UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Inferencia de gramáticas mediante redes neuronales recurrentes con regularización de estado

Autor: Luis Esteban Lepore Pérez de Obanos
Tutor: Luis Fernando Lago Fernández

junio 2023

Agradecimientos

Quiero expresar mi profundo agradecimiento a todas las personas que me han brindado su apoyo incondicional durante mi travesía hasta alcanzar este logro. En particular, quiero destacar el papel que mi mamá ha desempeñado al impulsarme a realizar el mejor trabajo posible y quiero agradecer a mi tutor por introducirme en el fascinante ámbito de estudio de este proyecto.

Resumen

En el presente trabajo de fin de grado se realiza un estudio sobre la inferencia de gramáticas regulares mediante redes neuronales recurrentes. El objetivo de este proyecto es añadir funcionalidad a la arquitectura de la red neuronal de Elman para implementar el mecanismo denominado “regularización de estados”. Este mecanismo restringe la cantidad de estados internos que la red puede adquirir, asegurando que durante su entrenamiento únicamente aprenda un conjunto finito de estos.

La regularización de estados de este proyecto fue puesta en práctica mediante un proceso de muestreo. Este proceso aplica una función que no es diferenciable, lo cual fuerza la introducción de técnicas de aprendizaje por refuerzo durante el entrenamiento del modelo. Una vez que el modelo fue implementado se puso a prueba mediante las gramáticas de Tomita, las cuales son un conjunto de siete gramáticas de complejidad creciente ampliamente utilizado en problemas de inferencia gramatical.

El procedimiento llevado a cabo para este proyecto se puede resumir en cuatro partes: diseñar e implementar una red neuronal recurrente con regularización de estados, entrenar dicha red de forma individual para cada una de las gramáticas de Tomita, extraer los autómatas finitos deterministas correspondientes a estas gramáticas a partir de las redes previamente entrenadas y comparar los autómatas extraídos con los definidos por las gramáticas de Tomita.

Palabras clave

Inferencia de gramáticas, red neuronal recurrente, red neuronal de Elman, regularización de estados, aprendizaje por refuerzo, gramáticas de Tomita, autómata finito determinista

Abstract

This bachelor thesis serves as a study on the inference of regular grammars through the use of recurrent neural networks. The objective of this project is to add functionality to the Elman neural network architecture in order to implement the "state-regularization" mechanism. This mechanism restricts the number of internal states that the network can acquire, ensuring that during its training it only learns a finite set of them.

The state-regularization used in this project was set in motion through a sampling process. Said process applies a non-differentiable function, which forces reinforcement learning techniques to be added during the model's training. Once implemented, the model was tested using Tomita's grammars, a group of seven grammars that grow in complexity, which are widely used in grammar inference problems.

The process completed in this project can be summarized in four parts: design and implement a state-regularized recurrent neural network, train said network individually for each one of Tomita grammars, extract the deterministic finite automaton corresponding to these grammars from the previously trained networks and compare the extracted automaton with those defined by Tomita's grammars.

Keywords

Grammar inference, recurrent neural network, Elman neural network, state-regularization, reinforcement learning, Tomita grammars, deterministic finite automaton

Índice

Listas

Lista de códigos

Lista de ecuaciones

Lista de figuras

Lista de tablas

1 Introducción

La inferencia gramatical es el proceso que consiste en aprender un lenguaje formal, normalmente infinito, a partir de un conjunto finito de datos. El objetivo es construir un modelo, generalmente una colección de reglas de derivación o un autómata finito, que reconozca los patrones comunes existentes en el conjunto de datos y que sea capaz de generalizar frente a datos nunca antes vistos.

Los primeros estudios sobre la inferencia de gramáticas datan de la década de 1960, donde se destaca la formalización conocida como “identificación de un lenguaje en el límite” [1]. Sin embargo, el avance del aprendizaje automático ha generado numerosas oportunidades en este ámbito de investigación. Una de estas posibilidades consiste en extraer las reglas de derivación de una gramática a partir de una red neuronal recurrente entrenada con ejemplos, tanto positivos como negativos, del lenguaje asociado a dicha gramática [2].

Las redes neuronales recurrentes son un tipo de redes neuronales artificiales especializadas en procesar datos con estructura temporal. Esto es debido a que su arquitectura introduce una conexión en la que el estado interno de la red depende tanto de la entrada actual como del estado interno previo. Este diseño específico las convierte en una herramienta muy útil para una gran variedad de tareas, como el reconocimiento de voz [3], el procesamiento del lenguaje natural [4] y la generación de texto [5]. No obstante, a pesar de sus beneficios, las redes neuronales recurrentes también presentan un desafío fundamental: actúan como cajas negras.

Este concepto de "caja negra" se refiere a la falta de interpretabilidad de las decisiones tomadas. A diferencia de otros modelos igual de potentes como las máquinas de Turing, donde se pueden seguir claramente las reglas y los pasos de inferencia, los estados internos de las redes neuronales recurrentes son difíciles de desglosar en componentes lógicos comprensibles para los humanos.

El objetivo de este trabajo de fin de grado es aportar una solución a esta falta de interpretabilidad mediante la implementación del mecanismo denominado “regularización de estados” [6]. Este mecanismo restringe la cantidad de estados internos que la red recurrente

puede adquirir, asegurando que durante su entrenamiento únicamente aprenda un conjunto finito de estos. Lo anterior produce que la red se comporte internamente como un autómata finito determinista, modelo cuyo funcionamiento es fácilmente interpretable.

Una de las formas en las que la regularización de estados se puede poner en práctica es mediante un proceso de muestreo. Dicho proceso fue el implementado en este proyecto y consiste en establecer el estado interno de la red recurrente como la combinación de las neuronas con mayor probabilidad de activación. Además, este proceso tiene la particularidad de que aplica una función que no es diferenciable, lo que impide que el gradiente pueda ser retropropagado de un extremo a otro de la red y, como consecuencia, fuerza la introducción de técnicas de aprendizaje por refuerzo durante el entrenamiento del modelo. Para comprobar que este conjunto de técnicas funcionan correctamente, el modelo desarrollado será puesto a prueba mediante las gramáticas de Tomita [7], las cuales son un conjunto de siete gramáticas de complejidad creciente ampliamente utilizado en problemas de inferencia gramatical.

Para terminar, este documento está dividido en seis capítulos, siendo el primero de ellos esta introducción. En el siguiente capítulo se introducen los conceptos básicos asociados a este proyecto y se presenta la situación actual sobre el campo de investigación de la inferencia de gramáticas regulares. En el tercer capítulo se detallan los conjuntos de datos utilizados y se describe el diseño del modelo de aprendizaje automático desarrollado y de los experimentos realizados. En el cuarto capítulo se detalla el código y las herramientas empleadas para poner en práctica el diseño del modelo. En el penúltimo capítulo se recopilan los resultados obtenidos y se realiza un análisis sobre los mismos. Finalmente, en el sexto y último capítulo se presentan las conclusiones.

2

Estado del arte

En este capítulo se explican los conceptos básicos a los que se hará referencia a lo largo del presente trabajo de fin de grado y se proporciona un estudio del estado del arte sobre el campo de investigación en el que se enmarca este proyecto. Siguiendo este esquema, las primeras secciones introducen progresivamente los conceptos relativos a: lenguajes regulares, aprendizaje automático y redes neuronales recurrentes. Por último, este capítulo concluye con un análisis de la situación actual del conocimiento sobre la inferencia de gramáticas regulares dentro del contexto de las redes neuronales recurrentes.

2.1. Lenguajes regulares y autómatas finitos deterministas

Para entender correctamente todo el contenido de este capítulo y el de los posteriores, es necesario primero conocer el significado de los términos: lenguaje, gramática y autómata. Al igual que establecer las relaciones existentes entre cada uno de estos conceptos.

Es común estar familiarizado con la noción de lenguajes naturales, tales como el español o el inglés. Sin embargo, para la mayoría de las personas resultaría difícil definir con precisión lo que significa la palabra "lenguaje". Para abordar el estudio de los lenguajes formales es necesario definir lo que es un alfabeto. Un alfabeto Σ es un conjunto finito y no vacío de símbolos. Al concatenar estos símbolos individuales entre sí se construye lo que se conoce como “cadena”. De forma que si Σ es un alfabeto, se denomina Σ^* al conjunto de cadenas obtenidas al concatenar cero o más símbolos de Σ. Finalmente, un lenguaje L se puede definir como un subconjunto de Σ^* [8].

Una de las formas en las que puede describirse un lenguaje es mediante una gramática. Una gramática [8] se define como una cuádrupla $G = (N, \Sigma, S, P)$, donde N es un conjunto finito de símbolos no terminales y al alfabeto Σ se le añade la restricción de que debe ser disjunto de N. S es un símbolo inicial perteneciente al conjunto N y P es un conjunto finito de reglas de derivación que se aplican desde el símbolo inicial S para obtener las cadenas del lenguaje (ver primera columna de la tabla 2.2).

De acuerdo con la jerarquía de Chomsky [9], existen cuatro tipos de gramáticas: tipo 0 o sin restricciones, tipo 1 o dependiente del contexto, tipo 2 o independiente del contexto y tipo 3 o regular (ver tabla 2.1). Esta clasificación se basa en las restricciones que tienen las reglas de derivación y al ser jerárquica cada nivel engloba a los siguientes.

Tipo	Lenguaje	Autómata	Ejemplo
0	Recursivamente enumerable.	Máquina de Turing.	$\mathrm{L} = \{\mathrm{w} \mid \mathrm{w}\ \mathrm{describe\ una\ máquina\ de\ Turing}\}$
1	Dependiente del contexto.	Autómata linealmente acotado.	$L = \{a^n b^n c^n \mid n > 0\}$
2	Independiente del contexto.	Autómata a pila.	$L = \{a^n b^n \mid n > 0\}$
3	Regular.	Autómata finito determinista.	$L = \{a^n \mid n > 0\}$

Tabla 2.1: Tipos de gramáticas. Asociación de cada tipo de gramática con su respectivo lenguaje, autómata y ejemplo.

Se profundiza a continuación en el último tipo mencionado en la tabla 2.1, el 3, por ser el tipo de gramática utilizado a lo largo de este trabajo de fin de grado. Para que una gramática sea regular debe cumplir que sus reglas de derivación P adopten una de las siguientes dos estructuras:

$$A \rightarrow x \mid Bx \mid \lambda \tag{2.1}$$

$$A \rightarrow x \mid xB \mid \lambda \tag{2.2}$$

donde $A, B \in N$, $x \in T$, λ corresponde a la cadena vacía y | se utiliza como el operador booleano OR con el fin de agrupar aquellas reglas que tengan el mismo lado izquierdo. En este caso se debe cumplir que el lado izquierdo de la regla debe contener un único símbolo no terminal (A), mientras que el lado derecho consta de un único símbolo terminal (x) o de un símbolo terminal y otro no terminal (xB, Bx) o de la cadena vacía (λ). Además, si el lado derecho presenta símbolos no terminales debe hacerlo siguiendo siempre la misma estructura de linealidad, es decir, si los símbolos no terminales aparecen siempre al inicio se trata de una gramática regular izquierda (ver ecuación 2.1), mientras que si aparecen siempre al final se trata de una gramática regular derecha (ver ecuación 2.2).

Un lenguaje regular también se puede describir mediante las expresiones regulares (ER) definidas por Kleene [10]. Una expresión regular es una secuencia de caracteres que se utiliza para definir el patrón o el conjunto de patrones de las cadenas pertenecientes a un lenguaje regular. Una expresión regular está formada por los símbolos de un alfabeto Σ, los paréntesis y los operadores de unión "+", de cierre estrella "$*$" y de concatenación "." (ver segunda columna de la tabla 2.2). El operador de concatenación suele ignorarse en la práctica para mejorar la legibilidad de la expresión, de forma que $a.b$ es equivalente a ab.

El último concepto que falta por introducir en esta primera subsección es el de autómata finito determinista (AFD). Un autómata finito determinista [8] o una máquina de estados determinista, es un modelo matemático que consiste en un conjunto finito de estados y una función de transición que, dado un estado origen y un símbolo de entrada, produce un estado destino (ver tercera columna de la tabla 2.2). La definición formal de un autómata finito consiste en una quíntupla $A = (Q, \Sigma, \delta, q_0, F)$, donde Q es el conjunto de los estados del autómata, Σ es el alfabeto, $\delta: Q \times \Sigma \rightarrow Q$ es la función de transición, $q_0 \in Q$ es el estado inicial y $F \subseteq Q$ es el conjunto de estados de aceptación. El funcionamiento de un autómata queda definido por su función de transición δ, esta establece que a partir de un estado $q \in Q$ y un símbolo del alfabeto $s \in \Sigma$, se obtiene otro estado q'. Además, si al terminar de procesar una cadena $w \in \Sigma^*$ el estado del autómata es un estado final $q_f \in Q$, entonces la cadena w pertenece al lenguaje asociado a dicho autómata. Se añade el concepto de determinismo cuando existe una única transición $\delta_{q,s} \in \delta$ para cada estado $q \in Q$ y símbolo $s \in \Sigma$.

Es importante mencionar que existe una estrecha relación entre gramáticas regulares, expresiones regulares y autómatas finitos deterministas. Todas estas herramientas son utilizadas para describir lenguajes regulares y existe tanto evidencia teórica (ver figura 2.1) como práctica (ver tabla 2.2) de que son equivalentes entre sí.

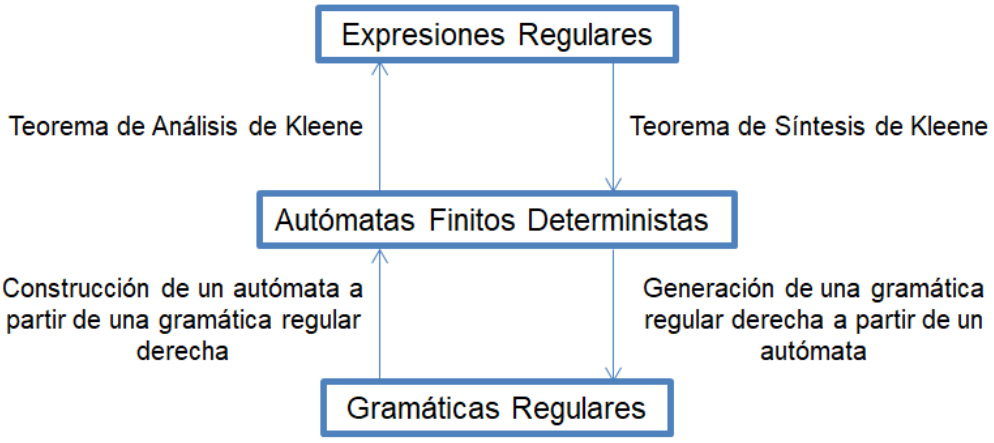


Figura 2.1: Equivalencia entre gramáticas regulares, ER y AFD.

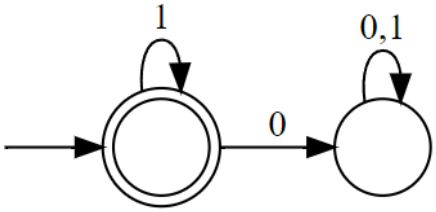
Gramática Regular	Expresión Regular	Autómata Finito Determinista
$A \rightarrow 1A \mid \lambda$	1^*	

Tabla 2.2: Ejemplo de gramáticas regulares, ER y AFD. Equivalencia entre una gramática regular con su expresión regular y autómata finito determinista correspondientes. Todos definen el lenguaje 1* (cero o más repeticiones del número 1) con Σ = {0, 1}.

