UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Desarrollo de Soporte de Redes Recurrentes en Plataforma de Entrenamiento

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

*Autor:* Díaz Bou, Pere

*Tutores:* Flich Cardo, José
López Rodríguez, Pedro Juan

Curso 2020-2021

# Resum

La recerca en intel·ligència artificial està creixent constantment i gràcies a això, ens ha permet resoldre problemes que abans pareixien impossibles.

Les Xarxes Neuronals son uns dels algorismes mes usats per a resoldre aquests problemes. Hi ha mes de un tipus de Xarxa Neuronal, cada una usada per a diferents aplicacions i, amb diferents complexitats. A més a més, implementarem les Xarxes Recurrents en *HELENNA*, una plataforma d'entrenament i inferència de Xarxes Neuronals. Per finalitzar, farem una avaluació funcional de la Xarxa Recurrent per validar la correcta implementació.

La implementació de la Xarxa Neuronal recurrent será capaç de resoldre problemes senzills com la predicció de dígits escrits a mà amb un alt nivell de precisió.

**Paraules clau:** Xarxes Neuronals Recurrents, Xarxes Neuronals, Intel·ligència artificial, plataforma d'entrenament

# Resumen

La investigación en inteligencia artificial está creciendo a un ritmo constante y gracias a ello, se están resolviendo problemas que hasta hace poco parecían imposibles.

Las Redes Neuronales son unos de los algoritmos mas usados para resolver estos problemas, y existen mas de un tipo de Red Neuronal. Cada una con cierta utilidad y complejidad. En este trabajo realizamos un análisis de las Redes Recurrentes. En concreto, analizamos su estructura y su funcionamiento. Posteriormente, implementaremos las Redes Recurrentes en *HELENNA*, una plataforma de entrenamiento e inferencia de Redes Neuronales. Para acabar, realizamos una evaluación funcional de la Red Recurrente para validar la correcta implementación.

La implementación de la Red Neuronal Recurrentes será capaz de resolver problemas básicos como la predicción de dígitos escritos a mano con un porcentaje de precisión alto.

**Palabras clave:** Redes neuronales, Redes neuronales recurrentes, Inteligencia artificial, plataforma de entrenamiento

# Abstract

Research in artificial intelligence is growing at incredible rates which allowed us to solve problems that seemed impossible before.

One of the best known algorithms to solve artificial intelligence problems is neural networks. There are different types of neural networks, with different complexities and applications. Moreover, we will implement Recurrent Neural Networks in *HELENNA*, a training and inference plataform for Neural Networks. At last, we will evaluate the functional aspect of the Recurrent Neural Network to validate the correctness of it.

Our recurrent neural network implementation will be able to predict with high accuracy handwritten digits.

**Key words:** Recurrent Neural Networks, Neural Networks, Artificial Intelligence, Training Plataform

# Índice general

# Índice de figuras



# Índice de tablas

# CAPÍTULO 1
# Introducción

Desde hace tiempo hemos buscado entender como funciona el cerebro y como es posible que podamos ser como somos. Somos unos animales tan inteligentes que queremos simular nuestra inteligencia y no hace mucho empezamos con el estudio de la inteligencia artificial.

La inteligencia artificial ha crecido mucho en poco tiempo y con ella, han aparecido muchas bifurcaciones donde estudiamos diferentes algoritmos, técnicas, teorías... Solo entre 2019 y 2020 el número de artículos sobre la inteligencia artificial creció un 34,5 % en comparación con 2018-2019, que creció un 19,6 %.

La red neuronal es uno de los algoritmos mas usados en este campo, siendo la representación mas aproximada del cerebro que hemos sido capaces de simular. Hay muchos tipos de redes neuronales cada uno enfocada a un tipo de problema diferente y con un nivel de complejidad diferente. En este trabajo nos vamos a centrar en uno de estos tipos, las redes neuronales recurrentes.

Las redes neuronales recurrentes aparecieron con la necesidad de procesar datos secuenciales, que una red neuronal normal, no podía tratar con eficacia. Veremos con detalle todas las operaciones que realiza y, la implementaremos en la plataforma de entrenamiento de redes neuronales *HELENNA*. Al ser este tipo de red neuronal tan utilizado para la investigación, es indispensable que estén disponibles en la plataforma para futuras investigaciones y que sea lo mas eficiente posible para aumentar la productividad del usuario.

## 1.1 Motivación

Las operaciones relacionadas con la inteligencia artificial requieren muchos recursos para poder realizar computaciones con cantidades increíbles de datos. Hay empresas que tratan con *petabytes* de información diariamente y donde una simple optimización puede ahorrar millones de euros.

La computación en la nube ha supuesto un *boom* en la última década. En la actualidad, la infraestructura es un servicio que ofrecen grandes empresas, donde podemos alquilar parte de *clusters* para ejecutar nuestras aplicaciones. Estos *clusters* están diseñados para ofrecer una potencia de calculo increíble con las nuevas arquitecturas para la aceleración de *hardware* y funciones mas específicas como las utilizadas en inteligencia artificial.

Hasta no hace mucho, la mayoría de *clusters* se alimentaban de la *CPU* para todo. Ahora, tenemos *clusters* de *GPUs* que mejoran de manera exponencial la eficiencia de operaciones que requieren mucho computo, en especial las relacionadas con la inteligen-

cia artificial. Por otra parte, también tenemos *hardware* mas específico como las *FPGAs* y *ASICs* que son mucho mas eficientes en algunos aspectos.

*Google* dispone de una de las infraestructuras mas grandes, compuesta de *clusters* de *TPUs*. Las *TPUs* son específicas para acelerar operaciones con *tensores* de dos y tres dimensiones que usamos constantemente en redes neuronales.

**Figura 1.1:** TPU Clusters de Google

Todas las plataformas de entrenamiento de redes neuronales que se están usando para investigar son muy completas. Sin embargo, no ofrecen todo el soporte a nuevo hardware. En *HELENNA*, podemos modificar y dar soporte a nuevo hardware como las «FPGAs» con mas flexibilidad, y por lo tanto, es necesario invertir tiempo en desarrollar y mejorar esta plataforma con la finalidad de mejorar la productividad de desarrollo y de investigación.

*HELENNA* busca poder exprimir lo máximo posible las arquitecturas con *CPUs*, *GPUs* y *FPGAs* en aplicaciones de redes neuronales, que tanto uso hacen de ellas para cálculos que requieren de mucha computación.

Sin embargo, *HELENNA* tiene carencia de algunos tipos de redes neuronales, siendo una de ellas las redes neuronales recurrentes que están siendo utilizadas en muchos tipos de aplicaciones. El procesamiento de lenguaje natural es un subcampo de la inteligencia artificial y de la lingüística que esta ganando mucha popularidad. Este campo usa con mucha frecuencia las redes neuronales recurrentes, ya que se especializa en el tratamiento de datos secuenciales.

Otro caso muy interesante es la conducción autónoma. Sabemos que un humano hace uso del entorno y del paso del tiempo para evaluar la situación en la conducción. El paso del tiempo aquí es muy importante. Un solo fotograma solo nos puede ayudar para identificar objectos, pero no cómo se comportan en el tiempo.

Las redes neuronales recurrentes nos ayudan en la evaluación de datos con el paso del tiempo. Por ejemplo, solo a partir de un vídeo podemos entrenar una red neuronal recurrente para que aprenda a deducir a qué velocidad va el coche. Los humanos hacemos algo parecido sin mirar el contador de velocidad, simplemente con mirar la diferencia de la posición de un objeto con respecto el tiempo.

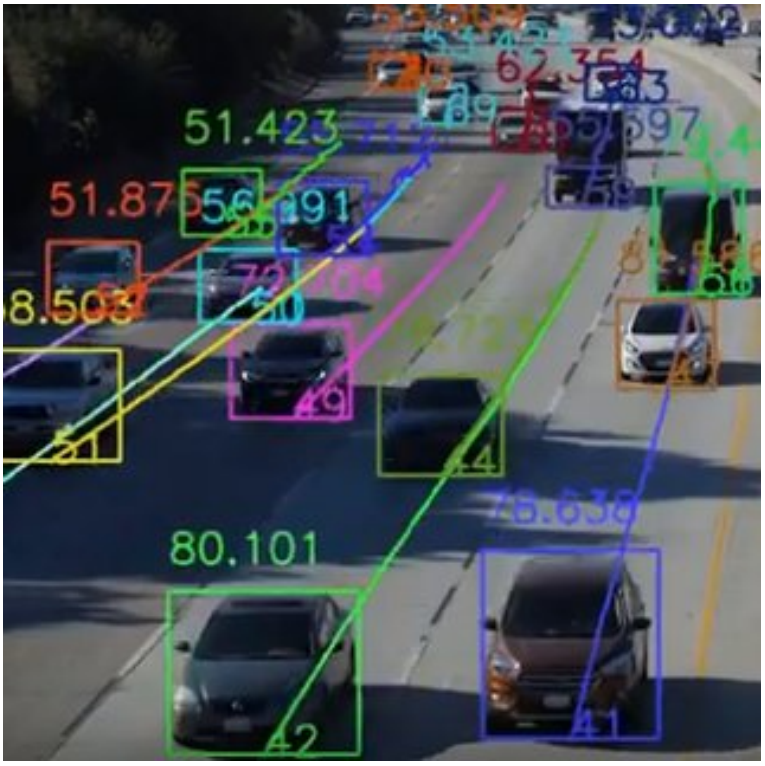



**Figura 1.2:** Predicción de velocidad de diferentes vehículos

## 1.2 Objetivos

En este trabajo extenderemos la plataforma de entrenamiento *HELENNA* con Redes Neuronales Recurrentes para que después pueda ser utilizado en futuras trabajos de investigación. También es de importancia que se acople a la plataforma de tal forma que utilice diferente *hardware* cuando sea especificado en la aplicación.

La implementación la complementamos con una validación de la misma para comprobar que ésta funciona como es esperado con algún caso sencillo y comparando con otros tipos de modelos que no hagan uso de la misma pero que resuelvan el mismo problema.

En este trabajo especificamos los siguientes objetivos:

1. Analizar cómo funcionan las redes neuronales en todos los pasos del aprendizaje para después saber cómo acoplar las Redes Neuronales Recurrentes.

2. Estudiar el comportamiento a bajo nivel de una Red Neuronal Recurrente, desde operaciones matemáticas a abstracciones que puedan ayudar a definirlas.

3. Introducir *HELENNA* y analizar la plataforma con el fin de utilizar eficientemente las funcionalidades que ofrece.

4. Desarrollar la versión inicial de una *RNN* funcional para resolver problemas de aprendizaje.

5. Validar el desarrollo con diferentes pruebas que demuestren que funciona. También queremos que se compare con aplicaciones ya existentes para tener un punto de referencia.

Además, externamente hemos colaborado con el grupo de Arquitecturas Paralelas («GAP») con diferentes profesores y con otros alumnos. Se han hecho reuniones semanalmente durante la duración del trabajo con la finalidad de mejorar la colaboración entre el grupo, compartir herramientas e ideas que puedan ser utilizadas en los diferentes trabajos que se han realizado.

## 1.3 Objetivos de Desarrollo Sostenible

La ONU en 2015, estableció los planes para el desarrollo sostenible con metas establecidas para cada objetivo. Hay 17 objetivos en total de los cuales, nos vamos a centrar en tres:

- Objetivo 9: **Agua, industria, innovación e infraestructura**

  La inteligencia artificial y las plataformas para el desarrollo de estas, nos permite investigar con mas productividad y por ende, resolver problemas que puedan mejorar la vida general de la población. Además, la investigación y puesta en marcha de nuevas empresas gracias a la investigación, puede influir de manera positiva en la economía.

- Objetivo 11: **Ciudades y comunidades sostenibles**

  Desde que se empezó a crecer la investigación en inteligencia artificial se ha visto mucha creatividad a la hora de resolver problemas de eficiencia en ciudades. Crear plataformas para resolver estos problemas es de vital importancia para la mejora de procesos urbanos.

- Objetivo 13: **Acción por el clima**

  La plataforma de entrenamiento *HELENNA* tiene como uno de los objetivos principales reducir el gasto energético lo máximo posible. Menor gasto de energía en un mundo en el que la cantidad energía gastada es enorme, es esencial para mejorar el medio ambiente.

## 1.4 Estructura de la Memoria

La memoria la estructuraremos de forma en que empecemos con lo más básico de las redes neuronales y terminemos con la validación de las redes neuronales recurrentes.

Primero veremos el estado del arte y contexto relacionado con nuestro trabajo acompañado de todas las herramientas que vamos a usar. Continuaremos con una parte relacionada con las redes neuronales, en la cual contaremos un poco su historia, después veremos la versión mas simple que es el perceptrón y, para acabar, llegaremos a ver cómo se implementa una red neuronal completa con sus funciones de activación, funciones de coste, y cómo aprende la red neuronal con la propagación hacía atrás.

Cuando ya sepamos cómo funcionan al nivel mas elemental las redes neuronales, comenzaremos a ver el tipo de redes neuronales recurrentes que son un poco más complejas y por lo tanto, veremos con mayor detalle todas las fases por las que pasa.

Después pasaremos a ver como funciona la plataforma de entrenamiento *HELENNA* y visto esto, podremos empezar a implementar por fin las redes neuronales recurrentes.

Para terminar, validaremos que la implementación ha sido exitosa, comparándola con diferentes tipos de modelos.

## 1.5 Relación con asignaturas de la carrera

Este proyecto esta relacionado con diferentes asignaturas que nos ha proporcionado una base necesaria. Entre ellas tenemos:

- Arquitectura e ingeniería de computadores: Este trabajo requiere ser capaz de adaptarse a diferentes arquitecturas de *hardware* para que se implemente el código de la forma mas eficiente posible.
- Mantenimiento y evolución de software: En este trabajo se ha hecho uso *git* para la gestión de versión del código.
- Diseño de software: en todo momento se busca seguir buenas practicas de código limpio y *refactorización* de partes que se pueden mejorar.
- Estructuras de datos y algoritmos: El uso de listas y algoritmos que hacen uso de ellas eficientemente, es importante para realizar el trabajo.
- Estructuras de Computadores: El conocimiento básico del funcionamiento interno de un computador es esencial para este trabajo.

# CAPÍTULO 2
# Estado del Arte y Redes Neuronales

## 2.1 Contexto

Actualmente existen diferentes plataformas para el entrenamiento de redes neuronales. Las dos mas conocidas siendo *Tensorflow* [35] y la otra, *PyTorch* [31]. Ambas son de código abierto y se usan para muchos tipos de aplicaciones ya sean para investigar o para aplicaciones comerciales.

Además, estas plataformas están complementadas por muchas librerías que extienden su utilidad. Por ejemplo, hay un proyecto entero que es una extensión de *PyTorch*, llamada *vision* [33] enfocada al campo de visión por computador. Incluso hay versiones de las mismas para sitios específicos: *Tensorflow.js* [36] nos permite ejecutar modelos en la web y *PyTorch Lightning* [32] nos permite crear redes neuronales mas intuitivamente con menos código.

Estas plataformas no ofrecen soporte directo para nuevo *hardware*, que es de importancia para los investigadores. El uso de *FPGAs* [42] para redes neuronales son relativamente recientes y no hay mucho soporte para ellas. *HELENNA* [45], la plataforma de entrenamiento que estamos desarrollando, se está encargando de solucionar este problema y así ayudar en futuros proyectos de investigación.

En *Tensorflow* y *PyTorch* hay ya soporte para la mayoría de redes neuronales existentes. Una de ellas, la red neuronal recurrente, que es mayoritariamente usada para problemas de procesamiento de lenguaje natural (*NLP*) como la codificación y decodificación de mensajes, predicción de noticias falsas e incluso para predicción de acciones de empresas bursátiles para *brokers*.

*HELENNA* dispone de una amplia gama de funciones esenciales para crear la mayoría de redes neuronales. Las redes neuronales recurrentes todavía no están implementadas en *HELENNA* y si fueran implementadas con las funciones existentes, la investigación con nuevo *hardware* en *NLP* mejoraría.

## 2.2 Historia de las redes neuronales

La evolución desde el primer organismo hasta la vida animal que ahora conocemos se atribuye a las diferentes mutaciones que lo han hecho posible. El chimpancé-humano, nuestro último ancestro común con los chimpancés, marcaría el comienzo de una sucesión de mutaciones que nos haría muy diferentes a los otros animales. El género *Homo*

desarrollaría la capacidad de andar a dos pies y el uso de herramientas, además del crecimiento exponencial de nuestro cerebro en especial el córtex frontal.

Millones de años pasarían y el *Homo sapiens* se coronaría como la última especie del género *Homo* gracias a su inteligencia superior que la distinguiría de otros animales.

El cerebro actualmente sigue siendo un misterio todavía sometido a innumerables estudios con los cuales todavía no podemos llegar a ser capaces de como funciona completamente.

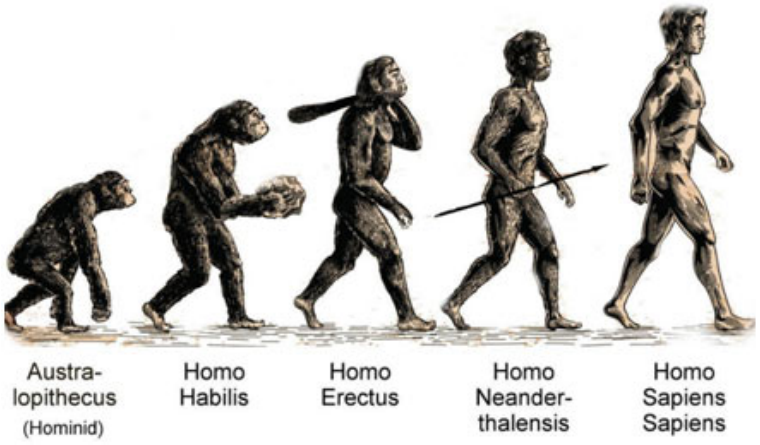



**Figura 2.1:** Evolución del ser humano

Aprender cómo funciona nuestra mente es de interés común entre diferentes disciplinas, una de ellas siendo la informática donde se lleva intentando simular el comportamiento desde hace unos 70 años.

En 1943 Warren S. McCulloch y Walter Pitts escribirían un articulo [11] en el cual intentarían razonar matemáticamente y con circuitos electrónicos el comportamiento cerebral. Más tarde en 1949, Donald Hebb [8], un psicólogo especializado en neuropsicología, publicaría «La organización del comportamiento» [23] donde expondría que las conexiones neuronales se fortalecían cada vez que había un impulso entre las sinapsis.

Hasta aquí, todo fue teórico pero es entonces cuando Nathaniel Rochester III [10], en 1955, publicaría una propuesta de investigación en inteligencia artificial donde intentarían por primera vez en la historia simular una red neuronal en las calculadoras IBM 701 [12] y 704 [13].

Pasados tres años (1958), un neuro-biólogo Frank Rosenblatt empezaría a investigar el perceptrón, el cual era capaz de clasificar binariamente una entrada. El perceptrón estaba limitado y se necesitaría más de una capa para poder hacer predicciones más complejas y por ello hubo un hiatus en el que no se avanzó mucho en las redes neuronales hasta que en 1986 se publicó un artículo [7] que presentaría un algoritmo para entrenar las redes multicapa, el cual reviviría el estudio. Este algoritmo se llama «backpropagation» y se basa en la propagación del error hacía atrás, como veremos posteriormente.

Actualmente, las redes neuronales son la combinación de diferentes capas de diferentes tipos, pero la podemos representar de forma sencilla en la Figura 2.2, donde podemos apreciar los diferentes niveles de capas siendo la primera la de entrada, la última la de salida y por último las centrales y ocultas encargadas de hacer la mayor parte de los cálculos. Las redes neuronales son increíblemente flexibles, pudiendo cambiar muchos parámetros, añadir capas e incluso más dimensiones.

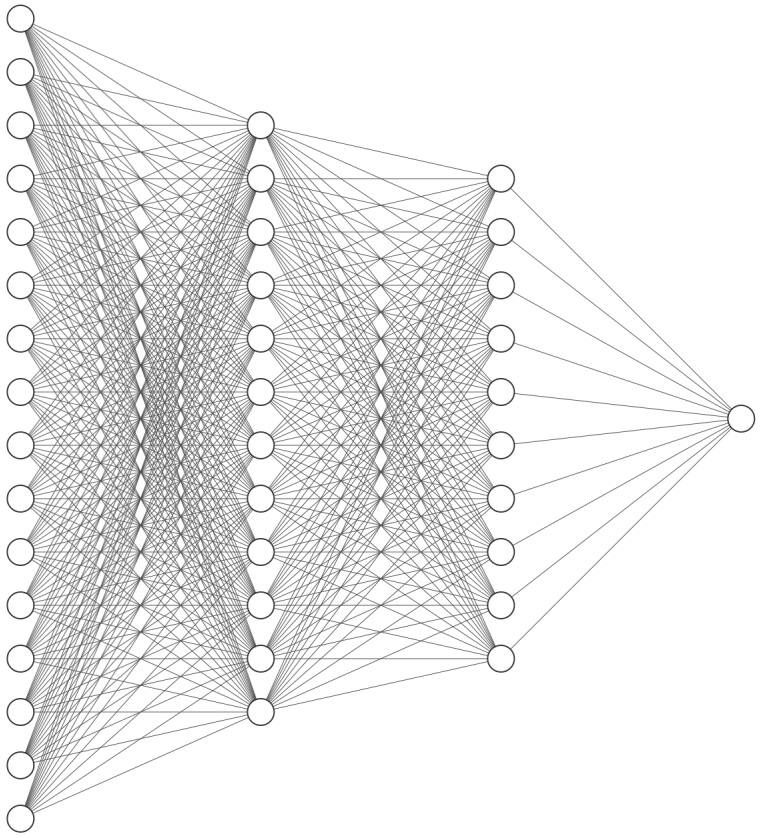


**Figura 2.2:** Red neuronal de 4 capas

Las redes neuronales es una rama de la inteligencia artificial donde existen otras de equivalente interés y que junto ellas se han llegado a resolver problemas de increíble complejidad que anteriormente habría sido imposible de abordar.

## 2.3 Perceptrón

El perceptrón (también conocido como neurona o «unit») desarrollado por Frank Rosenblatt, pasaría a ser la red neuronal mas simple que conocemos y entender su funcionamiento es esencial para entender cómo sistemas mas complejos funcionan, ya que estos siguen parte de sus principios y como veremos mas tarde en las Redes Recurrentes, los cálculos se asemejan mucho pero se añade un poco de complejidad.

En redes neuronales multicapa, cada neurona se le conoce como una unidad (perceptrón) y todas realizan los mismos cálculos con diferentes valores. Estos perceptrones están formados por un conjunto de valores de entrada $x_i$, con su respectivo peso $w_i$ el cual se usa para saber la importancia de esta entrada y un margen («bias») $b_i$ para subir o bajar el valor y poder *redirigir* la salida.

En las redes neuronales multicapa, cada unidad de la capa $l$ propagará la salida a todas las unidades de la siguiente capa $l + 1$ y por lo tanto, si una unidad recibe mas de una entrada, su cálculo se traduce al sumatorio por cada una. Suponiendo que $n =$ número de neuronas de la capa anterior $l - 1$:

$$\sum_{i=1}^{n}(x_i * w_i) + b_i$$

Esta función la utilizamos en mayor parte, para la predicción de una clase. Las clases son etiquetas que definen los tipos de salida que podemos esperar. Para aprender a partir de la predicción calculada y la clase que debería haberse predicho, usamos un algoritmo de entrenamiento que a, partir del resultado obtenido aplica una corrección a los pesos y «bias» para que así, nuestra red neuronal aprenda.

## 2.4 Funciones de activación

Las funciones de activación son las funciones que se aplican al resultado calculado por la neurona. En el caso mas simple, tendríamos una función lineal $f(x) = x$. En contraste, tenemos las funciones no lineales, las cuales son las mas utilizadas porque nos permiten realizar cálculos mas complejos, como las siguientes que veremos a continuación:

### 2.4.1. Sigmoidea

La función de activación sigmoidea es utilizada para predecir una probabilidad la cual la hace útil para la clasificación multiclase. Como podemos observar en la Figura 2.3, el resultado está contenido entre 0 y 1 y su función está definida como:

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

Por ejemplo, si hay un dígito escrito a mano y queremos predecir cual es, utilizamos la función sigmoidea. Con todos los dígitos posibles (multiclase) calculamos su probabilidad de que sea ese y escogemos la probabilidad más alta.

