(x, axis=-1):
    return x/K.sum(x, axis=axis, keepdims=True)
```

```
18  class NTMCell(SimpleRNNCell):
19      def __init__(self, units, memsize, memposlen,
20                   shift=(-1,1), # tupla con los rangos del location addressing
21                   activation='tanh', use_bias=True,
22                   kernel_initializer='glorot_uniform',
23                   recurrent_initializer='orthogonal',
24                   bias_initializer='zeros',
25                   kernel_regularizer=None,
26                   recurrent_regularizer=None,
27                   bias_regularizer=None,
28                   kernel_constraint=None,
29                   recurrent_constraint=None,
30                   bias_constraint=None,
31                   dropout=0., recurrent_dropout=0., **kwargs):
32          super(NTMCell, self).__init__(units=units)
33          self.memsize = memsize
34          self.memposlen = memposlen
35          self.n_shift = shift[1] -shift[0] + 1
36
37          self.controller = SimpleRNNCell(units=units, activation=activation,
38                                          use_bias=use_bias,
39                                          kernel_initializer=kernel_initializer,
40                                          recurrent_initializer=recurrent_initializer,
41                                          bias_initializer=bias_initializer,
42                                          kernel_regularizer=kernel_regularizer,
43                                          recurrent_regularizer=recurrent_regularizer,
44                                          bias_regularizer=bias_regularizer,
45                                          kernel_constraint=kernel_constraint,
46                                          recurrent_constraint=recurrent_constraint,
47                                          bias_constraint=bias_constraint,
48                                          dropout=dropout, recurrent_dropout=recurrent_dropout,
49                                          **kwargs)
50
51          self.e = Dense(self.memposlen,activation='sigmoid',name='e')
52          self.a = Dense(self.memposlen,activation='tanh',name='a')
53          self.k_r = Dense(self.memposlen,activation='tanh',name='k_r')
54          self.k_w = Dense(self.memposlen,activation='tanh',name='k_w')
55          self.beta_r = Dense(1,activation='softplus',name='beta_r')
56          self.beta_w = Dense(1,activation='softplus',name='beta_w')
57          self.g_r = Dense(1,activation='sigmoid',name='g_r')
58          self.g_w = Dense(1,activation='sigmoid',name='g_w')
59          self.s_r = Dense(self.n_shift,activation='softmax',name='s_r')
60          self.s_w = Dense(self.n_shift,activation='softmax',name='s_w')
61          self.gamma_r = Dense(1,activation='softmax',name='gamma_r')
62          self.gamma_w = Dense(1,activation='softmax',name='gamma_w')
63
64          self.state_size = (tf.TensorShape(self.controller.state_size), memposlen, memsize, memsize,
                tf.TensorShape((memsize,memposlen)))
65          self.output_size = self.controller.output_size
66
67          Id = np.eye(self.memsize)
68          shift_range = range(shift[0], shift[1]+1)
69          shift_ids = [np.roll(Id, n, axis=-1) for n in shift_range]
70          self.shift_ids = tf.constant(np.array(shift_ids), dtype=kwargs['dtype'])
```

```
72      def get_initial_state(self, inputs=None, batch_size=None, dtype=None):
73          if inputs is not None:
74              bsize = inputs.shape[0]
75              dt = inputs.dtype
76          else:
77              bsize = batch_size
78              dt = dtype
79
80          cont_state = self.controller.get_initial_state(inputs, batch_size, dtype)
81          read_vector = tf.fill((bsize,self.memposlen),1e-6)
82          ws_cols = np.zeros(self.memsize)
83          ws_cols[0] = 1.
84          ws_cols = tf.constant(ws_cols, dtype=dt)
85          ws_rows = tf.ones(bsize, dtype=dt)
86          w_r = ws_rows[:,None] *ws_cols[None,:]
87          w_w = ws_rows[:,None] *ws_cols[None,:]
88          M = tf.fill((bsize,self.memsize,self.memposlen),1e-6)
89
90          return (cont_state, read_vector, w_r, w_w, M)
91
92      def _read(self, w, M):
93          # ecuacion 3.6
94          return tf.matmul(w[:,None,:], M)[:,0,:]
95
96      def _write(self, w, e, a, M):
97          # ecuacion 3.7
98          we = tf.matmul(w[:,:,None], e[:,None,:])
99          M_e = tf.multiply(M, (1 -we))
100
101         # ecuacion 3.8
102         wa = tf.matmul(w[:,:,None], a[:,None,:])
103         return M_e + wa
104
105     def _gen_weights(self, k, beta, M, g, wt_1, s, gamma):
106         # Acceso por contenido
107         # ecuacion 3.2
108         w = beta *cosine_similarity(k[:,None,:],M)
109         w = softmax(w)
110
111         # Acceso por posicion
112         # ecuacion 3.3
113         w = g *w + (1-g) *wt_1
114
115         # ecuacion 3.4
116         shift_matrix = s[:,:,None,None] *self.shift_ids[None,:,:,:]
117         shift_matrix = K.sum(shift_matrix, axis=1)
118
119         w = tf.matmul(w[:,None,:], shift_matrix)[:,0,:]
120
121         # ecuacion 3.5
122         w = _normalize(K.pow(w,gamma))
123
124         return w
```

```
126     def build(self, input_shape):
127         shape = list(input_shape)
128         shape[-1] += self.memposlen
129         shape = tf.TensorShape(shape)
130         return self.controller.build(shape)
131 
132     def call(self, inputs, states, training=None):
133         # separar estados
134         prev_cont_state, prev_read_vector, prev_w_r, prev_w_w, prev_M = states
135 
136         # controlador con inputs + lectura, estado anterior
137         cont_input = concatenate([inputs,prev_read_vector], axis=1)
138 
139         # el estado debe ser una lista.
140         prev_cont_state = [prev_cont_state]
141         cont_output, cont_state = self.controller.call(cont_input,prev_cont_state)
142 
143         # separar output del controlador en parametros
144         # activaciones de la implementacion de MarkPKCollier
145         e = self.e(cont_output)
146         a = self.a(cont_output)
147         k_r = self.k_r(cont_output)
148         k_w = self.k_w(cont_output)
149         beta_r = self.beta_r(cont_output)
150         beta_w = self.beta_w(cont_output)
151         g_r = self.g_r(cont_output)
152         g_w = self.g_w(cont_output)
153         s_r = self.s_r(cont_output)
154         s_w = self.s_w(cont_output)
155         gamma_r = self.gamma_r(cont_output)+1
156         gamma_w = self.gamma_w(cont_output)+1
157 
158         # generar nuevos pesos de escritura y lectura
159         w_r = self._gen_weights(k_r, beta_r, prev_M, g_r, prev_w_r, s_r, gamma_r)
160         w_w = self._gen_weights(k_w, beta_w, prev_M, g_w, prev_w_w, s_w, gamma_w)
161 
162         # lectura
163         read_vector = self._read(w_r, prev_M)
164 
165         # escritura
166         M = self._write(w_w, e, a, prev_M)
167 
168         # montar nuevo estado
169         new_state = (cont_state[0], read_vector, w_r, w_w, M)
170 
171         return cont_output, new_state
```


UAM
UNIVERSIDAD AUTONOMA
DE MADRID