2.2. Aprendizaje automático y aprendizaje por refuerzo

Antes de hablar sobre aprendizaje automático (AA) y aprendizaje por refuerzo (AR) es importante dar, al menos, una definición de lo que es la inteligencia artificial (IA). La IA [11] en el contexto de la informática se define como el estudio de “agentes inteligentes”, los cuales son dispositivos que “perciben su entorno y toman decisiones para maximizar las probabilidades de éxito de un objetivo dado”.

El aprendizaje automático [12] es considerado una rama de la inteligencia artificial, este permite crear modelos que aprenden de la experiencia y mejoran su rendimiento a través de algoritmos computacionales. Estos algoritmos utilizan grandes conjuntos de datos, llamados conjuntos de entrenamiento, para reconocer patrones y “aprender” a tomar decisiones de forma autónoma. Después de suficientes repeticiones y ajustes del algoritmo, el modelo se vuelve capaz de recibir una entrada y predecir su salida. Estos resultados se comparan con un conjunto de validación (distinto al de entrenamiento) para calcular la precisión del algoritmo, el cual es ajustado iterativamente hasta maximizar la habilidad de predecir las salidas de datos de entrada nunca antes vistos.

Dino y Francesco Esposito [13] resumen los pasos generales a la hora de resolver un problema mediante aprendizaje automático de la siguiente manera:

- Recolección de datos. La cantidad, calidad y correlación de los datos son cruciales para un proyecto exitoso.
- Preparación de datos. Es necesario preprocesar y limpiar los datos pues estos pueden ser incompletos, ruidosos o incoherentes.
- Selección del modelo y entrenamiento. Se debe categorizar el problema en cuestión y seleccionar un algoritmo o una cadena de algoritmos para el entrenamiento.
- Evaluación. Obtención e interpretación de los resultados, en caso de que no se obtengan los esperados debe modificarse alguno de los puntos anteriores y repetir el proceso.

Uno de los paradigmas más comunes dentro del aprendizaje automático es el aprendizaje supervisado (AS). El aprendizaje supervisado [14] se aplica a los problemas donde la información disponible consiste en datos etiquetados. El término “dato etiquetado” se utiliza cuando los datos de entrada van acompañados de la salida correcta mediante una etiqueta. El objetivo de un algoritmo supervisado es aprender mediante los ejemplos del conjunto de entrenamiento una función que asocie los datos de entrada con la etiqueta correspondiente.

Por otro lado, el aprendizaje por refuerzo [15] es el problema al que se enfrenta un agente situado en un entorno dinámico que debe aprender un comportamiento a través de la técnica de ensayo y error. El aprendizaje por refuerzo difiere del aprendizaje supervisado en dos grandes

aspectos. El primero de ellos es que al agente no se le notifica cuál hubiera sido la decisión óptima tras la elección de una acción. La segunda diferencia es que las fases de entrenamiento y evaluación a menudo son concurrentes.

Formalmente, este tipo de modelo consiste en un conjunto discreto de estados del entorno S; un conjunto discreto de acciones del agente A; y un conjunto escalar R de señales de refuerzo, por ejemplo $\{-1, 1\}$ o los números reales $\mathbb{R}$. También se debe añadir una función de entrada I, la cual determina la forma en la que el agente visualiza su entorno, y dependiendo de cómo se defina puede ser fidedigna a este o una simplificación del mismo. En la figura 2.2 se resume la secuencia de pasos en el tiempo de un modelo entrenado mediante aprendizaje por refuerzo, donde el agente al encontrarse en el estado s_t y tomar la acción a_t hace que el entorno cambie al estado s_{t+1} y le devuelva la recompensa r_{t+1}.

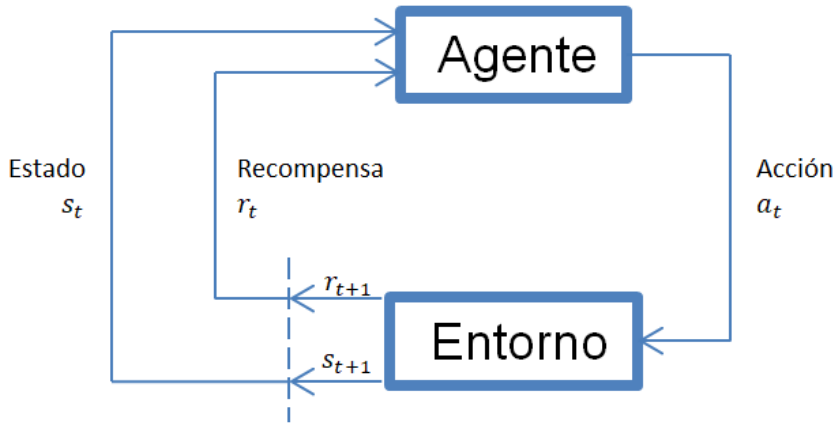


Figura 2.2: Interacción entre un agente y su entorno. Se ejemplifica la secuencia de pasos en el tiempo de un agente entrenado mediante aprendizaje por refuerzo.

El último concepto que falta por introducir con respecto al aprendizaje por refuerzo es la política: π. Esta es la relación entre los estados percibidos y la acción realizada por el agente en dichos estados. El objetivo del agente es encontrar aquella política que maximice la recompensa a largo plazo. Además, se espera que el entorno sea no determinista, de forma que tomar la misma acción en un mismo estado en ocasiones diferentes pueda resultar en un estado destino y/o señal de refuerzo distintos.

2.3. Redes neuronales artificiales

Las redes neuronales artificiales (RNA) son un modelo de aprendizaje automático inspirado en las neuronas existentes en el sistema nervioso animal. La equivalencia que se establece entre las neuronas biológicas y las artificiales [16] es la siguiente: dendritas como entradas, sinapsis como las conexiones ponderadas, cuerpo celular como una regla de propagación seguida de una función de activación y axón como salida (ver figura 2.3).

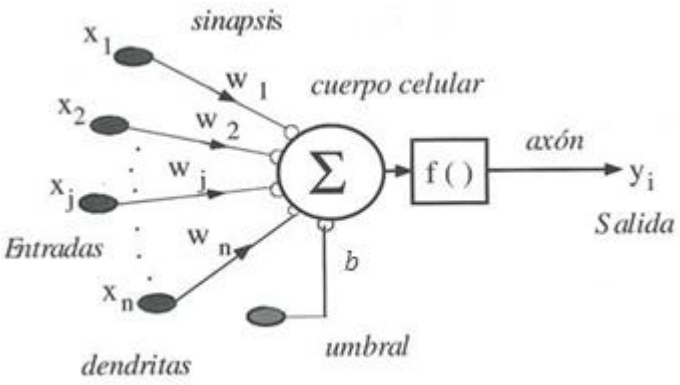


Figura 2.3: Modelo de neurona artificial estándar. Se ilustran los elementos básicos de una neurona artificial y se establecen las equivalencias con una neurona biológica. Extraído de http://grupo.us.es/gtocoma/pid/pid10/RedesNeuronales.htm#modeloneurona

La ecuación 2.3 determina la activación de una neurona y con n conexiones ponderadas:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.3}$$

donde x_i representa cada uno de los valores de entrada, w_i es el peso asociado a dicha entrada, b es el umbral, $\sum_{i=1}^{n} w_i x_i$ es la regla de propagación y f es la función de activación. Es importante recalcar que el umbral no es obligatorio, que la regla de propagación no tiene por qué ser una suma ponderada, aunque en la práctica casi siempre lo será, y que la función de activación es una función no lineal que debe ser diferenciable. Tradicionalmente las funciones de activación más utilizadas eran la sigmoidea y la tangente hiperbólica, sin embargo, hoy en día se prefiere optar por funciones ReLU [17] o similares.

Según Haykin [18], una red neuronal artificial es una agrupación masivamente paralela de una unidad simple de procesamiento (neurona) que puede adquirir conocimiento a través de un proceso de aprendizaje. Este conocimiento se ve reflejado en una actualización de los pesos de las conexiones mencionadas anteriormente.

La topología en la que diferentes neuronas se asocian para conformar una red se conoce con el nombre de arquitectura de la red neuronal. En general, las neuronas suelen estar agrupadas en unidades estructurales denominadas capas y dentro de una misma capa las neuronas suelen tener la misma función de activación. Se pueden distinguir tres tipos de capas (ver figura 2.4):

- Capa de entrada: compuesta por las neuronas que reciben datos del entorno.
- Capas ocultas: aquellas que no tienen conexiones directas con el entorno.
- Capa de salida: compuesta por las neuronas que proporcionan la respuesta final.

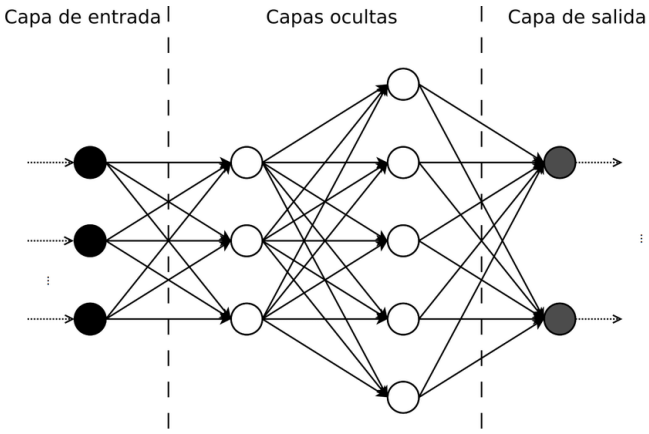


Figura 2.4: Red neuronal multicapa. Red neuronal artificial de cuatro capas con propagación hacia adelante. Extraído de https://www.researchgate.net/figure/Red-neuronal-artificial-de-cuatro-capas_fig1_323985249

Al ser un modelo de aprendizaje automático, las redes neuronales artificiales necesitan de una fase de entrenamiento. Esta consiste en establecer valores adecuados a los pesos de la red con el fin de minimizar el error obtenido al evaluar los datos de entrenamiento. Hay que tomar las precauciones necesarias para evitar casos de "sobreajuste" y "subajuste". El primero es propio de situaciones en las que se intenta disminuir el error a cuotas ínfimas, de forma que el modelo se especializa en los datos de entrenamiento y se vuelve incapaz de reconocer correctamente nuevos datos de entrada. El segundo caso sucede cuando el modelo no se entrena lo suficiente, de manera que el rendimiento es malo tanto para los datos de entrenamiento como para los de validación. Normalmente se busca el punto óptimo entre ambas situaciones con la finalidad de que la red sea capaz de generalizar.

Pese a que existen diferentes arquitecturas, la más utilizada en problemas de aprendizaje supervisado es la distribución por capas con propagación hacia adelante, donde cada capa está completamente conectada con la siguiente (ver figura 2.4). Este modelo define la activación de las neuronas de la capa i mediante la ecuación 2.4:

$$\boldsymbol{z_i = f(W_i z_{i-1} + b_i)} \tag{2.4}$$

donde z_i corresponde a la salida de la capa i, W_i es la matriz de pesos que conecta la capa anterior z_{i-1} con la actual, b_i es el vector de umbral correspondiente a dicha capa y f es la función de activación.

Al definir lo que es una red neuronal artificial se mencionó que el conocimiento adquirido se ve reflejado en una actualización de los pesos del modelo. Por lo tanto, es necesario realizar correcciones en dichos pesos y en los umbrales con el fin de minimizar una función de coste que mide la discrepancia entre la salida de la red y el resultado esperado.

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 [19], que consiste en calcular la derivada de la función de coste respecto de cada peso de la red. Analizando la gráfica de la función 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 se encontrará un mínimo local que puede ser o no un coste aceptable para la red.

El método más extendido para el cálculo del gradiente en redes neuronales es la retropropagación [20]. 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 continúa 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.

La magnitud de las correcciones que se realizan en dirección contraria al gradiente calculado se conoce como tasa de aprendizaje. Con una tasa de aprendizaje elevada se avanza rápido en la dirección deseada pero es posible que, estando cerca de la solución, se reaccione exageradamente y se aleje del mínimo de la función en lugar de acercarse al mismo. Por el contrario, una tasa de aprendizaje pequeña realiza pasos más consistentes pero es posible que se estanque en un mínimo local. Con estas ideas se puede concluir que lo óptimo es conseguir un balance entre ambas situaciones.

2.4. Redes neuronales recurrentes

Una red neuronal recurrente (RNR) es un tipo de red neuronal especializada en procesar datos secuenciales mediante una relación de recurrencia en sus conexiones internas. A continuación se describe la arquitectura de la red neuronal de Elman [21], cuyas ecuaciones en el instante de tiempo t son las siguientes:

$$\boldsymbol{h_t} = \sigma(W_{xh}\boldsymbol{x_t} + W_{hh}\boldsymbol{h_{t-1}} + \boldsymbol{b_h}) \tag{2.5}$$

$$\boldsymbol{y_t} = \sigma(W_{hy}\boldsymbol{h_t} + \boldsymbol{b_y}) \tag{2.6}$$

donde h_t es el estado actual de la red, h_{t-1} es el estado en el instante inmediatamente anterior, x_t es el vector de entrada e y_t es el vector de activación de salida. En este caso se consideran tres matrices de pesos, W_{xh} para la capa de entrada, W_{hh} para la conexión recurrente y W_{hy} para la capa de salida. También existen dos umbrales distintos, b_h para el cálculo del estado y b_y para el de la salida. Por último, la función sigmoidea realiza la operación de la ecuación 2.7 y se utiliza el símbolo σ para representarla.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

En la ecuación 2.5 se puede observar que la recurrencia va desde la capa oculta hacia sí misma, ya que el estado h_t depende del estado anterior h_{t-1}. Es importante remarcar este detalle ya que existen otros tipos de redes recurrentes, como por ejemplo la de Jordan [22], donde la recurrencia va desde la capa de salida hacia la oculta (h_t depende de y_{t-1}). Además, existen otros tipos de redes recurrentes más complejas, como las LSTM [23] y las GRU [24], sin embargo, no tendrán mayor relevancia en este trabajo pues se optó por la arquitectura clásica de Elman.

Para entender con mayor claridad el funcionamiento interno de una red recurrente se utiliza la figura 2.5. Se puede observar en la parte izquierda que la red tiene una única capa oculta h retroalimentada por sí misma. Además, en la parte derecha se representa el concepto conocido como “desenrollado”, el cual permite cambiar el punto de vista y analizar el modelo como una red neuronal multicapa donde todas las capas de entrada, recurrente y de salida comparten los pesos W_{xh}, W_{hh} y W_{hy}, respectivamente. Adicionalmente, al desenrollar la red recurrente se pone en evidencia como cada capa recurrente se ve influenciada por la anterior.