### 2.4.2. Tanh

Esta función se diferencia de la sigmoidea en que los valores se encuentran entre -1 y 1, convirtiéndola en simétrica, como se puede apreciar en la Figura 2.3. Uno de sus usos sería la clasificación en dos clases pero el uso mas importante es transformar los valores para que la media de una capa intermedia se acerque a 0 y por lo tanto haciendo que la próxima capa aprenda mejor. La tangente hiperbólica (tanh) viene definida como:

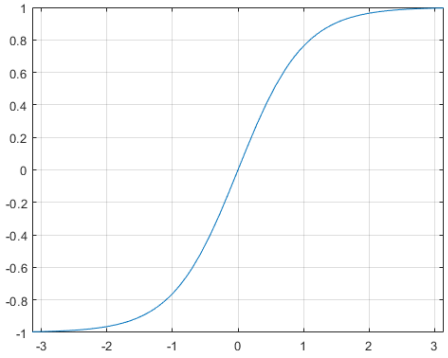
$$\tanh = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

### 2.4.3. ReLU

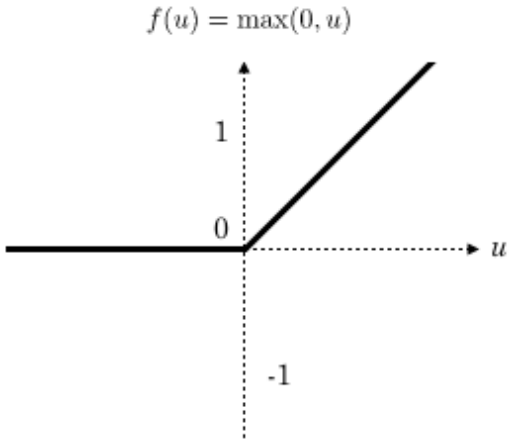
ReLU o «Rectificadora Lineal Unitaria» es una función que no requiere mucha computación gracias a su formula simple $relu(x) = max(0, x)$. Gracias a sus rápidos cálculos, es una de las funciones de activación mas usadas actualmente. Como se puede observar en la Figura 2.3, los valores son positivos teniendo una clara limitación para hacer predicciones más complejas.

### 2.4.4. Otras

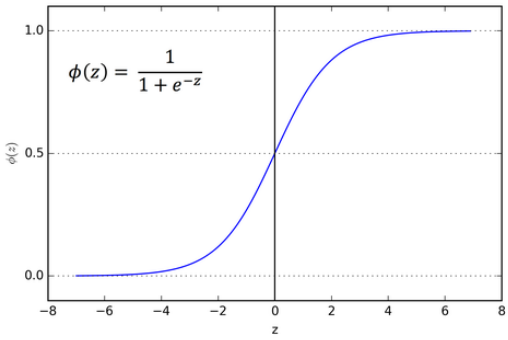
Las funciones de activación mencionadas son las mas comunes pero, como era de esperar, hay muchas mas. Hay desde variaciones de las anteriores como la PReLU, ELU,




(a) Tanh




(b) ReLU




(c) Sigmoidea

**Figura 2.3:** Funciones de activación

y la Soft step hasta nuevas como la Mish, la Gaussiana Softplus ... Todas ellas dan buenos resultados dependiendo del contexto y se requiere práctica para saber cual elegir.

## 2.5 Funciones de coste

Las funciones de coste, error o pérdida se utilizan para calcular la relación entre la salida calculada en la red neuronal y el valor que debería haber salido. Por lo tanto, si una salida es correcta tendrá un coste bajo y si falla este subirá pudiendo así estimar como nuestra red se está comportando y compensar lo bien que se están haciendo las predicciones.

A continuación, veremos dos funciones de coste muy habituales.

### 2.5.1. Mean Squared Error

Error cuadrático medio («mean squared error» en inglés) mide la precisión de la predicción a partir de la media del error al cuadrado. La formula sería:

$$MSE = 1/n \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

donde $Y_i$ es la salida, $\hat{Y}_i$ es la predicción esperada y $n$ el número predicciones.

Esta función es utilizada para ver si nos acercamos al resultado esperado. Por ejemplo, si estamos calculando la velocidad de un coche con una cámara, a partir de la velocidad real podemos ver si la velocidad calculada se acerca a ella y estimar si la red neuronal está acercándose a la realidad con los cálculos.

### 2.5.2. Cross entropy

La entropía en la teoría de la información es la cantidad de información que puede haber en todas las posibilidades de una variable aleatoria. Por otra parte, en inteligencia artificial, es la cantidad de bits necesarios para transmitir un evento aleatorio a partir de una distribución probabilística.

Cross-entropy se basa en esta teoría y se utiliza para modelos de clasificación multi-clase donde la salida de este modelo es la probabilidad de que sea una de esas clases. Por lo tanto, lo primero que se hace es calcular la distancia de los valores «verdaderos», es decir, la clase correcta, para poder acercar al máximo posible a esta salida esperada.

Supongamos que $n$ es el número de clases, $t_i$ el valor «verdadero» de la salida (si no es la clase esperada este sería 0 y en caso contrario 1) y por último, $p_i$ la probabilidad de que sea la clase $i$, la función viene dada por:

$$CE = \sum_{i=1}^{n} t_i log(p_i)$$

## 2.6 Gradient descent

Después de estimar el error, la pregunta se convierte en, cómo optimizamos nuestra red neuronal dado el coste? La respuesta más común es «Gradient descent», un algoritmo de optimización de los parámetros que busca minimizar el error.

«Gradient descent» [21] busca el coste local mínimo, es decir, posiblemente no se encuentra la mejor opción pero encontraremos una opción buena. Para encontrar este mínimo local, primero necesitamos buscar la pendiente para orientarnos y poder bajar. En la figura 2.4 se puede ver la diferencia entre un mínimo local y un mínimo global. El mínimo global es el punto mas bajo de la función y el mínimo local es un punto bajo de una pendiente hacía abajo.

La pendiente se calcularía con la derivada del coste respecto al parámetro que queramos optimizar. Por ejemplo para optimizar los pesos $w$ tenemos que el gradiente es $\frac{\partial L}{\partial w}$. A partir de la pendiente, modificaremos el valor de los parámetros, ponderando la pendiente con el ratio de aprendizaje «lr» que tendrá mayor importancia cuanto mayor sea el «batch size».

El «batch size» es una dimensión añadida a la entrada que representa el número de entradas que queremos utilizar para calcular el coste y actualizar muestras de los parámetros. Esto es importante dado que, para actualizarlos, tenemos en cuenta mas de una entrada pero a su vez aumenta el tiempo de computación de las salidas.

Como podemos ver en la Figura 2.5, diferentes ratios («lr») pueden converger mejor o peor. El «lr» como su nombre expone, es un número que indica como de rápido queremos que aprenda nuestro modelo (red neuronal). Un número muy pequeño puede hacer que el aprendizaje sea muy lento, y uno muy grande converger a mínimos subóptimos que haría decrecer la precisión.

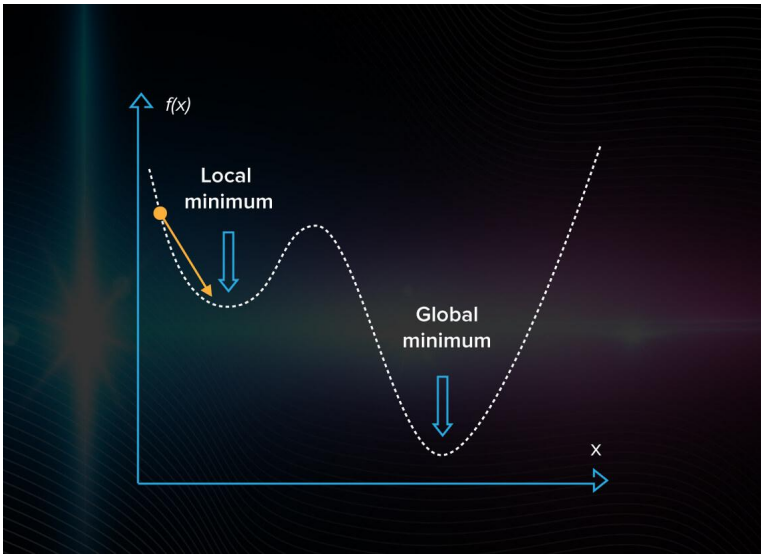



**Figura 2.4:** Gradient descent

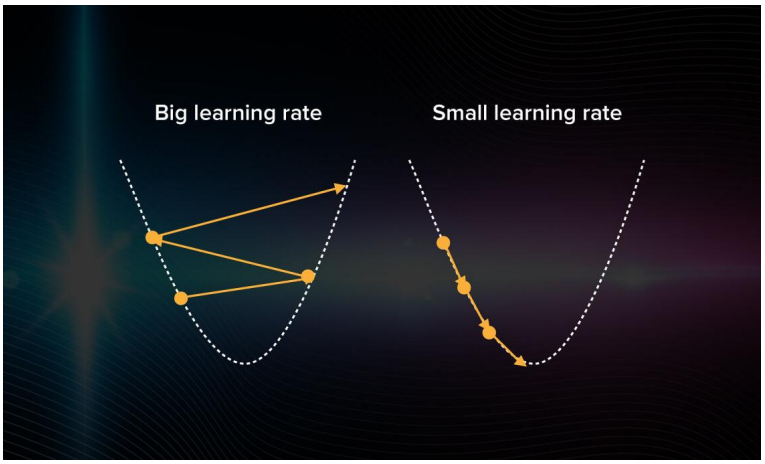



**Figura 2.5:** Diferencias en ratio de aprendizaje

Con este nuevo concepto podemos definir la formula más simple del «batch gradient descent» donde $\theta =$ parámetros, $lr =$learning rate, $J =$ Función de coste y $\nabla_\theta =$ Derivada o gradiente:

$$\theta = \theta - lr * \nabla_\theta J(\theta)$$

A partir de aquí vamos utilizar estas derivadas parciales que calculamos de los parámetros, como gradientes ya que son equivalentes en este contexto.

## 2.7 Backpropagation

Como hemos visto, con el «Gradient Descent», a partir de un error podemos actualizar los parámetros utilizando la derivada parcial pero hay un problema. Vamos a suponer que tenemos un modelo como el de la Figura 2.6 y queremos actualizar el peso $w1$ y por lo tanto, buscamos $\frac{\partial L}{\partial w_1}$.

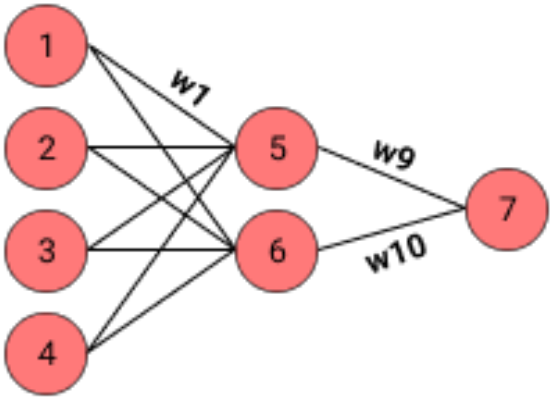



**Figura 2.6:** Red neuronal de 3 capas

Como podemos observar en la Figura 2.6 la pérdida $L$ es calculada basada en la salida de la neurona 7 el cual es:

$$o_7 = (o_5 * w_9) + (o_6 * w_{10}) + b_7$$

donde $o_i$ es la salida de la neurona $i$.

Actualizar $w_1$ requiere la pérdida causada por este parámetro, pero como hemos visto antes, solo sabemos la pérdida causada por la última neurona, es decir, necesitamos una manera de saber el error relacionado con el peso deseado.

En 1986, Rumelhart, Hinton y Williams publicaron un artículo [7] que presentaría como propagar el error hacía atrás y como vamos a ver ahora, se traduce en el uso de la «regla de la cadena» en la que conectamos el último resultado con uno anterior. Vamos a ver como se desarrollaría este calculo en el modelo para el primer peso.

Al calcular la pérdida obtenemos $\frac{\partial L}{\partial o_7}$ y como ya sabemos, $o_7 = (o_5 * w_9) + (o_6 * w_{10}) + b_7$. Después para calcular $o_7$ hemos utilizado $o_5$ que es el resultado de la neurona 5 y que a su vez, ha utilizado el parámetro del cual queremos saber su pérdida:

$$o_5 = (o_1 * w_1) + (o_2 * w_3) + (o_3 * w_5) + (o_4 * w_7) + b_5$$

.

Usando la regla de la cadena y, sabiendo que $w_1$ depende de $o_5$ y $o_5$ depende de $o_7$, podemos desarrollar la solución:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial o_7} \cdot \frac{\partial o_7}{\partial o_5} \cdot \frac{\partial o_5}{\partial w_1}$$

Obtenido el error con respecto de $w_1$, podemos actualizarlo con la formula de «gradient descent» tal que así: $\theta = \theta - lr * \nabla_\theta \frac{\partial L}{\partial w_1}$.

## 2.8 Tipos de Redes Neuronales

Las redes neuronales, no son únicas y por lo tanto, existen muchos tipos diferentes a parte de la ya vista. Vamos a ver cuales son estos tipos a continuación:

- **Red neuronal «Feedforward»**

  Este es el tipo mas básico y el que hemos visto en este capítulo. Se compone de una capa de entrada, cuantas capas ocultas como queramos y una capa de salida. Estas redes suelen ser usadas para el aprendizaje supervisado con datos numéricos. Las redes neuronales «feedforward» tiene algunas carencias como tratar con imágenes y datos secuenciales que se tratará de resolver con los siguientes dos tipos.

- **Red neuronal convolucional («CNN»)**

  Las «CNN» se usan principalmente para el tratamiento de imágenes de manera en la que podamos hacer más pequeñas las imágenes sin perder mucha información de ellas al *reescalarlas*. Estas redes usan los pixeles adyacentes para poder reducir un bloque de pixeles a otro bloque mas pequeño.

  Estas redes están compuestas por capas de convolución que siempre las suelen seguir capas de «pooling». Además después suelen haber capas densas (u otras) que se encargarían de clasificar la entrada. Cabe recalcar que una capa densa es equivalente a una capa de una red neuronal «feedforward». En la Figura 2.7 podemos ver visualmente como sería el caso de predecir qué número es el dígito escrito. Primero pasamos por dos convoluciones que reducirán las dimensiones y después pasaremos a las capas densas donde haremos la predicción.

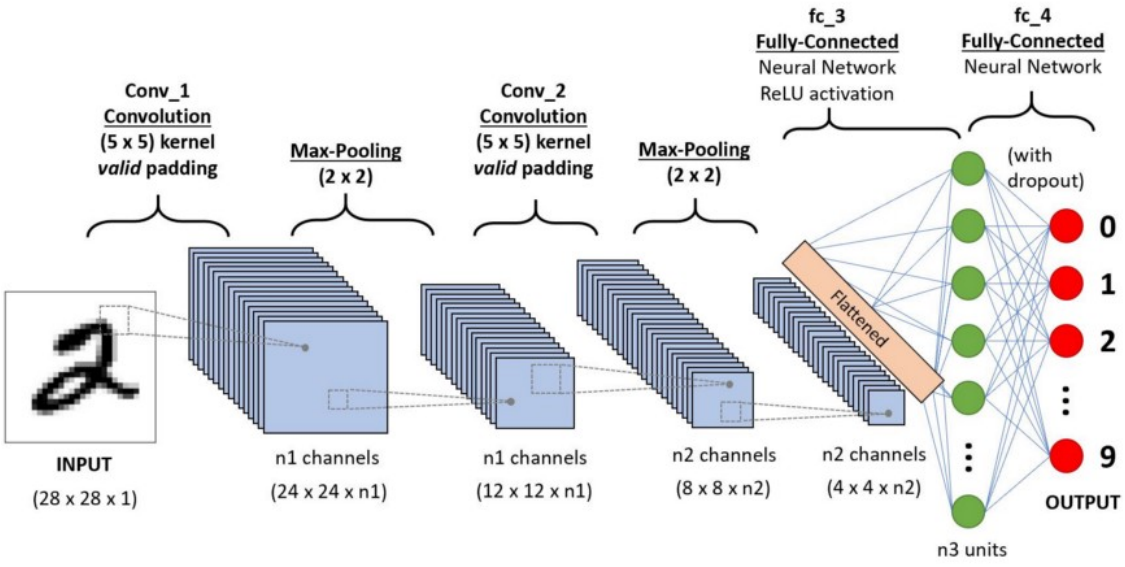



**Figura 2.7:** Red neuronal convolucional

- **Redes neuronales recurrentes («RNN»)**

  Las «RNN» son un tipo de red neuronal especializado para datos secuenciales. En términos simples esta red neuronal se retroalimenta con sus salidas para que éstas tengan utilidad para otras operaciones. No vamos a entrar en mucho más detalle, ya que en los próximos capítulos veremos con mas extensión como funcionan.

- **Transformers**

  Los «transformers» son la evolución natural de las «RNN» y por lo tanto un poco mas complejas. Hasta no hace mucho las aplicaciones de procesamiento de lenguaje natural eran hechas con «RNNs», pero tenían una limitación. Conforme una secuencia crece, el tiempo de ejecución de una «RNN» crece y por lo tanto puede llegar casos en los que sea poco eficiente. Los «transformers» entran en juego por

un mecanismo propio llamado «atención». El mecanismo de *atención* se encarga de escoger partes de una secuencia y elige si son importantes.

Estas redes neuronales son mayoritariamente utilizadas para capas de codificación-decodificación, es decir, tareas de traducción.

- **Red generativa anatagónica(«GAN»)**

  Estas redes neuronales son de lo mas diferente con respecto a otras. Se usan dos redes neuronales que se *enfrentan* y que tienen un rol especial. Una de ellas es la «generadora» que se encarga de generar datos falsos y por otro lado tenemos la «discriminadora» que intenta deducir cuales son falsas.

  Las «GAN» podemos utilizarlas por ejemplo, para deducir qué imagenes son falsas. Este es un problema que, conforme va evolucionando la «IA», es mas crítico porque están apareciendo herramientas como «DeepFake», que reemplaza la cara de una persona por otra y cada vez cuesta mas diferenciar la realidad.

## 2.9 Herramientas utilizadas

Para la realización de este trabajo se ha utilizado muchas herramientas diferentes con diferentes finalidades.

Empezando por la comunicación, este trabajo ha sido realizado en el Grupo de Arquitecturas Paralelas (*GAP*) con 4 profesores y 5 alumnos, cada uno con su trabajo con temas diferentes pero relacionados con *HELENNA*. Hemos utilizado *Microsoft Teams* para hacer reuniones semanales donde exponíamos nuestros avances y nuestros planes para la próxima semana. Por otra parte, usábamos *Slack* y *emails* para comunicarnos textualmente de forma mas asíncrona.

Para escribir el código se han utilizado diferentes herramientas que mejoran la productividad. Para empezar, se usa *Vim* [38] como editor de texto por defecto en conjunto con *Visual Studio Code* [39]. El sistema operativo usado es Ubuntu y se utiliza *ssh* para conectarse a diferentes *clusters*.

El proyecto de *HELENNA* está gestionado con el sistema de gestión de proyectos *git*. El código está escrito en *C* y algunas partes en *C++*, por lo que también usamos *CMake* [27] y *make* [30] para compilar. Para *debuggear memory leaks* hemos recurrido a *valgrind* [37].

Para acabar, se utiliza *Microsoft Excel* [28] para la generación de gráficas de resultados que emplearemos para validar la implementación de las redes neuronales recurrentes y, *Latex* [29] para escribir la memoria.

# CAPÍTULO 3
# Redes Recurrentes

En el capítulo anterior hemos introducido las redes neuronales más básicas. Sin embargo, presentan una limitación. Una de ellas es la capacidad de *aprender* a partir de una secuencia de datos, ya sea una oración dentro del libro de «Don Quijote», el precio alrededor de un periodo de tiempo de unas acciones o incluso una onda.

Entendemos datos secuenciales como un conjunto de elementos que tienen una relación entre si. Por lo tanto, tenemos un conjunto en el que el contexto es importante. Por ejemplo, *¿Quién es un buen chico?* es una frase que tiene sentido gracias a la combinación de las palabras y si quisiéramos analizar esta frase sintácticamente no se podría obtener fácilmente con una red neuronal normal pero en cambio, con una «RNN» es mucho más sencillo.

Otro ejemplo sería cómo predecir la velocidad de un vehículo. Con las redes neuronales básicas solo podemos analizar datos estáticos, en los cuales se tiene toda la información para inferir la salida para este dato. Por otra parte, si queremos saber la velocidad de un vehículo, a partir de un «frame» es imposible obtenerla. Por ello, como los humanos, podemos saber a que velocidad vamos gracias a puntos de referencias y a la diferencia de nuestra posición con relación del tiempo. Para ello, podemos usar las Redes Neuronales Recurrentes (*RNN*).

Como veremos, este nuevo tipo de red tiene diferentes abstracciones llamadas celdas que nos permitirán separar cada elemento en cada una de ellas y crear de alguna manera una especie de memoria que nos permitirá recordar elementos anteriores.

Primero veremos como usamos las «RNN» para crear las aplicaciones increíbles que hoy en día existen en el mundo. Después pasaremos a enumerar un amplio espectro de tipos y arquitecturas porque estas redes no son únicas. También veremos como cambia el proceso de «forwarding» con su nueva combinación de ecuaciones. Y por último, el «backpropagation», que tiene unos nuevos problemas pero, que explicaremos como se soluciona.

## 3.1 Aplicaciones Reales

Una de las dudas que se puede tener con las Redes Neuronales Recurrentes es cuál es su uso y cuando usarlas. La generación de texto es una aplicación de los usos de las *RNN* donde intentamos predecir la siguiente palabra a partir de la anterior. Por lo tanto, tendríamos una frase cualquiera que nos gustaría que se escribiera con nuestro modelo, y entrenamos este modelo con ella intentado predecir las palabras consecuentes.

La traducción de un idioma a otro también es una tarea obvia para nuestra red recurrente. Un ejemplo muy cercano a todos es la aplicación «Google translate» [22]. Esta aplicación esta fuertemente acoplada con el uso de las *RNNs* y está entrenada con una cantidad de datos enorme, que ha hecho que cada vez sea mas preciso. Este es un campo interesante, ya que a nosotros ya nos cuesta poder traducir frases que son complicadas de expresar con otro lenguaje pero, es muy interesante ver lo cerca que están estas aplicaciones de nuestra capacidad e incluso superior.

El reconocimiento de voz es otro caso muy actual y podemos ver sus resultado en productos como «Alexa» [41] o «Siri» [34] que son capaces de entender lo que decimos a un nivel todavía superficial pero que está creciendo exponencialmente. Estos tres ejemplos forman parte del procesamiento de lenguaje natural (*NLP*) que es una rama de la inteligencia artificial donde se trata el lenguaje natural, es decir, el que hablamos nosotros. Esta es una rama donde el uso de *RNNs* es abundante por la capacidad de *memorizar* cálculos anteriores como a continuación.

Otra aplicación de las «RNN» es en la predicción de series de tiempo [43]. Por ejemplo, podemos intentar predecir como fluctuará el precio de una acción de alguna empresa en el mercado bursátil analizando toda la trayectoria. Además puede utilizarse para predicción del tiempo o procesamiento de señales ya sea de sonido o imágenes.

Como podemos imaginar, la utilidad de las «RNN» es muy grande y por lo tanto, es interesante poder disponer de ella en cualquier herramienta de aprendizaje que usemos y, en nuestro caso, la implementaremos como veremos posteriormente en la herramienta *HELENNA*.

## 3.2 Estructura de una Red Recurrente

Los datos secuenciales son la esencia de una Red Neuronal Recurrente y, vamos a representarlos como vemos en la Figura 3.1. Como podemos apreciar, la forma mas sencilla de representar esta red es con lo que llamaremos «celdas», vistas como cuadrados en la Figura 3.1. Cada «celda» recibe una parte de la entrada completa y también estas pueden tener una salida.