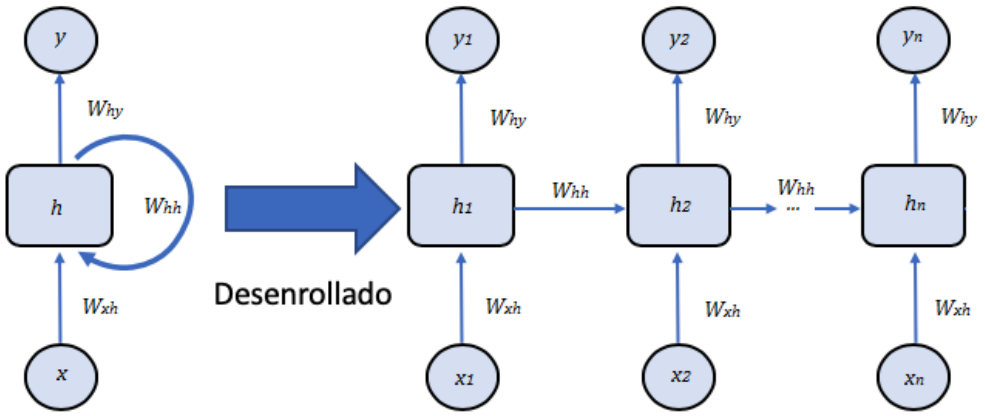


Figura 2.5: Red neuronal recurrente. Ilustración de los conceptos de recurrencia (izquierda) y desenrollado (derecha) de una red neuronal recurrente de Elman. Extraído de http://personal.cimat.mx:8181/~mrivera/cursos/aprendizaje_profundo/RNN_LTSM/rnn1.png

Una vez descrito el concepto de desenrollado es fácil observar que el estado h_t depende del estado anterior, h_{t-1}, y este a su vez depende de h_{t-2}. Esto produce que a la hora de calcular los gradientes de la función de coste respecto de los pesos y umbrales de la capa recurrente, sea necesario propagar el error por un número arbitrariamente grande de instantes de tiempo. Para evitar este inconveniente, en la práctica se utiliza la retropropagación truncada a través del tiempo, la cual permite establecer un límite para el número de estados anteriores a los que propagar el error del estado actual.

2.5. Inferencia de gramáticas regulares mediante RNR

El objetivo de esta sección es situar al lector dentro del contexto de la inferencia de gramáticas mediante redes neuronales recurrentes. Antes de entrar en materia es conveniente recalcar que este tipo de redes posee dos deficiencias: es difícil entender qué es exactamente lo que aprenden y el bajo desempeño que presentan cuando necesitan memoria a largo plazo, a pesar de que en teoría disponen de esta característica. A lo largo de esta sección se describirán las soluciones planteadas por distintos autores con el objetivo de subsanar el primero de estos inconvenientes, terminando con la propuesta de [6], la cual fue utilizada como inspiración para el presente proyecto.

Es importante mencionar que la idea de aprender lenguajes formales mediante redes recurrentes es un paradigma moderno, ya que existen otros algoritmos más tradicionales para la extracción de autómatas tales como RPNI [25] y LSTAR [26], donde la elección de uno u otro depende de la presentación de los datos disponible, tal y como se explica en [27]. No obstante, la inferencia de gramáticas clásica no es el objeto de estudio del presente trabajo, por lo que a continuación se volverá a poner el foco sobre el ámbito de las redes recurrentes.

En la revisión de Jacobsson realizada en 2005 [28] se puede apreciar la abrumadora cantidad de artículos en los que se busca una equivalencia entre RNR y AFD. En [29] se demuestra teóricamente que una red neuronal recurrente con una cantidad suficientemente grande de neuronas y una función de activación no lineal es un modelo de computación universal, lo que significa que es capaz de realizar cualquier cálculo que pueda ser realizado por una máquina de Turing [30], incluyendo el cómputo de lenguajes regulares y la extracción de sus correspondientes AFD. Esta extracción se remonta a la década de los noventa, en la que investigaciones como [31] y [32] utilizaron métodos de clusterización para proyectar el espacio de estados interno de redes recurrentes previamente entrenadas, con el objetivo de aplicarles a posteriori algoritmos de reducción de dimensionalidad.

En [33] se cambia el enfoque y se busca que las RNR sean más interpretables por los humanos. Este artículo demuestra que se pueden identificar patrones y relaciones entre las variables de entrada y salida, lo que facilita el entendimiento del funcionamiento interno y mejora la capacidad de depuración de este tipo de redes. Otra opción es la planteada en [2], donde se busca extraer reglas a partir de RNR entrenadas para la clasificación de texto. En particular, construye un clasificador simple basado en reglas con una precisión ligeramente peor a la de la red original, pero mucho más fácil de interpretar y explicar.

Investigaciones recientes introducen enfoques más sofisticados para la extracción de autómatas a partir de arquitecturas recurrentes complejas como LSTM o GRU. El artículo [34] propone un enfoque que utiliza consultas y contraejemplos para extraer autómatas a partir de redes neuronales recurrentes. Las consultas se utilizan para preguntar a la red neuronal sobre su comportamiento en ciertos estados y observar las respuestas, mientras que los contraejemplos se emplean para refinar el autómata.

Existen varias técnicas en cuanto a la regularización de RNR. La mayoría de ellas adaptan los enfoques de regularización para redes de propagación hacia adelante (sin bucles) al entorno recurrente. Entre las técnicas más representativas que se abordan están: regularización mediante eliminación aleatoria de neuronas [35], una variación del anterior que en lugar de aplicarse a las neuronas se aplica a los pesos de las conexiones de la capa oculta de una LSTM [36] y la inyección de ruido para mejorar la generalización de la red [37].

Sin embargo, es importante no confundir estos métodos clásicos de regularización con la regularización de estados implementada en este proyecto. La regularización es una técnica que mejora la generalización de un modelo mediante la simplificación del mismo. Por otro lado, la regularización de estados implica la aplicación de técnicas que controlan y limitan el crecimiento y la propagación de los estados internos de una RNR durante el entrenamiento.

Esta regularización de estados, fundamentada en [6], simplifica la extracción de autómatas mediante el aprendizaje de un conjunto finito de estados y una función de transición de estados interpretable, todo ello sin sacrificar precisión. Además, una única capa de una red neuronal recurrente con regularización de estados ofrece resultados comparables a los obtenidos con otros métodos de última generación. No obstante, este modelo no resuelve todos los problemas asociados a las RNR, ya que no garantiza una mejora en la convergencia y puede ocasionar mayores tiempos de entrenamiento.

3 Diseño

Tras haber desarrollado el estado del arte, corresponde a este tercer capítulo explicar el diseño de los experimentos y las ampliaciones realizadas a la arquitectura de la red de Elman descrita en la sección 2.4. Entrando más en detalle, en la primera sección se describen tanto los conjuntos de datos como las gramáticas regulares que se usan en los experimentos de este proyecto. En la siguiente sección se define el modelo de aprendizaje automático utilizado para desarrollar su respectiva implementación en el siguiente capítulo y por último se describe el diseño de los experimentos ejecutados para más adelante realizar una comparativa de resultados.

3.1. Descripción de los datos y gramáticas

Para realizar los experimentos de este trabajo se consideraron las gramáticas regulares sobre el alfabeto $\Sigma = \{a, b\}$ definidas en la tabla 3.1 y representadas en la figura 3.1. Esta tabla está formada por las gramáticas de Tomita [7], las cuales son un conjunto de referencia ampliamente utilizado en problemas de inferencia gramatical.

Debido a su diseño especializado en procesar datos de forma secuencial, las redes recurrentes tratan las cadenas del lenguaje como una serie de datos con estructura temporal. En el ámbito de la inferencia de gramáticas, esto se traduce en el procesamiento de tantos conjuntos de símbolos de la cadena de entrada como indique el tamaño del lote por cada instante de tiempo. Ahora bien, para que el modelo fuera capaz de procesar un conjunto de cadenas de longitud variable sin interrupciones se decidió, al igual que en [38], añadir un nuevo símbolo al alfabeto: el $. Este símbolo se usa como separador, es decir, delimita el fin de la cadena que se venía procesando y marca el inicio de la cadena que se procesará a continuación.

En cualquier problema de aprendizaje supervisado se requieren conjuntos de entrenamiento y validación. En este caso, dichos conjuntos están representados por ficheros de texto plano. Cada uno de estos conjuntos es específico para cada una de las gramáticas y está conformado por una dupla de ficheros, uno de entrada y otro de salida. Los datos de entrada se construyen fácilmente mediante un generador de cadenas aleatorias en el que se asigna a cada

uno de los símbolos de entrada $\{\$, a, b\}$ una probabilidad específica. Por otro lado, los datos de salida se pueden generar mediante el AFD correspondiente a la gramática que se esté analizando, de forma que la presencia de cada 1 o 0 en la salida indica si el autómata aceptó o no la cadena de entrada procesada desde el último $ hasta ese momento (ver figura 3.2).

Nombre	Lenguaje que define	Expresión regular
Tomita1	Cadenas con cero o más aes.	a^*
Tomita2	Cadenas con cero o más repeticiones de una a seguida de una b.	$(ab)^*$
Tomita3	Cadenas que no tengan un número impar de aes seguidas de un número impar de bes.	$b^*[aa(aa)^*b^* + a(aa)^*bb(bb)^*]^*(a+\lambda)$
Tomita4	Cadenas que no tengan tres bes consecutivas.	$(a+ba+bba)^*(bb+b+\lambda)$
Tomita5	Cadenas que tengan un número par de aes y de bes.	$[aa+bb+(ab+ba)(aa+bb)^*(ab+ba)]^*$
Tomita6	Cadenas donde la diferencia entre el número de aes y bes sea múltiplo de tres.	$[ba+(a+bb)(ab)^*(b+aa)]^*$
Tomita7	Cadenas con cero o más bes, seguidas de cero o más aes, seguidas de cero o más bes, seguidas de cero o más aes.	$b^*a^*b^*a^*$

Tabla 3.1: Descripción de las gramáticas de Tomita.

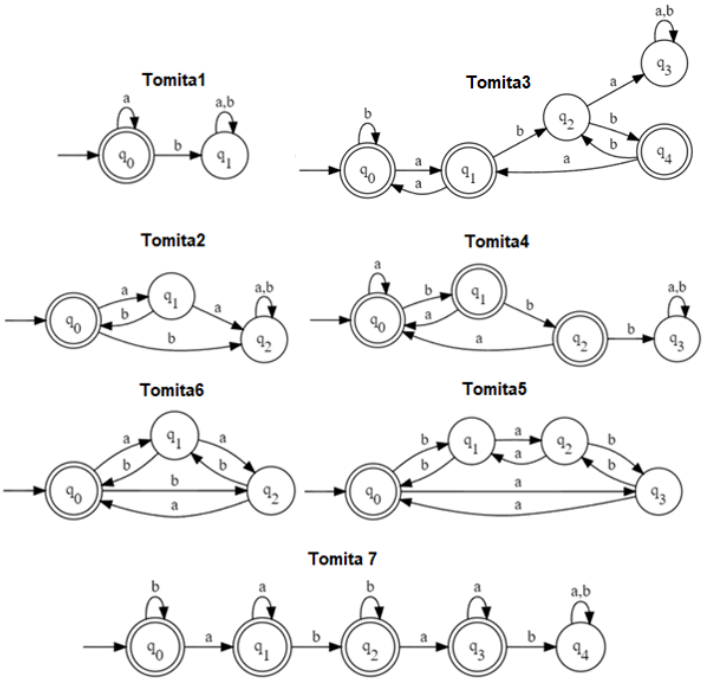


Figura 3.1: AFD mínimos correspondientes a las gramáticas de Tomita. Estos autómatas son equivalentes a las expresiones regulares correspondientes definidas en la última columna de la tabla 3.1.

```
$abab$aab$aaaaabbbaabaab$a$a$a$aaaabbabbababaaabaaa

1100011101111110000000001111111111100000000000000000
```

Figura 3.2: Extracto de los datos de entrada y salida para la gramática Tomita1. Aquí se puede apreciar un extracto de un conjunto de datos conformado por sus ficheros de entrada (arriba) y de salida (abajo) para la gramática Tomita1.

El último detalle que queda por explicar en cuanto a los conjuntos de datos utilizados en este proyecto es que existen cinco de ellos para cada una de las gramáticas (ver tabla 3.2) y que estos fueron obtenidos de [38]. El primero de estos conjuntos es *entrenamiento* y está formado por cincuenta mil símbolos con la misma probabilidad de aparición de aes que de bes. Los otros cuatro conjuntos son de validación, estos poseen variaciones con respecto al conjunto de entrenamiento en la cantidad de caracteres y/o en la probabilidad de aparición de los mismos y son utilizados para demostrar que el modelo es capaz de generalizar correctamente frente a distintos tipos de cadenas. El conjunto *grande* posee cien mil símbolos con la misma probabilidad de aparición de aes que de bes, el conjunto *largo* tiene veinte mil símbolos y una probabilidad de aparición del símbolo dólar mucho menor, por último, los conjuntos *muchas_a* y *muchas_b* tienen quince mil símbolos y una probabilidad de aparición muy elevada de aes o de bes, respectivamente.

Nombre	**N.º de caracteres**	**Prob. a**	**Prob. b**	**Prob.** $
Entrenamiento	50000	0.45	0.45	0.1
Grande	100000	0.45	0.45	0.1
Largo	20000	0.495	0.495	0.01
Muchas_a	15000	0.98	0.01	0.01
Muchas_b	15000	0.01	0.98	0.01

Tabla 3.2: Descripción de los conjuntos de datos.

3.2. Descripción del modelo

Como se introdujo al inicio de este capítulo, el modelo utilizado en este trabajo de fin de grado se basa en la arquitectura de la red de Elman. Partiendo de esta arquitectura como base, se propuso añadir funcionalidad hasta diseñar una red neuronal recurrente con regularización de estados, basándose en [6]. Este tipo específico de red recurrente limita la cantidad de estados posibles que el modelo puede aprender, lo que permite extraer fácilmente el AFD correspondiente a la gramática inferida.

En la figura 3.3 se muestra una representación gráfica del funcionamiento del modelo. En este diagrama se puede observar cómo los datos se combinan y transforman al realizar todos los cálculos necesarios con el objetivo de aceptar o rechazar la cadena de entrada leída hasta el instante de tiempo t.

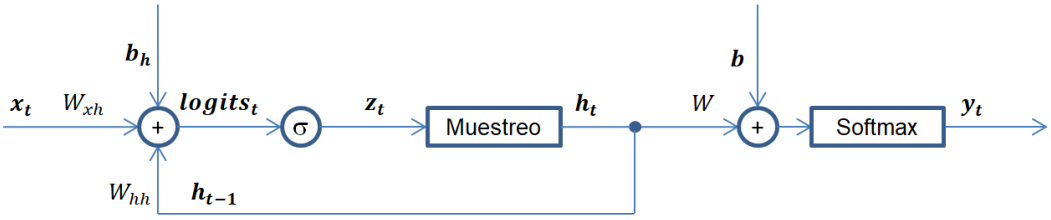


Figura 3.3: Diseño del modelo de aprendizaje automático.