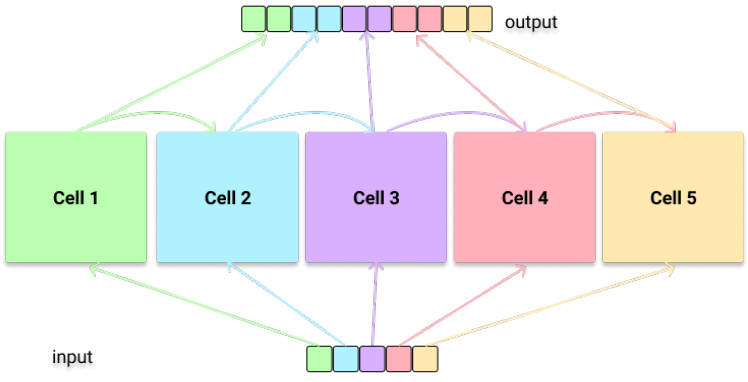



**Figura 3.1:** Estructura de una RNN

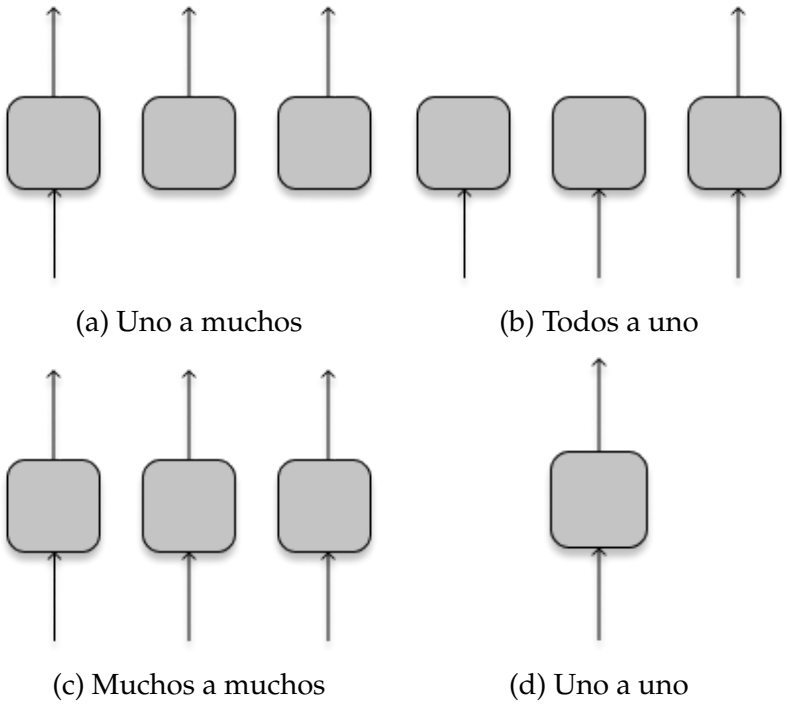
La clave de esta estructura es el estado intermedio que llamaremos *hidden_state*. Este *hidden_state* es transmitido a la siguiente «celda» que realizará los cálculos basados en su parte de la entrada y a partir del estado intermedio anterior. Esto es lo que nos per-

mite *memorizar* cálculos de un tiempo anterior para poder predecir con un contexto mas complejo.

## 3.3 Arquitecturas y tipos de RNN

Las *RNNs* las podemos categorizar en 4 arquitecturas diferentes y por muchos tipos.

Empezando por las arquitecturas, que podremos visualizar en la figura 3.2, tenemos una de *uno a muchos*, donde solo la primera «celda» recibe datos pero todas crean una salida. Una de las principales aplicaciones de esta es escribir un pie de foto a una imagen donde la primera entrada sería la imagen y las salidas las palabras del pie de foto.



(a) Uno a muchos (b) Todos a uno

(c) Muchos a muchos (d) Uno a uno

**Figura 3.2:** Arquitecturas de las RNN

En las *todos a uno*, solo la última «celda» produce una salida y todas las «celdas» reciben una entrada. Imaginemos que queremos saber la probabilidad de que un usuario vaya a hacer *click* a partir de una secuencia de acciones anteriormente realizadas. En este caso cada «celda» recibe cada acción y con toda esta información calculamos un solo resultado que representa si hace «click» o no.

Las *muchos a muchos*, como su nombre describe, es la arquitectura en la cual todas las «celdas» reciben una entrada y producen una salida. Su uso se focaliza en los modelos llamados «seq2seq» en los que a partir de una secuencia se crea otra secuencia. Por lo tanto, son útiles para traducir de un idioma a otro y para codificar un mensaje o decodificarlo. Este ultimo caso es el mas conocido ya que en modelos gigantes siempre hay partes en la cual se necesitar codificar y decodificar que se traduce en dos capas de «RNNs» como se puede apreciar en la Figura 3.3.

Por último, las *uno a uno*, no son muy útiles ya que es equivalente a una única neurona.

A continuación, vamos a presentar los diferentes tipos de «RNNs» que existen, ya que hay muchos tipos. El tipo mas básico y el que implementaremos como veremos mas tarde se puede apreciar en la Figura 3.4. Como podemos observar cada celda tiene con el

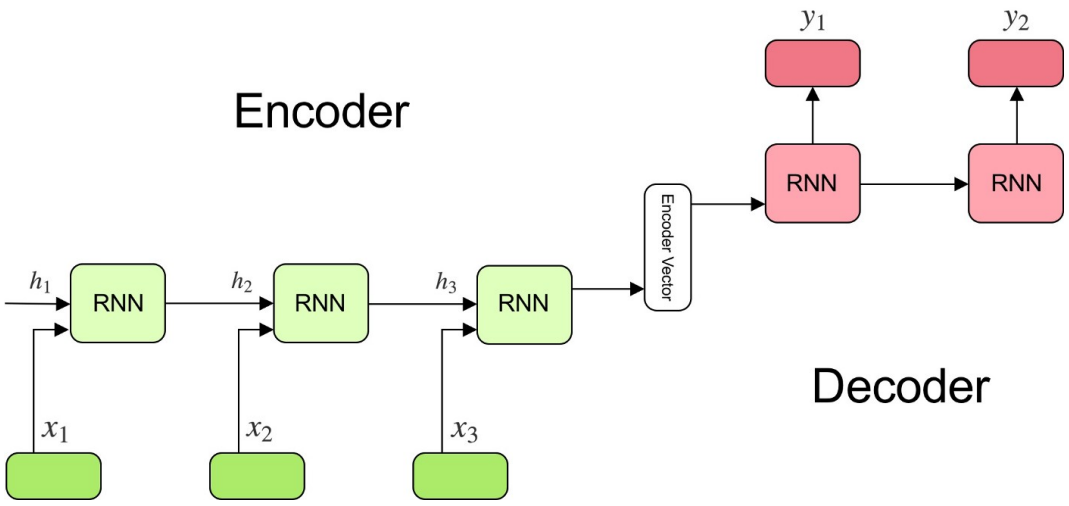



**Figura 3.3:** Proceso de codificar y decodificar

*hidden_state* anterior y la entrada actual una multiplicación con sus pesos respectivos y a la suma de estos se le aplica la función de activación. Este tipo tiene un problema. Como podemos observar, el estado intermedio se calcula con multiplicaciones y produciendo así una cadena que puede crear una salida demasiado grande que pueda desbordar el tamaño de un «float». Este problema se aprecia al calcular los gradientes de cada parámetro pudiendo llegar el caso de que haya «Vanishing Gradients» o «Exploding Gradients».

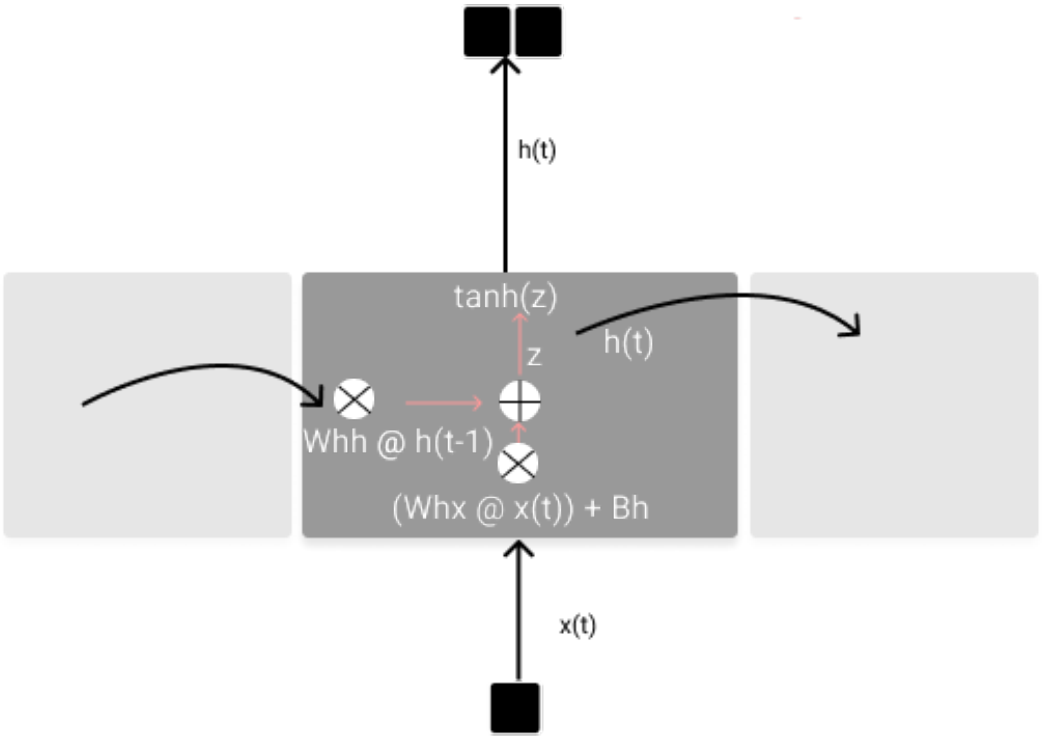



**Figura 3.4:** Representación de una celda en una RNN

«Vanishing Gradients» es un problema que ocurre en redes neuronales en el proceso de «backpropagation» donde se usa la regla de la cadena. A cada iteración, el gradiente se hace mas pequeño y puede llegar el caso que sea tan pequeño que haga el aprendi-

zaje demasiado lento. Esto es causado en parte por las derivadas de ciertas funciones de activación como la sigmoide o la tangencial.

Otro problema presente es «Exploding Gradients». Este caso se podría considerar el contrario al anterior, porque a cada iteración el gradiente se acumula más y creando un posible desbordamiento que produciría resultados «NaN».

Hay un tipo de «RNN» que es capaz de superponerse a estos problemas y se llama «LSTM» o «Long Short-Term Memory». Las «LSTM» introducen un nuevo termino que se llaman puertas o como se conocen más, «Gates» donde se toman diferentes decisiones sobre la estructura de la celda. Además de un estado intermedio, también se pasa un estado de celda a la siguiente celda que se denotará con $c_t$.

Para este tipo tenemos tres diferentes puertas: «Forget gate» que se encarga de decidir cuánta información usaremos del estado de la celda anterior y cuánta desecharemos. También está el «Input Gate», decidiendo cuánta información añadiremos al estado de celda. Y por último tenemos el «Output Gate», encargado de decidir qué información se envía a la siguiente iteración (celda).

Por último vamos a presentar otro tipo muy utilizado que es una variación de las «LSTM» en la que solo existen dos puertas. Las *Gated Recurrent Unit* («GRU») se componen de la «Update Gate» que es básicamente la combinación de la «Input Gate» y «Forget Gate», donde vemos que información queremos mantener y qué información añadir. La segunda puerta es la «Reset Gate» que decide qué pasar a la siguiente iteración y también junta el estado intermedio con el estado de la celda.

Además de estos tipos hay redes mas complejas como las redes «Elman», redes «Jordan», las de «Hopfield», bi-direccionales... Las Redes Recurrentes como hemos podido explorar, tienen un amplio uso gracias a la gran variedad de arquitecturas y tipos que nos permiten escoger cada par (tipo y arquitectura) por la utilidad del problema que queramos resolver.

## 3.4 Forwarding

En este apartado vamos ya a entrar en detalle qué operaciones realizamos en cada celda de una «RNN» y como estos datos se transmiten de un lugar a otro. Como veremos, las operaciones no son muy complejas pero es importante seguir un orden para conseguir el resultado esperado.

De aquí en adelante vamos a hablar de las «RNN» como una de tipo simple y con arquitectura de *todos a todos*. La razón de escoger la más simple es porque es necesaria para después desarrollar clases mas complejas como las anteriormente mostradas. En cuanto a la arquitectura, todas las celdas, sin importar la arquitectura, siempre hacen los mismos cálculos. Solo se cambia donde introducimos los datos y qué datos recogemos.

Primeramente vamos a ver cuales son los parámetros de la «RNN». Vamos a tener el $input\_size$ que denota el tamaño de la entrada de cada celda, el *hidden_size* que es el tamaño del estado intermedio y por último el número de iteraciones o celdas, que será simplemente el tamaño de la entrada total, es decir, la salida de la capa anterior, dividido por el $input\_size$. En la Figura 3.4 podemos ver mejor el orden de las operaciones y los parámetros que se hacen uso.

Con estos parámetros, ya podemos definir los «weights» (pesos) y los «biases» (márgenes). Empezando por la entrada, tenemos $x$, el cual será una matriz de dimensiones $input\_size \times 1$. En conjunción con la entrada tenemos $W_{hx}$ y $B_x$ que son la matriz de pesos y el margen respectivamente. $W_{hx}$ tiene las dimensiones $hidden\_size \times input\_size$ que

nos permite transformar la entrada a las dimensiones del estado intermedio. $B_x$ tiene las dimensiones $hidden\_state \times 1$.

Por otro lado tenemos $h_t$, que es de tamaño $hidden\_state \times 1$ y su respectiva matriz de pesos $W_{hh}$ tiene unas dimensiones $hidden\_state \times hidden\_state$. Por otra parte, tenemos también el «bias» del estado intermedio que es $B_h$ con unas dimensiones de $hidden\_state \times 1$.

Este conjunto de parámetros ha sido sometido a una evaluación de diseño, porque dependiendo diferentes plataformas de entrenamiento existente y definiciones de «RNN», añaden mas parámetros para poder establecer una salida que se calcula a partir del estado intermedio con una función de activación extra. Nosotros hemos decidido prescindir de ella porqué nos permite que esta estructura sea más flexible y en caso de querer transformar el estado intermedio a un tipo diferente de salida podemos añadir más capas al modelo a decisión del programador. Este diseño esta inspirado en «PyTorch», una de las plataformas de entrenamiento con mas renombre.

Con todos los parámetros ya definidos, podemos pasar al corazón de la «RNN», donde todas se realizan los cálculos. Como hemos visto previamente, cada celda se centra en calcular un estado intermedio a partir del estado intermedio anterior y la entrada de la celda. Los cálculos de esta celda se puede resumir en esta función:

$$h_t = tanh((W_{hx} \times x_t + B_x) + (W_{hh} \times h_{t-1} + B_h))$$

Vamos a ver primero cómo evoluciona la función empezando por la operación realizada con la entrada $W_{hx} \times x_t + B_x$. Como podemos apreciar, usamos la matriz de pesos $W_{hx}$ para hacer una multiplicación matricial con la entrada y gracias a las dimensiones de esta matriz podemos transformar la salida de esta operación a un tamaño de $hidden\_size \times 1$. Después procedemos a sumar con el «bias» $B_x$ para así redirigir para arriba o abajo cada elemento conforme cambie este parámetro.

Por otra parte vamos a realizar otro cálculo. $W_{hh} \times h_{t-1} + B_h$ simplemente multiplica el estado intermedio anterior por su matriz de pesos para dar mas o menos importancia a cada elemento del estado intermedio y después sumamos el «bias» $B_h$ como en la anterior operación.

Estos dos valores que hemos calculado los sumaremos y le llamaremos $z$. Una vez calculado este valor procedemos a enviarlo a la función de activación especificada. Normalmente, las dos funciones de activación mas utilizadas son la *tanh* y la *ReLU*, dependiendo del problema en cuestión. En este caso vamos a suponer que es la *tanh* y por lo tanto tendríamos el resultado final escrito como $h_t = tanh(z)$.

Con el «hidden_state» ya calculado podemos proceder a enviar este dato a la salida global de la «RNN» porque, como hemos decidido anteriormente, el «hidden_state» para nosotros equivale a la salida. Además, enviaremos el estado intermedio a la siguiente celda para que pueda realizar los mismos cálculos con diferente entrada.

Cuando ya tenemos todos los «hidden_state» tendremos una lista con todos ellos y esto corresponde al resultado de la red recurrente, la cual servirá para enviar a la próxima capa si es que hay alguna mas. Al final del proceso, las «RNN» están ocultas de tal forma que solo haya que especificar los primeros parametros que hemos visto y simplemente hay que enviar el *buffer* de entrada y a la «RNN» se encarga de producir el *buffer* de salida.

Con respecto a la inicialización de parámetros. El estado intermedio $h_0$ lo iniciaremos a 0 al igual que el «bias» $B_h$. Por otro lado las matrices de pesos las inicializaremos con «Xavier initialization». Hay muchas formas de inicializar matrices de pesos y cada una tiene sus beneficios pero esta es de las más empleadas.

## 3.5 Backwarding

Como cualquier proceso de «forwarding» en aplicaciones de redes neuronales, tiene que haber un proceso de «backwarding» donde calculamos los gradientes de cada parámetro para posteriormente actualizar estos parámetros dependiendo de la optimización que queramos utilizar.

Este proceso no cambia mucho con respecto a una red neuronal profunda, pero como hemos visto en el apartado anterior, las operaciones son mas complejas.

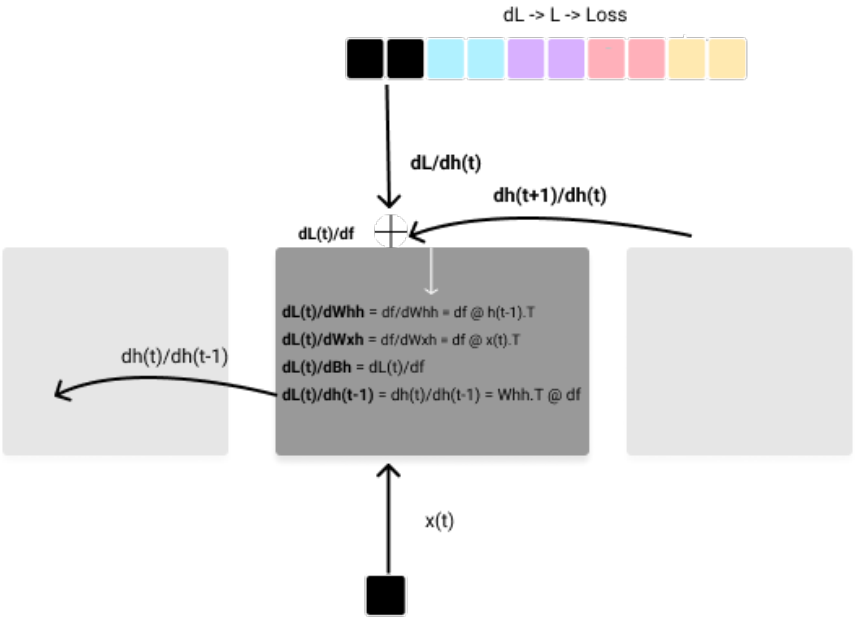



**Figura 3.5:** Operaciones de backpropagation en una celda

Anteriormente, el algoritmo de «backpropagation» nos ayudaba a propagar el error entre diferentes capas pero, en las redes recurrentes, necesitamos desenrollar la pérdida en diferentes tiempos (celdas). La pérdida de la «RNN» viene dada por un *buffer* con tamaño equivalente al numero de celdas por el *hidden_state* y por lo tanto, por cada celda tendrá su parte de la pérdida.

A la técnica que usaremos para calcular los gradientes en esta red recurrente le llamaremos «backpropagation trough time» (bptt). La única diferencia con respecto a una «NN» es que las matrices de pesos y el margen son compartidos entre cada iteración. Esto quiere decir que para calcular los gradientes de los pesos necesitaremos acumular los gradientes calculados empezando por la última celda hasta la primera.

Sabiendo que los parámetros son compartidos entre diferentes celdas, vamos a explicar cómo calcularíamos los gradientes a alto nivel. En otras palabras, vamos a intentar no entrar en detalles primero y después veremos como se traduce en las ecuaciones que se pueden observar en la Figura 3.5.

El proceso de «bptt» empieza como cualquier otro proceso de «backpropagation», recibiendo la pérdida dada la salida. En este caso vamos a imaginar que la «RNN» es la última capa antes de la salida. Gracias a una función de pérdida como podría ser «MSE» o «CE» obtendríamos la pérdida que ahora procederemos, a suministrar a la «RNN».

Ya con el error obtenido, distribuiremos una parte a cada celda dado el tamaño de salida de cada una de ellas. Empezaremos por la última celda ya que como ya hemos visto antes, es un proceso que va hacía atrás. Con la pérdida podemos calcular los gradientes de todos los parámetros dado la entrada y el estado intermedio. Como sabemos que estos parámetros son compartidos, tendremos que acumularlos de alguna manera. Por lo tanto, procederemos a propagar el error celda anterior. Con esta suma, volveremos a tener el error dado la actual celda y podemos volver a calcular los gradientes de los parámetros. Procederemos a acumular estos gradientes que vayamos calculando y repetiremos el proceso hasta llegar a la última celda. Cuando lleguemos al final ya podremos proceder a actualizar los parámetros y daremos por acabado el proceso de «backwarding» en esta red.

Antes, en una red neuronal corriente se hubiera calculado el gradiente de un peso o de un margen y lo habríamos actualizado sin mas. Esto crea un poco mas de complejidad a la hora de realizar la regla de la cadena pero tenemos unos cálculos parecidos y lo único diferente es que tendremos un sumatorio de los gradientes para así poder acumularlos y después proceder a actualizar.

Finalmente, con esta explicación a alto nivel, podemos empezar a definir como serán las operaciones y como llegaremos hasta el final. Volveremos a suponer que la «RNN» es la última capa y por lo tanto, la pérdida viene dada por la función de pérdida. Además, vamos a suponer que solo tenemos tres celdas para intentar simplificar las ecuaciones. La Figura 3.5 tiene una mayor visualización de las operaciones que vamos a definir.

Empezamos como antes, por el error obtenido. Vamos a asumir que la pérdida nos viene dada como $L$ y ésta contiene la pérdida relacionada con cada celda. A continuación procederemos a definir la pérdida de cada celda que vendrá dada como $\frac{\partial L}{\partial h_t}$. Podemos concluir entonces que:

$$L = \frac{\partial L}{\partial h_1} + \frac{\partial L}{\partial h_2} + \frac{\partial L}{\partial h_3}$$

Como hemos dicho antes, $L$ sería un lista con las diferentes pérdidas, con que, cada una de las sublistas tendrá el tamaño anteriormente visto, *hidden_state*. Con esto ya podremos distribuirlo a cada celda. Este proceso lo veremos con mas detalles en la implementación ya que, como veremos, tendremos una lista especial.

Como ya sabemos, $h_t = tanh(z)$. La pérdida debido al estado intermedio último es el que podemos observar en la descomposición de $L$, que es $\frac{\partial L}{\partial h_3}$. A partir de aquí, vamos a seguir la regla de la cadena porque necesitamos concatenar errores dadas las operaciones anteriores y con ello obtener los gradientes.

Ahora necesitamos calcular la pérdida dada $z$, ya que sería el anterior valor calculado a partir de los parámetros. Entonces, para calcular $\frac{\partial L}{\partial z_3}$ tendremos en cuenta la pérdida a partir del estado intermedio y con ello tendríamos:

$$\frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3}$$

Este último paso nos abre las puertas al cálculo de todos los gradientes de los parámetros $W_{hh}$, $W_{hx}$, $B_x$, $B_h$ respectivamente:

$$\frac{\partial L}{\partial W_{hh}} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial W_{hh}}$$

$$\frac{\partial L}{\partial W_{hx}} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial W_{hx}}$$

$$\frac{\partial L}{\partial B_x} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial B_x}$$

$$\frac{\partial L}{\partial B_h} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial B_h}$$

Si fuera el caso de una «NN» típica con esto ya procederíamos a actualizar los parámetros mencionados anteriormente, pero como son compartidos entre celdas vamos a tener que calcular el gradiente de cada celda y después sumarlos. Como se ha comentado antes, el estado intermedio 3 depende del estado intermedio 2, por lo tanto, vamos a tener que calcular la pérdida dado el estado intermedio 2 que se traduce a:

$$\frac{\partial h_3}{\partial h_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2}$$

Con la pérdida de la celda 3 dada la 2 podemos proceder a actualizar la pérdida *global* de la celda 2:

$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial h_2} + \frac{\partial h_3}{\partial h_2}$$

Cabe recordar que $\frac{\partial L}{\partial h_2}$ al principio es la sublista de $L$.

Con estas relaciones ya podemos definir cómo calcular los gradientes que nos servirán para el último paso y así hacer que nuestra «RNN» sea capaz de aprender de sus fallos (y que sea útil).

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=0}^{3} \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial W_{hh}}$$

$$\frac{\partial L}{\partial W_{hx}} = \sum_{t=0}^{3} \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial W_{hx}}$$

$$\frac{\partial L}{\partial B_x} = \sum_{t=0}^{3} \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial B_x}$$

$$\frac{\partial L}{\partial B_h} = \sum_{t=0}^{3} \frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial B_h}$$

Y por último, vamos a desarrollar las pérdidas dada cada celda para que las ecuaciones anteriores queden completas porque $\frac{\partial L}{\partial h_t}$ no depende sólo del la pérdida de la «RNN»:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_1} + (\frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1})$$

Si hubiésemos decidido añadir más operaciones a este proceso no cambaría mucho a parte de añadir mas capas de *dependencias* donde cada operación viene dada de otra. Además se usa derivadas parciales porque como ya sabemos, las funciones vienen dadas por diferentes variables y no solo una.