Este proceso consta de varias fases. La primera de ellas calcula el vector $logits_t$ (nombre comúnmente utilizado para los datos de entrada de las funciones de activación) obtenidos a partir de la suma de los vectores de entrada x_t, estado anterior h_{t-1} y umbral b_h, donde los dos primeros van multiplicados por los pesos W_{xh} y W_{hh}, respectivamente (ver ecuación 3.1).

$$\boldsymbol{logits_t = x_t W_{xh} + h_{t-1} W_{hh} + b_h} \tag{3.1}$$

La segunda fase consiste en aplicar la función sigmoidea σ al vector $logits_t$ para obtener el vector z_t (ver ecuación 3.2). Este último vector está compuesto de valores comprendidos en el rango $(0,1)$ y representa la probabilidad de activación de cada una de las neuronas.

$$\boldsymbol{z_t = \sigma(logits_t)} \tag{3.2}$$

La fase de muestreo tiene como objetivo aplicar la regularización de estados descrita al inicio de esta sección y puede ser llevada a cabo mediante dos procesos distintos. La primera variante consiste en generar tantos números aleatorios como neuronas tenga la capa recurrente mediante una distribución uniforme con media cero y varianza uno $U(0,1)$. De esta manera se generarán tantos números como indique el índice i y con el mismo rango de valores que la función sigmoidea. Con esto se cumple que h_{ti} es igual a uno si z_{ti} es mayor que el número aleatorio correspondiente o igual a cero en caso contrario (ver ecuación 3.3). Por otro lado, la segunda manera de muestrear aplica un sesgo constante, donde h_{ti} será igual a uno si z_{ti} es mayor que 0.5 o igual a cero en caso contrario (ver ecuación 3.4).

$$h_{ti} = \begin{cases} 0 & z_{ti} \leq U(0,1) \\ 1 & z_{ti} > U(0,1) \end{cases} \quad i \in [1, nunits] \tag{3.3}$$

$$h_{ti} = \begin{cases} 0 & z_{ti} \leq 0.5 \\ 1 & z_{ti} > 0.5 \end{cases} \quad i \in [1, nunits] \tag{3.4}$$

La primera variante se aplica durante la fase de entrenamiento, esto se debe a que la elección aleatoria resulta ser la mejor estrategia al no tener conocimiento previo sobre qué neuronas deben activarse. Sin embargo, durante la fase de validación los pesos ya tienen

valores definitivos, por lo que resulta más óptimo activar las neuronas de mayor magnitud y descartar el resto. En este caso se eligió un sesgo de 0.5 debido a que es la media del rango de la función sigmoidea, de forma que cualquier valor por encima de este sesgo se supone que es suficientemente alto como para activar la neurona correspondiente.

La última fase calcula la salida global y_t, esto se logra mediante el producto entre el vector del estado actual h_t, que resulta del muestreo de z_t, y su matriz de pesos asociada W. A este producto se le suma el vector de umbral b y todo ello se utiliza como entrada para la función de activación softmax (ver ecuaciones 3.5 y 3.6).

$$\boldsymbol{y_t} = softmax(\boldsymbol{h_t}W + \boldsymbol{b}) \tag{3.5}$$

$$softmax(\boldsymbol{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \tag{3.6}$$

En este caso particular, aunque se esté utilizando la función softmax para calcular y_t, también se podría emplear la función sigmoidea. Esto se debe a que el modelo solo necesita tomar una decisión entre dos clases: uno (aceptación) o cero (rechazo). La diferencia entre el uso de una función u otra radica en que la primera calcula tanto la probabilidad de aceptación como la de rechazo, mientras que la segunda función únicamente calcula la probabilidad de aceptación y la de rechazo se obtendría mediante el complemento de esta.

El objetivo del modelo es forzar que el comportamiento interno de la red neuronal recurrente sea equivalente al de un autómata finito determinista. De forma que la red transite a un estado h_t desde otro estado h_{t-1} a partir del símbolo x_t, donde la aceptación o rechazo del estado h_t queda definida por la salida y_t.

Para facilitar la comprensión del diseño descrito a lo largo de esta sección se utiliza la figura 3.4. Esta figura ejemplifica el flujo de los datos durante la validación del modelo cuando el número de neuronas de la capa recurrente (*nunits*) es igual a uno, los componentes de los vectores b_h y b son igual a cero (no se utilizan umbrales) y los valores de las matrices de pesos son elegidos aleatoriamente. Antes de comenzar con el ejemplo es importante mencionar que un modelo de aprendizaje automático únicamente puede ser alimentado con datos numéricos. Debido a lo anterior, se decidió utilizar la codificación *one-hot* para convertir cada símbolo de entrada en un vector donde sólo uno de sus componentes es igual a uno y el resto es igual a cero. Como consecuencia, en este proyecto los símbolos \$, a y b se representan mediante los vectores $(1,0,0)$, $(0,1,0)$ y $(0,0,1)$ respectivamente.

El ejemplo de la figura 3.4 comienza recibiendo como entrada x_t el símbolo \$ en su respectiva codificación *one-hot*. Este símbolo se multiplica por su matriz de pesos asociada W_{xh} y a este resultado se le suma el producto entre el estado anterior h_{t-1} y la matriz W_{hh} para calcular el valor de $logits_t$. El proceso continúa transformando $logits_t$ en la probabilidad de

activación z_t mediante la función sigmoidea. El siguiente paso aplica la función de muestreo comparando cada uno de los componentes del vector z_t, que para este caso en específico cuenta con un único componente de valor 0.51, con el sesgo 0.5. Debido a que se cumple la condición del muestreo, el estado actual h_t toma como valor (1). Para terminar, se aplica la función softmax al producto entre el estado actual h_t y la matriz W con el objetivo de calcular las probabilidades y_t. El vector y_t tiene dos componentes, el primero indica la probabilidad de rechazo y el segundo la de aceptación, ambos con respecto al estado actual h_t. Esto se traduce para el ejemplo de la figura en un 61% de probabilidades de rechazo para el estado $h_t = (1)$.

Debido a que la RNR de este ejemplo está compuesta de una única neurona, el estado h_t sólo puede tomar dos valores: (1) si se cumple la condición del muestreo o (0) en caso contrario. En general, habrá 2^{nunits} estados posibles, donde cada componente h_{ti} del estado h_t será igual a 1 si su respectiva probabilidad z_{ti} cumple con la condición del muestreo aplicado.

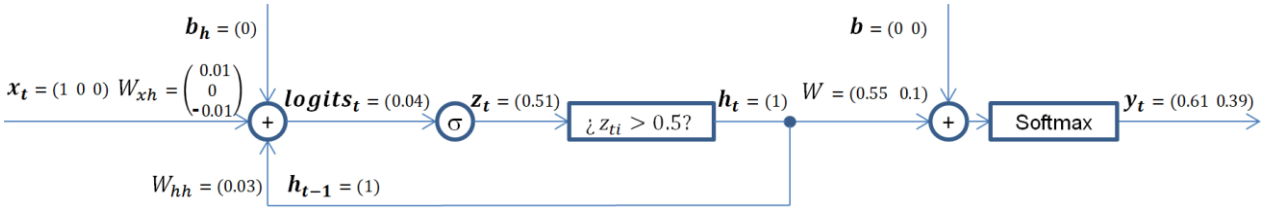


Figura 3.4: Ejemplo del funcionamiento del modelo de aprendizaje automático.

3.3. Descripción de los experimentos

El experimento realizado en este trabajo de fin de grado consistió en aplicar el modelo de aprendizaje automático, descrito en la subsección anterior, a cada uno de los conjuntos de datos de la tabla 3.2 disponibles para cada una de las gramáticas. El conjunto de conjunto de datos *entrenamiento* se utiliza durante la fase que lleva el mismo nombre, mientras que los otros cuatro conjuntos de datos se emplean para la fase de validación. Este proceso se repitió cinco veces con el objetivo de calcular medias y desviaciones estándar para comprobar empíricamente la estabilidad del modelo. También se mostrarán las gráficas, calculadas durante el entrenamiento, relativas a la minimización de la función de coste y a la maximización de la precisión del modelo a la hora de predecir las salidas correspondientes a las entradas que recibe. Por último, se presentarán los AFD extraídos a partir de los pesos de la red con el objetivo de interpretar el aprendizaje del modelo.

4 Implementación

Una vez descrito el diseño del modelo y el contexto sobre el que trabaja el mismo, corresponde a este capítulo documentar en profundidad cómo se puso en práctica toda la funcionalidad necesaria para llevar a cabo este proyecto. En particular, se describirán las herramientas utilizadas y el código generado para implementar el modelo de la figura 3.3, se detallarán minuciosamente los procesos de entrenamiento y validación, y se expondrá el método utilizado para extraer los AFD correspondientes a cada una de las gramáticas inferidas.

4.1. Implementación del modelo

En la mayoría de los problemas de aprendizaje automático es casi imposible construir un modelo óptimo al primer intento. La técnica a seguir suele ser ensayo y error, en donde se prueba con una primera aproximación y se van ajustando los parámetros necesarios hasta conseguir los resultados deseados. Este fue el procedimiento utilizado a lo largo del desarrollo de este proyecto y en el presente capítulo únicamente se detalla la implementación final del modelo.

Para la construcción del modelo llevado a cabo en este trabajo se emplearon distintas herramientas de Python tales como *Tensorflow* [39] y *Keras* [40]. Ambas son potentes bibliotecas de código abierto enfocadas al diseño de modelos de aprendizaje automático y redes neuronales artificiales.

4.1.1. Capas

En *Keras*, una capa es la unidad básica de construcción de una red neuronal. Cada capa representa una función que recibe un tensor (objeto matemático generalizado para representar datos multidimensionales) como entrada y produce otro a la salida. Además, las capas tienen una serie de parámetros que se ajustan durante el proceso de entrenamiento. Estos parámetros son los encargados del aprendizaje y se expresan mediante los pesos y umbrales correspondientes.

Al implementar el diseño de la figura 3.3 mediante la *Functional API* [41] de *Keras* se obtuvo el modelo de la figura 4.1. En este se pueden observar tres capas diferenciadas: entrada, recurrente y densa. La primera de ellas se encarga de recibir los datos del fichero de entrada en codificación *one-hot* y suministrarlos a la siguiente capa en forma de tensor.

La segunda capa es la recurrente, esta calcula los vectores $logits_t$ y z_t de acuerdo a las ecuaciones 3.1 y 3.2 descritas anteriormente. Además, esta capa también genera el estado actual h_t al someter al vector de activaciones z_t a alguno de los dos procesos de muestreo definidos mediante las ecuaciones 3.3 y 3.4.

Por último, la capa densa (también conocida como capa totalmente conectada) calcula la salida y_t de acuerdo a la anterior ecuación 3.5. En resumen, el modelo desarrollado consta de tres capas: entrada, recurrente y densa; una única entrada: x_t y tres salidas: $logits_t$, h_t e y_t.

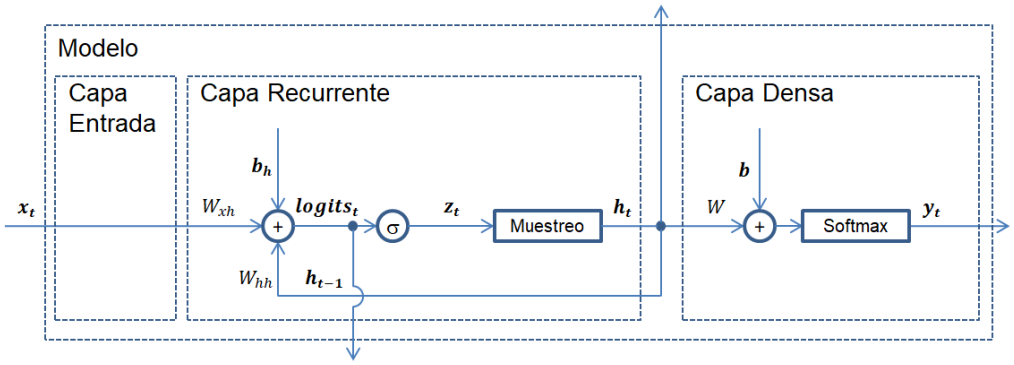


Figura 4.1: Implementación del modelo de aprendizaje automático por capas.

La figura 4.1 es equivalente al bloque de código 4.1. En este se puede apreciar cómo se construye el modelo en la línea 8 a partir de las capas de entrada, recurrente y densa instanciadas en las líneas 2, 3 y 7, respectivamente. La primera capa establece la dimensión de entrada del modelo, tema que se abordará en la subsección 4.1.2. La segunda capa se crea a partir de una celda recurrente personalizada con *nunits* neuronas, cuya explicación será descrita en la subsección 4.1.3. Además, esta capa establece los valores de dos parámetros a *True*, donde *return_sequences* (línea 4) indica que es necesario calcular la salida en cada instante de tiempo y *stateful* (línea 5) denota que es necesario suministrar el último estado del lote anterior al siguiente. Por último, la capa densa tiene *output_dim* neuronas y calcula las probabilidades de aceptación y rechazo del estado actual mediante la función de activación softmax. Esta última capa incluye por defecto la matriz de pesos W de dimensión *(nunits, output_dim)* inicializada con una distribución Glorot uniforme [42], lo que genera muestras comprendidas en el rango $[-limit, limit]$ de acuerdo a la ecuación 4.1, y el vector de umbral b de dimensión *(output_dim)* con valor inicial cero en todos sus componentes.

$$limit = \sqrt{\frac{6}{nunits + output\_dim}} \tag{4.1}$$

```
1  def create_model(nunits, batch_size, seq_len, input_dim, output_dim, train):

2      inputs = keras.Input(batch_shape=(batch_size, seq_len, input_dim))
3      logits, h = RNN(SampleRNNCell(nunits, train),
4                      return_sequences=True,
5                      stateful=True
6                     )(inputs)
7      y = Dense(output_dim, activation='softmax')(h)
8      model = keras.Model(inputs=inputs, outputs=[logits, h, y])

9      return model
```

Código 4.1: Implementación del modelo en *Keras*.

4.1.2. Dimensiones

En la figura 4.2 se pueden observar las dimensiones de las entradas y salidas de cada una de las capas que conforman el modelo. Debido a que cada capa está a continuación de la otra, la salida de cualquiera de ellas tiene que tener la misma dimensión que la entrada de la siguiente. Por este motivo se comentan a continuación únicamente las dimensiones de salida, ya que las de entrada se pueden recuperar fácilmente a partir de los resultados de la capa anterior. La salida de la capa de entrada tiene dimensión *(batch_size, seq_len, input_dim*), mientras que la salida de la capa recurrente está conformada por dos datos de dimensión *(batch_size, seq_len, nunits)* y, por último, la salida de la capa densa tiene dimensión *(batch_size, seq_len, output_dim)*. Cabe destacar que la salida de la capa recurrente no es exactamente igual a la entrada de la capa densa porque esta última recibe únicamente el estado actual h, ya que el vector $logits$ sólo es necesario para entrenar el modelo.

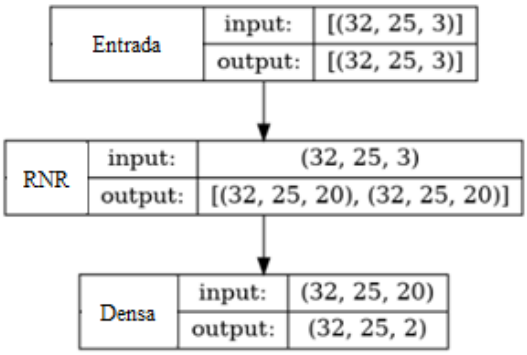


Figura 4.2: Dimensiones de las capas del modelo de aprendizaje automático.

Los significados de los parámetros que se utilizan para definir las dimensiones de las capas son los siguientes: *input_dim* representa la cantidad de caracteres distintos en el conjunto de entrada $\{\$, a, b\}$; *output_dim* tiene la misma función pero con el conjunto de salida $\{0,1\}$; *batch_size* representa el tamaño del lote, es decir, la cantidad de conjuntos de tamaño *seq_len* que son leídos en cada iteración del entrenamiento; *seq_len* fija el número de símbolos en codificación *one-hot* de la secuencia de entrada, lo que establece el límite de instantes de tiempo

a los cuales propagar el error hacia atrás al calcular el gradiente y *nunits* representa la cantidad de neuronas de la capa recurrente. En cuanto a sus valores numéricos, *input_dim* y *output_dim* siempre valdrán 3 y 2 respectivamente, mientras que *batch_size*, *seq_len* y *nunits* podrían variar pero será habitual a lo largo de este proyecto que tomen los valores 32, 25 y 20 respectivamente, porque con ellos se consiguieron los mejores resultados.