# CAPÍTULO 4
# Implementación en HELENNA

Después de describir de las redes recurrentes empezamos a preguntarnos cómo vamos a traducir todo este cúmulo de operaciones y abstracciones, así como vamos a complementar redes existentes y funciones para poder utilizarlo en aplicaciones reales y así poder resolver problemas que antes no podíamos.

Este trabajo se enfoca en implementar las «RNN» en la plataforma de entrenamiento de redes neuronales *HELENNA*. *HELENNA* nos permite hacer todo lo que esperaríamos de un «framework» de este estilo como podría ser *Tensorflow* o *PyTorch*, es decir, definir, entrenar y inferir a partir de un modelo de red neuronal.

*HELENNA* está desarrollado en *C* porque queremos tener control sobre todo lo que pase a bajo nivel y como es de esperar, tienes sus ventajas e inconvenientes. La principal pega es que tendremos que preocuparnos de no generar un acceso de memoria incorrecto y además, dada la menor expresividad del lenguaje tendremos que escribir más lineas. No obstante, esta aplicación requiere ser lo más eficiente posible. Por lo tanto, *C* nos permite tener todo el control que necesitamos para poder exprimir todo el potencial en cada operación.

Para empezar, estudiaremos cómo está estructurado el código fuente ya que tenemos a mano un repertorio extenso de funciones que nos van a ser muy útiles a la hora del desarrollo. Además, de las carpetas donde tendremos los «datasets» o la definición de los modelos. También veremos unos ejemplos de cómo funciona en general desde la compilación hasta la ejecución.

Después pasaremos a ver cómo extenderemos la aplicación para una capa extra, en este caso la «RNN» y, definiremos con estructuras de *C* las celdas con todas las variables que necesitemos para emular los cálculos previamente vistos.

Para finalizar, con las estructuras ya establecidas, procederemos a completar las famosas funciones de «forwarding» y «backwarding» especificas de las «RNN» que ya hemos definido anteriormente, pero veremos al fin todo el código necesario para hacerlo funcionar. Sin duda, en estas dos funciones, encontraremos la gran parte del código.

## 4.1 Arquitectura de HELENNA

*HELENNA* es una «framework» que permite la definición, entrenamiento y inferencia de redes neuronales. El código fuente está escrito en *C* además de tener algunas partes escritas en *C++*. Las siglas de *HELENNA* vienen de «*HE*terogeneous *LE*arning *N*eural *N*etworks *A*pplication» por el hecho de que tiene el objectivo del uso de arquitecturas heterogéneas de cómputo.

Una de las principales características de *HELENNA*, en contraste con «PyTorch» o «Tensorflow», es la definición de las redes neuronales desde un fichero de texto plano a diferencia de estas dos últimas, que se definen en código.

Vamos a ver un ejemplo de definir una red neuronal con *HELENNA*. Para ello vamos a utilizar el «dataset» de *MNIST*. *MNIST* es una base de datos de imágenes de dígitos escritos a mano con una dimensión de $28x28$.

```
input   fc 784
fc1     fc 10
relu1   relu
softmax softmax
```

**Código 4.1:** Definición red neuronal para MNIST simple

Como podemos ver en el código 4.1 tenemos una primera capa de entrada con 784 unidades equivalente a $28 \times 28$. La siguiente capa es una capa densa que produce una salida de 10 dígitos, ya que queremos los dígitos desde 0 hasta 9. Después aplicamos a estas salidas la función de activación *ReLU* porque no queremos números negativos. Y por último, aplicamos una capa *softmax* que se encargara de pasar las 10 salidas a probabilidades y por lo tanto al final tenemos la probabilidad de que sea cada dígito.

En «PyTorch» en cambio, para definir la misma red neuronal debemos crear un *script* en *Python* envolviendo el modelo en una clase que sobreescriba la función de «forward». Como podemos ver en el código 4.2, en «PyTorch» hemos tenido que hacer un poco más de trabajo ya que mezclamos parte de la definición de la red neuronal con la lógica de «forwarding» y, aunque siga siendo simple, al crecer el modelo se hace mas difícil.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        return F.log_softmax(x)
```

**Código 4.2:** Definición de red neuronal MNIST en PyTorch

Al igual que otras aplicaciones de entrenamiento mas maduras, podemos definir redes neuronales mas complejas, como por ejemplo podría ser con «CNNs». En el código 4.3 podemos ver como primero tendremos una capa de convolución seguida de una capa de «maxpooling». Después de reducir el tamaño de la imagen, pasamos el resultado a la última capa densa de 10 unidades como en el caso anterior.

```
input fc 784
conv1 conv WI 28 HI 28 CI 1 PW 1 PH 1 SW 1 SH 1 KW 3 KH 3 CO 1 MDF 1
relu1 relu
maxp1 maxpooling SW 2 SH 2 KW 2 KH 2 CO 1
fc1 fc 10
softmax softmax
```

**Código 4.3:** Definición red neuronal MNIST con convolución

### 4.1.1. Estructura de Ficheros

*HELENNA* esta compuesta de muchos ficheros diferentes que vamos a describir a continuación. Vamos a nombrar la gran mayoría y específicamente los que tienen una mayor

relación con este trabajo. Para empezar vamos a ver cuales son los ficheros principales y después pasaremos a los ficheros de las capas.

- **main.c**: Este es el punto de entrada del programa donde se inicializará los argumentos necesarios para los procesos internos. También llamará a la creación de «datasets», el «parseo» del modelo y para finalizar llamará a que empieze el entrenamiento o inferencia.
- **datasets.c**: Aquí es donde se encuentra el código encargado de leer los diferentes tipos de formatos de «datasets» soportados. Entre ellos se encuentran el formato «bin», «idx», «master» y «nnsim».
- **fit.c**: El bucle principal de entrenamiento se encuentra en este fichero. Se encarga de empezar el «forwarding» de la red neuronal. Posteriormente calcula la pérdida y, para acabar, empieza el «backwarding» para actualizar los parámetros.
- **inference.c**: Este fichero es parecido al anterior pero especializado para la inferencia.
- **globals.h**: En la cabecera *globals.h* nos encontramos con todo tipo de variables globales además de estructuras globales que son compartidas en todo el programa.
- **topology.c**: Todo lo relacionado con adaptar el fichero del modelo y crear la estructura de la red neuronal se hace en este fichero.
- **arithmetic.c**: Aquí reside el punto de entrada para todas las operaciones matriciales soportadas. Este fichero no contiene como están implementadas las operaciones, sin embargo, redirige al fichero correspondiente dependiendo si estamos usando la *CPU*, *CUBLAS*, *MKL* o *AVX*.
- **mem_allocate.c**: Todos los vectores que usamos en todo el programa y que usaremos en la implementación de las «RNN» se crean en este fichero. Estos vectores o «Tensores» estan implementados y abstraidos con *buffer*.
- **loss_functions.c**: Las funciones de pérdida están definidas en esta fichero. Entre ellas encontramos la «MSE» y «CE».
- **model.c**: Todo lo relacionado con guardar y cargar un fichero que contiene el modelo entrenado se encontrará aquí.

Las diferentes capas de una red neuronal estan implementadas en ficheros diferentes como pueden ser los siguientes:

- **base_layer.c**: Capa base de cada una de las siguientes capas.
- **fully_connected.c**: Capa densa
- **conv.c**: Capa convolucional.
- **avgpooling.c**: Capa max pooling.
- **avgpooling.c**: Capa average pooling.
- **relu.c**: Capa ReLU.
- **softmax.c**: Capa softmax.
- **sigmoid.c**: Capa sigmoidea.

### 4.1.2. Tensores y Buffers

En las aplicaciones de aprendizaje automático nos pasamos la mayoría del tiempo representando numéricamente valores escalares, vectores y matrices. Por lo tanto, llega un momento que necesitamos una abstracción de estas estructuras que llamamos «Tensores». Los «Tensores» son contenedores de datos que pueden ser utilizados con diferentes dimensiones. Una de las características mas importantes de los «Tensores» es que estos pueden definir las relaciones entre diferentes vectores.

Ya con los «tensores» podemos interpretar un «tensor» de 0 dimensiones como un escalar, uno de 1 dimensión como un vector y con 2 dimensiones una matriz.

Los «tensores» están implementados como *buffers* en *HELENNA* y como se puede observar en el código 4.4 tenemos *info_id* que es la lista de *buffers* que vamos a ir creando. Los *buffers* vienen definidos por la estructura *idDato* que dependiendo de si estamos usando la *CPU*, *MKL*, *OCL* o *CUDA*, el *buffer* será de un tipo o de otro. Al ser una estructura *union*, solo se puede dar uno de los casos nombrados.

```
typedef union { // union used in mem_allocate.c
    void *id_ptr; //CPU
    void *id_mkl; //MKL
    cl_mem id_ocl; //OCL
    float *id_cuda; //CUDA
} idDato;

extern idDato *info_id;
```

**Código 4.4:** Implementación de tensores en HELENNA

Por lo tanto, un *buffer* o «tensor» en *HELENNA*, es un *array* de datos que guardamos en otro *array*. Cada *buffer* está indexado con un *int* para que podamos acceder a el directamente. Este identificador es obtenido al llamar a *fn_allocate_buffer*. Esta función la podemos ver en el código 4.5 y como podemos observar, recibe como parámetros un *unsigned int size* que es el tamaño del *buffer* que vamos a crear, además de la capa y el tipo de buffer que lo usamos mas para depurar.

```
int fn_allocate_buffer(uintptr_t size, int layer, int buffer_type);
int fn_deallocate_buffer(int id_mem);
```

**Código 4.5:** Función de ubicación y desubicación de un buffer

En *C* es tan importante crear *buffer* como liberarlos, para ello usaremos *fn_deallocate_buffer* que, a partir de un identificador de un *buffer*, liberaremos la memoria relacionada con el *buffer*. No liberar espacio en memoria puede tener severos problemas de *memory leak* en el programa si no se tiene cuidado.

Con un identificador de un *buffer* y sabiendo las dimensiones podemos empezar a realizar operaciones con ellos. La mayor parte de estos *buffer* los tenemos definidos en el fichero *arithmetic.h* y podremos ver el contenido en el código A.1. Hay muchos tipos de operaciones disponibles pero, principalmente, usaremos multiplicaciones matriciales (*fn_matmul*), sumas y restas (*fn_matsub*, *fn_matadd*) y algunas otras que veremos con mas atención en las próximas secciones.

Relacionado con el soporte para la depuración, una de las funciones que mas utilizamos para observar como evoluciona el estado del programa es la del código 4.6. Al estar programado en bajo nivel y, al estar trabajando con matrices de gran tamaño, necesitamos comprimir los datos de los *buffer* para poder inspeccionar el programa con mayor eficiencia.

```
void fn_print_stats_vector(int mid_v, size_t size);
```

**Código 4.6:** Función para debuggear buffers

Podemos ver en 4.7 como sería la salida de la funcion anterior. Como podemos observar hemos imprimido estadísticas del *buffer* con identificador 20, de 4096 elementos que tiene un valor mínimo, un valor máximo y la media. Si quisiéramos inspeccionar este *buffer* sin una función como esta, tendríamos que ir elemento a elemento y sería un proceso mas tedioso.

```
id 20 (none l 1) Size 4096 min -0.058800 max 0.063383 avg 0.00002616
```

**Código 4.7:** Ejemplo de fn_print_stats

## 4.2 Extendiendo HELENNA con soporte para redes RNN

*HELENNA* es fácilmente extensible. Para añadir una nueva capa en la plataforma simplemente tenemos que añadir un fichero nuevo y, extender unas partes del código.

Cada capa debe definir unos métodos específicos para poder encajar en el sistema. Al estar programado en *C*, no disponemos de herencia de objetos como la que disponemos en la programación orientada a objetos. No obstante, disponemos de algo parecido.

En *HELENNA*, cada capa esta abstraída en una estructura llamada *layer* que podemos observar en el código A.2. Esta estructura tiene muchas variables, pero lo que nos interesa ahora son las funciones del final mostrada en el código 4.8.

```
void (*fn_parse)(int layer, char *parameters);
void (*fn_allocate)(int layer);
void (*fn_deallocate)(int layer);
void (*fn_initialize)(int layer);
void (*fn_forward)(int layer, int fbe, int lbe, int training);
void (*fn_backward)(int layer, int minibatch_size);
void (*fn_propagate_error)(int layer, int minibatch_size);
```

**Código 4.8:** Funciones de una capa

Al «parsear» la topología de la red neuronal, dependiendo del tipo de capa con la que nos encontremos, estableceremos las funciones con relación a la capa esperada. En la función 4.9, si nos encontramos con la capa de tipo «RNN», estableceremos las funciones correspondientes a la capa «RNN» que veremos con mas detalle después. Como cabe esperar, estas funciones deben estar escritas en la cabecera A.5.

```
static void fn_layer_set_functions(int l)
{
    layer *lptr = layers_ptr[l];

    switch (lptr->ltype)
    {
    case LAYER_RNN:
        lptr->fn_parse = rnn_parse_tp;
        lptr->fn_allocate = rnn_allocate_tp;
        lptr->fn_deallocate = rnn_deallocate_tp;
        lptr->fn_initialize = rnn_initialize_tp;
        lptr->fn_forward = rnn_forward;
        lptr->fn_backward = rnn_backward_tp;
        lptr->fn_propagate_error = rnn_propagate_error_tp;
        break;
  ...
    }
```

```
}
```

**Código 4.9:** Estableciendo funciones de la capa

El tipo *LAYER_RNN* esta definido en *globals.h* bajo el *enum e_layer_type* como se puede ver en el código 4.11. Con el ya podemos identificar nuestra nueva capa que ya será visible para toda la aplicación.

```
typedef enum e_layer_type {
    LAYER_FULLY_CONNECTED,
    LAYER_RNN,
    LAYER_CONVOLUTIONAL,
    LAYER_CONVOLUTIONAL_TRANSPOSE,
    LAYER_MAXPOOLING,
    LAYER_AVGPOOLING,
    LAYER_RELU,
    LAYER_SIGMOID,
    LAYER_SOFTMAX,
    LAYER_RESHAPE,
    ...
} layer_type;
```

**Código 4.10:** Definir tipos de capas en HELENNA

Además de esta preparación para crear una capa, hemos añadido el código suficiente para compilar nuestro nuevo fichero *rnn.h y rnn.c*.

## 4.3 Parsing

Cada capa tiene una función para «parsear» la capa en el fichero de definición de la red neuronal. En el código 4.1 hemos visto cómo definir la red neuronal, y ahora, queremos definir la «RNN» al igual que todas las demás capas.

Nuestra red neuronal recurrente requiere la definición de unos parámetros con los cuales vamos a poder definir el resto. Estos son en orden: el nombre de la capa, el tipo de red, el tamaño de entrada *input_size*, el tamaño intermedio *hidden_size*, el número de celdas *rnn_cell_count*, y por último, la función de activación *rnn_activation_function* que puede ser o *TANH* o *RELU*.

En el fichero del modelo de la red neuronal se vería tal que así:

```
rnn1 rnn 28 64 28 tanh
```

**Código 4.11:** Definir tipos de capas en HELENNA

En el código 4.12 vemos cómo recoger los parámetros a partir del modelo 4.11. Primero separamos cada parámetros en un «array» a partir de un delimitador, en este caso, cada parámetro está separado por un espacio. Después, transformamos cada uno a su tipo correspondiente. En el caso de tamaño de entrada, tamaño de salida y el número de celdas lo pasamos a *int* y la función de activación, nos aseguramos de qué tipo es.

Por último, cada capa dispone de un número de neuronas. También se puede entender como el número de salidas que tendremos y que en nuestro caso, tiene más sentido. Por lo tanto, nuestra red neuronal tendrá un número de neuronas igual al tamaño de estado intermedio por el número de celdas que haya (*hidden_state* $\times$ *rnn_cell_count*).

```
void rnn_parse_tp(int layer, char *parameters)
{
    char splitted_parameters[RNN_NUM_PARAMETERS][10];
    char *delimeter = " ";
```

```
    char *token = strtok(parameters, delimeter);
    for (int i = 0; i < RNN_NUM_PARAMETERS && token != NULL; i++)
    {
        strcpy(splitted_parameters[i], token);
        token = strtok(NULL, delimeter);
    }
    input_size = atoi(splitted_parameters[0]);
    hidden_size = atoi(splitted_parameters[1]);
    rnn_cell_count = atoi(splitted_parameters[2]);
    char *af = splitted_parameters[3];
    if (strcmp(af, "relu") == 0)
    {
        rnn_parameters.activation_function = RELU;
    } else {
        rnn_parameters.activation_function = TANH;
    }
    layers_ptr[layer]->previous_layer = layer - 1;
    layers_ptr[layer]->num_neurons = hidden_size * rnn_cell_count;

}
```

**Código 4.12:** Implentación parseo red neuronal recurrente

## 4.4 Estructuras de Datos

En las «RNN» representamos las diferentes iteraciones con una abstracción que llamaremos celdas y que, implementaremos con estructuras nativas de *C*. Además de las celdas, vamos a encapsular los *buffer* temporales, es decir, *buffer* intermedios que usaremos para guardar resultados. También vamos a encapsular los *buffer* relacionados con los parámetros globales de la «RNN». Y para acabar tenemos una última estructura para los gradientes que calcularemos en el «backwarding».

Para empezar, la estructura de la celda está implementada como se puede observar en el código 4.13, haciendo uso de 3 *buffer*. Recordemos que los *buffer* los indexamos por *ints*. Tenemos el *buffer* del estado intermedio anterior *mid_h_prev*, el del estado intermedio que vamos a calcular en esta celda *mid_h_next* y la entrada *mid_x*. Como en nuestra implementación la salida es equivalente al estado intermedio calculado, no vamos a requerir un último *buffer* de salida.

```
typedef struct _rnn_cell_td {
    int mid_h_next;
    int mid_h_prev;
    int mid_x;
} rnn_cell;
```

**Código 4.13:** Estrucutra de una celda

En cuanto a las variables temporales, en el código A.3, podemos observar la larga lista de *buffer* intermedios. Estos los usaremos y veremos con más atención en los siguientes apartados.

Los parámetros de la «RNN» correspondientes a las matrices de pesos y a los «bias» los tenemos definidos en la estructura *rnn_cell_parameters* que podemos ver en el código 4.14.

```
typedef struct _rnn_cell_parameters {
    int mid_Whh;
    int mid_Wxh;
    int mid_bias_x;
```

```
    int mid_bias_h;
    t_rnn_activation_function activation_function;
} rnn_cell_parameters;
```

**Código 4.14:** Parametros globales de la RNN

En último lugar, tenemos los gradientes. Los gradientes también vamos a guardarlos por separado en su propia estructura que observamos en el código 4.15.

```
typedef struct _gradients {
    int mid_dWxh;
    int mid_dWhh;
    int mid_dba;
    int mid_dbh;
} _rnn_gradients;
```

**Código 4.15:** Definición de gradientes de la RNN

## 4.5 Ubicación y desubicación de Buffers

En *C* se requiere hacer asignar y desasignar espacio en memoria para los *buffer* que vamos a usar. En *HELENNA*, como hemos mencionado antes, cada capa tiene una función para ello. Primero vamos a ver cómo implementamos *rnn_allocate_tp* (asignación) y después *rnn_deallocate_tp* (desasignación).

Para empezar, la función *rnn_allocate_tp* llama a las funciones que se encargan de asignar los *buffer* de cada estructura y a la función *fn_base_layer_allocate*, que se encarga de asignar las variables usadas por la capa como podría ser la entrada o la salida. En el código A.4 podemos ver con más detalle que está pasando.

El *batch_size*, como ha sido explicado anteriormente, nos sirve para hacer los cálculos con más de una entrada. Para representar el *batch_size* en nuestro código vamos a añadir a los *buffer* que lo requieran una dimensión mas. Por ejemplo, a la entrada de la celda su dimensión pasará de $input\_size \times 1$ a $input\_size \times minibatch\_size$ dónde *minibatch_size* representa el *batch_size*. También hay que tener en cuenta que *type* es el tipo de los valores que usaremos y, que en este caso va a ser siempre un *float*.

El caso de asignación de celdas es un poquito mas especial. Añadimos una celda de más porqué necesitamos guardar el último estado intermedio en algún sitio. Para optimizar el espacio, el estado intermedio calculado en la celda $t$ es igual al estado intermedio anterior de la celda posterior. Además, no hacemos el *allocate* del *buffer* de entrada en la última celda ya que esta no es una celda real.

Por otro lado, *rnn_deallocate_tp* es más simple. Tenemos como antes una función para deshacer cada estructura y la capa base.

## 4.6 Inicialización

Ya visto la asignación de *buffer*, vamos a ver como inicializar algunos de estos que van a necesitar un valor antes de empezar.

Para empezar, hay una matriz inicializada a unos, que vamos a usar para la derivada de la tanh en el «backwarding», y por lo tanto, la vamos a inicializar en esta función (*rnn_initialize_tp*). Solo vamos a necesitar una función como vemos en el código 4.16. Esta función a partir de la dimensiones ($hidden\_size \times minibatch\_size$) y una probabilidad

(1,0), genera unos en esa matriz. En nuestro caso queremos que sea una matriz completa de unos así que, ponemos una probabilidad de 1.

```
fn_matset_random_ones(tmp_variables.mid_ones_matrix, hidden_size,
    minibatch_size, 1.0);
```

**Código 4.16:** Inicialización matriz de unos

Por otro lado, vamos a inicializar los parámetros de la «RNN». Hay mas de una forma de inicializar los parámetros: inicialización a cero, inicialización a números aleatorios, inicialización de *He* y inicialización de *Xavier*. Estas dos últimas son mas modernas y nos ayuda en dos sentidos: ayuda a que empiece a aprender antes y, ayuda con el problema de «vanishing» y «exploding» gradients, que en «RNNs» esta muy presente.

El estándar para las funciones *tanh* y la *sigmoid* es la inicialización por *Xavier*. Suponiendo que *fan_in* es el número de unidades de la capa anterior, *fan_out* el número de unidades de la capa actual y *r* el rango que los valores que tendrán los parámetros, tenemos que la función es:

$$r = \sqrt{\frac{1}{fan\_in + fan\_out}}$$

Para la función *ReLU* se encontró que había problemas de convergencia con la inicialización de *Xavier* así que, con este tipo de funciones usamos la inicialización *He* que solo cambia en el *1* que pasa a ser un *2*:

$$r = \sqrt{\frac{2}{fan\_in + fan\_out}}$$

En el código 4.17 podemos ver cómo inicializamos los parámetros. Primero en *get_parameter_initialization_range* dependiendo de la función, calculamos el rango. Después, con este rango inicializamos los *buffer* con *fn_vect_random_float* que, a partir de las dimensiones y un rango de valores, inicializa aleatoriamente el *buffer* con ese rango de valores.

```
void rnn_initialize_parameters(int layer) {
    float range = get_parameter_initialization_range(layer);
    fn_vect_random_float(rnn_parameters.mid_Whh, (hidden_size * hidden_size), -
        range, range);
    fn_vect_random_float(rnn_parameters.mid_Wxh, (hidden_size * input_size), -
        range, range);
    fn_vect_random_float(rnn_parameters.mid_bias_x, hidden_size, -range, range)
        ;
    fn_vect_random_float(rnn_parameters.mid_bias_h, hidden_size, -range, range)
        ;
}

float get_parameter_initialization_range(int layer) {
    int fan_in = layers_ptr[layer - 1]->num_neurons;
    int fan_out = layers_ptr[layer]->num_neurons;
    float range;
    if(rnn_parameters.activation_function == TANH) {
        range = sqrt((1 / (fan_in + fan_out)));
    }
    else {
        range = sqrt((2 / (fan_in + fan_out)));
    }
    return range;
}
```

**Código 4.17:** Inicialización de parámetros con Xavier y He

## 4.7 Forwarding

Después de toda la inicialización, podemos proceder al corazón de la «RNN». En la función *rnn_forward* y *rnn_forward_cell* vamos a realizar una gran parte de los cálculos que esta capa realiza. En la Figura 3.1 se puede ver con mas claridad del proceso de «forwarding» de la *RNN* y, tiene relación con *rnn_forward*. Por otra parte en la Figura 3.4 tiene mas relación con *rnn_forward_cell* donde se muestra las operaciones que realiza la celda.

Vamos a empezar por la función *rnn_forward*. Esta es la encargada de llamar al «forwarding» de cada celda, preparando sus entradas antes. Como podemos ver en el código 4.18, primero inicializamos el estado intermedio de la primera celda a 0. Después, por cada celda $t$, copiamos la parte correspondiente de la salida de la capa anterior, a la celda correspondiente. Por lo tanto, la salida de la capa anterior será siempre del tamaño $input\_size \times minibatch\_size \times rnn\_cell\_count$ y la entrada de cada salida será $input\_size \times minibatch\_size$ con un «offset» de $rnn\_cell\_count$. Esto quiere decir que la dirección de memoria de la entrada de cada celda es $(input\_size \times minibatch\_size) + rnn\_cell\_count$.

```
1  void rnn_forward(int layer, int fbe, int lbe, int training)
2  {
3      int mid_y = layers_ptr[layer - 1]->mid_y;
4      fn_zero_vec(rnn_cells[0].mid_h_prev, hidden_size * minibatch_size);
5      for (int t = 0; t < rnn_cell_count; t++)
6      {
7          fn_copy(rnn_cells[t].mid_x, mid_y, 0, t * input_size * minibatch_size,
               input_size * minibatch_size);
8          rnn_forward_cell(t);
9      }
10     get_outputs(layer);
11     #ifdef DEBUG
12     rnn_print_outputs(layer);
13     #endif
14 }
```

**Código 4.18:** Código de rnn_forward

Al final de la ejecución de todas las celdas, procedemos a llamar a *get_outputs* que básicamente copia la salida de esta capa a la siguiente. Además, como veremos en este tipo de funciones, siempre tenemos el caso de, si tenemos definido *DEBUG*, imprima las estadísticas de los *buffer* para facilitar el proceso de depuración de la capa.

Con los pasos de preparación previos ya nos adentramos a la función *rnn_forward_cell* en la que realizaremos la formula para calcular el estado intermedio:

$$h_t = activation\_function((W_{hx} \times x_t + B_x) + (W_{hh} \times h_{t-1} + B_h))$$

En el código A.4 podemos ver la función completa, ya que vamos a ir por secciones de ella a continuación.

Para empezar, simplemente creamos las variables de todos los *buffer* que vamos a utilizar:

```
1  int x = rnn_cells[t].mid_x;
2  int h_prev = rnn_cells[t].mid_h_prev;
```

```
int h_next = rnn_cells[t + 1].mid_h_prev;
int W_hh = rnn_parameters.mid_Whh;
int W_hx = rnn_parameters.mid_Wxh;
int mid_bias_x = rnn_parameters.mid_bias_x;
int mid_bias_h = rnn_parameters.mid_bias_h;
//tmp
int multxw = tmp_variables.multxw;
int multhw = tmp_variables.multhw;
int sum_mults = tmp_variables.sum_mults;
int sum_bias_x = tmp_variables.sum_bias_x;
int sum_bias_h = tmp_variables.sum_bias_x;
```

**Código 4.19:** Variables de rnn_forward_cell

Ahora procedemos a hacer las multiplicaciones $W_{hx} \times x_t$ y $W_{hh} \times h_t$, respectivamente haciendo uso de la función *fn_matmul* como se puede ver en el código 4.20. *fn_matmul*, como se puede observar en la cabecera A.1, requiere primero del primer *buffer* y a continuación sus dimensiones ($W_{hx} = hidden\_size \times input\_size$), después el segundo parámetro ($x_t = input\_size \times minibatch\_size$), el *buffer* donde guardamos el resultado, el número de columnas resultantes y por último el *offset* que en este caso es 0. La segunda llamada a *fn_matmul* es equivalente pero para la multiplicación $W_{hh} \times h_t$. Los resultados los guardaremos en *multxw* para la primera multiplicación y *multhw* para la segunda.

```
fn_matmul(W_hx, hidden_size, input_size, x, input_size, minibatch_size, multxw,
    1, 0);
fn_matmul(W_hh, hidden_size, hidden_size, h_prev, hidden_size, minibatch_size,
    multhw, 1, 0);
```

**Código 4.20:** Multiplicaciones matriciales con los pesos

Ahora pasamos a 3 sumas consecutivas, la primera siendo la suma de las dos multiplicaciones anteriores, y las dos últimas para los dos «bias» que tenemos. Para estas operaciones utilizaremos *fn_matadd* que se parece mucho a la multiplicación pero sin los dos últimos argumentos. Por lo tanto, tendremos algo parecido a $sum\_mults = multhw + multxw$, $sum\_bias\_x = sum\_mults + mid\_bias\_x$ y para acabar $sum\_bias\_h = sum\_bias\_x + mid\_bias\_h$:

```
fn_matadd(multhw, hidden_size, minibatch_size, multxw, hidden_size,
    minibatch_size, sum_mults);
fn_matadd(sum_mults, hidden_size, minibatch_size, mid_bias_x, hidden_size,
    minibatch_size, sum_bias_x);
fn_matadd(sum_bias_x, hidden_size, minibatch_size, mid_bias_h, hidden_size,
    minibatch_size, sum_bias_h);
```

**Código 4.21:** Sumas matriciales con los bias

Para acabar, solo nos queda aplicarle la función de activación correspondiente a *sum_bias_h* que sería equivalente a $z$ que habíamos visto en el capítulo anterior. En el código 4.22 vemos como dependiendo de la función de activación llamamos a *fn_tanh* o *fn_matrix_relu*:

```
if (rnn_parameters.activation_function == TANH)
{
  fn_tanh(sum_bias_h, h_next, hidden_size * minibatch_size);
}
else
{
  fn_matrix_relu(sum_bias_h, h_next, hidden_size, minibatch_size);
}
```

**Código 4.22:** Implementación de la función de activación

El resultado lo guardamos en *h_next* que, como podemos ver en 4.19, es básicamente *mid_h_prev* de la siguiente capa. Y por eso tenemos una celda extra simplemente para contener el último resultado.

Con ya todos los resultados lo único que nos queda es transferirlos a la siguiente capa. En la función 4.23 podemos ver cómo simplemente copiamos los *buffer* de los estados intermedios desde la segunda celda hasta la capa extra.

```
void get_outputs(int layer)
{
    for (int t = 1; t <= rnn_cell_count; t++)
    {
        fn_copy(layers_ptr[layer]->mid_y, rnn_cells[t].mid_h_prev, (t - 1) *
            hidden_size * minibatch_size, 0, hidden_size * minibatch_size);
    }
}
```

**Código 4.23:** Transferencia de estados intermedios a la salida

## 4.8 Backpropagation through Time

Lo último que nos queda por implementar de la «RNN» es la función de *rnn_backward_tp*. Con ella, concluiremos todo el desarrollo y solo quedaría validar y comprobar. En esta función vamos a seguir un desarrollo parecido al anterior. Vamos a tener la función principal *rnn_backward_tp* y después tendremos *rnn_backward_cell*.

Vamos a empezar como antes con la principal. Para empezar, vamos a reiniciar los gradientes para que no se mantengan entre ejecuciones con *rnn_gradients_to_zero*. Como podemos ver en A.4, poner los *buffer* a 0 solo requiere llamar a la función *fn_zero_vect*. Después también vamos a poner a 0 *mid_dh* que es básicamente $\frac{\partial h_t}{\partial h_{t-1}}$ pero al empezar por la última celda, todavía no tendrá un valor.

Luego, vamos a recorrer cada celda hacía atrás, como se ve en el bucle, y antes de llamar a la función de «backward» copiaremos la pérdida dada la celda actual como vimos en el capítulo anterior $\frac{\partial L}{\partial h_t}$. La pérdida la tenemos en *mid_y_err* de la capa actual y que previamente, la capa posterior la había calculado. Esta pérdida la guardaremos en *mid_dl_dh*:

```
void rnn_backward_tp(int layer, int minibatch_size)
{
    rnn_gradients_to_zero();
    int y_err = layers_ptr[layer]->mid_y_err;
    fn_zero_vec(tmp_variables.mid_dh, hidden_size * minibatch_size);
    for (int t = rnn_cell_count - 1; t >= 0; t--)
    {
        fn_copy(tmp_variables.mid_dl_dh, y_err, 0, t * hidden_size *
            minibatch_size, hidden_size * minibatch_size);
        rnn_cell_backward_tp(t);
    }
    rnn_update_parameters();
}
```

**Código 4.24:** Implementación función backward de la RNN

Ahora dentro de la función de *rnn_cell_backward_tp*, que ejecutará cada celda, vamos a calcular los gradientes. Para empezar vamos a sumar la pérdida de la salida dada el estado intermedio $\frac{\partial L}{\partial h_t}$ con la pérdida del estado intermedio dado el estado intermedio anterior $\frac{\partial h_t}{\partial h_{t-1}}$ que guardaremos en *dh* y que en pseudocódigo sería $dh = dh + dl\_dh$:

```
fn_vec_add(dl_dh, dh, dh, hidden_size * minibatch_size);
```

**Código 4.25:** pérdida de la celda

Con la pérdida de la celda ya podemos calcular la pérdida de la función de activación, es decir, $\frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial z_t}$ donde $z_t$ es igual al resultado antes de la función de activación. Como tenemos dos funciones de activación diferentes, tenemos que implementar la derivada de las dos.

En el caso de tanh, su derivada es igual a $1 - h_t^2$ y la de la «ReLU» es 0 si $x < 0$ y 1 si $x > 0$. Para la primera no tenemos una función específica así que la implementaremos directamente. Para la segunda si que disponemos de una ya implementada. Como podemos ver en el código 4.26, para la derivada de tanh, primero vamos a elevar a cuadrado $h_t$ y después, vamos a utilizar la matriz de unos que habíamos inicializado al principio para hacer la resta. Para la derivada de la función «ReLU» utilizaremos la función *fn_matrix_relu_der* que ya se ocupa de hacer todo lo necesario. Al acabar este bloque de código vamos a disponer de *df* que es la derivada de la función dada la pérdida.

```
if (rnn_parameters.activation_function == TANH)
{
  // tanh der dL/dh * (1 - h^2) -> element mult
  fn_square(h_next, square_a, hidden_size * minibatch_size);
  fn_matsub(ones_matrix, hidden_size, minibatch_size, square_a, hidden_size,
      minibatch_size, df);
}
else
{
  fn_matrix_relu_der(h_next, hidden_size, minibatch_size, df);
}
```

**Código 4.26:** Implmentación de derivadas en la RNN

Con la derivada de la función y con la fórmula que hemos definido antes: $\frac{\partial L}{\partial h_t} \cdot \frac{\partial h_t}{\partial z_t}$ donde $z_t$ es el resultado de la función de activación, podemos calcular el resultado de ésta con la multiplicación de los dos vectores que guardaremos en *df_dh_next*:

```
fn_vect_mult(dh, df, df_dh_next, hidden_size * minibatch_size);
```

Con este último resultado ya podemos realizar el cálculo de los gradientes de cada parámetro. En el caso de los «bias», $B_h$ y $B_x$ van a ser iguales a *df_dh_next* ya que la derivada de $x$ es igual a 1.

En cuanto a las matrices, tenemos $W_{hh} \times h_{t-1}$ y la derivada parcial $\frac{\partial(W_{hh} \times h_{t-1})}{\partial W_{hh}}$ será igual a $h_t$. Con la misma lógica vamos a calcular la derivada de $W_{hx} \times x_t$, que será $x_t$. Con estos resultados, ya podemos multiplicar ambos por *df_dh_next*. En ambos casos, vamos a tener que transponer $h_{t-1}$ y $x_t$ para que las dimensiones sean correctas para la multiplicación y por lo tanto, nos quedaría algo como $df\_dh\_next \times h_{t-1}^T$ y $df\_dh\_next \times h_{t-1}^T$ que se traduce en el código 4.27:

```
fn_matmul_bt(df_dh_next, hidden_size, minibatch_size, xt, input_size,
    minibatch_size, tmp_variables.mid_dWxh,
   input_size, 0);
fn_matmul_bt(df_dh_next, hidden_size, minibatch_size, h_prev, hidden_size,
    minibatch_size, tmp_variables.mid_dWhh,
   hidden_size, 0);
```

**Código 4.27:** Implementación de calculo de gradientes de las matrices

*fn_matmul_bt* es la multiplicación $a \times b$ con $b$ transpuesta. También tenemos el caso contrario *fn_matmul_at* donde $a$ es transpuesta y que vamos a usar para calcular el gradiente del estado intermedio anterior. Para el estado intermedio anterior volvemos al cálculo $W_{hh} \times h_{t-1}$, pero en este caso la derivada será $W_{hh}$ y por lo tanto, el gradiente será $df\_dh\_next^T \times W_{hh}$:

```
fn_matmul_at(df_dh_next, hidden_size, minibatch_size, Whh, hidden_size,
    hidden_size, minibatch_size, 0, dh, 0);
```

**Código 4.28:** Implementación de calculo del gradiente del estado intermedio anterior

Estamos a punto de acabar. Hemos calculado todos los gradientes necesarios, incluso el del estado intermedio anterior que nos será de utilidad para la celda anterior ya que requiere de la pérdida dada la salida y dado el estado intermedio calculado. Lo único que nos queda para acabar el «backwarding» de la celda es acumular los gradientes. Anteriormente, mencionamos que en el *Backpropagation through time* los parámetros son compartidos y por lo tanto no vamos a tener todos los gradientes hasta que lleguemos a la primera celda así que vamos a acumular los gradientes y los guardaremos en la estructura creada para ello:

```
fn_matadd(gradients.mid_dWhh, hidden_size, hidden_size, tmp_variables.mid_dWhh,
        hidden_size, hidden_size, gradients.mid_dWhh);
fn_matadd(gradients.mid_dWxh, hidden_size, input_size, tmp_variables.mid_dWxh,
        hidden_size, input_size, gradients.mid_dWxh);
fn_matadd(gradients.mid_dba, hidden_size, 1, df_dh_next, hidden_size, 1,
    gradients.mid_dba);
fn_matadd(gradients.mid_dbh, hidden_size, 1, df_dh_next, hidden_size, 1,
    gradients.mid_dbh);
```

**Código 4.29:** Implementación de calculo del gradiente del estado intermedio anterior

Al fin tenemos todo lo necesario para actualizar los parámetros. En *HELENNA* tenemos una función que se encarga de esto llamada *fn_SGD* en la cual hemos añadido en el caso de una «RNN» que actualice todos los parámetros con los gradientes calculados. Como podemos ver en el código 4.30, para cada parámetro seguimos la formula de *gradient descent* $\theta = \theta - lr * \nabla_\theta \frac{\partial L}{\partial w_1}$.

```
fn_vect_scalar_product(learning_rate, gradients.mid_dWhh, hidden_size *
    hidden_size);
fn_vect_scalar_product(learning_rate, gradients.mid_dWxh, hidden_size *
    input_size);
fn_vect_scalar_product(learning_rate, gradients.mid_dba, hidden_size);
fn_vect_scalar_product(learning_rate, gradients.mid_dbh, hidden_size);
fn_matsub(rnn_parameters.mid_Whh, hidden_size, hidden_size, gradients.mid_dWhh,
  hidden_size, hidden_size, rnn_parameters.mid_Whh);
fn_matsub(rnn_parameters.mid_Wxh, hidden_size, input_size, gradients.mid_dWxh,
  hidden_size, input_size, rnn_parameters.mid_Wxh);
fn_matsub(rnn_parameters.mid_bias_x, hidden_size, 1, gradients.mid_dba,
  hidden_size, 1, rnn_parameters.mid_bias_x);
fn_matsub(rnn_parameters.mid_bias_h, hidden_size, 1, gradients.mid_dbh,
  hidden_size, 1, rnn_parameters.mid_bias_h);
```

**Código 4.30:** Actualización de parámetros de la RNN

## 4.9 Extendiendo la creación del modelo

Una red neuronal no es nada si no puede ser guardada después de entrenarse, y como era de esperar, en *HELENNA* tenemos una forma de guardar los modelos. Al añadir la

«RNN» hemos creado la necesidad de extender la creación y carga de un fichero en el que se guarde el modelo, y nos vamos a ocupar de ello ahora.

En el fichero *model.c* disponemos de toda la lógica relacionado con carga y creación de ficheros de modelos. En particular tenemos dos funciones en específico: *fn_save_model* que se encarga de escribir los *buffer* de los parámetros que necesitamos guardar, y que después al cargar con la función *fn_load_model* podamos recrear la red neuronal guardada.

En el caso de la «RNN», necesitamos guardar las matrices de peso y los «bias». Para ello, hemos extendido estas funciones para las capas «RNN» y hemos creado algunas funciones para hacer el proceso más limpio y con errores mas descriptivos en caso de fallo.

En el código A.6 hay dos funciones que destacan para nuestra capa: *load_rnn_parameters* y *save_rnn_parameters*. Estas se encargan de llamar a *try_fread* y *try_fwrite* que invocan las funciones que tenemos para leer y escribir *buffers*: *fread* y *fwrite*. En caso de que una de estas funciones falle, se aborta la ejecución con un mensaje de error prefijado por la capa.

# CAPÍTULO 5
# Pruebas/Evaluación

La implementación de la *RNN* en *HELENNA* no puede darse como acabada sin comprobar que funcione correctamente. En este capítulo vamos a validar que funciona probando diferentes hyperparametros, cambiando dimensiones de la *RNN* y comparandolo con *PyTorch* con la misma definición de modelo.

Probar un modelo no es una ciencia exacta dado el hecho de que trabajamos con probabilidades constantemente y como barajamos el *dataset* antes de entrenar, vamos a ver diferentes resultados en diferentes ejecuciones.

El entrenamiento tanto en *HELENNA* como en *PyTorch* separa todo el dataset en dos partes, el *dataset* de entrenamiento y el *dataset* de validación. En este capítulo siempre vamos a tener en cuenta la precisión de este último que nos sirve para comprobar que el entrenamiento ha funcionado.

El *dataset* que vamos a utilizar es el de *MNIST*. Cada ejemplo del *dataset* es una imagen y como la *RNN* usa datos secuenciales, lo que vamos a hacer es separar la imagen en filas para cada celda.

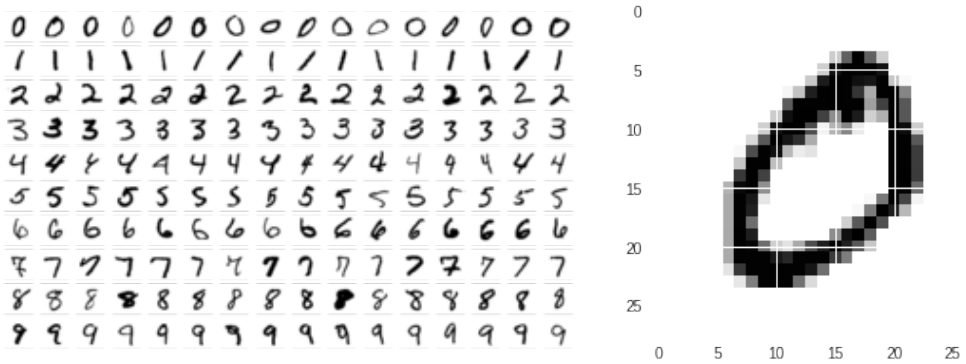
## 5.1 MNIST

*MNIST* es una base de datos de dígitos escritos a mano. Este *dataset* está compuesto de 60,000 imágenes para entrenar y 10,000 imágenes para testear. Las dimensiones de estas imágenes son de $28 \times 28$ píxeles, en total 784.

Este *dataset* se utiliza ampliamente para probar modelos nuevos como es nuestro caso, porque no es demasiado grande y no necesitamos crear modelos muy complejos para que el modelo aprenda a predecir los dígitos escritos. Este *dataset* tiene 10 clases que comprenden lo dígitos desde 0 hasta 9.

Las *RNN* tratan con datos secuenciales y *MNIST* no es precisamente un tipo de secuencia de datos esperados pero, *MNIST* es usado constantemente para la realización de pruebas en nuevas arquitecturas de Redes Neuronales y también lo usan mas personas para probar las *RNN* [26]. Normalmente, trabajamos con datos que evolucionan con el tiempo, *NLP*, vídeos etc... En este caso, vamos a dividir cada imagen del *dataset* de *MNIST* en 28 filas de 28 píxeles. Por lo tanto, a cada celda le enviaremos una fila para que al final, con la combinación de cada celda y sus filas, sea capaz de descifrar el dígito.

El modelo que vamos a utilizar para resolver este problema es simple. Vamos a tener una primera capa que será una *RNN* con 28 celdas (*rnn_cell_count*), las cuales recibirán 28 píxeles (*input_size*) y tendremos un *hidden_size* elegido arbitrariamente de 64. Después tendremos una capa densa de 10 unidades para la probabilidad de que sea cada dígito y



(a) Grupo de dígitos de MNIST (b) Dígito 0 en MNIST

**Figura 5.1:** Ejemplos de dígitos de MNIST

para acabar, una *ReLU* seguida de una *Softmax*. Vamos a ver mejor la definición de este en el código 5.1:

```
input fc 784
rnn1 rnn 28 64 28 tanh
fc2 fc 10
relu1 relu
softmax softmax
```

**Código 5.1:** Modelo MNIST con una RNN

## 5.2 Probando diferentes Hyperparámetros

Los hyperparámetros se diferencian de los parámetros en que no dependen de los datos provisionados ni se actualizan en la ejecución. Estos se especifican al principio del aprendizaje y no existe un estándar para ver cual es el mejor hyperparámetro para cada modelo.

La optimización de hyperparámetros es muy común para encontrar un conjunto de valores optimo para nuestro problema. Hay diferentes formas de optimizar los hyperparámetros:

- Búsqueda manual: De forma manual, gracias al juicio propio y experiencia, escogemos los hyperparámetros que creemos que van a ser óptimos.
- Búsqueda aleatoria: Escogemos los valores de los hyperparámetros aleatoriamente.
- Búsqueda en red: Definimos una red de valores que deberemos probar con todas las permutaciones posibles.
- Optimizacición Bayesiana: La optimización Bayesiana usa la probabilidad para encontrar el mínimo de una función a partir de un valor de entrada. Esta optimización requiere de tres parámetros: la función de pérdida que queremos optimizar, el rango de valores que queremos probar y, el algoritmo de búsqueda para encontrar valores para la próxima iteración. Lo bueno de esta búsqueda de hyperparámetros automática es que reduce el número de iteraciones para encontrar una buena solución.
- Algorítmos genéticos: Este tipo de algoritmo intenta implementar la selección natural en el contexto de inteligencia artificial. Lo que se busca con este algoritmo es

crear diferentes modelos con diferentes valores de hyperparámetros y al final de la ejecución, se desecharían los modelos que peor se han desenvuelto.

En nuestro caso vamos a ir con la búsqueda manual ya que no estamos intentando resolver un problema muy complejo y por lo tanto, vamos a escoger unos valores típicos que suelen funcionar. En el caso del *learning rate* tendremos un rango de valores de [1,0.1,0.01,0.001,0.0001] y para el *batch size* comprobaremos los valores $[1, 2, 4, 8, 16, 32, 64, 128]$.

Para realizar las pruebas con estos dos hyperparámetros vamos a recurrir a un *bash script* simple que recorre la lista de los valores y ejecuta *HELENNA* con los valores esperados 10 veces para poder sacar un promedio medio del resultado:

```
rnn() {
    OMP_NUM_THREADS=5 ./nnsim -dataset datasets/mnist -net nets/rnn/mnist -ne
        $epochs -lf ce -mbs $batch -lr $lr
}
lr=0.01
for batch in 1 2 4 8 16 32 64 128
do
    for iter in {1..10}
    do
  rnn
    done
done

batch=64
for lr in 1 0.1 0.01 0.001
do
    for iter in {1..10}
    do
        rnn
    done
done
```

**Código 5.2:** Script test hyperparámetros

Como podemos ver tenemos un valor por defecto para el *lr* y *batch* que sabemos que funciona bien para probar el otro hyperparámetro. Además, el número de épocas que vamos a utilizar para todas estas pruebas es 3. Este número ha sido escogido porque la mayoría de modelos de *MNIST* con 3 épocas se obtiene un resultado bueno.

### 5.2.1. Batch Size

Vamos a empezar por el *batch size*. Como ya hemos visto, el *batch size* denota la cantidad de ejemplos que vamos a usar para entrenar nuestra red neuronal en una iteración. Esto quiere decir que, cuanto mayor sea el *batch size*, más valores se tendrán en cuenta para actualizar los parámetros y por lo tanto, es mas probable que aprenda mejor. Sin embargo, a mayor volumen de ejemplos, mayor computación será necesaria ya que estaríamos realizando operaciones con matrices mucho mas grandes en dimensiones.

Vamos a ver como los valores $[1, 2, 4, 8, 16, 32, 64, 128]$ del *batch size* pueden afectar a la ejecución de nuestro modelo de *MNIST* con una RNN.

Después de ejecutar el script 5.2 hemos recolectado los resultados y hemos podido crear una gráfica para mostrar la evolución de las salidas:

Como podemos ver, con un *batch size* de 1 o 2, el coste(cuanto mas cerca de 0 mejor) y la precisión (cuanto mas cerca de 100 mejor) es significativamente peor que en los otros

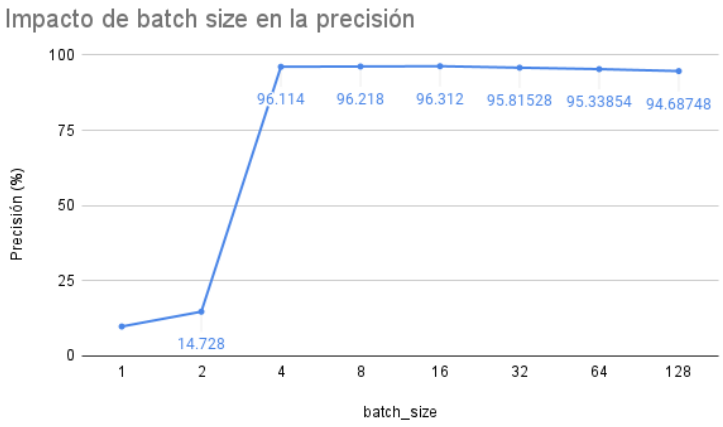



**Figura 5.2:** Precisión dado diferentes batch size

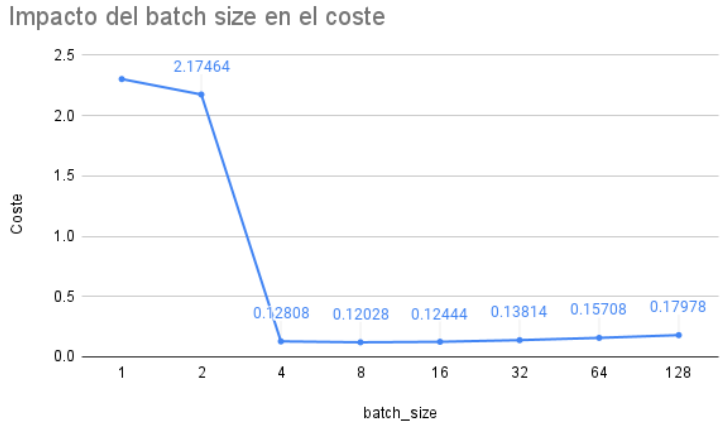



**Figura 5.3:** Coste dado diferentes batch size

casos. Esto se debe a que solo usamos 1 o 2 ejemplos de referencia para actualizar los parámetros.

A partir de un *batch size* de 4 empezamos a ver mejores resultados que no bajan de un 95 % de precisión y no sube de un coste de 0,2808.

### 5.2.2. Learning Rate

El learning rate controla lo rápido que queremos que nuestro modelo aprenda. Si proporcionamos un valor muy alto, puede hacer que lleguemos a un valor subóptimo y por otro lado, un valor muy pequeño puede hacer que se tarde mucho tiempo en converger a una solución óptima y por lo tanto, que se quede estancado en un óptimo local.

Como vamos a ver ahora con diferentes *lr* vamos a obtener diferentes resultados. Hemos elegido valores grandes y pequeños para demostrar que elegir un «learning rate» adecuado es muy importante como vamos a ver a continuación en la Figura 5.4 y 5.5:

Como podemos observar, con un «learning rate» de 1 y 0.1 obtenemos una precisión muy pobre, menor de un 20 %. Por otra parte, si bajamos a 0,01 tenemos un coste y precisión excelentes, 0,165 y 94 % respectivamente. A partir de 0,01 empezamos a ver como la precisión y el coste empieza a empeorar. Con 0,001 la precisión y el coste empeora un

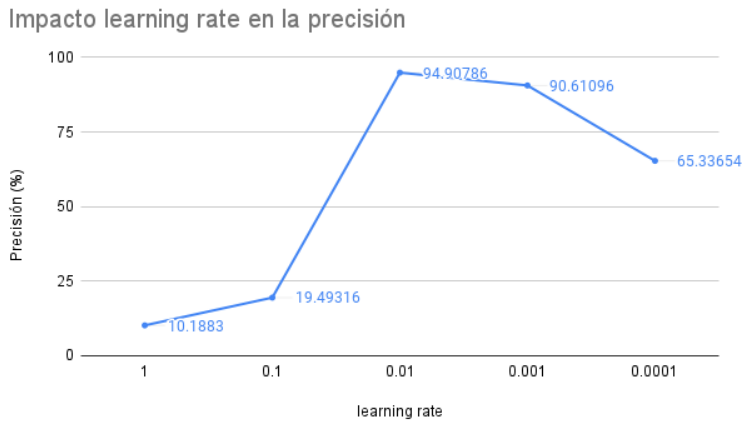



**Figura 5.4:** Precisión dado diferentes learning rate

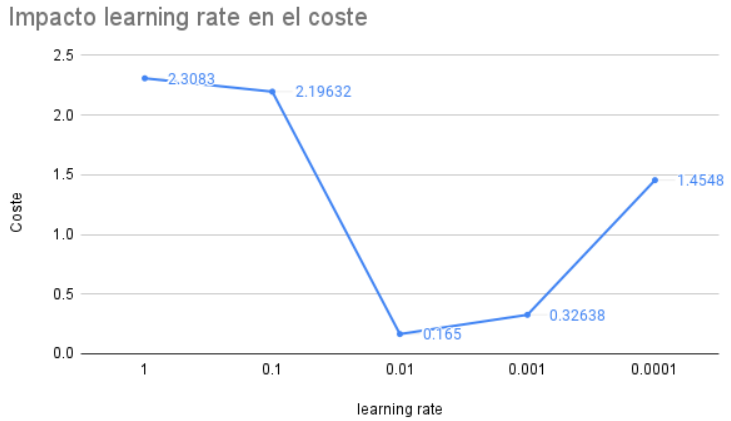



**Figura 5.5:** Coste dado diferentes learning rate

poquito pero en 0,0001 ya vemos una pérdida mucho mayor de 1,45 y una precisión de 65 %.

Estos resultados son un ejemplo claro de que no hay que escoger valores muy altos o muy pequeños. Además queda reflejado que por ahora, nuestra implementación de la *RNN* se está comportando de forma esperada.

## 5.3 Probando diferentes dimensiones

Ahora vamos a ver cómo puede afectar los tamaños del *input_size* y *hidden_size* a la precisión y el tiempo de ejecución de nuestra *RNN*.

### 5.3.1. Hidden Size

El tamaño del estado intermedio es una variable muy importante en el entrenamiento de nuestro modelo, ya que números muy bajos pueden causar pérdida de información que conllevarán un lento aprendizaje, pero, también puede disminuir considerablemente el tiempo de ejecución de la misma. Un valor muy alto puede hacer que tarde demasiado tiempo y además que se pierda información que no hará volver a tener una mala precisión, como se muestra en las Figuras 5.7 y 5.6.

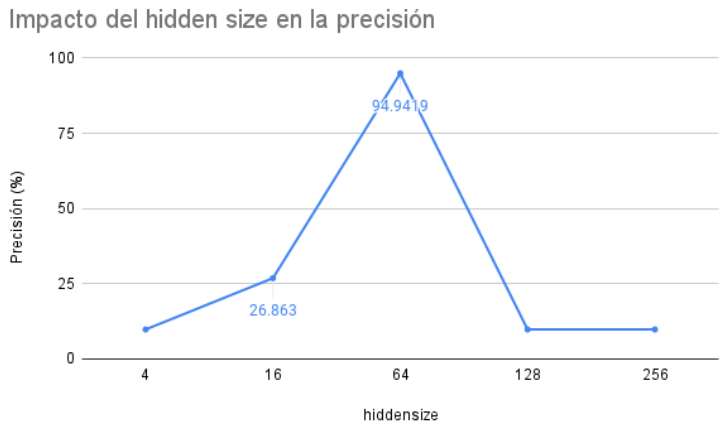



**Figura 5.6:** Impacto del hidden size en la precisión

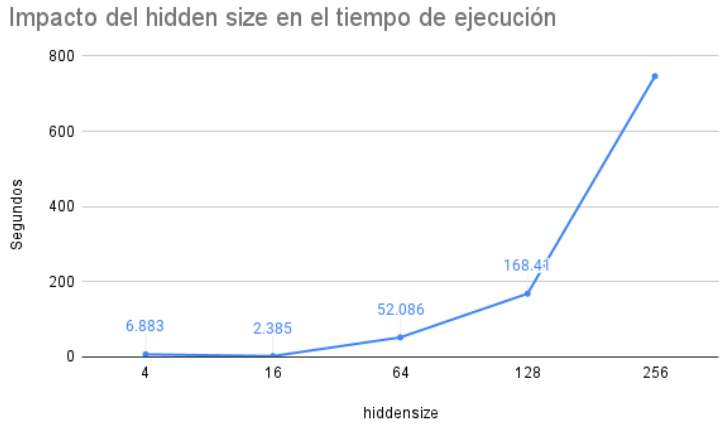



**Figura 5.7:** Impacto del hidden size en el tiempo de ejecución

Como podemos observar el *hidden_size* tiene un efecto muy significativo en el tiempo de ejecución y en la precisión. Con un *hidden size* muy pequeño podemos ver que el tiempo de ejecución es bajo y la precisión también. Con un $hidden\_size = 64$ vemos que tenemos un balance entre la precisión, que es bastante alta (95 %) y el tiempo de ejecución que es alto (52*s*) pero no tanto como los siguientes valores.

Con unos valores de 128 y 256 vemos como la precisión es la misma practicamente y encima, es demasiado baja. Esto se debe a que pasar un input 28 valores a un estado intermedio de 128 o 256 tras diferentes operaciones matriciales, causa un perdida de información parecida a la que pasa con valores pequeños. Además, el tiempo de ejecución crece exponencialmente y se hace imposible de entrenar con estos valores.

### 5.3.2. Input Size

El «input size» es también muy importante, ya que podemos definir el tamaño de cada iteración y cuanto menor sea el *input size* tendremos que añadir más celdas a la *RNN*. Añadir demasiadas celdas o muy pocas puede tener un impacto no deseado dependiendo del problema que estemos resolviendo.

En las Figuras 5.8 y 5.9 podemos ver el efecto de diferentes tamaños de entrada. En el caso de predecir los dígitos a partir de una imagen, escoger un tamaño muy pequeño ha implicado una baja precisión junto a un tiempo de ejecución elevado. Esto también

se puede deber a que el *hidden_state* es de 64 y por lo tanto, pasar de vectores de 2 a 64 puede ser muy ineficaz, dando este resultado. Por eso es importante escoger una buena combinación entre dimensiones.

Conforme aumenta el tamaño de entrada también decrece el número de celdas. En este caso, la precisión y tiempo de ejecución mejoran cuanto menos celdas haya, ya que este es un problema de predicción a partir de una sola imagen.

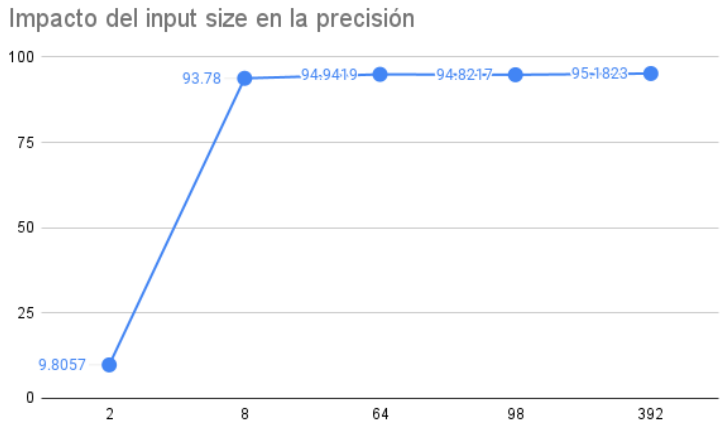



**Figura 5.8:** Impacto del input size en la precisión

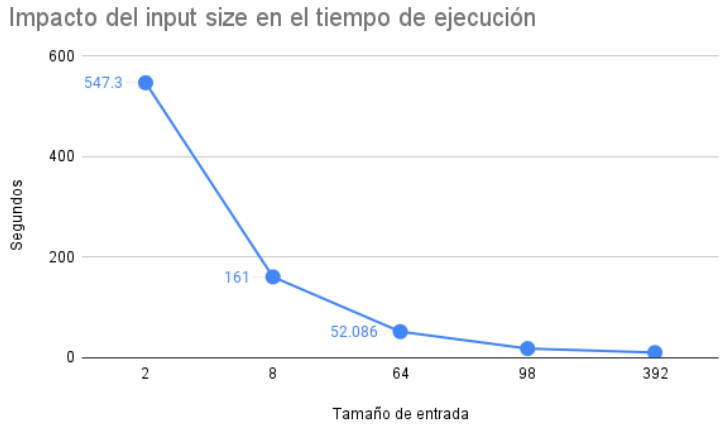



**Figura 5.9:** Impacto del input size en el tiempo de ejecución

## 5.4 Comparación con PyTorch

*PyTorch* es una de las plataformas de entrenamiento más usadas actualmente y que, como dispone de la *RNN* ya implementada y probada, podemos compararla con la nuestra con el mismo modelo de *MNIST* que hemos estudiado antes.

En el código A.7 podemos observar con mayor detenimiento el *script* que ejecuta de manera equivalente el mismo modelo que tenemos definido en *HELENNA*. Utilizamos 3 como el número de épocas al igual que en *HELENNA* y el mismo *dataset* de *MNIST*.

Para hacer esta prueba, vamos a usar una combinación de *input_size* y *hidden_size* que sabemos que da buenos resultados, y vamos a compararlos con *PyTorch* para validar que nuestra función funciona tal y como se espera correctamente. Para el *batch size*, utilizaremos uno de 32 y uno de 64.

Después de ejecutar el modelo en el *script* de *PyTorch* hemos obtenido los resultados que se muestran en la Figura 5.10.

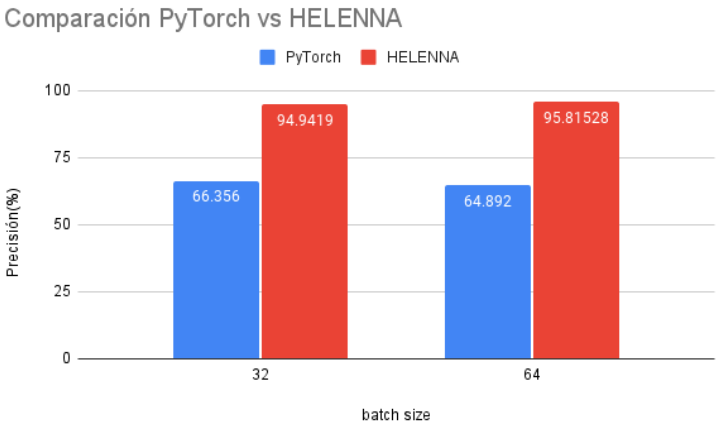



**Figura 5.10:** Modelo MNIST HELENNA vs PyTorch

Como puede observarse, los resultados obtenidos en *HELENNA* con respecto a la precisión son muy superiores a *PyTorch*. Cabe destacar que estos resultados son en base a ejecuciones cambiando los parámetros mínimos para hacer funcionar el modelo, es decir, en *PyTorch* existe mucha flexibilidad a la hora de cambiar parámetros y en *HELENNA* puede tener valores por defecto que causen esta gran diferencia.

Aclarado esto, tenemos resultados muy buenos con nuestra *RNN* con los que podemos verificar que nuestra implementación satisface nuestros objetivos.

# CAPÍTULO 6
# Conclusión

Para finalizar vamos a presentar unas consideraciones finales donde hablaremos sobre los objetivos planteados y si se han cumplido y después, vamos a hablar sobre cómo se podría continuar con el trabajo ya hecho por tal de mejorarlo y extenderlo mas.

## 6.1 Consideraciones finales

Después de todos estos capítulos podemos concluir que hemos completado todos los objetivos propuestos. Para empezar, hemos visto cómo funcionan las Redes Neuronales desde el perceptrón mas simple hasta el *backpropagation* que nos permite aprender de los errores.

Hemos visto como funcionan las *RNN* en un nivel abstracto y en un nivel bajo donde hemos presentado todas las operaciones que se hacen. Además hemos visto los diferentes tipos que existen de *RNN*.

También hemos visto como funciona *HELENNA* de manera general viendo la mayoría de ficheros que usaremos, además de las funciones que utilizamos para el desarrollo.

Hemos añadido soporte para las *RNN* en *HELENNA* implementando las siguientes funciones:

1. *Parseo* de *RNNs*.
2. Creación de todas las estructuras que engloban a una *RNN* básica, en especial, las celdas. Además, de la ubicación y desubicación de estas estructuras.
3. Inicialización de parámetros con la inicialización de «Xavier» y la inicialización de «He».
4. Función de *forwarding*.
5. Función de *backwarding*.
6. Actualización de los parámetros de la *RNN*.
7. Extensión de la creación y carga de modelos para las *RNN*.
8. Definición de diferentes modelos para el dataset de *MNIST*.

El desarrollo de la *RNN* ha sido completado y acoplado a la plataforma de manera que use todas las funciones que hacen uso de diferente *hardware*. Para acabar, hemos

validado la implementación con diferentes pruebas y además hemos contrastado los resultados con *PyTorch* que es muy utilzado y por lo tanto, podemos concluir que nuesta implementación pasa satisfactoriamente las pruebas.

## 6.2 Trabajo futuro

Aunque el trabajo de añadir soporte a *RNNs* haya acabado, queda mucho trabajo por hacer ya que hay algunas limitaciones en la implementación actual.

Para empezar, hemos probado las *RNN* con *MNIST* que es un *dataset* sencillo. Se debería probar la *RNN* con *datasets* más complejos, que necesitaran realmente la utilización de *RNNs*, por contener secuencias.

Solo hay soporte para una *RNN* en un único modelo. Se puede resolver creando una estructura que contenga todas las estructuras que hemos visto en el capítulo 4.

Como hemos visto en el capítulo 3, hay muchos tipos de *RNNs* que se usan en producción, en especial las *LSTM*. También existen los *transformers* que hemos visto en el capítulo 2 que se usa para aplicaciones equivalentes. En un futuro, será necesario extender la plataforma para dar soporte a estas nuevas capas.

Por el lado del rendimiento, hay algunas partes del código implementado que pueden ser optimizadas para que no haya copias innecesarias u operaciones extra.

Para acabar, hay muchos parámetros extra que se puede añadir pero que requieren mayor complejidad que pueden ser implementados. Uno de ellos es el número de capas que añade mas dimensiones de celdas.

# Bibliografía


[1] A. Aldo Faisal, Cheng Soon Ong and Marc Peter Deisenroth. *Mathematics for machine learning*. Cambridge University Press, 2020.

[2] Charles Elkan, John Berkowitz and Zachary C. Lipton. A Critical Review of Recurrent Neural Networks for Sequence Learning *Machine Learning*, 2015.

[3] Charlie Murphy Verified Perceptron Convergence Theorem *Software and its engineering*, Association for Computing Machinery, New York, NY, USA, 43–50

[4] The perceptron: A probabilistic model for information storge and organization in the brain. Consultado en https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf.

[5] David A. Patterson y John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, quinta edición, 2011.

[6] Peter Norvig and Stuart J. Russell. *Artificial Intelligence A Modern Approach*. Pearson Education Limited, tercera edición, 2016.

[7] Aprendiendo representaciones internas con propagación de errores. Consultado en https://web.stanford.edu/class/psych209a/ReadingsByDate/02_06/PDPVolIChapter8.pdf

[8] Biografía de Donald Hebb. Consultado en http://www.facmed.unam.mx/Libro-NeuroFisio/Personas/Hebb/Hebb.html

[9] Biografía Frank Rosenblatt. Consultado en https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon

[10] Biografía Nathaniel Rochester III. Consultado en https://modha.org/2006/09/nathaniel-rochester-iii-1919-2001/

[11] Cálculo lógico de ideas inherentes en actividad nerviosa. Consultado en https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf

[12] Calculadora IBM 701. Consultado en https://www.ibm.com/ibm/history/exhibits/701/701_intro.html

[13] Calculadora IBM 704. Consultado en https://dbpedia.org/page/IBM_704

[14] Cross Entropy Loss Consultado en https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy

[15] Como funciona el algoritmo de backpropagation Consultado en http://neuralnetworksanddeeplearning.com/chap2.html

[16] Documentación de RNNs en PyTorch Consultado en https://pytorch.org/docs/stable/generated/torch.nn.RNN.html.

[17] Entender la importancia del learning rate en modelos. Consultado en https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

[18] Evolución del cerebro humano. Consultado en https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5624727/

[19] Exploding y Vanishing Gradients Consultado en https://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf.

[20] Human brain result of 'extraordinarily fast' evolution Consultado en https://www.theguardian.com/science/2004/dec/29/evolution.science

[21] Gradient descent Consultado en https://medium.com/iitg-ai/into-the-depths-of-gradient-descent-52cf9ee92d36

[22] Google translate Consultado en https://translate.google.com/.

[23] La organización del comportamiento. Consultado en http://s-f-walker.org.uk/pubsebooks/pdfs/The_Organization_of_Behavior-Donald_O._Hebb.pdf.

[24] Implementación de RNNs en PyTorch Consultado en https://pytorch.org/docs/stable/_modules/torch/nn/modules/rnn.html#RNN.

[25] Mean Squared Error Consultado en https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

[26] MNIST con redes neuronales recurrentes Consultado en https://medium.com/machine-learning-algorithms/mnist-using-recurrent-neural-network-2d070a5915a2.

[27] Página principal de CMake Consultado en https://cmake.org/.

[28] Página principal de Microsoft Excel Consultado en https://www.microsoft.com/en-us/microsoft-365/excel.

[29] Página principal del proyecto Latex Consultado en https://www.latex-project.org/.

[30] Página principal de GNU Make Consultado en https://www.gnu.org/software/make/.

[31] Página principal de PyTorch Consultado en https://pytorch.org.

[32] Página principal de PyTorch Lightning Consultado en https://www.pytorchlightning.ai/.

[33] Repositorio de pytorch Vision Consultado en https://github.com/pytorch/vision.

[34] Pagina principal de Siri de Apple Consultado en https://www.apple.com/siri/.

[35] Página principal de Tensorflow Consultado en https://tensorflow.org.

[36] Página principal de Tensorflow.js Consultado en https://www.tensorflow.org/js.

[37] Página principal de Valgrind Consultado en https://www.valgrind.org/.

[38] Página principal de Vim Consultado en https://www.vim.org/.

[39] Página principal de Visual Studio Code de Microsoft Consultado en https://code.visualstudio.com/.

[40] Propuesta de proyecto de verano en inteligencia artificial, darmouth. Consultado en http://www-formal.stanford.edu/jmc/history/dartmouth.pdf

[41] Que es Alexa? Consultado en https://developer.amazon.com/en-GB/alexa.

[42] Que es una FPGA Consultado en https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html#:~:text=Field%20Programmable%20Gate%20Arrays%20(FPGAs,or%20functionality%20requirements%20after%20manufacturing..

[43] Que son las series de tiempo. Consultado en https://www.investopedia.com/terms/t/timeseries.asp.

[44] Ratio de masa del cerebro a masa del cuerpo. Consultado en https://www.researchgate.net/figure/Brain-mass-to-body-mass-ratio-Plot-of-the-relationship-between-brain-mass-in-g-and_fig2_286856709

[45] Repositorio de HELENNA Consultado en https://github.com/PEAK-UPV/HELENNA.

[46] Teorema de convergencia del perceptrón. Consultado en https://www.cs.princeton.edu/~tcm3/docs/mapl_2017.pdf.

# APÉNDICE A
# Código

```
//
// arithmetic functions
//

#ifndef __ARITHMETIC_H
#define __ARITHMETIC_H

#ifdef AVX_SUPPORT
#include <x86intrin.h>
#endif

#include <stdlib.h>

#include "globals.h"

void fn_read_buffer(int mid_src, type *dst, size_t size);
void fn_write_buffer(int mid_dst, type *src, int size);
void fn_vect_random_float(int mid_v, int size, type min, type max);
type fn_pow(int mid_v, int element, int base);
type fn_fabs(int mid_v, int element);
type fn_value(int mid_v, int element);
void fn_write_value(int mid_v, int element, type value);

void weight_reshape(int KW, int KH, int I, int O, int mid_in, int mid_out);
void fn_vect_step(int mid_a, int mid_b, int n);
void fn_vect_mult(int mid_a, int mid_b, int mid_c, int n);
type fn_vect_mult_add(int mid_a, int mid_b, int n);
type fn_vect_mult_add_with_offset(int mid_a, int offset_a, int mid_b, int
    offset_b, int n);
void fn_vect_scalar_product(float f, int mid_a, int n);

void fn_vect_max(int mid_a, int mid_b, int mid_c, int n);
type fn_vect_mult_add_with_stride(int mid_a, int mid_b, int stride_a, int
    stride_b, int n);
void fn_vect_XxW_stride_iterations(int mid_xin, int mid_w, int mid_o, int
    stride_vectors_x, int vector_size, int num_vectors);
void fn_vect_V2subV1xK(int mid_v1, type k, int mid_v2, int n);
void fn_GEMM(int rowMajor, int aTransp, int bTransp, int m, int n, int k, float
     alpha, int mid_a, size_t offset_a, int lda, int mid_b, size_t offset_b,
    int ldb, float beta, int mid_c, size_t offset_c, int ldc);
void fn_matmul(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b, int
    cols_b, int mid_c, int cols_c, int offset_cols_c);
void fn_matmul_bt(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b, int
     cols_b, int mid_c, int cols_c, int offset_c);
void fn_matmul_at(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b, int
     cols_b, int ld_b, int offset_b, int mid_c, int add);
void fn_matadd_col(int mid_a, int rows_a, int cols_a, int mid_v);
void fn_matrix_transpose(type *ptr_a, int rows_a, int cols_a, type *ptr_b);
```