4.1.3. Celda recurrente

Al implementar la capa recurrente fue necesario hacerlo a través de una celda recurrente utilizando la *Layers API* [43] de *Keras.* Esta celda es una clase cuyo nombre es *SampleRNNCell* y que se define mediante el código 4.2.

```
1  class SampleRNNCell(keras.layers.Layer):

2      def __init__(self, nunits, train, **kwargs):
3          self.nunits = nunits
4          self.train = train
5          self.state_size = tf.TensorShape([nunits])
6          super(SampleRNNCell, self).__init__(**kwargs)

7      def get_config(self):
8          config = super().get_config().copy()
9          config.update({'nunits': self.nunits, 'train': self.train})
10         return config

11     def build(self, input_shape):
12         self.Wxh = self.add_weight(shape=(input_shape[-1], self.nunits),
13                                    initializer='uniform', name='kernel')
14         self.Whh = self.add_weight(shape=(self.nunits, self.nunits),
15                                    initializer='uniform', name='recurrent_kernel')
16         self.b = self.add_weight(shape=(self.nunits,),
17                                  initializer='uniform', name='bias')
18         self.built = True

19     def call(self, inputs, states):
20         h_prev = states[0]
21         x = inputs
22         logits = keras.backend.dot(x, self.Wxh) +
23                  keras.backend.dot(h_prev, self.Whh) + self.b
24         z = tf.math.sigmoid(logits)
25         if self.train:
26             h = tf.cast(tf.random.uniform(z.shape) < z, dtype=tf.float32)
27         else:
28             h = tf.cast(0.5 < z, dtype=tf.float32)
29         return [logits, h], [h]
```

Código 4.2: Implementación de la celda recurrente en *Keras*.

Esta celda hereda de la clase base *keras.layers.Layer* y para instanciarla es necesario asignar el número de neuronas a utilizar e indicar si se utilizará para la fase de entrenamiento o validación. La celda posee un constructor y tres métodos que serán explicados a continuación. El método *get_config* de la línea 7 implementa la opción de poder guardar, y posteriormente recuperar, en un fichero de datos los pesos de un modelo ya entrenado. El método *build* de la línea 11 declara tres parámetros: los pesos W_{xh} de dimensión (*input_dim, nunits*) asociados a los datos de entrada, los pesos W_{hh} de dimensión (*nunits, nunits*) asociados a los estados y el vector de umbral b_h de dimensión (*nunits*), todos ellos inicializados mediante una distribución aleatoria uniforme en el rango $[-0.05, 0.05)$. Por último, el método *call* de la línea 19 define todos los cálculos que realiza la celda. Las líneas 22 y 23 son equivalentes a la ecuación 3.1, la línea 24 se corresponde con la ecuación 3.2 y las líneas 26 y 28 implementan las ecuaciones 3.3 y 3.4, respectivamente.

4.2. Descripción de la fase de entrenamiento

El entrenamiento del modelo (ver código A.1) consiste en un bucle anidado, donde el bucle externo se encarga de iterar por el número de épocas y el interno recorre los datos de entrada troceándolos según indique el tamaño del lote. Es importante recordar que en aprendizaje automático una época denota el paso completo del conjunto de datos de entrenamiento por el algoritmo, mientras que el lote controla la cantidad de muestras de este conjunto con las que trabajar a la vez a la hora de actualizar los pesos del modelo.

El bucle interno comienza aplicando un lote de datos de entrada al modelo para así calcular el estado actual h, el vector $logits$ y las probabilidades de aceptación o rechazo y_pred asociadas al estado h. A partir de este momento el entrenamiento se puede dividir en dos partes diferenciadas, una basada en el paradigma de aprendizaje supervisado y otra enfocada en el aprendizaje por refuerzo. La primera parte determina la discrepancia entre las probabilidades recién calculadas y las salidas correctas mediante la función de coste entropía cruzada categórica, cuyas fórmulas, simplificadas para un tamaño de lote igual a uno, se representan mediante las ecuaciones 4.2 y 4.3.

$$loss\_dense_j = -\sum_{k=1}^{output\_dim} y\_true_{jk} log_e(y\_pred_{jk}) \quad j \in [1, seq\_len] \tag{4.2}$$

$$loss\_dense\_mean = \frac{\sum_{j=1}^{seq\_len} loss\_dense_j}{seq\_len} \tag{4.3}$$

La segunda parte del entrenamiento estandariza el coste de la capa densa para utilizarlo como la recompensa r del aprendizaje por refuerzo. El objetivo consiste en evaluar el desempeño de cada elemento del lote en relación con el resto, de forma que se asigna una recompensa alta (positiva) a aquellos de máximo rendimiento y una recompensa baja (negativa)

a aquellos con el rendimiento mínimo. Con esto se logra que la red recuerde las mejores decisiones y olvide las peores.

Una vez obtenida la recompensa se puede computar el coste de la capa recurrente, esto se consigue calculando la entropía cruzada binaria entre h y $logits$, proceso que busca que la red tome la misma decisión ante las mismas condiciones, y multiplicando este resultado por la recompensa r correspondiente, siendo esta última la que define si debe recordarse u olvidarse dicha decisión. Estas operaciones se realizan para un tamaño de lote igual a uno mediante las ecuaciones 4.4, 4.5 y 4.6.

$$loss\_rnn_j = \frac{\sum_{k=1}^{nunits} max(logits_{jk}, 0) - h_{jk} * logits_{jk} + log_e(1 + e^{-|logits_{jk}|})}{nunits} \quad j \in [1, seq\_len]$$

(4.4)

$$loss\_rnn\_r_j = loss\_rnn_j * r \quad j \in [1, seq\_len]$$

(4.5)

$$loss\_rnn\_r\_mean = \frac{\sum_{j=1}^{seq\_len} loss\_rnn\_r_j}{seq\_len}$$

(4.6)

La razón por la cual se utilizan funciones de coste diferentes para las capas recurrente y densa es la presentación disponible de los datos. En el caso de la capa densa se utiliza una entropía cruzada categórica debido a que los datos y_true están en codificación *one-hot*, donde $(1,0)$ indica rechazo y $(0,1)$ denota aceptación. Además, los datos y_pred son de la forma (p_1, p_2), donde se cumple que $p_1 + p_2 = 1$ debido a que y_pred es el resultado de una función softmax. Por otro lado, en la capa recurrente se utiliza una entropía binaria cruzada para comparar cada componente del estado h con cada componente del vector $logits$. Ambos son vectores con *nunits* componentes, donde los de h pertenecen al conjunto $\{0,1\}$ y los de $logits$ pertenecen al conjunto de los números reales $\mathbb{R}$.

Una vez que los costes de ambas capas fueron calculados, se obtienen los gradientes asociados a estos y se aplican al modelo mediante optimizadores NAdam [44]. De forma que los parámetros W_{xh}, W_{hh} y b_h de la capa recurrente son entrenados con técnicas de aprendizaje por refuerzo, mientras que los parámetros W y b de la capa densa se entrenan mediante aprendizaje supervisado. Esta combinación de métodos se debe a que la operación de muestreo que realiza la capa recurrente no es diferenciable, lo que impide que el gradiente se pueda retropropagar más allá de la capa densa. Por lo tanto, el propio diseño del modelo fuerza la búsqueda de alternativas para actualizar los pesos y umbrales de la capa recurrente, lo que derivó en este proyecto la implementación del aprendizaje por refuerzo.

Por último, el entrenamiento se evalúa en cada una de las épocas en términos del coste medio de la capa densa (ver ecuación 4.3) y de la precisión obtenida (ver ecuación 4.9). La precisión en un problema de clasificación se define como el número correcto de predicciones

entre el número total de predicciones realizadas. La ecuación 4.9 representa el caso donde el tamaño del lote es igual a uno y requiere haber calculado previamente tanto el máximo entre las probabilidades de aceptación y rechazo de cada símbolo de la longitud de secuencia (ver ecuación 4.7), como las clases (aceptación o rechazo) con las que estos serán comparados (ver ecuación 4.8). Tanto el coste medio como la precisión obtenidos se utilizan para generar las gráficas y tablas correspondientes al capítulo 5.

$$predictions_j = \underset{k}{\text{argmax}} \; y\_pred_{jk} \quad k \in [1, output\_dim], j \in [1, seq\_len] \tag{4.7}$$

$$classes_j = \underset{k}{\text{argmax}} \; y\_true_{jk} \quad k \in [1, output\_dim], j \in [1, seq\_len] \tag{4.8}$$

$$accuracy = \frac{\sum_{j=1}^{seq\_len} predictions_j == classes_j}{seq\_len} \tag{4.9}$$

4.3. Descripción de la fase de validación

La fase de validación (ver código A.2) realiza un subconjunto de las operaciones definidas durante la sección anterior, de forma que la explicación de esta segunda fase hará referencia a aquellos conceptos previamente introducidos según sea necesario.

Antes de comenzar con la fase de validación es necesario crear con antelación un nuevo modelo asignando a *False* el argumento *train* de la celda recurrente. Esto es debido a que el proceso de muestreo no es el mismo para el entrenamiento que para la validación, por lo que se le debe indicar al modelo cuál de los dos debe llevar a cabo. Una vez creado el nuevo modelo se deben asignar a este los pesos del modelo previamente entrenado (ver código A.3) y de esta forma dar comienzo a la fase de validación.

La validación consiste en un bucle que trocea los datos de entrada según indique el tamaño del lote. Estos datos deben pertenecer a alguno de los cuatro conjuntos de validación de la tabla 3.2. Esta segunda fase únicamente calcula el coste medio de la capa de salida (ver ecuaciones 4.2 y 4.3) y la precisión de las predicciones realizadas (ver ecuaciones 4.7, 4.8 y 4.9) para estos nuevos datos de entrada. Estas métricas se utilizan en el capítulo 5 para compararlas con las de la fase de entrenamiento y comprobar la capacidad de generalización del modelo frente a datos nunca antes vistos.

4.4. Método para extraer los AFD

En secciones anteriores se introdujo que las redes recurrentes con regularización de estado facilitan la extracción del AFD que las define, proceso que permite que el funcionamiento interno de la red sea fácilmente interpretable por el humano. Esta extracción se llevó a cabo mediante el descubrimiento de todas las transiciones de la red y la distinción entre aquellos estados que son finales de los que no.

El método (ver código A.4) se implementó mediante una modificación del algoritmo de búsqueda en anchura, donde al comienzo de cada iteración del bucle se configura el estado actual de la capa recurrente como el estado extraído del inicio de la cola de estados pendientes. A continuación, se aplican al modelo las cadenas de entrada \$, a y b individualmente en codificación *one-hot* y se obtienen las salidas correspondientes. Estas salidas representan los estados a los que se transitan desde el estado actual a partir de las cadenas de entrada mencionadas anteriormente y las probabilidades de que estos nuevos estados sean finales o no. De esta forma, el algoritmo descubre la función de transición y los estados finales del autómata que define el comportamiento de la RNR. El bucle termina añadiendo a la cola de estados pendientes aquellos estados que no hayan sido visitados en iteraciones anteriores, de forma que el algoritmo parará una vez que hayan sido descubiertos todos los estados que conforman el autómata.

Una vez obtenidos todos los datos que conforman la quíntupla del AFD, se construye el autómata correspondiente, se minimiza y se traduce a lenguaje DOT (ver código A.5) para poder representarlo mediante el software de código abierto para visualización de grafos *Graphviz* [45].

5 Resultados

Tras haber descrito en profundidad el modelo de aprendizaje automático desarrollado para este proyecto, corresponde a este capítulo documentar los resultados obtenidos para cada uno de los conjuntos de datos de la tabla 3.2, disponibles para cada una de las gramáticas de Tomita definidas en la tabla 3.1. Los resultados suministrados incluyen datos numéricos en forma de tabla, gráficos y los AFD extraídos para cada una de las gramáticas, así como un análisis sobre los aspectos más relevantes de los mismos.

Todos los resultados presentados en este capítulo se obtuvieron mediante los bloques de código del capítulo 4 y del apéndice A, para valores de *batch_size*, *seq_len* y *nunits* iguales a 32, 25 y 20 respectivamente. Es importante mencionar que no hizo falta realizar una búsqueda exhaustiva para conseguir estos valores. Esto es debido a que el modelo no se ve afectado drásticamente por la elección de estos hiperparámetros y permite conseguir rápidamente valores que proporcionan los resultados esperados.

5.1. Rendimiento obtenido para cada gramática

La tabla 5.1 resume la media y la desviación estándar del coste (ver ecuación 4.3) y de la precisión (ver ecuación 4.9) de cinco entrenamientos, cada uno con el número de épocas indicado en la segunda columna, para cada una de las gramáticas de Tomita. El número de épocas con las que se entrenó el modelo para cada gramática comenzó en un valor fijo de quinientos y se fue incrementando hasta que ya no se observaron mejoras significativas en los resultados. Es importante resaltar que la precisión de la penúltima columna se obtiene al aplicar la ecuación 3.3 para el muestreo, lo que acarrea un pequeño error al introducir un factor de aleatoriedad. Por lo tanto, se decidió incorporar la última columna para mostrar los resultados obtenidos al aplicar la ecuación 3.4, la cual activa las neuronas cuya probabilidad es mayor que 0.5 y apaga el resto. Se puede apreciar que todos los modelos, a excepción de los entrenados con las gramáticas 5 y 6, alcanzan un coste cercano al cero y una precisión que tiende a uno.

Para confirmar la capacidad de generalización del modelo, cada uno de los cinco entrenamientos realizados para cada gramática fue puesto a prueba mediante los conjuntos de

validación *grande*, *largo*, *muchas_a* y *muchas_b*. Las tablas 5.2 y 5.3 resumen la media y la desviación estándar del coste y de la precisión de estas validaciones. Los resultados correspondientes a los conjuntos de datos *grande* y *largo* se presentan en la tabla 5.2, mientras que los relativos a los conjuntos de datos *muchas_a* y *muchas_b* se recopilan en la 5.3. Cabe recalcar que los resultados de estas tablas se obtienen al aplicar la ecuación 3.4 y por ello las precisiones expuestas guardan cierta similitud con la última columna de la tabla 5.1.

Gramática	**Épocas**	**Coste ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**
Tomita1	1000	0.0006 ± 0.0007	0.99993 ± 0.00007	1 ± 0
Tomita2	2000	0.0001 ± 0.0002	0.99999 ± 0.00002	1 ± 0
Tomita3	2500	0.1 ± 0.1	0.97 ± 0.05	0.99 ± 0.02
Tomita4	2000	0.014 ± 0.008	0.998 ± 0.001	1 ± 0
Tomita5	1000	0.2983 ± 0.0007	0.7876 ± 0.0009	0.786 ± 0.001
Tomita6	1000	0.5187 ± 0.0005	0.7224 ± 0.0002	0.72224 ± 0
Tomita7	2000	0.026 ± 0.008	0.994 ± 0.002	1 ± 0

Tabla 5.1: Resultados del entrenamiento para cada gramática de Tomita.