```
41 void fn_matmul_elwise(int mid_a, int rows_a, int cols_a, int mid_b, int mid_c);
42 void fn_vector_to_matrix(int mid_v, int elements, int mid_m);
43 void fn_matsub(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b, int
       cols_b, int mid_c);
44 void fn_matadd(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b, int
       cols_b, int mid_c);
45 void fn_mat_A_plus_b_L1(int mid_a, int rows_a, int cols_a, type beta, int mid_b
       , int rows_b, int cols_b, int mid_c);
46 void fn_mat_copy(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b, int
       cols_b);
47 void fn_mat_set(int mid_a, int rows_a, int cols_a, type value);
48 void fn_mat_reduce_rows(int mid_a, int rows_a, int cols_a, int mid_v);
49 void fn_vec_axpy(type a, int mid_x, int mid_y, int size_v);
50 void fn_matrix_relu(int mid_x, int mid_y, int rows, int cols);
51 void fn_matrix_relu_der(int mid_m1, int rows, int cols, int mid_m2);
52 void fn_matrix_sigmoid(int mid_x, int mid_y, int rows, int cols);
53 void fn_matrix_sigmoid_der(int mid_m1, int rows, int cols, int mid_m2);
54 void fn_matrix_softmax(int mid_x, int mid_y, int rows, int cols);
55 void fn_matrix_softmax_der(int mid_m1, int rows, int cols, int mid_m2);
56 void fn_print_matrix(int mid_m, int rows, int cols);
57 void fn_print_channel_as_rgb(int mid_red, int mid_green, int mid_blue, int x,
       int y, int bs, int eh, int b, int img, int lr, int inout);
58 void fn_print_image(int mid_x, int mid_y, int I, int O, int WI, int HI, int WO,
        int HO, int BS, int l, int e);
59 void fn_print_matrix_int(int mid_m, int rows, int cols);
60 void fn_print_matrix_as_image(int mid_m, int rows, int cols, int bs);
61 void fn_print_matrix_as_image_batch_entry(int mid_m, int rows, int cols, int bs
       , int b);
62 void fn_vect_limit(int mid_v1, type min_val, type max_val, int n);
63 void fn_vect_trunc(int mid_v1, type threshold, int n);
64 void fn_mat_limit(int mid_m, int rows, int cols, type min_val, type max_val);
65 void fn_vec_add(int mid_v1, int mid_v2, int mid_v3, int size);
66 void fn_vec_copy(int mid_v1, int mid_v2, int size);
67 void fn_vec_copy_with_stride(int mid_v1, int mid_v2, int stride_v1, int
       stride_v2, int size, int offset_v2);
68 void fn_vec_copy_with_offset(int mid_v1, int mid_v2, int size, int offset_v1,
       int offset_v2);
69 void fn_vec_copy_with_stride_from_cpumem(type *ptr_v1, int mid_v2, int
       stride_v1, int stride_v2, int size, int offset_v2);
70 void fn_init_params(int mid_p, int dim3, int dim2, int dim1, int dim0);
71 void fn_zero_vec(int mid_x, size_t vec_size);
72 void fn_set_vec(type value, int mid_x, size_t vec_size);
73 void fn_vec_quantize(int mid_v1, int size, int bits_int, int bits_frac);
74 void fn_mat_quantize(int mid_m, int rows, int cols, int bits_int, int bits_frac
       );
75
76
77 void fn_im2col(int I, int W, int H, int B, int KW, int KH, int WO, int HO, int
       PW, int PH, int SW, int SH, int mid_in, int mid_out, int first_row, int
       last_row, int first_col, int last_col);
78 void fn_col2im(int I, int W, int H, int B, int KW, int KH, int WO, int HO, int
       PW, int PH, int SW, int SH, int mid_in, int mid_out, int first_col, int
       last_col);
79
80 void fn_matmul_im2col(int mid_x, int mid_k, int mid_o, int I, int WI, int HI,
       int B, int KW, int KH, int O, int WO, int HO, int PW, int PH, int SW, int
       SH, int mid_buf, int cols_buf);
81 void fn_matmul_im2col_t(int mid_x, int mid_y, int mid_o, int I, int WI, int HI,
        int B, int KW, int KH, int O, int WO, int HO, int PW, int PH, int SW, int
       SH, int mid_buf, int rows_buf);
82 void fn_matmul_col2im(int mid_k, int mid_y, int mid_o, int I, int WI, int HI,
       int B, int KW, int KH, int O, int WO, int HO, int PW, int PH, int SW, int
       SH, int mid_buf, int cols_buf);
83
```

```
84
85
86  void im2col_bp_ant(int O, int WO, int HO, int I, int WI, int HI, int B, int KW,
        int KH, int PW, int PH, int SW, int SH, int mid_in, int mid_out);
87
88  void fn_padding(int I, int WI, int HI, int KW, int KH, int O, int WO, int HO,
        int B, int mid_in, int mid_out);
89  void fn_depadding(int I, int WI, int HI, int KW, int KH, int O, int WO, int HO,
        int B, int mid_in, int mid_out);
90
91  void fn_batch_normalization(int n, int B, int mid_gamma, int mid_beta, type eps
        , int mid_mu, int mid_xmu, int mid_sq, int mid_var,
92          int mid_sqrtvar, int mid_ivar, int mid_xhat, int mid_gammax,
93          int mid_in, int mid_out);
94  void fn_batch_normalization_compute_gradients(int n, int B, int mid_dgamma, int
        mid_dbeta, int mid_xhat, int mid_in);
95  void fn_batch_normalization_backward(int n, int B, int mid_gamma, type eps, int
        mid_xhat, int mid_dxhat,
96              int mid_ivar, int mid_divar, int mid_xmu,
97              int mid_dxmu1, int mid_dxmu2, int mid_dx1, int mid_dx2, int
                    mid_dmu,
98              int mid_sqrtvar, int mid_dsq, int mid_dsqrtvar, int mid_dvar, int
                    mid_var,
99              int mid_in, int mid_out);
100
101 void fn_maxpooling(int I, int WI, int HI, int SW, int SH, int KW, int KH, int O
        , int WO, int HO, int B, int mid_in, int mid_out, int mid_addr);
102 void fn_demaxpooling(int I, int WI, int HI, int SW, int SH, int KW, int KH, int
        O, int WO, int HO, int B, int mid_out, int mid_in, int mid_addr);
103
104 // fused operations
105 void fn_fused_demaxpooling_relu_der(int I, int WI, int HI, int SW, int SH, int
        KW, int KH, int O, int WO, int HO, int B, int mid_out, int mid_in, int
        mid_addr, int mid_out_prev);
106 void fn_fused_matmul_at_relu_der(int mid_a, int rows_a, int cols_a, int mid_b,
        int rows_b, int cols_b, int ld_b, int offset_b, int mid_c, int mid_out_prev
        );
107 void fn_fused_matmul_matadd_col(int mid_a, int rows_a, int cols_a, int mid_b,
        int rows_b, int cols_b, int mid_c, int cols_c, int offset_cols_c, int mid_v
        );
108
109
110 void fn_clip_vector(int mid_v, float min, float max, int n);
111
112 void fn_im2col_pack(int I, int W, int H, int B, int KW, int KH, int WO, int HO,
        int PW, int PH, int SW, int SH, int mid_in, int mid_out, int mid_k, int O,
        int rows_per_output);
113 void fn_matrix_pack(int mid_src, int rows, int cols, int mid_dst, int
        mid_shifts, int *width);
114 void fn_matmul_pack(int mid_a, int rows_a, int cols_a, int mid_b, int rows_b,
        int cols_b, int mid_c, int rows_per_output);
115 void fn_matset_random_ones(int mid_m, int rows, int cols, float prob);
116 void fn_print_timing_statistics();
117
118 void fn_copy(int mid_dst, int mid_src, int offset_dst, int offset_src, int size
        );
119
120 void fn_upsampling(int I, int WI, int HI, int O, int WO, int HO, int B, int SF,
        int mid_in, int mid_out);
121 void fn_downsampling(int I, int WI, int HI, int O, int WO, int HO, int B, int
        SF, int mid_in, int mid_out);
122
123 void fn_cross_entropy_der(int mid_a, int mid_b, int rows, int cols, int mid_c);
```

```
124 void fn_mean_square_error_der(int mid_a, int mid_b, int rows, int cols, int
        mid_c);
125 void fn_reshape(int mid_x, int mid_y, int I, int WI, int HI, int WO, int HO,
        int BS);
126 void fn_square(int mid_x, int mid_y, int size);
127 void fn_tanh(int mid_x, int mid_y, int size);
128 void fn_matadd_value(int mid_x, int mid_y, type value, int size);
129
130 #endif
```

**Código A.1:** Aritmetica disponible de buffers

```
1  typedef struct st_layer { // layer
2      layer_type ltype;
3      char layer_name[200];
4      int layer_iteration;       // for loops
5      int num_inputs_per_neuron; // number of inputs per neuron
6      int num_neurons; // number of neurons of the layer
7
8      // previous and next layer
9      int previous_layer;
10     int next_layer;
11
12     int id_weights;
13
14     // layer configuration
15     int num_weights; // number of weights
16     int num_bias; // number of biases
17
18     // memory identifiers
19     int mid_weights; // weights
20     int mid_weights_glob; // weights (global, for MPI)
21     int mid_bias; // bias
22     int mid_bias_glob; // bias (global, for MPI)
23     int mid_y; // y (output)
24     int mid_y_der; // y derivative
25     int mid_y_err; // y error
26     int mid_g_w; // x error (gradients)
27     int mid_g_w_momentum; // x error (gradients) at iteration i-1 (momentum)
28     int mid_g_w_glob; // x error (for MPI)
29     int mid_g_bias; // bias gradients
30     int mid_g_bias_momentum; // bias gradients at iteration i-1 (momentum)
31     int mid_g_bias_glob; // bias gradients (for MPI)
32     int mid_im2col_buffer; // im2col buffer
33     int mid_im2col_bp_buffer; // im2col buffer (backpropagation)
34     int mid_y_err_im2col; // im2col gradients at output
35     int mid_x_im2col; // im2col gradients at input
36     int mid_mask; // mask
37     int mid_wt; // temporal buffer for w
38     int mid_y_err_im2col_addr;
39     int mid_x_im2col_addr;
40     int mid_maxpooling_x_addr; // for maxpooling layer, links input pixels to
           output pixels
41
42     // specific memory objects for batch normalization layer
43     int mid_gamma;
44     int mid_beta;
45     int mid_mu;
46     int mid_xmu;
47     int mid_sq;
48     int mid_var;
49     int mid_sqrtvar;
50     int mid_ivar;
51     int mid_xhat;
```

```
52      int mid_gammax;
53      int mid_dbeta;
54      int mid_dgamma;
55      int mid_dxhat;
56      int mid_divar;
57      int mid_dxmu1;
58      int mid_dsqrtvar;
59      int mid_dvar;
60      int mid_dsq;
61      int mid_dxmu2;
62      int mid_dx1;
63      int mid_dmu;
64      int mid_dx2;
65
66      // fields for batch normalization layer
67      type eps;
68      int batchnorm_num_vectors; // number of vectors to normalize (either
            num_neurons for FC or O for CONVS)
69      int batchnorm_vectors_size; // data size to normalize (either batch size
            for FC or WO * HO * batch size for CONVS)
70
71      // pruning-related buffers
72      int mid_alpha;
73      int mid_alpha_der;
74      int mid_alpha_mask;
75      int mid_w_masked;
76      int mid_g_alpha;
77
78      // temporal buffer used to add error in the backpropagation process
79      int mid_temp_buffer;
80
81      // convolutional-related parameters
82      int I; // input channels
83      int O; // output channels
84      int WI; // width input channel
85      int HI; // height input channel
86      int WO; // width output channel
87      int HO; // height output channel
88      int KW; // kernel width
89      int KH; // kernel height
90      int PW; // padding width
91      int PH; // padding height
92      int SW; // stride width
93      int SH; // stride height
94      size_t cols_fused_im2col;
95      size_t rows_fused_im2col_t;
96
97      // dropout-related parameters
98      type dropout_rate; // dropout rate
99
100     // concat-related and add-related parameters
101     int concat_layer;
102     int add_layer;
103     int layer_output_splits;    // the output splits, means the output goes to
            two different layers
104
105
106
107     // upsampling-related parameters
108     int SF;                     // sampling factor
109
110     int buffer_output_rows; // rows to be processed per step in im2col+matmul
            pipelined operation
```

```
111     int buffer_input_rows; // rows to be processed per step in im2col_bp+matmul
            pipelined operation
112
113     // group convolutional-related parameters
114     int G; // number of groups in group convolutional layer
115     int IG; // number of input channels per group
116     int OG; // number of output channels per group
117
118     // rearrange convolutional-related parameters
119     int S; // stride
120
121     // perfect shuffle-related parameters
122
123     int ps_a; // number of inputs per switch
124     int ps_b; // number of outputs per switch
125     int ps_n; // number of stages in a multi-layer PS
126     int ps_i; // layer position in a multi-layer PS, in range [1, n]
127
128     // permutation-related parameters
129
130     int *perm;
131
132     // pruned fully connected-related parameters
133     int BR; // number of rows in a pruning block
134     int BC; // number of cols in a pruning block
135     int alpha_rows;
136     int alpha_cols;
137
138     // fused operations parameters
139     int fused_demaxpooling_relu_der;
140     int fused_matmul_at_relu_der;
141     int fused_matmul_matadd_col;
142
143     void (*fn_parse)(int layer, char *parameters);
144     void (*fn_allocate)(int layer);
145     void (*fn_deallocate)(int layer);
146     void (*fn_initialize)(int layer);
147     void (*fn_forward)(int layer, int fbe, int lbe, int training);
148     //void (*fn_backward)(int layer, int training, int minibatch_size);
149     void (*fn_backward)(int layer, int minibatch_size);
150     void (*fn_propagate_error)(int layer, int minibatch_size);
151
152     // stat-related variables
153     float float_operations_forward; // number of float operations performed in
            forward pass
154     float float_operations_backward; // number of float operations performed in
            backward pass
155     float float_operations_propagate; // number of float operations performed
            in propagation of error
156     float float_operations_update; // number of float operations performed in
            update of parameters
157 } layer;
```

**Código A.2:** Definición estructura de la capa

```
1  typedef struct _rnn_tmp_variables {
2      int multxw;
3      int multaw;
4      int multhy;
5      int sum_mults;
6      int sum_bias_a;
7      int sum_bias_y;
8      int sum_bias_h;
9      // grad
```

```
    int mid_square_a;
    int mid_ones_matrix;
    int mid_df;
    int mid_df_dh_next;
    int mid_dh_next;
    int mid_dl_dh;
    int mid_dh;
    int mid_dWhh;
    int mid_dWxh;
    int mid_dba;
    int mid_dbh;
} rnn_tmp_variables;
```

**Código A.3:** Buffers temporales de la RNN

```
#include "nnsim.h"
#include <math.h>

//#define DEBUG
//#define MANY_TO_ONE
void rnn_parse_tp(int layer, char *parameters)
{
    char splitted_parameters[RNN_NUM_PARAMETERS][10];
    char *delimeter = " ";
    char *token = strtok(parameters, delimeter);
    for (int i = 0; i < RNN_NUM_PARAMETERS && token != NULL; i++)
    {
        strcpy(splitted_parameters[i], token);
        token = strtok(NULL, delimeter);
    }
    input_size = atoi(splitted_parameters[0]);
    hidden_size = atoi(splitted_parameters[1]);
    rnn_cell_count = atoi(splitted_parameters[2]);
    char *af = splitted_parameters[3];
    if (strcmp(af, "relu") == 0)
    {
        rnn_parameters.activation_function = RELU;
    } else {
        rnn_parameters.activation_function = TANH;
    }
    layers_ptr[layer]->previous_layer = layer - 1;
    #ifdef MANY_TO_ONE
    layers_ptr[layer]->num_neurons = hidden_size;
    #else
    layers_ptr[layer]->num_neurons = hidden_size * rnn_cell_count;
    #endif

}

void rnn_forward(int layer, int fbe, int lbe, int training)
{
    int mid_y = layers_ptr[layer - 1]->mid_y;
    fn_zero_vec(rnn_cells[0].mid_h_prev, hidden_size * minibatch_size);
    for (int t = 0; t < rnn_cell_count; t++)
    {
        fn_copy(rnn_cells[t].mid_x, mid_y, 0, t * input_size * minibatch_size,
            input_size * minibatch_size); // TODO: optimize
        rnn_forward_cell(t);
    }
    get_outputs(layer);
    #ifdef DEBUG
    rnn_print_outputs(layer);
    #endif
}
```

```
49
50 // h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh)
51 void rnn_forward_cell(int t)
52 {
53     int x = rnn_cells[t].mid_x;
54     int h_prev = rnn_cells[t].mid_h_prev;
55     int h_next = rnn_cells[t + 1].mid_h_prev;
56     int W_hh = rnn_parameters.mid_Whh;
57     int W_hx = rnn_parameters.mid_Wxh;
58     int mid_bias_x = rnn_parameters.mid_bias_x;
59     int mid_bias_h = rnn_parameters.mid_bias_h;
60     //tmp
61     int multxw = tmp_variables.multxw;
62     int multhw = tmp_variables.multhw;
63     int sum_mults = tmp_variables.sum_mults;
64     int sum_bias_x = tmp_variables.sum_bias_x;
65     int sum_bias_h = tmp_variables.sum_bias_x;
66     fn_matmul(W_hx, hidden_size, input_size, x, input_size, minibatch_size,
           multxw, 1, 0);
67     fn_matmul(W_hh, hidden_size, hidden_size, h_prev, hidden_size,
           minibatch_size, multhw, 1, 0);
68
69     fn_matadd(multhw, hidden_size, minibatch_size, multxw, hidden_size,
           minibatch_size, sum_mults);
70     fn_matadd(sum_mults, hidden_size, minibatch_size, mid_bias_x, hidden_size,
           minibatch_size, sum_bias_x);
71     fn_matadd(sum_bias_x, hidden_size, minibatch_size, mid_bias_h, hidden_size,
            minibatch_size, sum_bias_h);
72     if (rnn_parameters.activation_function == TANH)
73     {
74         fn_tanh(sum_bias_h, h_next, hidden_size * minibatch_size);
75     }
76     else
77     {
78         fn_matrix_relu(sum_bias_h, h_next, hidden_size, minibatch_size);
79     }
80
81     #ifdef DEBUG
82     debug_forward(t);
83     #endif
84 }
85
86 void debug_forward(int t)
87 {
88     printf("stats iteration: %d\n", t);
89     rnn_print_parameters();
90     printf("mini %d\n", minibatch_size);
91     printf("x\n");
92     fn_print_stats_vector(rnn_cells[t].mid_x, input_size * minibatch_size);
93     printf("h\n");
94     fn_print_stats_vector(rnn_cells[t].mid_h_prev, hidden_size * minibatch_size
           );
95     printf("multxw\n");
96     fn_print_stats_vector(tmp_variables.multxw, hidden_size * minibatch_size);
97     printf("multhw\n");
98     fn_print_stats_vector(tmp_variables.multhw, hidden_size * minibatch_size);
99     printf("sum mults\n");
100    fn_print_stats_vector(tmp_variables.sum_mults, hidden_size * minibatch_size
           );
101    printf("sum bias h\n");
102    fn_print_stats_vector(tmp_variables.sum_bias_x, hidden_size *
           minibatch_size);
103    printf("h next\n");
```

```
104     fn_print_stats_vector(rnn_cells[t + 1].mid_h_prev, hidden_size *
            minibatch_size);
105 }
106 void rnn_forward_to_zero(int t)
107 {
108     fn_zero_vec(tmp_variables.multxw, hidden_size * minibatch_size);
109     fn_zero_vec(tmp_variables.multhw, hidden_size * minibatch_size);
110     fn_zero_vec(tmp_variables.sum_bias_x, hidden_size * minibatch_size);
111     fn_zero_vec(tmp_variables.sum_mults, hidden_size * minibatch_size);
112     fn_zero_vec(rnn_cells[t + 1].mid_h_prev, hidden_size * minibatch_size);
113 }
114
115
116 void get_outputs(int layer)
117 {
118     #ifdef DEBUG
119     fn_print_stats_vector(rnn_cells[rnn_cell_count].mid_h_prev, hidden_size *
            minibatch_size);
120     #endif
121     #ifdef MANY_TO_ONE
122     fn_copy(layers_ptr[layer]->mid_y, rnn_cells[rnn_cell_count].mid_h_prev, 0,
            0, hidden_size * minibatch_size);
123     #else
124     for (int t = 1; t <= rnn_cell_count; t++)
125     {
126         fn_copy(layers_ptr[layer]->mid_y, rnn_cells[t].mid_h_prev, (t - 1) *
                hidden_size * minibatch_size, 0, hidden_size * minibatch_size);
127     }
128     #endif
129 }
130 void rnn_print_outputs(int layer)
131 {
132     printf("\noutputs : \n");
133     fn_print_stats_vector(layers_ptr[layer]->mid_y, layers_ptr[layer]->
            num_neurons);
134 }
135 void rnn_backward_tp(int layer, int minibatch_size)
136 {
137     rnn_gradients_to_zero();
138     int y_err = layers_ptr[layer]->mid_y_err;
139     fn_zero_vec(tmp_variables.mid_dh, hidden_size * minibatch_size);
140     for (int t = rnn_cell_count - 1; t >= 0; t--)
141     {
142 #ifdef MANY_TO_ONE
143         if (t == rnn_cell_count - 1)
144         {
145             fn_copy(tmp_variables.mid_dl_dh, y_err, 0, t * hidden_size *
                    minibatch_size, hidden_size * minibatch_size);
146         }
147         else if(t == rnn_cell_count - 2)
148         {
149             fn_zero_vec(tmp_variables.mid_dl_dh, hidden_size * minibatch_size);
150         }
151 #else
152         fn_copy(tmp_variables.mid_dl_dh, y_err, 0, t * hidden_size *
                minibatch_size, hidden_size * minibatch_size);
153 #endif
154         rnn_cell_backward_tp(t);
155     }
156 }
157 void rnn_cell_backward_tp(int t)
158 {
159     int Wxh = rnn_parameters.mid_Wxh;
160     int Whh = rnn_parameters.mid_Whh;
```

```
161     int xt = rnn_cells[t].mid_x;
162     int h_next = rnn_cells[t + 1].mid_h_prev;
163     int h_prev = rnn_cells[t].mid_h_prev;
164     //tmp
165     int df = tmp_variables.mid_df;
166     int df_dh_next = tmp_variables.mid_df_dh_next;
167     int dl_dh = tmp_variables.mid_dl_dh;
168     int dh = tmp_variables.mid_dh;
169     int square_a = tmp_variables.mid_square_a;
170     int ones_matrix = tmp_variables.mid_ones_matrix;
171     #ifdef DEBUG
172     printf("\nDEBUGGING BACKWARD STEP: %d ------------------------\n", t);
173     printf("dh\n");
174     fn_print_stats_vector(tmp_variables.mid_dh, hidden_size);
175     #endif
176     // dh = dl_t + dh_t
177     fn_vec_add(dl_dh, dh, dh, hidden_size * minibatch_size);
178     if (rnn_parameters.activation_function == TANH)
179     {
180         // tanh der dL/dh * (1 - h^2) -> element mult
181         fn_square(h_next, square_a, hidden_size * minibatch_size);
182         fn_matsub(ones_matrix, hidden_size, minibatch_size, square_a,
                hidden_size, minibatch_size, df);
183     }
184     else
185     {
186         fn_matrix_relu_der(h_next, hidden_size, minibatch_size, df);
187     }
188     fn_vect_mult(dh, df, df_dh_next, hidden_size * minibatch_size);
189     // dWxh
190     fn_matmul_bt(df_dh_next, hidden_size, minibatch_size, xt, input_size,
            minibatch_size, tmp_variables.mid_dWxh,
191                 input_size, 0);
192     //dWhh
193     fn_matmul_bt(df_dh_next, hidden_size, minibatch_size, h_prev, hidden_size,
            minibatch_size, tmp_variables.mid_dWhh,
194                 hidden_size, 0);
195     // fn_matadd(df_dh_next, hidden_size, minibatch_size, ones_matrix,
            hidden_size, minibatch_size, tmp_variables.mid_dbh);
196     //dh_prev
197     fn_matmul_at(df_dh_next, hidden_size, minibatch_size, Whh, hidden_size,
            hidden_size, hidden_size, 0, dh, 0);
198     //fn_matmul(Whh, hidden_size, hidden_size, df_dh_next, hidden_size,
            minibatch_size, dh, minibatch_size, 0);
199
200     //add grads
201     fn_matadd(gradients.mid_dWhh, hidden_size, hidden_size, tmp_variables.
            mid_dWhh,
202               hidden_size, hidden_size, gradients.mid_dWhh);
203     fn_matadd(gradients.mid_dWxh, hidden_size, input_size, tmp_variables.
            mid_dWxh,
204               hidden_size, input_size, gradients.mid_dWxh);
205     fn_matadd(gradients.mid_dba, hidden_size, 1, df_dh_next, hidden_size, 1,
            gradients.mid_dba);
206     fn_matadd(gradients.mid_dbh, hidden_size, 1, df_dh_next, hidden_size, 1,
            gradients.mid_dbh);
207 #ifdef DEBUG
208     debug_backwarding(t);
209 #endif
210 }
211
212 void debug_backwarding(int t)
213 {
214     printf("dh_dl\n");
```

```
215     fn_print_stats_vector(tmp_variables.mid_dl_dh, hidden_size * minibatch_size
            );
216     printf("dh_prev\n");
217     fn_print_stats_vector(tmp_variables.mid_dh, hidden_size * minibatch_size);
218     printf("df\n");
219     fn_print_stats_vector(tmp_variables.mid_df, hidden_size * minibatch_size);
220     printf("df_dh_next\n");
221     fn_print_stats_vector(tmp_variables.mid_df_dh_next, hidden_size *
            minibatch_size);
222     printf("dwhh\n");
223     fn_print_stats_vector(gradients.mid_dWhh, hidden_size * hidden_size);
224     printf("dWxh\n");
225     fn_print_stats_vector(gradients.mid_dWxh, hidden_size * input_size);
226     printf("dba\n");
227     fn_print_stats_vector(gradients.mid_dba, hidden_size);
228 }
229 void rnn_print_parameters()
230 {
231     printf("Wxh\n");
232     fn_print_stats_vector(rnn_parameters.mid_Wxh, hidden_size * input_size);
233     printf("Whh\n");
234     fn_print_stats_vector(rnn_parameters.mid_Whh, hidden_size * hidden_size);
235     printf("Ba\n");
236     fn_print_stats_vector(rnn_parameters.mid_bias_x, hidden_size);
237 }
238
239 void rnn_clip_weights(int min, int max)
240 {
241     fn_clip_vector(gradients.mid_dWxh, min, max, hidden_size * input_size);
242     fn_clip_vector(gradients.mid_dWhh, min, max, hidden_size * hidden_size);
243     fn_clip_vector(gradients.mid_dba, min, max, hidden_size);
244 }
245
246 void rnn_gradients_to_zero()
247 {
248     fn_zero_vec(gradients.mid_dWxh, hidden_size * input_size);
249     fn_zero_vec(gradients.mid_dWhh, hidden_size * hidden_size);
250     fn_zero_vec(gradients.mid_dba, hidden_size);
251     fn_zero_vec(gradients.mid_dbh, hidden_size);
252 }
253
254 void rnn_propagate_error_tp(int layer, int batch_size)
255 {
256 }
257
258 void rnn_initialize_tp(int layer)
259 {
260     fn_matset_random_ones(tmp_variables.mid_ones_matrix, hidden_size,
            minibatch_size, 1.0);
261     rnn_initialize_parameters(layer);
262 }
263
264 void rnn_initialize_parameters(int layer) {
265     float range = get_parameter_initialization_range(layer);
266     fn_vect_random_float(rnn_parameters.mid_Whh, (hidden_size * hidden_size), -
            range, range);
267     fn_vect_random_float(rnn_parameters.mid_Wxh, (hidden_size * input_size), -
            range, range);
268     fn_vect_random_float(rnn_parameters.mid_bias_x, hidden_size, -range, range)
            ;
269     fn_vect_random_float(rnn_parameters.mid_bias_h, hidden_size, -range, range)
            ;
270 }
271
```

```
272 float get_parameter_initialization_range(int layer) {
273     int fan_in = layers_ptr[layer - 1]->num_neurons;
274     int fan_out = layers_ptr[layer]->num_neurons;
275     float range;
276     if(rnn_parameters.activation_function == TANH) {
277         range = sqrt((1 / (fan_in + fan_out)));
278     }
279     else {
280         range = sqrt((2 / (fan_in + fan_out)));
281     }
282     return range;
283 }
284
285 void rnn_allocate_tp(int layer)
286 {
287     fn_base_layer_allocate(layer, minibatch_size);
288     allocate_tmp_buffers(layer);
289     allocate_rnn_cells(layer);
290     allocate_rnn_parameters(layer);
291     allocate_rnn_gradients(layer);
292 }
293
294 void allocate_tmp_buffers(int layer)
295 {
296     tmp_variables.multxw = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
297                                               BUFFER_TYPE_NONE);
298     tmp_variables.multhw = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
299                                               BUFFER_TYPE_NONE);
300     tmp_variables.sum_bias_x = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
301                                                   BUFFER_TYPE_NONE);
302     tmp_variables.sum_bias_h = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
303                                                   BUFFER_TYPE_NONE);
304     tmp_variables.sum_mults = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
305                                                  BUFFER_TYPE_NONE);
306
307     tmp_variables.mid_df = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
308                                               BUFFER_TYPE_NONE);
309     tmp_variables.mid_df_dh_next = fn_allocate_buffer(sizeof(type) *
        hidden_size * minibatch_size, layer,
310                                                       BUFFER_TYPE_NONE);
311     tmp_variables.mid_square_a = fn_allocate_buffer(sizeof(type) * hidden_size
        * minibatch_size, layer,
312                                                     BUFFER_TYPE_NONE);
313     tmp_variables.mid_ones_matrix = fn_allocate_buffer(sizeof(type) *
        hidden_size * minibatch_size, layer,
314                                                        BUFFER_TYPE_NONE);
315
316     tmp_variables.mid_dWxh = fn_allocate_buffer(sizeof(type) * hidden_size *
        input_size, layer,
317                                                 BUFFER_TYPE_NONE);
318     tmp_variables.mid_dWhh = fn_allocate_buffer(sizeof(type) * hidden_size *
        hidden_size, layer,
319                                                 BUFFER_TYPE_NONE);
320     tmp_variables.mid_dba = fn_allocate_buffer(sizeof(type) * hidden_size,
        layer,
321                                                BUFFER_TYPE_NONE);
322     tmp_variables.mid_dl_dh = fn_allocate_buffer(sizeof(type) * hidden_size *
        minibatch_size, layer,
```

```
323                                                 BUFFER_TYPE_NONE);
324     tmp_variables.mid_dh = fn_allocate_buffer(sizeof(type) * hidden_size *
            minibatch_size, layer,
325                                                 BUFFER_TYPE_NONE);
326     tmp_variables.mid_dh_next = fn_allocate_buffer(sizeof(type) * hidden_size *
             minibatch_size, layer,
327                                                 BUFFER_TYPE_NONE);
328
329 }
330
331
332 void allocate_rnn_parameters(int layer)
333 {
334     rnn_parameters.mid_Whh = fn_allocate_buffer((sizeof(type) * hidden_size *
            hidden_size), layer,
335                                                 BUFFER_TYPE_NONE);
336     rnn_parameters.mid_Wxh = fn_allocate_buffer((sizeof(type) * hidden_size *
            input_size), layer,
337                                                 BUFFER_TYPE_NONE);
338     rnn_parameters.mid_bias_x = fn_allocate_buffer((sizeof(type) * hidden_size)
            , layer,
339                                                 BUFFER_TYPE_NONE);
340     rnn_parameters.mid_bias_h = fn_allocate_buffer((sizeof(type) * hidden_size)
            , layer,
341                                                 BUFFER_TYPE_NONE);
342     // float xavier_sqrt = 1.0 / (sqrt(hidden_size));
343     // fn_vect_random_float(rnn_parameters.mid_Whh, (hidden_size * hidden_size)
            , -5, 5);
344     // fn_vect_random_float(rnn_parameters.mid_Wxh, (hidden_size * input_size),
             -5, 5);
345     // fn_vect_random_float(rnn_parameters.mid_bias_x, hidden_size, -5, 5);
346     // fn_vect_random_float(rnn_parameters.mid_bias_h, hidden_size, -5, 5);
347     // fn_vect_scalar_product(xavier_sqrt, rnn_parameters.mid_Whh, hidden_size
            * hidden_size);
348     // fn_vect_scalar_product(xavier_sqrt, rnn_parameters.mid_Wxh, hidden_size
            * input_size);
349     // fn_vect_scalar_product(xavier_sqrt, rnn_parameters.mid_bias_x,
            hidden_size);
350     // fn_vect_scalar_product(xavier_sqrt, rnn_parameters.mid_bias_h,
            hidden_size);
351
352 }
353
354 void allocate_rnn_gradients(int layer)
355 {
356     gradients.mid_dWxh = fn_allocate_buffer(sizeof(type) * hidden_size *
            input_size, layer,
357                                             BUFFER_TYPE_NONE);
358     gradients.mid_dWhh = fn_allocate_buffer(sizeof(type) * hidden_size *
            hidden_size, layer,
359                                             BUFFER_TYPE_NONE);
360     gradients.mid_dba = fn_allocate_buffer(sizeof(type) * hidden_size, layer,
361                                            BUFFER_TYPE_NONE);
362     gradients.mid_dbh = fn_allocate_buffer(sizeof(type) * hidden_size, layer,
363                                            BUFFER_TYPE_NONE);
364 }
365 void allocate_rnn_cells(int layer)
366 {
367     rnn_cells = malloc(sizeof(rnn_cell) * rnn_cell_count);
368     for (int t = 0; t < rnn_cell_count + 1; t++)
369     {
370         // last(extra) cell only to store the last hidden state
371         rnn_cells[t].mid_h_prev = fn_allocate_buffer(sizeof(type) * hidden_size
                * minibatch_size, layer,
```

```
                                                                BUFFER_TYPE_NONE);
        if (t < rnn_cell_count)
        {
            rnn_cells[t].mid_x = fn_allocate_buffer(sizeof(type) * input_size *
                minibatch_size, layer,
                                                                BUFFER_TYPE_NONE);
        }
    }
}

void rnn_deallocate_tp(int layer)
{
    deallocate_rnn_cells();
    deallocate_tmp_buffers(layer);
    fn_base_layer_deallocate(layer);
}

void deallocate_rnn_parameters(int layer)
{
    fn_deallocate_buffer(rnn_parameters.mid_Whh);
    fn_deallocate_buffer(rnn_parameters.mid_Wxh);
    fn_deallocate_buffer(rnn_parameters.mid_bias_x);
    fn_deallocate_buffer(rnn_parameters.mid_bias_h);
}
void deallocate_tmp_buffers(int layer)
{
    fn_deallocate_buffer(tmp_variables.multxw);
    fn_deallocate_buffer(tmp_variables.multhw);
    fn_deallocate_buffer(tmp_variables.sum_bias_x);
    fn_deallocate_buffer(tmp_variables.sum_bias_h);
    fn_deallocate_buffer(tmp_variables.sum_mults);

    fn_deallocate_buffer(tmp_variables.mid_df);
    fn_deallocate_buffer(tmp_variables.mid_df_dh_next);
    fn_deallocate_buffer(tmp_variables.mid_square_a);
    fn_deallocate_buffer(tmp_variables.mid_ones_matrix);

    fn_deallocate_buffer(tmp_variables.mid_dWxh);
    fn_deallocate_buffer(tmp_variables.mid_dWhh);
    fn_deallocate_buffer(tmp_variables.mid_dba);
    fn_deallocate_buffer(tmp_variables.mid_dbh);
    fn_deallocate_buffer(tmp_variables.mid_dh);
    fn_deallocate_buffer(tmp_variables.mid_dl_dh);
    fn_deallocate_buffer(tmp_variables.mid_dh_next);
}

void deallocate_rnn_cells()
{
    for (int t = 0; t <= rnn_cell_count; t++)
    {
        fn_deallocate_buffer(rnn_cells[t].mid_h_prev);
        if (t < rnn_cell_count)
        {
            fn_deallocate_buffer(rnn_cells[t].mid_x);
        }
    }
    free(rnn_cells);
}
```

**Código A.4:** rnn.c

```
#pragma once

void rnn_parse_tp(int layer, char *parameters);
```

```
void rnn_allocate_tp(int layer);
void rnn_deallocate_tp(int layer);
void rnn_initialize_tp(int layer);
void rnn_forward_cell(int t);
void rnn_forward(int input_index, int fbe, int lbe, int training);
void rnn_cell_backward_tp(int t);
void rnn_backward_tp(int layer, int minibatch_size);
void rnn_propagate_error_tp(int l, int b);
void allocate_rnn_cells();
void deallocate_rnn_cells();
void allocate_tmp_buffers();
void deallocate_tmp_buffers();
void get_outputs(int layer);
void rnn_initialize_parameters(int layer);
float get_parameter_initialization_range(int layer);
```

**Código A.5:** rnn.h

```
/*
 * model.c (manages a model)
 */

#include "nnsim.h"

int fn_load_model(char *filename)
{
  FILE *fd;
  int n;

  fd = fopen(filename, "r");
  if (fd == NULL)
  {
    // In inference mode we need the model
    if (running_mode == RUNNING_MODE_INFERENCE)
    {
      fprintf(stderr, "Error, model could not be opened\n");
      exit(1);
    }
    // In training mode we can omit the model load process
    fprintf(stderr, "Warning, model could not be opened, standard
        initialization will be performed instead...\n");
    return 0;
  }

  // we load all parameters for each layer
  for (int l = 1; l <= num_hidden_layers + 1; l++)
  {
    switch (layers_ptr[l]->ltype)
    {
    case LAYER_FULLY_CONNECTED:
    case LAYER_CONVOLUTIONAL:
      // we load all weights
      n = layers_ptr[l]->num_weights;
      type *v = malloc(sizeof(type) * n);
      int reads = fread(v, sizeof(type), n, fd);
      if (reads != n)
      {
        printf("Error reading model\n");
        exit(1);
      }
      int size = n * sizeof(type);
      fn_write_buffer(layers_ptr[l]->mid_weights, v, size);
      free(v);
      // we now load all bias
```

```
46        n = layers_ptr[l]->num_bias;
47        v = malloc(sizeof(type) * n);
48        reads = fread(v, sizeof(type), n, fd);
49        if (reads != n)
50        {
51          printf("Error reading model\n");
52          exit(1);
53        }
54        size = n * sizeof(type);
55        fn_write_buffer(layers_ptr[l]->mid_bias, v, size);
56        free(v);
57        break;
58      case LAYER_PERMUTATION:
59      case LAYER_MAXPOOLING:
60      case LAYER_REARRANGE_CONVOLUTIONAL:
61      case LAYER_DROPOUT:
62      case LAYER_PADDING:
63      case LAYER_RELU:
64      case LAYER_SOFTMAX:
65        // no parameters
66        break;
67      case LAYER_PERFECT_SHUFFLE:
68      case LAYER_RNN:
69        load_rnn_parameters(fd);
70        break;
71      case LAYER_GROUP_CONVOLUTIONAL:
72        printf("TODO: layer not loaded!!!\n");
73        break;
74
75      case LAYER_BATCH_NORMALIZATION:
76        // we load gamma
77        n = layers_ptr[l]->batchnorm_num_vectors;
78        v = malloc(sizeof(type) * n);
79        reads = fread(v, sizeof(type), n, fd);
80        if (reads != n)
81        {
82          printf("Error reading model\n");
83          exit(1);
84        }
85        size = n * sizeof(type);
86        fn_write_buffer(layers_ptr[l]->mid_gamma, v, size);
87        free(v);
88        // we load beta
89        n = layers_ptr[l]->batchnorm_num_vectors;
90        v = malloc(sizeof(type) * n);
91        reads = fread(v, sizeof(type), n, fd);
92        if (reads != n)
93        {
94          printf("Error reading model\n");
95          exit(1);
96        }
97        size = n * sizeof(type);
98        fn_write_buffer(layers_ptr[l]->mid_beta, v, size);
99        free(v);
100       break;
101
102     default:
103       printf("Error, layer not known when loading model\n");
104       break;
105     }
106   }
107   fclose(fd);
108   return 1;
109 }
```

```
110
111 // saves a model
112 int fn_save_model(char *filename)
113 {
114   FILE *fd;
115   int n;
116
117   fd = fopen(filename, "w");
118   if (fd == NULL)
119   {
120     printf("Error, file could not be created\n");
121     printf("model will not be saved\n");
122     return 0;
123   }
124
125   // we save all parameters for each layer
126   printf("layers to save: %d\n", num_hidden_layers);
127   for (int l = 1; l <= num_hidden_layers + 1; l++)
128   {
129     switch (layers_ptr[l]->ltype)
130     {
131     case LAYER_FULLY_CONNECTED:
132     case LAYER_CONVOLUTIONAL:
133       // we save weights
134       n = layers_ptr[l]->num_weights;
135       int size = n * sizeof(type);
136       type *v = malloc(size);
137       fn_read_buffer(layers_ptr[l]->mid_weights, v, size);
138       int writes = fwrite(v, sizeof(type), n, fd);
139       if (writes != n)
140       {
141         printf("Error writing model\n");
142         exit(1);
143       }
144       free(v);
145       // we save bias
146       n = layers_ptr[l]->num_bias;
147       size = n * sizeof(type);
148       v = malloc(size);
149       fn_read_buffer(layers_ptr[l]->mid_bias, v, size);
150       writes = fwrite(v, sizeof(type), n, fd);
151       if (writes != n)
152       {
153         printf("Error writing model\n");
154         exit(1);
155       }
156       free(v);
157       break;
158     case LAYER_PERMUTATION:
159     case LAYER_MAXPOOLING:
160     case LAYER_REARRANGE_CONVOLUTIONAL:
161     case LAYER_DROPOUT:
162     case LAYER_PADDING:
163     case LAYER_RELU:
164     case LAYER_SOFTMAX:
165       // no parameters
166       break;
167     case LAYER_PERFECT_SHUFFLE:
168     case LAYER_RNN:
169       save_rnn_parameters(fd);
170     case LAYER_GROUP_CONVOLUTIONAL:
171       printf("TODO: layer not saved!!!\n");
172       break;
173
```

```
174     case LAYER_BATCH_NORMALIZATION:
175       // we save gamma
176       n = layers_ptr[l]->batchnorm_num_vectors;
177       size = n * sizeof(type);
178       v = malloc(size);
179       fn_read_buffer(layers_ptr[l]->mid_gamma, v, size);
180       writes = fwrite(v, sizeof(type), n, fd);
181       if (writes != n)
182       {
183         printf("Error writing model\n");
184         exit(1);
185       }
186       free(v);
187       // we save beta
188       n = layers_ptr[l]->batchnorm_num_vectors;
189       size = n * sizeof(type);
190       v = malloc(size);
191       fn_read_buffer(layers_ptr[l]->mid_beta, v, size);
192       writes = fwrite(v, sizeof(type), n, fd);
193       if (writes != n)
194       {
195         printf("Error writing model\n");
196         exit(1);
197       }
198       free(v);
199       break;
200
201     default:
202       printf("Error, layer not known when saving model\n");
203       break;
204     }
205   }
206
207   fclose(fd);
208   return 1;
209 }
210
211 void load_rnn_parameters(FILE *fd)
212 {
213   try_fread(rnn_parameters.mid_bias_x, hidden_size, fd, "RNN_LAYER", "Couldn't
        load bias_x");
214   try_fread(rnn_parameters.mid_bias_h, hidden_size, fd, "RNN_LAYER", "Couldn't
        load bias_h");
215   try_fread(rnn_parameters.mid_Wxh, hidden_size * input_size, fd, "RNN_LAYER",
        "Couldn't load Wxh");
216   try_fread(rnn_parameters.mid_Whh, hidden_size * hidden_size, fd, "RNN_LAYER",
         "Couldn't load Whh");
217 }
218 void try_fread(int mid, int size, FILE *fd, const char *layer_name, const char
     *error_message)
219 {
220   size_t stype = sizeof(type);
221   type *v = malloc(stype * size);
222   int reads = fread(v, stype, size, fd);
223   if (reads != size)
224     raise_load_error(layer_name, error_message, reads);
225   fn_write_buffer(mid, v, size * stype);
226   free(v);
227 }
228 void save_rnn_parameters(FILE *fd)
229 {
230   try_fwrite(rnn_parameters.mid_bias_x, hidden_size, fd, "RNN_LAYER", "Couldn't
        write bias_x");
```

```
231   try_fwrite(rnn_parameters.mid_bias_h, hidden_size, fd, "RNN_LAYER", "Couldn't
          write bias_h");
232   try_fwrite(rnn_parameters.mid_Wxh, hidden_size * input_size, fd, "RNN_LAYER",
          "Couldn't write Wxh");
233   try_fwrite(rnn_parameters.mid_Whh, hidden_size * hidden_size, fd, "RNN_LAYER"
        , "Couldn't write Whh");
234 }
235
236 void try_fwrite(int mid, int size, FILE *fd, const char *layer_name, const char
      *error_message)
237 {
238   type *v;
239   size_t stype = sizeof(type);
240   v = malloc(size * stype);
241   fn_read_buffer(mid, v, size * stype);
242   int writes = fwrite(v, stype, size, fd);
243   free(v);
244   if (writes != size)
245     raise_save_error(layer_name, error_message, writes);
246 }
247 void raise_save_error(char *layer_name, char *error_message, int writes)
248 {
249   printf("Error saving model in %s layer. (model.c).\n%s\nNum of writes: %d\n",
          layer_name, error_message, writes);
250   exit(1);
251 }
252 void raise_load_error(char *layer_name, char *error_message, int reads)
253 {
254   printf("Error loading model in %s layer. (model.c).\n%s\nNum of reads: %d\n",
          layer_name, error_message, reads);
255   exit(1);
256 }
```

**Código A.6:** model.c

```
1  import torch
2  from torch import nn
3  from torch.nn import init
4  import torchvision
5  import torchvision.transforms as transforms
6  import sys
7  import time
8  from torch.autograd import Variable
9  import time
10
11 device = torch.device('cpu')
12
13 batch_size=32
14 input_size=28
15 time_step=28
16 hidden_size=64
17 num_layers=1
18 many_to_many = True
19 torch.set_num_threads(3)
20
21
22 trainset = torchvision.datasets.MNIST(root=".",train=True,transform=transforms.
       ToTensor(),
23         download=True)
24 testset = torchvision.datasets.MNIST(root=".",train=False,transform=transforms.
       ToTensor())
25
26 trainloader = torch.utils.data.DataLoader(dataset=trainset,batch_size=
       batch_size,shuffle=True)
```

```
testloader = torch.utils.data.DataLoader(dataset=testset,batch_size=batch_size,
    shuffle=False)


lr = 0.01
num_epochs = 3

class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size,
            num_layers=1, batch_first=True)
        if many_to_many:
            self.Linear=nn.Linear(hidden_size * 28, 10)
        else:
            self.Linear=nn.Linear(hidden_size, 10)
        self.relu = nn.ReLU()
        self.soft = nn.Softmax(dim=1)

    def forward(self, x):
        r_out, h_out = self.rnn(x, None)
        last = r_out[:,-1,:]
        if many_to_many:
            r_out = r_out.reshape(r_out.shape[0], r_out.shape[1] * r_out.shape
                [2])
        else:
            r_out = last
        output = self.Linear(r_out)#out:(batch, time_step, hidden_size) select
            the last output (category)
        output = self.relu(output)
        output = self.soft(output)
        return output


model = RNNModel(input_size, hidden_size, num_layers).to(device)

optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0)

loss = nn.CrossEntropyLoss()#Not one-hot form


for epoch in range(num_epochs):
    start = time.time()
    correct = 0
    total = 0
    for i, (images, labels) in enumerate(trainloader):
        images = Variable(images.view(-1, time_step, input_size))
        labels = Variable(labels)

        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted.cpu()==labels).sum()
        l = loss(outputs, labels)
        optimizer.zero_grad()
        l.backward()
        optimizer.step()
    print('train acc: %.4f' %(100.0 * correct / total) )


    print('epoch %d, loss %.4f, time %.1f sec'
            %(epoch+1, l.data.cpu().numpy(), time.time()-start))
```

```
87      correct = 0
88      total = 0
89      for images, labels in testloader:
90          images = Variable(images.view(-1, time_step, input_size))
91          outputs = model(images)
92          _, predicted = torch.max(outputs.data, 1)
93          total += labels.size(0)
94          correct += (predicted.cpu()==labels).sum()
95
96
97      if epoch == num_epochs - 1:
98          print('test acc: %.4f' %(100.0 * correct / total) )
```

**Código A.7:** Modelo MNIST en PyTorch