	Grande		**Largo**	
Gramática	**Coste ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**	**Coste ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**
Tomita1	0.00007 ± 0.00002	1 ± 0	0.00008 ± 0.00003	1 ± 0
Tomita2	0.000005 ± 0.000002	1 ± 0	0.000005 ± 0.000003	1 ± 0
Tomita3	0.03 ± 0.07	0.99 ± 0.02	0.02 ± 0.04	0.997 ± 0.007
Tomita4	0.0015 ± 0.0008	1 ± 0	0.0011 ± 0.0007	1 ± 0
Tomita5	0.2955 ± 0.0002	0.787 ± 0.001	0.3415 ± 0.0002	0.754 ± 0.001
Tomita6	0.5178 ± 0.0007	0.7215 ± 0	0.6246 ± 0.0002	0.6728 ± 0
Tomita7	0.0026 ± 0.0009	1 ± 0	0.0022 ± 0.0007	1 ± 0

Tabla 5.2: Resultados de la validación con los conjuntos de datos *grande* y *largo.*

	Muchas_a		**Muchas_b**	
Gramática	**Coste ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**	**Coste ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**
Tomita1	0.00004 ± 0.00002	1 ± 0	0.00008 ± 0.00003	1 ± 0
Tomita2	0.000005 ± 0.000003	1 ± 0	0.000006 ± 0.000003	1 ± 0
Tomita3	0.02 ± 0.04	1 ± 0	0.1 ± 0.2	0.97± 0.07
Tomita4	0.0008 ± 0.0005	1 ± 0	0.0011 ± 0.0008	1 ± 0
Tomita5	0.34 ± 0.002	0.77 ± 0.09	0.34 ± 0.002	0.77 ± 0.09
Tomita6	0.6255 ± 0.0007	0.6744 ± 0	0.6266 ± 0.0006	0.674 ± 0
Tomita7	0.005 ± 0.002	1 ± 0	0.001 ± 0.0004	1 ± 0

Tabla 5.3: Resultados de la validación con los conjuntos de datos *muchas_a* y *muchas_b*.

Analizando estos resultados, se puede concluir que el modelo es capaz de inferir correctamente todas las gramáticas de Tomita a excepción de la 5 y la 6. Es oportuno mencionar que la gramática 3 obtuvo resultados un poco peores que el resto de gramáticas correctamente inferidas debido a que uno de sus cinco entrenamientos no produjo los resultados esperados. Sin embargo, el resto de sus ejecuciones convergieron adecuadamente.

5.2. Gráficos del entrenamiento para cada gramática

En las figuras 5.1 y 5.2 se presentan los gráficos relativos a la evolución del coste y de la precisión conforme aumenta el número de épocas. Los gráficos corresponden únicamente al entrenamiento que produjo los mejores resultados, ya que la idea es observar la convergencia del modelo. En la práctica, esta decisión únicamente afecta a la gramática 3, ya que el resto de gramáticas produjeron curvas muy similares en sus cinco entrenamientos.

Se puede observar que las gramáticas de Tomita 1 y 2 convergen rápidamente (coste cercano a cero y precisión que tiende a uno), mientras que las 3, 4 y 7 lo hacen de forma más progresiva. Nuevamente se aprecia como las gramáticas 5 y 6 tienen un peor rendimiento que el resto y que este no mejora significativamente aunque aumente el número de épocas.

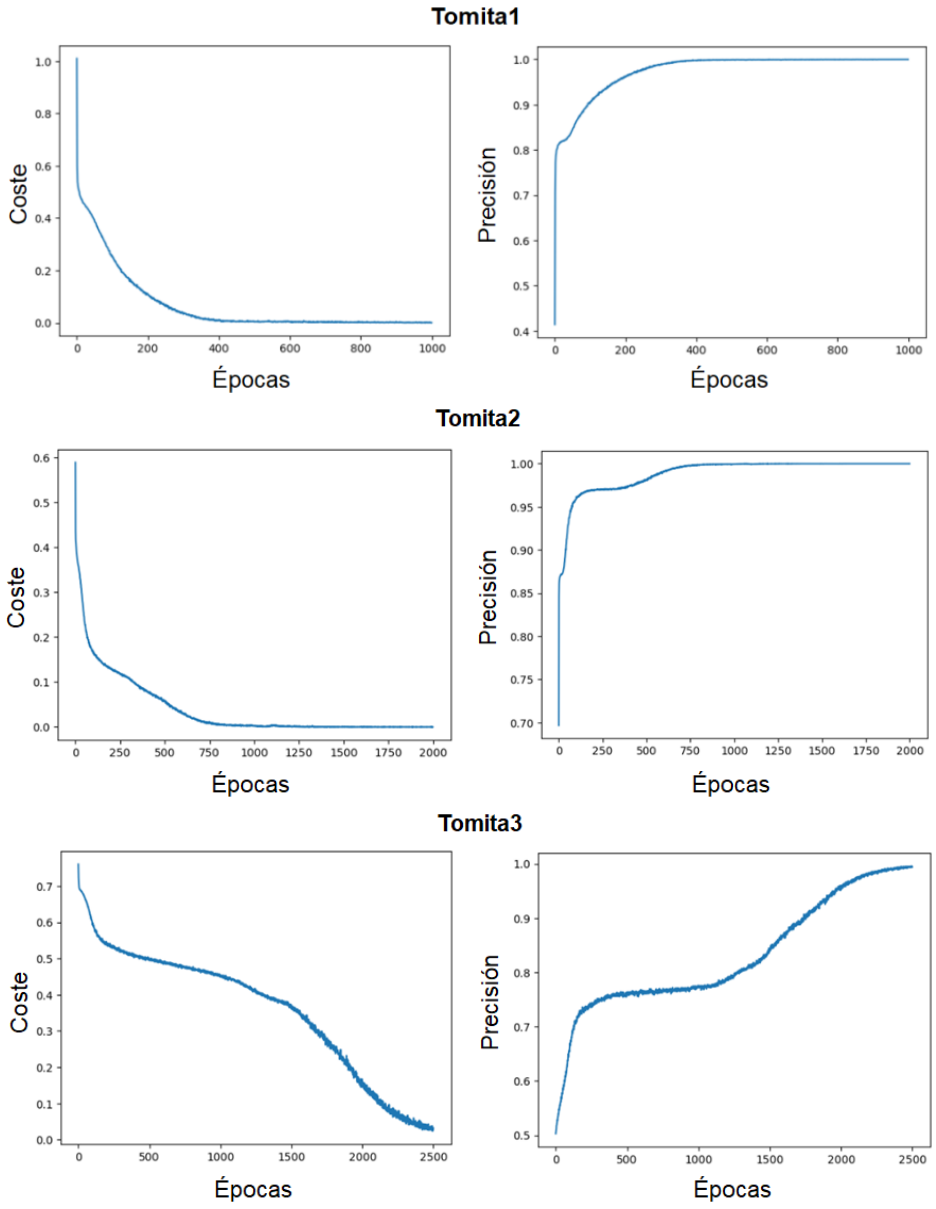


Figura 5.1: Evolución del coste y de la precisión por época para Tomita 1, 2 y 3.

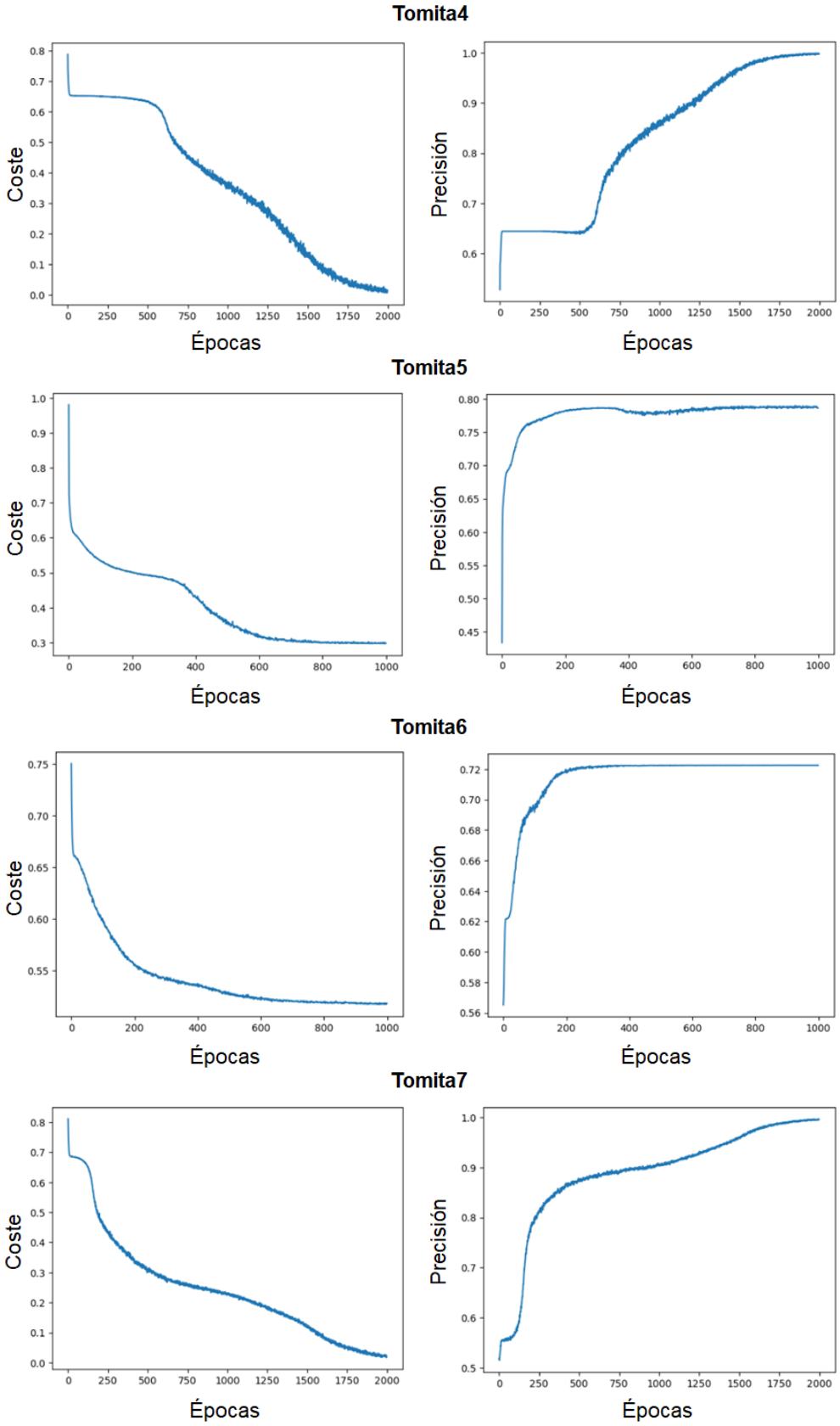


Figura 5.2: Evolución del coste y de la precisión por época para Tomita 4, 5, 6 y 7.

5.3. AFD extraídos

En la figura 5.3 se exponen los AFD mínimos extraídos para cada una de las gramáticas de Tomita mediante el método descrito en la sección 4.4. Estos autómatas corresponden únicamente al entrenamiento que produjo los mejores resultados para cada gramática. Si se compara esta figura con la 3.1, se puede observar que los autómatas extraídos para las gramáticas 1, 2, 3, 4 y 7 coinciden con los autómatas teóricos. Es importante mencionar que se eliminaron manualmente todas las transiciones correspondientes al símbolo $ de los autómatas de la figura 5.3 con el objetivo de mejorar la legibilidad de los mismos. Estas transiciones originalmente iban desde todos los estados de cada autómata hacia el estado inicial del mismo.

Sin embargo, los autómatas extraídos para las gramáticas de Tomita 5 y 6 no se corresponden con sus análogos de la figura 3.1. El primero de estos autómatas acepta el lenguaje definido mediante la expresión regular $((a+b)(a+b))^*$, la cual describe cualquier cadena de longitud par formada por aes o bes. En cambio, debería describir cualquier cadena de longitud par con un número par de aes. La precisión obtenida en la sección 5.1 se debe a que el lenguaje asociado a la gramática de Tomita 5 es un subconjunto del definido por el autómata extraído y, por lo tanto, existen cadenas que pertenecen a ambos lenguajes.

Por otro lado, el autómata extraído para la gramática de Tomita 6 sólo acepta la cadena vacía λ. No obstante debería aceptar las cadenas donde la diferencia entre el número de aes y bes es múltiplo de tres. Para explicar la precisión obtenida en la sección 5.1, se utiliza el hecho de que si se elige un número natural al azar existe $1/3$ de posibilidades de que sea múltiplo de tres. Por lo tanto, si el autómata extraído rechaza todas las cadenas no vacías acertará $2/3$ de las veces. Además, esta proporción se ve incrementada al aceptar las cadenas vacías, ya que la gramática de Tomita 6 también lo hace.

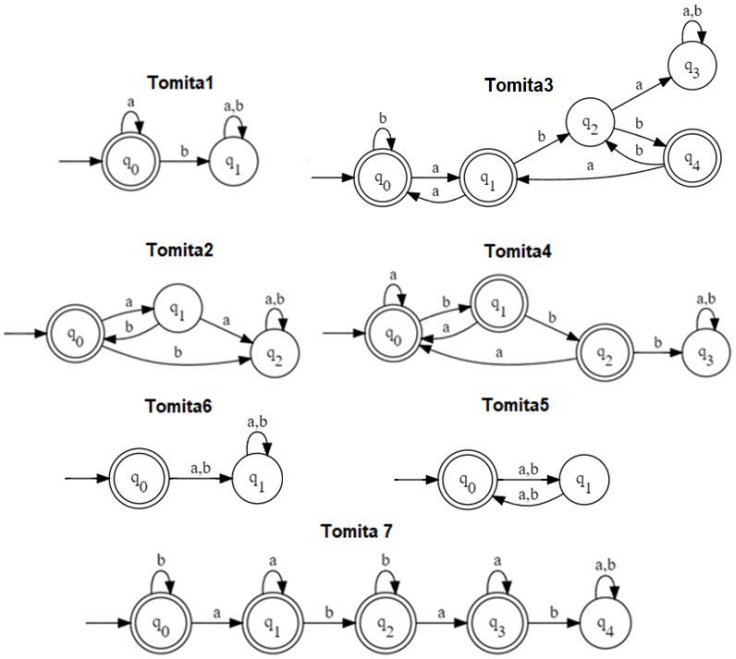


Figura 5.3: AFD mínimos extraídos para cada gramática de Tomita.

5.4. Búsqueda de mejoras para gramáticas de Tomita 5 y 6

En vista de que los resultados obtenidos para las gramáticas 5 y 6 no fueron satisfactorios, se aplicaron ciertas modificaciones al modelo desarrollado con el objetivo de mejorar su rendimiento. Concretamente se realizaron las siguientes modificaciones:

- Se añadió una capa densa adicional con una función de activación ReLU y un número de neuronas comprendido en el rango $[10, 30]$, situada entre la capa recurrente y la de salida.
- Se añadió un regularizador L1, con parámetro comprendido en el rango $[0.0001, 0.1]$, a la capa densa.
- Se probaron distintos valores, comprendidos en el rango $[10, 500]$, para la cantidad de neuronas de la capa recurrente.
- Se probaron distintos valores, comprendidos en el rango $[1, 320]$, para el tamaño del lote.
- Se probaron distintos valores, comprendidos en el rango $[1, 250]$, para la longitud de secuencia.

Sin embargo, estos cambios no produjeron mejoras en el rendimiento del modelo. Debido a esto se realizó un último experimento donde sólo se aumentó el número de épocas. La estrategia utilizada consistió en entrenar el modelo empleando intervalos de cinco mil épocas y analizando las curvas de coste y precisión obtenidas para cada uno de estos. Este procedimiento se aplicó individualmente a las gramáticas 5 y 6 y se extendió hasta un total de veinticinco mil épocas, ya que durante este intervalo se dejaron de apreciar mejoras significativas en los resultados obtenidos. En la figura 5.4 se puede apreciar que la mejora en la precisión durante el último intervalo es sólo de 2.33% y 1.26% para las gramáticas 5 y 6, respectivamente.

Es oportuno mencionar que el entrenamiento del modelo tiene una duración aproximada de una hora cada quinientas épocas, lo que limitó a uno la cantidad de entrenamientos realizados para esta sección. En la tabla 5.4 se calculan las mejoras obtenidas en el coste y la precisión al extender los entrenamientos para las gramáticas 5 y 6. Aunque esta prueba logra mejorar los resultados de la tabla 5.1 para las mismas gramáticas, estos nuevos resultados siguen siendo peores que los obtenidos para el resto de gramáticas con una cantidad de épocas entre diez y veinticinco veces menor.

	1000 épocas (5 tr.)		**25000 épocas (1 tr.)**		**Mejora**	
Gramática	**Coste ($\bar{x} \pm \sigma$)**	**Precisión ($\bar{x} \pm \sigma$)**	**Coste**	**Precisión**	**Coste**	**Precisión**
Tomita5	0.2983 ± 0.0007	0.7876 ± 0.0009	0.2077	0.8795	-30.37%	11.67%
Tomita6	0.5187 ± 0.0005	0.7224 ± 0.0002	0.3832	0.8022	-26.12%	11.05%

Tabla 5.4: Comparación entre los entrenamientos de 1000 y 25000 épocas para Tomita 5 y 6.

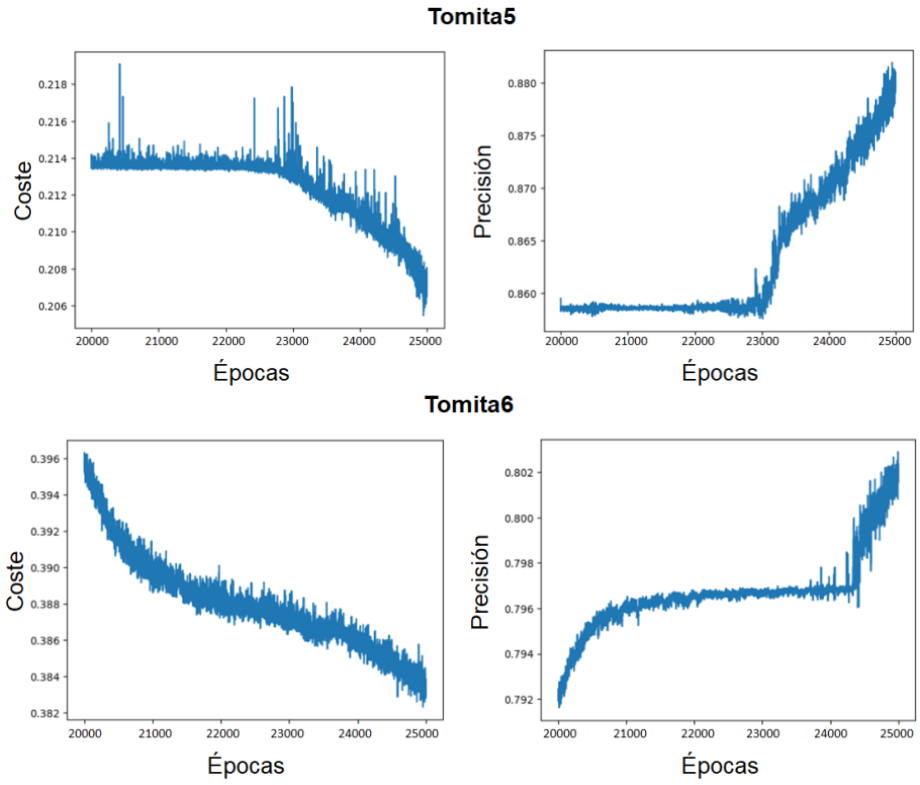


Figura 5.4: Coste y precisión entre las épocas 20000 y 25000 para Tomita 5 y 6.

En la tabla 5.5 se muestran los resultados de la validación para el entrenamiento extendido realizado a las gramáticas de Tomita 5 y 6. Al comparar estos resultados con los de las tablas 5.2 y 5.3 se pueden observar las mejoras conseguidas mediante este experimento. En particular, la gramática 5 mejora notablemente para todos los conjuntos de datos exceptuando el conjunto *largo*, mientras que la gramática 6 únicamente mejora para el conjunto de datos *grande y* obtiene casi los mismos resultados para el resto de conjuntos.

	Grande		Largo		Muchas_a		Muchas_b	
Gramática	**Coste**	**Precisión**	**Coste**	**Precisión**	**Coste**	**Precisión**	**Coste**	**Precisión**
Tomita5	0.1898	0.9066	0.3397	0.7721	0.2825	0.9221	0.1109	0.9873
Tomita6	0.3801	0.8059	0.6067	0.6834	0.601	0.6859	0.6089	0.6836

Tabla 5.5: Resultados de la validación del entrenamiento de 25000 épocas para Tomita 5 y 6.

Para poder interpretar los resultados de este experimento se utiliza la figura 5.5. En esta se pueden apreciar los AFD mínimos extraídos para las gramáticas de Tomita 5 y 6 después de veinte mil épocas de entrenamiento. No se muestran los AFD mínimos extraídos al cabo de veinticinco mil épocas debido a su gran complejidad (33 estados para Tomita 5 y 12 para Tomita 6). Si estos autómatas se comparan con sus análogos obtenidos en la figura 5.3 se puede observar como el modelo aumenta su complejidad con el objetivo de incrementar su precisión, sin que esto garantice una mejora en la generalización (compárese la precisión para veinticinco mil épocas de la tabla 5.4 con las precisiones de la tabla 5.5).

En particular, el autómata extraído para la gramática de Tomita 5 mantiene la restricción observada en la figura 5.3, la cual consiste en aceptar cadenas de longitud par formada por aes o bes. Sin embargo, el conjunto de cadenas aceptadas se ve reducido para incrementar la precisión del modelo al aumentar la complejidad del autómata. Por otro lado, el autómata extraído para la gramática de Tomita 6 únicamente acepta las siguientes siete cadenas: λ, ab, ba, aaa, bbb, $abab$ y $baab$, todas ellas pertenecientes a la gramática de Tomita 6.

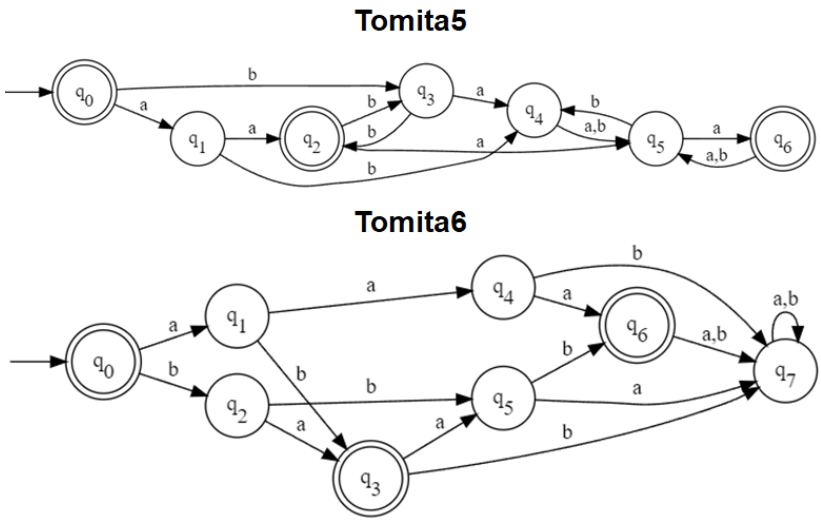


Figura 5.5: AFD mínimos extraídos tras 20000 épocas para Tomita 5 y 6.

5.5. Profundización en las gramáticas de Tomita 5 y 6

En vista de que el experimento realizado en la sección 5.4 tampoco produjo resultados satisfactorios se optó por probar una última alternativa:

1. Inicializar las matrices de pesos W_{xh} y W_{hh} con combinaciones de valores que forzaran la implementación de la solución correcta para las gramáticas de Tomita 5 y 6.
2. Entrenar el modelo con un número reducido de épocas y analizar su comportamiento para determinar si se mantiene en el rango de la solución forzada o si se desvía de ella.

De este modo, sería posible distinguir si el modelo es incapaz de implementar la solución correcta o, por el contrario, únicamente es incapaz de encontrarla por sí mismo.

Inicialmente este procedimiento se aplicó a la gramática de Tomita 1, debido a su nivel de simplicidad, para verificar la validez del mismo. Se comprobó que asignando los pesos de la figura 5.6 el modelo se mantuvo a lo largo de cien épocas con una precisión igual a uno (ver figura 5.7).

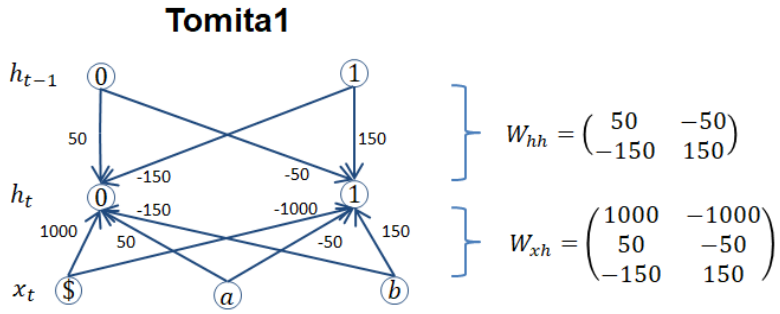


Figura 5.6: Grafo de neuronas con los pesos que resuelven Tomita 1.

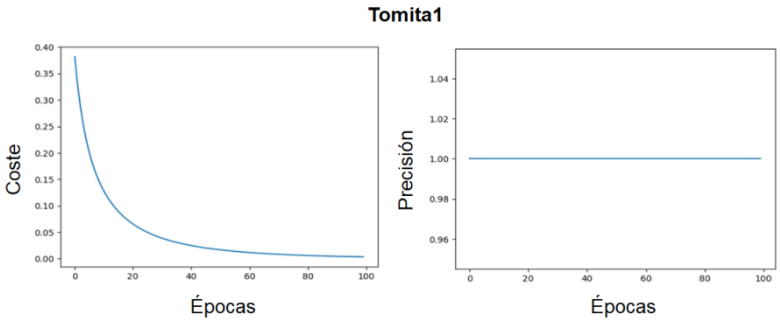


Figura 5.7: Coste y precisión con pesos predefinidos para Tomita 1.

Sin embargo, cuando se intentó aplicar esta estrategia a las gramáticas de Tomita 5 y 6 no fue posible encontrar una combinación correcta de valores para los pesos. Para profundizar en el análisis de por qué esto ocurría se realizó la figura 5.8. En esta figura se pueden apreciar las neuronas de la capa recurrente y las interconexiones que existen entre ellas. El objetivo es representar gráficamente los autómatas correspondientes a cada gramática de Tomita en forma de grafos que tengan las mismas dimensiones que las matrices de pesos asociadas.

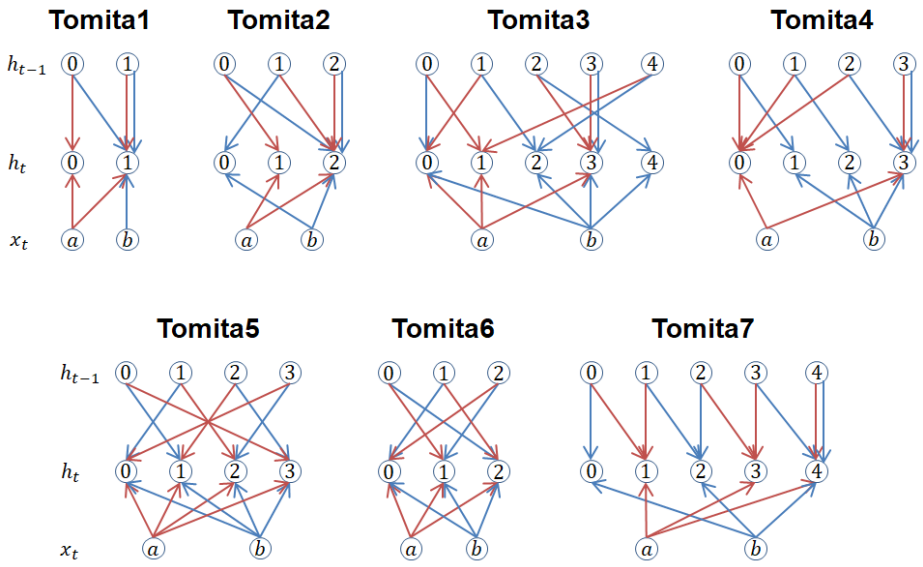


Figura 5.8: Grafos de neuronas para cada gramática de Tomita.

La primera fila de neuronas en cada grafo de la figura 5.8 representa los estados anteriores h_{t-1}, la segunda se corresponde con el estado actual h_t al que se transita desde h_{t-1} y la última fila muestra el símbolo x_t que permite dicha transición. Las transiciones correspondientes a cada símbolo se diferencian mediante el siguiente esquema de colores: rojo para a y azul para b. Es oportuno mencionar que las transiciones mediante el símbolo $ fueron eliminadas y que se omiten las flechas correspondientes a las transiciones que no son posibles. Esto es debido a que la asignación de valores para estas conexiones es trivial: las flechas omitidas deben tener negativos de gran magnitud, para evitar que se produzcan transiciones prohibidas, y la flecha que va del símbolo $ al estado 0 debe tener un valor positivo muy alto, con el fin de garantizar que la lectura del símbolo $ provoque siempre la transición hacia el estado inicial del autómata.

Para asegurar la correcta interpretación de los grafos, a continuación se explican las posibles transiciones a partir del símbolo a para el autómata que define la gramática de Tomita 1:

- Al encontrarse en el estado 0 y recibir como entrada el símbolo a debe mantenerse en el estado 0, esta situación se ve reflejada mediante las dos flechas rojas situadas más a la izquierda del grafo.
- Al hallarse en el estado 1 y recibir como entrada el símbolo a debe mantenerse en el estado 1, esta situación se representa mediante las dos flechas rojas restantes.

La solución a este problema consiste en hallar una combinación de números que cumplan dos condiciones:

1. La suma de los valores asignados a dos flechas del mismo color que apuntan a la misma neurona debe ser positiva.
2. La suma de los valores asignados a dos flechas de distinto color que apuntan a la misma neurona debe ser negativa.

Estas condiciones convierten a las gramáticas de Tomita 5 y 6 en problemas cuya resolución es mucho más compleja que la del resto. Esto es debido a que ambas gramáticas permiten transitar a todos los estados del autómata mediante los símbolos a y b (obsérvese que todas las conexiones posibles entre los símbolos x_t y los estados h_t tienen una flecha que las representa). De forma que si se asigna a una flecha un valor alto para favorecer una transición mediante el símbolo a, sucederá lo mismo para el símbolo b. Esto representa un problema cuando, desde un mismo estado h_{t-1} se debe transitar a estados h_t diferentes en función del símbolo x_t computado (se bifurcan los caminos), situación que ocurre para todos los estados h_{t-1} de las gramáticas 5 y 6.

6

Conclusiones

En este proyecto se implementó una red neuronal recurrente con regularización de estados. El modelo de aprendizaje automático desarrollado a lo largo de este trabajo se basa en una de las dos alternativas propuestas en [6], concretamente en la que plantea un proceso de muestreo para determinar el valor del estado actual de la red recurrente. Esta elección acarreó consigo el uso de técnicas de aprendizaje por refuerzo, ya que la operación de muestreo no es diferenciable y, por lo tanto, el gradiente no puede ser retropropagado de extremo a extremo. Esto derivó en la fase de entrenamiento de la sección 4.2 que combina exitosamente dos formas distintas de aprendizaje: por refuerzo y supervisado.

El objetivo de la regularización de estados implementada en este proyecto es restringir la cantidad de estados internos que la red recurrente puede adquirir, asegurando de esta forma que aprenda un conjunto finito de estados durante su entrenamiento. Lo anterior produce que la red recurrente se comporte internamente como un autómata finito determinista, incrementando de esta manera la interpretabilidad del modelo y eliminando su condición inherente de caja negra. Esto se comprobó en la sección 5.3, donde se pueden apreciar los autómatas correspondientes a las gramáticas inferidas extraídos mediante el método de la sección 4.4.

Para poner a prueba el modelo desarrollado, cuyo diseño e implementación se describen en las secciones 3.2 y 4.1 respectivamente, se utilizaron las gramáticas de Tomita. Los resultados obtenidos para estas pruebas se recopilan en el capítulo 5, los cuales evidencian que el modelo es capaz de inferir correctamente todas las gramáticas de Tomita, salvo la 5 y la 6, para las que aun así se obtiene cierto grado de precisión.

A lo largo de la sección 5.4 se listan los experimentos realizados con el objetivo de mejorar el rendimiento del modelo para las gramáticas de Tomita 5 y 6. La única variación que proporcionó cierta mejora fue aumentar significativamente el número de épocas del entrenamiento. Dicha mejora consiguió aumentar la precisión del modelo a expensas de incrementar su complejidad, sin que esto garantizara una mejora en la generalización. Este comportamiento se debe a que el modelo desarrollado es incapaz de, al menos, hallar los autómatas óptimos para estas gramáticas cuando utiliza el número mínimo de neuronas posible.

Declaración que se basa en la sección 5.5, donde se plantean los inconvenientes que presentan los autómatas que permiten transitar a todos sus estados mediante los símbolos a y b.

En resumen, en este proyecto se desarrolló un modelo de aprendizaje automático especializado en la inferencia gramatical. Esto se llevó a cabo mediante una red neuronal recurrente con regularización de estados con el objetivo de facilitar la extracción del autómata implementado internamente por dicha red. El marco de este proyecto podría ampliarse al analizar en profundidad la problemática que presentan las gramáticas de Tomita 5 y 6, adaptar el modelo para que sea capaz de procesar problemas más complejos no regulares e implementar un fase de entrenamiento que aplique recompensas descontadas, donde se asigna un valor menor a las recompensas futuras en comparación con las inmediatas.

Bibliografía

[1] E. M. Gold, “Language identification in the limit,” *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.

[2] W. J. Murdoch and A. Szlam, “Automatic Rule Extraction from Long Short Term Memory Networks,” 2017, doi: 10.48550/arxiv.1702.02540.

[3] A. Graves, A. Mohamed, and G. Hinton, “Speech Recognition with Deep Recurrent Neural Networks,” 2013, doi: 10.48550/arxiv.1303.5778.

[4] W. Yin, K. Kann, M. Yu, and H. Schütze, “Comparative Study of CNN and RNN for Natural Language Processing,” 2017, doi: 10.48550/arxiv.1702.01923.

[5] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017–1024, 2011.

[6] C. Wang and M. Niepert, “State-Regularized Recurrent Neural Networks,” 2019, doi: 10.48550/arxiv.1901.08817.

[7] M. Tomita, “Dynamic construction of finite automata from examples using hill-climbing,” in *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, (Ann Arbor, Michigan), pp. 105–108, 1982.

[8] P. Linz, *An introduction to formal languages and automata*, 6th ed. Burlington, Massachusetts: Jones and Bartlett Learning, 2017.

[9] N. Chomsky, “Three models for the description of language,” *IRE Transactions on Information Theory*, vol. 2, pp. 113–124, 1956.

[10] S. C. Kleene, “Representation of events in nerve nets and finite automata,” in *Automata Studies* (C. Shannon and J. McCarthy, eds.), pp. 3–41, Princeton, NJ: Princeton University Press, 1956.

[11] D. Poole, A. Mackworth, and R. Goebel, *Computational Intelligence: A Logical Approach,* pp. 1–22, Oxford University Press, 1998.

[12] J. M. Helm *et al.*, “Machine Learning and Artificial Intelligence: Definitions, Applications, and Future Directions,” *Current reviews in musculoskeletal medicine*, vol. 13, no. 1, pp. 69–76, 2020.

[13] D. Esposito and F. Esposito, *Introducing machine learning*, 1st edition. 2020.

[14] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson Education, 2021.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *The Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[16] R. Rojas, *Neural Networks: A Systematic Introduction*, Springer-Verlag Berlin Heidelberg, 1996.

[17] Y. Bai, “RELU-Function and Derived Function Review,” *SHS web of conferences*, vol. 144, p. 2006, 2022.

[18] S. S. Haykin, *Neural networks : a comprehensive foundation*, 2nd ed. New Jersey: Prentice Hall International, 1999.
[19] C. Lemaréchal, “Cauchy and the Gradient Method,” *Documenta Mathematica*, vol. ISMP, pp. 251–254, 2012.
[20] D. E. Rumelhart and J. L. McClelland, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pp. 318–362, MIT Press, 1987.
[21] J. L. Elman, “Finding structure in time,” *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.
[22] M. I. Jordan, "Serial Order: A Parallel Distributed Processing Approach," in *Advances in Psychology*, vol. 121, pp. 471–495, 1997.
[23] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[24] K. Cho et al., “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,” 2014, doi: 10.48550/arxiv.1406.1078.
[25] J. Oncina and P. Garcia, "Inferring regular languages in polynomial update time," in *Pattern Recognition and Image Analysis*, vol. 1, pp. 49–61, 1992.
[26] D. Angluin, “Learning regular sets from queries and counterexamples,” *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.
[27] C. de la Higuera, *Grammatical Inference*. Cambridge: Cambridge University Press, 2010.
[28] H. Jacobsson, “Rule extraction from recurrent neural networks: A taxonomy and review,” *Neural Computation*, vol. 17, no. 6, pp. 1223–1263, 2005.
[29] H. Siegelmann and E. Sontag, “On the computational power of neural nets,” *J. Comput. Syst. Sci.*, vol. 50, pp. 132–150, 1995.
[30] A. M. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society*, vol. s2–42, no. 1, pp. 230–265, 1937.
[31] C. L. Giles, D. Chen, C. B. Miller, H. H. Chen, G. Z. Sun, and Y. C. Lee, “Second-order recurrent neural networks for grammatical inference,” in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. 2, pp. 273–281, 1991.
[32] Z. Zeng, R. M. Goodman, and P. Smyth, “Learning Finite State Machines With Self-Clustering Recurrent Networks,” *Neural computation*, vol. 5, no. 6, pp. 976–990, 1993.
[33] J. N. Foerster, J. Gilmer, J. Chorowski, J. Sohl-Dickstein, and D. Sussillo, “Input Switched Affine Networks: An RNN Architecture Designed for Interpretability," 2017, doi: 10.48550/arxiv.1611.09434.
[34] G. Weiss, Y. Goldberg, and E. Yahav, “Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples,” 2017, doi: 10.48550/arxiv.1711.09576.
[35] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent Neural Network Regularization,” 2014, doi: 10.48550/arxiv.1409.2329.
[36] S. Merity, N. S. Keskar, and R. Socher, “Regularizing and Optimizing LSTM Language Models,” 2017, doi: 10.48550/arxiv.1708.02182.
[37] A. B. Dieng, R. Ranganath, J. Altosaar, and D. M. Blei, “Noisin: Unbiased Regularization for Recurrent Neural Networks,” 2018, doi: 10.48550/arxiv.1805.01500.
[38] C. Oliva and L. Lago-Fernández, “Interpretability of recurrent neural networks trained on regular languages,” in *15th International Work-Conference on Artificial Neural Networks, IWANN*, pp. 14–25, 2019.
[39] “Tensorflow.” https://www.tensorflow.org/.

[40] “Keras.” https://keras.io/.
[41] "Keras Functional API." https://keras.io/guides/functional_api/
[42] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International Conference on Artificial Intelligence and Statistics*, 2010.
[43] "Keras Layers API." https://keras.io/api/layers/.
[44] T. Dozat, "Incorporating Nesterov Momentum into Adam," In *Proceedings of the 4th International Conference on Learning Representations*, pp 1–4. 2016.
[45] "Graphviz." https://graphviz.org/.

Apéndices

A

Implementación del modelo

Este apéndice contiene el código más relevante relacionado con el modelo de aprendizaje automático utilizado a lo largo de este proyecto. Es importante recordar que tanto el código del modelo como el de la celda recurrente fueron descritos previamente en el capítulo 4. En este apéndice se muestra el código relativo a las fases de entrenamiento y validación, el método utilizado para copiar los pesos de un modelo origen a otro destino, el algoritmo implementado para la extracción del AFD a partir de los pesos de un modelo ya entrenado y la traducción de este AFD a lenguaje DOT para poder representarlo gráficamente.

A.1. Entrenamiento

```
1  def train_model(model, num_epochs, x, y, batch_size):
2      optimizer_dense = tf.keras.optimizers.Nadam()
3      optimizer_rnn = tf.keras.optimizers.Nadam()
4      model_loss = np.zeros(num_epochs)
5      model_accuracy = np.zeros(num_epochs)
6      for epoch in range(num_epochs):
7          epoch_loss = []
8          epoch_accuracy = []
9          model.reset_states()
10         for i in range(0, x.shape[0], batch_size):
11             batch_x = x[i:i+batch_size, :, :]
12             y_true = y[i:i+batch_size, :, :]
13             with tf.GradientTape(persistent=True) as tape:
14                 logits, h, y_pred = model(batch_x)
15                 loss_dense = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
16                 loss_dense_mean = tf.math.reduce_mean(loss_dense)
17                 r = -loss_dense.numpy().mean(axis=1)
18                 r -= np.mean(r)
19                 r /= np.std(r)
20                 r = r[:, None]
21                 loss_rnn = tf.keras.losses.binary_crossentropy(h, logits, from_logits=True)
22                 loss_rnn_r = loss_rnn * r
23                 loss_rnn_r_mean = tf.math.reduce_mean(loss_rnn_r)
24                 variables_rnn = model.trainable_variables[:3]
25                 variables_dense = model.trainable_variables[3:]
26                 grads_dense = tape.gradient(loss_dense_mean, variables_dense)
27                 grads_rnn = tape.gradient(loss_rnn_mean, variables_rnn)
28                 optimizer_dense.apply_gradients(zip(grads_dense, variables_dense))
29                 optimizer_rnn.apply_gradients(zip(grads_rnn, variables_rnn))
30                 predictions = np.argmax(y_pred.numpy(), axis=2)
31                 classes = np.argmax(y_true, axis=2)
32                 epoch_accuracy.append(np.mean(predictions == classes))
33                 epoch_loss.append(loss_dense_mean.numpy())
34         model_loss[epoch] = np.array(epoch_loss).mean()
35         model_accuracy[epoch] = np.array(epoch_accuracy).mean()
36     return model_loss, model_accuracy
```

Código A.1: Implementación del entrenamiento en *Keras*.

A.2. Validación

```
1  def test_model(model, x, y, batch_size):

2    model.reset_states()
3    mean_loss = []
4    mean_accuracy = []

5    for i in range(0, x.shape[0], batch_size):
6      batch_x = x[i:i+batch_size, :, :]
7      y_true = y[i:i+batch_size, :, :]

8      _, _, y_pred = model(batch_x)
9      loss_dense = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
10     loss_dense_mean = tf.math.reduce_mean(loss_dense)

11     predictions = np.argmax(y_pred.numpy(), axis=2)
12     classes = np.argmax(y_true, axis=2)
13     mean_accuracy.append(np.mean(predictions == classes))
14     mean_loss.append(loss_dense_mean)

15   return np.array(mean_loss).mean(), np.array(mean_accuracy).mean()
```

Código A.2: Implementación de la validación en *Keras*.

A.3. Copiar pesos

```
1  def copy_weights(source_model, dest_model):

2    for source_layer, dest_layer in zip(source_model.layers, dest_model.layers):
3    dest_layer.set_weights(source_layer.get_weights())
```

Código A.3: Copiar pesos en *Keras*.

A.4. Extracción AFD

```
def extract_AFD(nunits, model):

    x = np.array([[1,0,0],[0,1,0],[0,0,1]])[:,None,:]
    zero_state = tuple(np.zeros(nunits).tolist())
    model.layers[1].reset_states(states=np.array([zero_state, zero_state, zero_state]))
    aux_state = tuple(model(x)[0][0][0].numpy().tolist())
    model.layers[1].reset_states(states=np.array([aux_state, aux_state, aux_state]))
    initial_state = tuple(model(x)[0][0][0].numpy().tolist())
    queue = []
    discovered = [initial_state]
    connections = {tuple(initial_state): {}}
    queue.append(initial_state)
    accept = {}

    while queue:
        state = queue.pop(0)
        model.layers[1].reset_states(states=np.array([state, state, state]))
        _, new_states, probabilities = model(x)
        i = 0
        for aux in new_states:
            new_state = tuple(aux[0].numpy().tolist())
            if state not in connections:
                connections[state] = {}
            connections[state][i] = new_state
            if new_state not in accept:
                accept[new_state] = True if probabilities[i][0][0] < 0.5 else False
            if new_state not in discovered:
                discovered.append(new_state)
                queue.append(new_state)
            i += 1

return connections, accept
```

Código A.4: Método para extraer AFD.

A.5. Traducción del AFD extraído a lenguaje DOT

```
1  def write_dot(states, final, transitions, file):
2      with open(file, "w") as f:
3          f.write("digraph {\n")
4          f.write("  rankdir=LR;\n\n")
5          f.write("  start [style=invis];\n")
6          for s, is_final in zip(states, final):
7              if is_final:
8                  f.write(f'  q{s}[label=<q<sub>{s}</sub>>, shape="doublecircle", ' +
9                          'color="black"];\n')
10             else:
11                 f.write(f'  q{s}[label=<q<sub>{s}</sub>>, shape="circle", ' +
12                         'color="black"];\n')
13         f.write("\n")
14         f.write(f'  start -> q{states[0]}[color="black"];\n')
15         for s, symbols in transitions.items():
16             label = ",".join(symbols)
17             f.write(f'  q{s[0]} -> q{s[1]}[label="{label}", color="black", ' +
18                     'fontcolor="black"];\n')
19         f.write("}\n")
```

Código A.5: Traducción de AFD a lenguaje DOT.